Product Requirements Document (PRD) ACR-QA v2.1: Language-Agnostic Code Review Platform

Ahmed Mahmoud Abbas
Student ID: 222101213
King Salman International University (KSIU)
Supervisor: Dr. Samy AbdelNabi

January 21, 2026

Document Information

Field	Value
Project Name	ACR-QA v2.1 (Automated Code Review & Quality Assurance)
Owner	Ahmed Mahmoud Abbas (Student ID: 222101213)
Supervisor	Dr. Samy AbdelNabi
Institution	King Salman International University (KSIU)
Timeline	October 2025 - June 2026 (8 months)
Version	2.1 (Platform Version)
Last Updated	January 21, 2026

Contents

1 Executive Summary

1.1 Product Vision

ACR-QA v2.1 is a language-agnostic, on-premises code review platform that automatically detects bad practices, security vulnerabilities, design anti-patterns, and style violations in pull requests. It uses Retrieval-Augmented Generation (RAG) with Cerebras AI to provide evidence-grounded, natural language explanations that help developers understand and fix issues. The MVP focuses on analyzing pull request diffs only, not entire repositories, to keep scope feasible for an 8-month academic project.

1.2 Problem Statement

- **Current Pain:** Code review quality varies by reviewer availability; manual reviews miss security issues; commercial tools cost $10k-50k/year; cloud-based tools can't handle proprietary code
- **Target Users:** University instructors (grading student PRs), small dev teams (5-20 engineers), open-source maintainers, technical recruiters
- **Key Gap:** Existing tools lack explanations or use generic AI that hallucinates incorrect guidance

1.3 Core Innovation (v2.0 Differentiators)

1. **Canonical Findings Schema:** Universal JSON format normalizes outputs from disparate tools (Ruff, Semgrep, ESLint) across languages
2. **RAG-Enhanced Explanations:** Evidence-grounded prompts reduce hallucinations 42-68% vs. direct LLM calls
3. **Provenance-First Architecture:** Stores raw tool outputs, LLM prompts/responses, and user feedback for reproducible evaluation
4. **Adapter SDK:** Pluggable language support (Python first, JavaScript/Java next) via standardized interface

2 Product Objectives & Success Metrics

2.1 Primary Goals

Goal	Metric	Target	Timeline
Multi-Language Platform	Languages supported	Python (100%), +1 language	Nov 2025 / Mar 2026
Detection Quality	Precision (high-severity rules)	≥70%	Feb 2026
	False positive rate	<30%	Feb 2026
AI Explanation Quality	User rating (1-5 scale)	≥3.0 median	Feb 2026
	LLM vs template preference	LLM rated ≥0.5 higher	Feb 2026

Goal	Metric	Target	Timeline
Performance	Analysis latency (PR <200 lines)	≤90 seconds	Jan 2026
Deployment	On-prem setup time	≤30 minutes	Apr 2026
Cost	Total recurring cost	$0 (zero)	Ongoing
PR Review Experience	All findings posted as inline PR comments	100% of detected issues appear as GitHub review comments	Jan 2026

2.2 Academic Requirements

1. Working software system with Docker Compose packaging
2. Evaluation report: precision/recall on 80+ labeled issues
3. User study: 8-10 participants rating explanation usefulness
4. Adapter SDK documentation proving extensibility
5. Demonstration video showing end-to-end workflow

3 User Personas & Use Cases

3.1 Primary Personas

3.1.1 Persona 1: University Instructor (Dr. Sarah)

- **Context:** Teaches Software Engineering to 120 students; receives 300+ PRs/semester
- **Pain:** Can't manually review all PRs; students repeat same mistakes; no time for personalized feedback
- **Jobs-to-be-Done:** Automatically grade PRs for code quality; provide consistent feedback; track student progress
- **Success Criteria:** Reduces review time from 10min/PR to 2min/PR; students understand why code failed

