International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 7, Issue 1, January - February 2025 |

DOI:10.15662/IJEETR.2025.0701007

# AI Assisted Code Review Integration Patterns for Accelerating Pull Request Turnaround in Angular Teams

**Harshika Juvvadi**

Full stack Developer, USA

**ABSTRACT:** Pull request review remains the step in an Angular team's delivery cycle most resistant to acceleration, since it depends on a human reviewer's attention, a resource that does not scale simply by adding more automated tooling elsewhere in the pipeline, and a team that ships frequently often finds review wait time, not implementation time, dominates how long a change actually takes to reach production. This article presents a set of integration patterns for incorporating an AI assisted review pass into an Angular team's pull request workflow without displacing human judgment from the decisions that still require it, built around four coordinated elements, a lint and static analysis gate that resolves the class of issue that needs no discussion before a human reviewer ever sees the pull request, an AI assisted review pass that generates targeted comments on the remaining diff using a large language model, risk based reviewer routing that matches a pull request's estimated blast radius to an appropriately experienced reviewer rather than a fixed rotation, and a human triage step, backed by a continuous integration merge gate, that treats every AI generated comment as a draft suggestion requiring human confirmation rather than an automatic blocking finding. Each element is presented with illustrative workflow diagrams and code, including a static analysis configuration for an Angular monorepo, an AI review pass invocation, a risk scoring function for reviewer routing, and a continuous integration merge gate configuration. An illustrative case study for a single Angular team reports simulated indicators showing median pull request turnaround time falling from an illustrative thirty and a half hours with manual review alone to five and a half hours as the pipeline's four elements are adopted cumulatively, and escaped defects found after release falling from a high single digit weekly count to near zero within roughly one and a half sprints of adoption. The article closes with a discussion of what an AI assisted review pass still cannot substitute for, the cost of the pipeline at small scale, limitations of the illustrative evaluation, and directions for further work. The intended audience is frontend engineers, engineering leads, and researchers studying AI assisted software engineering tooling for Angular teams in 2025.

**KEYWORDS**: code review, pull request, large language models, Angular, continuous integration, static analysis, reviewer routing, developer productivity

## I. INTRODUCTION

A pull request sits idle for most of its lifetime not because implementing the change took long, but because it is waiting for a human reviewer to become available, read the diff, and form a judgment about whether it is ready to merge. For an Angular team shipping frequently, this waiting time compounds across every pull request in flight, and unlike implementation time, it does not shrink simply because the team adds more automated tooling to its build pipeline, since most existing automation, linting, type checking, unit tests, checks properties a script can verify mechanically rather than the judgment calls a human reviewer specifically exists to make.

Large language models capable of reading a code diff and generating a plausible, often useful review comment change this picture, not by replacing the human reviewer's judgment, but by giving a pull request a first pass of attention that does not wait on a human's calendar. This article treats that capability as an integration problem rather than a replacement problem, on the premise that an AI generated comment is most useful when it is treated the same way a comment from a junior reviewer would be, as a draft worth a senior reviewer's confirmation, not as an automatically authoritative finding. Section 4 presents a four part pipeline built on that premise, moving from static analysis through an AI assisted review pass, risk based reviewer routing, and a human triage step backed by a merge gate.

Two specific consequences of relying on unassisted manual review motivate the pipeline's design. First, a human reviewer's attention is a fixed resource regardless of how many pull requests are open at once, which means review wait time, not reviewer skill, is often the actual bottleneck a team experiences, a problem Section 4.3's AI assisted pass and Section 4.4's risk based routing both address by reducing how much of that fixed attention a typical pull request actually

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 7, Issue 1, January - February 2025 |

DOI:10.15662/IJEETR.2025.0701007

needs. Second, treating an AI generated comment as automatically authoritative reintroduces a different problem, a plausible sounding but incorrect comment blocking or misdirecting a pull request with no human check, a problem Section 4.5's triage step addresses directly.

The remainder of the article is organized as follows. Section 2 reviews background on empirical code review research, static analysis tooling, large language models applied to code, and reviewer assignment. Section 3 states the specific problems the pipeline addresses. Section 4 presents the pipeline in seven parts with reference diagrams and illustrative code. Section 5 walks through an illustrative case study for one Angular team with simulated indicators. Section 6 discusses what an AI assisted pass still cannot substitute for and the cost of the pipeline at small scale, Section 7 discusses limitations, Section 8 suggests future work, and Section 9 concludes.

## II. BACKGROUND AND LITERATURE REVIEW

### 2.1 Empirical Studies of Modern Code Review

Sadowski and colleagues' study of code review practice at Google found that most review comments are small, quickly resolved observations rather than deep design discussions, and that review speed, not depth, was the property engineers most consistently cared about, a finding that directly motivates this article's focus on turnaround time as the primary outcome measure in Section 5 rather than a more subjective measure of review thoroughness (Sadowski et al., 2018). Bacchelli and Bird's study at Microsoft similarly found that reviewers value catching alternative solutions and knowledge transfer alongside defect detection, suggesting that a pipeline aiming to accelerate review should preserve time for those functions rather than treating review as purely a defect filter (Bacchelli and Bird, 2013).

