GitHub Pull Request Operations Playbook

Scaling Code Review for High-Velocity Engineering Teams

Engineering Operations Guide

Version 1.0 — January 19, 2026

Abstract

This operational playbook provides a comprehensive framework for managing pull requests (PRs) at scale in GitHub-based development workflows. As engineering teams grow and codebases expand, the pull request review process often becomes a critical bottleneck that impedes velocity and developer satisfaction. This guide synthesizes industry best practices, automation strategies, and organizational patterns to establish a sustainable, high-throughput code review operation.

The playbook addresses three fundamental pillars: **(1)** process standardization to ensure consistency and efficiency, **(2)** intelligent automation to shift cognitive load from humans to machines, and **(3)** cultural practices that foster psychological safety and continuous improvement. Organizations implementing these strategies typically observe 40–60% reduction in time-to-merge, improved code quality metrics, and enhanced developer experience.

This document is designed for engineering leaders, platform teams, and development managers responsible for establishing or optimizing code review operations at scale.

Contents

1 Executive Summary

1.1 The Challenge: Pull Requests at Scale

Modern software engineering organizations face a fundamental scaling challenge: as teams grow and release velocity increases, the traditional pull request workflow becomes a bottleneck. Left unmanaged, this manifests as:

- **Extended review latency**: PRs sitting for days waiting for review, blocking dependent work
- **Context-switching overhead**: Reviewers juggling dozens of large, complex PRs
- **Merge conflicts**: Large, long-lived PRs colliding with concurrent work
- **Quality degradation**: Rushed reviews or reviewer fatigue leading to bugs escaping to production
- **Developer frustration**: Motivation erosion when contributions languish unreviewed

1.2 The Solution Framework

This playbook prescribes a three-pillar operating model:

1. **Process Standardization** — Establish clear, repeatable patterns for PR structure, workflow, and communication
2. **Intelligent Automation** — Leverage CI/CD, bots, and tooling to handle mechanical validation and routing
3. **Cultural Foundation** — Build organizational norms that prioritize rapid iteration and psychological safety

1.3 Expected Outcomes

Organizations implementing this playbook typically achieve:

Metric	Typical Baseline	Post-Implementation
Median time-to-merge	3–5 days	4–12 hours
PR size (LOC)	500–1000+	100–300
Review iterations	3–5 rounds	1–2 rounds
Merge conflicts/week	10–20	2–5
Developer satisfaction	6/10	8–9/10

Table 1: Typical performance improvements after implementing PR operations playbook

1.4 Implementation Sequencing

For fastest return on investment, implement in this order:

1. **Quick wins (Week 1–2)**: Branch protection rules + required CI + PR templates
2. **Review routing (Week 3–4)**: CODEOWNERS implementation + automated review requests
3. **Automation layer (Month 2)**: Auto-merge policies + dependency update bots

4. **Advanced workflows (Month 3+)**: Stacked PR tooling + review operations dashboards

2 Foundational Principles

2.1 Principle 1: Small Pull Requests Are Non-Negotiable

The single most impactful intervention for PR scalability is enforcing small, incremental changes. Research and industry practice consistently demonstrate:

- **Review speed**: PRs under 200 lines are reviewed 5x faster than PRs over 500 lines
- **Defect detection**: Reviewers find 2–3x more issues in smaller PRs due to reduced cognitive load
- **Merge conflicts**: Small PRs reduce conflict probability by 60–80%
- **Rollback safety**: Small changes are easier to revert when issues arise

Guideline: Target 100–300 lines of changed code per PR. Break larger features into sequential, independently reviewable units.

2.2 Principle 2: Automation Handles Mechanics, Humans Handle Design

Effective code review is about *design validation*, not mechanical correctness. Automate all verifiable checks:

- Syntax and compilation
- Code style and formatting
- Test coverage and passage
- Security vulnerabilities
- License compliance
- Documentation completeness

Guideline: If a review comment can be generated by a linter, static analyzer, or test, it should be automated. Human reviewers focus on architecture, maintainability, and edge case reasoning.

2.3 Principle 3: Process Standardization Reduces Cognitive Overhead

Consistency eliminates decision fatigue. Standard processes for PR structure, review workflow, and merge criteria create a shared mental model across the team.

Guideline: Document explicit standards for PR creation, review expectations, approval criteria, and merge procedures. Encode these standards in templates and automation.

2.4 Principle 4: Psychological Safety Enables High Velocity

Teams that ship small PRs frequently require trust and a no-blame culture. Developers must feel safe:

- Submitting work-in-progress for early feedback

- Making mistakes that are caught in review
- Asking clarifying questions without judgment
- Proposing experimental or unconventional approaches

Guideline: Frame code review as collaborative learning, not gatekeeping. Celebrate improvement over perfection.

3 Process Strategies

3.1 PR Structure Standards

3.1.1 Optimal PR Size

Size Range	Typical Review Time	Recommendation
1–50 LOC	5–15 minutes	Ideal for bug fixes, config changes
50–200 LOC	15–30 minutes	Good for feature additions
200–400 LOC	30–60 minutes	Upper limit for maintainability
400–1000 LOC	1–3 hours	Requires decomposition
1000+ LOC	3+ hours	**Must be split**

Table 2: PR size guidelines and expected review investment

3.1.2 Decomposition Strategies

For large changes, use these patterns:

1. **Vertical slicing**: Break features into end-to-end thin slices
    - PR 1: Database schema + migrations
    - PR 2: API endpoint (minimal implementation)
    - PR 3: Business logic
    - PR 4: Frontend integration
2. **Horizontal layering**: Separate infrastructure from implementation
    - PR 1: New module structure + interfaces
    - PR 2: Core implementation
    - PR 3: Integration with existing system
    - PR 4: Tests and documentation
3. **Preparatory refactoring**: Extract groundwork into preceding PRs
    - PR 1: Extract reusable utilities
    - PR 2: Refactor existing code for extensibility
    - PR 3: Add new feature using prepared foundation

3.2 Pull Request Templates

Implement comprehensive PR templates to standardize information capture:

3.2.1 Required Template Elements

1. **Change description**
   - What changed and why
   - Link to issue/ticket/design doc
   - Context for reviewers unfamiliar with the area
2. **Risk assessment**
   - Blast radius: what could break?
   - Performance impact
   - Security implications
   - Data migration risks
3. **Test plan**
   - Unit test coverage
   - Integration test scenarios
   - Manual testing performed
   - Edge cases considered
4. **Deployment considerations**
   - Feature flags used?
   - Rollout strategy (canary, blue-green, etc.)
   - Rollback plan
   - Dependencies on other systems
5. **Reviewer checklist**
   - Code matches requirements
   - Tests are comprehensive
   - Documentation is updated
   - No obvious security issues
   - Performance impact is acceptable

See Appendix A for a complete template example.

3.3 Stacked Pull Request Workflow

For dependent changes, use a stacking workflow to maintain small PRs while enabling sequential development.