3.1.2 Persona 2: Small Team Tech Lead (Omar)

- **Context:** Leads 8-person dev team at Egyptian startup; can't afford SonarQube Enterprise
- **Pain:** Junior devs push bad code; manual reviews miss security issues; cloud tools violate data policy
- **Jobs-to-be-Done:** Enforce quality gates on PRs; educate juniors via AI explanations; deploy on-prem
- **Success Criteria:** Catches SQL injection before production; costs $0/month; runs on existing server

3.1.3 Persona 3: Open-Source Maintainer (Fatima)

- **Context:** Maintains popular Python library; receives PRs from 100+ external contributors
- **Pain:** Contributors ignore style guide; duplicate code gets merged; explaining issues wastes time
- **Jobs-to-be-Done:** Auto-comment on PRs with guidance; reduce back-and-forth; maintain code quality
- **Success Criteria:** 50% fewer "please fix style" comments; contributors self-correct before re-submission

3.2 User Workflows

3.2.1 Workflow 1: Student Submits PR (Primary)

1. Student opens PR with homework solution
2. GitHub webhook triggers ACR-QA analysis (30-90s)
3. System posts comment: "Found 3 issues: [AI explanations with line numbers]"
4. Student reads explanations, fixes code, pushes update
5. Re-analysis confirms fixes, instructor reviews only logic

3.2.2 Workflow 2: Team Lead Reviews Dashboard

1. Tech lead opens ACR-QA dashboard (React UI)
2. Views trend: "Security findings down 40% this month"
3. Clicks finding: sees code, AI explanation, false positive button
4. Marks 2 findings as FP (overly strict rules for their domain)
5. System adjusts thresholds for future PRs

3.2.3 Workflow 3: Maintainer Configures Rules

1. Maintainer edits `rules.yml` in repo
2. Adds custom rule: "No `requests.get()` without timeout"
3. Commits rule definition with rationale + example
4. Next PR triggers analysis using new rule
5. AI explanation cites the custom rule definition

4 Functional Requirements

4.1 Core Features (MVP - Phase 1, Nov-Jan 2026)

4.1.1 F1: Python Code Analysis

Description: Detect 5 categories of issues in Python code
Categories:

- **Bad Practices:** Mutable defaults, unused variables, dead code
- **Style Violations:** PEP 8 compliance, import ordering, line length
- **Design Smells:** Too many parameters (>5), large classes (>300 lines), high complexity (CC >10)
- **Security Issues:** SQL injection patterns, unsafe eval/exec, weak crypto
- **Code Duplication:** Token-based similarity (>80% match over 50+ tokens)

Tools Used: Ruff, Vulture, Radon, Semgrep, Bandit, jscpd
Input: Python files (.py) from PR diff
Output: Canonical findings with severity, confidence, line numbers. Severity rules: security issues and potential crashes = high, design smells affecting maintainability = medium, stylistic issues and minor formatting problems = low.

4.1.2 F2: Canonical Findings Schema

Description: Universal JSON format normalizing all tool outputs
Schema:

```
{
  "finding_id": "uuid-v4",
  "rule_id": "SOLID-001",
  "category": "design | security | style | duplication | unused",
  "severity": "high | medium | low",
  "confidence": 0.87,
  "file": "app/auth.py",
  "line": 42,
  "column": 10,
  "language": "python",
  "evidence": {
    "snippet": "def authenticate(user, pass, token, session, db):",
    "tool_output": {"radon": {"complexity": 12}},
    "context_before": ["line 39", "line 40", "line 41"],
    "context_after": ["line 43", "line 44", "line 45"]
  },
  "explanation": "AI-generated natural language...",
  "explanation_source": "llm | template",
  "timestamp": "2025-11-23T17:30:00Z"
}
```

Normalizer: Maps Ruff, Semgrep, Vulture, Radon, jscpd → canonical format
Rationale: Enables language-agnostic dashboard, cross-language comparisons, consistent API

