Technical Challenge - Code Review and Deployment Pipeline Orchestration

Format: Structured interview with whiteboarding/documentation
Assessment Focus: Problem decomposition, AI prompting strategy, system design

Please Fill in your Responses in the Response markdown boxes

Challenge Scenario

You are tasked with creating an AI-powered system that can handle the complete lifecycle of code review and deployment pipeline management for a mid-size software company. The system needs to:

Current Pain Points:

- Manual code reviews take 2-3 days per PR
- Inconsistent review quality across teams
- Deployment failures due to missed edge cases
- Security vulnerabilities slip through reviews
- No standardized deployment process across projects
- Rollback decisions are manual and slow

Business Requirements:

- Reduce review time to <4 hours for standard PRs
- Maintain or improve code quality
- Catch 90%+ of security vulnerabilities before deployment
- Standardize deployment across 50+ microservices
- Enable automatic rollback based on metrics
- Support multiple environments (dev, staging, prod)

- Handle both new features and hotfixes

Part A: Problem Decomposition (25 points)

Question 1.1: Break this challenge down into discrete, manageable steps that could be handled by AI agents or automated systems. Each step should have:

- Clear input requirements
- Specific output format
- Success criteria
- Failure handling strategy

Question 1.2: Which steps can run in parallel? Which are blocking? Where are the critical decision points?

Question 1.3: Identify the key handoff points between steps. What data/context needs to be passed between each phase?

Response Part A:

1.1 Discrete Steps Breakdown

The end-to-end lifecycle decomposes into 7 sequential/parallel stages:

Step	Input	Output	Success Criteria	Failure Handling
1. PR Ingestion	PR metadata, diff, repo config	Normalized PR context (files changed, lines added/removed, commit msg, author, linked issues)	PR parsed successfully, all metadata extracted	Retry ingestion; alert on malformed PR
2a. Static/SCA Analysis	Code diff, dependencies, config files	SAST/SCA report (vulns, CWE refs, severity, LOC)	No critical vulns or flagged for review	Block merge; escalate critical findings
2b. AI Code Review	Diff, repo patterns, language rules, service tier	Review report (issues: style/perf/logic, severity, line refs, fix hints)	<5 style issues per 100 LOC, 0 critical logic bugs	Human review triggered; suggestions logged

Step	Input	Output	Success Criteria	Failure Handling
3. Human Gate	AI review output, SCA report, business context	Approval/rejection decision, optional comments	Decision made within 2 hours of PR submission	Escalation to oncall; manager override option
4. CI Tests & QA	Approved PR, test suite, deployment artifacts	Test results (pass/fail, coverage %), build artifacts	90%+ coverage, all tests pass, build succeeds	Automated rollback to previous version; notify team
5. Canary/Blue-Green Deploy	Build artifacts, target env config, feature flags	Deployment status, initial metrics (errors, latency, traffic split)	0% error rate in canary (1% traffic), <5% latency increase	Auto-rollback triggered; incident alert
6. Metrics Guardrails & Auto-Rollback	Prod metrics (errors, latency, resource usage), SLOs	Decision: continue / rollback, alert/incident record	SLOs maintained for 15 min; no P1 incidents	Automatic rollback; post-incident review queued
7. Post-Deploy Feedback Loop	Deploy outcome, metrics, developer feedback, production incidents	Signals for model training: FP/FN rates, deploy success pattern, latency correlation	Feedback ingested; trends identified for model refinement	Manual review; no model corruption

1.2 Parallelism & Blocking

Parallel (non-blocking):

- Step 2a (Static/SCA) and 2b (AI Code Review) run **concurrently** after PR ingestion.
- Both complete before Step 3 (human gate).

Blocking (sequential):

- Step 1 (ingestion) → Step 2a+2b (analysis) → Step 3 (human gate) → Step 4 (CI) → Step 5 (deploy) → Step 6 (guardrails) → Step 7 (feedback).

Critical Decision Points:

