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 re 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 handle 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 dec points?

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

Response Part A: Problem Decomposition

Question 1.1: Discrete Steps with Clear Requirements

Step 1: PR Intake & Analysis

- **Input**: PR metadata (files changed, commit messages, author, base/target branch)
- **Output**: Structured PR profile (language, scope, risk level, estimated complexity)
- **Success Criteria**: All files parsed, dependencies identified, risk score calculated
- **Failure Handling**: Flag for manual review if parsing fails; default to high-risk category

Step 2: Code Quality Analysis (Automated Linting)

- **Input**: Changed files, project configuration (.eslintrc, .pylintrc, etc.)
- **Output**: Lint violations with severity (error/warning), line numbers, auto-fix suggestions
- **Success Criteria**: All files linted, violations categorized
- **Failure Handling**: Partial results accepted; unsupported files skipped with notification

Step 3: Security Vulnerability Scan

- **Input**: Changed files, dependency manifests (package.json, requirements.txt)
- **Output**: CVE list, OWASP Top 10 violations, severity scores, remediation steps
- **Success Criteria**: All security rules executed, dependencies scanned for known vulnerabilities
- **Failure Handling**: Fail-safe to block if scanner errors; manual security review required

Step 4 : AI-Powered Code Review (Logic & Best Practices)

- **Input**: Code diff, PR description, project context (README, architecture docs)
- **Output**: Structured review comments (bugs, performance issues, maintainability, suggestions)
- **Success Criteria**: Review completed within 10 minutes, > 5 actionable comments ge
- **Failure Handling**: Timeout triggers simplified review; retry with reduced context if LLM fails

Step 5 : Test Coverage & Validation

- **Input**: Test files, code coverage report, CI test results
- **Output**: Coverage delta, missing test scenarios, risk areas
- **Success Criteria**: Coverage calculated, critical paths identified
- **Failure Handling**: Warn if coverage unavailable; recommend manual testing

Step 6 : Performance Impact Analysis

- **Input**: Changed files, performance benchmarks, database queries
- **Output**: Performance risk score, query efficiency analysis, N+1 detection
- **Success Criteria**: All queries analyzed, benchmark comparisons made
- **Failure Handling**: Skip if benchmarks unavailable; flag for manual perf testing

Step 7 : Review Aggregation & Prioritization

- **Input**: Outputs from Steps 2-6
- **Output**: Unified review report with priority-ranked issues (blocking, recommended, optional)
- **Success Criteria**: All issues categorized, duplicates removed, actionable summary created
- **Failure Handling**: Partial aggregation accepted; manual review fills gaps

Step 8 : Developer Notification & Collaboration

- **Input**: Aggregated review, developer contact info
- **Output**: PR comments posted, Slack/email notifications sent
- **Success Criteria**: All stakeholders notified, comments linked to code lines
- **Failure Handling**: Retry notification 3x; escalate to manual if fails

Step 9 : Approval Decision (Human-in-the-Loop)

- **Input**: Review report, team policy (e.g., "2 approvals required")
- **Output**: Approval status, merge eligibility

- **Success Criteria**: Policy enforced, decision logged
- **Failure Handling**: Default to "needs review" if logic unclear

Step 10: Deployment Pipeline Orchestration

- **Input**: Approved PR, target environment (dev/staging/prod), deployment config
- **Output**: Deployment job triggered, health checks scheduled
- **Success Criteria**: Deployment initiated, metrics baseline captured
- **Failure Handling**: Rollback on health check failure; alert on-call engineer

Step 11: Post-Deployment Monitoring

- **Input**: Deployment metadata, metrics (error rate, latency, CPU), logs
- **Output**: Health status, anomaly alerts, rollback recommendation
- **Success Criteria**: Metrics monitored for 1 hour, no anomalies detected
- **Failure Handling**: Auto-rollback if error rate > 5%; page on-call if critical

Step 12: Feedback Loop & Learning

- **Input**: Deployment outcome, false positive/negative flags from developers
- **Output**: Updated AI model weights, revised review rules
- **Success Criteria**: Feedback processed, model retrained weekly
- **Failure Handling**: Manual review of edge cases; fallback to previous model version

---

Question 1.2: Parallelization & Dependencies

Parallel Execution:

- **Steps** 2, 3, 4, 5, 6 can run **in parallel** after Step 1 completes (all analy same PR independently)
- **Step** 11 monitoring runs in parallel with production traffic (non-blocking)

Blocking Dependencies:

```
Step 1 (PR Intake)
    ↓
├─ Step 2 (Linting) ────┐
├─ Step 3 (Security) ───┤
├─ Step 4 (AI Review) ──┤─→ Step 7 (Aggregation)
├─ Step 5 (Tests) ──────┤        ↓
└─ Step 6 (Performance)─┘   Step 8 (Notification)
                                 ↓
```

```
Step 9 (Approval)
     ↓
Step 10 (Deployment)
     ↓
Step 11 (Monitoring)
     ↓
Step 12 (Feedback Loop)
```

Critical Decision Points:

1. **After Step 3 (Security)**: If critical CVEs found → **block merge** (hard stop)
2. **After Step 7 (Aggregation)**: If > 10 blocking issues → **reject PR** automatically
3. **After Step 9 (Approval)**: Human override required for high-risk deploys
4. **After Step 11 (Monitoring)**: Auto-rollback if error rate > 5% or latency > 2x baseline

Question 1.3: Key Handoff Points & Data Context

Handoff	From → To	Data Passed	Format
1 → 2-6	PR Intake → Analysis Steps	File paths, diff, metadata, project config	JSON: `{pr_id, files [...], languag base_branch}`
2-6 → 7	Analysis → Aggregation	Individual findings	JSON: `{step_na issues: [{seve message, line file}]}`
7 → 8	Aggregation → Notification	Unified report	Markdown + JSON `{blocking: [. warnings: [.. summary}`
8 → 9	Notification → Approval	Review status, approver list	JSON: `{approvals_ne current_appro blocking_issu`
9 → 10	Approval → Deployment	Merge commit SHA, environment	JSON: `{commit_ env: "prod", config: {...}`
10 → 11	Deployment → Monitoring	Deployment timestamp, service name, metrics baseline	JSON: `{deploy_time, service, baseline_metr {error_rate, latency}}`
11 → 12	Monitoring → Feedback	Outcome (success/ rollback), developer feedback	JSON: `{outcome false_positive [...], missed_issues [...]}`

Context Preservation Strategy:

- **State Store**: Redis/DynamoDB to persist PR state across steps
- **Message Queue**: Kafka/SQS for async step communication
- **Tracing**: OpenTelemetry for distributed tracing across steps
- **Audit Log**: Immutable log of all decisions and handoffs for compliance

Part B: AI Prompting Strategy (30 points)

Question 2.1: For 2 consecutive major steps you identified, design specific AI prompts th 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 consist results?

