Magistrate: A Repository-Context Aware Multi-Agent Framework for Automated Code Review

Anonymous ACL submission

Abstract

As Large Language Models make code writing faster than ever, the bottleneck has shifted to the critical task of reviewing code before merge: ensuring correctness, security, and compliance with project guidelines. We introduce **Magistrate**, an automated code review framework designed to augment human reviewers by detecting complex, cross-file logic errors that single-pass analysis misses. Unlike existing tools that analyze isolated diffs, Magistrate employs a hierarchical multi-agent architecture: a *Delegator* partitions changed files into semantically coherent batches, while parallel *IssueDetector* agents combine static analysis (ast-grep) with LLM-based semantic reasoning over the full repository context. We also present **Magistrate-Bench**, a benchmark of 2,042 Pull Requests across 12 programming languages, with complete repository workspaces cached at evaluation time to enable realistic dependency tracing. Evaluated on a 108-PR subset across four frontier models (Gemini 3 Flash, Devstral, Minimax M2, and Grok 4.1 Fast), Magistrate consistently improves F1 scores by 2.2–5.5$\times$ over single-shot baselines, with the best model (Gemini 3 Flash) achieving an F1 of 0.214 and a hallucination rate of just 0.6%. Across all models, Magistrate identified **997 valid issues** that human reviewers had overlooked, demonstrating its utility as a complementary filter for objective correctness issues.

1 Introduction

Automated code review remains a formidable challenge for AI systems despite significant advances in code understanding. While Large Language Models (LLMs) have achieved state-of-the-art results in code *generation* (Jimenez et al., 2024), their application to *review* has been constrained by insufficient access to systemic context. Current benchmarks and tools often treat a Pull Request (PR) as an isolated text-processing task (feeding a model a diff snippet and expecting a comprehensive critique). This "Diff-Only Myopia" (Zeng et al., 2025) prevents agents from reasoning about global implications of local changes, such as breaking downstream dependencies or violating project-wide invariants.

Recent benchmarks have begun addressing this limitation. SWR-Bench (Zeng et al., 2025) and CodeFuse-CR-Bench (Guo et al., 2025) establish that repository-level context is essential for realistic evaluation. ContextCRBench (Hu et al., 2025) further demonstrates the importance of enriched metadata such as issue descriptions and PR context. However, while these benchmarks provide *data* for repository-level review, they do not prescribe *architectural solutions* for agents to effectively consume massive context without suffering from information overload or context window limitations.

We present **Magistrate**, a multi-agent framework designed to orchestrate hybrid static and semantic analysis over the complete repository state. Unlike single-model approaches or simple tool loops, Magistrate employs a structured three-phase architecture: **(1) Delegation**, **(2) Detection**, and **(3) Aggregation**. A `Delegator` agent partitions changed files into semantically coherent batches based on module boundaries and test-implementation coupling. Specialized `IssueDetector` agents then analyze each batch in parallel, combining static pre-scanning via ast-grep with LLM-based reasoning augmented by code exploration tools (`ripgrep`, `git log`, `cat`). A final aggregation phase deduplicates findings and produces a structured review report.

Critically, we position Magistrate not as a replacement for human reviewers, but as a **complementary filter**. Our analysis reveals that human reviewers predominantly focus on subjective style and design decisions, often overlooking subtle logic errors. By contrast, Magistrate is architected to prioritize objective correctness. This distinct

focus explains the system's divergent performance characteristics: while it may not reproduce every human stylistic comment (lower Recall on ground truth), it excels at surfacing critical logic bugs that humans miss.

Our core contributions are:

1. **Magistrate Architecture:** A multi-agent framework that orchestrates delegation and hybrid static/semantic analysis to detect complex cross-file issues.

2. **Magistrate-Bench:** A dataset of 2,042 PRs across 12 languages, featuring full repository workspaces cached at evaluation time to enable realistic dependency tracing.

3. **Evaluation Protocol:** A rigorous "Valid Extras" methodology that uses workspace-backed verification to distinguish between hallucinations and valid issues overlooked by human reviewers.

2 Related Work

We situate our contributions within code review automation, benchmarks, and multi-agent systems.

2.1 Code Review Automation

Code review is fundamentally a knowledge task requiring broad contextual awareness. Recent surveys (Tufano et al., 2024; Heumüller and Ortmeier, 2025) document the limitations of LLM-based approaches in handling complex, context-dependent scenarios. A key emerging direction is integrating static analysis: Jaoua et al. (Jaoua et al., 2025) and BitsAI-CR (Sun et al., 2025) demonstrate that grounding LLMs in static findings reduces hallucinations. Magistrate extends this by combining deterministic static analysis (ast-grep) with active workspace exploration, prioritizing correctness over convention.

2.2 Benchmarks for Code Review

Benchmark design has evolved from function-level to repository-level evaluation. SWR-Bench (Zeng et al., 2025) and CodeFuse-CR-Bench (Guo et al., 2025) established that diff-only evaluation is insufficient. ContextCRBench (Hu et al., 2025) demonstrated the value of enriched textual context. Table1 summarizes the landscape. Magistrate-Bench extends these by caching complete repository workspaces at evaluation time, enabling realistic dependency tracing across 12 languages.

2.3 Multi-Agent Systems

Multi-agent architectures like RevAgent (Li et al., 2025) and CodeAgent (Ma et al., 2024) assign specialized roles (Reviewer, Author) to agents. However, conversational systems can suffer from "prompt drifting." Agentless (Xia et al., 2024) showed that hierarchical localization can outperform complex agents on repair tasks. Magistrate adopts this hierarchical philosophy (Delegation → Detection) but retains the tool-use capabilities of SWE-agent (Yang et al., 2024) for active verification, avoiding drift through "Stateless Batching."