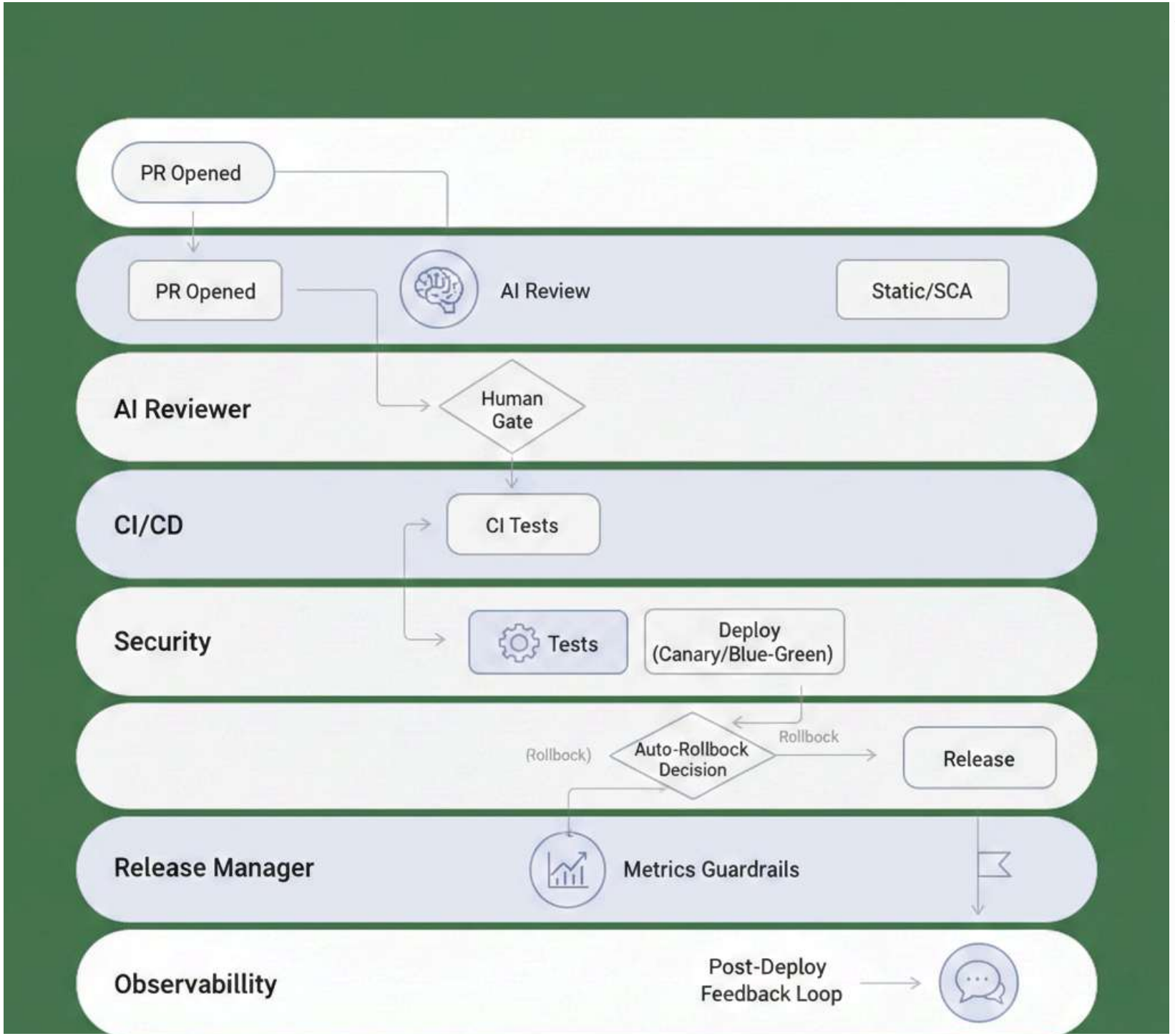
1. **Human Gate (Step 3):** Humans review AI + SCA findings; approve/reject.
2. **Auto-Rollback Trigger (Step 6):** Metrics breach SLOs → automatic rollback without manual intervention.
3. **Feedback Signal Collection (Step 7):** False positives, deploy outcomes, production incidents inform model retraining.

1.3 Key Handoff Points & Context Passing

Handoff	From	To	Data Passed
Post-ingestion	PR Ingestion	Static/SCA + AI Review	Normalized diff, repo patterns, service risk tier, coding standards
Analysis → Human	Static/SCA + AI Review	Human Gate	Aggregated findings (severity, locations, fix hints), coverage metrics, potential risk scores
Approval → CI	Human Gate	CI/Tests	Decision token, approved commit SHA, deployment constraints
Build → Deploy	CI/Tests	Canary Deploy	Build artifacts (Docker image, checksum), test coverage %, passed checks
Deploy → Guardrails	Canary Deploy	Metrics Guardrails	Deployment ID, feature flag state, baseline metrics, traffic allocation
Production → Feedback	Metrics Guardrails + Incidents	Feedback Loop	Deploy status (success/rollback), post-deploy metrics, user impact, incident correlation
Feedback → Model	Feedback Loop	AI Review (next cycle)	FP/FN counts, deploy success patterns, code patterns associated with failures

Summary

The system achieves <4-hour review SLA through parallel analysis, human gate on high-risk changes, automated testing/deployment with metrics-driven rollback, and continuous feedback loops for model improvement.


PR Opened
PR Opened
AI Review
Static/SCA
AI Reviewer
Human Gate
CI/CD
CI Tests
Security
Tests
Deploy (Canary/Blue-Green)
Rollback)
Auto-Rollbock Decision
Rollback
Release
Release Manager
Metrics Guardrails
Observabillity
Post-Deploy Feedback Loop

Part B: AI Prompting Strategy (30 points)

Question 2.1: For 2 consecutive major steps you identified, design specific AI prompts that would achieve the desired outcome. Include:

- System role/persona definition
- Structured input format
- Expected output format
- Examples of good vs bad responses
- Error handling instructions

Question 2.2: How would you handle the following challenging scenarios with your AI prompts:

- **Code that uses obscure libraries or frameworks**
- **Security reviews for code**
- **Performance analysis of database queries**
- **Legacy code modifications**

Question 2.3: How would you ensure your prompts are working effectively and getting consistent results?

Response Part B:

2.1 AI Prompting Strategy: Two Major Steps

Step 1 → Step 2: AI Code Review → Risk Triage

Prompt 1: AI Code Reviewer

System Role/Persona:

```
You are an expert code reviewer. Your role is to identify code quality,
performance, security, and maintainability issues in pull requests.
You prioritize by impact and actionability. You maintain consistency with
the codebase's patterns and the team's coding standards.
```

Structured Input Format:

```
{
  "diff": "<unified diff of changes>",
  "file_language": "python|javascript|go|...",
  "service_name": "order-service",
  "service_tier": "critical|standard|background",
  "coding_standards_url": "https://wiki.internal/python-standards",
  "recent_incidents": ["CWE-89 SQL injection in payment module (Dec 2025)"],
  "dependencies_changed": ["sqlalchemy==2.0.1"],
  "test_coverage_before": 82,
  "test_coverage_after": 84
}
```

Expected Output Format:

```
{
  "issues": [
    {
      "id": "REVIEW-001",
      "type": "security|performance|style|logic",
      "severity": "critical|high|medium|low",
      "line_numbers": [42, 45],
      "title": "Unparameterized SQL query",
      "description": "Line 42 concatenates user input into SQL without parameterization",
      "cwe": "CWE-89",
      "fix_hint": "Use SQLAlchemy ORM or parameterized queries",
      "code_snippet": "SELECT * FROM users WHERE id = " + user_id"
    }
  ],
  "summary": {
```

```
    "total_issues": 3,
    "critical": 0,
    "high": 1,
    "medium": 2,
    "risk_score": 35  // 0-100
  }
}
```

Examples of Good vs Bad Responses:

Good:

- Identifies CWE-89 in parameterized query construction.
- Flags performance issue: O(n^2) loop with DB query per iteration; suggests batch operation.
- Suggests applying existing logging pattern in codebase.