Response Part B: AI Prompting Strategy

Question 2.1: Prompt Design for Two Consecutive Steps

PROMPT 1: Step 4 - AI-Powered Code Review

System Role:

```
You are an expert software engineer with 15+ years of experience
reviewing code across
multiple languages and frameworks. Your role is to provide constructiv
actionable
feedback that improves code quality, security, and maintainability.

You follow these principles:
- Identify bugs, edge cases, and potential runtime errors
- Suggest performance optimizations with measurable impact
- Ensure code follows language-specific best practices
- Highlight security vulnerabilities (injection, auth issues, data lea
- Recommend refactoring only when it significantly improves readabilit
or performance
- Be concise and specific - cite line numbers and provide code example
```

```
  - Distinguish between critical issues (must fix) and suggestions (nice
  have)
```

Structured Input Format:

```
{
  "pr_metadata": {
    "pr_id": "PR-12345",
    "title": "Add user authentication endpoint",
    "description": "Implements JWT-based auth for /api/login",
    "author": "jane.doe",
    "target_branch": "main"
  },
  "code_diff": "... [git diff output] ...",
  "project_context": {
    "language": "Python",
    "framework": "FastAPI",
    "architecture": "microservices",
    "database": "PostgreSQL"
  },
  "coding_standards": {
    "max_function_length": 50,
    "enforce_type_hints": true,
    "security_rules": ["no-hardcoded-secrets", "sql-parameterization"]
  }
}
```

Expected Output Format:

```
{
  "review_summary":
"Found 1 critical security issue, 2 performance concerns, 3 style suggest
  "issues": [
    {
      "severity": "critical",
      "category": "security",
      "file": "app/auth.py",
      "line": 45,
      "message": "Hardcoded JWT secret key exposes authentication to atta
      "recommendation": "Move secret to environment variable: SECRET_KEY
os.getenv('JWT_SECRET')",
      "code_snippet": "SECRET_KEY = 'hardcoded-secret-123'  # ← CRITICAL"
    },
    {
      "severity": "high",
      "category": "performance",
      "file": "app/db.py",
      "line": 78,
      "message": "N+1 query detected - fetching users in loop causes 1000
queries",
      "recommendation": "Use JOIN or batch query: users =
db.query(User).filter(User.id.in_(user_ids)).all()",
      "code_snippet": "for user_id in user_ids:\n    user =
db.query(User).get(user_id)  # ← N+1"
    },
    {
      "severity": "medium",
      "category": "maintainability",
      "file": "app/utils.py",
      "line": 23,
      "message": "Function validate_input() is 85 lines - exceeds 50-line
limit",
      "recommendation": "Split into smaller functions: validate_email(),
validate_password(), validate_username()"
    }
  ],
  "positive_feedback": [
    "Excellent use of type hints throughout",
    "Good error handling with custom exceptions"
  ],
  "overall_risk_score": 7.5
}
```

Example Good Response:

```
✅ Specific: "Line 45: Hardcoded secret key" (not "Security issue
somewhere")
✅ Actionable: Provides exact fix with code example
✅ Prioritized: severity: "critical" vs "medium"
✅ Constructive: Includes positive feedback
```

Example Bad Response:

```
❌ Vague: "The code has some problems"
❌ No location: "There's a security issue" (no line number)
❌ Unhelpful: "Rewrite everything" (not actionable)
❌ No severity: All issues treated equally
```

Error Handling Instructions:

```
IF code_diff is empty:
  RETURN {"error": "No code changes detected", "action": "skip_review"

IF language not in [Python, JavaScript, TypeScript, Go, Java, Ruby]:
  RETURN {"warning": "Unsupported language", "action":
"fallback_to_generic_review"}

IF unable to parse code (syntax errors):
  RETURN {"issues": [], "warning": "Code parsing failed - recommend
manual review"}

IF timeout (>10 minutes):
  RETURN partial results with {"status": "incomplete", "reviewed_files
[...]}
```

PROMPT 2 : Step 7 - Review Aggregation & Prioritization

System Role:

```
    You are a technical project manager responsible for triaging code revi
    findings
    from multiple automated systems. Your goal is to consolidate,
    deduplicate, and
    prioritize issues so developers can focus on the most critical problem
    first.

    You follow these rules:
    - Merge duplicate issues (same line, similar message) into one
    - Prioritize critical/high severity issues at the top
    - Group issues by file to reduce context switching
    - Flag issues that conflict with each other
    - Provide a concise executive summary for non-technical stakeholders
    - Calculate an overall "merge readiness score" (0-100)
```

Structured Input Format:

```
{
  "pr_id": "PR-12345",
  "findings_from_steps": {
    "linting": {
      "tool": "eslint",
      "issues": [
        {"file": "app.js", "line": 23, "severity": "error", "message": "U
variable 'x'"},
        {"file": "app.js", "line": 45, "severity": "warning", "message":
"Console.log detected"}
      ]
    },
    "security": {
      "tool": "Snyk",
      "issues": [
        {"file": "auth.py", "line": 45, "severity": "critical", "cve":
"CWE-798", "message": "Hardcoded secret"}
      ]
    },
    "ai_review": {
      "tool": "GPT-4",
      "issues": [
        {"file": "auth.py", "line": 45, "severity": "critical", "message"
"Hardcoded JWT secret key"},
        {"file": "db.py", "line": 78, "severity": "high", "message":
"N+1 query detected"}
      ]
    },
    "test_coverage": {
      "coverage_delta": -5.2,
      "missing_tests": ["test_login_invalid_password"]
    }
  }
}
```

Expected Output Format:

```
{
  "executive_summary": "PR has 1 critical security issue (blocking), 1
performance issue (recommended fix), and 2 minor linting warnings. Test
coverage decreased by 5.2%. Recommend addressing security issue before me
  "merge_readiness_score": 45,
  "blocking_issues": [
    {
      "id": "AGG-001",
      "severity": "critical",
      "category": "security",
      "file": "auth.py",
      "line": 45,
      "deduplicated_from": ["security.issues[0]", "ai_review.issues[0]"],
      "consolidated_message": "Hardcoded JWT secret key (CWE-798) - must
moved to environment variable",
      "sources": ["Snyk", "GPT-4"],
      "recommendation": "BLOCK merge until fixed"
    }
  ],
  "recommended_fixes": [
    {
      "id": "AGG-002",
      "severity": "high",
      "file": "db.py",
      "line": 78,
      "message": "N+1 query - use batch query or JOIN",
      "estimated_impact": "Reduces DB calls from 1000+ to 1"
    }
  ],
  "optional_improvements": [
    {
      "id": "AGG-003",
      "severity": "low",
      "file": "app.js",
      "line": 23,
      "message": "Remove unused variable 'x'"
    }
  ],
  "test_coverage_alert": {
    "status": "degraded",
    "delta": -5.2,
    "recommendation": "Add tests for login failure scenarios"
  }
}
```

Good vs Bad Examples:

Good Response:

```
✅ Deduplication: Merged "Hardcoded secret" from Snyk + GPT-4 into one
issue
✅ Prioritization: Critical issues listed first, blocking merge
✅ Actionable summary: "Fix auth.py line 45 before merge"
✅ Risk assessment: merge_readiness_score = 45 (not ready)
```

