DEVELOPER TOOLS

The Developer's Claude Toolkit

40+ battle-tested prompts for code review, debugging, architecture, and shipping faster with AI

by Rook

First Edition · February 2026

Table of Contents

01 50+ Expert Prompts for Shipping Better Code

02 How to Use This Toolkit

03 Prompt 1: The Root Cause Analyst

04 Prompt 2: The Silent Failure Detective

05 Prompt 3: The Stack Trace Decoder

06 Prompt 4: The Reproduction Builder

07 Prompt 5: The Environment Diff Debugger

08 Prompt 6: The Race Condition Hunter

09 Prompt 7: The Dependency Hell Navigator

10 Prompt 8: The Memory Leak Tracker

11 Prompt 9: The Senior Engineer Review

12 Prompt 10: The Security Audit

13 Prompt 11: The Performance Review

14 Prompt 12: The API Contract Review

15 Prompt 13: The Error Handling Audit

16 Prompt 14: The Test Gap Finder

17 Prompt 15: The "Explain This to a New Hire" Review

18 Prompt 16: The Naming Audit

19 Prompt 17: The Architecture Decision Record

20 Prompt 18: The System Design Collaborator

21 Prompt 19: The Database Schema Designer

22 Prompt 20: The API Design Workshop

23 Prompt 21: The Refactoring Strategy

24 Prompt 22: The Monolith Decomposition Planner

25 Prompt 23: The Design Pattern Advisor

26 Prompt 24: The Migration Planner

27 Prompt 25: The README Generator

28 Prompt 26: The Code Documentation Writer

29 Prompt 27: The Changelog Writer

30 Prompt 28: The Runbook Writer

31 Prompt 29: The ADR From Existing Code

32 Prompt 30: The Architecture Diagram Describer

33 Prompt 31: The Technical RFC Writer

34 Prompt 32: The Unit Test Writer

35 Prompt 33: The Integration Test Designer

36 Prompt 34: The Property-Based Test Generator

37 Prompt 35: The Test Data Factory

38 Prompt 36: The Mutation Testing Analyst

39 Prompt 37: The Contract Test Writer

40 Prompt 38: The End-to-End Test Scenario Writer

41 Prompt 39: The Test Refactoring Guide

42 Prompt 40: The Code Smell Identifier

43 Prompt 41: The Extract and Inject Refactorer

44 Prompt 42: The Performance Refactorer

45 Prompt 43: The Legacy Code Moderniser

46 Prompt 44: The Dead Code Eliminator

47 Prompt 45: The Error Handling Moderniser

48 Prompt 46: The Query Optimiser

49 Prompt 47: The Caching Strategy Designer

50 Prompt 48: The N+1 Query Detector

51 Prompt 49: The Bundle Size Reducer

52 Prompt 50: The CI Pipeline Designer

53 Prompt 51: The Dockerfile Optimiser

54 Prompt 52: The Infrastructure as Code Reviewer

55 Prompt 53: The Incident Post-Mortem Writer

56 Prompt 54: The Chain-of-Thought Debugger

57 Prompt 55: The Rubber Duck Pro

58 Prompt 56: The Code Review Reviewer

59 Prompt 57: The Specification to Implementation

60 Prompt 58: The Git History Archaeologist

61 Recipe 1: The Full Code Review Pipeline

62 Recipe 2: The Bug Investigation Flow

63 Recipe 3: The System Design Pipeline

64 Recipe 4: The Legacy Code Rescue

65 Recipe 5: The Performance Rescue

The Developer's Claude Toolkit

CHAPTER 01

50+ Expert Prompts for Shipping Better Code

A practical prompt library for professional developers who use AI to build, debug, review, test, and document real software.

No fluff. No "write me a to-do app." Every prompt here was forged in production codebases and refined through hundreds of iterations.

> *"The difference between a junior and senior developer using AI isn't the model — it's the prompt."*

CHAPTER 02

How to Use This Toolkit

▸ The Anatomy of a Great Prompt

Every prompt in this toolkit follows a proven structure:

1. **Role & Context** — Tell Claude *who* it is and *what* it's looking at
2. **Constraint** — Narrow the scope so the output is actionable
3. **Format** — Specify exactly what you want back
4. **Anti-patterns** — Tell it what to *avoid* (this is where most people fail)

▸ Customisation Keys

Throughout this document, you'll see placeholders in `{curly braces}`. Replace these with your specifics:

- `{language}` — Your programming language (Python, TypeScript, Rust, etc.)
- `{framework}` — Your framework (Django, Next.js, Rails, etc.)
- `{codebase_context}` — Brief description of your project
- `{paste_code_here}` — Where your actual code goes
- `{error_message}` — The error you're seeing

▸ Pro Tips Before You Start

- **Paste real code.** Claude works best with actual code, not descriptions of code.
- **Include error messages verbatim.** Don't paraphrase — paste the full stack trace.
- **Chain prompts.** Use the output of one prompt as input to the next. The prompts in each section are designed to work in sequence.
- **Push back.** If Claude's first answer is generic, say "That's too surface-level. Go deeper." It will.

Part 1: Debugging

From cryptic stack traces to "it works on my machine" — prompts that actually find the bug.

CHAPTER 03

Prompt 1: The Root Cause Analyst

When to use: You have an error and you've already spent 20 minutes staring at it.

▸ The Prompt

```
I'm debugging an issue in a {language} {framework} application.

Here's the error:
{error_message}

Here's the relevant code:
{paste_code_here}

Don't just tell me what the error means. I need:
1. The root cause (not the symptom)
2. Why this specific code triggers it (trace the execution path)
3. The fix, with code
4. What else in the codebase might have the same underlying issue

If you need more context (e.g., config files, related modules), tell me exactly wha
```

▸ Why It Works

Most developers prompt with “fix this error” and get a surface-level response. This prompt forces Claude to distinguish between *symptoms* and *root causes*. Point 4 is the real power — it turns a single bug fix into a codebase-wide audit.

▸ Example Output (abbreviated)

> ***Root cause:*** *The* `user_id` *field is being passed as a string from the request body, but your SQLAlchemy model defines it as* `Integer`*. The ORM doesn't auto-cast in this context because you're using* `filter_by()` *which passes the value through without type coercion, unlike* `filter()` *with explicit column comparison.*
>
> ***Why this code triggers it:*** *Line 34 calls* `request.json.get("user_id")` *— Flask's* `request.json` *preserves JSON types, and your frontend is sending* `"42"` *(string) not* `42` *(integer)...*
>
> ***Other locations at risk:*** *Any endpoint using* `filter_by()` *with values from* `request.json` *without casting. Check* `routes/orders.py:67` *and* `routes/payments.py:23` *— they follow the same pattern.*

▸ Customisation Tips

- Add your ORM, database, and deployment environment for more precise answers
- If the error is intermittent, say so — it changes the debugging approach entirely
- Include the test that's failing, if you have one

CHAPTER 04

Prompt 2: The Silent Failure Detective

When to use: No error. No crash. It just... doesn't work right.

▸ The Prompt

```
Something is wrong but there's no error. Here's what I expect vs what happens:

Expected behaviour: {what_should_happen}
Actual behaviour: {what_actually_happens}

Here's the code path I think is involved:
{paste_code_here}

Walk through this code line by line as if you're a debugger. At each decision point
- What the value actually is at this point (based on the code logic)
- What I probably assumed it would be
- Where the assumption breaks

Be specific. Reference line numbers.
```

▸ Why It Works

Silent failures are the hardest bugs because there's no error message to google. This prompt turns Claude into a mental debugger — it forces a line-by-line trace that surfaces the *assumption gap* between what you think the code does and what it actually does.

