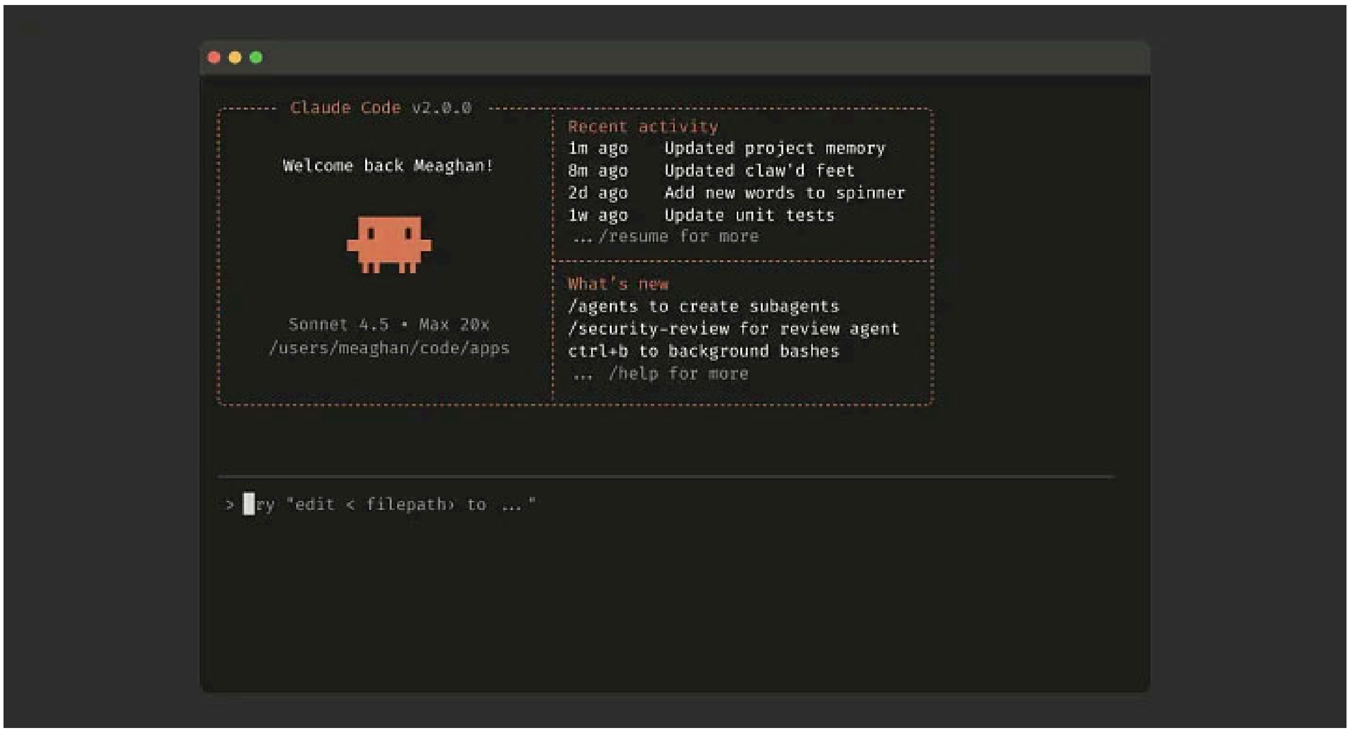
10 Production-Grade Claude Code Prompts — Complete Claude Code Prompting Guide

This Complete Guide will help you using Claude Code more efficiently by prompting it better. This is How I work with Claude Code on daily basis



```
Claude Code v2.0.0
                                   Recent activity
                                   1m ago    Updated project memory
     Welcome back Meaghan!         8m ago    Updated claw'd feet
                                   2d ago    Add new words to spinner
                                   1w ago    Update unit tests
                                    .../resume for more

                                   What's new
                                   /agents to create subagents
     Sonnet 4.5 • Max 20x          /security-review for review agent
  /users/meaghan/code/apps         ctrl+b to background bashes
                                    ...  /help for more

> Try "edit < filepath> to ..."
```

Claude Code — The #1 Agentic Coding Tool that lives in your Terminal

10 Claude Code Prompts That Eliminated 15 Hours of Debugging Hell Last Week

Last Tuesday at 11 PM, I watched Claude Code generate a beautiful authentication middleware. Clean code. Perfect patterns. It even added error handling.