Bad Response:

```
❌ Duplicates: Lists same issue twice from different tools
❌ No prioritization: All issues mixed together
❌ Vague summary: "Some issues found"
❌ No decision: Doesn't say if PR should be blocked
```

Error Handling:

```
IF no issues found in any step:
  RETURN {"status": "clean", "merge_readiness_score": 100,
"recommendation": "Approve"}

IF findings_from_steps is empty:
  RETURN {"error": "No analysis results available", "action":
"manual_review_required"}

IF conflicting recommendations (e.g., security says block, AI says
approve):
  RETURN {"conflict_detected": true, "escalate_to": "senior_engineer"}
```

Question 2.2: Handling Challenging Scenarios

Scenario 1: Code Using Obscure Libraries/Frameworks

Prompt Enhancement:

```
CONTEXT: You may encounter code using unfamiliar libraries or framewor

INSTRUCTIONS:
1. If you recognize the library, provide specific advice (e.g., "React
Hook useEffect dependency array is incorrect")
2. If you don't recognize the library:
   a. Analyze the code's *intent* based on function names, patterns, a
comments
   b. Provide general software engineering advice (error handling,
validation, etc.)
   c. Flag for human review: "Unknown library 'obscure-lib' - recommen
domain expert review"
3. DO NOT hallucinate library-specific advice if unsure
```

```
4. Search project documentation (README, docs/) for library usage
patterns

EXAMPLE:
Input: Code using "FastHTML" (obscure Python framework)
Output:
- "Unknown framework 'FastHTML' detected"
- "General observation: Function fetch_data() lacks error handling for
HTTP failures"
- "Recommendation: Add try/except for
requests.exceptions.RequestException"
- "FLAG: Requires review by engineer familiar with FastHTML"
```

Scenario 2 : Security Reviews for Code

Prompt Enhancement:

```
SECURITY REVIEW MODE:

Focus areas (OWASP Top 10 + CWE):
1. **Injection (SQL, NoSQL, Command)**: Check for parameterized querie
input sanitization
2. **Authentication**: Verify password hashing (bcrypt/Argon2), sessio
management, MFA
3. **Sensitive Data Exposure**: No secrets in code, encryption at rest
transit, proper access controls
4. **XML External Entities (XXE)**: Disable external entity processing
parsers
5. **Broken Access Control**: Verify authorization checks before data
access
6. **Security Misconfiguration**: Check default credentials, debug mod
off, CORS policies
7. **XSS**: Validate output encoding, CSP headers
8. **Insecure Deserialization**: Avoid pickle, eval, exec with untrust
data
9. **Using Components with Known Vulnerabilities**: Check dependency
versions against CVE databases
10. **Insufficient Logging**: Ensure security events (auth failures,
access denials) are logged

OUTPUT REQUIREMENTS:
- Map findings to CWE IDs: "CWE-89: SQL Injection on line 45"
- Provide exploit scenarios: "Attacker can inject '1 OR 1=1' to bypass
authentication"
- Severity based on CVSS: Critical (9-10), High (7-8.9), Medium (4-6.9
Low (0-3.9)
- Include remediation code examples

EXAMPLE:
Input: `query = f"SELECT * FROM users WHERE id = {user_id}"`
```

```
Output:
- Severity: CRITICAL
- CWE: CWE-89 (SQL Injection)
- Exploit: "user_id = '1 OR 1=1; DROP TABLE users--' bypasses auth and
deletes data"
- Fix: `query = "SELECT * FROM users WHERE id = ?"; cursor.execute(que
(user_id,))`
```

Scenario 3 : Performance Analysis of Database Queries

Prompt Enhancement:

```
PERFORMANCE ANALYSIS MODE:

Database query anti-patterns to detect:
1. **N+1 Queries**: Loop with individual SELECT statements
2. **Missing Indexes**: WHERE/JOIN on unindexed columns
3. **SELECT ***: Fetching unnecessary columns
4. **Unbounded Queries**: No LIMIT on result sets
5. **Implicit Type Conversion**: Mismatched column types in WHERE clau
6. **Subquery in SELECT**: Can often be optimized with JOIN
7. **Lack of Query Caching**: Repeated identical queries

ANALYSIS STEPS:
1. Extract all SQL queries from code changes
2. Identify query patterns (loop → N+1, missing LIMIT → unbounded)
3. Estimate impact: queries_per_request * avg_execution_time
4. Provide optimized alternatives with EXPLAIN PLAN if possible

OUTPUT FORMAT:
{
  "query": "SELECT * FROM users WHERE email = ?",
  "issue": "SELECT * fetches all 50 columns when only email, name
needed",
  "impact": "Increases network transfer by 400%, slows query by 2x",
  "optimized_query": "SELECT email, name FROM users WHERE email = ?",
  "add_index": "CREATE INDEX idx_users_email ON users(email)"
}

EDGE CASE: If query uses ORM (SQLAlchemy, Hibernate):
- Analyze ORM patterns: eager loading, lazy loading, N+1 detection
- Suggest ORM optimizations: joinedload(), selectinload()
```

Scenario 4 : Legacy Code Modifications

Prompt Enhancement:

```
LEGACY CODE REVIEW MODE:
Context: Code may be old (5-10+ years), lack tests, use deprecated
patterns.

ADJUSTED EXPECTATIONS:
- **Tolerate older patterns**: Don't demand full refactor (risk of
breaking changes)
- **Focus on safety**: Ensure new code doesn't break existing
functionality
- **Test requirements**: Require tests for new code, even if legacy la
them
- **Incremental improvement**: Suggest small, safe refactorings alongs
new features

REVIEW PRIORITIES (in order):
1. Does new code introduce security vulnerabilities?
2. Does new code break backward compatibility?
3. Is new code tested (even if legacy isn't)?
4. Does new code follow current best practices (without requiring lega
refactor)?

OUTPUT GUIDELINES:
- "New code adds SQL injection risk - fix required"  ✅
- "Legacy code lacks error handling - acceptable, but new code should
include it"  ✅
- "Entire file should be rewritten"  ❌ (too risky for legacy)

EXAMPLE:
Input: Adding a new API endpoint to a 10-year-old Express app (uses
callbacks, no async/await)
Output:
- "Legacy code uses callbacks - acceptable for existing routes"
- "NEW code in routes/new-endpoint.js should use async/await (modern b
practice)"
- "Recommend adding tests for new endpoint (even though legacy routes
lack them)"
- "Do NOT suggest rewriting all legacy routes to async/await (high ris
out of scope)"
```

Question 2.3: Ensuring Prompt Effectiveness & Consistency

Strategy 1: Automated Prompt Testing

```python
# Create test suite of PRs with known issues
test_cases = [
    {
        "pr": "test-pr-sql-injection.diff",
        "expected_issues": [
            {"severity": "critical", "category": "security", "line": 45,
"cwe": "CWE-89"}
        ]
    },
    {
        "pr": "test-pr-n+1-query.diff",
        "expected_issues": [
            {"severity": "high", "category": "performance", "pattern": "N
        ]
    }
]

# Run AI review on test cases weekly
for test in test_cases:
    result = ai_review(test["pr"])
    assert all(expected in result["issues"] for expected in
test["expected_issues"])
```