4.1.3 F3: RAG-Enhanced AI Explanations

Description: Generate natural language explanations using Cerebras LLM with evidence-grounding
Process:

1. Load rule definition from `rules.yml` (description, rationale, remediation, examples)
2. Retrieve 3-6 lines of code context around issue
3. Construct prompt: "Explain using ONLY this rule definition and code context"

4. Call Cerebras API (llama3.1-8b, temperature=0.3, max tokens=150)

5. Validate: Check if response cites rule id; if not, use template fallback

6. Log prompt + response to provenance DB

Rules Catalog (`rules.yml`):

```yaml
SOLID-001:
  name: "Too Many Parameters"
  category: "design"
  severity: "medium"
  description: "Functions with >5 parameters violate Single
     Responsibility"
  rationale: "Complex signatures indicate function does too much"
  remediation: "Extract parameters into dataclass or config object"
  example_good: |
    @dataclass
    class Config:
      user: str; token: str
    def auth(cfg: Config): ...
  example_bad: |
    def auth(user, pass, token, session, db): ...
```

Cost: ~\$0.0014 per PR analysis (50-200 explanations at \$0.60/1M tokens)
Fallback: Template-based explanation if LLM confidence <0.7 or API fails

4.1.4 F4: GitHub PR Integration

Description: Automatically analyze PRs and post findings as comments
Trigger Options:

- **GitHub Action (Phase 1):** Workflow file in `.github/workflows/`
- **Webhook Endpoint (Phase 2):** Flask/FastAPI server receives PR events
- **Manual trigger:** PR comment `acr-qa review` starts analysis (optional mode for demos)

Flow:

1. PR opened/updated → GitHub triggers action/webhook
2. Fetch PR diff via GitHub API (pygithub library)
3. Extract changed files + line ranges
4. Run analysis on changed code only (not entire repo)
5. Compute severity for each finding (high/medium/low) and sort comments by severity so the most critical issues appear first
6. Post findings as PR review comments with line annotations
7. Store PR metadata + findings in database

Comment Format:

```
**ACR-QA Detected Issue**
**Rule**: SOLID-001 (Too Many Parameters)
**Severity**: Medium
**File**: 'app/auth.py:42'

This function has 5 parameters, which violates the Single
Responsibility Principle. Complex parameter lists indicate
the function is doing too much and becomes hard to test
and maintain.

**Suggested Fix**:
Extract related parameters into a dataclass:

@dataclass
class AuthConfig:
  username: str
  password: str
  token: str

def authenticate(config: AuthConfig, session, db): ...

[Mark as False Positive](#) | [View Details](#)
```

Rate Limiting: Max 1 analysis/PR/minute to avoid spam

4.1.5 F5: Provenance Database

Description: PostgreSQL stores all analysis data for reproducibility and evaluation
Tables:

- **analyses:** PR metadata, timestamp, status, total findings count
- **findings:** Canonical finding objects (see F2 schema)
- **raw_outputs:** Original JSON from each tool (Ruff, Semgrep, etc.)
- **llm_interactions:** Prompts sent, responses received, model, temperature, cost
- **feedback:** User marks (false positive, helpful, unclear)

Stores: Analysis metadata, LLM prompts/responses, cost/latency metrics, rate limit events (for monitoring and debugging)
Enables Observability:

- Full audit trail of all decisions (why was this finding flagged?)
- Cost tracking per PR (Cerebras token count)
- Performance monitoring (latency percentiles, queue depth)
- Rate limit debugging (when did limits kick in?)

Retention: Unlimited (storage ∼85MB for entire project)
Backup: Docker volume persists across container restarts

4.1.6 F6: Findings & Metrics Interface

Description: Provide developers with access to findings and metrics through appropriate interfaces (not a heavy UI, just access).