2.4 Evaluation Metrics

Traditional metrics like BLEU correlate poorly with review quality. We adopt the philosophy of CRScore (Naik et al., 2025) and DeepCRCEval (Lu et al., 2025) by using a "Valid Extras" methodology that validates discoveries against the workspace, distinguishing real bugs from hallucinations.

3 Magistrate Framework

Magistrate is a hierarchical multi-agent system designed around a central `MagistrateOrchestrator` that coordinates the review process. When a pull request is submitted for review, the orchestrator first dispatches the task to a `Delegator` agent, which analyzes the changed files and creates an intelligent execution plan. This plan is then executed by parallel `IssueDetector` agents, each analyzing a batch of related files. Finally, the orchestrator aggregates all findings into a unified review report (Figure 1).

3.1 Delegation

The `Delegator` agent performs two critical functions. First, it analyzes the PR metadata to generate a high-level `PRSummary`, establishing the intent of the changes. Second, it generates a `DelegationPlan` that groups files into semantically coherent batches using heuristics such as module cohesion and test-implementation pairing.

3.2 Analysis

In the analysis stage, each `DelegationBatch` is processed by a dedicated `IssueDetector` agent instance running in parallel. Each detector operates in a hybrid mode combining static and semantic analysis:

Static Pre-Scan. The agent first executes `ast-grep` with a suite of security and quality rules.

Feature	Magistrate-Bench	SWR-Bench	CodeFuse-CR	ContextCRBench	SWE-bench
Task Unit	Pull Request	Pull Request	Pull Request	Code Chunk	GitHub Issue
Scale	2,042 PRs	1,000 PRs	601 PRs	67,910 Chunks	2,294 Issues
Languages	12	Python	Python	9	Python
Code Context	Full Repository	Full Repository	Full Repository	Diff Only	Full Repository
Verification	Tool-Verified	Text-Match	Model Score	Text-Match	Test-Execution

Table 1: Comparison of code review benchmarks. Magistrate-Bench distinguishes itself by combining **Full Repository** context (unlike ContextCRBench) with active **Tool-Verified** evaluation (unlike SWR-Bench).

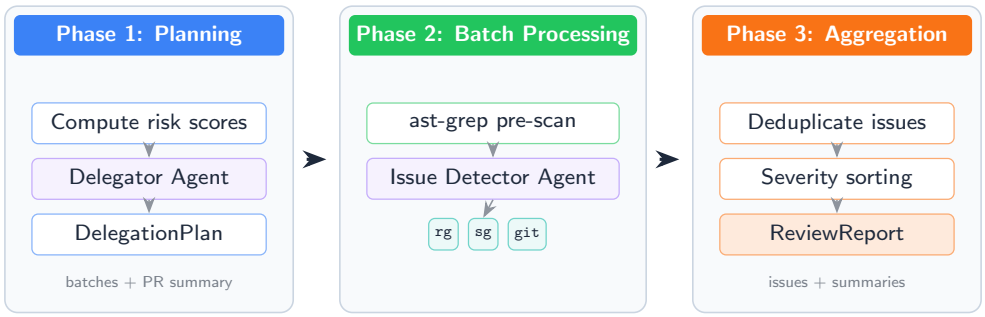

Figure 1: Magistrate execution flow. Phase 0 prepares the workspace from cached repo tarballs; Phase 1 Delegator pulls README/CONTRIBUTING and commit context, batches related files, and emits the PR summary. Phase 2 runs parallel IssueDetector instances with ast-grep prescan, tool-backed evidence gathering (`batch_execute_tool`), and post-LLM verification. Phase 3 deduplicates by file/line/title, orders by severity, and writes `review.json`.

These deterministic findings are injected into the LLM's context as hints.

Semantic Analysis. The LLM validates the pre-scan findings while searching for deeper logic errors that static analysis cannot catch. It utilizes a guarded tool palette (detailed in Appendix C) to explore the codebase.

Verification. A programmatic post-processing layer validates that every reported issue cites a file path and line number that physically exists in the PR, filtering out hallucinated references.

3.3 Aggregation

Upon the completion of all detector agents, the `MagistrateOrchestrator` aggregates the validated issues from every batch, performing a deduplication pass to merge identical findings before producing the final `ReviewReport`.

3.4 End-to-End Execution Workflow

The complete review pipeline proceeds through the following stages:

1. **Task Submission**: A review request is queued, triggering a dedicated worker container for isolated execution.

2. **Workspace Preparation**: The worker restores a cached repository workspace corresponding to the exact base and head commits, ensuring byte-for-byte reproducibility.

3. **Phase 0 (Static Prescan)**: The IssueDetector executes a static rule-based prescan using `ast-grep`, producing grounded evidence injected into subsequent prompts.

4. **Phase 1 (Delegation)**: The Delegator produces a `PRSummary` and `DelegationPlan`, grouping files into semantically coherent batches to mitigate context-window overflow.

5. **Phase 2 (Parallel Batch Analysis)**: Each batch is processed by an IssueDetector instance. Detectors operate in parallel to improve throughput.

6. **Phase 2b (Verification and Filtering)**: All issues generated by detectors undergo a structured post-processing stage. Invalid line numbers and nonexistent file paths are removed.

7. **Phase 3 (Aggregation)**: The orchestrator deduplicates issues, normalizes severity labels, and emits the final `review.json` artifact.

8. **Return to Server**: The worker uploads results and terminates. No state persists, guaranteeing isolation.