Metrics to Track:

- **Precision**: % of AI-flagged issues that are true positives (target: > 80%)
- **Recall**: % of actual issues detected by AI (target: > 90% for critical issues)
- **Consistency**: % of identical PRs receiving same review (target: > 95%)
- **False Positive Rate**: % of flagged issues developers mark as "not an issue" (target: < 10%)

Strategy 2: Human Feedback Loop

```
// After each review, collect developer feedback
{
  "pr_id": "PR-12345",
  "ai_review_id": "REV-98765",
  "developer_feedback": {
    "issue_AGG-001": {"accurate": true, "helpful": true},
    "issue_AGG-002": {"accurate": false, "reason": "This is intentional f
legacy compatibility"},
    "issue_AGG-003": {"accurate": true, "helpful": false, "reason":
"Too minor, wasted time"}
  }
}
```

Use feedback to:

1. Retrain AI model with false positives/negatives
2. Adjust severity thresholds (e.g., if devs ignore "medium" issues, elevate important ones to "high
3. Refine prompts (e.g., if AI misses N+1 queries, add more examples to prompt)

Strategy 3: A/B Testing Prompts

- **Run two prompt versions** on same PRs for 2 weeks
- Compare: precision, recall, developer satisfaction scores

- Example: Test "strict security mode" vs "balanced mode" prompts
- Gradually roll out winning prompt to 100% of reviews

Strategy 4: Prompt Version Control

```yaml
# prompts/code-review-v2.3.yaml
version: "2.3"
updated: "2025-01-15"
changes:
  - "Added CWE mapping for security issues"
  - "Reduced false positives for legacy code (exclude callback patterns)"
tests_passing: 45/50
precision: 85%
recall: 92%
rollout_status: "production"
```

Maintain prompt history:

- Git version control for all prompts
- Rollback capability if new prompt performs worse
- Changelog documenting why each change was made

Part C: System Architecture & Reusability (25 points

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

- 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: System Architecture & Reusability

Question 3.1: Making the System Reusable Across Projects/Tea

1. Configuration Management

Hierarchical Configuration Model:

```yaml
# config/global-defaults.yaml
review_pipeline:
  timeout: 600  # 10 minutes
  parallel_steps: true
  required_checks: ["security", "linting", "ai_review"]

deployment:
  approval_policy: "two_approvals"
  auto_rollback_threshold:
    error_rate_increase: 0.05  # 5%
    latency_increase: 2.0      # 2x
# config/team-overrides/backend-team.yaml
review_pipeline:
  timeout: 900  # Backend needs more time for complex reviews
  coding_standards:
    language: "Python"
    max_function_length: 50
    enforce_type_hints: true
  security_rules:
    - "no-hardcoded-secrets"
    - "sql-parameterization"
    - "require-auth-decorators"
# config/project-overrides/payment-service.yaml
# Inherits from backend-team + global-defaults
deployment:
  approval_policy: "three_approvals"  # Higher risk = more approvals
  environments:
    - name: "dev"
      auto_deploy: true
    - name: "staging"
      auto_deploy: true
      smoke_tests_required: true
    - name: "prod"
      auto_deploy: false  # Manual trigger for payment service
      monitoring_duration: 7200  # 2 hours
```

Configuration Resolution Order:

1. **Global defaults** (applies to all)
2. **Team overrides** (applies to team's projects)
3. **Project overrides** (highest priority)

---

2 . Language/Framework Variations

Plugin Architecture for Language Support:

```
# plugins/base.py
class LanguagePlugin(ABC):
    @abstractmethod
    def lint(self, files: List[str]) -> LintResults:
        pass

    @abstractmethod
    def extract_dependencies(self, manifest_file: str) -> List[Dependency
        pass

    @abstractmethod
    def run_tests(self, test_command: str) -> TestResults:
        pass

# plugins/python_plugin.py
class PythonPlugin(LanguagePlugin):
    def lint(self, files):
        return run_tool("pylint", files) + run_tool("mypy", files)

    def extract_dependencies(self, manifest_file):
        # Parse requirements.txt or pyproject.toml
        return parse_python_dependencies(manifest_file)

    def run_tests(self, test_command):
        return subprocess.run(["pytest", "--cov", "--json-report"])

# plugins/javascript_plugin.py
class JavaScriptPlugin(LanguagePlugin):
    def lint(self, files):
        return run_tool("eslint", files) + run_tool("tsc", "--noEmit")

    def extract_dependencies(self, manifest_file):
        # Parse package.json
        return parse_npm_dependencies(manifest_file)

    def run_tests(self, test_command):
        return subprocess.run(["npm", "test", "--", "--coverage", "--json
```

Auto-Detection:

```
def detect_language(pr_files):
    file_extensions = {f.split(".")[-1] for f in pr_files}

    if "py" in file_extensions:
        return "python"
    elif "js" in file_extensions or "ts" in file_extensions:
        return "javascript"
    elif "go" in file_extensions:
        return "go"
    # ... etc

    # Fallback: check for manifest files
    if "package.json" in pr_files:
        return "javascript"
    elif "requirements.txt" in pr_files or "pyproject.toml" in pr_files:
        return "python"
```

---

3 . Different Deployment Targets (Cloud Providers, On-Prem)

Deployment Adapter Pattern:

```
# deployers/base.py
class DeploymentTarget(ABC):
    @abstractmethod
    def deploy(self, artifact: Artifact, environment: str) -> DeploymentR
        pass

    @abstractmethod
    def rollback(self, deployment_id: str) -> RollbackResult:
        pass

    @abstractmethod
    def get_metrics(self, service_name: str) -> Metrics:
        pass

# deployers/aws.py
class AWSDeployer(DeploymentTarget):
    def deploy(self, artifact, environment):
        # Deploy to ECS/EKS/Lambda based on service type
        if artifact.type == "container":
            return self.deploy_to_ecs(artifact, environment)
        elif artifact.type == "lambda":
            return self.deploy_lambda(artifact, environment)

    def get_metrics(self, service_name):
        # Fetch from CloudWatch
        return cloudwatch.get_metrics(service_name, metrics=["error_rate"
"latency", "cpu"])

# deployers/kubernetes.py
class KubernetesDeployer(DeploymentTarget):
    def deploy(self, artifact, environment):
        # Apply Kubernetes manifests via kubectl or Helm
        return kubectl.apply(artifact.manifest, namespace=environment)

    def get_metrics(self, service_name):
        # Fetch from Prometheus
        return
prometheus.query(f'rate(http_requests_total{{service="{service_name}"}}[5
```

```python
# deployers/on_prem.py
class OnPremDeployer(DeploymentTarget):
    def deploy(self, artifact, environment):
        # SSH to servers, rsync files, restart services
        for server in self.get_servers(environment):
            ssh.upload(artifact.path, server, "/opt/app")
            ssh.run(server, "systemctl restart app")
```

Configuration:

```yaml
# project config
deployment_target: "aws"  # or "kubernetes", "on_prem", "azure", "gcp"
deployment_config:
  aws:
    region: "us-east-1"
    ecs_cluster: "prod-cluster"
    task_definition: "my-service:latest"
  kubernetes:
    context: "prod-cluster"
    namespace: "production"
    helm_chart: "charts/my-service"
```

