ACR-QA

Automated Code Review & Quality Assurance Platform

Graduation Project Proposal

Presented by:

Ahmed Mahmoud Abbas

Student ID: 222101213

ahmed222101213@ksiu.edu.eg

Under the Supervision of:

Dr. Samy AbdelNabi

Department of Computer Science

Faculty of Computer Science and Engineering

King Salman International University

November 5, 2025

Contents

1 Question 1: Project Title and Aims

1.1 Specific Title

ACR-QA: Automated Code Review & Quality Assurance Platform — Language-Agnostic with RAG-Enhanced AI Explanations

1.2 Project Aims

1. **Build Multi-Language Code Review Platform:** Design and implement a platform with pluggable language adapters to detect bad practices, security vulnerabilities, code duplication, style violations, and design anti-patterns across multiple programming languages starting with Python.

2. **Implement RAG-Enhanced AI Explanations:** Integrate a hybrid explanation engine using Retrieval-Augmented Generation (RAG) to ground AI explanations in rule definitions, reducing hallucinations. Include template fallback for offline deployments.

3. **Achieve Measurable Quality:** Deliver detectors reaching 70% precision on high-severity rules with ¡30% false positive rate, validated through labeled test datasets.

4. **Validate Through User Study:** Conduct pilot study with 8-10 participants achieving 3.0/5.0 median usefulness rating, with AI explanations scoring higher than template-based explanations.

5. **Enable On-Premises Deployment:** Package complete system as Docker Compose application running locally without external dependencies or recurring costs.

1.3 Tangible Deliverables

- Working software system with multiple language adapters (Python completed, additional languages to be determined)
- Docker Compose deployment package with complete documentation
- Rule catalog (`config/rules.yml`) with 50+ detection rules
- Evaluation report with precision/recall metrics and user study results
- Adapter SDK documentation enabling future language additions
- Web dashboard for viewing findings and providing feedback
- Demonstration video showing end-to-end workflow

2 Question 2: Project Proposal

2.1 Problem Definition

2.1.1 Real Need and Application

Code review is a critical bottleneck in software development. Review quality varies dramatically based on reviewer availability and workload, leading to missed security vulnerabilities and design issues that reach production. Traditional linters catch only style violations, missing deeper concerns like security patterns, design anti-patterns, and maintainability issues.

Commercial static analysis tools (SonarQube Enterprise, Coverity) cost $10,000-50,000 annually, prohibitive for small teams, open-source projects, and educational institutions. Cloud-based tools require uploading proprietary code to external services, which many organizations cannot accept due to security policies.

When automated tools do provide feedback, they often lack explanations of why issues matter and how to fix them. When AI is used for explanations, hallucinations create false guidance that erodes developer trust.

2.1.2 Project Scope

In Scope:

- Multi-language architecture with pluggable adapters (Python implemented first, additional languages selected during development)
- Detection categories: unused code, style violations, security patterns, code duplication, complexity, design smells
- RAG-enhanced explanation engine with AI integration (using free Cerebras AI models API) and template fallback
- Canonical findings schema normalizing outputs from multiple tools
- GitHub/GitLab PR integration with automated commenting
- Web dashboard with false positive feedback loop
- On-premises Docker Compose deployment with zero recurring costs
- Evaluation with labeled datasets and user study

Out of Scope:

- Deep interprocedural dataflow analysis
- Dynamic analysis or test execution
- Automatic code fix commits

- Production horizontal autoscaling
- Training custom AI models from scratch

2.1.3 Target Customers

1. **University instructors** teaching software engineering courses needing automated feedback on student pull requests
2. **Small development teams** (5-20 engineers) lacking budgets for enterprise tools but needing quality gates
3. **Open-source maintainers** receiving hundreds of PRs from varied contributors
4. **Technical recruiters** providing standardized feedback on coding assessments

2.2 Suggested Solution

2.2.1 System Architecture

The system follows a layered adapter architecture enabling language-agnostic analysis. The Adapter Layer routes files by extension to language-specific toolchains. The Python adapter uses Ruff for fast style checking, Vulture for dead code detection, Radon for complexity metrics, Semgrep for security patterns, Bandit for Python-specific security analysis, and jscpd for duplication detection across languages. Additional language adapters will follow this same pattern with appropriate toolchains.

The Normalization Layer maps tool outputs to a canonical findings schema stored in PostgreSQL. Each finding includes standardized fields: rule ID, severity, confidence score, location, and raw tool output for provenance. This decouples language specifics from core platform logic, enabling the same explanation engine and dashboard to work across all supported languages.