This workflow enforces strict grounding, keeps LLM context under control, and ensures consistent behavior through typed I/O schemas.

4 Magistrate-Bench Dataset

To rigorously test repository-aware agents, we constructed **Magistrate-Bench**, a new benchmark designed to overcome the contextual myopia of existing datasets.

4.1 Construction Pipeline

Our collection process ensures high-quality, reproducible ground truth through a rigorous multi-stage pipeline (Figure 2).

Repository & PR Selection: We targeted active repositories (> 500 stars, > 1500 PRs, active within 150 days) across 12 languages, explicitly excluding tutorials and algorithm lists. We selected only merged PRs with a minimum of 6 distinct human review threads.

Temporal State Reconstruction: Review comments typically address the state *before* fixes are pushed. We algorithmically identify the last commit hash immediately preceding the first human review timestamp. We then fetch the cumulative diff from the PR base to this specific commit, ensuring the agent evaluates the exact state seen by the human reviewer.

Hybrid Extraction & Verification: We employ a hybrid strategy to minimize noise. We extract exact file paths, lines, and comment bodies via the GitHub API. An LLM agent (Temperature=0) acts as a classifier, categorizing threads into issue types (e.g., Logic, Security). Finally, a programmatic verification step cross-references every extracted issue against the actual diff, discarding findings that reference unchanged files.

Diversity & Standardization: We enforced a strict cap of 10 PRs per repository to ensure generalization. Extracted issues are mapped to a standardized taxonomy to facilitate direct comparison with existing benchmarks. See Appendix D for detailed distribution statistics.

4.2 Dataset Statistics

Magistrate-Bench exceeds the scale and diversity of prior benchmarks (see Table 5 in Appendix). It contains 2,042 PRs covering 12 languages, with an average of 5.94 issues per PR.

5 Experimental Design

Evaluating automated code review presents a fundamental challenge: agent-reported issues not present in ground truth may represent either hallucinations *or* valid issues that human reviewers overlooked. Traditional precision metrics conflate these cases, penalizing systems for finding real bugs. We address this through a rigorous verification pipeline that distinguishes *Hallucinations* from *Valid Extras*.

5.1 Research Questions

We structure our evaluation around the following:

- **RQ1 (Effectiveness):** Does repository-aware orchestration improve the detection of logic errors (Recall/F1) compared to single-shot baselines?
- **RQ2 (Reliability):** Does the workspace-backed verification layer significantly reduce hallucination rates?
- **RQ3 (Efficiency):** What is the trade-off between detection quality and computational cost?

5.2 Experimental Setup

We evaluate on a stratified subset of 108 PRs from Magistrate-Bench (Python, JS/TS, Java, Rust). For each PR, we compare (i) a single-shot **Raw** baseline (prompting the model with the diff) against (ii) the orchestrated **Magistrate** system, using the same underlying LLM backbone.

5.3 Verification Methodology

To ensure the soundness of our "Valid Extras" (issues found by agents but missed by humans), we employ a multi-stage verification process designed to minimize "LLM-as-a-judge" bias:

1. Structural Filtering: We programmatically verify that every reported file path and line number exists in the PR diff. This step rigorously filters out hallucinations referencing non-existent files or invalid line coordinates, ensuring that all candidate issues are spatially grounded before semantic verification.

2. Claim-Blind Evidence Gathering: To verify semantic claims (e.g., "This function call is missing a required argument"), we use a strictly **Claim-Blind** protocol to eliminate confirmation bias. An "Evidence Agent" is given a neutral query derived from the issue (e.g., "What are the arguments for function X?") *without access to the agent's claim*

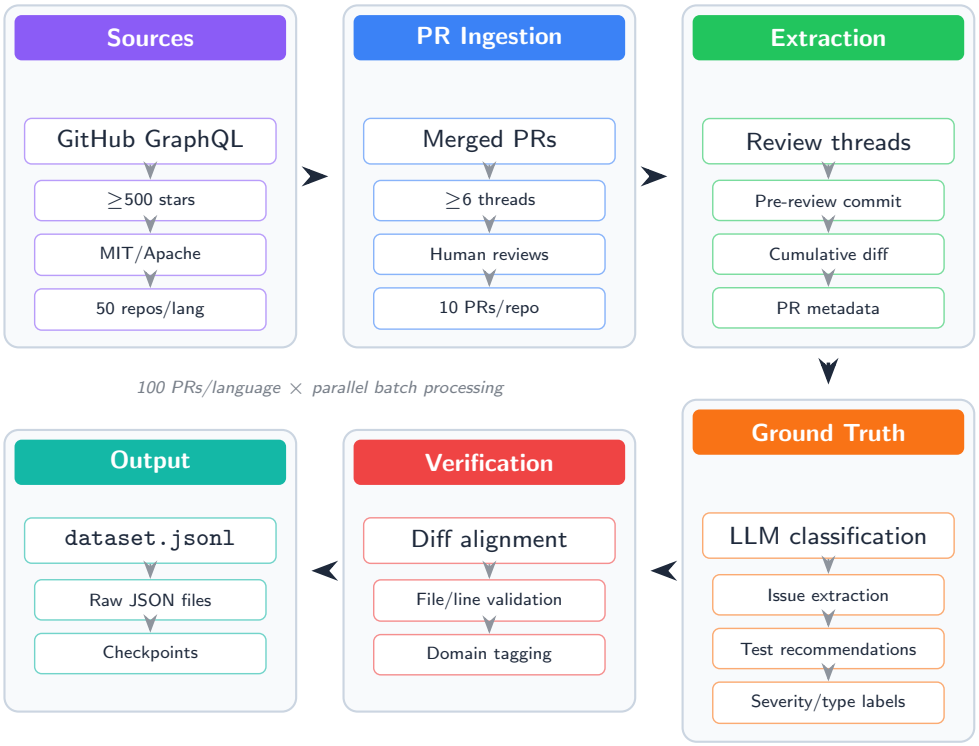

Figure 2: Magistrate-Bench construction. Filters select merged PRs with human reviews across diverse languages. We scrape metadata/diffs/threads and clone repos; build unified diffs and cache full-repo workspaces for evaluation; parse review threads into structured issues/tests with an LLM parser (T=0); remove bots/LGTM noise; publish JSONL/CSV.

or verdict. It must use repository tools (`grep`, `cat`) to retrieve the ground truth code state. This prevents the verifier from simply agreeing with the detector's plausible-sounding rhetoric.