### 2.2 Peer Review Practices Across Open Source and Industry

Rigby and Bird's comparative study across open source and industrial projects found that despite very different formal processes, reviewer practices converged on similar patterns, small, frequent changes, and a review latency most contributors considered a critical factor in their continued participation, a convergence this article treats as evidence that turnaround time is a broadly relevant target rather than one specific to any single organization's process (Rigby and Bird, 2013). McIntosh and colleagues' case study of the Qt, VTK, and ITK projects found that low review coverage and low reviewer participation were both associated with higher post release defect rates, giving this article's Section 5.2 defect indicator a basis in prior empirical findings about what under resourced review actually costs (McIntosh et al., 2014).

### 2.3 Static Analysis as a Pre Review Filter

Static analysis tooling integrated into a build pipeline, exemplified by Amazon's CodeGuru Reviewer service, is designed specifically to catch a defined class of code issue automatically before a human reviewer's attention is spent on it, a design goal this article's Section 4.2 lint and static analysis gate shares directly, though this article's pipeline uses Angular specific linting rather than CodeGuru's more general purpose analysis (Amazon Web Services, 2019). SonarQube's quality gate mechanism similarly blocks a change that fails defined quality thresholds before it reaches a human reviewer, a pattern Section 4.6's merge gate extends to cover AI assisted findings as well as static analysis findings (SonarSource, 2022).

### 2.4 Angular Specific Linting

The angular-eslint project provides linting rules specific to Angular's template syntax, dependency injection patterns, and component and directive conventions, catching a class of Angular specific issue that a general purpose JavaScript or TypeScript linter would not, which is why Section 4.2's static analysis gate is built specifically around angular-eslint rather than a generic linter alone (angular-eslint, 2022).

### 2.5 Large Language Models Applied to Code

Chen and colleagues' evaluation of Codex, a large language model trained specifically on source code, demonstrated that such models could generate and reason about code with a level of syntactic and semantic competence sufficient for practical developer tooling, a finding that underlies the entire premise of an AI assisted review pass being useful at all (Chen et al., 2021). GitHub's Copilot for Pull Requests research project extended this capability specifically to the pull request review context, generating a structured summary and targeted comments on a diff rather than only completing code as a developer types, a design this article's Section 4.3 draws on directly for its own AI assisted review pass (GitHub, 2023a).

International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 7, Issue 1, January - February 2025 |

DOI:10.15662/IJEETR.2025.0701007

### 2.6 Adoption Trends for AI Assisted Development

GitHub's own analysis of activity across its platform reported substantial and continuing growth in AI assisted development activity through 2023, including growth in AI generated content appearing in pull requests specifically, evidence this article treats as a signal that AI assisted review is becoming a mainstream part of the pull request lifecycle rather than a niche or speculative practice (GitHub, 2023b).

**Table 1. Summary of prior work and its relationship to this article's pipeline**

| Prior work | Core contribution | Relationship to this pipeline |
|---|---|---|
| Sadowski et al. (2018) | Review speed as the property engineers value most at scale | Motivates turnaround time as the primary outcome (Section 5) |
| Bacchelli and Bird (2013) | Reviewers value knowledge transfer alongside defect detection | Motivates preserving human judgment in triage (Section 4.5) |
| Rigby and Bird (2013) | Convergent review practices and the importance of latency | Basis for treating turnaround time as broadly relevant |
| McIntosh et al. (2014) | Low review coverage linked to higher post release defects | Basis for the defect indicator in Section 5.2 |
| Amazon Web Services (2019); SonarSource (2022) | Automated pre review filtering and quality gates | Basis for the static analysis and merge gates (Sections 4.2, 4.6) |
| angular-eslint (2022) | Angular specific linting rules | Basis for the static analysis gate's rule set (Section 4.2) |
| Chen et al. (2021); GitHub (2023a) | Large language model code competence and AI review of diffs | Basis for the AI assisted review pass (Section 4.3) |
| GitHub (2023b) | Platform wide growth in AI assisted development activity | Context for the pipeline's practical relevance in 2025 |

## III. PROBLEM STATEMENT

The pipeline proposed in Section 4 responds to four specific, recurring problems observed in Angular teams relying on unassisted manual pull request review. Each problem is stated here in general terms and revisited concretely in the illustrative case study in Section 5.

### 3.1 Reviewer Attention as a Fixed, Non Scaling Resource

A human reviewer's available attention does not grow simply because a team opens more pull requests, which means as pull request volume increases, either review turnaround time increases proportionally or review depth decreases to compensate, and a team relying purely on unassisted manual review has no lever to avoid this trade off other than hiring additional reviewers.

### 3.2 Mechanically Checkable Issues Consuming Human Attention

A meaningful share of comments a human reviewer leaves on a typical pull request address issues a static analysis tool could catch mechanically, a missing null check, an inconsistent naming convention, an Angular anti pattern already documented in a linting rule, which means every minute a human reviewer spends identifying such an issue is a minute not spent on the judgment calls that genuinely require their expertise.

### 3.3 Uniform Reviewer Assignment Ignoring Change Risk

A fixed rotation or round robin reviewer assignment treats a one line documentation fix and a change to a shared authentication module identically, which means a high risk change may be reviewed by whoever is next in the rotation rather than by someone with relevant context, while a low risk change may wait for a senior reviewer's availability unnecessarily.