▸ Example Output (abbreviated)

> ***Line 12:*** `const filtered = items.filter(item => item.status == "active")` *You're using loose equality (* `==` *). If* `item.status` *is the boolean* `true` *(which your API returns for active items),* `true == "active"` *is* `false` *in JavaScript. This silently filters out every active item.*
>
> ***Your assumption:*** `status` *is a string like* `"active"` *or* `"inactive"`*.* ***Reality:*** *Your API returns* `{ status: true }` *for active items.*

▸ Customisation Tips

- Add recent changes: "I changed X yesterday and it broke" narrows the search massively
- Include sample data if you have it — paste a JSON response, a database row, etc.
- For async issues, mention the timing: "It works when I step through slowly but fails at full speed"

CHAPTER 05

Prompt 3: The Stack Trace Decoder

When to use: You have a wall of stack trace and no idea where to start.

▸ The Prompt

```
Here's a stack trace from my {language} application. I need you to:

1. Identify which frames are MY code vs library/framework code
2. Point me to the exact line in MY code where the issue originates
3. Explain what the library was trying to do when it failed
4. Give me the fix

Stack trace:
{paste_full_stack_trace}

My project structure looks like:
{brief_structure}

Don't explain what a stack trace is. I know. Just decode this one.
```

▸ Why It Works

The final line — "Don't explain what a stack trace is" — is crucial. Without it, you'll get 3 paragraphs of intro before the actual answer. Asking Claude to separate *your* frames from *library* frames cuts through the noise immediately.

▸ Customisation Tips

- Paste the FULL stack trace, not a trimmed version — the context frames matter
- Include your `requirements.txt` or `package.json` versions if the error might be version-related
- If you have multiple stack traces (e.g., the error happens in different places), paste all of them

CHAPTER 06

Prompt 4: The Reproduction Builder

When to use: You need to reproduce a bug in isolation to understand it.

▸ The Prompt

```
I'm seeing a bug in my {language} application. I want to build a minimal reproducti

Here's the bug: {describe_the_bug}
Here's the code where it manifests: {paste_code_here}
Here's the environment: {runtime_version_os_etc}

Create a single-file reproduction script that:
1. Has zero external dependencies (or absolute minimum)
2. Demonstrates the exact bug
3. Includes comments explaining what should happen vs what does happen
4. Can be run with a single command you provide

Don't simplify the bug away — the repro must trigger the same root cause.
```

▸ Why It Works

Minimal reproductions are the gold standard of bug reports, but developers rarely create them because it's tedious. This prompt automates the tedious part. The key constraint — "Don't simplify the bug away" — prevents Claude from creating a repro that technically runs but doesn't actually exhibit the issue.

▸ Customisation Tips

- Include your OS and runtime version — reproduction cases are often platform-sensitive
- If the bug involves concurrency, say so explicitly: "This is a race condition" changes the repro strategy
- Add "Make it a pytest/jest test" if you want it in test form

CHAPTER 07

Prompt 5: The Environment Diff Debugger

When to use: “Works on my machine” / “Only fails in production.”

▸ The Prompt

```
I have a bug that only appears in {environment_a} but not in {environment_b}.

The code is identical. Here are the differences I know about:
- {env_diff_1} (e.g., Python 3.11 vs 3.12)
- {env_diff_2} (e.g., PostgreSQL 14 vs 16)
- {env_diff_3} (e.g., Linux vs macOS)

The bug: {describe_bug}
The code: {paste_relevant_code}

Give me a ranked list of the most likely causes, starting with the environment diff
```

▸ Why It Works

Environment bugs are about narrowing the diff. This prompt structures the investigation as a ranked hypothesis list rather than a single guess, which matches how experienced developers actually debug these issues — you test hypotheses in order of likelihood.

▸ Customisation Tips

- Include Docker vs bare metal, managed vs self-hosted database, etc.
- If you have access to both environments, ask for verification commands for each hypothesis
- Add deployment method (systemd, Docker Compose, Kubernetes) — it often matters

CHAPTER 08

Prompt 6: The Race Condition Hunter

When to use: Intermittent failures, flaky tests, “it only happens under load.”

▸ The Prompt

```
I suspect a race condition in this {language} code. It fails intermittently, roughl

Here's the code:
{paste_code_here}

Analyse this for concurrency issues:
1. Identify every shared mutable state
2. Map out all possible interleavings that could cause incorrect behaviour
3. Show me the specific interleaving (as a timeline) that causes the bug
4. Propose a fix that doesn't just "add a lock everywhere" — I want the minimal cor

Assume I understand concurrency. Skip the basics.
```

▸ Why It Works

Race conditions require *thinking in timelines*, which is exactly what this prompt demands. Asking Claude to map interleavings and show a specific failing timeline produces concrete, verifiable analysis rather than hand-wavy “you might have a race condition” responses.

▸ Customisation Tips

- Specify your concurrency model: threads, async/await, goroutines, actors, etc.
- Include the test that flakes, if you have one
- Add your load characteristics: “We see this under 500 concurrent requests”

CHAPTER 09

Prompt 7: The Dependency Hell Navigator

When to use: Version conflicts, import errors, “No matching distribution found.”

▸ The Prompt

```
I'm getting a dependency conflict in my {language} project.

Error: {paste_error}

Here's my dependency file:
{paste_requirements_or_package_json}

Here's my lock file (if relevant):
{paste_lock_file_relevant_section}

I need:
1. Which packages are conflicting and why
2. The version combination that satisfies all constraints
3. If no combination exists, the least painful workaround
4. Any dependencies I should pin more tightly to prevent this recurring

Don't suggest "just upgrade everything" — I need to understand the constraint graph
```

▸ Why It Works

Dependency resolution is essentially constraint satisfaction. The final line prevents the lazy answer and forces Claude to actually trace the version constraints. Including the lock file is crucial because it shows what *was* working before.

▸ Customisation Tips

- Include your Python/Node/Rust version — it constrains the solution space
- If you're in a monorepo, mention which packages share dependencies
- Add your deployment target (e.g., AWS Lambda) — some environments have package restrictions

---

CHAPTER 10

Prompt 8: The Memory Leak Tracker

When to use: Your application's memory usage grows over time.

▸ The Prompt

```
I suspect a memory leak in my {language} application. Memory grows by approximately

Here's the code I think is involved:
{paste_code_here}

Analyse this for memory leaks:
1. Identify objects that are created but never freed/dereferenced
2. Look for growing collections (lists, maps, caches) that are never pruned
3. Check for closure captures that hold references longer than intended
4. Check for event listener/callback registration without cleanup

For each issue found, show the exact fix and explain why it leaks.

Also suggest how I can verify the leak is fixed (profiling commands, metrics to wat
```

▸ Why It Works

Memory leaks have specific patterns — unclosed resources, growing collections, captured references. This prompt enumerates them explicitly, ensuring Claude checks for each category rather than guessing at the most common one.

▸ Customisation Tips

- Include your profiling data if you have it (heap snapshots, memory graphs)
- Mention your deployment context: “This runs as a long-lived server” vs “This is a batch job”
- For Python, mention if you’re using `__del__`, weak references, or `gc.collect()` — they change the analysis

---

Part 2: Code Review

Prompts that catch what linters miss — logic errors, design smells, security holes, and performance traps.

CHAPTER 11

Prompt 9: The Senior Engineer Review

When to use: You want a thorough code review before merging.

