Code Broker A Multi Agent System for Automated Code Quality Assessment

Samer Attrah

samiratra95@gmail.com

Independent researcher https://orcid.org/0009-0006-8090-1256

Research Article

Keywords: AI Agents, Code Quality Assessment, Multi Agent Systems, Software Engineering, Google Agent Development Kit ADK, Python.

Posted Date: May 8th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9619569/v1

Additional Declarations: The authors declare no competing interests.

Code Broker: A Multi-Agent System for Automated Code Quality Assessment

Samer Attrah
Independent researcher
ORCiD: 0009-0006-8090-1256
samiratra95@gmail.com

April 2026

Abstract

We present **Code Broker**, a multi-agent system built with Google's Agent Development Kit (ADK) that analyses Python code from files, local directories, or GitHub repositories and generates actionable quality assessment reports. The system employs a hierarchical five-agent architecture in which a root orchestrator coordinates a sequential pipeline agent, which in turn dispatches three specialised agents in parallel—a Correctness Assessor, a Style Assessor, and a Description Generator—before synthesising findings through an Improvement Recommender. Reports score four dimensions—correctness, security, style, and maintainability—and are rendered in both Markdown and HTML. Code Broker combines LLM-based reasoning with deterministic static-analysis signals from Pylint, uses asynchronous execution with retry logic to improve robustness, and explores lightweight session memory for retaining and querying prior assessment context. We position the paper as a technical report on system design and prompt/tool orchestration, and present a preliminary qualitative evaluation on representative Python codebases. The results suggest that parallel specialised agents produce readable, developer-oriented feedback, while also highlighting current limitations in evaluation depth, security tooling, large-repository handling, and the current use of only in-memory persistence. All code and reproducibility materials are available at [27].

Keywords: AI Agents, Code Quality Assessment, Multi-Agent Systems, Software Engineering, Google Agent Development Kit (ADK), Python.

1 Introduction

Software quality assessment is a critical yet time-intensive activity in the software development lifecycle. Manual code review is subject to reviewer fatigue, inconsistency, and bottlenecks in busy teams. Automated static analysis tools such as Pylint, Flake8, or SonarQube partially address this gap; however, they operate at the syntactic and stylistic level and rarely provide the nuanced, context-sensitive feedback that a skilled senior engineer would offer.

The advent of large language model (LLM)-powered agents opens a new paradigm: an agent that can *understand* code semantics, reason about logical correctness, identify security anti-patterns, and recommend domain-specific improvements—all in natural language. Code Broker explores this paradigm by building a multi-agent system that combines the strengths of traditional static analysis with the contextual reasoning capabilities of LLM agents.

Code Broker was developed as part of the *Google/Kaggle 5-Day AI Agents Intensive* capstone project [1]. The current implementation targets Python codebases and accepts three input modalities: individual files, local directory trees, and remote GitHub repositories.

1.1 Motivation

The core motivation behind Code Broker is to make code review support more accessible. Junior developers and solo practitioners often lack access to experienced reviewers and fast reviewing with numerical accessment and quality measurement metrics. Code Broker aims to provide fast, structured feedback that goes beyond line-level linting to encompass architecture, logic, security, and maintainability. Additionally, a multi-agent architecture enables *concurrent* specialised feedback rather than serialising all analysis through a single monolithic prompt.

1.2 Contributions

The primary contributions of this work are:

1. A working five-agent hierarchical architecture for code quality assessment, implemented with Google ADK.

2. A parallel assessment strategy decoupling correctness, style, and description analysis for speed and modularity.
3. Integration of Pylint as an ADK tool, grounding LLM outputs with concrete static-analysis evidence.
4. A preliminary qualitative evaluation on representative Python codebases, framed as an initial system study rather than a full benchmark.
5. Open-source release of all code, notebooks, and documentation [27].

2 Related Work

The development of Code Broker is situated at the intersection of multi-agent systems, automated software engineering, and large language model (LLM) orchestration. This section reviews the theoretical and practical foundations that underpin our proposed architecture. We begin by defining AI agents and the multi-agent paradigms that enable complex task decomposition. Next, we describe the Google Agent Development Kit (ADK), the framework utilized for constructing our hierarchical agent pipeline. We then survey contemporary approaches to automated code review, highlighting the shift from deterministic static analysis to context-aware LLM agents. Finally, we discuss emerging research in human-agent collaboration and normative coordination, which provides the sociotechnical context for deploying such systems in real-world development environments.