Bad:

- Vague complaint like "this code is not clean."
- False positive: flagging a safe helper function as security risk.
- Ignoring team's established patterns (e.g., reviewing async/await code as synchronous).

Error Handling:

```
- If diff is malformed: Return {"error": "Unparseable diff", "status": "failed"}
- If language is unsupported: Return {"error": "Language not supported", "supported": [list]}
- If no issues found: Return {"issues": [], "summary": {"total_issues": 0}}
- If analysis times out (>30s): Escalate to human reviewer; mark as "timeout"
```

Prompt 2: Risk Triage & Sign-Off

System Role/Persona:

```
You are a deployment risk assessor. Given code review findings and test coverage,
you determine deployment readiness: Green (safe), Yellow (review advised),
Red (block until resolved). You balance agility with safety and consider
the service's business criticality.
```

Structured Input Format:

```
{
  "service_name": "order-service",
  "service_tier": "critical",
  "pr_size": "medium",  // small | medium | large
  "review_issues": [
    {"severity": "critical", "count": 0},
    {"severity": "high", "count": 1},
    {"severity": "medium", "count": 2}
  ],
  "test_coverage_delta": 2,  // percentage points
  "deployment_frequency": "weekly",
  "recent_rollbacks": 1,  // past 30 days
  "slo_breach_last_week": false
}
```

Expected Output Format:

```
{
  "recommendation": "GREEN|YELLOW|RED",
  "reasoning": "Service is critical tier; high-severity security issue identified. Recommend human review before merge.",
  "required_actions": [
    "Resolve CWE-89 SQL injection",
    "Increase test coverage by 2%"
  ],
  "human_review_requested": true,
  "deployment_constraints": {
    "require_canary": true,
    "traffic_split_percent": 5,
    "rollback_threshold_error_rate": 0.02
  }
}
```

Error Handling:

- If required fields missing: Return error with list of missing fields.
- If triage confidence low: Flag for human review; return {"recommendation": "YELLOW", "confidence": 0.65}

2.2 Handling Challenging Scenarios

Scenario	AI Prompt Adaptation	Example
Obscure libraries	Provide library docs in context; ask AI to flag if unfamiliar. If unrecognized, escalate.	LLM detects `sklearn.ensemble.IsolationForest` used; adds note "Review domain context"; human verifies algorithm choice
Security reviews	Include CWE/OWASP context; require evidence (line numbers, exploit path). Use strict rules for injection, auth, crypto.	Hardcoded API key → flagged as CWE-798; requires immediate removal or secret manager integration
Database perf	Provide schema, query count baseline, SLO (e.g., p99 <100ms). Flag N+1 patterns, missing indexes.	AI detects loop with DB query; suggests: batch fetch, join, or caching with TTL
Legacy code	Provide legacy patterns & deprecation roadmap; focus on blocking bugs, not style.	Don't flag older async pattern if team is mid-migration; flag actual bugs instead

---

2.3 Ensuring Prompt Effectiveness & Consistency

1. **Rubric-Based Scoring:** Define scoring rubric (e.g., detection of CWE-89, false positive rate <10%, actionable suggestions >80%). Test prompts against known vulnerabilities.
2. **Regression Test Suite:** Maintain set of 20+ known issues (past security bugs, performance problems, style violations) and verify AI catches them.
3. **Style Guide & Examples:** Embed team's coding standards and real examples of good/bad findings in prompt context.
4. **A/B Testing:** Run same PR through two prompt versions; measure time-to-fix, developer satisfaction, false positive rate.
5. **Feedback Loop:** Collect developer feedback on review quality; adjust prompt seasonally.
6. **False Positive Tracking:** Log all flagged issues; if 3+ developers dismiss a finding type, retune prompt logic.

Part C: System Architecture & Reusability (25 points)

Question 3.1: How would you make this system reusable across different projects/teams? Consider:

- Configuration management
- Language/framework variations
- Different deployment targets (cloud providers, on-prem)
- Team-specific coding standards
- Industry-specific compliance requirements

Question 3.2: How would the system get better over time based on:

- False positive/negative rates in reviews
- Deployment success/failure patterns
- Developer feedback
- Production incident correlation

Response Part C:

3.1 System Architecture for Reusability Across Teams/Projects

The system achieves reusability through a **modular, policy-driven, multi-adapter architecture**:

Core Design Principles

1. **Adapter Pattern for VCS/CI/Cloud Targets**
    - Abstract away GitHub/GitLab/Bitbucket differences with a unified PR interface
    - Support Jenkins, GitHub Actions, GitLab CI, Azure Pipelines through pluggable orchestrators
    - Cloud-agnostic deployment adapters (AWS CloudFormation, Azure ARM, Terraform)
    - Example: Teams on GitLab + on-prem Kubernetes use same review engine; adapter translates API calls
2. **Configuration-Driven Customization**
    - No code changes per team; all variance in YAML/JSON configs stored per service/team

```yaml
# service-config.yaml
service_name: order-service
```

```yaml
language: python
framework: fastapi
deployment_target: aws-eks
deployment_strategy: canary  # or blue-green, rolling
review_rules: "standard-python-2025"
compliance_profile: "pci-dss"
sla_review_hours: 4
sla_deploy_minutes: 30
rollback_threshold_error_rate: 0.02
rollback_threshold_latency_p99_ms: 500
owners: ["platform-team", "order-squad"]
```