---

4 . Team-Specific Coding Standards

Customizable Rule Engine:

```python
# Coding standard rules defined as plugins
class CodingStandardRule(ABC):
    @abstractmethod
    def check(self, code: str, context: Dict) -> List[Violation]:
        pass

# team_standards/backend_team.py
class NoGlobalVariablesRule(CodingStandardRule):
    def check(self, code, context):
        violations = []
        ast_tree = ast.parse(code)
        for node in ast.walk(ast_tree):
            if isinstance(node, ast.Global):
                violations.append(Violation(
                    line=node.lineno,
                    message="Global variables prohibited (team standard)"
                    severity="medium"
                ))
        return violations

class RequireDocsringsRule(CodingStandardRule):
    def check(self, code, context):
        # Check that all public functions have docstrings
        # ...
```

Team Configuration:

```yaml
# config/team-standards/backend-team.yaml
coding_standards:
  rules:
    - name: "NoGlobalVariables"
      enabled: true
    - name: "RequireDocstrings"
      enabled: true
      config:
        min_function_length: 10  # Only enforce for functions >10 lines
    - name: "MaxComplexity"
      enabled: true
      config:
        max_cyclomatic_complexity: 15
```

---

5 . Industry-Specific Compliance Requirements

Compliance Framework System:

```python
# compliance/frameworks.py
class ComplianceFramework(ABC):
    @abstractmethod
    def get_required_checks(self) -> List[ComplianceCheck]:
        pass

# compliance/pci_dss.py
class PCIDSS(ComplianceFramework):
    def get_required_checks(self):
        return [
            SecurityCheck("no-hardcoded-secrets", severity="critical"),
            SecurityCheck("encrypt-sensitive-data", severity="critical"),
            SecurityCheck("log-access-to-cardholder-data", severity="high
            DeploymentCheck("require-change-approval", severity="high"),
            DeploymentCheck("automated-security-scan", severity="high"),
        ]

# compliance/hipaa.py
class HIPAA(ComplianceFramework):
    def get_required_checks(self):
        return [
            SecurityCheck("encrypt-phi-at-rest", severity="critical"),
            SecurityCheck("encrypt-phi-in-transit", severity="critical"),
            SecurityCheck("access-control-for-phi", severity="critical"),
            AuditCheck("log-phi-access", severity="high"),
            DeploymentCheck("require-security-review", severity="high"),
        ]
```

Project Configuration:

```yaml
# config/project/payment-service.yaml
compliance_frameworks:
  - "PCI-DSS"  # Credit card processing
```

```
# config/project/patient-portal.yaml
compliance_frameworks:
  - "HIPAA"  # Healthcare data
  - "SOC2"   # Security controls
```

Enforcement:

```
def enforce_compliance(pr, project_config):
    frameworks = [load_framework(f) for f in
project_config.compliance_frameworks]
    required_checks = []

    for framework in frameworks:
        required_checks.extend(framework.get_required_checks())

    # Run all required checks
    results = run_checks(pr, required_checks)

    # BLOCK merge if any critical compliance check fails
    critical_failures = [r for r in results if r.severity == "critical" a
r.status == "failed"]
    if critical_failures:
        return ReviewDecision(
            status="blocked",
            reason=f"Compliance violations: {', '.join([f.name for f in
critical_failures])}"
        )
```

Question 3.2: Continuous Improvement & Learning

1. Reducing False Positive/Negative Rates

Feedback Collection:

```
# After each review, prompt developer for feedback
@post_review
def collect_feedback(review_id, pr_id):
    # Add interactive buttons to PR comment
    github.add_comment(pr_id, f"""
    ## AI Review Complete

    Found 5 issues. Were these findings helpful?

    - Issue #1: SQL injection on line 45
      👍 Accurate | 👎 False Positive | 🤷 Not Sure

    - Issue #2: N+1 query on line 78
      👍 Accurate | 👎 False Positive | 🤷 Not Sure
    """)

# Store feedback
class ReviewFeedback:
    review_id: str
    issue_id: str
    developer_rating: Literal["accurate", "false_positive", "false_negati
"not_helpful"]
    developer_comment: Optional[str]
    timestamp: datetime
```

Model Retraining:

```
def retrain_model_weekly():
    # Collect feedback from past week
    feedback = db.query(ReviewFeedback).filter(
        ReviewFeedback.timestamp > datetime.now() - timedelta(days=7)
    )

    # Identify patterns in false positives
    false_positives = [f for f in feedback if f.developer_rating ==
"false_positive"]

    # Example: If AI consistently flags "legacy callback patterns" as iss
    # but developers mark them as false positives, add exclusion rule
    if count_pattern(false_positives, "legacy callback") > 10:
        add_exclusion_rule("ignore callback patterns in files matching **
legacy/**")

    # Fine-tune LLM with feedback examples
    training_data = [
        {"code": f.code, "expected_issues": f.actual_issues, "model_outpu
f.ai_issues}
        for f in feedback
    ]
    fine_tune_model(training_data)
```

---

2 . Learning from Deployment Success/Failure Patterns

Deployment Outcome Tracking:

```
class DeploymentOutcome:
    deployment_id: str
    pr_id: str
    environment: str
    status: Literal["success", "rollback", "partial_failure"]
    issues_detected: List[str]  # e.g., ["error_rate_spike", "memory_leak
    review_warnings_ignored: List[str]  # Issues AI flagged but were igno

# Track correlation between ignored warnings and deployment failures
def analyze_deployment_failures():
    failed_deployments = db.query(DeploymentOutcome).filter(
        DeploymentOutcome.status.in_(["rollback", "partial_failure"])
    )

    correlation = {}
    for deployment in failed_deployments:
        for warning in deployment.review_warnings_ignored:
            correlation[warning.category] = correlation.get(warning.categ
0) + 1

    # Example output: {"performance": 15, "null_pointer": 8, "thread_safe
3}
    # → Increase severity of "performance" warnings (they often cause pro
issues)

    if correlation.get("performance", 0) > 10:
        update_rule_severity("performance_warnings", from_="medium", to="
```

Predictive Deployment Risk Scoring:

```
def calculate_deployment_risk(pr):
    risk_score = 0

    # Historical pattern: PRs with >5 unresolved warnings have 40% rollba
rate
    if len(pr.unresolved_warnings) > 5:
        risk_score += 40

    # Historical pattern: Changes to auth code have 25% higher failure ra
    if any("auth" in file.path for file in pr.changed_files):
        risk_score += 25

    # Historical pattern: Friday deploys have 30% higher rollback rate
    if datetime.now().weekday() == 4:  # Friday
        risk_score += 30

    # Recommend additional testing if risk > 50
    if risk_score > 50:
        return DeploymentRecommendation(
            risk_level="high",
            actions=["Run full regression suite", "Deploy to staging for
24hrs", "Schedule on-call engineer"]
        )
```