3. Objective Verdict: A separate Verifier compares the retrieved ground-truth evidence against the agent's original claim. This separation ensures the system relies on codebase reality, not rhetorical plausibility.

5.4 Enhanced Metrics

We report verification-aware metrics:

- TP**:** Matches ground truth.
- V **(Valid Extras):** Unmatched but verified as correct.
- H **(Hallucinations):** Unmatched and verified as incorrect.

Enhanced Precision: $P = \frac{TP+V}{TP+V+H}$

Recall: $R = \frac{TP}{|GT|}$, where $|GT|$ is the total number of issues in the ground truth.

F1 Score: $F1 = 2 \cdot \frac{P \cdot R}{P+R}$

Hallucination Rate: $HR = \frac{H}{TP+V+H}$

These metrics reward agents for genuine discoveries (V) while penalizing incorrect hallucinations (H), providing a fair assessment of real-world utility.

6 Results

We report verified results on the 108-PR evaluation subset described in Section 5, comparing single-shot **Raw** baselines against the orchestrated **Magistrate** system across four frontier models.

6.1 Effectiveness (RQ1)

Table 2 summarizes performance. Magistrate improves F1 by 2.2–5.5× and increases verified *Valid Extras*. Gemini 3 Flash achieves the strongest results, while Devstral shows the largest relative improvement over its baseline.

It is important to contextualize the absolute Recall scores. Human-annotated ground truth is heavily skewed toward subjective style and design preferences (see Table 3), which Magistrate is often prompted to ignore in favor of objective correctness. Consequently, matching ground truth requires not just finding bugs, but mimicking human stylistic preferences. The "Valid Extras" metric is therefore a more accurate proxy for the system's utility in

Model	Agent	Enh. Prec.	Recall	F1	Valid Extras	Hallucinations	Hall. Rate
Gemini 3 Flash	Raw	0.804	0.052	0.097	94	32	19.6%
	Magistrate	**0.994**	**0.120**	**0.214**	**432**	3	**0.6%**
Devstral	Raw	0.600	0.007	0.014	13	12	40.0%
	Magistrate	**0.903**	**0.040**	**0.077**	**194**	24	**9.7%**
Minimax M2	Raw	0.737	0.038	0.072	46	26	26.3%
	Magistrate	**0.820**	**0.099**	**0.177**	**197**	59	**18.0%**
Grok 4.1 Fast	Raw	0.625	0.013	0.026	10	12	37.5%
	Magistrate	**0.787**	**0.038**	**0.073**	**174**	55	**21.3%**

Table 2: Overall performance comparison across models. Magistrate consistently improves over Raw baseline across all models, with F1 improvements of 2.2–5.5× and substantial reductions in hallucination rate.

detecting objective defects. Magistrate identifies **997** verified issues absent from human review, indicating strong complementary coverage.

6.2 Reliability (RQ2)

Magistrate consistently reduces hallucination rates (Table 2). Gemini 3 Flash drops from 19.6% to 0.6%, with other models showing similar trends. This validates the effectiveness of the workspace-backed verification layer.

6.3 Qualitative Alignment

Table 3 reveals a clear divergence: human reviewers focus on **Style/Design** (500 issues), while Magistrate excels at **Bug/Logic** (200 Valid Extras for Gemini). This confirms our hypothesis that AI agents function best as **complementary filters** for objective correctness, allowing humans to focus on high-level architectural decisions and subjective style.

Category	GT	Gemini VE	Devstral VE	Minimax VE
Style/Design	500	103	41	66
Bug/Logic	121	**200**	**99**	**87**
Security	7	8	3	3
Docs	85	50	14	20
Testing	0	22	32	15

Table 3: Condensed category analysis (Valid Extras).

6.4 Cost-Efficiency (RQ3)

Magistrate consumes significantly more tokens (10–60×) than single-shot baselines. However, Table 4 shows that when measured per *valid issue found*, the trade-off is favorable. Grok 4.1 Fast achieves near-parity cost efficiency while finding 10× more issues.

While the absolute token cost ($0.50–$1.00 per PR) is higher than a simple API call, it must be weighed against the cost of a software defect.

Model	Token OH	Valid OH	Efficiency
Gemini 3 Flash	62.6×	4.0×	15.8×
Devstral	53.2×	12.4×	4.3×
Minimax M2	42.1×	3.7×	11.5×
Grok 4.1 Fast	10.8×	11.2×	1.0×

Table 4: Token efficiency analysis. Token OH = total token overhead (Magistrate/Raw), Valid OH = valid issues ratio (Magistrate/Raw), Efficiency = tokens per valid issue ratio (Magistrate/Raw). Despite higher token usage, Magistrate achieves better issue detection density, with efficiency ratios ranging from 2.45× (Devstral on small PRs) to 22.9× (Gemini on small PRs).