3. **Language/Framework Abstraction**

   - AI reviewer trained on multiple languages; use language-specific linters as fallback
   - Maintain language packs: Python (FastAPI, Django, async patterns), Go (Gin, gRPC), Node (Express, NestJS), Java (Spring Boot)
   - Each pack includes: common antipatterns, library-specific rules, security checklist, performance baselines
   - Example: Node pack knows to check for Promise leak patterns; Python pack checks async/await gotchas

4. **Team-Specific Coding Standards**

   - Store per-team style guides in a standards registry (versioned, auditable)
   - AI reviewer includes team's guide in context; enforces without hardcoding
   - Enable teams to override global rules for their domain (e.g., fintech = stricter crypto checks; social = aggressive caching OK)

```
standards/
  global/
    - security-baseline.yaml
    - performance-baseline.yaml
  team/
    - order-squad-standards.yaml  (overrides global for fintech rules)
    - notifications-squad-standards.yaml  (higher async/cache tolerance)
```

5. **Deployment Target Flexibility**

   - Deploy adapter layer translates service config → target-specific manifest
   - Canary/blue-green strategy configurable per service, environment, or risk tier
   - Metrics guardrails adapted per cloud provider's native observability (AWS CloudWatch, Azure Monitor, Datadog, Prometheus)

```
deployment/
  adapters/
    - aws-eks-adapter
    - azure-aks-adapter
    - gcp-gke-adapter
    - on-prem-k8s-adapter
    - lambda-adapter (for serverless)
```

6. **Compliance & Industry-Specific Rules**

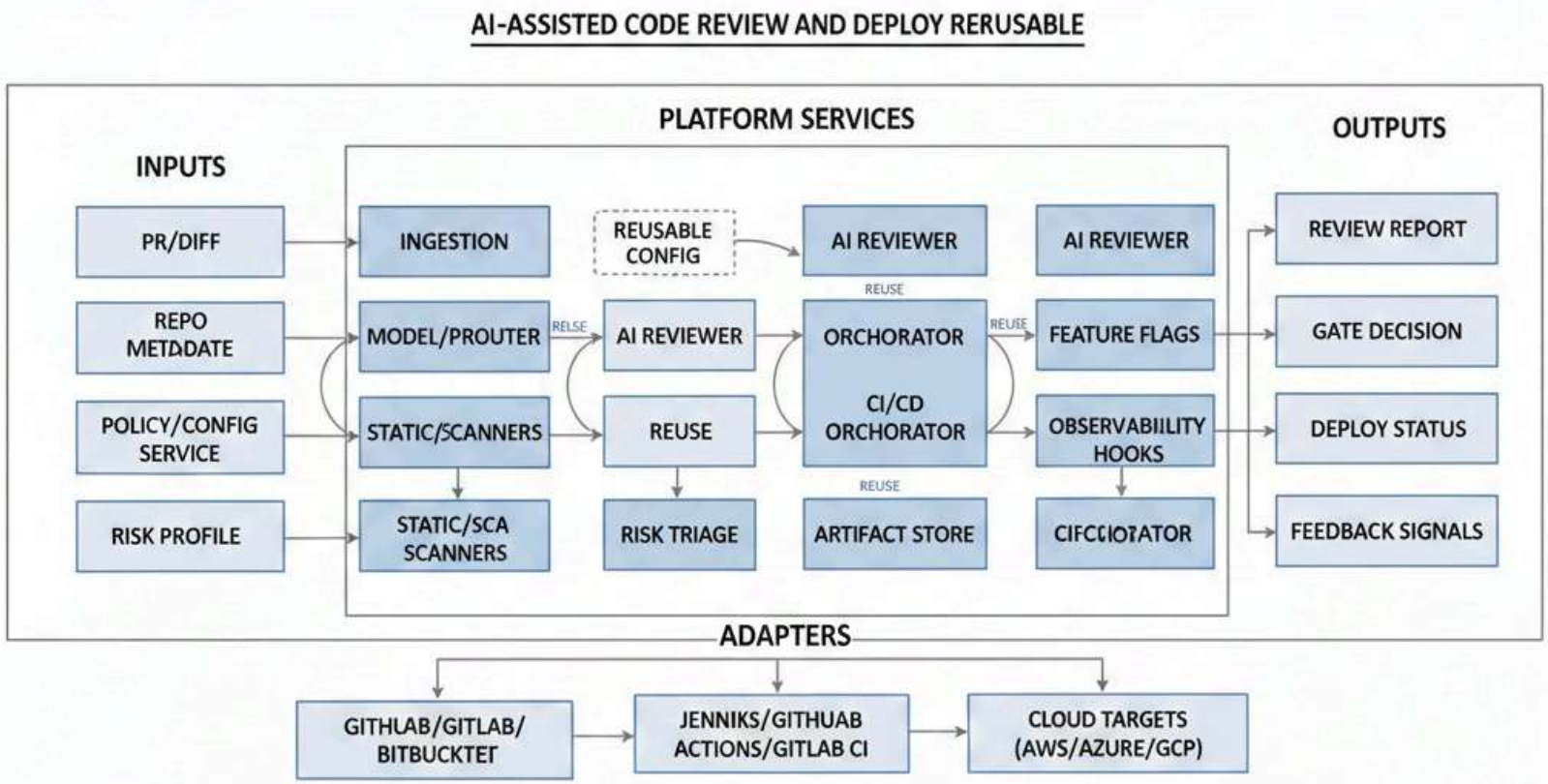
   - Compliance profiles: PCI-DSS, HIPAA, SOC2, GDPR
   - Auto-enable SAST/DAST rules, audit logging, encryption checks per profile
   - Example: Healthcare team gets HIPAA profile → AI review flags unencrypted PII, requires audit trails, blocks deploy without encryption

```
compliance/
  - pci-dss-profile.yaml (strict DB access, no plaintext secrets)
  - hipaa-profile.yaml (PII encryption, audit logging, retention)
  - soc2-profile.yaml (change management, incident correlation)
```