The Explanation Layer uses RAG to retrieve relevant rule definitions from a vector-indexed knowledge base, then generates natural language explanations via Cerebras AI API with evidence-grounded prompting. All prompts and responses are logged to a provenance database for auditability and hallucination detection. Template fallback provides deterministic explanations when AI confidence is low or API unavailable, ensuring the system functions offline with zero recurring costs.

Integration with GitHub/GitLab occurs via webhooks that trigger analysis jobs queued in Redis. Results post as PR comments and display in a React dashboard where developers mark false positives, feeding back into rule tuning. On-premises deployment via Docker Compose ensures no external dependencies beyond optional LLM API calls.

2.2.2 System Components

The system comprises nine integrated components working together:

The Ingest Service captures GitHub/GitLab webhook events on pull request creation or updates, extracts diff hunks, and enqueues analysis jobs. The Job Queue (Redis/BullMQ)

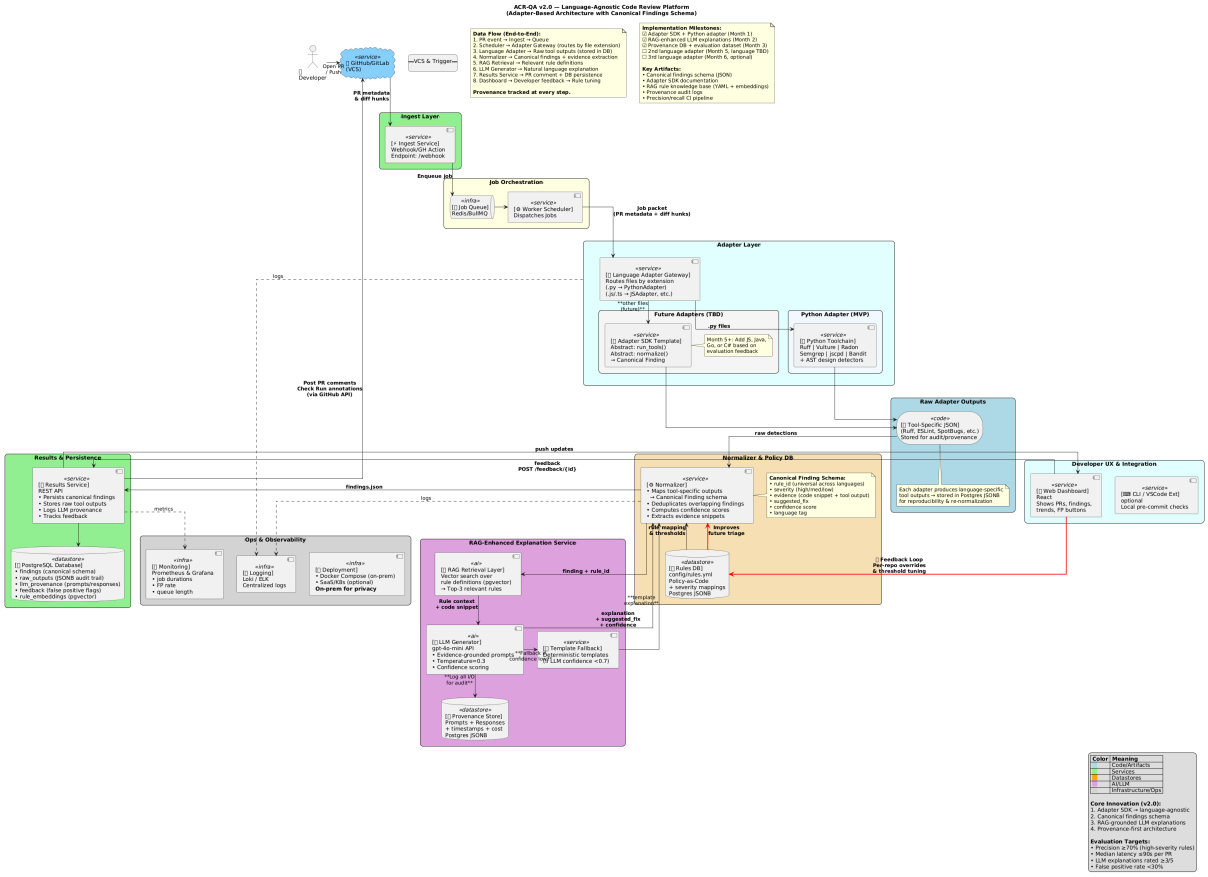

Figure 1: ACR-QA System Architecture showing all components and data flows

manages asynchronous processing enabling parallel analysis. The Worker Scheduler routes jobs to appropriate language adapters based on file extensions.