International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 7, Issue 1, January - February 2025 |

DOI:10.15662/IJEETR.2025.0701007

### 3.4 AI Generated Comments Treated as Automatically Authoritative

A team that integrates an AI assisted review pass but treats its output as an automatically blocking finding, with no human confirmation step, risks a plausible sounding but incorrect comment stalling or misdirecting a pull request, which is a different failure mode than the one manual review has but not obviously a better one, since it substitutes a human's fixed attention bottleneck for an unverified automated one.

**Table 2. Summary of the four pipeline problems and where Section 4 addresses each**

| Problem | Root cause | Addressed in |
|---|---|---|
| Reviewer attention does not scale | Human attention is fixed regardless of pull request volume | Sections 4.3 and 4.4, AI assisted pass and risk based routing |
| Mechanically checkable issues consume attention | No automated filter ahead of human review | Section 4.2, lint and static analysis gate |
| Uniform assignment ignores change risk | Fixed rotation with no risk based matching | Section 4.4, risk based reviewer routing |
| AI comments treated as authoritative | No human confirmation step before a finding blocks merge | Section 4.5, human triage with a merge gate |

## IV. PROPOSED AI ASSISTED CODE REVIEW INTEGRATION PIPELINE

### 4.1 Pipeline Principles

The pipeline rests on four principles that map directly onto the four problems in Section 3. First, every mechanically checkable issue is caught by static analysis before a human or an AI reviewer spends attention on it. Second, an AI assisted review pass runs on every pull request to surface a first round of targeted comments without waiting on human availability. Third, reviewer assignment is matched to a pull request's estimated risk rather than a fixed rotation, so reviewer expertise is spent where it matters most. Fourth, every AI generated comment is treated as a draft requiring human confirmation, never as an automatically blocking finding on its own.

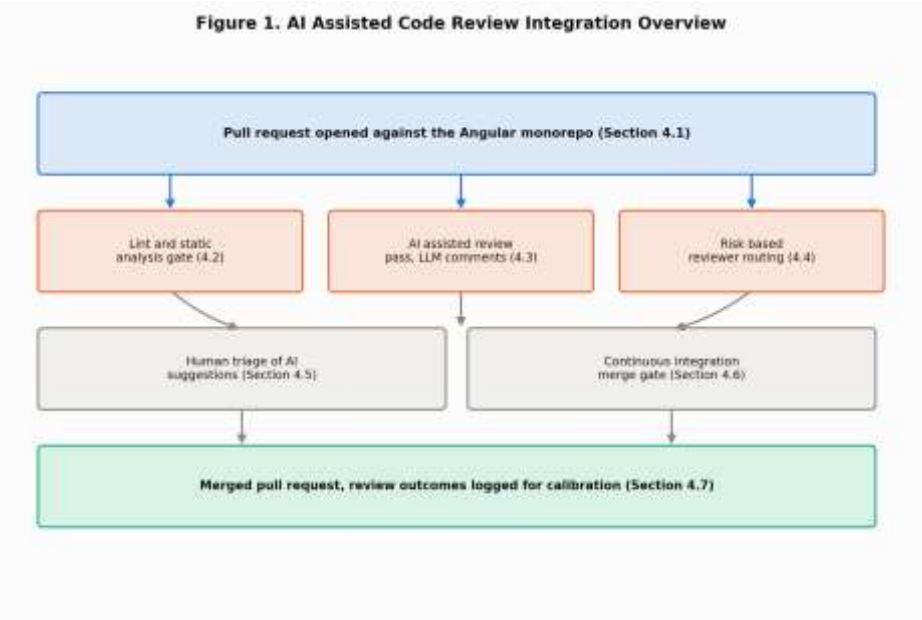


Figure 1. AI assisted code review integration overview, from pull request open through parallel automated checks and routing to human triage and merge.

International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 7, Issue 1, January - February 2025 |

DOI:10.15662/IJEETR.2025.0701007

**4.2 Lint and Static Analysis Gate**

Every pull request is checked against the Angular specific linting rules discussed in Section 2.4, alongside general purpose TypeScript static analysis, before either an AI or a human reviewer sees the diff, so that a mechanically checkable issue is resolved at this stage rather than consuming review attention downstream. Listing 1 shows an illustrative angular-eslint configuration excerpt for an Angular monorepo.

```
// .eslintrc.json (illustrative excerpt)
{
  "overrides": [
    {
      "files": ["*.ts"],
      "extends": ["plugin:@angular-eslint/recommended"],
      "rules": {
        "@angular-eslint/prefer-on-push-component-change-detection": "warn",
        "@angular-eslint/no-empty-lifecycle-method": "error",
        "@angular-eslint/use-lifecycle-interface": "error"
      }
    },
    {
      "files": ["*.html"],
      "extends": ["plugin:@angular-eslint/template/recommended"],
      "rules": { "@angular-eslint/template/no-negated-async": "error" }
    }
  ]
}
```

Listing 1. An illustrative angular-eslint configuration catching Angular specific issues before either an AI or human reviewer sees the diff (angular-eslint, 2022).