▸ The Prompt

```
Review this {language} code as a senior engineer with 15 years of experience. Be di

{paste_code_here}

For each issue, categorise it:
- □ BLOCKER: Must fix before merging (bugs, security issues, data loss risks)
- □ SHOULD FIX: Will cause problems later (poor patterns, maintainability issues)
- □ NIT: Style/preference (take it or leave it)

For each issue:
1. The exact line(s)
2. What's wrong and why it matters
3. The fix (show code, not just description)

Skip praise. Skip the "overall this looks good" fluff. Just the issues.

Things I especially care about: {your_concerns — e.g., "SQL injection", "error hand
```

▸ Why It Works

The severity categorisation is the key innovation. Without it, Claude treats a missing docstring the same as a SQL injection vulnerability. The “skip praise” instruction eliminates the padding that wastes your reading time.

▸ Example Output (abbreviated)

 BLOCKER — Line 45: SQL Injection ```python

Current

*cursor.execute(f"SELECT * FROM users WHERE email = '{email}'")*

Fix

*cursor.execute("SELECT * FROM users WHERE email = %s", (email,)) ``` Direct string interpolation into SQL. Any user can drop your database.*

 SHOULD FIX — Lines 23-31: Unbounded Query ```python

Current

users = User.query.all() # Loads entire table into memory

Fix

users = User.query.paginate(page=page, per_page=50) ``.all()` on a growing table will eventually OOM your server.

▸ Customisation Tips

- Add your team's style guide or conventions: "We use dependency injection, not globals"
- Specify your threat model: "This handles financial data" changes the security review entirely
- For PRs, paste the diff rather than the full file — Claude will focus on what changed

---

CHAPTER 12

Prompt 10: The Security Audit

When to use: Before deploying user-facing code, authentication flows, or data handling.

▸ The Prompt

```
Perform a security audit on this {language} code. Assume an attacker who:
- Controls all user input (forms, headers, query params, file uploads)
- Can observe network traffic
- Has a valid user account (for privilege escalation checks)

{paste_code_here}

Check for:
1. Injection attacks (SQL, NoSQL, command, LDAP, template)
2. Authentication/authorisation flaws (broken access control, IDOR)
3. Data exposure (logging secrets, verbose errors, missing encryption)
4. Input validation gaps (missing sanitisation, type confusion)
5. Business logic flaws (race conditions in payments, negative quantities)

For each vulnerability:
- Severity (Critical/High/Medium/Low)
- Attack scenario (how would someone exploit this, step by step?)
- Fix (show the secure version)
- How to test the fix

Don't list theoretical issues that don't apply to this code. Only flag what's actua
```

▸ Why It Works

The attacker persona and the “step by step exploit” requirement turn this from a checklist exercise into a genuine threat analysis. The “don’t list theoretical issues” constraint prevents the common problem of getting 20 generic OWASP warnings that don’t apply.

▸ Customisation Tips

- Add your authentication method (JWT, session cookies, OAuth) for targeted analysis
- Include your middleware stack — it often handles security concerns you’ve forgotten about
- For APIs, paste your route definitions alongside the handler code
- Specify compliance requirements if relevant: “This must be PCI-DSS compliant”

CHAPTER 13

Prompt 11: The Performance Review

When to use: Code works correctly but you suspect it's slow or will be at scale.

▸ The Prompt

```
Analyse this {language} code for performance issues. The current data scale is {cur

{paste_code_here}

For each issue:
1. What's the current time/space complexity?
2. At what data scale does it become a problem? (Be specific — 1K rows? 100K? 1M?)
3. What's the optimised version and its new complexity?
4. What's the tradeoff? (Memory vs speed, readability vs performance, etc.)

Don't micro-optimise. Only flag issues that will actually matter at our expected sc

Database context (if relevant): {database_type_and_indices}
```

▸ Why It Works

The scale parameters make this concrete rather than theoretical. "This is $O(n^2)$" is academic; "This will take 45 seconds when your user table hits 50K rows" is actionable. The tradeoff requirement prevents premature optimisation.

▸ Customisation Tips

- Include your database schema and existing indices for query optimisation
- Mention caching infrastructure you already have (Redis, CDN, etc.)
- Specify latency requirements: "API responses must be under 200ms at p95"

CHAPTER 14

Prompt 12: The API Contract Review

When to use: Reviewing API endpoints before they go public.

▸ The Prompt

```
Review this API endpoint for production readiness.

Route: {method} {path}
Handler code:
{paste_code_here}

Request/response schemas (if defined):
{paste_schemas}

Check against these criteria:
1. Is the contract clear? (Could a frontend dev use this without asking you questio
2. Error handling: Does every failure path return a proper error response with stat
3. Validation: Is every input validated before use?
4. Idempotency: Is this safe to retry? If not, should it be?
5. Pagination: If this returns a list, is it bounded?
6. Versioning: Will this be painful to change later?

Show me the improved version with all issues fixed.
```

▸ Why It Works

API design mistakes are expensive to fix after launch because clients depend on the contract. This prompt covers the six most common API review failures that experienced backend developers check for. The “could a frontend dev use this” framing catches unclear contracts.

▸ Customisation Tips

- Add your API conventions: REST, GraphQL, RPC, etc.
- Include authentication middleware if it affects the endpoint
- Specify your error response format if you have a standard

CHAPTER 15

Prompt 13: The Error Handling Audit

When to use: You suspect your error handling is inconsistent or incomplete.

▸ The Prompt

```
Audit the error handling in this {language} code.

{paste_code_here}

For every operation that can fail (I/O, network, parsing, external services), answe
1. Is the failure caught?
2. Is the error message useful for debugging? (Not "Something went wrong")
3. Is the error logged with enough context to diagnose the issue from logs alone?
4. Is cleanup performed? (Connections closed, temp files removed, transactions roll
5. Is the error propagated correctly? (Not silently swallowed, not leaking internal

Show me a rewritten version with proper error handling throughout. Use {error_patte
```

▸ Why It Works

Most code reviews check *if* errors are handled. This prompt checks *how well* they're handled by testing five specific qualities of good error handling. The "diagnose from logs alone" criterion is particularly valuable — it ensures errors are logged with sufficient context.

▸ Customisation Tips

- Specify your logging framework and format (structured JSON logs, etc.)
- Add your monitoring setup: "We use Sentry" / "We use Datadog" — it affects error reporting
- Include your user-facing error format if you have one

CHAPTER 16

Prompt 14: The Test Gap Finder

When to use: You have tests, but you're not sure they cover enough.

▸ The Prompt

```
Here's my {language} code and its existing tests:

Source code:
{paste_source_code}

Existing tests:
{paste_test_code}

Identify every gap:
1. Code paths with no test coverage (be specific — which branches, which conditions
2. Edge cases not tested (boundary values, empty inputs, nulls, very large inputs)
3. Error paths not tested (what happens when dependencies fail?)
4. Integration points not tested (database calls, API calls, file I/O)

For each gap, write the missing test. Use {test_framework}.

Priority order: Start with the gaps most likely to hide bugs.
```

▸ Why It Works

Developers tend to test the happy path. This prompt systematically identifies what's *not* tested by checking four categories of coverage gaps. The priority ordering ensures you write the most valuable tests first.

▸ Customisation Tips

- Include your test fixtures/factories if you have them
- Mention your mocking strategy: "We use dependency injection" vs "We mock at the module level"
- Specify if you need unit tests, integration tests, or both

CHAPTER 17

Prompt 15: The "Explain This to a New Hire" Review

When to use: Complex code that needs to be understandable by the team.