Language Adapters execute configured tools on changed files and produce standardized JSON results. The Normalizer reads tool outputs and configuration files to map detections to canonical rule IDs, deduplicate overlapping findings, and assign severity scores.

The Explanation Engine performs semantic search over rule definitions, constructs evidence-grounded prompts, queries Cerebras AI API, validates outputs against retrieved context, and falls back to templates when needed. The Results Service persists findings and provenance to PostgreSQL and posts formatted comments to pull requests.

The Web Dashboard provides a React-based interface displaying findings with explanations, allowing developers to mark false positives and view trends. The Feedback Loop collects false positive markings and updates per-repository rule thresholds to improve precision over time.

2.2.3 Development Activities

Requirements Specification (October, Completed): Defined rules.yml catalog structure, canonical findings schema, performance targets (90 second analysis time for typical PRs), and evaluation metrics framework.

Analysis and Design (October-November): Designed adapter interface contract,

database schema for findings and provenance tracking, REST API contracts for all services, RAG retrieval architecture using vector embeddings.

Implementation Phase 1 - Python Adapter (October, Completed): Implemented and validated Python adapter with all tools integrated, producing canonical findings for test repositories.

Implementation Phase 2 - RAG Explanation Engine (November-December): Build rule knowledge base with 50+ definitions, generate vector embeddings for semantic search, integrate Cerebras AI API with evidence-grounded prompting, implement confidence scoring and template fallback logic, create provenance tracking system.

Implementation Phase 3 - Integration (December-January): Develop GitHub/GitLab PR commenting service, build web dashboard with React, implement false positive feedback mechanism, create Docker Compose packaging.

Implementation Phase 4 - Multi-Language Expansion (January-March): Select and implement second language adapter, validate adapter SDK with real implementation, test cross-language analysis on polyglot repositories, extend rule catalog for new language.

Testing and Evaluation (February-April): Create seeded datasets with 80+ labeled issues across supported languages, compute precision/recall metrics per rule category, conduct user study with 8-10 participants, compare AI vs. template explanation quality, tune detection thresholds based on empirical results.

Documentation and Finalization (May-June): Write adapter SDK guide with examples, create deployment documentation, develop API reference documentation, record demonstration video, prepare final evaluation report.

3 Question 3: Project Schedule

3.1 Timeline Overview

The project spans 8 months (October 2025 - June 2026) organized into well-defined phases with clear milestones and built-in contingency planning.

3.2 Realistic Timescales and Progress

The schedule reflects actual progress: Python adapter completed ahead of schedule in October, enabling earlier focus on RAG implementation and explanation quality. Weekly supervisor meetings ensure timely identification of issues and course corrections when needed.

3.3 Project Stages and Milestones

Phase 1: Foundation (October 2025) - COMPLETED

- Python adapter implemented and validated
- Canonical findings schema defined
- Initial rule catalog created
- **Milestone:** Working Python analysis producing structured output

Phase 2: AI Explanation Engine (November-December 2025)

- RAG system with vector embeddings
- Cerebras AI integration with provenance tracking
- Template fallback implementation
- **Milestone:** AI explanations generated with evidence grounding

Phase 3: Integration (December 2025-January 2026)

- GitHub/GitLab PR commenting
- Web dashboard development
- Docker Compose packaging
- **Milestone:** End-to-end PR analysis with automated comments

Phase 4: Evaluation Setup (January-February 2026)

- Seeded dataset creation with labeled issues

- User study protocol design
- Nightly CI for precision/recall tracking
- **Milestone:** Evaluation framework operational

Phase 5: Multi-Language Expansion (February-March 2026)

- Second language adapter implementation
- Cross-language testing
- Rule catalog extension
- **Milestone:** Multi-language platform demonstrated

Phase 6: Evaluation and Tuning (March-April 2026)

- Precision/recall computation
- User study execution (8-10 participants)
- Threshold tuning based on feedback
- **Milestone:** 70% precision achieved, 3/5 user rating

Phase 7: Production Readiness (April-May 2026)

- Performance optimization
- Security hardening
- Monitoring integration (Prometheus/Grafana)
- **Milestone:** System handles 10 concurrent analyses

Phase 8: Documentation and Submission (May-June 2026)

- Complete documentation package
- Demonstration video recording
- Final report writing
- **Milestone:** All deliverables submitted

3.4 Flexibility and Contingency Planning (Plan B)

The schedule incorporates multiple contingency strategies to ensure project success even when facing unexpected challenges.