It also broke our entire staging environment.

The bug? Claude implemented session validation that worked flawlessly for logged-in users but threw 500 errors for guest routes. The tests passed because Claude had mocked the session store. Three hours of emergency debugging later, I realized the problem wasn't Claude — it was how I prompted it.

That session cost us a deployment delay and taught me something critical: Claude Code's intelligence is a multiplier on your prompt engineering. Bad prompts produce elegant-looking code that fails in production. Good prompts produce code that actually works.

Over three months of production deployments — refactoring authentication systems, debugging race conditions in payment processors, shipping features under deadline pressure — I've developed ten prompts that consistently produce reliable results. These aren't theoretical patterns from documentation. They're battle-tested protocols that solve specific failure modes I've encountered while shipping real software.

Each prompt addresses a breakdown I've personally experienced: permission fatigue that destroys flow state, shallow solutions that ignore architecture, tests that pass but validate nothing. The kind of problems that make you wonder if AI assistance is worth the debugging overhead.

Here's what actually works.

Prompt #1: Permission Eliminator — Reclaim Your Flow State

The Failure Mode: You're deep in implementation, your mental model is solid, you're making progress. Then Claude asks: "Can I edit this file?" You approve. "Can I modify this function?" You approve. "Can I update this import?" You approve.

Forty-seven permission requests later, you've lost your train of thought. Your flow state is shattered. The feature that should have taken an hour has consumed your entire afternoon.

The Solution:

bash

```
claude --dangerously-skip-permissions
```

Add this alias to your shell configuration:

bash

```
alias cc='claude --dangerously-skip-permissions'
```

Why This Works: The flag disables permission prompts for the entire session. Claude operates autonomously within your working directory. You're explicitly choosing velocity over guardrails — a reasonable tradeoff for the right context.

When to Use: Green-field projects where breaking things is cheap. Isolated feature branches where you can review changes before merging. Prototyping sessions where speed matters more than precision.

When Not to Use: Production hotfixes where a wrong edit creates an incident. Shared codebases without proper branching strategy. Any situation where the cost of reviewing bad changes exceeds the time saved.

Real Results: Harper Reed, who served as CTO for Obama's 2012 campaign, reports completing green-field projects in 30–45 minutes using this workflow — projects that previously took multiple hours due to constant permission interruptions.

Source: *Anthropic's official CLI documentation, Harper Reed's development blog (2025).*

CLI reference - Claude Docs

Complete reference for Claude Code command-line interface, including commands and flags.

docs.claude.com

Once you've eliminated the permission bottleneck, the next challenge emerges: getting Claude to think before it codes. This brings us to the most critical pattern in the entire system.

Prompt #2: Agentic Planner — Force Architectural Thinking Before Execution

The Failure Mode: You ask Claude to add real-time notifications. Claude immediately starts writing WebSocket code. Twenty minutes later, you have a working implementation that bypasses your message queue, ignores your existing notification system, and introduces a completely different state management pattern.

The code works. But it's the wrong solution. You've now got three ways to handle notifications instead of one, and nobody will understand why when they find this in six months.

The Prompt:

```
**shift+tab to enter Plan Mode**

I need to [specific feature/fix]. Before writing ANY code:
1. Research: Search web for current best practices on [technology/pattern]
2. Analyze: Read relevant files in this codebase
3. Plan: Create step-by-step implementation plan
4. Estimate: Token budget and completion time
5. Present: Show me the plan for approval
CRITICAL: Do NOT write code until I explicitly approve the plan.
```

Why This Works: Plan Mode (`shift+tab`) activates extended thinking without executing changes. The research step forces Claude to check current patterns—not rely on training data from 2023. The explicit "do not code" creates a checkpoint where you can catch architectural mismatches before they become code.

Thinking Budget Levels (from Anthropic's official documentation):

- `think` → Basic extended thinking for routine decisions
- `think hard` → Moderate reasoning for complex logic
- `think harder` → High reasoning budget for architectural choices

- `ultrathink` → Maximum computation for critical decisions (expensive, use strategically)

Application Examples:

```
# For architecture decisions
"ultrathink about this caching strategy: we have read-heavy traffic with
occasional bulk updates. Current Redis TTL is 15 seconds but cache hit
rate is only 45%. Consider tag-based invalidation, write-through patterns,
and distributed caching."

# For complex refactoring
"think harder about migrating our state management from Redux to Zustand.
We have 47 connected components, middleware for API calls, and devtools
integration. Show me a migration path that maintains functionality at
each step."
# For production debugging
"think about this race condition in our payment processor: orders sometimes
get double-charged when users click submit twice within 500ms. We have
idempotency keys but they're not working correctly."
```

Pro Tip: Save approved plans to `plan.md` and reference them in follow-up prompts: `"Continue from step 3 in @plan.md"`. This creates continuity across sessions and prevents re-planning when you resume work.