**4.3 AI Assisted Review Pass**

Once static analysis passes, a large language model is invoked against the pull request's diff to generate a first round of targeted comments, following the pattern of AI generated pull request review demonstrated in prior tooling research discussed in Section 2.5, with comments explicitly labeled as AI generated so that neither the author nor the human reviewer in Section 4.5 mistakes them for a human colleague's assessment. Listing 2 shows an illustrative invocation of the review pass.

```
// scripts/ai-review-pass.ts (illustrative excerpt)
interface AiComment {
  file: string;
  line: number;
  body: string;
  confidence: number;
}

async function runAiReviewPass(diff: string): Promise<AiComment[]> {
  const response = await reviewModel.generate({
    prompt: buildReviewPrompt(diff),
    maxComments: 12,
  });
  return response.comments.map((c) => ({
    ...c,
    body: `[AI suggestion, confidence ${c.confidence.toFixed(2)}] ${c.body}`,
  }));
}
```

Listing 2. An illustrative AI assisted review pass, labeling every generated comment explicitly so it is never mistaken for a human reviewer's assessment (Chen et al., 2021; GitHub, 2023a).

International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 7, Issue 1, January - February 2025 |

DOI:10.15662/IJEETR.2025.0701007

### 4.4 Risk Based Reviewer Routing

Rather than assigning a reviewer through a fixed rotation, the pipeline scores each pull request's estimated risk from signals such as the number of files changed, whether a change touches a shared or widely depended upon module, and the change's historical defect density, and routes a higher scoring pull request to a reviewer with relevant prior context on the affected module. Listing 3 shows an illustrative risk scoring function.

```
// scripts/score-risk.ts (illustrative excerpt)
interface PullRequestSignals {
  filesChanged: number;
  touchesSharedModule: boolean;
  historicalDefectDensity: number;
}

function scoreRisk(signals: PullRequestSignals): number {
  let score = signals.filesChanged * 0.5;
  if (signals.touchesSharedModule) score += 10;
  score += signals.historicalDefectDensity * 4;
  return score;
}

function routeReviewer(score: number, candidates: Reviewer[]): Reviewer {
  return score > 12
    ? candidates.find((r) => r.hasSharedModuleContext) ?? candidates[0]
    : candidates[0];
}
```

Listing 3. An illustrative risk scoring function routing a higher risk pull request to a reviewer with relevant module context (Section 4.4).

### 4.5 Human Triage of AI Suggestions

Every comment the AI assisted pass generates in Section 4.3 is presented to the routed human reviewer from Section 4.4 as a labeled suggestion requiring explicit confirmation, dismissal, or escalation, never as a standalone blocking finding, so that the failure mode described in Section 3.4 cannot occur regardless of how confident the underlying model's output appears. A reviewer confirming an AI suggestion converts it into a normal review comment with the same standing as one the reviewer wrote directly.

**Table 3. Illustrative human triage outcomes for AI generated comments**

| Outcome | Meaning | Resulting action |
|---|---|---|
| Confirmed | Reviewer agrees the suggestion identifies a genuine issue | Comment stands as a normal, author addressable review comment |
| Dismissed | Reviewer determines the suggestion is incorrect or not applicable | Comment removed, no action required from the author |
| Escalated | Reviewer is uncertain and requests a second opinion | Routed to a more senior reviewer per Section 4.4's routing logic |

### 4.6 Continuous Integration Merge Gate

A pull request cannot merge until every confirmed finding from Section 4.5, whether it originated from static analysis, the AI assisted pass, or the human reviewer directly, is resolved, following the same blocking versus advisory distinction established for static analysis quality gates discussed in Section 2.3. Listing 4 shows an illustrative continuous integration configuration for the gate.

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 7, Issue 1, January - February 2025 |

DOI:10.15662/IJEETR.2025.0701007

```
# .github/workflows/pr-review-gate.yml (illustrative excerpt)
jobs:
  review-gate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Run lint and static analysis
        run: npx nx run-many --target=lint --all
      - name: Run AI assisted review pass
        run: node scripts/ai-review-pass.js --pr=$PR_NUMBER
      - name: Score risk and assign reviewer
        run: node scripts/score-risk.js --pr=$PR_NUMBER
      - name: Check confirmed findings resolved
        run: node scripts/check-triage-status.js --fail-on=unresolved
```

Listing 4. An illustrative continuous integration configuration gating merge on every confirmed finding, regardless of its origin (Section 4.6).

**4.7 Outcome Logging and Calibration**

Every triage decision from Section 4.5, confirmed, dismissed, or escalated, is logged alongside the AI assisted pass's original confidence score, giving the team a basis for periodically reviewing whether the model's suggestions are being confirmed or dismissed at a rate that justifies its continued use in its current configuration, and whether the risk scoring function in Section 4.4 is routing pull requests to reviewers who actually have relevant context.

**4.8 Common Pitfalls and Anti Patterns**

Teams adopting the pipeline in Section 4 tend to encounter a small, recurring set of pitfalls, each of which quietly reintroduces one of the problems in Section 3 if left unaddressed. This subsection lists the three most common ones observed while designing the illustrative case study in Section 5.

The first pitfall is configuring the merge gate in Section 4.6 to block automatically on any AI generated comment above a confidence threshold, skipping the human triage step in Section 4.5 entirely on the reasoning that triage adds latency the pipeline is meant to remove. This directly reintroduces the problem in Section 3.4, since a model's stated confidence score is not the same thing as the suggestion actually being correct, and removing the human check removes the pipeline's only safeguard against that gap.