If RAG Integration Delayed (November-December): Fall back to template-based explanations only which still achieves a functional PR commenting system. RAG can be added in Phase 4 instead as an enhancement rather than core requirement.

If Second Language Adapter Delayed (January-March): Proceed with Python-only platform while focusing on deeper Python rule coverage including advanced security rules and design patterns. This still demonstrates the adapter SDK design conceptually through comprehensive documentation.

If User Study Recruitment Fails: Expand to online participants via GitHub and Reddit developer communities. Alternative approach uses qualitative feedback from supervisor plus 3-5 industry contacts to validate usefulness.

If Evaluation Precision Falls Below 70%: Tune detection thresholds more conservatively, focus on high-confidence rules only, and document limitations with clear improvement path in final report. This maintains academic honesty while still delivering working system.

Built-in Schedule Buffers: June is allocated primarily to finalization rather than critical development, providing natural contingency period. Weekly supervisor check-ins enable early problem detection before issues become blocking. Multi-language expansion is optional enhancement rather than core requirement, meaning Python-only delivery still constitutes complete project.

3.5 Gantt Chart

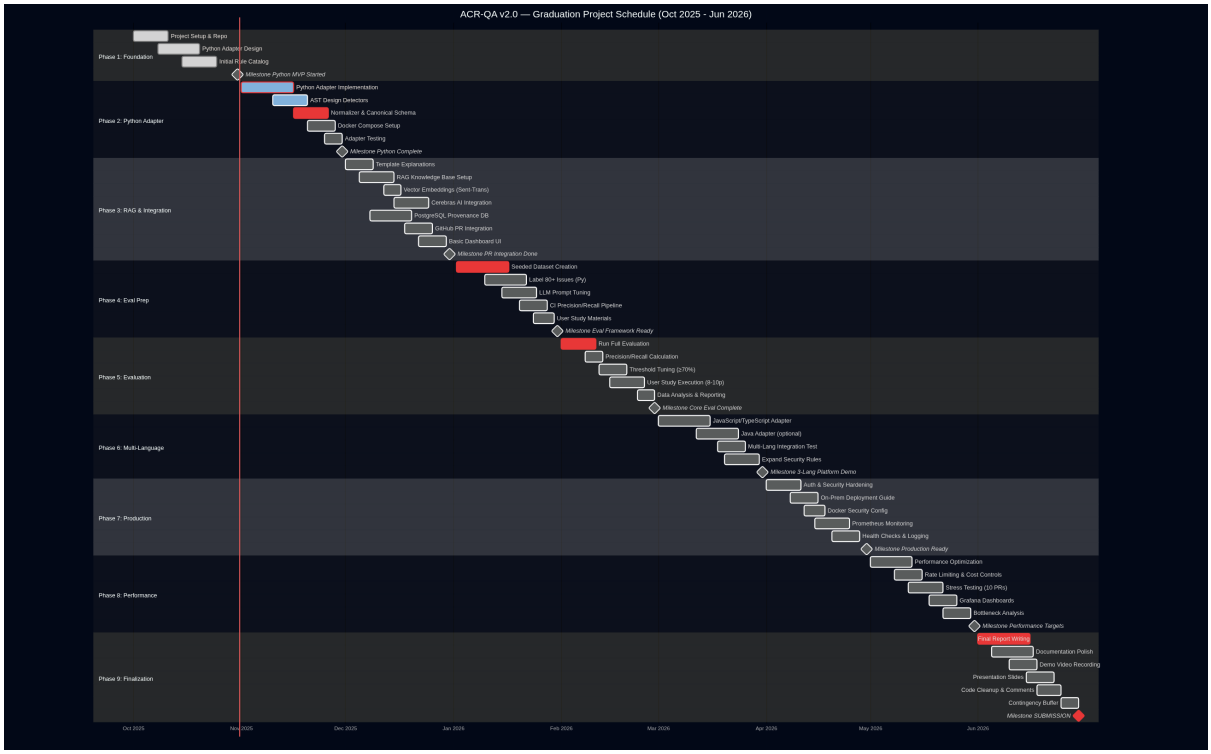

Figure 2: ACR-QA Project Gantt Chart (October 2025 - June 2026)

4 Question 4: Literature Search and Bibliography

4.1 Search Strategy

My literature search employed a systematic three-pronged approach combining official documentation, academic research, and industry best practices.

