Product Requirements Document (PRD)
ACR-QA v2.4: Language-Agnostic Code Review Platform

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

January 28, 2026

Document Information

Field	Value
Project Name	ACR-QA v2.4 (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	2.4 (Platform Version)
Last Updated	January 28, 2026

Contents

1 Executive Summary

1.1 Product Vision

ACR-QA v2.4 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. The system is packaged as a production-ready, Docker-based pipeline with Prometheus metrics, Grafana dashboards, k6 load testing validation, and Pydantic schema validation, aligning with modern backend/DevOps observability and reliability practices.

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

Goal	Metric	Target	Timeline
AI Explanation Quality	User rating (1-5 scale)	≥3.0 median	Feb 2026
	LLM vs template preference	LLM rated ≥0.5 higher	Feb 2026
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 Metrics & Marks FP

1. Tech lead runs `acr-qa review <pr-url>` to view findings (Terminal UI with Rich library)
2. OR (Phase 2): Opens Prometheus/Grafana dashboard in browser to view trends ("Security findings down 40% this month")
3. Marks 2 findings as false positive (API endpoint: POST /findings/{id}/mark-false-positive)
4. System records feedback in database
5. System adjusts thresholds for future PRs based on FP feedback

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 10 categories of issues in Python code with 32 rules

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. Explanations include `[RULE-ID](config/rules.yml)` citations and "How to Fix" remediation.

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. Also supports GitLab CI/CD with MR comments via `.gitlab-ci.yml`.

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). Includes OWASP Top 10 and SANS Top 25 compliance scoring.

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 $\geq$ 80%
  - Uses ESLint, similar pattern

  Stretch Goal (Lower Priority):

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

  Acceptance:

  Python adapter achieves $\geq$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`): $\text{Precision} = \text{TP}$ $\frac{}{TP+FP} \text{Recall} = \frac{TP}{TP+FN}$ $F1 Score = \frac{2 \times (Precision \times Reca}{Precision + Recall}$

  CI Integration: Nightly runs compute metrics on seeded dataset
  Target: Precision $\geq$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, $\geq$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)

4.2.5 F11: Observability & Monitoring Stack

Description: Production-grade monitoring infrastructure enabling operators to understand system behavior, identify bottlenecks, and troubleshoot failures under load. Provides visibility into analysis latency, error rates, LLM token usage, and rule performance over time.

Components:

A. Prometheus Metrics Exporter (30 hrs)

- Expose `/metrics` endpoint (prometheus client library)
- Track metrics per analysis:
  - `analysis_latency_ms` (histogram, by severity level)
  - `llm_token_count` (counter, cumulative)
  - `error_rate_percent` (gauge)
  - `false_positive_rate_percent` (gauge, per rule_id)

  - `redis_queue_depth` (gauge, jobs waiting)
  - `cerebras_api_latency_ms` (histogram)

- Prometheus scrapes every 15 seconds
- Metrics retention: 30 days (configurable)

Example metrics output:

```
# HELP analysis_latency_ms Analysis time in milliseconds
# TYPE analysis_latency_ms histogram
analysis_latency_ms_bucket{severity="high",le="30"} 5
analysis_latency_ms_bucket{severity="high",le="60"} 18
analysis_latency_ms_bucket{severity="high",le="90"} 22
analysis_latency_ms_count{severity="high"} 22
analysis_latency_ms_sum{severity="high"} 1204

llm_token_count_total 45620
error_rate_percent 0.5
false_positive_rate_by_rule{rule_id="UNUSED-001"} 25.0
redis_queue_depth 3
```

B. Grafana Dashboard (35 hrs)

- Install Grafana in docker-compose
- Create 5 key panels:

Panel 1: "Analysis Latency by Severity"

  - Line chart: X-axis = time, Y-axis = latency (ms)
  - 3 lines: high-severity, medium-severity, low-severity
  - SLA threshold: 90s (red line)
  - Queries: `analysis_latency_ms` by severity
  - Shows trend over hours/days
  - Alert: if latency $>$ 120s for 5 mins $\rightarrow$ red

Panel 2: "LLM Token Cost Trend"

  - Area chart: cumulative token count over time
  - Cost calculation: tokens / 1M $\times$ \$0.60
  - Daily cost label (e.g., "Today: \$0.47")
  - Query: `rate(llm_token_count[1d])`

Panel 3: "System Error Rate"

  - Gauge (0-100%): Current error %
  - Red zone: >10%, Yellow zone: 5-10%, Green: <5%
  - Query: `error_rate_percent`