Real Results: Anthropic's engineering team cites this as their primary workflow for any change requiring upfront architectural thinking. It eliminates the "start over" cycle that can waste two to three hours when Claude solves the wrong problem correctly.

Source: *Anthropic Engineering Blog, "Claude Code Best Practices" (2025).*

Claude Code Best Practices

A blog post covering tips and tricks that have proven effective for using Claude Code across various codebases…

www.anthropic.com

Planning prevents bad architecture. But even good architecture fails without proper testing. The next prompt ensures your test suite actually validates something.

Prompt #3: TDD Enforcer — Eliminate Hallucination Through Failing Tests

The Failure Mode: Claude writes beautiful tests. They all pass. You feel confident. You ship to production.

Then users report the feature doesn't work. You investigate and discover Claude mocked every external dependency. The database? Mocked. The email service? Mocked. The payment gateway? Mocked. Your test suite passes because it's testing mock objects, not your actual implementation.

The Prompt:

```
We're doing strict TDD. Here's the workflow:

1. Write tests first based on these requirements: [requirements]
2. Run tests and confirm they FAIL (do NOT mock implementations)
3. Commit the failing tests: "test: [feature] failing tests"
4. STOP and wait for my approval
5. Then implement to make tests pass
6. Commit: "feat: [feature] implementation"
The tests must fail at step 2. If they pass, you've mocked something. Retry.
```

Why This Works: Tests create unambiguous success criteria. Either they pass or they fail — no room for hallucination. The "no mocking" rule is critical. When Claude mocks external services, tests pass but tell you nothing about whether your code actually works.

Concrete Example from Last Week:

```
**TDD Mode: Password Reset Feature**

Requirements:
- User requests reset, receives email with token
- Token expires after one hour
- Invalid tokens return 404
- Malformed emails return 400
Write failing tests:
1. POST /reset-password with valid email
   → sends email, returns 200
2. POST /reset-password with invalid email
   → returns 404
3. POST /reset-password with malformed email
   → returns 400
4. Token validation after 61 minutes
   → returns 401, token rejected
Run tests - confirm all 4 fail.
Commit: "test: password reset failing tests"
STOP - I'll review test quality.
Then implement:
- Token generation with crypto.randomBytes
- Email dispatch through SendGrid
- Expiration check in middleware
Commit: "feat: password reset implementation"
NO MOCKING of email service, database, or token generation.
```

Real Results: Harper Reed describes this as his primary anti-hallucination technique. Tests either pass or fail. The failing-first requirement proves your tests actually verify behavior instead of validating mock objects.

Source: *Anthropic's best practices documentation, Harper Reed's Twitter thread on AI-assisted TDD (October 2025).*

Anthropic Courses

Learn to build with Claude AI through Anthropic's comprehensive courses and training programs.

anthropic.skilljar.com

Testing validates logic, but visual work requires a different approach. Screenshots bridge the gap between design intent and implementation.

Prompt #4: Multi-Modal Context — Show Claude What You Want

The Failure Mode: You ask Claude Code to build a dashboard card component. Claude generates something that technically meets the requirements: *it displays data, has hover states, and includes proper accessibility attributes*.

But the spacing is wrong. The colors don't match your design system. The typography hierarchy is inverted. The component looks "close enough" but fails every design review. You spend two hours tweaking CSS that Claude could have generated correctly if it had just seen the design.