3.3.1 What Are Stacked PRs?

Stacked PRs are a series of small, dependent pull requests where each PR builds on the previous one. Instead of a single 2000-line PR, you create:

- PR 1 (→ main): Add database tables (50 LOC)
- PR 2 (→ PR 1): Add repository layer (100 LOC)
- PR 3 (→ PR 2): Add service layer (150 LOC)
- PR 4 (→ PR 3): Add API endpoints (120 LOC)

3.3.2 Benefits

- **Parallel review**: Different reviewers can work on different layers simultaneously
- **Fail fast**: Issues caught early in the stack prevent compounding problems
- **Incremental merging**: Merge and ship value as each layer is approved
- **Context preservation**: Each PR maintains focused scope

3.3.3 Tooling Options

Tool	**Approach**	**Key Features**
Graphite	Web-based + CLI	Visual stack management, auto-rebase, sync with GitHub
git-stack	CLI-only	Lightweight, no external service
stacked-PRs	GitHub Actions	Native GitHub, no CLI required
ghstack	Meta/Facebook	Heavy automation, complex setups

Table 3: Stacked PR tooling comparison

3.3.4 Workflow Procedures

1. **Create the stack**

```
# Using Graphite as example
git checkout main
git checkout -b feature/step-1
# Make changes, commit
gt stack create

git checkout -b feature/step-2
# Make changes, commit
gt stack create

git checkout -b feature/step-3
# Make changes, commit
gt stack create
```

```

# Push entire stack
gt stack push

```

2. **Review process**
   - Review PRs from bottom of stack (closest to main) first
   - Each PR approved independently
   - Merge bottom PR first, tool auto-rebases stack
3. **Handle feedback**
   - Checkout the specific branch in stack
   - Make changes, commit
   - Tool propagates changes up the stack
   - Push updates

3.4 Draft Pull Requests

Encourage liberal use of draft PRs for:

- **Early design feedback**: Share architectural approaches before full implementation
- **CI validation**: Verify tests pass before formal review request
- **Exploratory work**: Demonstrate prototypes or proof-of-concepts
- **Incremental progress**: Show work-in-progress on long-running features

Guideline: Draft PRs should be marked "Ready for Review" only when:

- All CI checks pass
- Author has self-reviewed the changes
- Description and test plan are complete
- No known issues remain

3.5 Commit Message Standards

Enforce structured commit messages for maintainability:

3.5.1 Conventional Commits Format

```
<type>(<scope>): <subject>

<body>

<footer>
```

Types: `feat`, `fix`, `docs`, `style`, `refactor`, `perf`, `test`, `chore`

Example:

```
feat(auth): implement OAuth2 token refresh

Add automatic token refresh when access token expires.
Includes retry logic with exponential backoff.

Closes #1234
```

4 Automation and Tooling

4.1 Continuous Integration Pipeline

CI/CD is the foundation of automated PR validation. Every PR must trigger:

4.1.1 Essential CI Checks

1. **Build verification**
   - Compilation succeeds
   - Dependencies resolve
   - No syntax errors
2. **Test execution**
   - Unit tests pass (100% execution)
   - Integration tests pass
   - Coverage meets threshold (e.g., 80%)
   - No flaky test failures
3. **Static analysis**
   - Linting (code style)
   - Type checking
   - Complexity analysis
   - Dead code detection
4. **Security scanning**
   - Dependency vulnerability check
   - Secret detection
   - SAST (static application security testing)
   - License compliance
5. **Performance regression**

- Benchmark comparison against baseline
- Memory leak detection
- Build time regression

4.1.2 GitHub Actions Example

```
name: Pull Request Validation

on:
  pull_request:
    types: [opened, synchronize, reopened]

jobs:
  validate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ’20’
          cache: ’npm’

      - name: Install dependencies
        run: npm ci

      - name: Lint
        run: npm run lint

      - name: Type check
        run: npm run type-check

      - name: Run tests
        run: npm test -- --coverage

      - name: Upload coverage
        uses: codecov/codecov-action@v3

      - name: Security audit
        run: npm audit --audit-level=moderate

      - name: Check bundle size
        run: npm run build && npm run check-size
```

4.2 Code Owners (CODEOWNERS)

Automate review routing and ensure domain expertise:

4.2.1 CODEOWNERS File Structure

```
# Default owners for everything
*       @org/engineering-team

# Frontend ownership
```

```
/src/frontend/            @org/frontend-team
/src/components/          @org/frontend-team @tech-lead-ui

# Backend services
/src/api/                 @org/backend-team
/src/services/auth/       @org/security-team @backend-team
/src/services/payments/   @org/payments-team @backend-team

# Infrastructure
/terraform/               @org/platform-team
/.github/workflows/       @org/devops-team
/k8s/                     @org/platform-team @org/sre-team

# Documentation
/docs/                    @org/tech-writers @tech-lead-docs
*.md                      @org/tech-writers

# Critical security files
/src/auth/                @org/security-team
/src/crypto/              @org/security-team @ciso

# Database migrations
/migrations/              @org/dba-team @backend-team
```

4.2.2 CODEOWNERS Best Practices

- **Specificity**: More specific paths override general rules
- **Required reviews**: Use branch protection to enforce CODEOWNERS approvals
- **Load balancing**: Rotate individuals in CODEOWNERS to prevent bottlenecks
- **Fallback owners**: Always specify a default owner (*)
- **Cross-training**: Include secondary reviewers to build redundancy

4.3 Branch Protection Rules

Enforce quality gates before merge:

4.3.1 Recommended Protection Rules

4.3.2 Configuration Example

```
# .github/branch-protection.yml (using API or web UI)
required_pull_request_reviews:
  dismissal_restrictions:
    users: []
    teams: []
  dismiss_stale_reviews: true
  require_code_owner_reviews: true
  required_approving_review_count: 2

required_status_checks:
  strict: true
```

Rule	**Configuration**
Require pull request	**On** — All changes must go through PR
Required approvals	**2 approvals** (adjust based on team size)
Dismiss stale approvals	**On** — Re-review after new commits
Require review from CODEOWNERS	**On** — Enforce domain expertise
Required status checks	**Specify all CI jobs** (lint, test, security)
Require branches up to date	**On** — Prevent merge conflicts
Require linear history	**On** — No merge commits (rebase or squash)
Require signed commits	**Optional** — For compliance requirements
Include administrators	**On** — No special bypass privileges
Restrict who can push	**On** — Only automated systems and admins

Table 4: Branch protection rule recommendations for main branch

```
  contexts:
    - "ci/lint"
    - "ci/test"
    - "ci/security-scan"
    - "ci/build"

enforce_admins: true
required_linear_history: true
allow_force_pushes: false
allow_deletions: false
```

4.4 Automated Merge and Bots

Reduce manual intervention for low-risk PRs:

4.4.1 Auto-Merge Policies

Define categories of PRs eligible for automated merge:

PR Type	**Criteria**	**Auto-Merge Policy**
Dependency updates	Dependabot, renovate	Auto-merge if CI passes, minor/patch only
Documentation	Only `*.md` changes	Auto-merge if 1 approval + CI pass
Config tweaks	`*.yml`, `*.json`	Auto-merge if 1 approval + CI pass
Code formatting	Only style changes	Auto-merge if CI passes
Test additions	Only `test/` directory	Auto-merge if 1 approval + tests pass

Table 5: Auto-merge policy matrix

4.4.2 Dependabot Configuration

```
# .github/dependabot.yml
version: 2
updates:
  - package-ecosystem: "npm"
    directory: "/"
    schedule:
      interval: "weekly"
      day: "monday"
      time: "09:00"
    open-pull-requests-limit: 5
    reviewers:
      - "org/frontend-team"
    labels:
      - "dependencies"
      - "auto-merge-candidate"
    commit-message:
      prefix: "chore"
      include: "scope"

  - package-ecosystem: "docker"
    directory: "/"
    schedule:
      interval: "weekly"
    reviewers:
      - "org/platform-team"
```

4.4.3 Auto-Merge GitHub Action

```
name: Auto-merge

on:
  pull_request:
    types: [labeled, synchronize]
  check_suite:
    types: [completed]

jobs:
  auto-merge:
    runs-on: ubuntu-latest
    if: |
      contains(github.event.pull_request.labels.*.name, ’auto-merge’) &&
      github.event.pull_request.user.login == ’dependabot[bot]’
    steps:
      - name: Auto-merge
        uses: pascalgn/automerge-action@v0.15.6
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          MERGE_LABELS: "auto-merge,!work-in-progress"
          MERGE_METHOD: "squash"
          MERGE_COMMIT_MESSAGE: "pull-request-title"
          MERGE_RETRIES: 6
          MERGE_RETRY_SLEEP: 10000
```

4.5 Code Quality Thresholds

Block PRs that degrade quality metrics:

4.5.1 Coverage Thresholds

```
# jest.config.js (for JavaScript/TypeScript)
module.exports = {
  coverageThreshold: {
    global: {
      branches: 80,
      functions: 80,
      lines: 80,
      statements: 80
    },
    ’./src/critical/’: {
      branches: 95,
      functions: 95,
      lines: 95,
      statements: 95
    }
  }
};
```

4.5.2 Complexity Limits

```
# .eslintrc.json
{
  "rules": {
    "complexity": ["error", { "max": 10 }],
    "max-depth": ["error", 4],
    "max-lines": ["error", { "max": 300, "skipBlankLines": true }],
    "max-lines-per-function": ["error", { "max": 50 }]
  }
}
```

5 Organizational Culture and Guidelines

5.1 Psychological Safety Framework

5.1.1 No-Blame Code Review Principles

1. **Assume good intent**: Reviewers presume authors made reasonable decisions with available context

2. **Ask questions, don’t make demands**: Frame feedback as inquiry ("Could we consider...?") rather than directives

3. **Separate person from code**: Critique the implementation, never the implementer

4. **Public praise, private criticism**: Celebrate good work openly; address concerns in PR comments, not public channels

5. **Learn together**: Treat reviews as mutual learning opportunities

5.1.2 Example Feedback Patterns

5.2 CONTRIBUTING.md Guidelines

Establish a comprehensive contributor guide:

Avoid	Prefer
"This is wrong"	"I'm concerned this might cause X. Could we discuss alternatives?"
"Why didn't you...?"	"Would it make sense to also consider...?"
"You should know better"	"For future reference, our team convention is..."
"This is terrible"	"This approach works, but I wonder if [alternative] might be more maintainable because..."

Table 6: Code review feedback anti-patterns vs. constructive patterns

5.2.1 Required Sections

1. **Getting started**
    - Development environment setup
    - How to run tests locally
    - How to build the project
2. **Pull request process**
    - PR size expectations
    - Required PR template usage
    - How to request reviews
    - Expected review turnaround time
3. **Code standards**
    - Coding style guide (link to style guide)
    - Naming conventions
    - Comment and documentation requirements
    - Test coverage expectations
4. **Review standards**
    - What reviewers should look for
    - Approval criteria
    - How to handle disagreements
5. **Merge criteria**
    - Required approvals
    - CI passing requirements
    - How conflicts are resolved

See Appendix B for a complete template.

5.3 Review Service Level Objectives (SLOs)

Define explicit expectations for review latency:

5.3.1 Recommended SLOs

PR Type	First Review	Final Approval
Hotfix / Critical	1 hour	2 hours
Small PR (<200 LOC)	4 business hours	1 business day
Medium PR (200–400 LOC)	1 business day	2 business days
Large PR (>400 LOC)	*Should be decomposed*	
Documentation-only	1 business day	2 business days
Dependency updates	Automated	Automated

Table 7: Review SLO guidelines by PR type

5.3.2 Monitoring and Accountability

- Track SLO compliance in team dashboards
- Identify bottleneck reviewers
- Rotate review responsibilities to prevent burnout
- Escalate SLO violations to engineering management

5.4 Review Rotation and Capacity Planning

Distribute review workload equitably:

5.4.1 Rotation Strategies

1. **Round-robin**: Assign reviews sequentially across team members
2. **Expertise-based**: Route to domain experts via CODEOWNERS, with fallback rotation
3. **Load-balanced**: Assign to reviewer with fewest active review requests
4. **Hybrid**: Combine expertise routing with load balancing

5.4.2 Capacity Allocation

Reserve explicit time for code review:

- **Daily review windows**: Block 30–60 minutes for review
- **Review-only days**: Designate one day/week for focused review work
- **Interrupt-driven**: Respond to review requests within SLO windows

Guideline: Senior engineers should allocate 20–30% of time to code review; mid-level engineers 10–20%.

6 Implementation Roadmap

6.1 Phase 1: Foundation (Weeks 1–2)

6.1.1 Objectives

- Prevent bad merges with automated validation
- Standardize PR structure
- Establish baseline metrics

6.1.2 Action Items

Week	Task	Owner	Success Metric
1	Configure branch protection on main	Platform team	Protection enabled
1	Implement basic CI (lint, test, build)	DevOps team	All PRs run CI
1	Create PR template	Tech lead	Template adoption >80%
1–2	Document PR size guidelines	Tech lead	Published in wiki
2	Set up code coverage reporting	DevOps team	Coverage visible on PRs
2	Establish baseline metrics	Engineering manager	Dashboard created

Table 8: Phase 1 implementation tasks

6.1.3 Metrics to Track

- PRs created per day
- Median PR size (LOC)
- Time to first review
- Time to merge
- CI failure rate

6.2 Phase 2: Intelligent Routing (Weeks 3–4)

6.2.1 Objectives

- Automate review assignment
- Ensure domain expertise in reviews
- Reduce review latency

6.2.2 Action Items

Week	Task	Owner	Success Metric
3	Define code ownership boundaries	Tech leads	CODEOWNERS file
3	Implement CODEOWNERS file	Platform team	Auto-assignment works
3	Configure required CODEOWNERS review	Platform team	Protection enforced
4	Set up review load balancing	DevOps team	Balanced distribution
4	Document review SLOs	Engineering manager	SLOs published