Panel 4: "False Positive Rate per Rule"

  - Bar chart: Each bar = rule_id, height = FP %
  - Sort descending (highest FP first)

  - Threshold line at 30% (acceptable)
  - Query: `false_positive_rate_by_rule`
  - Identify which rules need threshold tuning

Panel 5: "Redis Queue Depth"

  - Line chart: jobs waiting in queue
  - Alert: if depth > 10 jobs, queue is backed up
  - Shows when system is overwhelmed
  - Query: `redis_queue_depth`

- Dashboard JSON export (for sharing/versioning)
- Annotations: Mark major events (e.g., "Precision tuning applied")
- Time range: 24h/7d/30d selectable

C. Performance Baseline Documentation

- Measure before load testing:
  - p50 analysis latency: 45ms
  - p95 analysis latency: 75ms
  - p99 analysis latency: 120ms
  - Mean error rate: <1%
  - Mean FP rate: 28%
- Document in `PERFORMANCE_BASELINE.md`

Acceptance:

`/metrics` endpoint returns Prometheus-formatted output

Grafana connects to Prometheus (data visible in all 5 panels)

Dashboard saves & loads correctly (JSON export works)

Alert thresholds trigger as expected

Historical data persists across container restarts

4.2.6 F12: Load Testing & Stress Analysis

Description: Validate system reliability and identify performance bottlenecks under realistic and extreme load. Prove system can handle concurrent PR analyses without degradation or failure. Document scaling constraints and recommendations.

Components:

A. k6 Load Testing Suite (25 hrs)

Test Framework: k6 (https://k6.io/)

- Open-source, cloud-ready
- JavaScript-based test scripts
- Built-in metrics: latency, throughput, error rate

- HTML report generation

Scenario 1: "Ramp-up Test" (Gradual Load)
Goal: Measure latency as load increases
Load profile:

- 0 min: 0 PRs/min
- 1 min: 5 PRs/min
- 2 min: 10 PRs/min
- 3 min: 20 PRs/min
- 4 min: 50 PRs/min (peak)
- 5 min: Hold at 50 PRs/min for 2 min
- 6 min: Ramp down to 0

Success Criteria:

p95 latency stays <120s throughout

Error rate <5% at peak

No system crashes/OOM errors

Redis queue handles all jobs (none stuck)

Metrics captured:

- Latency distribution (p50, p95, p99)
- Throughput (requests/sec)
- Error rate %
- Queue depth over time

Export: HTML report with graphs
Scenario 2: "Steady State Test" (Sustained Load)
Goal: Ensure system stability under constant stress
Load profile:

- 0 min: Ramp to 20 PRs/min over 2 mins
- 2 min: Hold at 20 PRs/min for 10 mins
- 12 min: Ramp down

Success Criteria:

Latency stable (no degradation over 10 mins)

Error rate <3%

Memory doesn't leak (Docker memory usage flat)

Responses arrive in expected time

Metrics: Same as ramp-up

Scenario 3: "Spike Test" (Sudden Traffic Burst)

Goal: Measure recovery from traffic spikes

Load profile:

- 0 min: Baseline 5 PRs/min
- 1 min: Spike to 100 PRs/min for 30 secs
- 1.5 min: Return to 5 PRs/min

Success Criteria:

No errors during spike

Latency spikes but recovers

Queue depth returns to normal within 1 min

No data loss

Metrics: Peak latency, recovery time

k6 Script Structure (pseudo-code):

```
import http from 'k6/http';
import { sleep, group } from 'k6';

export let options = {
  stages: [
    { duration: '1m', target: 5 },    // Ramp-up
    { duration: '2m', target: 50 },
    { duration: '2m', target: 50 },   // Hold
    { duration: '1m', target: 0 },    // Ramp-down
  ],
  thresholds: {
    http_req_duration: ['p(95)<120000'],  // p95 latency < 120s
    http_req_failed: ['rate<0.05'],       // error rate < 5%
  },
};

export default function() {
  group('Analyze PR', () => {
    let res = http.post(
      'http://localhost:8000/analyze',
      {
        pr_url: 'https://github.com/...',
        repo_name: 'test-repo',
      }
    );
    sleep(1);
  });
}
```

B. Stress Test Manual Report (10 hrs)

- Document findings in `STRESS_TEST_REPORT.md`
- Include:
  - Test scenarios run (dates, times)