Features:

Phase 1 (MVP):

- **Terminal UI (Rich library)**
  - Display findings with syntax highlighting
  - Sort by severity (high → low)
  - Show rule ID, file, line, explanation
  - Usage: `acr-qa review <pr-url> --local`
- **Manual Reporting**
  - Console output with findings table
  - Markdown export for GitHub comments
  - JSON export for metrics aggregation

Phase 2 (Optional):

- **REST API endpoint: GET /findings**
  - Returns canonical findings as JSON
  - Enables external tools to consume results
  - No frontend UI required for MVP

Acceptance:

Terminal UI displays 50 findings without lag

Output is readable from 10ft away (font size OK)

JSON export is valid and parseable

4.2 Extended Features (Phase 2, Feb-Jun 2026)

4.2.1 F7: Multi-Language Support

Description: Design a pluggable adapter pattern allowing future language support. Phase 1 focuses on Python; other languages are gated behind performance criteria.

Features:

Phase 1 (MVP):

- **Python Adapter**
  - Ruff, Semgrep, Vulture, Radon, Bandit, jscpd
  - Fully tested and validated
  - Target: 70%+ precision on high-severity rules

Phase 2 (Conditional):

- **JavaScript/TypeScript Adapter**
  - Gate: Only start if Phase 1 Python precision ≥ 80%

- Uses ESLint, similar pattern

Stretch Goal (Lower Priority):

- **Java Adapter** (only if time permits after JS)

Acceptance:

Python adapter achieves ≥70% precision on labeled dataset

Adapter SDK documented (enabling future languages)

Gate enforced: No new languages start until gating criteria met

4.2.2 F8: Evaluation Framework

Seeded Dataset: 80-100 manually labeled issues

- 20 duplications, 20 style, 20 design, 20 security, 20 unused code
- Ground truth: True Positive (TP) or False Positive (FP)

Metrics Calculation (`compute_metrics.py`): Precision $= \text{TP}\frac{}{TP+FPRecall=\frac{TP}{TP+FN}F1Score=\frac{2\times(Precision\times Reca}{Precision+Recall}}$
CI Integration: Nightly runs compute metrics on seeded dataset
Target: Precision ≥70% for high-severity rules

4.2.3 F9: User Study Tools

Comparison Setup: 20 findings with dual explanations (LLM + template)
Google Form: Code snippet, two explanations (randomized order), rating scale
Questions:

1. "Rate Explanation A usefulness (1-5)"
2. "Rate Explanation B usefulness (1-5)"
3. "Which is clearer? A / B / Equal"
4. "Would you follow this guidance? Yes / No"

Target: 8-10 participants, ≥3.0/5.0 median rating

4.2.4 F10: Configuration & Feedback

Description: Allow teams to customize ACR-QA behavior and provide feedback to improve future runs.
Features:
Phase 1 (MVP):

- **False Positive Marking (Backend)**
  - API endpoint: POST /findings/{id}/mark-false-positive
  - Records user feedback in database
  - Enables trend analysis: % of findings marked as FP per rule
- **Provenance & Logging**
  - All analysis decisions logged (why was this rule triggered?)

  - Export: JSON with prompts, responses, timestamps
  - For debugging and user study analysis

Phase 2:

- **Configuration File (.acr-ignore)**
  - Repository owners can ignore specific rules or files
  - Format: Same as .gitignore
  - Example:

```
tests/ STYLE-*
generated/ COMPLEXITY-001
```

  - Enables: Infrastructure-as-Code approach to rule management

- **Metrics Dashboarding**
  - Queries computed over findings database
  - Export: CSV/JSON with precision, recall, FP rate over time
  - Display: Terminal UI or simple HTML report

Acceptance:

FP marking stored in database

Queries can extract: "% of findings marked FP per rule"

Provenance export includes all decision metadata

.acr-ignore file is parsed and honored (Phase 2)