2.1 AI Agents and Multi-Agent Systems

An AI agent is an autonomous system that perceives its environment, reasons about goals, selects actions, and executes those actions via tools [2, 3]. The ReAct paradigm [2] forms a conceptual basis for many modern LLM agents. Multi-agent systems (MAS) extend this idea by decomposing complex tasks across specialised agents coordinated by an orchestrator [11]. In software engineering settings, this decomposition is attractive because review tasks naturally separate into analysis, evidence gathering, summarisation, and recommendation stages.

2.2 Google Agent Development Kit (ADK)

Google's ADK is an open-source Python framework designed to simplify the construction, orchestration, and evaluation of AI agents. Key ADK abstractions used in Code Broker include:

- **LLMAgent**: A language-model-backed agent that accepts a system prompt, a set of tools, and sub-agents.
- **SequentialAgent**: Chains agents in a fixed order, passing state between steps.
- **ParallelAgent**: Dispatches multiple child agents concurrently and merges their outputs.
- **Runner**: Manages the execution lifecycle, session state, and event loop.
- **AgentTool**: Wraps an agent as a callable tool, enabling agent-of-agents patterns.
- **Memory Service**: Provides session persistence and semantic retrieval via `InMemoryMemoryService`, enabling context preservation across assessments and query-based memory search.

2.3 Automated Code Review

Prior work on automated code review falls into three broad categories. *Static analysis tools* [4] such as Pylint provide deterministic warnings and style checks but limited semantic interpretation. *Learning-based approaches* [5] support defect prediction and code understanding, but often depend on benchmark-specific supervision. *LLM-based agents* [6, 7] leverage planning and tool use to generate contextual review feedback. Recent systems such as *HyperAgent* [17] and *Agyn* [9] frame software engineering as a collaborative team process, while survey work highlights both the promise and reliability challenges of agentic software engineering pipelines [13, 14].

Code Broker sits at the intersection of these categories: it uses Gemini for reasoning, Pylint for deterministic evidence, and a simple hierarchical orchestration pattern to separate description, correctness, style, and recommendation tasks. Relative to prior work, the goal here is not to propose a new theory, but to document a concrete, reproducible code-assessment workflow built with ADK and to examine its practical trade-offs in a capstone setting.

2.4 Human-Agent Collaboration and Norms

As agents become more autonomous, the strategic allocation of tasks between humans and AI Agents becomes important. Ronanki [15] discusses trustworthy human-agent collaboration, while Dam et al. [12] study normative coordination in human-AI engineering teams. These works are relevant as design context, although Code Broker itself remains a developer-assistance tool with human interpretation of the final report still required.

3 Methodology

The design of a multi-agent system (MAS) involves navigating a complex landscape of task requirements, framework constraints, and model performance. In the development of Code Broker, the orchestration logic—including the specific hierarchy of agents and the selection of integrated tools—was engineered to align with the capabilities of the Google Agent Development

Kit (ADK) and the high-capacity reasoning of the Gemini models. This configuration ensures a robust assessment process that balances deep semantic analysis with operational efficiency and execution speed. This section details the system's architecture, individual agent roles, and the integrated methodology used for code quality evaluation.

3.1 Overview

Code Broker is organised as a *hierarchical multi-agent system* with five distinct agents arranged in two layers (Figure 1). The orchestrator coordinates a pipeline which fans out to parallel specialists before a synthesiser merges results. The design goal is modularity: each agent has a narrow role, an explicit output contract, and limited responsibility for a single review dimension.

3.2 Agent Descriptions

1. Report Generator (Orchestrator). The root agent of the system. It accepts the user's input (file path, directory path, or GitHub URL), invokes the Sequential Pipeline Agent as a sub-agent tool, and formats the final consolidated report in Markdown and HTML. Its system prompt instructs it to maintain a professional tone and to surface actionable, developer-friendly language.