Given that the industry average cost to fix a bug post-release is orders of magnitude higher than finding it during review, paying a marginal token cost to uncover critical logic errors (which humans missed) represents a high-ROI investment.

7 Conclusion and Future Work

We presented Magistrate, a repository-aware automated code review framework comprising: (1) a hierarchical multi-agent architecture that orchestrates delegation, detection, and aggregation; (2) Magistrate-Bench, a benchmark of 2,042 PRs across 12 languages with cached repository workspaces; and (3) an evaluation protocol that distinguishes hallucinations from valid discoveries through workspace-backed verification.

Our experiments on a 108-PR subset across four frontier models (Gemini 3 Flash, Devstral, Minimax M2, and Grok 4.1 Fast) demonstrate that orchestrated multi-agent review consistently improves F1 scores by 2.2–5.5× over single-shot baselines. The best-performing model (Gemini 3 Flash) achieves an F1 of 0.214 with an exceptionally low hallucination rate of 0.6%. Critically, across all models, Magistrate identified **997** verified issues that are absent from the human-review

ground truth, with individual models contributing 432 (Gemini), 194 (Devstral), 197 (Minimax), and 174 (Grok) Valid Extras respectively. This validates Magistrate's utility as a complementary filter that excels at objective correctness issues rather than stylistic concerns.

The consistency of improvements across diverse model providers (Google, Mistral, Minimax, xAI) suggests that the Magistrate architecture provides robust value independent of the specific LLM backbone, making it a practical choice for production deployment.

Future Work. Three directions merit investigation: (1) *Dynamic Analysis Integration* (verifying suspected bugs through test execution or sandboxed evaluation to reduce false positives); (2) *Personalization* (adapting review agents to repository-specific conventions and maintainer preferences); and (3) *Heterogeneous Model Orchestration* (using high-capability models for delegation while routing granular detection to cost-effective alternatives).

8 Limitations

We identify several limitations in our current work concerning internal consistency and evaluation metrics.

LLM Non-determinism. LLMs exhibit inherent stochasticity. While we employ temperature=0 for the Delegator and low temperature (0.2) for analysis to reduce variance, complete elimination of stochasticity is not possible. Future work should include results from multiple evaluation runs with confidence intervals to establish statistical significance rigorously.

Implementation Fidelity. Bugs in the orchestration logic or tool wrappers could skew results. We mitigate this through extensive unit testing of the `batch_execute_tool` and manual verification of agent traces to ensure tools are being invoked and parsed correctly, but rare edge cases may persist.

Ground Truth Quality. Human code reviews are not a perfect oracle; reviewers miss bugs. We address this via the "Valid Extras" protocol, but there remains a risk that our workspace-backed verifier (LLM with tool access) itself makes mistakes. We audit a sample of verifier decisions to estimate this error rate, but scaling this audit is labor-intensive.

Metric Validity. Standard precision/recall metrics penalize agents for finding valid issues that humans missed. Our modified metrics attempt to correct this, but rely on the accuracy of the verification step.

9 Ethics Statement

Our work introduces **Magistrate**, an automated code review framework. We adhere to the ACL Ethics Policy and consider the broader impact of this technology.

Human-in-the-Loop. Magistrate is designed to augment, not replace, human reviewers. Automated systems can hallucinate or miss subtle context (as noted in our Limitations). We explicitly position this tool as a "filter" for objective correctness, relying on human judgment for architectural and stylistic decisions. Over-reliance on automation without human oversight could lead to a degradation in code quality or the acceptance of subtle security vulnerabilities.

Bias and Fairness. The LLMs used in this study (Gemini, Devstral, etc.) are trained on public code repositories, which may contain biased language or non-inclusive terminology. Furthermore, the models may perform unevenly across different programming languages, potentially disadvantaging developers working in less-represented languages (though our benchmark covers 12 languages to mitigate this).

Energy Consumption. Multi-agent systems with iterative reasoning (like Magistrate) consume significantly more energy than single-shot inference. We analyze this trade-off in our results, highlighting that while computational cost is higher, the density of valid issues found justifies the resource usage for critical codebases. Future work will focus on optimizing token efficiency.

Potential for Misuse. While designed for defense (code review), the vulnerability detection capabilities could theoretically be adapted for malicious code auditing. However, we believe the defensive benefits of securing open-source software outweigh this risk, and the system requires a full repository context to function effectively.

References

coderabbitai. 2024. ast-grep-essentials: Community-led collection of essential ast-grep rules. https://github.com/coderabbitai/ast-grep-essentials. Accessed: 2025.

Hanyang Guo, Xunjin Zheng, Zihan Liao, Hang Yu, Peng Di, Ziyin Zhang, and Hong-Ning Dai. 2025. CodeFuse-CR-Bench: A comprehensiveness-aware benchmark for end-to-end code review evaluation in Python projects. *arXiv preprint arXiv:2509.14856*.

Robert Heumüller and Frank Ortmeier. 2025. Previously on... automating code review. *arXiv preprint arXiv:2508.18003*.

Ruida Hu, Xinchen Wang, Xin-Cheng Wen, Zhao Zhang, Bo Jiang, Pengfei Gao, Chao Peng, and Cuiyun Gao. 2025. Benchmarking LLMs for fine-grained code review with enriched context in practice. *arXiv preprint arXiv:2511.07017*.

Imen Jaoua, Oussama Ben Sghaier, and Houari Sahraoui. 2025. Combining large language models with static analyzers for code review generation. *arXiv preprint arXiv:2502.06633*.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can language models resolve real-world GitHub issues? *arXiv preprint arXiv:2310.06770*.