First, I consulted official tool documentation including Semgrep, Ruff, ESLint, and SpotBugs documentation to understand practical integration requirements, output formats, and tool capabilities. These primary sources provide authoritative information essential for implementation decisions.

Second, I searched academic databases including IEEE Xplore and ACM Digital Library using targeted keywords: "retrieval augmented generation code", "multi-language static analysis", "code review automation", and "code smell detection". I focused on recent papers from 2020-2024 addressing hallucination mitigation and adapter architectures to ensure current best practices.

Third, I reviewed engineering blogs from AI companies and developer tool vendors for practical insights on RAG implementation, prompt engineering patterns, and system architecture decisions. These sources bridge the gap between academic research and production deployment.

This combination ensures both theoretical grounding from peer-reviewed research and practical implementability from real-world engineering experience.

4.2 References

References are formatted in Harvard style as per Question 4 requirements:

1. CustomGPT (2025) *RAG API vs Traditional LLM APIs: Why Developers Choose Context-Aware Solutions*, CustomGPT Engineering Blog. Available at: https://customgpt.ai/rag-api-vs-traditional-llm-apis/ (Accessed: 30 October 2025).

2. IEEE (2023) *Towards Multi-Language Static Code Analysis*, 2023 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 81-88. Available at: https://ieeexplore.ieee.org/document/10301332/ (Accessed: 30 October 2025).

3. Semgrep (2024) *Semgrep Documentation: Lightweight static analysis for many languages*. Available at: https://semgrep.dev/docs/ (Accessed: 20 October 2025).

4. Johnson, B., Song, Y., Murphy-Hill, E. and Bowdidge, R. (2013) *Why don't software developers use static analysis tools to find bugs?*, Proceedings of the 2013 International Conference on Software Engineering, pp. 672-681.

5. Fontana, F.A., Mäntylä, M.V., Zanoni, M. and Marino, A. (2016) *Comparing and experimenting machine learning techniques for code smell detection*, Empirical Software Engineering, 21(3), pp. 1143-1191.

4.3 Document Summary and Evaluation

4.3.1 Document 1: CustomGPT (2025) - RAG API vs Traditional LLM APIs

Summary:

This article presents a comparative analysis of retrieval-augmented generation (RAG) versus direct large language model API calls for developer tools. The central finding demonstrates that RAG architectures reduce hallucinations by 42-68% through grounding model outputs in retrieved contextual information rather than relying solely on parametric knowledge.

The article explains the RAG workflow in four stages: First, rule definitions and documentation are embedded into vector representations during system setup. Second, at inference time, relevant context is retrieved using semantic similarity search. Third, retrieved context is injected into the prompt alongside the user query. Fourth, the language model generates responses constrained by the provided context.

For code review applications specifically, this approach means embedding rule definitions including descriptions, rationale, code examples, and remediation steps in a vector database. When a code issue is detected, the system retrieves the most relevant rule definitions and includes them in the prompt to ground the AI's explanation.

The implementation uses pgvector (PostgreSQL extension) rather than specialized vector databases, reducing infrastructure complexity. The article demonstrates this architecture

with a customer support chatbot where RAG-grounded responses had 68% fewer factual errors compared to direct API calls.

Key technical specifications include: text embeddings using 1536-dimensional vectors, cosine similarity for retrieval, top-3 context injection to limit prompt size, and temperature=0.3 for reduced creativity and higher factual accuracy.

Evaluation of Usefulness:

This document has been instrumental in three critical aspects of my project design and implementation planning.

First, the article's empirical evidence (42-68% hallucination reduction) directly validates my architectural decision to implement RAG rather than using AI models directly. This is essential because code review explanations must be factually accurate to maintain developer trust. The documented improvement provides a concrete target for my own evaluation: I can measure hallucination rates with and without RAG to demonstrate effectiveness in my user study.

Second, the article's recommendation to use pgvector with PostgreSQL solves a major practical problem: how to implement vector search without adding complex infrastructure dependencies. Using pgvector means my on-premises Docker Compose deployment remains self-contained with no external vector database required. This aligns perfectly with my zero-cost requirement and simplifies deployment for target customers including universities and small teams with limited infrastructure expertise.

Third, by demonstrating that RAG adds minimal overhead while dramatically improving output quality, the article helps justify the architecture to stakeholders. Since I'm using free Cerebras AI models, the RAG approach maintains zero recurring costs while substantially improving explanation quality compared to template-only approaches.

The article's empirical methodology—measuring factual error rates through manual evaluation—provides a template for my own user study design. I can ask participants to rate explanation accuracy and correctness, comparing RAG-enabled versus template-based explanations to quantify improvement.