2. Sequential Pipeline Agent. A `SequentialAgent` that manages the overall assessment workflow. It first invokes the Parallel Assessment Agent and, once all parallel outputs are available, hands results to the Improvement Recommender. It also handles pre-processing: 1) reading file contents, 2) cloning or fetching GitHub repositories, and 3) chunking large files if necessary.

3. Parallel Assessment Agent. A `ParallelAgent` that dispatches three child agents simultaneously, reducing total latency. It merges their independent outputs into a structured intermediate representation consumed by the Improvement Recommender.

4a. Correctness Assessor. Analyses the logical and functional correctness of the code. It checks for algorithmic errors, off-by-one mistakes, improper error handling, and potential runtime exceptions. It also runs Pylint via an ADK tool and incorporates the output into its reasoning. In the current system, security-related observations are also surfaced here when they are inferable from code structure or lint findings.

4b. Style Assessor. Evaluates PEP 8 compliance, naming conventions, code organisation, and readability. It examines documentation coverage (docstrings, inline comments), complexity metrics, and adherence to project conventions where detectable.

4c. Description Generator. Produces a concise natural-language summary of what the code does—its purpose, main components, and architectural patterns. This grounds the other assessors and provides context for the final report.

5. Improvement Recommender. Synthesises all upstream outputs into a prioritised list of improvement recommendations. It scores the codebase on four dimensions (Table 1) and produces the final report.

3.3 Scoring Dimensions

Table 1: Code Broker scoring dimensions (0–10 scale).

Dimension	Description
Correctness	Logical and functional accuracy; presence of bugs, runtime errors, or incorrect algorithmic behaviour.
Security	Heuristic vulnerability assessment: possible injection risks, unsafe deserialization, hard-coded secrets, and insecure dependency usage inferred from code review. The current implementation does not yet include a dedicated security scanner.
Style	PEP 8 compliance, naming conventions, code readability, documentation quality.
Maintainability	Modularity, coupling, cohesion, test coverage indicators, complexity metrics.

3.4 Implementation Details and Tool Integration

The current prototype was developed and exercised as a notebook-driven system around Google ADK components, with the report generator acting as the top-level entry point. Prompts were manually iterated during the capstone development cycle and tuned for structured output, evidence citation, and concise recommendation synthesis. The implementation should therefore be read as a reproducible system report, but not yet as a controlled study of prompt variants or orchestration strategies.

In parallel with the notebook prototype, a separate packaging effort has been started [28] to migrate Code Broker into a reusable Python distribution with a `src/code_broker` layout, a `pyproject.toml` build definition, and a `code-broker` command-line entry point. This packaging work is relevant to the research trajectory because it pushes the system from a course artifact toward a deployable developer tool, with clearer module boundaries for agents, tools, configuration, reporting, and runtime orchestration.

3.5 Technology Stack

Code Broker is implemented in Python 3.14 and relies on the following key dependencies:

- **google-adk**: Multi-agent orchestration framework.
- **google-generativeai**: Gemini model API access.