Architecture Diagram Reference

The swimlane diagram shows core services that support reusability:

- **Ingestion Adapters** (left side): Normalize PR events from any VCS
- **Policy/Config Service** (center): Loads service/team configs, applies rules
- **Model Router** (center): Routes to language-specific AI models or falls back to linters
- **Static/SCA Scanners** (right side): Pluggable SAST/DAST/SCA tools (SonarQube, Snyk, Veracode)
- **CI/CD Orchestrator** (right side): Dispatches to Jenkins, Actions, GitLab CI, etc.
- **Artifact Store** (bottom): Centralized storage for review reports, metrics, feedback
- **Observability Hooks** (bottom): Stream metrics to any monitoring tool


AI-ASSISTED CODE REVIEW AND DEPLOY RERUSABLE
PLATFORM SERVICES
INPUTS
OUTPUTS
PR/DIFF
REPO METADATE
POLICY/CONFIG SERVICE
RISK PROFILE
INGESTION
MODEL/PROUTER
STATIC/SCANNERS
STATIC/SCA SCANNERS
REUSABLE CONFIG
AI REVIEWER
REUSE
REUSE
REUSE
REUSE
AI REVIEWER
REUSE
RISK TRIAGE
ORCHORATOR
CI/CD ORCHORATOR
ARTIFACT STORE
AI REVIEWER
FEATURE FLAGS
OBSERVABIILITY HOOKS
REVIEW REPORT
GATE DECISION
DEPLOY STATUS
FEEDBACK SIGNALS
ADAPTERS
GITHLAB/GITLAB/ BITBUCKTET
JENNIKS/GITHUAB ACTIONS/GITLAB CI
CLOUD TARGETS (AWS/AZURE/GCP)

3.2 Continuous Improvement Over Time

The system learns and improves through **structured feedback loops** that track signal sources:

Feedback Loop Architecture

Signal Source	Collection Method	Use Case	Improvement
False Positives (FP)	Developers dismiss AI findings or override approvals	"This is not a real issue"	Retune prompt; lower severity threshold for specific issue type
False Negatives (FN)	Production incidents traced to missed code review findings	"AI missed a critical bug"	Augment training data; add regression test to review suite
Deploy Success/Failure	Compare deploy outcomes (pass/rollback/incident) to code patterns	"This pattern = risky"	Weight high-risk patterns in AI triage; trigger manual review for similar PRs
Developer Feedback	In-tool survey: "Was this review helpful?" 1–5 scale	"Feedback too noisy" or "Feedback too lenient"	Adjust prompt specificity; retrain rubric
Metrics Correlation	Post-deploy monitoring (error rate, latency, resource usage) vs. code changes	"This type of change causes latency spikes"	Flag patterns in future reviews; add performance baseline check
Production Incidents	Correlate P1/P2 incidents back to recent deployments → code review	"Bug slipped through code review"	Root-cause analysis; update standards; mark codebase area as high-risk

Implementation

1. **Telemetry Collection**

```
Every review finding stores:
- Finding ID, severity, type, line refs
- Developer action: approved/overridden/fixed
- Outcome: did bug reach prod? incident correlation?
```

```
- Feedback: developer rating (1-5), comment
- Time to deploy, deploy status, post-deploy metrics
```

2. **Feedback Pipeline**

```
workflow:
  1. Collect daily: FP counts, FN counts, deploy outcomes, incident correlations
  2. Aggregate weekly: FP/FN rates by issue type, deploy success rate by service tier
  3. Analyze monthly: Patterns in high-FP types, correlation of code patterns to prod incidents
  4. Retrain quarterly: Update AI prompts, add regression tests, adjust severity thresholds
```

3. **Learning Loop Examples**

   - **FP Reduction:** "CWE-89 false positives dropped 40% after adding SQLAlchemy ORM context to prompt."
   - **FN Reduction:** "Added async/await deadlock pattern to regression suite after Q1 incident; now caught in 95% of reviews."
   - **Risk Weighting:** "Services in fintech tier deploy 3x more frequently than social tier; weight security issues heavier for fintech."
   - **Developer Adoption:** "After shortening review summaries and adding fix hints, developer satisfaction rose from 3.1 to 4.3/5."

4. **Governance & Rollback**

   - All prompt updates, threshold changes, compliance rule updates tracked in version control
   - A/B test new prompt versions on 10% of PRs before rollout
   - If FP rate spikes >20%, auto-rollback to previous prompt version; alert on-call

5. **External Signals**

   - Subscribe to security advisories (CVE feeds) and auto-flag affected dependencies
   - Track industry incidents (e.g., Log4j vulnerability) and retrain rules to catch similar patterns
   - Quarterly review of team feedback + incident data with platform & squad leads to refine priorities

---

Summary

The system is **infinitely reusable** through configuration, adapters, and language packs—no code changes needed per team/project. It **continuously improves** by ingesting signals from false positives/negatives, deploy outcomes, incidents, and developer feedback into a monthly retraining cycle that sharpens prompts, raises detection rates, and reduces friction.