GitHub - alirezarezvani/claude-code-tresor: A world-class collection of Claude Code utilities…

A world-class collection of Claude Code utilities: autonomous skills, expert agents, slash commands, and prompts that…

github.com

The Prompt:

```
**Paste screenshot: macOS cmd+ctrl+shift+4, paste with ctrl+v**
**Or drag-drop images directly into prompt**

Build this [component] matching this design exactly:
[attach screenshot or paste Figma URL]
Requirements:
- Match exact spacing, colors, typography from the image
- Use our design system: @design-system.ts
- Implement accessibility (ARIA labels, keyboard navigation)
- Include hover/focus states as shown
Reference similar implementations: @existing-component.tsx
```

Why This Works: Vision combined with code context produces precise implementations. Claude sees the exact measurements (16px padding not 20px), the specific colors (`#3B82F6` from your design system, not a similar blue), and the typography hierarchy that makes the design work.

Screenshot Methods:

- **macOS**: `cmd+ctrl+shift+4` → screenshot to clipboard, `ctrl+v` to paste into terminal
- **Drag-drop**: Drop images directly into the Claude Code prompt
- **URLs**: Paste Figma or design tool URLs for Claude to fetch and analyze

When This Matters:

- Building UI components from Figma mockups
- Debugging visual regressions (attach screenshots of expected vs actual)
- Implementing data visualizations from reference charts
- Creating error states that match design specifications

Pro Tip: Include reference implementations from your codebase. Claude matches both the visual design AND your existing code patterns, maintaining consistency across components.

Real Results: First implementations match design requirements at 80–90% fidelity instead of 40–60%. This eliminates two to three rounds of "close but not quite" CSS adjustments.

Source: Anthropic's official documentation on multi-modal prompting (2025).

Prompt engineering overview - Claude Docs

This guide assumes that you have: If not, we highly suggest you spend time establishing that first. Check out A clear…

docs.claude.com

Visual precision matters for interfaces. For bugs, you need systematic investigation that finds root causes instead of patching symptoms.

Prompt #5: Systematic Debugging Protocol — Fix Problems, Not Symptoms

The Failure Mode: Users report checkout failures. You ask Claude to fix it. Claude adds a try-catch block around the payment API call. Tests pass. You ship it.

Next week, users report the same issue. Claude's fix caught the error but didn't address why the payment API was failing: a race condition in session validation that only occurred under specific timing. The try-catch hid the real problem.

The Prompt:

```
**shift+tab for Plan Mode**

Debug this issue systematically:
ERROR: [paste complete error message and stack trace]
Context:
- Started after: [recent changes or deployment]
- Occurs when: [specific reproduction steps]
- Works correctly when: [scenarios where it doesn't fail]
Investigation Protocol:
1. Analyze stack trace - identify actual problem, not just error symptom
2. Trace execution path across files that leads to this error
3. Identify root cause (logic bug, race condition, missing validation, etc.)
4. Explain WHY the bug occurred (architecture gap, edge case, etc.)
5. Propose fix that addresses the root cause
6. Suggest prevention strategy (tests, type guards, updated documentation)
7. Update CLAUDE.md with learnings to prevent similar issues
Do NOT suggest quick fixes. I need root cause analysis.
```

Why This Works: ***Plan Mode investigates without making changes to your code or codebase*** — safe for production issues. The systematic protocol prevents *“try this and see if it works”* debugging. The *“explain WHY”* requirement builds institutional knowledge instead of accumulating patches.

BUILDING A DOCUMENTATION GENERATOR — Your Agent’s first useful job

Article 2 of 7 — Foundation Series: I’ve promised myself I’d keep our docs updated at least a dozen times, and fourteen…

alirezarezvani.medium.com

What Claude Debugs Well:

- **Logic Bugs**: Traces execution with sample data, identifies mathematical errors and off-by-one mistakes
- **Performance Issues**: Spots O(n²) operations, identifies database query inefficiencies
- **Race Conditions**: Analyzes async timing, finds state synchronization problems
- **Integration Failures**: Reviews API contracts, catches authentication timing issues
- **State Corruption**: Follows data flow through Redux/Zustand, finds mutation bugs

Real Example: Filippo Valsorda, author of Go’s cryptography libraries, reported Claude Code successfully identified three “fairly complex low-level bugs” in fresh code — one-shot debugging that would have taken hours to track down manually. He uses Claude to locate the root cause, then implements the fix himself with full understanding of why it failed.

Pro Tip: After fixing, ask: *“What edge cases could still cause this function to fail?”* Claude stress-tests your logic with inputs you didn’t consider during implementation.

Real Results: Multiple developer reports show debugging time reduced from three hours to one hour per issue. More importantly, bugs stay fixed because root causes are addressed instead of symptoms patched.