5 Non-Functional Requirements

5.1 Performance

5.2 Scalability

- **MVP Scope:** Single Docker Compose instance (1 worker)
- **Future:** Redis queue enables horizontal worker scaling (out of scope for graduation)
- **Storage Growth:** $\sim$70KB per PR $\times$ 1000 PRs = 70MB (trivial)

5.3 Security & Privacy

- **On-Premises Deployment:** No code leaves customer infrastructure
- **API Keys:** Stored in `.env` file (not in Git); Docker secrets in production
- **LLM Data:** Code snippets sent to Cerebras API (documented in privacy policy)
- **Only minimal code context** (the offending snippet and a few surrounding lines) is sent to the LLM, never entire repositories, to reduce exposure risk
- **Local LLM Option:** Architecture supports offline mode with template explanations only

Metric	Target	Measurement
Analysis Latency	≤90s for PR <200 lines *Scope: Larger PRs may take longer and are out-of-scope for formal evaluation.*	90th percentile
LLM Response Time	≤600ms per explanation	Median
Database Query Time	≤100ms for dashboard load	95th percentile
Concurrent PRs	10 simultaneous analyses	Stress test
Rate Limiting (Concurrency Control)	≤1 analysis/PR/min ≤60 GitHub API/hour ≤50 Cerebras API/hour	Token Bucket Algorithm (Redis) Prevents API quota exhaustion; LLM cost control

Table 2: Performance Requirements

- **Secret Management (Phase 2 Optional):**
  - Phase 1: API keys stored in `.env` (development-safe, documented in `.gitignore`)
  - Phase 2: Optional upgrade to Docker Secrets for production deployments
    - Example: `docker-compose.yml` with `secrets:` section
    - Fallback to `.env` for local development

5.4 Reliability

- **Uptime:** Not applicable (on-prem, no SLA)
- **Error Handling:** Graceful degradation (LLM fails → use template)
- **Data Integrity:** PostgreSQL transactions ensure atomic saves
- **Backup:** Docker volume persists; users responsible for backup strategy

5.5 Usability

- **Setup Time:** 5 minutes from git clone to first analysis (via: `make setup && make up`)
- **One-Click Deploy:** Makefile targets for setup, start, stop, test, clean
  - Commands: `make setup`, `make up`, `make down`, `make test`, `make clean`
- **Local CLI Support (Phase 2 Optional):** `acr-qa scan .` for pre-push validation
- **Documentation:** README, architecture docs, API reference, video tutorial
- **Error Messages:** Plain English (no stack traces to end users)
- **Accessibility:** Terminal UI supports screen readers (basic)

6 Technical Architecture

6.1 System Components

GitHub/GitLab (Code Repository)
↓ Webhook / GitHub Action
Ingest Service (Python) - Receives PR events, extracts diffs
↓ Enqueue job
Redis Job Queue (BullMQ) - Manages async analysis tasks
↓ Dispatch to adapter
Adapter Gateway - Routes files by extension to language adapter
↓
Python Adapter (Ruff, Semgrep, Vulture, Radon, Bandit, jscpd)
Future Adapters (JS, Java, Go) - TBD Phase 2
↓ Tool outputs (JSON)
Normalizer - Maps tool-specific JSON → Canonical Finding Schema
↓ Canonical findings
Rate Limiter (Token Bucket / Redis)
- Enforce: ≤1 analysis per PR per minute
- Enforce: ≤60 GitHub API calls/hour
- Enforce: ≤50 Cerebras API calls/hour
↓
Severity & Prioritization Layer - Assigns severity levels to findings and sorts them so that the most critical issues are surfaced first in GitHub PR comments and dashboard views
↓
RAG Explanation Engine
1. Retrieve rule from rules.yml (semantic search)
2. Construct evidence-grounded prompt
3. Call Cerebras API (llama3.1-8b)
4. Validate output (cites rule_id?)
5. Fallback to template if confidence <0.7
↓ Findings + explanations
Results Service
1. Save to PostgreSQL (findings, raw outputs, LLM logs)
2. Post PR comments via GitHub API
3. Update dashboard data