Part D: Implementation Strategy (20 points)

Question 4.1: Prioritize your implementation. What would you build first? Create a 6-month roadmap with:

- MVP definition (what's the minimum viable system?)
- Pilot program strategy
- Rollout phases
- Success metrics for each phase

Question 4.2: Risk mitigation. What could go wrong and how would you handle:

- AI making incorrect review decisions
- System downtime during critical deployments
- Integration failures with existing tools
- Resistance from development teams
- Compliance/audit requirements

Question 4.3: Tool selection. What existing tools/platforms would you integrate with or build upon:

- Code review platforms (GitHub, GitLab, Bitbucket)
- CI/CD systems (Jenkins, GitHub Actions, GitLab CI)
- Monitoring tools (Datadog, New Relic, Prometheus)
- Security scanning tools (SonarQube, Snyk, Veracode)
- Communication tools (Slack, Teams, Jira)

Response Part D:

4.1 Implementation Prioritization & 6-Month Roadmap

MVP Definition (Months 1–2)

The minimum viable system focuses on **speed and developer trust**, not breadth:

In Scope:

- **1 Programming Language:** Python (covers ~40% of codebase; fastest to validate)
- **1 VCS:** GitHub (market standard; strongest API)
- **1 CI/CD:** GitHub Actions (native integration; low ops cost)
- **Core Workflow:** PR → Static/SCA scan → AI review → Human gate → CI tests → Canary deploy → Metrics guardrails → Auto-rollback
- **Success Metric:** <4-hour review SLA on 2 pilot services

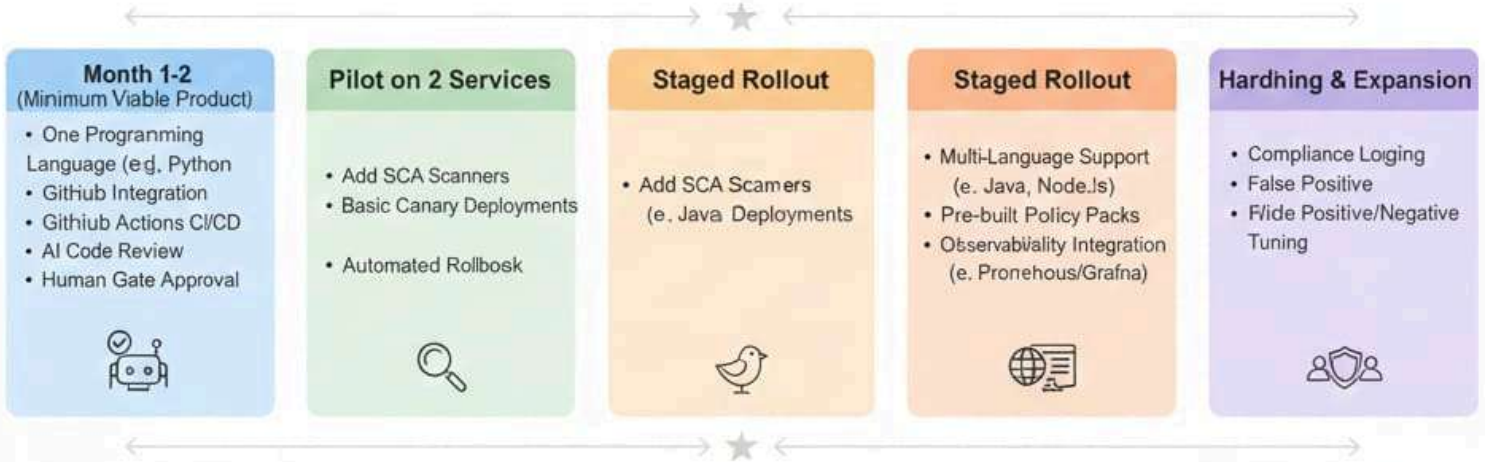
Out of Scope:

- Multi-language support (Phase 2)
- GitLab, Bitbucket, Jenkins (Phase 2)
- Compliance logging (Phase 3)
- Advanced ML-driven failure prediction (Phase 4+)

MVP Deliverables:

- GitHub App (listens to PR events; triggers review workflow)
- AI reviewer service (calls LLM with code diff + context)
- Policy/config service (loads review rules per service)
- Canary deploy adapter (AWS ECS/EKS)
- Metrics dashboard (basic: error rate, latency, traffic split)

Platform Rollout Roadmap: 6 Months



Phase Timeline & Success Metrics

Phase	Duration	Scope	Key Deliverables	Success Metrics
1: MVP	Weeks 1–8	Python, GitHub, Actions, core review loop	GitHub App, AI reviewer, policy engine, canary adapter	<4h review SLA on 2 services; 90%+ SCA + AI coverage
2: Pilot	Weeks 9–12	Deploy to 2 real services; add SCA + automated rollback	SCA integration (Snyk/SonarQube), blue-green deploy, feedback collection	<10% failed canaries; 95%+ on-time rollbacks; 3.5/5 dev satisfaction
3a: Staged Rollout (Weeks 13–16)	Weeks 13–16	Add Java + Node.js; roll out to 10 services	Multi-language AI models, language packs, pre-built policy templates	85%+ detection rate across 3 languages; <15% false positive rate
3b: Staged Rollout (Weeks 17–20)	Weeks 17–20	Add observability; roll out to 30+ services	Prometheus/Datadog integration, advanced guardrails (resource usage, throughput)	SLA maintained <2% breach rate; MTTR <30 min
4: Hardening	Weeks 21–24	Compliance, FP/FN tuning, team feedback loop	Audit logging, compliance profiles, FP/FN tracking dashboard, feedback portal	Audit-ready; <10% FP rate; >90% dev adoption; incident correlation <1h