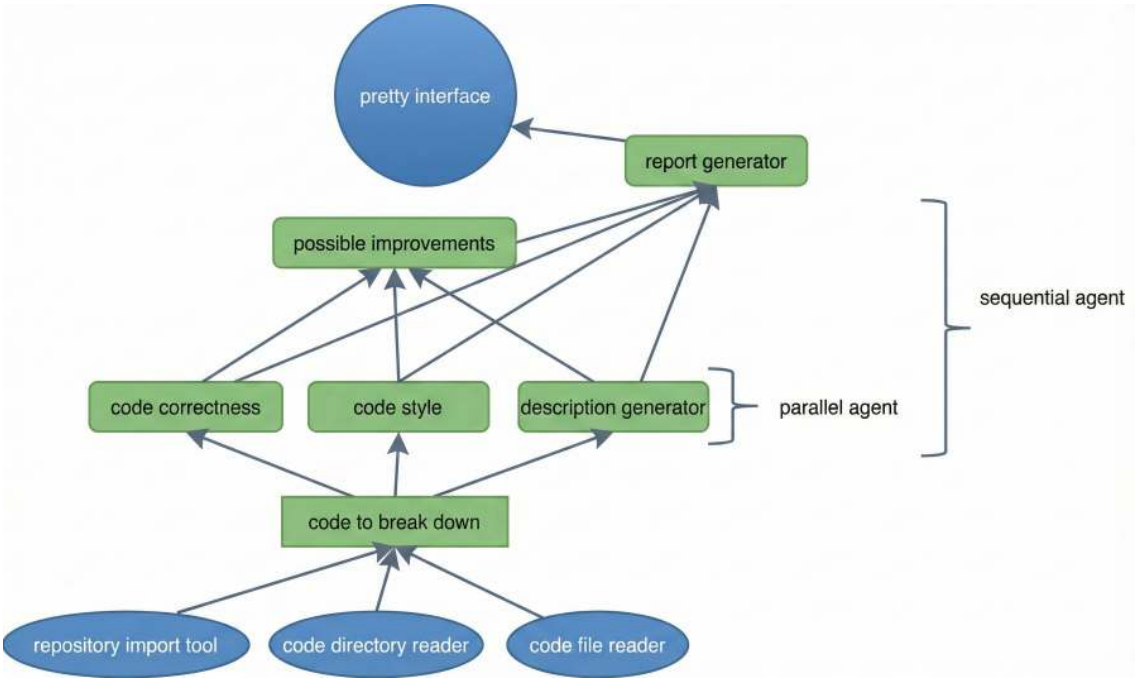

Figure 1: Hierarchical five-agent architecture of Code Broker. The orchestrator coordinates a sequential pipeline, which fans-out to three parallel assessors before the Improvement Recommender synthesises a final report.

- **PyGithub**: GitHub repository fetching and file enumeration.
- **pylint**: Static Python code analysis, invoked as a subprocess tool.
- **Jupyter / IPython**: Interactive execution environment via `notebooks/code_broker.ipynb`.
- **markdown2**: Markdown-to-HTML rendering for report output.

3.6 Reproducibility Notes

The implementation uses Gemini-family models accessed through the Google generative AI stack, but the exact serving model can be configured externally in the runtime environment. For this reason, outputs should be considered *configuration-dependent*. In the experiments reported here, we focus on workflow behaviour and report characteristics rather than strict model-to-model comparison. Future versions of the paper should pin the exact model identifier, decoding settings, and prompt templates in an appendix or artifact bundle.

3.7 Session Memory and Retrieval

The notebook prototype also includes an initial memory workflow built on ADK's `InMemoryMemoryService`. After a report-generation run completes, the active session can be written into memory via `add_session_to_memory`, and subsequent queries can retrieve prior report context through `search_memory`. In practical terms, this provides a lightweight mechanism for asking questions such as what previous reports said about a repository or codebase without rerunning the full analysis pipeline.

At present, this memory mechanism remains transient and notebook-scoped: it is useful for demonstrating conversational continuity and report recall, but it is not yet a persistent longitudinal store. This is an important distinction for the research framing. The current implementation shows that memory can enrich post-analysis interaction, while a more mature version would need durable storage, repository-level indexing, and stronger controls over what historical findings are retained and resurfaced.

3.8 Input Handling

The system supports three input modalities:

1. **Single file**: The file is read from the local filesystem, chunked if it exceeds the model's context window, and passed directly to the pipeline.
2. **Directory**: All Python (`.py`) files under the target directory are enumerated recursively, and each is assessed individually before the results are aggregated.
3. **GitHub repository URL**: The PyGithub library fetches repository metadata and file contents via the GitHub API, respecting rate limits and authentication via an optional `GITHUB_TOKEN` environment variable.

For directories and repositories, file-level outputs are aggregated into a repository-level summary by the downstream recommendation stage. When a file is too large for direct inclusion in the prompt context, it is chunked before analysis. This chunking strategy improves coverage, but it can also reduce cross-file and

long-range reasoning quality; we therefore treat current large-repository handling as a practical compromise rather than a solved problem.

3.9 Pylint Tool Integration

The Pylint integration is wrapped as an ADK `FunctionTool`. The tool accepts a code string, writes it to a temporary file, runs Pylint as a subprocess, and returns the structured JSON output. The Correctness Assessor invokes this tool and incorporates the linting findings into its analysis.