Table 9: Phase 2 implementation tasks

6.2.3 Metrics to Track

- Review assignment accuracy (correct expertise)
- Review load distribution (Gini coefficient)
- SLO compliance percentage

6.3 Phase 3: Automation Layer (Weeks 5–8)

6.3.1 Objectives

- Automate low-risk PRs
- Reduce manual merge operations
- Improve dependency update cadence

6.3.2 Action Items

Week	Task	Owner	Success Metric
5	Configure Dependabot	DevOps team	Weekly updates
5	Define auto-merge policies	Tech leads	Policy doc
6	Implement auto-merge action	Platform team	Auto-merge working
6	Set up dependency auto-merge	DevOps team	Deps merge without manual intervention
7	Add security scanning to CI	Security team	Vulns blocked
8	Implement PR size enforcement	Platform team	Large PRs warned

Table 10: Phase 3 implementation tasks

6.3.3 Metrics to Track

- Percentage of PRs auto-merged
- Dependency update cycle time

- Security vulnerabilities caught pre-merge

6.4 Phase 4: Advanced Workflows (Weeks 9–12)

6.4.1 Objectives

- Enable stacked PR workflows for complex changes
- Implement review operations dashboards
- Establish continuous improvement processes

6.4.2 Action Items

Week	Task	Owner	Success Metric
9	Evaluate stacked PR tooling	Tech leads	Tool selected
10	Pilot stacked PRs with one team	Platform team	5 successful stacks
11	Build review ops dashboard	DevOps team	Dashboard live
11	Document stacked PR workflow	Tech writer	Docs published
12	Conduct retrospective	Engineering manager	Action items identified
12	Roll out stacked PRs org-wide	Platform team	All teams enabled

Table 11: Phase 4 implementation tasks

6.4.3 Metrics to Track

- Stacked PR adoption rate
- Complex change cycle time (vs. baseline)
- Developer satisfaction (survey)

6.5 Ongoing Operations

6.5.1 Weekly Activities

- Review SLO compliance dashboard
- Triage stale PRs (>5 days old)
- Review load balancing across team

6.5.2 Monthly Activities

- Review metrics trends
- Update CODEOWNERS as needed
- Refine auto-merge policies based on learnings

6.5.3 Quarterly Activities

- Developer satisfaction survey
- Process retrospective
- Benchmark against industry standards

7 Operational Procedures

7.1 Daily Review Operations

7.1.1 Morning Review Triage (15 minutes)

1. **Check review dashboard**
   - Identify SLO violations
   - Note blocked PRs
   - Review queue depth
2. **Prioritize review queue**
   - Hotfixes first
   - Small PRs (<100 LOC) next
   - Oldest PRs within SLO window
   - Large PRs last (after confirming they shouldn't be split)
3. **Assign urgent reviews**
   - Manually assign critical reviews if auto-assignment missed
   - Escalate SLO violations to team leads

7.1.2 Review Execution Best Practices

1. **Pre-review**
   - Read PR description and test plan
   - Check CI status
   - Review commit history
   - Identify potential risks
2. **During review**
   - Start with high-level architecture
   - Check for security issues
   - Verify test coverage
   - Review error handling

   - Check documentation updates

3. **Post-review**
   - Summarize key feedback
   - Indicate blocking vs. non-blocking comments
   - Set expectations for next steps
   - Approve or request changes explicitly

7.2 Handling Stale Pull Requests

7.2.1 Definition

A PR is considered stale when:

- No activity for 5+ business days
- Author has not responded to feedback in 3+ business days
- Merge conflicts exist for 2+ business days

7.2.2 Stale PR Triage Process

1. **Identify stale PRs** (automated bot labels)
2. **Categorize by reason**
   - Awaiting author response
   - Awaiting reviewer response
   - Merge conflicts
   - CI failures
   - Unclear next steps
3. **Take action**
   - **Author blockers**: Ping author, offer to pair
   - **Reviewer blockers**: Reassign or escalate
   - **Conflicts**: Notify author, offer rebase assistance
   - **Unclear**: Triage with tech lead
4. **Close abandoned PRs**
   - No response after 2 pings over 10 days
   - Comment with reason and offer to reopen

7.3 Handling Large Pull Requests

7.3.1 Detection

Automatically flag PRs with:

- >400 lines changed
- >50 files changed
- >10 commits

7.3.2 Response Protocol

1. **Immediate feedback** (within 1 hour)
   - Comment on PR: "This PR is large. Let's discuss decomposition strategy."
   - Suggest meeting or pairing session
2. **Decomposition workshop**
   - Review with author
   - Identify independently reviewable units
   - Plan sequence of smaller PRs
3. **Options**
   - Close large PR, create stack of small PRs
   - If truly atomic, schedule dedicated review session with multiple reviewers
   - For emergency/urgent changes, fast-track with senior reviewer pair

7.4 Conflict Resolution

7.4.1 Technical Disagreements

When reviewer and author disagree on approach:

1. **Document both perspectives**
   - Author explains reasoning
   - Reviewer explains concern
2. **Escalation path**
   - Involve tech lead or architect
   - Schedule synchronous discussion
   - Document decision and rationale
3. **Resolution criteria**
   - Security/correctness concerns: reviewer judgment prevails
   - Style/preference: defer to existing patterns or CODEOWNERS

- Architecture: tech lead or architect decides
- If urgent: senior engineer makes call, document for future discussion

7.4.2 Process Violations

When PR violates established guidelines:

1. **Automated enforcement** (preferred)
   - CI fails for violations
   - PR cannot merge until resolved
2. **Manual enforcement**
   - Reviewer politely points to guideline
   - Request changes
   - If pattern persists, discuss with engineer and manager

8 Metrics and Continuous Improvement

8.1 Key Performance Indicators

8.1.1 Velocity Metrics

Metric	Target	Measurement
Time to first review	<4 hours (small PRs)	P50, P90, P99
Time to merge	<24 hours (small PRs)	P50, P90, P99
PRs merged per day	Baseline + 20%	Daily average
PRs created per day	Baseline + 30%	Daily average
Merge frequency	Multiple per day	Per developer

Table 12: Velocity KPIs

8.1.2 Quality Metrics

Metric	Target	Measurement
CI pass rate	>95% first attempt	Percentage
Test coverage	>80% overall	Per PR delta
Code review comments	3–8 per PR	Average
Reverts/rollbacks	<2% of merges	Percentage
Security issues caught	100% in PR	Pre-merge detections

Table 13: Quality KPIs

8.1.3 Efficiency Metrics

8.1.4 Experience Metrics

Metric	**Target**	**Measurement**
PR size	<200 LOC median	P50, P90
Review iterations	<2 rounds	Average rounds to merge
Merge conflicts	<5% of PRs	Percentage
Auto-merge rate	>30% eligible PRs	Percentage
Stale PR rate	<10%	PRs open >5 days