Going forward, I will implement the specific techniques described: temperature=0.3 for model calls to ensure factual accuracy, top-3 context retrieval to balance completeness with prompt size limits, and comprehensive provenance logging to enable post-hoc hallucination detection by comparing generated explanations against retrieved rule definitions.

4.3.2 Document 2: Johnson et al. (2013) - Why don't software developers use static analysis tools?

Summary:

This empirical study investigated why developers often ignore or disable static analysis tools despite their proven ability to find bugs. Through interviews with professional developers and analysis of tool usage patterns, the researchers identified three primary barriers: high false positive rates eroding trust, lack of actionable guidance on how to fix issues, and poor integration with existing workflows.

The study found that developers are willing to tolerate false positives if tools provide

clear explanations of why issues matter and specific remediation guidance. However, most static analysis tools provide only brief error codes or generic messages, forcing developers to research issues separately. This friction leads to tool abandonment.

Evaluation of Usefulness:

This research directly validates my project's focus on explanation quality. The finding that developers need actionable guidance justifies my RAG-enhanced explanation engine that provides not just detection but contextualized explanations with remediation steps. The study's emphasis on false positive management validates my inclusion of a feedback loop where developers can mark false positives to improve precision over time.

5 Question 5: Equipment and Software Requirements

5.1 Development Hardware

Development Machine (Already Available):

- Personal laptop with Intel Core i5 or equivalent (4+ cores)
- 16GB RAM (minimum 8GB)
- 50GB free storage for Docker images, databases, and test repositories
- Stable internet connection for API calls and dependency downloads

No specialized or expensive equipment is required. Standard development hardware is sufficient for running Docker containers, PostgreSQL database, and local testing of all system components.

5.2 Core Development Software

Python 3.11+ (Already Installed): Primary implementation language for platform services, adapters, and custom detectors. Python provides rich ecosystem for web APIs (FastAPI), database interaction (SQLAlchemy), and testing (pytest). The built-in AST module enables custom design smell detectors without external dependencies.

Node.js 18+ (Future Installation): Required for language adapters that use JavaScript-based analysis tools. Many static analysis tools are Node.js-based, making this essential for multi-language support expansion.

Docker and Docker Compose (Already Installed): Non-negotiable requirement for containerization and orchestration of all services. Essential for on-premises deliverable requirement where evaluators and end users must deploy via `docker-compose up`. Ensures consistent environment across development and deployment.

5.3 Infrastructure Components

PostgreSQL 15 with pgvector Extension: Primary database for findings storage, provenance tracking, and vector embeddings. The pgvector extension enables semantic search for RAG without requiring specialized vector databases. Single database for both relational and vector data simplifies architecture and reduces infrastructure complexity.

Redis 7: Job queue for asynchronous analysis tasks. Enables parallel processing of multiple files and non-blocking webhook responses. Industry-standard message broker with proven reliability for CI/CD tools.

Both components will be deployed via Docker Compose as part of the integrated system package.

5.4 Python Analysis Tools

Ruff: Fast Python linter for style violations and anti-patterns. Written in Rust, it runs 10-100× faster than traditional Python linters like Flake8 or Pylint. Replaces multiple tools with a single binary, reducing integration complexity.

Vulture: Specialized tool for detecting unused code and dead imports. Finds unreachable code that general-purpose linters miss through static reachability analysis.

Radon: Calculates cyclomatic complexity and maintainability metrics. Provides objective quantification of code complexity essential for detecting overly complex functions that require refactoring.

Semgrep: Pattern-based security and bug detection with multi-language support. Uses YAML-based rules that are easy to customize. Extensive community rule library covers common security patterns including SQL injection, XSS, and authentication bypass vulnerabilities.

Bandit: Python-specific security analysis tool. Detects Python-specific vulnerabilities including dangerous eval/exec usage, insecure pickle operations, and weak cryptographic functions.

jscpd: Language-agnostic code duplication detection. Works across all future supported languages using token-based similarity matching, eliminating need for language-specific duplication detectors.

All Python tools are already integrated in the completed Python adapter from Phase 1.

5.5 AI and RAG Infrastructure

Cerebras AI API (Zero Cost): Provides free access to high-performance AI models for generating natural language explanations. Eliminates recurring costs while enabling sophisticated explanation generation. System architecture includes template fallback ensuring functionality even when API is unavailable.

Sentence Transformers (Open Source): Generates vector embeddings for rule definitions. Free, open-source alternative to commercial embedding APIs. Can run locally without external dependencies, supporting offline deployment requirement.