Source: *ClaudeLog debugging guide, Filippo Valsorda's cryptography blog (November 2025), Anthropic engineering practices.*

Claude Code Best Practices

A blog post covering tips and tricks that have proven effective for using Claude Code across various codebases…

www.anthropic.com

Debugging finds problems in small scopes. But large codebases require active context management to maintain Claude's effectiveness.

Prompt #6: Context Window Manager — Stay Efficient in Large Codebases

The Failure Mode: Your codebase has 50,000 lines across 800 files. You ask Claude to refactor the authentication middleware. Claude starts reading unrelated files: the payment processor, the admin dashboard, the email templates. Token count explodes. Responses become generic because Claude is trying to hold too much irrelevant code in context.

Performance degrades. What should take 20 minutes stretches to two hours as Claude thrashes through unnecessary context.

The Prompt:

```
**For this task, use ONLY these specific files:**

[Use tab-completion to add targeted files]
- @src/auth/middleware.ts
- @src/auth/session.ts
- @types/user.ts
- @config/security.ts
Ignore everything else in the repository.
```

```
Task: [your specific objective]
If you need additional context, STOP and ask me which files to add.
Don't read files speculatively.
```

Supporting Commands:

bash

```bash
# Clear conversation history when changing focus
/clear

# Compact history while preserving essential context
/compact Keep implementation details for auth refactor,
discard earlier discussion about email templates
# Use pipe mode for targeted queries
claude -p "explain the session validation logic" < src/auth/middleware.ts
```

Why This Works: Explicit file scoping prevents context pollution. Claude focuses computational resources on relevant code instead of scanning thousands of lines that don't matter for the current task. The *"ask before reading"* rule stops speculative context expansion.

Large Codebase Strategies:

1. **Modular Sessions**: One terminal session per feature domain. Don't mix authentication work with payment processing work.
2. **Git Worktrees**: Multiple branches in separate directories → parallel Claude sessions without merge conflicts
3. **Hierarchical CLAUDE.md**: Project-wide patterns in root `.claude/CLAUDE.md` , module-specific guidance in subdirectories

Pro Tip: Use `/memory` to review what Claude currently knows about your project. Edit this file to remove outdated context or add new architectural decisions.

Real Results: Proper context scoping is the difference between Claude that "understands" your codebase and Claude that guesses based on insufficient information. Well-managed context maintains quality in projects exceeding 50,000 lines.

Source: Anthropic's context management documentation, senior engineer best practices (2025).

Context management maintains quality. Performance analysis finds the inefficiencies you didn't know existed.

Master Claude Memory in 7 Steps: Cut Context Loss by 80% with Project-Scoped Recall

alirezarezvani.medium.com

Prompt #7: Performance Analyzer — Find Bottlenecks Without Profilers

The Failure Mode: Your dashboard loads slowly. Users complain. You suspect database queries, but setting up profiling takes time and interpreting flamegraphs requires focus you don't have during a sprint.

Meanwhile, the dashboard is losing users because it takes four seconds to load data that should appear in under one second.

The Prompt:

```
**Performance Analysis Request**
```

```
Analyze this code for performance bottlenecks:
@[file or function path]
```

```
Focus on:
1. Time complexity - flag O(n²) or worse operations
2. Unnecessary re-renders (React) or redundant computations
3. Database query efficiency (N+1 problems, missing indexes)
4. Memory leaks or excessive allocations
5. Blocking operations that should be async

For each issue:
- Explain the performance impact (quantify where possible)
- Show the problematic code pattern
- Provide optimized implementation
- Estimate improvement magnitude

Prioritize by impact: identify the changes that matter most.
```

Why This Works: Claude spots algorithmic inefficiencies instantly — patterns that require careful profiler analysis to discover manually. It catches common antipatterns (nested loops, missing memoization, synchronous I/O in hot paths) that developers overlook during implementation.