Figure 1: System Architecture

6.2 Technology Stack

Layer	Technology	Rationale
Language	Python 3.11+	Rich ecosystem, fast prototyping, AST support
Database	PostgreSQL 15+	JSONB for raw outputs, vector search for RAG
Queue	Redis 7 + BullMQ	Async job processing, proven for CI/CD tools
LLM API	Cerebras (llama3.1-8b)	Free tier, 60-70× faster than OpenAI, $0.60/1M tokens

Layer	Technology	Rationale
Containerization	Docker Compose	On-prem deployment, zero-config startup
Static Analysis	Ruff, Semgrep, Vulture, Radon, Bandit, jscpd	Industry-standard tools, broad coverage
Dashboard	Rich (Python terminal UI) + Optional FastAPI	Terminal UI for MVP, REST API for Phase 2 extensibility
VCS Integration	GitHub API (pygithub)	Primary platform, GitLab in Phase 2
Testing	pytest	Standard Python testing framework

6.2.1 Additional Infrastructure (Phase 1 Enhancements)

Layer	Technology	Rationale
Data Validation	Pydantic 2.0+	Runtime schema validation, automatic serialization
Rate Limiting	Redis Token Bucket Algorithm	Handle GitHub/Cerebras limits
Setup & Deployment	Makefile + Shell Scripts	One-click setup, DevOps best practices
Secrets Management	Docker Compose Secrets (Phase 1: .env fallback)	Production-grade key handling (defer to Phase 2 if needed)

6.3 Data Models

Canonical Finding (Core Data Structure):

```
from pydantic import BaseModel, Field, validator
from typing import List, Optional
from datetime import datetime

class Evidence(BaseModel):
    snippet: str = Field(..., description="Code line causing issue")
    tool_output: dict = Field(..., description="Raw JSON from tool")
    context_before: List[str] = Field(..., max_length=3,
                                      description="3 lines before")
    context_after: List[str] = Field(..., max_length=3,
                                     description="3 lines after")

class CanonicalFinding(BaseModel):
    finding_id: str = Field(..., description="UUID")
    rule_id: str = Field(..., description="e.g., UNUSED-001")
    category: str = Field(...,
                          pattern="^(design|security|style|duplication|
                             unused)$")
    severity: str = Field(..., pattern="^(high|medium|low)$")
    confidence: float = Field(..., ge=0.0, le=1.0)
```

```
    file: str = Field(..., description="Relative path")
    line: int = Field(..., ge=1)
    column: int = Field(..., ge=0)
    language: str = Field(default="python")
    evidence: Evidence
    explanation: Optional[str] = None
    explanation_source: Optional[str] = Field(None,
                                              pattern="^(llm|template)
                                                 $")
    timestamp: datetime

    class Config:
        json_schema_extra = {
            "example": {
                "finding_id": "abc-123-def",
                "rule_id": "UNUSED-001",
                "category": "unused_code",
                "severity": "medium",
                "confidence": 0.95,
                "file": "src/main.py",
                "line": 42,
                "column": 1,
                "language": "python",
                "evidence": {
                    "snippet": "import os",
                    "tool_output": {"code": "F401"},
                    "context_before": ["import sys", "import json"],
                    "context_after": ["", "def main():"]
                },
                "explanation": "Import os is never used in this module.
                   ",
                "explanation_source": "llm",
                "timestamp": "2025-01-21T02:47:00Z"
            }
        }
```

Severity is defined as: high = security or bug risk (e.g., injections, unsafe calls, crashes), medium = design and maintainability issues (e.g., long functions, too many parameters), and low = style and cosmetic issues (e.g., formatting, naming). This prioritization is used to order findings in PR comments and reports.