Table 14: Efficiency KPIs

Metric	**Target**	**Measurement**
Developer satisfaction	>8/10	Quarterly survey
Review load balance	Gini <0.3	Distribution
SLO compliance	>90%	Percentage
Reviewer burnout	<20% high load	Self-reported

Table 15: Experience KPIs

8.2 Dashboard Implementation

8.2.1 Essential Visualizations

1. **PR flow diagram**
   - PRs opened, reviewed, merged per day
   - Cumulative flow diagram
2. **Cycle time trends**
   - Time to first review (trend over weeks)
   - Time to merge (trend over weeks)
   - Breakdown by PR size
3. **SLO compliance**
   - Percentage meeting review SLO
   - Percentage meeting merge SLO
   - Violations by reviewer
4. **Review workload**
   - Active reviews per person
   - Review comments per person
   - Time spent in review (if tracked)
5. **Quality indicators**
   - CI pass rate trend
   - Code coverage trend
   - Revert rate

8.2.2 Tooling Options

- **GitHub Insights**: Native GitHub analytics
- **LinearB**: Engineering intelligence platform
- **Swarmia**: Engineering effectiveness dashboard
- **Jellyfish**: Engineering management platform
- **Custom dashboards**: Grafana + GitHub API

8.3 Retrospective Cadence

8.3.1 Monthly Team Retrospective

Agenda (60 minutes):

1. Review metrics vs. targets (10 min)
2. Discuss top friction points (20 min)
3. Identify successful patterns (15 min)
4. Propose process experiments (10 min)
5. Commit to action items (5 min)

Focus questions:

- Which PRs took longest? Why?
- Where did we exceed SLOs? What worked?
- What caused most review iterations?
- Which automations saved the most time?
- What's still painful?

8.3.2 Quarterly Process Review

Deep-dive analysis:

1. Trend analysis across 3 months
2. Benchmark against initial baseline
3. Compare against industry standards
4. Developer satisfaction survey review
5. Major process changes assessment

Outputs:

- Updated process documentation
- Revised SLOs if needed
- New automation opportunities identified

- Training needs assessment

8.4 Continuous Improvement Experiments

8.4.1 A/B Testing Process Changes

1. **Hypothesis**: "Reducing required approvals from 2 to 1 for small PRs will decrease time-to-merge by 30% without increasing revert rate"
2. **Experiment design**
   - Control group: Team A (2 approvals)
   - Treatment group: Team B (1 approval for <100 LOC PRs)
   - Duration: 2 weeks
3. **Metrics**
   - Time to merge (primary)
   - Revert rate (safety check)
   - Developer satisfaction (survey)
4. **Decision criteria**
   - Adopt if time-to-merge improves >20% AND revert rate doesn't increase >10%

9 Troubleshooting Common Issues

9.1 Problem: Review Latency Increasing

9.1.1 Symptoms

- Time-to-first-review >8 hours
- SLO violations increasing
- Developer complaints about waiting

9.1.2 Diagnosis

1. Check review load distribution
   - Is one person a bottleneck?
   - Are CODEOWNERS too concentrated?
2. Analyze PR characteristics
   - Are PRs getting larger?
   - More complex changes?
   - Insufficient context in descriptions?
3. Review team capacity

  - Team members on PTO?
  - Increased PR volume?
  - Other priorities consuming time?

9.1.3 Solutions

- **Short-term**
  - Temporarily reduce approval requirements
  - Assign backup reviewers
  - Escalate urgent PRs to management
- **Long-term**
  - Expand CODEOWNERS to more team members
  - Implement review rotation
  - Reinforce PR size guidelines
  - Add more automated validation

9.2 Problem: High Merge Conflict Rate

9.2.1 Symptoms

- >10% of PRs have merge conflicts
- Developers spending significant time rebasing
- Frustration with "conflict hell"

9.2.2 Diagnosis

1. Identify conflict hotspots
   - Which files conflict most?
   - Are conflicts in shared utilities?
   - Architectural coupling issues?
2. Analyze PR patterns
   - Are PRs too large?
   - Too many PRs touching same areas?
   - Long-lived feature branches?

9.2.3 Solutions

- **Immediate**
  - Enable "require branches up to date" protection

  - Implement auto-rebase bot
  - Coordinate work on high-conflict areas
- **Structural**
  - Refactor hotspot files for better modularity
  - Use feature flags instead of long-lived branches
  - Adopt stacked PRs for dependent changes
  - Reduce PR size to minimize conflict window

9.3 Problem: Low Auto-Merge Adoption

9.3.1 Symptoms

- <10% of eligible PRs auto-merge
- Manual intervention still required
- Dependency updates pile up

9.3.2 Diagnosis

1. Check auto-merge configuration
   - Are labels applied correctly?
   - CI passing consistently?
   - Branch protection conflicts?
2. Review eligibility criteria
   - Too strict policies?
   - Missing automation for eligible categories?

9.3.3 Solutions

- Expand auto-merge categories
- Improve CI stability
- Auto-label eligible PRs
- Document auto-merge policies clearly

9.4 Problem: Quality Regressions

9.4.1 Symptoms

- Increased bug reports from production
- Rising revert rate
- Failed deployments

9.4.2 Diagnosis

1. Analyze reverted PRs
   - What was missed in review?
   - Test coverage gaps?
   - Rushed reviews?
2. Review velocity vs. quality tradeoff
   - Are reviewers sacrificing thoroughness for speed?
   - Insufficient approval requirements?

9.4.3 Solutions

- **Process tightening**
  - Increase approval requirements temporarily
  - Add mandatory security/architecture reviews for certain areas
  - Implement pre-merge integration testing
- **Automation strengthening**
  - Expand test coverage requirements
  - Add more static analysis rules
  - Implement regression test requirements
- **Cultural reinforcement**
  - Review postmortems for reverted PRs
  - Share learnings widely
  - Celebrate catching issues in review

10 Advanced Topics

10.1 Monorepo-Specific Strategies

Monorepos require special considerations for PR operations at scale:

10.1.1 Challenges

- Large checkout sizes
- Slow CI due to breadth
- Difficult to scope reviews
- Cross-team coordination

10.1.2 Solutions

1. **Affected path detection**

```yaml
# GitHub Action to run tests only for affected packages
name: Selective CI
on: [pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0
      - name: Get changed packages
        id: packages
        run: |
          CHANGED=$(git diff --name-only HEAD^ HEAD | grep -oP '^packages/\K[^/]+' | sort -u)
          echo "packages=$CHANGED" >> $GITHUB_OUTPUT
      - name: Run tests
        run: |
          for pkg in ${{ steps.packages.outputs.packages }}; do
            npm test --workspace=$pkg
          done
```

2. **CODEOWNERS granularity**

```
# Monorepo CODEOWNERS with package ownership
/packages/frontend/      @org/frontend-team
/packages/backend/       @org/backend-team
/packages/shared/        @org/platform-team @org/frontend-team @org/backend-team
/packages/mobile/        @org/mobile-team
/libs/                   @org/platform-team
/.github/                @org/devops-team
```