  - Hardware specs (CPU, RAM, Docker limits)
  - Raw latency data (p50, p95, p99)
  - Error logs (if any failures)
  - Peak throughput achieved
  - Bottleneck identification
    - Example: "At 60 concurrent PRs, Redis queue maxes out. Recommendation: increase Redis memory or scale horizontally."
  - Recommendations for production scaling
    - Example: "For 10x growth, recommend Kubernetes with 3 worker pods"

C. Performance Regression Testing

- CI/CD integration: Run k6 tests on every major commit
- Compare against baseline (`PERFORMANCE_BASELINE.md`)
- Fail CI if latency regresses >20% or error rate >5%
- GitHub Actions job: `tests/ci_load_test.yml`

Acceptance:

k6 script executes without errors

Ramp-up test: p95 latency <120s, error rate <5%

Steady state test: latency stable over 10 mins, <3% errors

Spike test: recovery time <1 min

HTML report generated with graphs

Stress test report identifies 1+ optimization opportunity

CI fails gracefully if thresholds exceeded

4.2.7 F13: Production Readiness & Quality Assurance

Description: Implement production-grade reliability, testability, and operational excellence practices to ensure the system is production-ready and maintainable. These features focus on depth of implementation rather than breadth of new capabilities.

Components:

A. Circuit Breaker Pattern for API Resilience (10 hrs)

Objective: Prevent cascading failures when external APIs (Cerebras, GitHub) fail.

Implementation:

- Monitor Cerebras API response times and error rates
- States: CLOSED (normal) → OPEN (too many failures) → HALF_OPEN (testing recovery)
- When OPEN: immediately fall back to template explanations (no retry delay)
- When HALF_OPEN: allow 1 probe request to check if API recovered
- Thresholds:
  - CLOSE→OPEN: 5 failures in 10 seconds OR latency >5 seconds for 3+ requests

- OPEN→HALF_OPEN: 30 second timeout
- HALF_OPEN→CLOSED: first probe succeeds
- HALF_OPEN→OPEN: probe fails

Code example:

```
from circuitbreaker import circuit

@circuit(failure_threshold=5, recovery_timeout=30)
def call_cerebras_api(prompt):
    response = cerebras_api.explain(prompt)
    return response

# Usage:
try:
    explanation = call_cerebras_api(prompt)
except CircuitBreakerListener as e:
    explanation = template_fallback(prompt)  # Fast fallback
```

Monitoring:

- Log every state transition (CLOSED→OPEN, etc.) to PostgreSQL
- Prometheus metric: `circuit_breaker_state{service="cerebras"}`
- Alert if circuit breaker opens >3 times/day

Acceptance:

Circuit breaker transitions correctly through all states

Fallback triggered immediately when circuit opens (no retry delay)

Recovery successful after 30s timeout + probe succeeds

Prometheus metric updates in real-time

B. Response Caching Layer (8 hrs)
Objective: Reduce LLM API calls and latency by caching identical findings.
Implementation:

- Cache key: hash(rule_id + code_snippet)
- Cache storage: Redis
- TTL: 24 hours per rule (configurable)
- Cache hit types:
  1. Identical rule + code → return cached explanation
  2. Same rule, similar code (>85% match) → return cached, mark as variant

Code example:

```
import hashlib
import redis

cache = redis.Redis(host='localhost', port=6379)

def get_explanation(rule_id, code_snippet):
```

```
    cache_key = f"explain:{rule_id}:{hashlib.md5(code_snippet.encode())
        .hexdigest()}"

    # Check cache
    if cached := cache.get(cache_key):
        return json.loads(cached)

    # Cache miss: call LLM
    explanation = cerebras_api.explain(prompt)

    # Store in cache (24h TTL)
    cache.setex(cache_key, 86400, json.dumps(explanation))
    return explanation
```

Monitoring:

- Prometheus metric: `explanation_cache_hit_ratio`
- Log cache stats daily: hit rate, miss rate, size
- Alert if hit rate <30% (misconfigured)

Acceptance:

Identical findings return cached explanation (latency <10ms)

Cache size stays <100MB (bounded by eviction policy)

Hit rate >60% on seeded dataset (typical)

Cache invalidation works (old entries expire after 24h)

C. Structured Logging for Production (5 hrs)
Objective: Replace print() statements with JSON-formatted logs for machine parsing.
Implementation:

- Library: structlog
- Output: JSON to stdout (Docker logs), optionally to file
- Every log includes: timestamp, level, event_name, actor, metadata