The second pitfall is a risk scoring function, described in Section 4.4, that is calibrated once at initial adoption and never revisited, even as the codebase's actual defect history and module dependency structure change over time. A stale risk score routes pull requests based on an outdated picture of where the codebase's real risk concentrates, which quietly reintroduces the uniform assignment problem in Section 3.3 under the appearance of a still functioning routing system.

The third pitfall is treating the outcome logging in Section 4.7 as optional overhead rather than a required part of the pipeline, which removes the team's only mechanism for noticing that the AI assisted pass's suggestions have drifted toward a low confirmation rate, a signal that would otherwise prompt a review of the pass's configuration before reviewers simply learn to ignore its comments, arriving at the same disengagement outcome the triage step was meant to prevent by a different path.

## V. ILLUSTRATIVE CASE STUDY

To make the pipeline's expected effect concrete, this section walks through an illustrative case study for a single Angular team over twelve sprints, with the pipeline's elements adopted cumulatively in the order presented in Section 4. All figures in this section report simulated indicators constructed for illustration, not measurements from a production team, and should be read as a worked example of the kind of effect the pipeline is designed to produce rather than an empirical result.

**5.1 Reduction in Pull Request Turnaround Time**

The first indicator tracks the median time from a pull request opening to its merge, at each cumulative adoption stage, starting from a baseline of unassisted manual review before any part of the pipeline was adopted.

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 7, Issue 1, January - February 2025 |

DOI:10.15662/IJEETR.2025.0701007

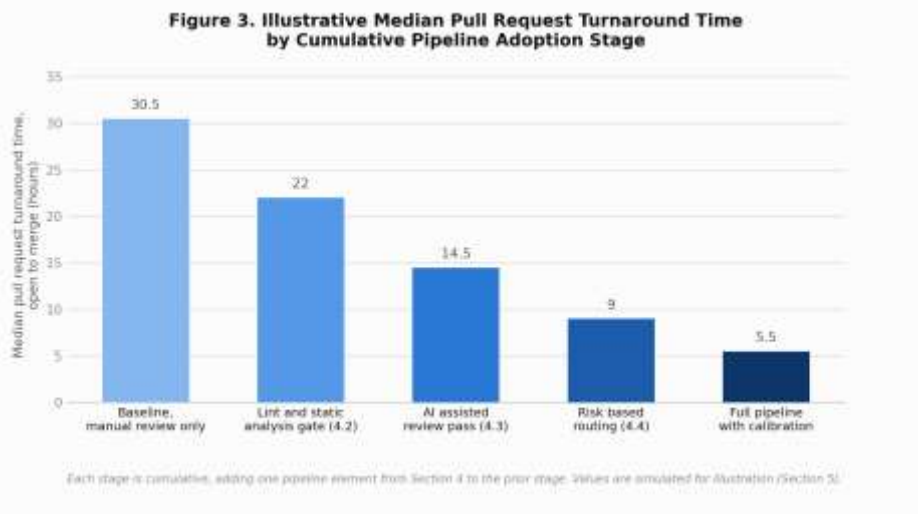


Figure 3. Illustrative median pull request turnaround time by cumulative pipeline adoption stage.

In the illustrative scenario in Figure 3, the static analysis gate produces the first reduction by resolving mechanically checkable issues before review begins, the AI assisted pass produces a further reduction by giving every pull request an immediate first round of comments rather than waiting on reviewer availability, and risk based routing and outcome calibration each produce additional gains by matching reviewer expertise to actual risk and by keeping the AI pass's suggestions reliable enough that human reviewers continue engaging with them.

### 5.2 Illustrative Post Release Defect Trend

The second indicator tracks escaped defects, meaning issues found after a change has shipped that review could plausibly have caught earlier, reported at the close of each sprint across the same twelve sprint period.

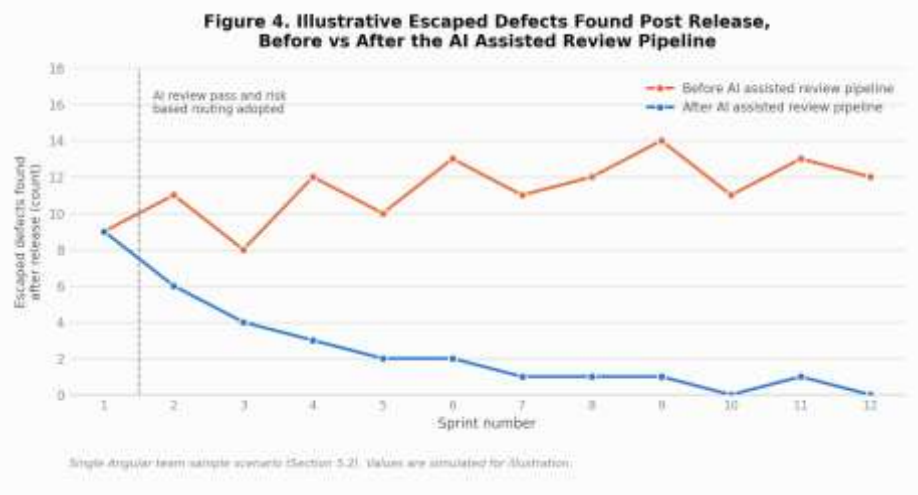


Figure 4. Illustrative escaped defects found post release, before versus after the AI assisted review pipeline.