3. **Scoped reviews**

   - Label PRs by affected packages
   - Route reviews based on package
   - Allow parallel reviews of independent packages

10.2 Cross-Repository Dependencies

Managing PRs across multiple repositories:

10.2.1 Coordination Patterns

1. **Synchronized merges**

   - Create PRs in dependent repos simultaneously
   - Link PRs in descriptions

   - Merge in dependency order
   - Use GitHub linking: “Depends on org/repo#123”

2. **Backward compatibility window**
   - Maintain compatibility for N versions
   - Allows independent deployment
   - Deprecate old APIs over time
3. **Feature flags for cross-repo features**
   - Ship code in both repos independently
   - Enable feature once both deployed
   - Roll back single repo if needed

10.3 Security-Sensitive Repositories

Additional PR considerations for high-security codebases:

10.3.1 Enhanced Controls

- **Mandatory security review**
  - Designated security team approval required
  - Automated threat modeling
  - Secret scanning enforcement
- **Signed commits**
  - Require GPG/SSH signatures
  - Verify commit identity
  - Audit trail for compliance
- **Two-person rule**
  - Minimum 2 approvals for production code
  - At least one from security team
  - No self-merge allowed
- **Automated compliance checks**
  - SAST (Semgrep, CodeQL)
  - Dependency vulnerability scanning
  - Secrets detection (Gitleaks, TruffleHog)
  - License compliance

10.4 Open Source Repository Considerations

Managing external contributions:

10.4.1 Contributor Experience

- Clear CONTRIBUTING.md with expectations
- First-time contributor welcome bot
- Detailed PR template for external contributors
- Gracious feedback on rejected PRs

10.4.2 Trust and Verification

- Required CLA (Contributor License Agreement)
- Enhanced review for external PRs
- Fork-based workflow
- Limit CI resources for forks (prevent abuse)

A Pull Request Template Example

```
## Description
<!-- Provide a clear and concise description of your changes -->

**What changed:**

**Why it changed:**

**Related issues/tickets:**
<!-- Link to GitHub issues, Jira tickets, design docs, etc. -->
Closes #

## Type of Change
<!-- Check all that apply -->
- [ ] Bug fix (non-breaking change which fixes an issue)
- [ ] New feature (non-breaking change which adds functionality)
- [ ] Breaking change (fix or feature that would cause existing
   functionality to not work as expected)
- [ ] Refactoring (no functional changes)
- [ ] Documentation update
- [ ] Configuration change

## Risk Assessment
<!-- Help reviewers understand potential impact -->

**Blast radius:**
<!-- What could break? Which users/systems are affected? -->

**Performance impact:**
<!-- Does this change performance? Better/worse? By how much? -->
```

```
**Security implications:**
<!-- Are there authentication, authorization, or data privacy concerns?
   -->

**Database changes:**
<!-- Schema changes? Data migrations? Backward compatible? -->

## Testing
<!-- Describe the tests you ran and how to reproduce -->

**Test coverage:**
- [ ] Unit tests added/updated
- [ ] Integration tests added/updated
- [ ] End-to-end tests added/updated

**Test scenarios covered:**
1.
2.
3.

**Manual testing performed:**
<!-- What did you test manually? Include steps to reproduce -->

**Edge cases considered:**
<!-- What edge cases did you think about? -->

## Deployment Considerations

**Feature flags:**
<!-- Is this behind a feature flag? Which one? -->

**Rollout strategy:**
<!-- Canary? Blue-green? All at once? -->

**Rollback plan:**
<!-- How do we roll back if something goes wrong? -->

**Dependencies:**
<!-- Does this depend on other services, configs, or deployments? -->

**Monitoring:**
<!-- What metrics/logs should we watch? -->

## Documentation
<!-- Check all that apply -->
- [ ] Code comments added/updated
- [ ] README updated
- [ ] API documentation updated
- [ ] Architecture decision record (ADR) created
- [ ] No documentation needed

## Reviewer Checklist
<!-- Reviewers: please confirm before approving -->
```

```
- [ ] Code matches requirements and design
- [ ] Tests are comprehensive and pass
- [ ] No obvious security vulnerabilities
- [ ] Performance impact is acceptable
- [ ] Documentation is updated
- [ ] Error handling is appropriate
- [ ] Code follows team standards and conventions

## Screenshots/Videos
<!-- If applicable, add screenshots or videos to demonstrate changes -->

## Additional Context
<!-- Add any other context or information reviewers should know -->
```

B CONTRIBUTING.md Template

```
# Contributing to [Project Name]

Thank you for considering contributing to our project! This document
   outlines our development process and expectations.

## Table of Contents
- [Getting Started](#getting-started)
- [Development Workflow](#development-workflow)
- [Pull Request Process](#pull-request-process)
- [Code Standards](#code-standards)
- [Review Standards](#review-standards)
- [Communication](#communication)

## Getting Started

### Prerequisites
- Node.js 20+
- npm 10+
- Git 2.40+

### Setup
‘‘‘bash
# Clone the repository
git clone https://github.com/org/repo.git
cd repo

# Install dependencies
npm install

# Run tests
npm test

# Start development server
npm run dev
‘‘‘

### Running Tests
```

~~~~
```bash
# Unit tests
npm test

# Integration tests
npm run test:integration

# Test coverage
npm run test:coverage

# Lint
npm run lint

# Type check
npm run type-check
```

## Development Workflow

### Branching Strategy
- `main` - production-ready code
- `feature/[name]` - new features
- `fix/[name]` - bug fixes
- `docs/[name]` - documentation updates

### Commit Messages
We follow [Conventional Commits](https://www.conventionalcommits.org/):

```
<type>(<scope>): <subject>

<body>

<footer>
```

**Types:**
- `feat` - New feature
- `fix` - Bug fix
- `docs` - Documentation
- `style` - Formatting, no code change
- `refactor` - Code restructuring
- `perf` - Performance improvement
- `test` - Adding tests
- `chore` - Maintenance

**Example:**
```
feat(auth): add OAuth2 token refresh

Implement automatic token refresh with exponential backoff
retry logic.