Figure 2: Simplified Pylint ADK tool wrapper.

```
import subprocess, tempfile, json
def run_pylint(code:  str) -> dict:
  """Run Pylint on the provided Python source
  and return results."""
  with tempfile.NamedTemporaryFile(
    suffix=".py", mode="w", delete=False
  ) as f:
    f.write(code)
    tmp_path = f.name
  result = subprocess.run(
    ["pylint", tmp_path, "--output-format=json"],
    capture_output=True, text=True
  )
  try:
    return json.loads(result.stdout)
  except json.JSONDecodeError:
    return {"error":  result.stdout}
```

3.10 Asynchronous Processing and Retry Logic

All agent invocations are performed asynchronously using Python's `asyncio`. The ParallelAgent dispatches three coroutines concurrently, and results are gathered with `asyncio.gather`. A lightweight exponential-backoff retry decorator wraps each agent call to handle transient API errors (rate limits, timeouts). The retry policy uses a maximum of three attempts with jitter. This improves robustness in practice, but it also means end-to-end latency depends on external API behaviour and should not be interpreted as a stable benchmark result.

3.11 Report Generation

The final report is structured as follows:

1. **Executive Summary**: One-paragraph description of the codebase generated by the Description Generator.
2. **Scores Table**: Four-dimension score table with a brief rationale per dimension.
3. **Correctness Analysis**: Detailed findings from the Correctness Assessor, including Pylint output.
4. **Style Analysis**: Findings from the Style Assessor.
5. **Improvement Recommendations**: Numbered, prioritised list of actions from the Improvement Recommender.
6. **Conclusion**: Overall assessment and suggested next steps.

Reports are rendered in both Markdown (for downstream toolchain integration) and HTML (for direct browser viewing), with syntax-highlighted code snippets where relevant.

4 Results

4.1 Evaluation Setup

We evaluate Code Broker as a *preliminary system study* rather than a full benchmark. The goal of this section is to assess whether the agent pipeline produces coherent, actionable reports across representative Python inputs. Crucially, Code Broker is not only intended for assessing and evaluating code quality, but also for summarizing and facilitating the understanding of code structures and logic, particularly for unfamiliar codebases. Therefore, the evaluation also considers the clarity and accuracy of the generated summaries and their effectiveness in aiding developer comprehension.

The evaluation used a small set of representative Python codebases drawn from three categories:

- **Toy scripts**: Small, self-contained utility scripts (50–200 lines).
- **Medium projects**: Open-source utilities with multiple modules (500–2000 lines).
- **GitHub repositories**: Public repositories fetched via the GitHub API.

Each case was reviewed manually using four questions: (1) whether the generated description matched the apparent purpose of the code, (2) whether the correctness findings were specific and evidence-backed, (3) whether recommendations were actionable for a developer, and (4) whether the overall report was readable as a standalone artifact. We did not use ground-truth bug labels, inter-rater agreement, or a formal baseline in this version of the study; those remain future work.

4.2 Observed Outcomes

Across the cases examined, the system generally produced well-structured reports with readable sectioning and concrete next steps. The Description Generator was especially useful for unfamiliar repositories because it provided orientation before the detailed assessment sections. The Correctness Assessor's use of Pylint output improved traceability by anchoring at least part of the review in line-level evidence instead of free-form model claims.

The most consistent weaknesses appeared on larger repositories and in the security dimension. For multi-file projects, chunking and file-level aggregation sometimes reduced architectural coherence. For security, the absence of a dedicated scanner meant that the security score was best interpreted as a heuristic review signal rather than as a comprehensive vulnerability assessment.

4.3 Report Quality Dimensions

Table 2 summarises qualitative observations across evaluation cases.

Table 2: Preliminary qualitative evaluation summary across representative codebases.

Quality Aspect	Toy Scripts	Medium Projects	GitHub Repos
Description Accuracy	High	High	Medium
Correctness Coverage	High	Medium	Medium
Style Feedback Relevance	High	High	High
Recommendation Actionability	High	High	Medium
Report Readability	High	High	High

4.4 Threats to Validity and Limitations

The present evaluation and system have the following limitations:

1. **Context window constraints**: Very large codebases exceed Gemini's context window, requiring chunking strategies that may fragment logical structure.