Shuochuan Li, Dong Wang, Patanamon Thongtanunam, Zan Wang, Jiuqiao Yu, and Junjie Chen. 2025. Issue-oriented agent-based framework for automated review comment generation. *arXiv preprint arXiv:2511.00517*.

Junyi Lu, Xiaojia Li, Zihan Hua, Lei Yu, Shiqi Cheng, Li Yang, Fengjun Zhang, and Chun Zuo. 2025. DeepCRCEval: Revisiting the evaluation of code review comment generation. *arXiv preprint arXiv:2412.18291*.

Ke Ma, Zhaochen Meng, Weicheng Wang, Yu Wang, Ying Zhang, Min Zhang, and Zhiyuan Liu. 2024. Codeagent: Enhancing code generation with tool-integrated agent systems. *arXiv preprint arXiv:2402.02172*.

Atharva Naik, Marcus Alenius, Daniel Fried, and Carolyn Rose. 2025. CRScore: Grounding automated evaluation of code review comments in code claims and smells. *arXiv preprint arXiv:2409.19801*.

Tianyang Sun and 1 others. 2025. BitsAI-CR: Automated code review via LLM in practice. *arXiv preprint arXiv:2501.15134*.

Rosalia Tufano, Ozren Dabić, Antonio Mastropaolo, Matteo Ciniselli, and Gabriele Bavota. 2024. Code review automation: Strengths and weaknesses of the state of the art. *arXiv preprint arXiv:2401.05136*.

Chunqiu Steven Xia, Yinlin Deng, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lunt, Shunyu Yao, and Karthik Narasimhan. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*.

Zhengran Zeng, Ruikai Shi, Keke Han, Yixin Li, Kaicheng Sun, Yidong Wang, Zhuohao Yu, Rui Xie, Wei Ye, and Shikun Zhang. 2025. Benchmarking and studying the LLM-based code review. *arXiv preprint arXiv:2509.01494*.

A Detailed System Architecture

This appendix provides a comprehensive breakdown of the Magistrate architecture.

A.1 Orchestration Layer

The orchestration layer manages the lifecycle of review tasks and coordinates computational resources via a FastAPI Server and Docker Orchestrator.

A.2 Execution Layer

Worker containers provide the execution environment, built using a multi-stage Docker build process (Python + Bun runtime + system tools).

A.3 Agent Layer

The agent layer implements the core review logic through a hierarchical multi-agent architecture. **MagistrateOrchestrator** coordinates the workflow. **Delegator** analyzes metadata to create an execution plan. **Issue Detector** analyzes batches in parallel using `batch_execute_tool`.

B Agent I/O Schemas and Prompt Structure

We implement typed inputs/outputs for each agent via Pydantic models.

Delegator (Planner) Prompt

Role: Analyze all changed files, generate a PR summary, and create optimal batches for parallel Issue Detector execution.
Input: List of DelegationCandidate objects with file paths, status, change metrics, and computed risk scores.
Output: DelegationPlan with ordered batches (each batch = 1-5 files), priority rankings, and rationale per batch. Also returns PRSummary.

```python
class PRSummary(BaseModel):
    summary: str  # concise PR overview
    key_changes: List[str]  # bullet points
    overall_impact: str  # risk assessment

class DelegationBatch(BaseModel):
    batch_name: str
    priority: int  # lower = higher priority
    files: List[str]  # file paths in batch
    rationale: str

class DelegationPlan(BaseModel):
    summary: PRSummary
    batches: List[DelegationBatch]
    notes: Optional[str]
```

Issue Detector Batch Prompt

Role: Analyze a delegation batch of related files; detect issues with evidence; verify downstream impacts.
Input: Batch name, priority, rationale, list of FileChange objects, PR context.
Output: IssueDetectorBatchResult containing batch summary, shared analysis notes, shared tools used, and per-file FileAnalysisResult objects.

```python
class FileAnalysisResult(BaseModel):
    file_path: str
    file_summary: str  # 2-3 sentences
    issues: List[ReviewIssue]
    analysis_notes: str
    tools_used: List[str]

class IssueDetectorBatchResult(BaseModel):
    batch_name: str
    batch_summary: str
    shared_analysis_notes: str
    shared_tools_used: List[str]
    files: List[FileAnalysisResult]
```

C Tool Boxes and Capabilities

batch_execute_tool: Parallel Shell Execution

Purpose: Execute multiple shell commands concurrently for fast evidence gathering.
Available Commands: ripgrep (`rg`), ast-grep (`ag`/`sg`), sed, git, ls, find, cat, head, tail, wc, grep, awk.
Usage: Issue Detector constructs command lists (e.g., search for function calls, check git history, find TODOs).
Example:

```
batch_execute_tool([
  "rg -n 'validateToken' --glob '!
      node_modules/**' src/",
  "git log --oneline -5 src/auth.py",
  "ast-grep --lang py -p 'def $FUNC($ARGS)
      :'"
])
```

D Dataset Statistics

Table 5 provides a detailed breakdown of the Magistrate-Bench dataset.