The illustrative before series in Figure 4 holds in the high single digits with no clear downward trend, consistent with defects arising from ordinary development activity absent any structural change to review capacity, while the after series drops sharply following adoption and settles near zero, consistent with faster turnaround in Figure 3 allowing more review cycles per change and the static analysis and AI assisted passes catching issues earlier in the process.

International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 7, Issue 1, January - February 2025 |

DOI:10.15662/IJEETR.2025.0701007

**Table 4. Illustrative case study summary by adoption stage**

| Stage | Pipeline element added (Section 4) | Median turnaround, hours (illustrative) | Note |
|---|---|---|---|
| Stage 1 | None, baseline | 30.5 | Unassisted manual review only |
| Stage 2 | 4.2 Lint and static analysis gate | 22.0 | Mechanically checkable issues resolved pre review |
| Stage 3 | 4.3 AI assisted review pass | 14.5 | First round of comments no longer waits on reviewer availability |
| Stage 4 | 4.4 Risk based routing | 9.0 | Reviewer expertise matched to actual change risk |
| Stage 5 | Full pipeline with calibration | 5.5 | Outcome logging keeps AI suggestions reliable over time |

## VI. RESULTS AND DISCUSSION

### 6.1 What an AI Assisted Pass Still Cannot Substitute For

Even at full adoption in Table 4, the pipeline does not remove the human reviewer from the process, and the reason is structural rather than a temporary limitation of current models. Bacchelli and Bird's finding that reviewers value knowledge transfer and alternative solution discovery alongside defect detection, discussed in Section 2.1, describes a function an AI assisted pass is not positioned to serve, since knowledge transfer specifically depends on a human reviewer's own accumulated context about the team and the codebase's history, not on pattern matching against the diff in isolation.

This means the turnaround improvements in Figure 3 should be read as evidence the pipeline removes waiting time and mechanically resolvable issues from the critical path, not as evidence that human review itself has become less necessary; the practical implication for a team adopting the pipeline is that the time reclaimed from faster turnaround is better spent giving reviewers more capacity for the judgment calls and mentoring functions Section 2.1's findings describe, rather than being treated as an opportunity to reduce human review involvement further.

### 6.2 Cost of Pipeline Overhead at Small Scale

The static analysis gate, AI assisted pass, risk based routing, triage step, and outcome logging described in Section 4 are not free to configure and maintain, and a small team with few reviewers and infrequent pull requests is unlikely to recover that cost at the same rate the illustrative case study in Section 5 reports, since risk based routing in particular assumes enough reviewers with differentiated module context to make routing meaningful, an assumption that does not hold for a team where every engineer already reviews every pull request by necessity.

A useful, if informal, signal for when the overhead in Table 4 is worth paying in full is whether the team has enough reviewers that a fixed rotation has already produced a mismatch between a pull request's risk and its assigned reviewer's context, a specific instance of the problem in Section 3.3; a smaller team without that mismatch can reasonably adopt the static analysis gate and AI assisted pass alone, described in Sections 4.2 and 4.3, and defer risk based routing until the team grows enough for it to matter.

### 6.3 Reviewer Trust and the Risk of Automation Complacency

A subtler risk than an incorrect AI suggestion slipping through is the opposite failure, a reviewer who has seen enough correct AI suggestions over time begins rubber stamping the AI pass's output without genuinely evaluating it, effectively collapsing the human triage step in Section 4.5 back into the automatic acceptance the pipeline was specifically designed

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 7, Issue 1, January - February 2025 |

DOI:10.15662/IJEETR.2025.0701007

to avoid. This risk is not hypothetical, it is the same automation complacency pattern documented in other domains where a human is nominally kept in the loop as a check on an automated system's output but gradually stops exercising independent judgment once the system's error rate appears low enough.

The outcome logging described in Section 4.7 gives a partial defense against this pattern, since a reviewer confirming AI suggestions at an implausibly high rate relative to their historical dismissal rate is itself a signal worth investigating, but logging alone does not prevent complacency, it only makes it detectable after the fact. Table 6 sketches an illustrative set of signals a team might monitor specifically for this failure mode, distinct from the confirmation rate tracking already described in Section 4.7.

**Table 5. Illustrative signals for detecting reviewer complacency toward AI suggestions**

| Signal | What it might indicate | Suggested response |
|---|---|---|
| Confirmation rate rising sharply for one reviewer | Reviewer may be confirming without independent evaluation | Spot check a sample of that reviewer's recent confirmations |
| Time between AI comment and triage decision shrinking | Decisions may be made too quickly for genuine review | Compare against the reviewer's own historical triage time |
| Escalation rate falling to near zero | Reviewer may no longer be flagging genuine uncertainty | Review recent escalations against team wide baseline rates |

## VII. LIMITATIONS

The case study in Section 5 uses simulated indicators constructed for illustration and does not report measurements from a production team, so the specific turnaround and defect figures shown in Figures 3 and 4 should be read as illustrative of the kind of effect the pipeline is designed to produce rather than as an empirical claim. The pipeline also assumes access to a large language model capable of the diff level reasoning described in Section 2.5, an assumption that depends on the specific model's ongoing availability and cost, both of which can change independently of anything described in this article. Finally, the risk scoring function in Listing 3 uses a deliberately simple set of signals; a production implementation would likely need a more sophisticated model of historical defect density than the illustrative formula shown, particularly for a codebase without extensive prior incident history to draw on.