7 Implementation Roadmap

7.1 Phase 1: Foundation (Oct-Jan 2026) - CURRENT

Month	Deliverable	Status
Oct 2025	Python adapter, Docker setup, database schema	Complete
Nov 2025	Canonical schema, normalizer, GitHub Action, evidence-grounded prompts	In Progress
Dec 2025	RAG retrieval, severity scoring, PR comment templates, Pydantic schema validation, Rate limiting (Token Bucket), One-click Makefile, provenance export	Planned

Month	Deliverable	Status
Jan 2026	Seeded dataset, evaluation metrics, user study prep, manual acr-qa review trigger	Planned

7.2 Phase 2: Evaluation & Optimization (Feb-Jun 2026)

Month	Deliverable
Feb 2026	User Study Execution (5-8 participants), Precision/Recall Computation (labeled dataset), Threshold Tuning (optimize for false positive rate), .acr-ignore Support (Configuration as Code)
Mar 2026	JavaScript/TypeScript Adapter (if Python > 80% precision), Advanced Metrics Dashboard (SQL queries + CSV export), Performance Optimization (latency tuning, caching)
Apr 2026	CI/CD Integration Examples (GitHub Actions, GitLab CI), Documentation & Tutorials, Production Deployment Guide (on-prem)
May 2026	Load Testing Report (optional; not required for MVP), Final Hardening & Edge Cases, Demo Video Recording
Jun 2026	Final Report & Thesis Writing, Submission

8 Success Criteria & Acceptance Tests

8.1 MVP Acceptance (End of Phase 1)

8.1.1 Test 1: GitHub PR Integration

- Open test PR with 10 code issues
- System analyzes within 90 seconds
- Posts 10 comments with AI explanations
- Each comment cites a rule id
- Findings are ordered so that high-severity issues appear at the top of the PR review
- Provenance DB logs all LLM interactions

8.1.2 Test 2: Canonical Schema

- Run Ruff, Semgrep, Vulture on sample code
- Normalizer produces findings with universal rule ids
- Database stores findings in canonical format
- Dashboard displays unified view across tools

8.1.3 Test 2b: Rate Limiting & Reliability

- Simulate 10 concurrent PR analysis requests
- Verify ≤1 analysis queued per repo per minute (Token Bucket enforcement)
- Verify Redis connection retry (exponential backoff) if Redis temporarily down
- Verify all jobs eventually process (no stuck jobs)
- Log all rate-limit events for monitoring

8.1.4 Test 3: RAG Explanations

- Load 20 rules from `rules.yml`
- Generate explanations for 20 diverse findings
- 100% of explanations cite correct rule id
- <10% require template fallback

8.1.5 Test 3b: Schema Validation (Pydantic)

- Generate 20 findings with Pydantic CanonicalFinding models
- Verify all findings serialize to valid JSON
- Verify invalid data (e.g., severity="urgent") is rejected with clear error
- Verify schema validation errors logged without crashing system

8.1.6 Test 4: Evaluation

- Label 80 findings as TP/FP
- Compute precision: ≥70% for high-severity
- Compute recall: ≥60% overall
- Document methodology in thesis

8.1.7 Test 5: User Study Validation (Pilot)

Objective: Validate that LLM-generated explanations are more useful than template-based explanations.

Setup:

- Recruit 5–8 participants (friends, colleagues, online volunteers)
- Prepare 10 diverse findings (2 duplication, 2 security, 2 style, 2 design, 2 complexity)
- For each finding: Show both LLM explanation and template version

Procedure:

- Participants rate each explanation 1–5 ("How useful is this?")
- Randomize LLM vs template order (avoid bias)
- Collect via simple Google Form or survey

Success Criteria:

LLM median rating > template median rating (target: LLM 4.0/5, templates 3.0/5; acceptable if trend is consistent)

Statistically significant difference (t-test $p < 0.10$) OR qualitative preference evident in comments

At least 60% of participants prefer LLM explanations

Deliverable:

- Report with: ratings distribution, mean/median, t-test results
- Quotes from participant feedback
- Brief analysis: "Why did LLM score higher?"