▸ The Prompt

```
A new developer is joining our team next week. They'll need to understand this code

{paste_code_here}

1. Rate the readability from 1-10. Be honest.
2. Identify every "WTF moment" — places where a competent developer would stop and
3. For each WTF moment:
   - Is it necessary complexity (inherent to the problem)?
   - Or accidental complexity (could be simplified)?
4. Rewrite the accidental complexity parts to be self-documenting.
5. For the necessary complexity parts, add comments that explain the *why* (not the

Don't dumb down the code. Make it clear without making it verbose.
```

▸ Why It Works

"Is this readable?" is subjective. "Would a new hire understand this?" is concrete and testable. The distinction between *necessary* and *accidental* complexity prevents the common mistake of over-simplifying code that's complex because the problem is complex.

▸ Customisation Tips

- Specify the new hire's expected level: "Senior with no domain knowledge" vs "Junior developer"
- Add domain-specific jargon that should be explained: "They won't know what a 'settlement window' is"
- Include your team's naming conventions

CHAPTER 18

Prompt 16: The Naming Audit

When to use: Variables, functions, and classes that no longer describe what they do.

▸ The Prompt

```
Audit every name in this code — variables, functions, classes, parameters.

{paste_code_here}

For each name, evaluate:
1. Does it describe what the thing IS or DOES? (not how it's implemented)
2. Is it the right level of specificity? (not too vague like `data`, not too verbos
3. Is it consistent with other names in the codebase?
4. Would someone searching the codebase find this using obvious search terms?

Show me a renamed version. For every rename, explain why the new name is better.

Don't rename things that are already good. Only flag genuine improvements.
```

▸ Why It Works

Naming is the most impactful readability improvement and the easiest to fix. The search criterion (point 4) is often overlooked — great names are also great search terms.

▸ Customisation Tips

- Include your naming conventions: camelCase, snake_case, Hungarian notation, etc.
- Paste related files so Claude can check consistency across modules
- Add domain glossary terms if your project uses specific terminology

---

Part 3: Architecture & Design

Prompts for when you need to think before you code — system design, patterns, trade-offs, and decisions.

CHAPTER 19

Prompt 17: The Architecture Decision Record

When to use: You need to make a technical decision and document why.

▸ The Prompt

```
I need to make an architecture decision. Help me think through it as an ADR (Archit

Context: {describe_the_situation}
The decision I need to make: {what_you_need_to_decide}
Constraints: {time_budget_team_size_etc}

Generate a complete ADR with:
1. **Status:** Proposed
2. **Context:** Why this decision needs to be made now
3. **Options considered:** At least 3 realistic options (not strawmen)
4. **Decision matrix:** Compare options on: complexity, scalability, team familiari
5. **Recommendation:** Which option and why
6. **Consequences:** What we gain, what we give up, what we'll need to revisit late
7. **Migration path:** If we're changing from an existing approach, how do we get t

Be opinionated. Give me a recommendation, not "it depends."
```

▸ Why It Works

The “be opinionated” instruction is critical. Without it, Claude will present all options as equally valid. Real architecture decisions require someone to make a call. The decision matrix forces structured comparison rather than vibes-based decisions.

▸ Example Output (abbreviated)

Decision: Message Queue for Event Processing

CRITERION	REDIS STREAMS	RABBITMQ	KAFKA
Complexity	*Low*	*Medium*	*High*
Scalability	*Medium*	*Medium*	*Very High*
Team familiarity	*High*	*Low*	*Low*
Migration effort	*Low*	*Medium*	*High*
Operational cost	*Low*	*Medium*	*High*

Recommendation: *Redis Streams. You're processing ~5K events/day with a team of 3. Kafka is overkill. RabbitMQ is a good fit technically but nobody on the team has operated it. Redis Streams gives you at-least-once delivery, consumer groups, and your team already runs Redis. Revisit when you hit 500K events/day.*

▸ Customisation Tips

- Include your current tech stack — the best choice depends on what you already operate
- Specify your team size and skill set — this dramatically affects the recommendation
- Add your growth projections for the next 12-18 months

CHAPTER 20

Prompt 18: The System Design Collaborator

When to use: You're designing a new system or feature from scratch.

▸ The Prompt

```
I'm designing a {type_of_system} that needs to:
{list_requirements}

Current infrastructure: {what_you_already_run}
Team: {size_and_skills}
Timeline: {when_it_needs_to_work}

Walk me through the design:
1. Start with the simplest architecture that meets the requirements
2. Identify the first bottleneck it will hit
3. Show how to evolve the design to handle that bottleneck
4. Repeat for the next 2 bottlenecks

At each stage, show me:
- Component diagram (as ASCII art or Mermaid)
- Data flow for the primary use case
- What breaks if each component goes down

Don't over-engineer for day 1. Design for today but show me the evolution path.
```

▸ Why It Works

The iterative approach — "simplest first, then evolve" — matches how good systems are actually built. Most system design prompts ask for the final architecture, which leads to over-engineered solutions. This prompt produces a roadmap.

▸ Customisation Tips

- Include your SLA requirements (uptime, latency, throughput)
- Mention regulatory constraints (data residency, audit logging)
- Specify if this is greenfield or needs to integrate with existing systems

---

CHAPTER 21

Prompt 19: The Database Schema Designer

When to use: Designing or restructuring a database schema.

▸ The Prompt

```
I'm designing a database schema for {describe_domain}.

Key entities and their relationships:
{list_entities_and_how_they_relate}

Query patterns (ranked by frequency):
1. {most_common_query}
2. {second_most_common}
3. {third_most_common}

Write patterns:
1. {most_common_write}

Constraints:
- Database: {PostgreSQL/MySQL/MongoDB/etc.}
- Expected data volume: {rows_per_table_after_1_year}
- Read/write ratio: {e.g., 90/10}

Give me:
1. Complete schema with all tables, columns, types, and constraints
2. Indices that support the query patterns above (and explain WHY each index helps)
3. What this schema makes easy and what it makes hard
4. The first migration you'll need when requirements change (predict it)
```

▸ Why It Works

Starting with query patterns rather than entities produces better schemas because the schema should be optimised for how it's *used*, not how it's *modelled*. The "predict the first migration" question forces future-proofing without over-engineering.

▸ Customisation Tips

- Include sample queries if you have them
- Specify if you need multi-tenancy support
- Mention your ORM — some schemas work better with certain ORMs

---

CHAPTER 22

Prompt 20: The API Design Workshop

When to use: Designing an API from scratch or restructuring an existing one.

▸ The Prompt

```
I'm designing a {REST/GraphQL/gRPC} API for {domain_description}.

Users of this API: {who_consumes_it — frontend, mobile app, third parties, etc.}

Core operations:
{list_operations}

Design the API with:
1. Resource/endpoint structure with HTTP methods and paths
2. Request/response schemas for each endpoint (JSON examples, not just types)
3. Error response format (consistent across all endpoints)
4. Authentication and authorisation approach
5. Pagination, filtering, and sorting strategy
6. Rate limiting tiers

Follow these principles:
- Predictable: If a developer knows one endpoint, they can guess the rest
- Evolvable: We can add features without breaking existing clients
- Debuggable: Errors tell you what went wrong and how to fix it

Show me the OpenAPI/Swagger spec (YAML) for the top 5 endpoints.
```

▸ Why It Works

The three principles (predictable, evolvable, debuggable) give Claude an opinionated framework to make consistent decisions. Asking for JSON examples rather than just types produces concrete, testable contracts. The OpenAPI spec output is immediately usable.