---

3 . Learning from Developer Feedback

Feedback Analysis Dashboard:

```
# Weekly metrics
metrics = {
    "review_accuracy": {
        "precision": 0.85,  # 85% of flagged issues were real
        "recall": 0.92,     # 92% of real issues were detected
        "f1_score": 0.88
    },
    "developer_satisfaction": {
        "helpful_rate": 0.78,  # 78% of reviews marked as helpful
        "avg_time_saved": "90 minutes",
        "false_positive_rate": 0.15
    },
    "top_false_positive_categories": [
        {"category": "legacy_code_patterns", "count": 45},
        {"category": "intentional_performance_tradeoff", "count": 23}
    ]
}

# Auto-adjust prompts based on feedback
if metrics["false_positive_rate"] > 0.20:
    send_alert("High false positive rate - review prompt configuration")

if metrics["top_false_positive_categories"][0]["count"] > 30:
    category = metrics["top_false_positive_categories"][0]["category"]
    add_prompt_instruction(f"Reduce sensitivity for {category}")
```

---

4 . Production Incident Correlation

Link Reviews to Incidents:

```
class ProductionIncident:
    incident_id: str
    service: str
    timestamp: datetime
    root_cause: str
    related_pr_id: Optional[str]  # PR that introduced the bug

# Find incidents caused by code merged despite warnings
def correlate_incidents_with_reviews():
    incidents = db.query(ProductionIncident).filter(
        ProductionIncident.timestamp > datetime.now() - timedelta(days=30
    )

    for incident in incidents:
        if incident.related_pr_id:
            review = db.query(Review).filter(Review.pr_id ==
incident.related_pr_id).first()

            # Check if AI review flagged the root cause
            for warning in review.warnings:
                if warning.category == incident.root_cause:
                    # AI caught it but was ignored - increase severity fo
future
                    log_missed_catch(warning.category, incident.severity)

# Adjust blocking rules based on incidents
incident_patterns = analyze_incident_patterns()
# Example: "Memory leaks caused 5 incidents this month, all flagged as 'm
warnings"
# → Upgrade memory leak warnings to 'high' (blocking)
```

Continuous Learning Loop:

```
Developer Feedback → False Positive Reduction
        ↓
Deployment Outcomes → Risk Scoring Improvement
        ↓
Production Incidents → Severity Calibration
        ↓
Weekly Retraining → Better Reviews → [Loop back]
```

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 up

- 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: Implementation Strategy

Question 4.1: 6-Month Implementation Roadmap

MVP Definition (Month 1-2): Minimum Viable System

Scope: Core code review automation for 1 pilot team

Features Included:

- ✅ PR intake and metadata extraction
- ✅ Automated linting (ESLint/Pylint)
- ✅ Basic security scanning (Snyk/Bandit)
- ✅ Simple AI code review (GPT-4 with basic prompts)
- ✅ Review aggregation and PR commenting
- ✅ Manual approval workflow (no auto-deploy yet)
- ✅ Slack notifications

Features Excluded (for later phases):

- ❌ Deployment automation
- ❌ Advanced AI prompts
- ❌ Multi-language support (start with Python only)
- ❌ Performance analysis
- ❌ Compliance frameworks

Success Metrics for MVP:

- Review time < 2 hours for 80% of PRs (vs 2-3 days baseline)
- Developer satisfaction ≥ 70% ("helpful" rating)
- False positive rate < 25%
- Zero critical security issues slip through

Deliverables:

- Working system for Python projects
- Integration with GitHub
- Basic dashboard showing review metrics
- Documentation for developers

Phase 1 (Month 1-2): MVP Development & Pilot

Week	Activities	Deliverables
Week 1-2	- Set up infrastructure (GitHub App, AWS/cloud resources) - Implement PR webhook listener - Build linting integration (Pylint)	- GitHub App registered - Webhook receiver deploye - Linting step functional
Week 3-4	- Integrate security scanner (Bandit for Python) - Build basic AI review prompt - Implement review aggregation logic	- Security scan working - AI review returning comme - Aggregated report generat
Week 5-6	- Add PR commenting (post results to GitHub) - Build Slack notifications - Create simple metrics dashboard	- Comments appear on PRs - Slack alerts working - Dashboard shows review c avg time
Week 7-8	- **Pilot launch** with 1 backend team (5-10 engineers) - Daily standups to gather feedback - Bug fixes and UX improvements	- System processing real PR - Feedback collected - Initial metrics report

Pilot Team Selection Criteria:

- Willing to experiment and provide feedback
- Python-based projects (MVP language)
- Not working on critical production features (reduce risk)
- ~ 5-10 engineers (manageable scale)

Success Criteria for Pilot:

- Process ≥ 50 PRs successfully
- ≥ 70% developer satisfaction
- Identify and fix ≥ 10 bugs/UX issues
- Measure baseline metrics (review time, issue detection rate)

Phase 2 (Month 3-4): Expand & Enhance

Goals: Add deployment automation, support more languages, refine AI prompts

Week	Activities	Deliverables
Week 9-10	- Add JavaScript/TypeScript support (ESLint, npm audit) - Refine AI prompts based on pilot feedback - Implement test coverage analysis	- JS/TS projects supported - AI prompt v2 deployed - Coverage reports in review
Week 11-12	- Build deployment pipeline orchestration - Integrate with CI/CD (GitHub Actions) - Implement dev environment auto-deploy	- Deployment to dev automa - CI/CD integration working - Deploy button in PR UI
Week 13-14	- Add post-deployment monitoring (basic health checks) - Implement simple rollback automation - Expand to 2 more teams (frontend, DevOps)	- Monitoring step functional - Auto-rollback on errors - 3 teams using syster
Week 15-16	- Performance testing and optimization - Dashboard v2 (deployment metrics, rollback tracking) - Documentation updates	- System handles 100 week - Dashboard shows deploy s rate - Runbooks updated

Success Metrics for Phase 2:

- Review time < 1 hour for 80% of PRs
- Deployment success rate ≥ 95%
- Rollback rate < 5%
- Support 3 teams, 20-30 engineers

Phase 3 (Month 5-6): Scale & Compliance

Goals: Company-wide rollout, add compliance frameworks, advanced features

Week	Activities	Deliverables
Week 17-18	- Add compliance frameworks (PCI-DSS, SOC 2) - Implement performance analysis (N+1 query detection) - Add Go and Java language support	- Compliance checks active - Performance warnings ger - 4 languages support
Week 19-20	- Build feedback loop system (thumbs up/down on reviews) - Implement weekly model retraining - Add advanced deployment strategies (canary, blue-green)	- Feedback collection workir - Model improves weekly - Canary deploys available
Week 21-22	- **Company-wide rollout** to all 50 teams - Training sessions for engineering teams - On-call rotation for support	- All teams onboarded - Training materials publishe - Support process defined
Week 23-24	- Monitoring and optimization at scale - Cost analysis and optimization - Retrospective and roadmap planning	- System handles 500 week - Cost optimized - 6-month retrospective r

Success Metrics for Phase 3:

- Review time < 4 hours for 90% of PRs (business goal achieved)
- Security detection rate ≥ 90%
- Deployment success rate ≥ 97%
- Developer satisfaction ≥ 80%
- Cost < $10,000/month