2. **Limited evaluation protocol**: The study is qualitative, small-scale, and author-assessed. It does not yet include labelled benchmarks, blind human evaluation, or statistical comparison against baselines.

3. **Ephemeral memory**: The notebook demonstrates session memory and memory search, but the current setup uses in-memory services only. Memory is therefore not durable across deployments and does not yet provide repository-scale historical tracking.

4. **Python-centric**: Pylint integration is Python-specific; extending to other languages requires additional tool wrappers.

5. **Weak security grounding**: The system reports a security score, but current security analysis is heuristic and not yet backed by a dedicated scanner such as Bandit.

6. **No execution**: The system performs static analysis only; dynamic bugs (e.g., race conditions) may be missed. Future iterations could incorporate formal model checking [24] or runtime verification layers [23] to monitor execution events and ensure normative compliance [12].

7. **LLM hallucinations**: Like all LLM-based systems, the assessors may occasionally produce plausible-sounding but incorrect findings, particularly for domain-specific logic.

8. **Rate limits**: GitHub API and Gemini API rate limits can slow analysis of repositories with many files.

5 Discussion

This section provides a critical analysis of the system's performance, focusing on the efficacy of the agentic orchestration and the quality of the generated multi-modal assessments. We examine the impact of prompt engineering on agent behavior, the resolution of common coordination challenges, and the practical value of the resulting reports for developer comprehension. Furthermore, we discuss the trade-offs between analysis depth and execution speed identified during our preliminary system study.

5.1 Agent Prompt Engineering

Effective prompt design was critical to achieving high-quality agent outputs. Key principles applied:

Role specification. Each agent's system prompt begins with a concise role statement (e.g., *"You are an expert Python code correctness assessor..."*), setting persona and expertise level.

Output format constraints. Agents are instructed to return structured output with explicit section headers (e.g., `## Findings`, `## Score`) to facilitate downstream parsing by the Improvement Recommender.

Evidence grounding. The Correctness Assessor is explicitly instructed to *cite line numbers and Pylint message codes* when reporting issues, reducing vague or unsupported claims.

Synthesis instructions. The Improvement Recommender's prompt instructs it to *de-duplicate* findings from the three parallel agents, *rank* recommendations by severity and impact, and produce at most ten concrete action items.

Tone calibration. All agents are instructed to use constructive, professional language suitable for a developer audience, avoiding condescension or excessive praise.

5.2 Lessons Learned

This project was developed over an intensive five-day period as part of the Google/Kaggle AI Agents Intensive. Key lessons learned:

1. **Parallelism improves workflow responsiveness**: Running the three assessors in parallel reduced waiting time in exploratory runs and yielded more independent perspectives, as each agent focused on its specialisation without being influenced by the others. We do not yet report a formal latency benchmark.

2. **Tool grounding reduces hallucinations**: Incorporating deterministic tools (Pylint) dramatically improved the factual accuracy of the Correctness Assessor. Pure LLM reasoning without tool grounding sometimes missed clear Pylint errors.

3. **Hierarchical orchestration scales**: The two-layer architecture (Orchestrator → Pipeline → Parallel Agents) cleanly separated concerns and made it easy to add new assessors without modifying the top-level logic.

4. **Context management is critical**: Handling large repositories required careful chunking and summarisation strategies to avoid context overflow. Future work will explore retrieval-augmented approaches.

5. **Prompt iteration is non-trivial**: Agent quality is highly sensitive to prompt wording. Future work will leverage online prompt optimization frameworks like *HiveMind* [22] for contribution-guided refinement.

5.3 Robustness and Reliability

Ensuring reliability in production-like environments remains a challenge. We propose using chaos engineering [25] to proactively identify vulnerabilities such as hallucinations or communication failures. Furthermore, adding dedicated security analysis agents can mitigate risks of code injection and poisoning attacks [16].

5.4 Future Directions

Several directions for future improvement are identified:

- **Pip package maturation**: Continue the migration from notebook-centric execution to a fully installable `pip` package with a stable CLI, pinned dependencies, artifact versioning, and test coverage for packaged modules. This would make the system easier to reproduce, evaluate, and integrate into developer workflows.