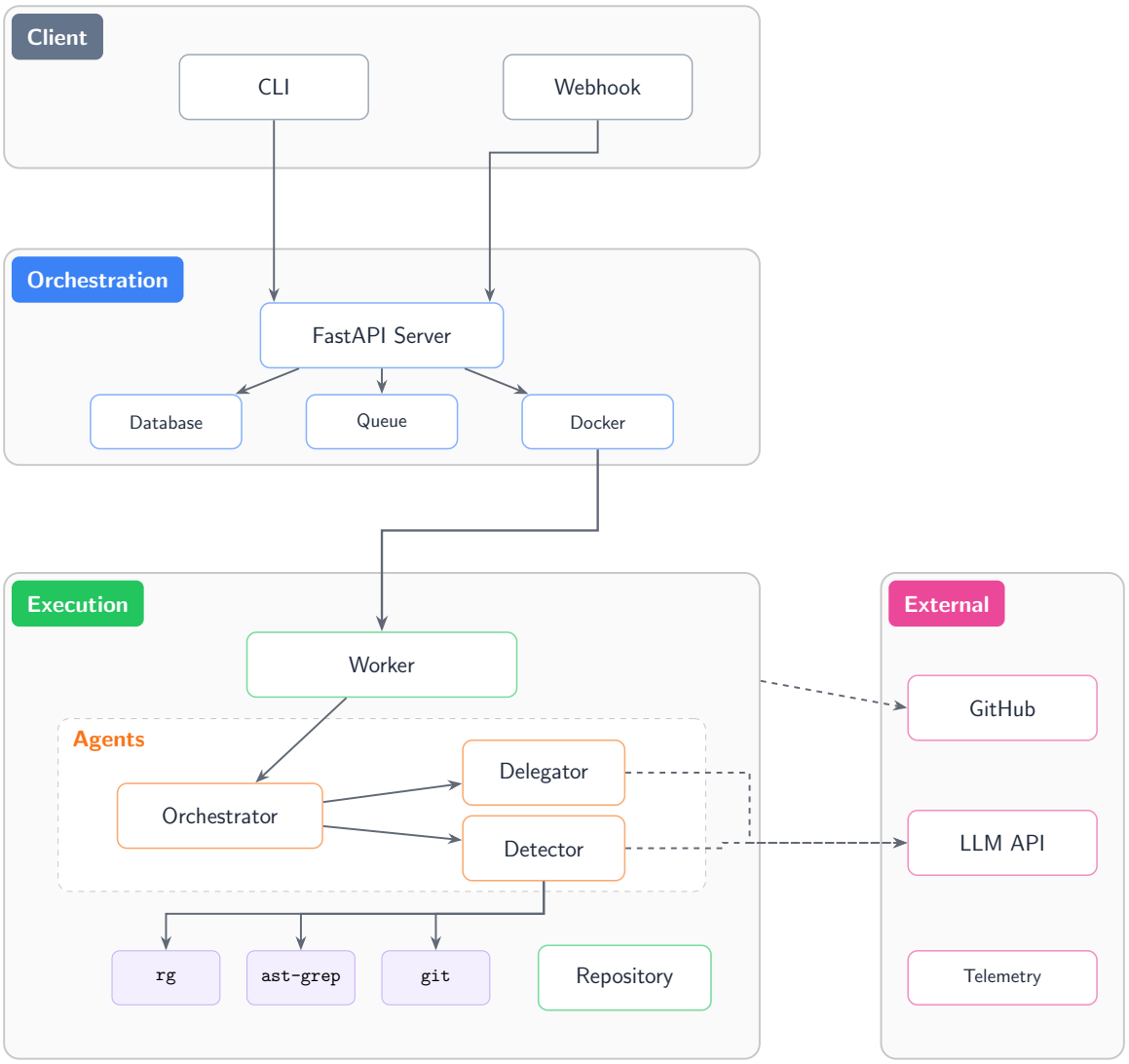

Figure 3: System architecture showing three layers: Orchestration, Execution, and Agent Layer.

General Stats		Issue Types	
Total PRs	2,042	Style	4,969
Total Repos	245	Design	2,523
Total Issues	12,121	Docs	1,931
Avg. Issues/PR	5.94	Logic	1,120
Avg. Tests/PR	0.29	Bug	973

Language Distribution

Python	389	Java	186	C#	98
JavaScript	272	Go	182	Swift	96
TypeScript	232	PHP	105	C++	90
Rust	207	Dart	99	Ruby	86

Table 5: Magistrate-Bench statistics showing dataset scale and issue type distribution.

E Server–Client Architecture and System Integration

E.1 Overview

Magistrate follows a three-layer architecture: (1) an orchestration layer (FastAPI server, SQLite task database, Docker scheduler), (2) an execution layer (one worker container per pull request), and (3) an agent layer (Orchestrator, Delegator, and IssueDetector agents). This separation isolates infrastructure concerns from agent logic and ensures reproducible, containerized evaluation.

E.2 Client Interaction

The review process begins when the client submits a request to `POST /api/review-pr`. The payload includes:

- repository identifier,
- commit SHA and diff metadata,
- PR title and description.

The server inserts a new task into the SQLite queue and returns a task identifier.

E.3 Orchestration Layer

The FastAPI server acts as the controller for task creation, status tracking, and result retrieval. A lightweight Docker orchestrator continuously polls the task database. When a new task is available, the orchestrator provisions a fresh worker container, mounts the cached repository workspace, and provides the task metadata. No computation is performed on the host machine; all review logic executes inside the container.

E.4 Execution Layer

Each worker container executes the full pipeline:

1. Extract cached repository tarball corresponding to the target commit.
2. Run Phase 0 static prescan (via `ast-grep`).
3. Invoke the Delegator to generate the `PRSummary` and `DelegationPlan`.
4. Spawn parallel IssueDetector runs over the assigned batches.
5. Perform post-LLM verification and filtering.
6. Aggregate all validated issues and write the final `review.json`.

The container then uploads the results back to the server and terminates.