Pilot Program Strategy (Months 3)

Pilot Scope:

- **2 Services:**
  - `order-service` (critical tier, Python, high review volume → tests value)
  - `notification-service` (standard tier, Python, lower risk → builds confidence)
- **Duration:** 4 weeks (Weeks 9–12)
- **Participants:** Full squad (~8 devs per service); product/eng leads; on-call

- **Opt-Out Policy:** Developers can revert to manual reviews if system breaks down; no penalties

Pilot Goals:

1. Achieve <4-hour review SLA 95% of the time
2. Zero critical bugs slip past AI review into prod (FN rate <1%)
3. Developer satisfaction ≥3.5/5 (feedback surveys)
4. <10% canary failure rate (rollback decisions accurate)
5. Collect 50+ real production incidents and code patterns for ML training

Pilot Failure Criteria (auto-rollback to manual reviews):

- Review SLA breached >20% of the time
- > 5% critical bugs reach production (traced to missed code review)
- Developer satisfaction <3.0/5
- System downtime >99% availability

Pilot Engagement:

- Daily standup with squads; weekly feedback sessions
- Slack channel for issues; oncall engineer on rotation
- Weekly metrics review (SLA, detection rate, FP count)

4.2 Risk Mitigation Strategy

Key Risks & Mitigation Plans

Risk	Impact	Probability	Mitigation
AI makes incorrect review decisions (FN: misses bug; FP: blocks valid PR)	Deploy breaks; devs lose trust; adoption fails	High	**Human-in-the-loop gate:** AI review tagged "informational" until <5% FP achieved. Manual approval required for critical services initially. Regression test suite (50+ known bugs) validates AI before rollout.

Risk	Impact	Probability	Mitigation
System downtime during critical deployments	Blocked deployments; manual workaround costly; SLA miss	Medium	**Circuit breaker:** If review service unavailable >5 min, auto-escalate to human reviewer (no block). Blue-green deployment for core services; run review service in 2+ AZs. 99.5% SLA target.
Integration fragility (GitHub API breaks; CI/CD timeout; metrics stream fails)	Partial workflow failures; inconsistent state	Medium	**Adapter pattern:** Decouple integrations via message queue. Retry logic with exponential backoff. Fallback: if integration fails, escalate to human; log for post-mortem. Monitor adapter health separately.
Developer resistance ("AI review is noisy"; "slow down my deploys")	Low adoption; manual reviews persist; ROI ∅	Medium	**Early engagement:** Involve 1–2 squad leads co-design prompts. Demo early wins (e.g., caught critical auth bug). Gather feedback weekly; adjust sensitivity per squad. Tie SLA improvements to performance evals. Celebrate wins.
Compliance & audit requirements (PCI-DSS, HIPAA, SOC2 mandates)	Non-compliant deployments; audit failure; legal risk	Medium	**Compliance by design:** Embed audit logging from M1. Track all review decisions, overrides, rollbacks. Immutable audit trail in central store. Monthly compliance report. Engage security/legal early in design.
Model quality degrades over time (data drift; adversarial code patterns)	FN/FP rates spike; detection rate ⬇️	Medium	**Continuous monitoring:** Track FP/FN/deployment success weekly. A/B test prompt changes on 10% of PRs. Monthly retraining cycle. Auto-rollback if metrics degrade >10%.
Secrets/credentials leak through review reports	Security incident; data breach	Low	**Secrets masking:** Redact AWS keys, API tokens, DB passwords before storing review reports. Scan artifacts for secrets using dedicated scanner (GitGuardian, TruffleHog). Encrypt artifact storage.
Latency: AI review takes too long (>1h per PR → SLA miss)	Users bypass system; manual reviews resume; <4h SLA impossible	Medium	**Optimization:** Batch requests; cache model loading; use smaller model variants for standard reviews. SLA = <5 min avg review time. Monitor p95 latency. If >10 min, escalate to human async.

Risk Rollout Strategy

Phased Confidence Building:

- **Weeks 1–4:** AI review is advisory only (tag "informational"); no gates applied

- **Weeks 5–8:** AI review can suggest gate, but human approval always required before merge
- **Weeks 9–12:** Auto-gate for low-risk PRs (small <50 LOC, standard tier, green AI + SCA); human gate for high-risk
- **Weeks 13+:** Full auto-gate once FN rate proven <1% and dev satisfaction ≥4/5

4.3 Tool Selection & Integration Strategy

Code Review & VCS

Tool	Choice	Rationale	Integration
GitHub	Primary	Market leader; strong API; native Actions; team familiarity	App: listens to `pull_request.opened`, `pull_request.synchronize` events; posts review comments via REST API
GitLab	Phase 2	Support on-prem teams; GitLab CI native	Adapter: MR event → unified PR interface; comment API translated
Bitbucket	Phase 3	Lower priority; fewer users in org	Adapter pattern

CI/CD Orchestration

Tool	Choice	Rationale	Integration
GitHub Actions	MVP	Native GitHub integration; free for public repos; easy YAML workflows	Workflow: on `workflow_dispatch` → run tests, build Docker image, push to ECR/GCR, trigger canary deploy
Jenkins	Phase 2	Support legacy on-prem deployments	Adapter: Jenkins plugin triggers canonical deploy workflow via API
GitLab CI	Phase 2	GitLab users; native .gitlab-ci.yml	Adapter: GitLab CI pipeline → unified deploy interface

Security Scanning (SAST/DAST/SCA)