- **Multi-language support**: Adding language-specific static analysis tools (e.g., ESLint for JavaScript, Checkstyle for Java) to extend beyond Python.

- **Incremental analysis**: Integrating with Git diff APIs to provide assessment of only changed files in pull requests, enabling CI/CD integration.

- **Memory and sessions**: Extending the current notebook-level memory flow into persistent assessment history per repository using ADK session and memory services, enabling longitudinal tracking of code quality, retrieval of earlier reports, and multi-turn follow-up analysis grounded in prior runs.

- **Evaluation benchmark**: Constructing a labelled benchmark of code samples with ground-truth quality annotations for quantitative evaluation.

- **Interactive review mode**: Adding a conversational mode in which developers can ask follow-up questions about specific findings.

- **Security scanner integration**: Incorporating dedicated security scanners (e.g., Bandit for Python) alongside Pylint.

- **Commercial Viability**: Exploring monetization models for Code Broker, such as a premium SaaS tier offering advanced security deep-scans, priority execution infrastructure, and enterprise-grade repository indexing to support sustainable development and professional use-cases.

6 Conclusion

Code Broker demonstrates that a hierarchical multi-agent architecture can produce structured, developer-oriented code quality assessments that go beyond traditional static analysis alone. By parallelising specialised agents, integrating deterministic tools, and leveraging Google's ADK orchestration framework, the system provides a practical workflow for exploratory code review in Python projects. As presented here, the contribution is best understood as a technical report on system design and an initial qualitative evaluation rather than as a definitive benchmark study.

The capstone context provided a useful setting for rapidly iterating on agent design, prompt structure, and tool integration. The next step for this work is twofold: strengthen the empirical section with fixed model settings, stronger baselines, dedicated security tooling, and a benchmark with labelled review outcomes; and complete the transition to a distributable Python package so the system can be evaluated and adopted outside the notebook environment.

7 Declarations

7.1 Ethics approval and consent to participate

Not applicable.

7.2 Consent for publication

Not applicable.

7.3 Availability of data and materials

All code and reproducibility materials are available at the project repository [27].

7.4 Competing interests

The author declares that there are no competing interests.

7.5 Funding

No funding was received for this work.

7.6 Authors' contributions

SA is the sole author of this work and was responsible for system design, implementation, and evaluation.

7.7 Acknowledgements

The author thanks Google and Kaggle for organising the 5-Day AI Agents Intensive course and capstone competition. Thanks to the Google ADK team for the open-source framework. This project is released under the Apache License 2.0.

References

[1] Google & Kaggle. (2026). *Inside Kaggle's AI Agents Intensive Course with Google*. Google Blog. https://blog.google/innovation-and-ai/technology/developers-tools/ai-agents-intensive-recap/

[2] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). *ReAct: Synergizing Reasoning and Acting in Language Models*. International Conference on Learning Representations (ICLR).

[3] Weng, L. (2023). *LLM-powered Autonomous Agents*. Lil'Log. https://lilianweng.github.io/posts/2023-06-23-agent/

[4] Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A., & Gall, H. C. (2019). *How developers engage with static analysis tools in different contexts*. Empirical Software Engineering, 24(2), 1419–1457.

[5] Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., et al. (2021). *CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation*. NeurIPS Datasets and Benchmarks Track.

[6] Hong, S., Zheng, X., Chen, J., Cheng, Y., Wang, J., Zhang, C., et al. (2023). *MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework*. arXiv preprint arXiv:2308.00352.

[7] Qian, C., Cong, X., Yang, C., Chen, W., Su, Y., Xu, J., et al. (2023). *Communicative Agents for Software Development*. arXiv preprint arXiv:2307.07924.

[8] Chen, J., Guo, X., Chen, S., Cheung, S. C., & Shen, J. (2025). *Multi-Agent Systems for Dataset Adaptation in Software Engineering: Capabilities, Limitations, and Future Directions*. arXiv preprint arXiv:2511.21380.

[9] Benkovich, N., & Valkov, V. (2026). *Agyn: A Multi-Agent System for Team-Based Autonomous Software Engineering*. arXiv preprint arXiv:2602.01465.