E.5 Agent Layer

All agents communicate using typed Pydantic models to avoid drift in LLM output formats. The key structures include:

- `DelegationCandidate` (file path, size, risk score),
- `PRSummary` (overview, key changes, impact),
- `DelegationPlan` (ordered batches with rationale),
- `FileAnalysisResult` (summary, issues, notes),
- `ReviewIssue` (title, description, file, line, severity).

All objects serialize to JSON, ensuring reproducibility across multiple evaluation runs.

E.6 Tool Integration

The agents operate under a guarded tool palette:

- `ast-grep` for syntactic pattern matching,
- `ripgrep` for project-wide search,
- `git log` and `git diff` for history mining,
- `batch_execute_tool` for parallel shell execution.

These tools enable the agent to gather concrete evidence without hallucinated paths or nonexistent lines.

F Dataset, Models, and I/O Schemas

F.1 Dataset Structure

Each example consists of:

- PR metadata: commit SHA, diff hunks, filenames, change statistics,
- repository workspace: extracted from cached tarballs,
- ground truth issues: title, description, file, line, and human reviewer notes.

The dataset is stored in JSONL format for efficient streaming.

F.2 ReviewIssue Schema

Agents emit issues in a strictly typed structure:

```
class ReviewIssue:
    title: str
    description: str
    file_path: str
    line: int
    severity: str
```

F.3 Batch Issue Matching

To match agent issues to ground truth issues, we use a single-shot LLM call (*BatchIssueMatcher*). The model receives:

- all agent issues,
- all GT issues,
- relevant diff and code snippets.

It returns one-to-one correspondences and identifies unmatched agent issues (false positives) and unmatched GT issues (false negatives).

F.4 Outputs and Persistence

All outputs (summaries, batch results, detected issues, verification metadata) are written to a single `review.json` object per PR. This file mirrors the internal Pydantic structures and is used directly by the evaluation pipeline.

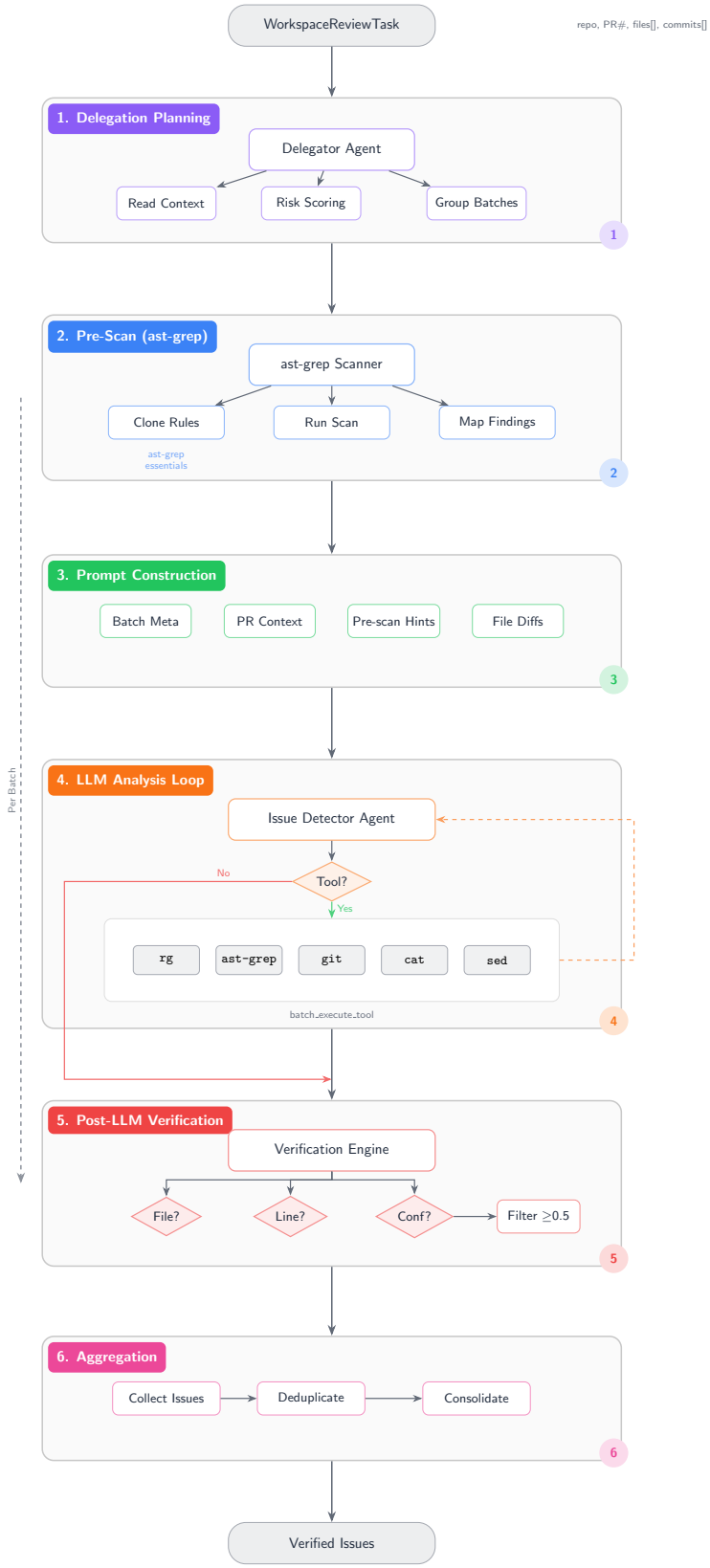

Figure 4: Issue Detector pipeline showing six phases: delegation planning, pre-scan using ast-grep essentials (coderabbitai, 2024), prompt construction, LLM analysis with tool access, post-LLM verification, and aggregation.