Rollout Strategy: Gradual Expansion

```
Month 1-2:  1 team (Python)          →  10 engineers
Month 3-4:  3 teams (Python, JS)     →  30 engineers
Month 5-6:  50 teams (all languages) →  500 engineers
```

Risk Mitigation: Gradual rollout allows us to:

- Fix bugs at small scale before company-wide impact
- Refine prompts based on real usage
- Build confidence with early success stories
- Adjust infrastructure for scale

Question 4.2: Risk Mitigation Strategies

Risk 1: AI Making Incorrect Review Decisions

Probability: High (AI is not perfect)

Impact: Medium-High (developers lose trust, real issues missed)

Mitigation Strategies:

1. **Human-in-the-Loop for Critical Decisions**
    - AI provides recommendations, humans make final approval
    - Auto-block only for critical security issues (high confidence)
    - All other issues are warnings, not blockers
2. **Confidence Scoring**

```
if ai_review.confidence_score < 0.7:
    flag_for_human_review()
elif ai_review.severity == "critical" and ai_review.confidence_score <
    require_security_team_review()
```

3. **Feedback Loop with Monitoring**
    - Track false positive/negative rates daily
    - Automatic alerts if false positive rate > 20%
    - Weekly review of flagged edge cases
4. **Shadow Mode for New Features**
    - Run new AI features in "shadow mode" (log results, don't act on them)
    - Compare shadow results with human reviews for 2 weeks
    - Only enable blocking if accuracy > 85%
5. **Override Mechanism**
    - Developers can override AI decisions with justification
    - Senior engineers can approve despite AI blocks
    - Track override patterns to improve AI

Risk 2: System Downtime During Critical Deployments

Probability: Medium (systems fail)

Impact: Critical (blocks urgent hotfixes)

Mitigation Strategies:

1. **Bypass Mode for Emergencies**

   ```yaml
   # .github/bypass-ai-review
   reason: "Production outage - database down"
   approved_by: "senior-engineer@company.com"
   incident_ticket: "INC-12345"
   ```

   - Must include incident ticket
   - Requires senior engineer approval
   - Post-incident review mandatory

2. **High Availability Architecture**

   - Multi-region deployment (primary: us-east-1, failover: us-west-2)
   - Load balancer with health checks
   - Database replication and automated failover
   - **SLA**: 99.9% uptime (< 44 minutes downtime/month)

3. **Graceful Degradation**

   ```python
   if ai_service.is_down():
       # Fall back to basic checks only
       run_linting()
       run_security_scan()
       skip_ai_review()
       post_comment("AI review temporarily unavailable - proceeding with checks")
   ```

4. **Circuit Breaker Pattern**

   - If AI service fails 3x in 5 minutes, circuit opens
   - System falls back to basic review mode
   - Automatic recovery when service is healthy

5. **Incident Response Plan**

   - On-call rotation for platform team
   - Runbooks for common failures
   - PagerDuty integration for critical alerts
   - **Response SLA**: Acknowledge in 15 min, resolve in 2 hours

---

Risk 3: Integration Failures with Existing Tools

Probability: Medium (complex integrations)

Impact: Medium (delays rollout)

Mitigation Strategies:

1. **Abstraction Layers for External Tools**

   ```python
   # If GitHub API changes, only update adapter
   class GitHubAdapter:
       def get_pr_diff(self, pr_id):
           # Wraps GitHub API

   class GitLabAdapter:
       def get_pr_diff(self, mr_id):
           # Wraps GitLab API
   ```

2. **Comprehensive Integration Testing**

   - Automated tests against sandbox GitHub/GitLab instances
   - Contract tests for API integrations
   - Weekly integration test suite run
   - **Target**: > 90% integration test coverage

3. **Phased Integration**

   - Start with GitHub only (highest priority)
   - Add GitLab in Phase 2
   - Bitbucket in Phase 3 (if needed)
   - Don't block on perfect multi-platform support

4. **Vendor API Monitoring**

   - Subscribe to GitHub/GitLab API changelogs
   - Monitor for deprecated endpoints
   - Test against beta APIs before they go live

5. **Fallback to Manual Processes**

   - If integration fails, send review via email
   - Provide web UI for manual review retrieval
   - Don't block developers while fixing integration

Risk 4: Resistance from Development Teams

Probability: Medium-High (change management is hard)
Impact: High (adoption fails, project canceled)

Mitigation Strategies:

1. **Early Involvement & Co-Design**
    - Form working group with representatives from each team
    - Solicit feedback on design decisions
    - Run bi-weekly demos to show progress
    - Incorporate team suggestions (e.g., "Add Python type hint checks")
2. **Opt-In Pilot Program**
    - Teams volunteer to be early adopters (not forced)
    - Provide white-glove support during pilot
    - Showcase success stories internally
3. **Clear Value Proposition**
    - Publish metrics: "Pilot team reduced review time by 60%"
    - Show security catches: "AI detected 15 SQL injection risks"
    - Highlight time savings: "Avg 90 minutes saved per PR"
4. **Education & Training**
    - Lunch-and-learn sessions: "How AI code review works"
    - Office hours for questions
    - Documentation: "How to interpret AI feedback"
    - Video tutorials
5. **Gradual Enforcement**
    - Month 1-2: AI feedback is optional, informational only
    - Month 3-4: AI blocks critical security issues only
    - Month 5-6: Full enforcement (can still override)
6. **Developer Feedback Channel**
    - Slack channel: #ai-review-feedback
    - Monthly surveys on satisfaction

- Public roadmap showing requested features
- Fast response to complaints (< 24 hours)

Risk 5: Compliance/Audit Requirements

Probability: Medium (regulated industries)
Impact: High (legal/regulatory issues)

Mitigation Strategies:

1. **Audit Trail for All Decisions**

```
AuditLog.record(
    timestamp=datetime.utcnow(),
    action="pr_approved",
    pr_id="PR-12345",
    approver="jane.doe@company.com",
    ai_recommendation="approve",
    human_decision="approve",
    justification="All checks passed",
    compliance_frameworks=["PCI-DSS", "SOC2"]
)
```

2. **Immutable Logs**
   - Store audit logs in append-only S3 bucket
   - Object Lock enabled (cannot delete/modify)
   - Retention: 7 years (compliance requirement)
3. **Explainability for AI Decisions**
   - AI must cite reasons for each finding
   - Link to CWE/OWASP standards
   - Provide remediation steps
   - No "black box" decisions
4. **Human Approval for Compliance-Critical Systems**
   - Payment systems: Require security team approval
   - Healthcare systems: Require HIPAA compliance check
   - Financial systems: Require finance team approval
5. **Regular Compliance Audits**
   - Quarterly internal audit of AI decisions
   - Annual external audit (SOC 2)

    - Penetration testing of AI system itself
    - Review all overrides (were they justified?)