Closes #123
~~~~

```
‘‘‘

## Pull Request Process

### Before Creating a PR

1. **Ensure CI passes locally**
   ‘‘‘bash
   npm run lint
   npm test
   npm run type-check
   ‘‘‘

2. **Rebase on latest main**
   ‘‘‘bash
   git fetch origin
   git rebase origin/main
   ‘‘‘

3. **Self-review your changes**
   - Read through the diff
   - Remove debug code
   - Check for console.logs or commented code

### PR Size Guidelines
- **Target:** 100-300 lines of changed code
- **Maximum:** 400 lines (larger PRs require justification)
- **If >400 lines:** Discuss decomposition strategy with team

### PR Template
Use the provided PR template and fill out all sections:
- Description and context
- Risk assessment
- Test plan
- Deployment considerations
- Reviewer checklist

### Review SLOs
- **First review:** Within 4 business hours for small PRs
- **Approval:** Within 1 business day for small PRs
- **Hotfixes:** Within 1 hour

## Code Standards

### Style Guide
- Follow existing code style
- Use Prettier for formatting (runs on commit)
- Use ESLint rules (blocking CI failures)

### File Organization
‘‘‘
src/
  components/      # React components
  services/        # Business logic
  utils/           # Shared utilities
  types/           # TypeScript types
  __tests__/       # Tests co-located with code
‘‘‘

### Testing Requirements
- **Unit test coverage:** >80% for new code
- **Integration tests:** For API endpoints and critical paths
- **Test naming:** ‘describe‘ blocks for grouping, ‘it‘ for specific behaviors

**Example:**
‘‘‘typescript
describe(’UserService’, () => {
  describe(’createUser’, () => {
    it(’should create user with valid data’, () => {
      // test
    });

    it(’should throw error for duplicate email’, () => {
      // test
    });
  });
});
‘‘‘

### Documentation
- **Code comments:** For complex logic, not obvious code
- **JSDoc:** For public APIs and exported functions
- **README:** Update if adding features or changing setup

## Review Standards

### What Reviewers Look For
1. **Correctness:** Does it work? Does it meet requirements?
2. **Design:** Is the approach sound? Is it maintainable?
3. **Tests:** Are edge cases covered? Are tests meaningful?
4. **Security:** Any vulnerabilities? Data validation?
5. **Performance:** Any obvious inefficiencies?
6. **Documentation:** Are changes documented?

### Providing Feedback
- **Be respectful and constructive**
- **Ask questions rather than making demands**
  -     "Could we consider using X here because Y?"
  -     "This is wrong, use X instead"
- **Distinguish blocking vs. non-blocking comments**
  - Blocking: "This has a security issue"
  - Non-blocking: "Consider renaming for clarity"
- **Approve explicitly:** Use GitHub’s review feature

### Receiving Feedback
- **Assume good intent**
- **Ask for clarification if needed**
- **Respond to all comments** (even if just acknowledging)
- **Don't take it personally** - code review is about the code, not you

### Handling Disagreements
1. **Discuss in the PR** - explain your reasoning
2. **If unresolved** - involve tech lead or architect
3. **Document decision** - explain why approach was chosen
4. **Move forward** - don't let disagreements block progress unnecessarily

## Communication

### Getting Help
- **Slack:** #engineering channel for quick questions
- **GitHub Discussions:** For design discussions
- **Office Hours:** Tuesdays 2-3pm with tech leads

### Reporting Issues
Use GitHub Issues with:
- Clear title
- Steps to reproduce
- Expected vs. actual behavior
- Environment details

### Code of Conduct
- Be respectful and professional
- Welcome new contributors
- Focus on constructive feedback
- Assume good intent

## Review Process Details

### Approval Requirements
- **2 approvals** from team members
- **1 approval from CODEOWNERS** for specific files
- **All CI checks passing**
- **No unresolved discussions**

### Merge Process
1. Ensure all feedback is addressed
2. Get required approvals
3. Ensure CI is green
4. **Squash merge** (default) or rebase
5. Delete branch after merge

## Questions?

If you have questions about the contribution process, reach out to:
- Slack: #engineering
- Email: engineering@example.com
- GitHub Discussions

Thank you for contributing!
```

C GitHub Actions CI/CD Examples

C.1 Comprehensive PR Validation

```
name: PR Validation

on:
  pull_request:
    types: [opened, synchronize, reopened, ready_for_review]

# Cancel previous runs when new commits pushed
concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true

jobs:
  # Skip checks for draft PRs
  check-draft:
    runs-on: ubuntu-latest
    outputs:
      is-draft: ${{ steps.draft-check.outputs.is-draft }}
    steps:
      - id: draft-check
        run: |
          if [ "${{ github.event.pull_request.draft }}" = "true" ]; then
            echo "is-draft=true" >> $GITHUB_OUTPUT
          else
            echo "is-draft=false" >> $GITHUB_OUTPUT
          fi

  # Validate PR size
  pr-size-check:
    runs-on: ubuntu-latest
    needs: check-draft
    if: needs.check-draft.outputs.is-draft == ’false’
    steps:
      - uses: actions/checkout@v4

      - name: Check PR size
        uses: actions/github-script@v7
        with:
          script: |
            const pr = context.payload.pull_request;
            const additions = pr.additions;
            const deletions = pr.deletions;
            const totalChanges = additions + deletions;

            const warningThreshold = 400;
            const errorThreshold = 1000;

            if (totalChanges > errorThreshold) {
              core.setFailed(
                ‘PR is too large (${totalChanges} LOC). ‘ +
                ‘Please consider splitting into smaller PRs. ‘ +
                ‘See CONTRIBUTING.md for guidelines.‘
              );
            } else if (totalChanges > warningThreshold) {
              core.warning(
                ‘PR is large (${totalChanges} LOC). ‘ +
                `Consider splitting for faster review.`
              );
            }

  # Lint and format check
  lint:
    runs-on: ubuntu-latest
    needs: check-draft
    if: needs.check-draft.outputs.is-draft == 'false'
    steps:
      - uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Run ESLint
        run: npm run lint

      - name: Check formatting
        run: npm run format:check

  # Type checking
  type-check:
    runs-on: ubuntu-latest
    needs: check-draft
    if: needs.check-draft.outputs.is-draft == 'false'
    steps:
      - uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Type check
        run: npm run type-check

  # Unit and integration tests
  test:
    runs-on: ubuntu-latest
    needs: check-draft
    if: needs.check-draft.outputs.is-draft == 'false'
    steps:
      - uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Run tests
        run: npm test -- --coverage --ci

      - name: Upload coverage
        uses: codecov/codecov-action@v3
        with:
          token: ${{ secrets.CODECOV_TOKEN }}
          files: ./coverage/lcov.info
          flags: unittests
          fail_ci_if_error: true

      - name: Check coverage threshold
        run: |
          npm run test:coverage-check || {
            echo "Coverage below threshold"
            exit 1
          }

  # Security scanning
  security:
    runs-on: ubuntu-latest
    needs: check-draft
    if: needs.check-draft.outputs.is-draft == 'false'
    steps:
      - uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Run npm audit
        run: npm audit --audit-level=moderate

      - name: Run Snyk security scan
        uses: snyk/actions/node@master
        env:
          SNYK_TOKEN: ${{ secrets.SNYK_TOKEN }}
        with:
          args: --severity-threshold=high

      - name: Secret scanning
        uses: trufflesecurity/trufflehog@main
        with:
          path: ./
          base: ${{ github.event.pull_request.base.sha }}
          head: ${{ github.event.pull_request.head.sha }}

  # Build check
  build:
    runs-on: ubuntu-latest
    needs: check-draft
    if: needs.check-draft.outputs.is-draft == ’false’
    steps:
      - uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: ’20’
          cache: ’npm’

      - name: Install dependencies
        run: npm ci

      - name: Build
        run: npm run build

      - name: Check bundle size
        uses: andresz1/size-limit-action@v1
        with:
          github_token: ${{ secrets.GITHUB_TOKEN }}