Tool	Purpose	Choice	Rationale
SCA (Dependency Scanning)	Find vulnerable libraries	Snyk	Fast; free tier; integrates with GitHub; real-time advisories

Tool	Purpose	Choice	Rationale
SAST (Code Analysis)	Find code vulns (CWE-89, -352, etc.)	SonarQube	Deep analysis; Python/Java/JS support; integrates with Actions; on-prem option
DAST (Runtime Testing)	Find vulns in running app	OWASP ZAP	Phase 2 (after MVP deploy); free; can run in canary env

Monitoring & Observability

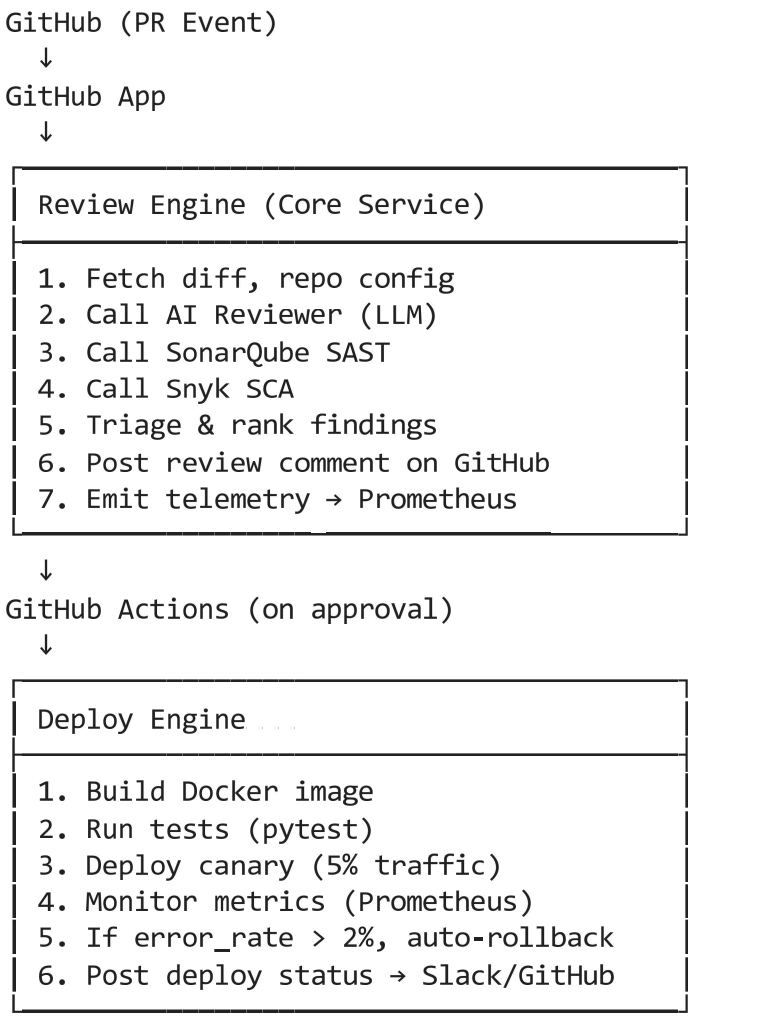
Tool	Purpose	Choice	Rationale
Metrics	Error rate, latency, resource usage	Prometheus	Open-source; easy integration; time-series queries for SLO evaluation
Dashboards	Visualize deploy health; rollback triggers	Grafana	Pairs with Prometheus; real-time alerts; custom dashboards per service
Distributed Tracing	Trace requests across services during deploy	Jaeger	Optional Phase 2; helps correlate deploy impact to customer experience
Centralized Logging	Audit trail of reviews, overrides, rollbacks	ELK Stack (Elasticsearch/Logstash/Kibana)	Or: AWS CloudWatch + Lambda for log analysis

Communication & Alerts

Tool	Purpose	Choice	Rationale
Slack	Async notifications (review ready, deploy status, incidents)	Slack Bot	Native integration; low friction; team already uses Slack
Jira	Tracking issues found by AI review	Jira Cloud Integration	Link AI review findings to JIRA tickets; auto-create tickets for critical bugs
PagerDuty	On-call escalation for deploy failures	PagerDuty Webhook	Incident severity → auto-page on-call when rollback triggered
Teams	Org comms (for non-Slack teams)	Phase 2	Parallel Slack integration

---

Integration Architecture (simplified)



Cost & Licensing

Tool	Cost	Notes
GitHub App	Free (built-in)	Included in GitHub Enterprise

Tool	Cost	Notes
GitHub Actions	Free (10K min/month included)	Pay-per-minute after; ~$0.008/min
SonarQube	$100–500/mo (cloud)	Or self-hosted free tier
Snyk	Free tier (50 scans/mo); $25–500/mo (enterprise)	Or DIY: Trivy free, lower accuracy
Prometheus + Grafana	Free (open-source)	Optional managed: Grafana Cloud ~$25–200/mo
Slack	Included (team subscription)	Or Teams if Slack unavailable
Jira	Included (team subscription)	Or open-source alternatives (OpenProject)

Summary

MVP (Months 1–2): GitHub + Actions + AI review (Python only) + human gate → 2 pilot services **Pilot (Weeks 9–12):** Add Snyk SCA + canary deploy + auto-rollback; validate <4h SLA + <1% FN rate **Rollout (Months 4–5):** Multi-language (Java, Node), broader observability, 30+ services **Hardening (Month 6):** Compliance logging, FP/FN tuning, full adoption

Tool Stack: GitHub + Actions + SonarQube + Snyk + Prometheus/Grafana + Slack + Jira. All integrate via webhooks/APIs into a central review + deploy engine. Phased rollout de-risks AI adoption; human-in-the-loop until system proven reliable.