Performance Wins Claude Identifies:

- **React Components**: `useEffect` missing dependencies causing render loops, expensive calculations not wrapped in `useMemo`
- **Database Queries**: N+1 problems where one query becomes 50, missing indexes on foreign keys
- **Algorithms**: O(n²) sorting that should be O(n log n), unnecessary array copies creating garbage
- **Async Operations**: Synchronous file I/O blocking the Node.js event loop
- **Memory Management**: Event listeners never cleaned up, database connections never closed

Concrete Example from Last Sprint:

```
Analyzing dashboard component performance:

Found 3 critical issues:
1. Line 67-89: User data aggregation uses nested loops
   Current: O(n²) - 3.2 seconds for 1000 records
   Solution: Use Map for O(n) grouping
   Impact: 200ms for 1000 records (16x faster)
2. Line 112: useEffect re-fetches on every render
   Current: 50+ API calls per minute
   Solution: Add proper dependency array [userId, dateRange]
   Impact: Reduces to 2-3 API calls per session
3. Database: user_orders table missing index on user_id
   Current: 200ms query time for user dashboard
   Solution: CREATE INDEX idx_user_orders_user_id ON user_orders(user_id)
   Impact: 5ms query time (40x faster)
```

Pro Tip: Combine with extended thinking for complex analysis: `"ultrathink about performance bottlenecks in this data aggregation pipeline"`

Real Results: Catches 60–80% of performance issues without profiling tools. The remaining 20–40% require actual profiling, but Claude dramatically narrows the search space.

Source: Developer case studies, Anthropic's code analysis documentation (2025).

Claude Code overview - Claude Docs

Learn about Claude Code, Anthropic's agentic coding tool that lives in your terminal and helps you turn ideas into…

docs.claude.com

Performance optimization improves what exists. Documentation ensures future developers understand why it exists.

Prompt #8: Documentation Generator — Maintain Living Documentation

The Failure Mode: You ship the authentication refactor. It works perfectly. You promise yourself you'll write documentation "tomorrow."

Six months pass. A new developer joins. They ask how authentication works. You spend two hours explaining the session management, token refresh logic, and security considerations because you never documented it.

The new developer implements a feature that bypasses your security middleware because they didn't know it existed. Now you have a security vulnerability in production.

The Prompt:

```
**Documentation Mode**

Generate comprehensive documentation for:
@[file, module, or feature path]
Structure:
1. **Purpose**: What this code does (2-3 sentences)
2. **Architecture**: How it integrates with the system
3. **Key Components**: Main functions/classes and their responsibilities
4. **Usage Examples**: Common use cases with code snippets
5. **Edge Cases**: Known limitations, gotchas, performance notes
6. **Testing**: How to test this code, what to verify
7. **Dependencies**: External libraries and why they're needed
Format: [README.md | JSDoc comments | inline documentation]
Also update CLAUDE.md with patterns from this implementation.
```

Why This Works: Claude reads the implementation directly — documentation matches reality by definition. It catches edge cases from actual code, not from fading memory. Structured format ensures consistency across different modules.

Documentation Types by Context:

- **README.md**: Module-level documentation for architectural understanding
- **JSDoc/TSDoc**: Function-level API documentation for implementation details
- **ARCHITECTURE.md**: System design and component relationships
- **CLAUDE.md**: Project patterns and conventions for AI-assisted development

Practical Example from Yesterday:

```
Just finished implementing rate limiting middleware.

"Document the rate limiting system for future developers:
- How token bucket algorithm works
- Where limits are configured (environment variables)
- Common error scenarios and client handling
- Example usage in route handlers
- Performance impact and Redis dependency
- Security considerations (distributed attacks)
Format as README.md and also document in @CLAUDE.md
under 'Security Patterns' section"
```

Pro Tip: Request documentation as you build, not after: `"Implement password reset feature, then generate documentation covering the email flow, token lifecycle, and security model."`

Real Results: Documentation remains current because it's generated from actual code. New team members onboard faster because documentation reflects reality. Code reviews automatically include documentation updates.

Source: Medium articles on Claude Code workflows, enterprise adoption case studies like Treasure Data (2025).

Documentation explains the code. Code review evaluates whether it should be merged.

From Assistant to Autonomous Engineer: The 9-Month Technical Evolution of Claude Code

alirezarezvani.medium.com

Prompt #9: Code Review Assistant — Catch What Linters Miss

The Failure Mode: You run the linter. No errors. You run the tests. Everything passes. You submit the pull request.

Your tech lead reviews it and requests changes: *“This variable name is misleading.” “This comment is stale.” “This violates our architectural pattern.” “What happens if the network fails here?”*

All valid feedback. None of it caught by automated tools. You spend another hour addressing subjective quality issues that you could have fixed before requesting review.