Both components will be integrated during November-December RAG implementation phase.

5.6 Frontend and Monitoring

React 18 + TailwindCSS: Modern component framework for web dashboard development. Large ecosystem provides extensive libraries for data visualization and UI components. TailwindCSS enables rapid UI development without custom CSS.

pytest: Industry-standard Python testing framework with powerful fixtures and parametrization. Used throughout development for unit and integration testing.

Prometheus + Grafana: Free and open-source monitoring stack for metrics collection and visualization. Enables tracking system performance, detection accuracy trends, and

user feedback patterns.

5.7 Cost Summary

Total Project Cost: $0

All components are free and open-source with no recurring costs:

- All analysis tools: Free and open-source
- Docker, PostgreSQL, Redis: Free and open-source
- Cerebras AI API: Free access tier
- Embedding generation: Open-source local models
- Development tools: Free (VS Code, Git)
- Testing and monitoring: Free and open-source

This zero-cost approach makes the project accessible to all target customers (universities, small teams, open-source projects) without requiring any budget allocation.

5.8 Technology Selection Justification

Why Python for Platform Services: Dominant language for DevOps and backend automation, aligning with student expertise from previous projects. Rich ecosystem for AI integration including OpenAI SDK and vector libraries. Fast prototyping capability enables meeting aggressive 8-month timeline.

Why PostgreSQL with pgvector: JSONB support efficiently stores canonical findings plus provenance data. pgvector extension enables RAG without separate vector database, reducing infrastructure complexity. Industry-standard for on-premises deployments meeting target customer requirements for self-hosted solutions.

Why Redis + BullMQ: Proven asynchronous job queue pattern used by production CI/CD tools. Lightweight compared to alternatives like Kafka or RabbitMQ while providing necessary functionality. Docker Compose compatible ensuring seamless on-premises deployment.

Why Cerebras AI API: Free tier eliminates recurring costs critical for budget-constrained target customers. High performance enables responsive user experience. API-compatible architecture allows fallback to other providers if needed, avoiding vendor lock-in.

Why Docker Compose Over Kubernetes: On-premises deployment requirement from target customers including small teams and universities. Single-command setup aligns with 30-minute deployment goal stated in aims. Avoids cloud vendor lock-in and complex orchestration overhead.

Why Ruff Over Traditional Linters: 10-100× faster execution written in Rust. Replaces six separate tools (Flake8, isort, Black, etc.) reducing integration complexity and

maintenance burden. Active development with adoption by major projects including Pandas and FastAPI.

Why Semgrep for Multi-Language Analysis: Single rule engine works across Python, JavaScript, Java, and Go. Reduces adapter implementation complexity by using one security tool instead of language-specific alternatives. Open-source with 2000+ community rules eliminating licensing costs.

Why React for Dashboard: Aligns with student's frontend experience from EasyBus Egypt project documented in CV. Component model naturally fits feedback-driven UI including marking false positives and viewing provenance. Large ecosystem for data visualization including Recharts and React Table.

All technology choices prioritize: (1) on-premises deployment capability, (2) zero recurring costs, (3) alignment with student skillset, and (4) proven production-ready tools used by industry.

5.9 What Is NOT Needed

To avoid unnecessary complexity and costs, the following are explicitly excluded:

- Specialized vector databases (Pinecone, Weaviate) — using PostgreSQL with pgvector instead
- Commercial static analysis tools (SonarQube Enterprise, Coverity) — using free open-source alternatives
- Paid API services beyond free tiers — using free Cerebras AI and open-source embedding models
- Cloud hosting infrastructure — Docker Compose enables local deployment
- Specialized hardware (GPU clusters) — standard laptop sufficient for all development and testing
- Kubernetes or complex orchestration — Docker Compose meets all deployment requirements

6 Question 6: Work Plan for Next 2 Weeks

6.1 Week 1 (Days 1-7): Python Adapter Finalization

Concrete Tasks:

1. Complete comprehensive test suite for Python adapter covering all detection categories (style, security, complexity, duplication, design smells) with target of 90%+ test coverage
2. Add edge case handling for unusual Python syntax patterns including decorators, context managers, and async/await constructs
3. Document adapter output format and canonical schema mapping with examples for each rule category
4. Create reference implementation guide for future adapter developers including step-by-step integration instructions
5. Validate adapter performance on 5 diverse open-source Python repositories of varying sizes and complexity

Deliverable: Production-ready Python adapter with 90%+ test coverage, comprehensive documentation, and validated canonical output schema ready for integration with RAG system.