▸ Customisation Tips

- Include existing API conventions if you have them (header naming, envelope format, etc.)

- Specify webhook requirements if the API needs push capabilities
- Add caching requirements (ETags, Cache-Control, etc.)

CHAPTER 23

Prompt 21: The Refactoring Strategy

When to use: You know the code needs restructuring but aren't sure where to start.

▸ The Prompt

```
This codebase/module has grown organically and needs restructuring. Here's the curr

{paste_code_or_describe_structure}

Current pain points:
{list_pain_points — e.g., "hard to test", "changes require touching 5 files", "new

Design a refactoring plan that:
1. Can be executed in small, safe steps (each step passes all tests)
2. Prioritises the highest-impact changes first
3. Doesn't require a feature freeze
4. Includes a rollback plan for each step

For each step:
- What changes
- What the code looks like after (show before/after)
- How to verify it worked (specific tests to run)
- Time estimate (hours, not days)

I want to chip away at this over {time_period}, not do a big rewrite.
```

▸ Why It Works

The incremental approach is the key constraint. Big rewrites fail. Small, verified steps succeed. The "doesn't require a feature freeze" constraint ensures the refactoring plan is realistic for a team that's also shipping features.

▸ Customisation Tips

- Include your test suite status: “80% coverage” vs “no tests” changes the approach
- Mention team size: solo refactoring vs coordinated across developers
- Specify your deployment frequency: continuous vs weekly releases

---

CHAPTER 24

Prompt 22: The Monolith Decomposition Planner

When to use: Breaking a monolith into services (or deciding not to).

▸ The Prompt

```
I have a {language} monolith that handles:
{list_major_features}

Pain points:
{why_you_want_to_decompose}

Team: {size} developers
Deploys: {frequency}
Database: {shared_db_details}

Before jumping to microservices, answer honestly:
1. Would a well-structured modular monolith solve these pain points? Why or why not
2. If services are the right answer, which boundary should I extract FIRST? (Based
3. How do I handle the shared database? (Strangler fig? Separate DBs? Shared with s
4. What's the communication pattern? (Sync REST/gRPC? Async events? Both?)
5. What operational complexity am I signing up for? (Be brutally honest)

Give me a 6-month roadmap with milestones, not a theoretical architecture diagram.
```

▸ Why It Works

The “answer honestly” framing and question 1 prevent the common trap of decomposing for the sake of it. Most monoliths should stay monoliths. This prompt forces Claude to justify decomposition before

planning it.

▸ Customisation Tips

- Include your monitoring and observability setup — it constrains what's operationally feasible
- Mention your CI/CD pipeline maturity
- Specify team structure: one team or multiple teams with different domains

---

CHAPTER 25

Prompt 23: The Design Pattern Advisor

When to use: You have a recurring problem and suspect there's a pattern for it.

▸ The Prompt

```
I keep running into this problem in my {language} codebase:

{describe_the_recurring_problem}

Here's a concrete example:
{paste_example_code}

1. Is there a design pattern that solves this? (Name it, but don't lecture me on th
2. Show me the pattern applied to MY code (not a generic Animal/Dog example)
3. Show me what the code looks like BEFORE and AFTER the pattern is applied
4. What are the downsides? When should I NOT use this pattern?
5. Is there a simpler solution that doesn't require a formal pattern?

If the simpler solution is better, just give me that. I don't want patterns for the
```

▸ Why It Works

Question 5 is the secret weapon. Most pattern advice jumps straight to the Gang of Four without considering whether a simple function or a dictionary lookup would do the job. This prompt ensures you get the right level of abstraction.

▸ Customisation Tips

- Paste multiple examples of the recurring problem — patterns become clearer with more instances
- Include your team's experience level: "We're all senior" vs "Mixed levels" affects the recommendation
- Mention if you're using a framework that might already provide the pattern

---

CHAPTER 26

Prompt 24: The Migration Planner

When to use: Moving between databases, frameworks, languages, or cloud providers.

▸ The Prompt

```
I'm migrating from {current_tech} to {target_tech}.

Current state:
- Codebase size: {lines_of_code_or_file_count}
- Usage of current tech: {how_deeply_integrated}
- Data volume: {if_database_migration}

Why we're migrating: {business_reasons}
Deadline: {when}

Create a migration plan that:
1. Runs both old and new in parallel during transition (no big bang)
2. Has a verification step at each stage (how do we know the migration is correct?)
3. Includes a rollback plan (what if we need to go back at each stage?)
4. Identifies the riskiest step (and how to de-risk it)
5. Accounts for ongoing development (features don't stop during migration)

What automated tooling can help with this specific migration?
What's the one thing teams always underestimate when doing this migration?
```

▸ Why It Works

The final two questions surface practical wisdom. The “one thing teams always underestimate” question consistently produces the most valuable insight — it’s the kind of thing you’d only learn from someone who’s done this migration before.

▸ Customisation Tips

- Include your testing strategy — it determines how confidently you can verify the migration
- Mention data sensitivity: “This is financial data” requires more careful migration
- Specify if the migration needs to be invisible to users or if you can have planned downtime

---

Part 4: Documentation

Prompts that produce documentation people actually read — because if nobody reads it, it doesn’t exist.

CHAPTER 27

Prompt 25: The README Generator

When to use: Your project needs a README that doesn’t make people close the tab.

▸ The Prompt

```
Write a README for this project. It should make a developer go from "what is this?"

Project: {name}
Language: {language}
What it does: {one_paragraph_description}
Key source files: {paste_main_entry_point_or_describe_structure}

Structure the README as:
1. **One-line description** (what it does, not what it is)
2. **Quick start** (3-5 commands to go from clone to running — tested, not theoreti
3. **Why this exists** (what problem it solves, 2-3 sentences max)
4. **Core concepts** (the 3-5 things someone needs to understand to use this effect
5. **Configuration** (table of env vars / config options with defaults and descript
6. **Examples** (3 real-world usage examples, increasing in complexity)
7. **Troubleshooting** (top 5 things that go wrong and how to fix them)
8. **Contributing** (how to run tests, code style, PR process)

Rules:
- No badges unless they're genuinely informative (build status yes, "made with love
- Code examples must be copy-pasteable (no `...` truncation)
- Every command must include expected output
```

▸ Why It Works

The 5-minute goal is the key constraint. Most READMEs are either a one-liner or a novel. This structure gives developers the fastest path to productivity while still being comprehensive. The “code examples must be copy-pasteable” rule catches the most common README sin.

▸ Customisation Tips

- Include your actual setup commands so Claude can verify they work
- Paste your `.env.example` for the configuration section
- Add your most common support questions for the troubleshooting section

---

CHAPTER 28

Prompt 26: The Code Documentation Writer

When to use: Complex code that needs inline documentation.

▸ The Prompt

```
Add documentation to this {language} code. Follow these rules strictly:

{paste_code_here}

Rules:
1. Document the WHY, not the WHAT. If the code says `x = x + 1`, don't write "incre
2. Every function/method gets a docstring with: purpose, parameters (with types and
3. Complex algorithms get a block comment explaining the approach and why alternati
4. Magic numbers become named constants with explanatory comments
5. TODO/FIXME comments include: who should fix it, why it's deferred, and what the

Don't over-document simple code. A clear name is better than a comment.

Output format: The complete code with documentation added. Don't separate docs fron
```

▸ Why It Works

Rule 1 alone makes this prompt 10x better than “add comments to this code.” The explicit instruction to not document the obvious prevents the clutter that makes developers ignore documentation entirely.