Code example:

```
import structlog

logger = structlog.get_logger()

# Phase 1: Analysis starts
logger.info("analysis_started",
    pr_id="123",
    repo="myapp",
    files_changed=5,
    timestamp=datetime.now().isoformat()
)

# Phase 2: Tool execution
logger.info("tool_executed",
    tool="ruff",
    latency_ms=240,
```

```
    findings=12,
    status="success"
)

# Phase 3: LLM explanation
logger.info("explanation_generated",
    rule_id="SOLID-001",
    source="llm",
    latency_ms=350,
    tokens_used=42,
    cost_usd=0.000025
)

# Error: API failure with circuit breaker
logger.error("api_call_failed",
    service="cerebras",
    status_code=503,
    retry_attempt=2,
    fallback="template"
)
```

Monitoring:

- Prometheus metric: `log_events_total{level, event_name}`
- Query logs with grep: `cat docker.log | jq 'select(.event_name=="api_call_failed")'`
- ELK integration ready (Phase 3): parse JSON logs to Elasticsearch

Acceptance:

All log outputs are valid JSON

Log includes: timestamp, level, event_name, actor, context

No print() statements in codebase

Log level configurable via env var (DEBUG, INFO, WARNING, ERROR)

D. Database Connection Pooling (5 hrs)
Objective: Reuse database connections efficiently, prevent connection exhaustion.
Implementation:

- Library: SQLAlchemy QueuePool
- Pool size: 10 connections (normal traffic)
- Max overflow: 20 extra connections (burst traffic)
- Connection recycle: 3600s (1 hour) to handle DB restarts
- Timeout: 30s to get connection from pool

Code example:

```
from sqlalchemy import create_engine
from sqlalchemy.pool import QueuePool

engine = create_engine(
    DATABASE_URL,
    poolclass=QueuePool,
```

```
    pool_size=10,           # Base pool
    max_overflow=20,        # Burst capacity
    pool_recycle=3600,      # Recycle every hour
    pool_pre_ping=True,     # Check connection alive before use
    echo=False              # Disable query logging
)

# Usage: sessions automatically borrow/return from pool
with Session(engine) as session:
    findings = session.query(Finding).filter(...).all()
```

Monitoring:

- Prometheus metric: `db_pool_size`, `db_pool_checked_in`, `db_pool_overflow_count`
- Alert if checked_out > 15 (pool saturated)
- Log pool exhaustion events

Acceptance:

No "too many connections" errors under load (k6 test)

Pool size stays within limits (debug logs)

Connection reuse >80% (not creating new ones each time)

Latency stable under sustained load (no connection wait times)

E. Comprehensive Unit Test Suite (15 hrs)
Objective: Maintain 85%+ code coverage with focused, fast tests.
Test categories:
Schema Validation (Pydantic) - 10 tests

```
def test_canonical_finding_valid():
    finding = CanonicalFinding(
        finding_id="abc-123",
        rule_id="SOLID-001",
        category="design",
        severity="medium",
        confidence=0.87,
        # ... full schema
    )
    assert finding.severity in ["high", "medium", "low"]
    assert 0 <= finding.confidence <= 1
    assert finding.serialize()  # Can convert to JSON

def test_canonical_finding_invalid_severity():
    with pytest.raises(ValidationError):
        CanonicalFinding(..., severity="urgent")  # Invalid
```

Rate Limiting (Token Bucket) - 8 tests

```
def test_token_bucket_allows_under_limit():
    limiter = TokenBucket(rate=1, capacity=5)
    assert limiter.allow_request() == True

def test_token_bucket_blocks_over_limit():
    limiter = TokenBucket(rate=1, capacity=1)
    limiter.allow_request()
    assert limiter.allow_request() == False
```

Normalizer (Tool Output Mapping) - 12 tests
Circuit Breaker - 6 tests
RAG Fallback - 8 tests
Integration Tests - 10 tests
Configuration:

- pytest.ini: min 85% coverage, fail if lower
- GitHub Actions: run tests on every commit
- Run time: <30 seconds (fast feedback)

Acceptance:

85%+ code coverage

All tests pass on CI/CD

Test runtime <30 seconds

Every critical path tested

F. Error Handling & Graceful Degradation (8 hrs)
Objective: Document and test all failure modes. System should never crash; always fall back.
Failure modes:

Failure	Impact	Fallback
Cerebras API timeout	Can't generate LLM explanation	Use template explanation
Cerebras API rate limit	Can't generate explanations	Queue job, retry later
GitHub API rate limit	Can't post comments	Queue findings, post when quota refills
Redis down	Can't rate limit	In-memory token bucket
PostgreSQL down	Can't store findings	Log to file, retry on reconnect
Semgrep crash	Can't run static analysis	Skip tool, continue with others

Documentation: Create `FAILURE_MODES.md` with all failure scenarios, impacts, and recovery procedures.
Acceptance:

Every failure mode documented

Every failure handled with fallback (no crashes)

Logging captures root cause

System recovers automatically or alerts operator

G. System Profiling Report (10 hrs)
Objective: Identify performance bottlenecks with data.
Create `PERFORMANCE_PROFILE.md` documenting:

- Top 10 functions by execution time
- Memory usage by component

- Optimization opportunities
- Recommendations for scaling

Acceptance:

Profile identifies top 3 bottlenecks

Memory usage <100MB

Recommendations documented

Profile can be re-run after optimization

H. Deployment & Operations Runbook (5 hrs)
Objective: Clear step-by-step guide for deploying to production.
Create `DEPLOYMENT.md` covering:

- Pre-deployment checklist
- Single-machine deployment steps
- Scaling to multiple workers (optional)
- Rollback procedures
- Post-deployment verification
- Support runbook for common issues

Acceptance:

Runbook is step-by-step (no assumptions)

Covers rollback procedures

Includes post-deployment verification

New operator can deploy without help

I. Cost Analysis & Scalability Report (8 hrs)
Objective: Quantify expenses and scaling limits.
Create `COST_ANALYSIS.md` documenting:

- Monthly cost breakdown at current scale (1000 PRs/month)
- Scaling limits (concurrent PRs, throughput, database connections)
- Cost projections at 10x scale
- Recommendations for horizontal scaling

Key findings:

- Current monthly cost: $0.03 (Cerebras API only)
- At 10x scale: $250-1400/month (with infrastructure)
- Bottleneck: 10 concurrent PRs (single container)
- Scaling solution: Kubernetes with 3 worker pods

Acceptance:

Cost broken down by component

Scaling limits identified

Remedies proposed

Scalability roadmap clear

J. Security Audit Report (8 hrs)
Objective: Document security practices and compliance.
Create `SECURITY_AUDIT.md` covering:

- Data protection (API keys, code confidentiality, database security)
- Input validation (Pydantic, SQL injection prevention)
- Rate limiting (prevents DoS attacks)
- Access control (authentication, authorization)
- Audit trail (complete logging for compliance)
- Risk assessment with mitigation strategies

Key practices:

- No API keys in Git (enforced by .gitignore)
- Minimal code context sent to LLM (3-6 lines only)
- All inputs validated via Pydantic schemas
- Rate limiting prevents abuse
- Complete audit trail for compliance

Acceptance:

Security practices documented

Risks identified & mitigated

Compliance roadmap clear

Enterprise concerns addressed

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:** ∼70KB per PR × 1000 PRs = 70MB (trivial)

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

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
- **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
Caching	Redis	Response caching with 7-day TTL for explanation reuse
LLM API	Cerebras (llama3.1-8b)	Free tier, 60-70× faster than OpenAI, $0.60/1M tokens
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
Monitoring	Prometheus + Grafana	Metrics collection + visualization (Phase 2, 30-35 hrs)
Load Testing	k6 (Grafana k6)	Cloud-ready load testing framework (Phase 2, 25 hrs)