[10] Zhang, W., Zhou, Y., Qu, H., & Li, H. (2026). *Loosely-Structured Software: Engineering Context, Structure, and Evolution Entropy in Runtime-Rewired Multi-Agent Systems*. arXiv preprint arXiv:2603.15690.

[11] Cai, Y., Li, R., Liang, P., Shahin, M., & Li, Z. (2025). *Designing LLM-based Multi-Agent Systems for Software Engineering Tasks: Quality Attributes, Design Patterns and Rationale*. arXiv preprint arXiv:2511.08475.

[12] Dam, H. K., Mahala, G., Hoda, R., Zheng, X., & Conati, C. (2025). *Towards autonomous normative multi-agent systems for Human-AI software engineering teams*. arXiv preprint arXiv:2512.02329.

[13] Tang, Y., & Runkler, T. (2026). *LLM-Based Agentic Systems for Software Engineering: Challenges and Opportunities*. arXiv preprint arXiv:2601.09822.

[14] He, J., Treude, C., & Lo, D. (2024). *LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision and the Road Ahead*. arXiv preprint arXiv:2404.04834.

[15] Ronanki, K. (2025). *Facilitating Trustworthy Human-Agent Collaboration in LLM-based Multi-Agent System oriented Software Engineering*. arXiv preprint arXiv:2505.04251.

[16] Bowers, B., Khapre, S., & Kalita, J. (2025). *Analyzing Code Injection Attacks on LLM-based Multi-Agent Systems in Software Development*. arXiv preprint arXiv:2512.21818.

[17] Phan, H. N., Nguyen, T. N., Nguyen, P. X., & Bui, N. D. Q. (2024). *HyperAgent: Generalist Software Engineering Agents to Solve Coding Tasks at Scale*. arXiv preprint arXiv:2409.16299.

[18] Zhu, A., Dugan, L., & Callison-Burch, C. (2024). *ReDel: A Toolkit for LLM-Powered Recursive Multi-Agent Systems*. arXiv preprint arXiv:2408.02248.

[19] Elhashemy, H., Lotfy, Y., & Tang, Y. (2025). *Bridging the Prototype-Production Gap: A Multi-Agent System for Notebooks Transformation*. arXiv preprint arXiv:2511.07257.

[20] Weyns, D., & Oquendo, F. (2019). *An Architectural Style for Self-Adaptive Multi-Agent Systems*. arXiv preprint arXiv:1909.03475.

[21] Amaral, C. J., Hübner, J. F., & Kampik, T. (2020). *Towards Jacamo-rest: A Resource-Oriented Abstraction for Managing Multi-Agent Systems*. arXiv preprint arXiv:2006.05619.

[22] Xia, Y., Wang, T., Zhang, S., Weng, Z., Cao, B., & Liew, S. C. (2025). *HiveMind: Contribution-Guided Online Prompt Optimization of LLM Multi-Agent Systems*. arXiv preprint arXiv:2512.06432.

[23] Engelmann, D. C., Ferrando, A., Panisson, A. R., Ancona, D., Bordini, R. H., & Mascardi, V. (2022). *RV4JaCa – Runtime Verification for Multi-Agent Systems*. arXiv preprint arXiv:2207.09708.

[24] Ferrando, A., & Malvone, V. (2024). *VITAMIN: A Compositional Framework for Model Checking of Multi-Agent Systems*. arXiv preprint arXiv:2403.02170.

[25] Owotogbe, J. (2025). *Assessing and Enhancing the Robustness of LLM-based Multi-Agent Systems Through Chaos Engineering*. arXiv preprint arXiv:2505.03096.

[26] Goyal, M., & Bhasin, P. (2025). *Moving From Monolithic To Microservices Architecture for Multi-Agent Systems*. arXiv preprint arXiv:2505.07838.

[27] Attrah, S. (2026). *Code Broker: Multi-Agent System for Automated Code Quality Assessment*. Main Project Repository. https://github.com/Samir-atra/agents_intensive_dev

[28] Attrah, S. (2026). *Code Broker Package: Reusable Python Distribution for Automated Code Assessment*. GitHub Repository. https://github.com/Samir-atra/Code_broker_pkg