  # All checks passed
  all-checks:
    runs-on: ubuntu-latest
    needs: [pr-size-check, lint, type-check, test, security, build]
    if: always()
    steps:
      - name: Check if all jobs succeeded
        run: |
          if [ "${{ contains(needs.*.result, ’failure’) }}" = "true" ]; then
            echo "Some checks failed"
            exit 1
          fi
          echo "All checks passed!"
```

C.2 Auto-Merge for Dependabot

```
name: Dependabot Auto-Merge

on:
  pull_request:
    types: [opened, synchronize]

permissions:
  pull-requests: write
  contents: write

jobs:
  auto-merge:
    runs-on: ubuntu-latest
    if: |
      github.actor == ’dependabot[bot]’ &&
      github.event_name == ’pull_request’
    steps:
      - name: Fetch PR metadata
        id: metadata
    uses: dependabot/fetch-metadata@v1
    with:
     github-token: ${{ secrets.GITHUB_TOKEN }}

   - name: Auto-approve minor and patch updates
    if: |
     steps.metadata.outputs.update-type == 'version-update:semver-minor' ||
     steps.metadata.outputs.update-type == 'version-update:semver-patch'
    run: |
     gh pr review --approve "$PR_URL"
    env:
     PR_URL: ${{ github.event.pull_request.html_url }}
     GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}

   - name: Enable auto-merge for minor and patch
    if: |
     steps.metadata.outputs.update-type == 'version-update:semver-minor' ||
     steps.metadata.outputs.update-type == 'version-update:semver-patch'
    run: |
     gh pr merge --auto --squash "$PR_URL"
    env:
     PR_URL: ${{ github.event.pull_request.html_url }}
     GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}

   - name: Comment on major updates
    if: steps.metadata.outputs.update-type == 'version-update:semver-major'
    uses: actions/github-script@v7
    with:
     script: |
      github.rest.issues.createComment({
       issue_number: context.issue.number,
       owner: context.repo.owner,
       repo: context.repo.repo,
       body: '    This is a **major version update**. Please review changelog and test thoroughly before merging.'
      })
```

D Metrics Dashboard Query Examples

D.1 GitHub GraphQL Queries

D.1.1 PR Velocity Metrics

```
query PRVelocityMetrics($owner: String!, $repo: String!, $since: DateTime!) {
 repository(owner: $owner, name: $repo) {
  pullRequests(
   first: 100
   orderBy: {field: CREATED_AT, direction: DESC}
   states: [MERGED]
   baseRefName: "main"
  ) {
   nodes {
    number
    title
    createdAt
    mergedAt
    additions
```

```
        deletions
        changedFiles
        reviews(first: 10) {
          totalCount
          nodes {
            submittedAt
            state
          }
        }
        timelineItems(first: 1, itemTypes: [READY_FOR_REVIEW_EVENT]) {
          nodes {
            ... on ReadyForReviewEvent {
              createdAt
            }
          }
        }
        commits {
          totalCount
        }
      }
      pageInfo {
        hasNextPage
        endCursor
      }
    }
  }
}
```

D.1.2 Review Workload Distribution

```
query ReviewWorkload($owner: String!, $repo: String!, $since: DateTime!) {
  repository(owner: $owner, name: $repo) {
    pullRequests(
      first: 100
      orderBy: {field: CREATED_AT, direction: DESC}
      states: [MERGED, OPEN]
    ) {
      nodes {
        reviews(first: 50) {
          nodes {
            author {
              login
            }
            submittedAt
            state
            comments {
              totalCount
            }
          }
        }
        reviewRequests(first: 10) {
          nodes {
            requestedReviewer {
              ... on User {
                login
              }
            }
          }
        }
      }
    }
  }
}
```

D.2 Analysis Scripts

D.2.1 Calculate Time-to-Merge Statistics

```python
import json
from datetime import datetime
import statistics

def analyze_pr_velocity(prs):
    """Calculate PR velocity metrics"""
    time_to_first_review = []
    time_to_merge = []
    pr_sizes = []

    for pr in prs:
        created = datetime.fromisoformat(
            pr['createdAt'].replace('Z', '+00:00')
        )
        merged = datetime.fromisoformat(
            pr['mergedAt'].replace('Z', '+00:00')
        )

        # Time to merge
        ttm = (merged - created).total_seconds() / 3600  # hours
        time_to_merge.append(ttm)

        # PR size
        size = pr['additions'] + pr['deletions']
        pr_sizes.append(size)

        # Time to first review
        reviews = pr['reviews']['nodes']
        if reviews:
            first_review = min(
                datetime.fromisoformat(r['submittedAt'].replace('Z', '+00:00'))
                for r in reviews
            )
            ttfr = (first_review - created).total_seconds() / 3600
            time_to_first_review.append(ttfr)

    return {
        'time_to_merge': {
            'median': statistics.median(time_to_merge),
            'p90': statistics.quantiles(time_to_merge, n=10)[8],
            'p99': statistics.quantiles(time_to_merge, n=100)[98],
        },
        'time_to_first_review': {
            'median': statistics.median(time_to_first_review),
            'p90': statistics.quantiles(time_to_first_review, n=10)[8],
        },
        'pr_size': {
            'median': statistics.median(pr_sizes),
            'p90': statistics.quantiles(pr_sizes, n=10)[8],
        },
        'total_prs': len(prs),
    }

# Usage
with open('pr_data.json') as f:
    data = json.load(f)
    prs = data['data']['repository']['pullRequests']['nodes']
    metrics = analyze_pr_velocity(prs)
    print(json.dumps(metrics, indent=2))
```

E References and Further Reading

E.1 Industry Research

- **DORA Metrics**. "Accelerate: State of DevOps Reports" (2019–2024). Comprehensive research on software delivery performance.
- **Google Engineering Practices**. "Code Review Developer Guide" (2020). Best practices from Google's engineering org.
- **Microsoft Research**. "Code Reviews Do Not Find Bugs: How the Current Code Review Best Practice Slows Us Down" (2015). Analysis of 1M+ code reviews.
- **SmartBear**. "Best Practices for Peer Code Review" (2024). Industry survey of 600+ developers.

E.2 Tools and Platforms

- **Graphite**: https://graphite.dev — Stacked PR workflows
- **GitHub Actions**: https://docs.github.com/actions — CI/CD automation
- **Dependabot**: https://github.com/dependabot — Dependency updates
- **CodeQL**: https://codeql.github.com — Static analysis and security
- **Codecov**: https://codecov.io — Code coverage tracking

E.3 Books

- Forsgren, N., Humble, J., & Kim, G. (2018). *Accelerate: The Science of Lean Software and DevOps*. IT Revolution Press.

- Kim, G., Debois, P., Willis, J., & Humble, J. (2016). *The DevOps Handbook.* IT Revolution Press.
- Winters, T., Manshreck, T., & Wright, H. (2020). *Software Engineering at Google.* O'Reilly Media.