Timeline: Feb–Mar 2026 (4 weeks for recruitment + analysis)

9 Risk Management

Risk	Probability	Impact	Mitigation
Cerebras API downtime	Medium	High	Template fallback; store all prompts for replay
GitHub rate limits	Low	Medium	Cache PR diffs; batch comment posts
Low precision (<70%)	Medium	High	Tune thresholds conservatively; focus on high-confidence rules
User study recruitment fails	Medium	Medium	Expand to online (Reddit, GitHub); offer small incentive
Scope creep (too many languages)	High	High	Gate: Python must hit 70% precision before adding JS
Database migration issues	Low	Low	Version schema; test migrations in staging
Enterprise feature creep (e.g., full codebase context engine, advanced analytics)	Medium	High	Limit scope to PR diffs, 1-2 languages, and clearly documented non-goals; defer full-repo context and enterprise features beyond graduation
Pydantic serialization bugs	Low	Medium	Unit test all CanonicalFinding serialization; mock Pydantic validators
Rate limiting not enforcing	Medium	High	Integration test Token Bucket with mock Redis; verify queue behavior under load
Docker Secrets setup complexity	Low	Medium	Document .env as Phase 1; defer Docker Secrets to Phase 2

10 Open Questions & Decisions Needed

10.1 Immediate (Week 1)

- **GitHub Action vs Webhook?** → Recommendation: Action first (simpler)
- **Rules.yml structure?** → Recommendation: YAML (version controlled), DB later
- **Template fallback format?** → Recommendation: Jinja2 templates with same structure as LLM output
- **Pydantic vs Dataclasses?** → Recommendation: Pydantic (industry standard, automatic validation + serialization)
- **Rate Limiting Algorithm?** → Recommendation: Token Bucket in Redis (proven, handles concurrent requests)
- **Setup Tool?** → Recommendation: Makefile (simple, portable, DevOps standard)

10.2 Phase 2 (Jan-Mar)

- **Second language: JS or Java?** → Depends on user study feedback
- **Self-hosted LLM option?** → Optional; Ollama + Llama 3.1 8B documented
- **GitLab support priority?** → Low unless user requests
- **Should future versions add full-repository context analysis, or remain PR-diff focused to keep complexity and costs low?**

11 Appendices

11.1 Glossary

Canonical Finding Normalized detection result in universal JSON schema

RAG (Retrieval-Augmented Generation) LLM technique that injects retrieved context into prompts

Provenance Complete audit trail of analysis (tool outputs, prompts, responses)

Adapter Language-specific module that runs tools and normalizes outputs

False Positive (FP) Detection flagged as issue but is actually correct code

True Positive (TP) Detection correctly identifies a real code issue

11.2 References

1. CustomGPT (2025) "RAG API vs Traditional LLM APIs" - 42-68% hallucination reduction
2. IEEE (2023) "Towards Multi-Language Static Code Analysis" - Adapter pattern validation
3. Semgrep Documentation - Pattern-based rule engine design
4. Johnson et al. (2013) "Why don't developers use static analysis?" - User study methodology

5. Pydantic Documentation (2025) - "Data Validation with Python"
6. Redis Token Bucket Pattern - Rate limiting at scale
7. IEEE (2024) - "DevOps Best Practices for Python Services"

11.3 Document Change Log

Date	**Version**	**Changes**	**Author**
Nov 23, 2025	1.0	Initial PRD creation	Ahmed Abbas
Jan 21, 2026	2.1	Enhanced Phase 1 with Pydantic, rate limiting, Makefile, Docker Secrets; clarified Phase 2 scope (JS + CLI optional); relaxed user study acceptance criteria	Ahmed Abbas

End of Product Requirements Document