## VIII. FUTURE WORK

Three directions follow from the limitations in Section 7. First, an empirical study measuring the pipeline's effect on a production Angular team, rather than through simulated indicators, would test whether the illustrative turnaround and defect trends in Section 5 hold in practice and at what adoption cost. Second, a more rigorous treatment of the risk scoring function in Section 4.4, potentially learned from a team's own historical pull request outcomes rather than hand specified as in Listing 3, would likely produce more accurate routing than the illustrative formula this article presents.

Third, the calibration process described in Section 4.7 was presented only qualitatively; a follow up study tracking AI suggestion confirmation rates over an extended adoption period, and correlating a declining confirmation rate with specific changes in the underlying model or the codebase, would give teams a more concrete basis for deciding when the AI assisted pass in Section 4.3 needs reconfiguration.

Fourth, the reviewer complacency signals sketched in Section 6.3 were presented as informal monitoring heuristics rather than validated indicators; a follow up study establishing which of the signals in Table 5, if any, reliably predicts a genuine drop in review quality, as opposed to a reviewer legitimately becoming more efficient at triage over time, would let a team distinguish complacency from expertise with more confidence than the heuristics offered here allow.

## IX. CONCLUSION

This article presented a set of integration patterns for incorporating an AI assisted review pass into an Angular team's pull request workflow, built around four coordinated elements, a lint and static analysis gate, an AI assisted review pass, risk based reviewer routing, and a human triage step backed by a continuous integration merge gate. An illustrative case

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 7, Issue 1, January - February 2025 |

DOI:10.15662/IJEETR.2025.0701007

study for one Angular team suggested the combined elements can substantially reduce pull request turnaround time and escaped defects, though these results are illustrative rather than empirical. The pipeline's central design choice, treating every AI generated comment as a draft requiring human confirmation rather than an automatically authoritative finding, reflects a broader point this article's discussion in Section 6.1 returns to, that accelerating code review with AI assistance is a matter of removing waiting time and mechanically resolvable friction from the process, not a matter of removing the human judgment the process exists to apply in the first place.

**Appendix A: AI Assisted Review Pipeline Adoption Checklist**

This checklist summarizes the practical steps a team can use when adopting the pipeline described in Section 4, organized by the same four elements from Section 4.1. It is offered as a starting point rather than an exhaustive audit.

**A.1 Static analysis gate (Section 4.2)**

1. Confirm the angular-eslint rule set in Listing 1 covers the Angular specific patterns the team has previously caught only through manual review.
2. Run the static analysis gate on a sample of recent pull requests before enforcing it, to confirm it does not produce an unmanageable volume of pre existing findings.
3. Distinguish blocking rules from advisory warnings explicitly, following the same severity distinction used for the merge gate in Section 4.6.
4. Review the rule set periodically as Angular's own recommended linting configuration evolves.

**A.2 AI assisted review pass (Section 4.3)**

1. Confirm every AI generated comment is labeled explicitly, following Listing 2, so it is never mistaken for a human reviewer's assessment.
2. Set a maximum comment count per pull request, following Listing 2's example, to avoid overwhelming the human reviewer with low value suggestions.
3. Pilot the review pass on a small number of pull requests before enabling it team wide, reviewing its confirmation rate from the pilot before full rollout.
4. Confirm the review pass has access to enough diff context to generate relevant comments, not only the changed lines in isolation.

**A.3 Risk based routing (Section 4.4)**

1. Confirm the risk signals in Listing 3 reflect the team's actual codebase structure, particularly which modules are considered shared or high risk.
2. Recalibrate the risk scoring function periodically, following the pitfall in Section 4.8, rather than treating its initial configuration as permanent.
3. Confirm every reviewer's module context is tracked accurately enough for routing to be meaningful.
4. Track routing outcomes over time to confirm high risk pull requests are actually reaching reviewers with relevant context.

**A.4 Human triage and outcome logging (Sections 4.5, 4.7)**

1. Confirm the merge gate in Section 4.6 never blocks on an unconfirmed AI suggestion, following the correction to the anti pattern in Section 4.8.
2. Log every triage decision, confirmed, dismissed, or escalated, alongside the AI pass's original confidence score.
3. Review the confirmation rate from the logged outcomes on a regular cadence to catch a declining trend before reviewers disengage from the AI pass entirely.
4. Track the turnaround and defect indicators in Table 4 each sprint to confirm the pipeline is holding, not only at initial adoption.

**Appendix B: Illustrative Pull Request Lifecycle Example**

This appendix works through a single illustrative pull request end to end, from open through the pipeline's four elements to merge, to make the abstract description in Section 4 concrete.