▸ Customisation Tips

- Specify your docstring format: Google style, NumPy style, JSDoc, Javadoc, etc.
- Add “This code will be read by {audience}” for targeted detail level
- Include your documentation linter rules if you have them (e.g., `pydocstyle`)

CHAPTER 29

Prompt 27: The Changelog Writer

When to use: You need to communicate what changed to users who don’t read code.

▸ The Prompt

```
Here are the git commits/changes since the last release:

{paste_git_log_or_diff_summary}

Write a changelog entry for version {version} that:
1. Groups changes into: Added, Changed, Fixed, Removed, Security, Deprecated
2. Writes each entry from the USER's perspective, not the developer's perspective
3. Leads with impact: "You can now..." not "Implemented feature..."
4. For breaking changes: explains exactly what the user needs to change, with befor
5. Includes migration steps if needed

Bad: "Refactored authentication module"
Good: "Sign-in now works with magic links — no password required. Existing passwor

Skip internal changes (refactors, dependency updates, CI fixes) unless they affect
```

▸ Why It Works

The bad/good example is the most important part of this prompt. It teaches Claude the *voice* you want. The “user’s perspective” instruction transforms technical git messages into readable changelogs.

▸ Customisation Tips

- Specify your audience: developers using your API vs end users vs both
- Include your previous changelog entries for consistent tone
- Add “Keep it under {n} entries — combine related changes”

CHAPTER 30

Prompt 28: The Runbook Writer

When to use: You need operational documentation for incidents and deployments.

▸ The Prompt

```
Write a runbook for: {scenario — e.g., "Database failover", "Deploying to productio

Context:
- Infrastructure: {describe_setup}
- Monitoring: {what_tools_you_use}
- On-call rotation: {how_it_works}

The runbook must be usable by someone who:
- Has never seen this system before
- Is being paged at 3 AM
- Has access to {list_of_tools_and_dashboards}

Structure:
1. **Symptoms:** How do you know this is happening? (Specific alerts, metrics, logs
2. **Impact:** What's affected and how urgently? (Decision tree: page vs wait until
3. **Diagnosis:** Step-by-step commands to determine the specific cause
4. **Resolution:** Step-by-step fix for each likely cause (with copy-pasteable comm
5. **Verification:** How to confirm it's actually fixed (not just "it stopped error
6. **Follow-up:** What to do after the immediate fix (post-mortem, preventive measu

Every command must include expected output. Every step must include "if this doesn'
```

▸ Why It Works

The “3 AM, never seen this system” constraint is the quality test. If a runbook can’t be followed by a sleep-deprived engineer who’s never touched the system, it’s not a runbook — it’s a knowledge dump. The fallback instructions (“if this doesn’t work”) are what separate good runbooks from mediocre ones.

▸ Customisation Tips

- Include your actual monitoring dashboard URLs
- Add your escalation path: “If step 3 doesn’t resolve it, page {person}”
- Specify your communication protocol: “Post updates in #incidents every 15 minutes”

CHAPTER 31

Prompt 29: The ADR From Existing Code

When to use: You need to retroactively document why code was written a certain way.

▸ The Prompt

```
This code exists but nobody documented why it was built this way. I need a retroact

{paste_code_here}

Based on the code itself, reverse-engineer:
1. What problem was this solving?
2. What were the likely alternatives? (What would a different developer have done?)
3. Why was this approach chosen over the alternatives? (Infer from the code's chara
4. What are the trade-offs of this approach?
5. Under what conditions should we revisit this decision?

Be transparent about what you're inferring vs what's obvious from the code. Mark in

Also flag any "this looks like it was written in a hurry" indicators — they suggest
```

▸ Why It Works

Legacy codebases are full of undocumented decisions. This prompt uses the code itself as evidence to reconstruct the reasoning. The “mark inferences clearly” instruction maintains intellectual honesty — you don’t want false confidence in the reconstructed reasoning.

▸ Customisation Tips

- Include git blame / commit messages if available — they provide additional context
- Paste related code or tests that might reveal intent
- Add “The original author is no longer on the team” for Claude to be more speculative

---

CHAPTER 32

Prompt 30: The Architecture Diagram Describer

When to use: You need clear text-based architecture documentation.

▸ The Prompt

```
I need architecture documentation for my system. Here's what we run:

Components:
{list_components_and_what_they_do}

How they communicate:
{describe_data_flows}

Create:
1. A Mermaid diagram showing all components and their interactions
2. A text description of each component (what it does, what it depends on, what dep
3. The critical path (which failures cause the most damage?)
4. A "where does data live?" summary (which component is the source of truth for wh

Keep the Mermaid diagram readable — max 15 nodes. If the system is larger, create m
```

▸ Why It Works

Mermaid diagrams are version-controllable, diff-able, and render in GitHub/GitLab. The 15-node limit prevents unreadable spaghetti diagrams. The “source of truth” summary is consistently the most useful part of architecture documentation.

▸ Customisation Tips

- Specify sync vs async communication for each connection
- Include your deployment topology: “These three services run on the same box”
- Add SLA requirements per component: “This must be up 99.9%”

CHAPTER 33

Prompt 31: The Technical RFC Writer

When to use: Proposing a significant technical change to your team.

▸ The Prompt

```
I want to propose the following technical change to my team:

{describe_the_change}

Current state: {how_things_work_now}
Motivation: {why_change_is_needed}

Write a technical RFC that will convince sceptical senior engineers. Include:

1. **Summary** (2-3 sentences a busy person can skim)
2. **Motivation** (the specific pain this addresses, with data/examples if possible
3. **Detailed Design** (how it works, with enough detail to implement without follo
4. **Alternatives Considered** (at least 2, with genuine analysis of why they were
5. **Migration Plan** (how we get from here to there without breaking things)
6. **Risks & Mitigations** (what could go wrong, how we'll detect it, how we'll rol
7. **Open Questions** (what you don't know yet and need input on)
8. **Success Metrics** (how we'll know this worked, measured 3 months after launch)

Tone: Technical but not academic. Confident but not arrogant. Acknowledge trade-off
```

▸ Why It Works

The “convince sceptical senior engineers” framing sets the quality bar correctly. RFCs that don’t address counterarguments get rejected. The success metrics section is often omitted but is crucial — it makes the proposal accountable.

▸ Customisation Tips

- Include your team’s RFC template if you have one
- Add known objections: “The team will probably push back on X because...”
- Specify your decision-making process: “We need consensus” vs “Tech lead decides”

Part 5: Testing

From unit tests to property-based testing — prompts that produce tests developers actually want to maintain.

CHAPTER 34

Prompt 32: The Unit Test Writer

When to use: You need comprehensive unit tests for a function or class.

▸ The Prompt

```
Write unit tests for this {language} code using {test_framework}.

{paste_code_here}

Cover these categories:
1. **Happy path:** Normal, expected inputs produce correct output
2. **Edge cases:** Empty inputs, boundary values, single-element collections, very
3. **Error cases:** Invalid inputs, null/undefined, wrong types, missing required f
4. **State transitions:** If the code modifies state, test before/after for each op

Rules:
- Each test tests ONE thing (single assertion unless testing a sequence)
- Test names describe the behaviour, not the method: `test_rejects_negative_quantit
- No test should depend on another test's output
- Include setup/teardown if needed, but explain what it does
- Mock external dependencies, don't mock the code under test

Generate at least {number} tests. Group them logically.
```

▸ Why It Works

The four categories ensure systematic coverage. The naming rule produces self-documenting tests that serve as executable specifications. The “don’t mock the code under test” rule prevents the common mistake of testing mocks instead of actual behaviour.