6. **Compliance Framework Integration**
    - Built-in PCI-DSS, HIPAA, SOC 2 checks
    - Auto-generate compliance reports
    - Alert on compliance violations immediately

Question 4.3: Tool Selection & Integration

Code Review Platforms

Primary: GitHub (highest priority - most teams use it)

Integration Approach:

- **GitHub App** (not webhook) - better permissions, easier auth
- **Required Permissions**:
    - Read: Repository content, pull requests, checks
    - Write: Pull request comments, commit statuses, checks
- **Webhook Events**: `pull_request` (opened, synchronized), `pull_request_review`

Secondary: GitLab, Bitbucket (Phase 2/3)

- Use adapter pattern to support multiple platforms
- Unified internal PR representation

Libraries:

```python
# GitHub
from github import Github
github_client = Github(os.getenv("GITHUB_APP_TOKEN"))

# GitLab
import gitlab
gitlab_client = gitlab.Gitlab(os.getenv("GITLAB_URL"),
private_token=os.getenv("GITLAB_TOKEN"))
```

CI/CD Systems

Primary: GitHub Actions (native integration)

Integration:

```yaml
# .github/workflows/ai-review.yml
name: AI Code Review
on: [pull_request]
jobs:
  ai-review:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Run AI Review
        uses: company/ai-review-action@v1
        with:
          pr_number: ${{ github.event.pull_request.number }}
          api_key: ${{ secrets.AI_REVIEW_API_KEY }}
```

Secondary: Jenkins, GitLab CI, CircleCI

- Provide generic webhook endpoint for any CI system
- CI system calls our API at end of build

Deployment Tools:

- **AWS**: boto 3 for ECS/Lambda deployments, CDK for infrastructure
- **Kubernetes**: kubectl, Helm charts
- **Terraform**: For multi-cloud infrastructure

Monitoring Tools

Production Metrics: Datadog (assumed current monitoring)

Integration:

```python
from datadog import initialize, api

# Fetch metrics during post-deployment monitoring
def get_deployment_metrics(service_name):
    query = f"avg:trace.flask.request.duration{{service:{service_name}}}"
    metrics = api.Metric.query(start=deploy_time, end=deploy_time + 3600,
query=query)
    return metrics
```

Alternatives: Prometheus, New Relic

- Use adapter pattern (same as deployment targets)
- Normalize metrics format internally

Internal Metrics (for AI review system itself):

- **Prometheus** + **Grafana** for system health dashboards
- Track: review latency, AI API costs, false positive rate
- Alerts: If review time > 10 min or cost >$100/day

Security Scanning Tools

Static Analysis (SAST):

- **Python**: Bandit, Semgrep
- **JavaScript**/**TypeScript**: ESLint security plugin, Semgrep
- **Go**: gosec
- **Java**: SpotBugs, Semgrep

Dependency Scanning (SCA):

- **Snyk** (commercial, good API, supports all languages)
- **npm audit** (JavaScript)
- **pip-audit** (Python)
- **OWASP Dependency-Check** (Java)

Integration:

```python
# Snyk API
import snyk
client = snyk.SnykClient(os.getenv("SNYK_TOKEN"))
vulns = client.test_python(requirements_file="requirements.txt")

# Bandit
import bandit
from bandit.core import manager
mgr = manager.BanditManager(config_file, "file")
mgr.discover_files([file_path])
mgr.run_tests()
results = mgr.get_issue_list()
```

Cost Optimization:

- Use open-source tools for basic checks (Bandit, gosec)
- Use Snyk for comprehensive SCA (pay per scan)
- Cache results for unchanged dependencies

Communication Tools

Slack (primary)

```
from slack_sdk import WebClient
slack = WebClient(token=os.getenv("SLACK_BOT_TOKEN"))

slack.chat_postMessage(
    channel="#dev-team",
    text=f"PR <{pr_url}|#{pr_number}> ready for review\n\n:white_check_ma
All checks passed\n:warning: 2 performance warnings"
)
```

Microsoft Teams (if needed)

```
import pymsteams
teams_message = pymsteams.connectorcard(os.getenv("TEAMS_WEBHOOK_URL"))
teams_message.text("PR review complete")
teams_message.send()
```

Jira (for tracking issues)

```
from jira import JIRA
jira = JIRA(server=os.getenv("JIRA_URL"), basic_auth=(user, token))

# Auto-create tickets for critical issues
if issue.severity == "critical":
    jira.create_issue(
        project="SEC",
        summary=f"Critical security issue in PR {pr_id}",
        description=issue.description,
        issuetype="Bug",
        priority="Highest"
    )
```

LLM Platform for AI Review

Primary: OpenAI GPT-4 (or GPT-4 Turbo)

```
from openai import OpenAI
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

response = client.chat.completions.create(
    model="gpt-4-turbo",
    messages=[
        {"role": "system", "content": code_review_prompt},
        {"role": "user", "content": pr_diff}
    ],
    temperature=0.1
)
```

Alternatives:

- **Anthropic Claude** (good for code, longer context)
- **AWS Bedrock** (Claude via AWS, better for enterprises)
- **Self-hosted**: CodeLlama, StarCoder (cost-effective at scale)

Cost Management:

- Use GPT-3.5 for simple PRs (<100 lines changed)
- Use GPT-4 for complex PRs (>100 lines or security-sensitive)
- Cache AI responses for identical code (avoid re-analysis)

Estimated Costs:

- GPT-4: ~$0.10 per review (3k input tokens, 500 output)
- At 500 PRs/week: ~$50/week = $200/month
- Total AI cost: < $500/month (well within budget)

Technology Stack Summary

Category	Tool	Purpose
Code Hosting	GitHub (primary), GitLab (secondary)	Source control, PR manage
CI/CD	GitHub Actions, Jenkins	Build & test automation
AI/LLM	OpenAI GPT-4, AWS Bedrock	Code review intelligence
Security Scanning	Snyk, Bandit, Semgrep	Vulnerability detection
Monitoring	Datadog, Prometheus, Grafana	Deployment metrics, system
Communication	Slack, Microsoft Teams, Jira	Notifications, issue tracking
Deployment	AWS (ECS/Lambda), Kubernetes	Production deployment
Infrastructure	Terraform, AWS CDK	Infrastructure as code
Database	PostgreSQL (audit logs), Redis (cache)	Data persistence
Orchestration	AWS Step Functions, Temporal	Workflow management

Total Estimated Cost: $5,000-8,000/month

- Cloud infrastructure: $3,000/month
- Snyk licenses: $1,500/month

- OpenAI API: $500/month
- Datadog: $2,000/month
- Slack: Free (existing)

Success Dashboard

KPIs to Track:

Metric	Baseline	Target (Month 6)	Measurement
Review Time	2-3 days	< 4 hours	Time from PR ope approval
Security Detection	~60%	≥ 90%	Critical vulns caug deploy
Deployment Success Rate	85%	≥ 97%	Deploys without ro
Developer Satisfaction	N/A	≥ 80%	Monthly survey (1 scale)
False Positive Rate	N/A	< 15%	Issues marked "no helpful"
Cost per Review	N/A	< $2	Total cost / review

Weekly Review Meetings:

- Review metrics dashboard
- Discuss edge cases and failures
- Prioritize improvements
- Celebrate wins (security catches, time saved)