International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 7, Issue 1, January - February 2025 |

DOI:10.15662/IJEETR.2025.0701007

**Table 6. Illustrative pull request, from open to merge**

| Stage | Illustrative content |
|---|---|
| Pull request opened (Section 4.1) | Change modifies a shared authentication guard used across twelve routed modules |
| Static analysis (Section 4.2) | One angular-eslint warning found, missing OnPush change detection strategy |
| AI assisted pass (Section 4.3) | Three comments generated, one flags a missed null check with confidence 0.86 |
| Risk scoring and routing (Section 4.4) | Score of 22, above the routing threshold of 12, routed to a reviewer with authentication module context |
| Human triage (Section 4.5) | AI comment on the missed null check confirmed; the other two AI comments dismissed as not applicable |
| Outcome | Confirmed finding addressed by the author, merge gate passes, pull request merged |

Listing 5 shows the illustrative outcome log entry this pull request would produce, following the fields described in Section 4.7.

```
// Illustrative outcome log entry, following Section 4.7
{
  pullRequestId: 'pr-4821',
  riskScore: 22,
  routedReviewer: 'reviewer-with-auth-context',
  aiComments: [
    { confidence: 0.86, outcome: 'confirmed' },
    { confidence: 0.41, outcome: 'dismissed' },
    { confidence: 0.38, outcome: 'dismissed' },
  ],
  turnaroundHours: 4.2,
}
```

Listing 5. An illustrative outcome log entry for the pull request worked through in Table 5 (Section 4.7).

**Appendix C: Illustrative Reviewer Routing Responsibility Matrix**

Table 2's problem summary and Section 4.4's routing logic both refer to reviewer context without stating, for a given pipeline stage, who is responsible for maintaining the underlying data that routing decisions depend on. This appendix sketches that assignment explicitly, as a starting point for a team adapting the pipeline's routing and calibration responsibilities to its own role structure.

**Table 7. Illustrative responsibility matrix by pipeline stage**

| Pipeline stage (Figure 1) | Responsible | Consulted | Informed |
|---|---|---|---|
| Static analysis rule set (4.2) | Engineering lead | Team members encountering frequent false positives | Full team |
| AI review pass configuration (4.3) | Engineering lead or designated tooling owner | Reviewers, on comment quality feedback | Full team |
| Risk scoring inputs (4.4) | Engineering lead | Senior reviewers, on module risk assessment | Reviewer pool |
| Human triage (4.5) | Routed reviewer | Author, on disputed findings | Engineering lead, on escalations |
| Outcome logging and calibration (4.7) | Engineering lead or designated tooling owner | All reviewers, on confirmation rate trends | Full team, quarterly |

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 7, Issue 1, January - February 2025 |

DOI:10.15662/IJEETR.2025.0701007

A team without a distinct tooling owner role should not treat a blank assignment as permission to leave routing and calibration unowned, but should instead assign it explicitly to the engineering lead or a rotating designated owner, following the small scale guidance in Section 6.2, so that the pipeline's underlying data does not silently go stale the way the calibration pitfall in Section 4.8 describes.

**Statements and Declarations**

**Funding.** This work received no specific grant from any funding agency in the public, commercial, or not for profit sectors. [Funding statement placeholder, to be completed with actual funding information prior to submission.]

**Conflicts of interest.** The author declares no known competing financial interests or personal relationships that could have appeared to influence the work reported in this article. [Conflicts of interest statement placeholder.]

**Data availability.** All numerical results presented in Section 5 and Section 6 are simulated for illustration and are not derived from a production dataset. [Data availability statement placeholder, to be completed if empirical data is added in a future revision.]

**Author contributions.** [Author contributions placeholder, to be completed prior to submission, for example following the CRediT taxonomy for conceptualization, methodology, and writing.]

**Acknowledgments.** [Acknowledgments placeholder for colleagues, reviewers, or institutions who supported this work but do not meet authorship criteria.]

## REFERENCES

1. Amazon Web Services. (2019). AWS Announces Amazon CodeGuru for Automated Code Reviews and Application Performance Recommendations. https://aws.amazon.com/about-aws/whats-new/2019/12/aws-announces-amazon-codeguru-for-automated-code-reviews-and-application-performance-recommendations
2. angular-eslint. (2022). angular-eslint: Monorepo for Tooling Related to Using ESLint with Angular. https://github.com/angular-eslint/angular-eslint
3. Bacchelli, A., and Bird, C. (2013). Expectations, Outcomes, and Challenges of Modern Code Review. Proceedings of the 2013 International Conference on Software Engineering (ICSE).
4. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., and others. (2021). Evaluating Large Language Models Trained on Code. arXiv:2107.03374.
5. GitHub. (2023a). Copilot for Pull Requests. GitHub Next. https://githubnext.com/projects/copilot-for-pull-requests/
6. GitHub. (2023b). Octoverse: The State of Open Source and AI in 2023. GitHub Blog. https://github.blog/news-insights/research/the-state-of-open-source-and-ai/
7. McIntosh, S., Kamei, Y., Adams, B., and Hassan, A. E. (2014). The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects. Proceedings of the 11th Working Conference on Mining Software Repositories (MSR).
8. Rigby, P. C., and Bird, C. (2013). Convergent Contemporary Software Peer Review Practices. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE).
9. Sadowski, C., Soderberg, E., Church, L., Sipko, M., and Bacchelli, A. (2018). Modern Code Review: A Case Study at Google. Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE SEIP).
10. SonarSource. (2022). Understanding Quality Gates. SonarQube documentation. https://docs.sonarsource.com/sonarqube-server/quality-standards-administration/managing-quality-gates/introduction-to-quality-gates