Code Review

Code Review

- **The Core Premise:** Code reviews transcend syntax checking; they serve as critical gates for quality, security, and knowledge sharing.
- **Beyond the Classroom:** Transitioning from individual scripting to collaborative enterprise codebases.
- **The Goal:** Balancing rigorous technical standards with high empathy and psychological safety.

Why Code Reviews Feel Unfamiliar

- **In Academia:** Code is typically a *solo performance* evaluated by a single professor or automated grader for correctness.
- **In Industry:** Code is a *team activity* where readability, design, and longevity matter more than just making tests pass.
- **Design Decisions:** Writing code that outlives its author and remains maintainable for years.

Why Code Reviews Matter

- **Catching Bugs Early:** Finding a logic error before merging is exponentially cheaper than fixing it in production.
- **The Interdisciplinary Lens:** Your code must be readable to teammates with diverse technical backgrounds or future maintainers.
- **The "Future You" Argument:** *"The person whose code you are critiquing on a Tuesday morning might be you in six months when you have to debug this at 3:00 AM."*

What is a Code Review?

- **Definition:** A systematic peer examination of source code to find bugs, improve quality, and spread knowledge.
- **Spreading Tribal Knowledge:** Capturing unwritten rules, context, and institutional memory not found in official documentation.
- **The Roles:**
  - **The Author:** Writes code, opens the PR, explains design choices, and responds to feedback.
  - **The Reviewer:** Reads, questions, and tests the logic before merge.

The Pull Request Lifecycle

1. **Build:** Author builds feature locally on a dedicated branch (for example, `feature/user-auth`).
2. **Prepare:** Write a clear title/description, link issues, and **self-review** first.
3. **Assign:** Request reviews from peers on the team.
4. **Review:** Inspect code, leave comments, request changes, or approve.
5. **Iterate & Merge:** Make updates, get approval, and merge into the `main` branch.

PR Sizing & Management

- **The Ideal Size:** Aim for under **300–400 lines of code** per review.
- **Why Size Matters:** Massive PRs lead to "rubber-stamping" (approving without reading) because reviewers experience cognitive overload.
- **Turnaround Time:** Smaller PRs get reviewed faster and unblock the rest of the team more efficiently.

Code Review Culture & Etiquette

- **It's Not Personal:** The code is being reviewed, not the author's intelligence or worth. Decouple ego from code.
- **Assume Positive Intent:** Reviewers aren't nitpicking; they are trying to protect the product and help you grow.
- **Acknowledge Good Work:** Don't just point out flaws! If a teammate wrote an elegant algorithm, leave a comment: "*Clever use of this design pattern here! Very clean.*"

Framing Feedback Effectively

- **The Rule:** Train reviewers to ask questions rather than give authoritarian commands.
- **Bad Review Comment:**
  - *"This function is a total mess. Rewrite it using dependency injection."* (Triggers defensiveness, zero guidance).
- **Good Review Comment:**
  - *"This function handles validation, DB persistence, and email dispatch. What do you think about extracting the email logic into a separate service so it's easier to unit test?"* (Educational, collaborative, open-ended).

The Universal Checklist (Part 1)

- **Functionality & Edge Cases:** Does the code solve the problem? Does it handle invalid user input and network failures gracefully?
- **Readability & Maintainability:** Can a peer understand this code next week without an explanation? Are variables named logically?
- **Test Coverage:** Are there automated unit or integration tests for new features? Do existing tests still pass?

The Universal Checklist (Part 2)

- **Security & Input Validation:** Is external input properly sanitized? Are there hardcoded secrets or injection vectors?
- **The Division of Labor (Linters vs. Humans):**
  - Let automated linters handle code style (indentation, spacing, formatting).
  - Human reviewers spend their energy on logic, security, and architecture.

Version Control & Automation

- **Feature Branches:** Always use dedicated branches (`feature/user-auth`) instead of pushing directly to `main`.
- **Atomic Commits:** Commits should tell a chronological story, making debugging and bisecting manageable.
- **Static Analysis:** Use tools to catch code smells automatically before a human reviews it.
- **CI/CD Quality Gates:** Block merges if test coverage drops or architectural rules are broken.

Key Takeaways

- **The Balance:** Good code reviews blend rigorous technical standards with empathy and psychological safety.
- **Career Growth:** Participating in code reviews is your fastest track to senior-level engineering skills.