The Prompt:

```
**Code Review Mode**

Review this PR for subjective quality issues:
@[files changed in this PR]
Focus on:
1. **Clarity**: Misleading names, confusing logic, missing explanations
2. **Maintainability**: Will future developers understand this?
3. **Architecture**: Does this fit our patterns? Any code smells?
4. **Edge Cases**: What could break? Missing validation?
5. **Testing**: Are tests comprehensive and meaningful?
6. **Documentation**: Self-documenting code vs. needs comments
Provide specific line-by-line feedback:
- What's problematic and why
- How it impacts maintenance
- Concrete improvement suggestion
Be direct and critical. I want honest feedback.
```

Why This Works: Claude identifies patterns automated tools ignore — the subjective elements that separate maintainable code from technical debt. It catches typos in comments, stale TODOs, and confusing names that work but hurt readability.

Review Focus Areas:

- **Naming Precision**: Variables like `temp`, `data2`, or `handleClick2` that don't communicate purpose

- **Comment Accuracy**: Stale comments contradicting code, missing explanations for non-obvious logic
- **Pattern Consistency**: Deviations from established codebase patterns
- **Architectural Alignment**: Tight coupling, God objects, violations of separation of concerns
- **Security Awareness**: Potential injection vulnerabilities, authentication bypasses, data exposure

Example from This Morning's Review:

```
Reviewing payment processing implementation:

Line 47: Variable 'result' is too generic for payment confirmation data.
→ Rename to 'paymentConfirmation' for clarity about contents and purpose.
Line 89: Comment says "validates input" but function also processes payment.
→ Violates single responsibility. Split into validatePaymentRequest()
  and processPayment().
Line 112: Missing error handling for network timeouts.
→ Wrap in try-catch with specific handling for timeout scenarios.
  Current code leaves users seeing generic "something went wrong."
Line 156: Test mocks payment gateway, doesn't verify integration.
→ Add integration test with sandbox API key to catch real failures.
```

Pro Tip: Run this on your own code before submitting PRs. Fix obvious issues Claude identifies. Present cleaner code to human reviewers. Reduce review cycle time.

Real Results: Catches 70–80% of "change requested" feedback before human review. Code quality improves because subjective issues surface early in development, not late in review.

Source: *Anthropic's official documentation on code review capabilities (2025).*

Claude Skills Tutorials & Toolkit: 7 Steps How to Actually Ship Fully Customized AI For Your Needs

Step-by-step guide to Claude Skills with real business applications. 15-minute build tutorial + toolkit with claude...

Review catches issues in individual changes. Institutional knowledge prevents entire categories of problems.

Prompt #10: Continuous Learning Loop — Build Institutional Knowledge

The Failure Mode: Every Claude session starts from zero. You explain your architecture. Claude generates code. Session ends.

Next day: new session, same explanations. You describe your state management approach again. You clarify why you chose Zustand over Redux again. You explain the authentication flow again.

Three months later, you're still explaining the same architectural decisions in every session because Claude has no memory across conversations.

The Solution — Systematic CLAUDE.md Updates:

The Prompt:

```
**After completing any significant change:**

Update CLAUDE.md with what we learned:
Section: [Architecture | Patterns | Testing | Gotchas]
Document:
1. What problem did this solve?
2. Why this approach over alternatives?
3. What patterns should future changes follow?
4. What mistakes should be avoided?
Write clear, actionable guidelines for future sessions.
```

CLAUDE.md Structure (evolves with your project):

markdown

~~~
# Project Context

## Architecture Decisions
- State: Zustand (not Redux) - 75% less boilerplate, simpler mental model
- Data Fetching: React Query for caching and request deduplication
- Auth: JWT in httpOnly cookies - prevents XSS, handles refresh automatically
- Database: Prisma ORM with PostgreSQL - type safety, migration management
## Code Style & Patterns
- ES modules only (no CommonJS anywhere)
- Functional components with hooks (class components deprecated)
- Error boundaries around feature boundaries (not every component)
- Custom hooks for shared logic (prefix with 'use')
## Testing Requirements
- Unit tests for all business logic (React Testing Library)
- Integration tests for all API routes (supertest)
- No mocking of database (use test database with migrations)
- Snapshot tests prohibited (too brittle, prefer behavior tests)
## Common Commands
```bash
npm test               # Run full test suite
npm run test:watch     # Watch mode for development
npm run lint           # ESLint + Prettier check
npm run type-check     # TypeScript validation
npm run dev            # Local development (port 3000)
npm run build          # Production build
```

## Known Gotchas & Edge Cases
- Sessions expire after 24 hours - client must handle 401 and redirect
- File uploads capped at 10MB - show error before upload attempt
- Rate limiting: 100 req/min per user - implement exponential backoff
- Webhook signatures must be verified - never trust payload without HMAC chec
- Date handling: all times stored as UTC, convert to user timezone in client
## Performance Considerations
- Dashboard queries expensive - always paginate, never fetch all users
- Image uploads must be client-side optimized (max 2MB after compression)
- N+1 query danger in user relationships - use Prisma 'include' correctly
- Redis cache TTL: 5 minutes for user data, 1 hour for static content
~~~