6.2 Week 2 (Days 8-14): RAG System Foundation

Concrete Tasks:

1. Build comprehensive rule knowledge base with 50+ rule definitions including descriptions, rationale, code examples, and remediation steps for all supported detection categories
2. Set up PostgreSQL with pgvector extension in Docker Compose environment and verify vector operations
3. Implement vector embedding generation using Sentence Transformers for all rule definitions with 1536-dimensional vectors
4. Create semantic search module for retrieving relevant rules based on finding context using cosine similarity
5. Develop prompt engineering framework with evidence-grounding templates following best practices from literature review
6. Begin Cerebras AI API integration with initial test prompts and response validation

Deliverable: Operational RAG retrieval system successfully finding relevant rules for test findings, plus initial AI API integration with provenance logging.

6.3 Priority Justification

Completes Critical Foundation: The Python adapter must be production-ready with comprehensive testing before adding more languages. Week 1 finalizes this foundation with proper testing and documentation, ensuring the adapter SDK pattern is proven and replicable for future language additions.

Unlocks Core Innovation: The RAG-enhanced explanation engine is the project's primary technical innovation and main differentiator from existing code review tools. Without RAG implementation, the system would be merely another linter aggregator. Week 2 establishes the RAG infrastructure that will be used throughout all remaining project phases.

Enables Evaluation Work: The evaluation phase scheduled for January-February requires both working detections and explanations. Completing Python detection in Week 1 and RAG foundation in Week 2 means evaluation dataset creation can begin immediately in Week 3, maintaining schedule alignment and avoiding delays in the critical evaluation phase.

De-Risks Technical Uncertainty: RAG integration with AI APIs represents the highest technical risk in the project since it involves external dependencies and novel architecture. Starting this work in Week 2 provides maximum time buffer to iterate, debug, and potentially pivot to template-only explanations if necessary. Early prototyping reveals integration challenges before they become blocking issues that could jeopardize the entire project timeline.

Natural Progression Path: The sequence from Python adapter to RAG system to second language adapter represents logical development progression. RAG must work correctly with Python detections before extending to additional languages, because the explanation engine is designed to be language-agnostic and reuses the same rule retrieval mechanism across all adapters. This validates the architecture before scaling horizontally.

Supervisor Feedback Opportunity: Completing Python adapter finalization and demonstrating initial RAG retrieval within two weeks provides concrete artifacts for the next supervisor meeting. This enables early validation of both the adapter pattern and the RAG approach, ensuring the architectural decisions are sound before significant additional development investment. Early feedback can prevent costly rework later in the project.

Addresses Core Success Criteria: Both deliverables directly address project aims stated in Section 1.2. Week 1 work supports Aim 1 (multi-language platform with adapter architecture). Week 2 work supports Aim 2 (RAG-enhanced AI explanations). Completing these foundational elements early maximizes probability of achieving Aim 3 ($\geq$70% precision) and Aim 4 (successful user study) because they provide the necessary infrastructure for evaluation and tuning.

If Python finalization and RAG foundation succeed within the next two weeks as planned, the project's overall success probability becomes very high. The core technical challenges will be proven feasible, and subsequent work including additional language adapters, dashboard development, and evaluation follows established patterns with significantly lower risk profile.

Declaration of No Plagiarism and AI Technology use

(This page must be signed and submitted with your assignment)

I hereby declare that this submitted TMA work is a result of my own efforts and I have not plagiarized any other person's work. I have provided all references of information that I have used and quoted in my TMA work.

Did you use any AI technology tools? YES No

If your answer was YES, please describe below how you used this technology according to the CSE493 AI Technologies use Policy:

I used AI assistants to help structure and refine this proposal document. Specifically for organizing my ideas into proper academic format, improving technical explanations, proofreading grammar, and helping with LaTeX formatting. The core project concept, architecture decisions, and technical approach are my own work - AI just helped me communicate them clearly.
The AI components are clearly disclosed in the project design and are essential to the value proposition providing explainable, context-aware feedback to developers. The platform also includes template-based fallbacks for environments where AI cannot be used, ensuring full functionality without external APIs.

All other components (Python adapter code, normalizer, database design, Docker configuration, custom AST detectors) will be implemented by me without AI code generation assistance. I'm building the integration layer and orchestration logic myself - the AI models are tools being integrated, similar to how the project integrates PostgreSQL or Redis.

Sudent Name: Ahmed Mahmoud

Abbas

Signature: Ahmed Mahmoud Abbas

Date: 5/11/2025