▸ Customisation Tips

- Include your test utilities and factories
- Specify your mocking library and approach

- Add “Use parameterised tests for variations of the same scenario” for cleaner test suites
- Include your CI constraints: “Tests must complete in under 30 seconds”

CHAPTER 35

Prompt 33: The Integration Test Designer

When to use: Testing how components work together (API routes, database operations, etc.).

▸ The Prompt

```
Design integration tests for this {language} {component_type} (e.g., API endpoint,

Source code:
{paste_code_here}

External dependencies this code touches:
{list_databases_apis_queues_etc}

For each integration test:
1. What's being tested (the integration boundary, not business logic)
2. Setup: What state needs to exist before the test (database seeds, mock server co
3. Action: The exact call being made
4. Assertions: What to check (response, database state, side effects)
5. Cleanup: How to reset state after the test

Handle these scenarios:
- Happy path through the full integration
- External service is down/slow (timeout, 500 error)
- Data integrity (concurrent modifications, partial failures)
- Authentication/authorisation at the integration level

Use {test_framework}. Use {real_or_containerised} dependencies where possible, mock
```

▸ Why It Works

Integration tests are harder than unit tests because they require managing state across components. This prompt explicitly addresses setup and cleanup, which are where most integration tests go wrong. The “mock only when necessary” instruction produces more realistic tests.

▸ Customisation Tips

- Include your Docker Compose / Testcontainers setup
- Specify your test database strategy: shared, per-test, per-suite
- Add your CI environment constraints: “We can’t run Docker in CI”

CHAPTER 36

Prompt 34: The Property-Based Test Generator

When to use: You want tests that find bugs you didn’t think of.

▸ The Prompt

```
Generate property-based tests for this {language} code using {property_testing_libr

{paste_code_here}

For each function/method, identify:
1. **Invariants:** What must ALWAYS be true regardless of input? (e.g., "encode the
2. **Relationships:** How do outputs relate to inputs? (e.g., "sort output has same
3. **Equivalences:** Is there a simpler (slower) reference implementation to compar
4. **Boundaries:** What constraints must outputs satisfy? (e.g., "result is never n

For each property:
- Write the test with appropriate generators
- Include a shrinking strategy for failures
- Set a sensible example count (not too few to catch bugs, not too many to slow CI)

Also identify inputs that should be specifically included in the strategy (known ed
```

▸ Why It Works

Property-based testing is the most effective testing technique most developers don't use. This prompt bridges the gap by teaching Claude to identify properties from code, which is the hardest part. The four property categories (invariants, relationships, equivalences, boundaries) provide a systematic framework.

▸ Customisation Tips

- Include your custom data types — Claude can generate composite strategies
- Specify CI time budget: "Property tests must run in under 60 seconds"
- Add known edge cases your codebase has hit before

CHAPTER 37

Prompt 35: The Test Data Factory

When to use: You need realistic test data that's easy to create and maintain.

▸ The Prompt

```
Create a test data factory for my {language} domain models.

Models:
{paste_model_definitions}

Requirements:
1. Every factory produces valid data by default (passes all validations)
2. Every field can be overridden: `build_user(email="specific@test.com")`
3. Relationships are handled: `build_order()` automatically creates a valid user
4. Sequences are used for unique fields (email_1@test.com, email_2@test.com)
5. Edge case factories for specific test scenarios: `build_expired_subscription()`,

Use {factory_library — e.g., Factory Boy, Fishery, ExMachina} if applicable, otherw

Also create:
- A seed script that populates a test database with a realistic dataset
- Documentation showing how to use the factories in tests
```

▸ Why It Works

Test data is the foundation of a good test suite. Without factories, every test constructs its own data, leading to brittle, repetitive test code. The “valid by default” rule ensures factories are trustworthy — you only override what the specific test cares about.

▸ Customisation Tips

- Include your validation rules so the factory respects them
- Add your most common test scenarios: “We often test users with 0, 1, and many orders”
- Specify locale requirements: “Addresses should be UK format”

CHAPTER 38

Prompt 36: The Mutation Testing Analyst

When to use: You have tests but aren’t sure they actually catch bugs.

▸ The Prompt

```
I have this {language} code and these tests. Analyse the test quality through mutat

Source code:
{paste_source_code}

Tests:
{paste_tests}

Manually apply these mutation categories to the source code:
1. **Boundary mutations:** Change `<` to `⩽`, `>` to `⩾`
2. **Value mutations:** Change `+1` to `-1`, `0` to `1`, `true` to `false`
3. **Removal mutations:** Remove function calls, null checks, error handling
4. **Logic mutations:** Swap `&&` and `||`, negate conditions
5. **Return mutations:** Return null/empty instead of the computed value

For each mutation:
- Show the mutated code
- Does any existing test catch it? (Cite the specific test)
- If not, write a test that would catch it

Score: What percentage of mutations are caught? Which areas are undertested?
```

▸ Why It Works

Mutation testing measures the *effectiveness* of your tests, not just coverage. A test suite can have 100% line coverage but still miss bugs if the assertions are weak. This prompt simulates what mutation testing tools do, but with explanations.

▸ Customisation Tips

- Focus on critical code paths: “Only mutate the payment processing code”
- Include your mutation testing tool config if you have one (Stryker, mutmut, Pitest)
- Specify your survival threshold: “No mutation should survive in authentication code”

CHAPTER 39

Prompt 37: The Contract Test Writer

When to use: Testing API contracts between services.

▸ The Prompt

```
Write contract tests for this API interaction:

Consumer (the service making the call):
{describe_or_paste_consumer_code}

Provider (the service being called):
{describe_or_paste_provider_endpoint}

Current contract (request/response examples):
{paste_example_request_response}

Generate:
1. **Consumer-side contract tests** that verify:
   - We send the right request format
   - We handle every documented response correctly (success AND errors)
   - We handle undocumented responses gracefully (500, timeout, malformed JSON)

2. **Provider-side contract tests** that verify:
   - We respond to valid requests with the documented format
   - We reject invalid requests with proper error responses
   - Our response schema matches what consumers expect

3. **Breaking change detection:**
   - Tests that would FAIL if either side makes a breaking change
   - Categorise changes as breaking vs non-breaking

Use {contract_testing_framework — e.g., Pact, Spring Cloud Contract}.
```

▸ Why It Works

Contract tests are the most practical way to test service boundaries without running the entire system. The “breaking change detection” section is particularly valuable — it turns contract tests into a change management tool.

▸ Customisation Tips

- Include your API versioning strategy
- Specify your contract broker setup (Pact Broker, etc.)

- Add “Include backward compatibility tests for the last 2 versions”

CHAPTER 40

Prompt 38: The End-to-End Test Scenario Writer

When to use: Defining E2E test scenarios for critical user flows.

▸ The Prompt

```
Write end-to-end test scenarios for this critical user flow:

Flow: {describe_the_user_journey — e.g., "User signs up, verifies email, creates fi

System components involved:
{list_frontend_backend_db_email_etc}

Write test scenarios in Gherkin (Given/When/Then) for:
1. The happy path (everything works perfectly)
2. Each failure point (what happens when each component fails?)
3. Concurrent users (what if two users do this simultaneously?)
4. Recovery (user starts the flow, gets interrupted, comes back later)
5. Edge cases (back button, double click, expired session, browser refresh)

For each scenario:
- Specify what to assert (not just "it works" — what specifically should be true?)
- Specify what to clean up after
- Flag which scenarios are stable enough for CI vs which should be manual/scheduled

Then translate the top 5 scenarios into {e2e_framework — e.g., Playwright, Cypress,
```

▸ Why It Works

E2E tests are expensive to write and maintain, so choosing the right scenarios is critical. This prompt ensures you cover not just the happy path but also the realistic failure modes that users actually encounter (back button, double click, expired session).