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):

```python
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
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 Execution (k6 ramp-up/steady/spike scenarios), Stress Test Report & Analysis, Performance Regression CI/CD Integration, Circuit Breaker Implementation, Caching Layer, Comprehensive Unit Tests (85%+ coverage), Profiling Analysis, Error Handling Documentation, Final Hardening & Edge Cases, Demo Video Recording (showing Grafana dashboard + Prometheus metrics)
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
- **Test Coverage:** 45 unit tests across 6 test suites with pytest-cov achieving 70% threshold

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 $\leq$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)

8.1.8 Test 6: Observability Stack

- Prometheus `/metrics` endpoint returns valid metrics
- Grafana connects to Prometheus (all panels load data)
- All 5 dashboard panels display correctly
- Alert thresholds trigger as expected (test: latency spike → red alert)
- Historical metrics persist across container restart

8.1.9 Test 7: Load Testing

- k6 ramp-up scenario completes: p95 latency <120s, error rate <5%
- k6 steady state (10 mins): latency stable, <3% errors
- k6 spike test: recovery within 1 min
- HTML report generated with graphs
- Stress test report identifies ≥1 bottleneck + recommendation

8.1.10 Test 8: Production Readiness (Quality Assurance)

A. Circuit Breaker

Circuit breaker transitions correctly: CLOSED → OPEN → HALF_OPEN → CLOSED

When OPEN, fallback to template (no retry delay)

Recovery successful after 30s timeout + probe succeeds

Prometheus metric updates: `circuit_breaker_state{service="cerebras"}`

B. Caching Layer

Identical findings return cached explanation (latency <10ms)

Cache hit rate >60% on seeded dataset

Cache size stays <100MB

Old entries expire after 24h TTL

C. Structured Logging

All logs are valid JSON format

Every log includes: timestamp, level, event_name, actor, context

No print() statements in codebase

Log level configurable via env var

D. Database Connection Pooling

No "too many connections" errors under load (k6 stress test)

Pool size stays within limits (debug logs verify)

Connection reuse >80%

Latency stable under sustained load

E. Unit Tests

Code coverage ≥85% (pytest report)

All tests pass on CI/CD

Runtime <30 seconds

All critical paths tested

F. Error Handling & Graceful Degradation

Every failure mode documented (`FAILURE_MODES.md`)

No crashes: always has fallback

Root cause logged for debugging

System recovers automatically or alerts operator

G. Performance Profiling

Profiling identifies top 3 bottlenecks

Memory usage <100MB

Optimization recommendations documented

Profile can be re-run after changes

H. Deployment Runbook

`DEPLOYMENT.md` is step-by-step (no assumptions)

Covers rollback procedures

Post-deployment verification checklist

New operator can deploy independently

I. Cost & Scalability Analysis

Monthly cost broken down by component

Scaling limits identified (10 concurrent → 50 with Kubernetes)

Cost at 10x volume estimated

Scaling roadmap clear

J. Security Audit

API keys protected (never in Git)

Code snippets minimized (3-6 lines only)

Input validation via Pydantic (no SQL injection)

Rate limiting prevents DoS

Audit trail complete (timestamps, sources)

Enterprise recommendations documented

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
Prometheus scraping fails	Low	Medium	Test /metrics endpoint manually; add retry logic
k6 test flakiness (network)	Medium	Low	Run tests 3x, take median; use local test endpoint
Grafana dashboard too complex	Low	Medium	Start with 3 panels, add 2 more if time
Load testing reveals blocker	Medium	High	Mitigation: Fix top bottleneck, retest, document findings
Circuit breaker misconfigured	Low	Medium	Unit test state transitions; monitor state changes
Cache coherency issues	Low	Low	TTL-based invalidation (24h); rebuild on failure
Logging overhead (JSON serialization)	Low	Low	Async logging; benchmark <5% CPU overhead
Connection pool exhaustion	Medium	High	Monitor pool size; alert if checked_out>15
Unit test maintenance burden	Medium	Low	Use pytest fixtures; keep tests focused

Risk	Probability	Impact	Mitigation
Profiling runtime overhead	Low	Low	Profile in dev only; disable in prod
Deployment mistakes	Medium	High	Runbook checklist; dry-run on staging first
Cost analysis becomes outdated	Low	Low	Re-analyze quarterly; alert on cost spike

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"
8. Prometheus Documentation (2025) - "Metrics and Alerting"
9. Grafana Documentation (2025) - "Dashboard Best Practices"
10. k6 Documentation (2025) - "Load Testing with k6"
11. SRE Handbook - "Performance Testing & Scaling"

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
Jan 24, 2026	2.2	Added Observability Stack (F11) and Load Testing (F12) to Phase 2 for production-grade monitoring and performance validation	Ahmed Abbas
Jan 24, 2026	2.3	Added Production Readiness & Quality Assurance (F13): Circuit Breaker, Caching, Structured Logging, Connection Pooling, 85%+ Unit Tests, Error Handling, Profiling, Deployment Runbook, Cost Analysis, Security Audit. Updated Product Vision with DevOps angle.	Ahmed Abbas

Date	Version	Changes	Author
Jan 28, 2026	2.4	Phase 1 Completion Updates: Added GitLab CI/CD support, Added response caching with Redis (7-day TTL), Added OWASP/SANS compliance reporting, Expanded knowledge base to 32 rules across 10 categories, Added 45 unit tests with 70% coverage threshold, Added source citations and remediation in reports	Ahmed Abbas

End of Product Requirements Document