Why This Works: Each session builds on previous knowledge instead of starting fresh. Architectural decisions persist across weeks. Common mistakes get documented once, prevented forever. New team members inherit institutional knowledge instantly.

Maintenance Rhythm:

- **Daily**: Add gotchas when you discover edge cases during development
- **Weekly**: Review for outdated information, update patterns that evolved
- **After Major Changes**: Document new architectural decisions and their rationale
- **When Bugs Fixed**: Add prevention strategies to "Gotchas" section

Real Results: Sessions become 40–60% more effective after maintaining CLAUDE.md for one month. Claude suggests solutions matching your architecture without explanation. Bug prevention improves because common mistakes are explicitly documented.

Source: Anthropic's memory system documentation, senior engineer best practices from production Claude Code usage (2025).

How to Actually Use This System

Theory doesn't ship features. Here's the implementation path that works.

Week 1: Foundation Set up the infrastructure for effective Claude sessions:

- Add shell alias: `alias cc='claude --dangerously-skip-permissions'`
- Create initial CLAUDE.md with your project's architecture and commands
- Practice entering Plan Mode (`shift+tab`) before starting work

Week 2: Development Workflow
Integrate into daily coding:

- Use TDD Enforcer for next feature (write failing tests first)
- Add screenshots when building UI components
- Practice explicit file scoping for complex tasks

Week 3: Quality & Maintenance Build quality checks into the process:

- Run Code Review Assistant before submitting your next PR
- Apply Systematic Debugging Protocol to current bugs
- Document one module using Documentation Generator

Week 4: Advanced Optimization Layer in performance and knowledge building:

- Run Performance Analyzer on slow endpoints
- Refine CLAUDE.md based on patterns that emerged
- Train team members on the three prompts they'll use most

Start with Permission Eliminator and Plan Mode. These two create immediate impact. Add one new pattern per week. Within a month, you'll have a system that transforms Claude from occasionally helpful to consistently productive.

The Compounding Effect Nobody Talks About

These prompts don't work in isolation. They compound:

Plan Mode produces better architectures. TDD Enforcer validates those architectures with meaningful tests. Code Review catches issues before they reach production. Documentation Generator explains implementation decisions. CLAUDE.md updates make every future session smarter.

After three months using this system, my Claude sessions are three to four times more productive than my first week. The AI hasn't gotten smarter. My prompts now elicit expert behavior instead of surface-level solutions.

The difference between frustrating AI assistance and genuine productivity multiplier isn't the model — it's the structure you provide. These ten prompts give Claude the context and constraints it needs to be genuinely useful instead of occasionally impressive.

Start Here

Bookmark these resources for when you need deeper context:

Official Documentation:

- Anthropic: Claude Code Best Practices — Engineering team's patterns
- Claude Code CLI Usage — Complete command reference
- Prompt Engineering Guide — Fundamentals that apply to all prompting

Community Resources:

- ClaudeLog — Comprehensive tutorials and troubleshooting guides
- r/ClaudeCode — Active community for pattern sharing and problem-solving

Referenced Case Studies:

- Harper Reed's development blog on TDD with Claude Code
- Filippo Valsorda's cryptography debugging analysis
- Treasure Data's enterprise deployment results (2025)

Pick two prompts. Use them for a week. Add another. Build the system incrementally instead of trying to adopt everything at once.

The goal isn't to use all ten prompts in every session. The goal is to have the right prompt ready when you hit a specific problem: permission fatigue, shallow solutions, meaningless tests, poor performance.

These prompts eliminated 15 hours of debugging for me last week. They'll do the same for you once you internalize which pattern solves which problem.