▸ Customisation Tips

- Include your feature flags — E2E tests need to account for different feature states
- Specify your test environment: staging URL, test accounts, etc.
- Add flakiness tolerance: "These run in CI — they cannot be flaky"

CHAPTER 41

Prompt 39: The Test Refactoring Guide

When to use: Your test suite is slow, flaky, or hard to maintain.

▸ The Prompt

```
My test suite has become a liability. Here's a representative sample:

{paste_problematic_tests}

Problems I'm experiencing:
{list_problems — e.g., "takes 20 minutes", "3 tests flake randomly", "adding a fiel

Analyse and fix:
1. **Coupling:** Which tests break when unrelated code changes? Extract those into
2. **Duplication:** Where is setup duplicated? Extract shared fixtures/factories.
3. **Brittleness:** Which tests test implementation details instead of behaviour? R
4. **Speed:** Which tests are slow and why? Can any integration tests become unit t
5. **Flakiness:** What's causing non-determinism? (Time, randomness, shared state,

Give me:
- Refactored test code
- A recommended test categorisation (fast unit / integration / slow E2E)
- A CI strategy (what runs on every push vs nightly)
```

▸ Why It Works

Test suites degrade just like production code, but most teams never refactor them. This prompt addresses the five most common test suite pathologies with specific fixes. The CI strategy output is

immediately actionable.

▸ Customisation Tips

- Include your CI timing requirements
- Specify your test parallelisation setup
- Add “We’re migrating from {old_framework} to {new_framework}” if applicable

---

Part 6: Refactoring

Prompts for improving existing code without changing what it does — safely, incrementally, and with confidence.

CHAPTER 42

Prompt 40: The Code Smell Identifier

When to use: You know code is off but can’t articulate why.

▸ The Prompt

```
Identify code smells in this {language} code. I want a diagnosis, not generic advic

{paste_code_here}

For each smell:
1. Name it (use established terminology: God class, feature envy, shotgun surgery,
2. Show the exact lines that exhibit it
3. Explain the CONCRETE consequence (not "it's bad practice" — what specific proble
4. Rate the urgency: Fix now / Fix soon / Fix when you're in the area
5. Show the refactored version

Only flag smells that have practical consequences. Skip style-level nitpicks.
```

▸ Why It Works

The “concrete consequence” requirement eliminates the common problem of listing code smells as abstract principles. “Long method” means nothing; “this function handles user validation AND payment processing AND email sending, so any bug fix requires understanding all three domains” is actionable.

▸ Customisation Tips

- Include the broader module structure for “shotgun surgery” and “divergent change” detection
- Specify your team’s tolerance: “We prefer explicit over clever”
- Add your codebase age: “This is 5 years old” helps prioritise legacy vs recent smells

---

CHAPTER 43

Prompt 41: The Extract and Inject Refactorer

When to use: Making code testable by extracting dependencies.

▸ The Prompt

```
This {language} code is hard to test because it has embedded dependencies (database

{paste_code_here}

Refactor it using dependency injection:
1. Identify every external dependency (anything that touches the outside world)
2. Extract each into an interface/protocol/abstract class
3. Inject dependencies through the constructor
4. Show the refactored production code
5. Show a test that uses mock/fake implementations
6. Show the wiring code (where dependencies are assembled)

Rules:
- Don't create interfaces for things that don't need to be swapped (pure utility fu
- Keep the public API identical — callers shouldn't need to change
- Use {di_approach — e.g., "constructor injection", "function parameters", "depende
```

▸ Why It Works

Dependency injection is the most impactful refactoring technique for testability, but developers often over-apply it. The “don’t create interfaces for things that don’t need to be swapped” rule prevents the common over-abstraction that makes DI-heavy codebases hard to navigate.

▸ Customisation Tips

- Specify your DI framework if you use one (Spring, Dagger, Inversify, etc.)
- Include existing tests so the refactoring can be verified
- Add “This code is called from {n} places” to understand the blast radius

---

CHAPTER 44

Prompt 42: The Performance Refactorer

When to use: Code is correct but slow and needs optimisation.

▸ The Prompt

```
This {language} code is too slow. Current performance: {current_metrics — e.g., "3
Target performance: {target — e.g., "under 500ms for 100K rows"}.

{paste_code_here}

Optimise it:
1. Profile the code conceptually — where is time being spent? (Use Big-O analysis)
2. Identify the single biggest bottleneck
3. Fix that bottleneck and show the new code
4. If still not at target, repeat for the next bottleneck
5. Stop when we reach the target or hit diminishing returns

For each optimisation:
- Show before/after code
- Explain the expected speedup with reasoning (don't guess — show the math)
- Note any tradeoffs (memory usage, code complexity, maintainability)

Constraints:
- Keep the same public interface
- Don't sacrifice correctness (especially edge cases)
- {additional_constraints — e.g., "must work with streaming data", "cannot add new
```

▸ Why It Works

The iterative approach (fix one bottleneck, measure, repeat) matches real-world performance optimisation. The “show the math” requirement prevents wishful thinking about speedups. Stopping at “diminishing returns” prevents over-optimisation.

▸ Customisation Tips

- Include profiling data if you have it (cProfile output, flame graphs)
- Specify your hardware constraints: “This runs on a 2-core VM with 4GB RAM”
- Add your scaling model: “Data grows linearly at 10K rows/month”

CHAPTER 45

Prompt 43: The Legacy Code Moderniser

When to use: You've inherited old code and need to bring it up to modern standards.

▸ The Prompt

```
This {language} code was written for {old_version — e.g., "Python 2.7", "Node 10",

{paste_code_here}

Modernise it:
1. Replace deprecated APIs with their modern equivalents
2. Apply modern language features that improve clarity (async/await, pattern matchi
3. Replace hand-rolled implementations with standard library equivalents
4. Update error handling to modern patterns
5. Fix any security issues that exist in the old patterns

For each change:
- Show before/after
- Explain what's better about the modern version (not just "it's newer")
- Flag any behaviour changes (however subtle)

Don't change the logic. Don't add features. Just modernise the implementation.
```

▸ Why It Works

The "don't change the logic" constraint is essential. Modernisation and feature changes mixed together are impossible to review. By isolating the modernisation, each change can be verified independently.

▸ Customisation Tips

- Include your linter configuration — it may already flag some of these
- Specify your minimum supported version: "Must still work on Node 18+"
- Add your migration timeline: "We're doing this file by file over 3 sprints"

CHAPTER 46

Prompt 44: The Dead Code Eliminator

When to use: Clearing out code nobody uses any more.

▸ The Prompt

```
Analyse this {language} code for dead code — anything that's defined but never used

{paste_code_here}

Entry points (these are the roots — everything else must be reachable from these):
{list_entry_points — e.g., "API routes in routes.py", "main() function", "exported

Identify:
1. **Unreachable code:** Functions/classes/variables that nothing calls
2. **Dead branches:** Conditions that can never be true (based on the type system o
3. **Feature flags that are always on/off** (permanent flags that can be removed)
4. **Compatibility code for versions we no longer support**
5. **Commented-out code** (it's in git history — just delete it)

For each piece of dead code:
- Show what can be safely removed
- Explain why it's dead (not just "it's unused" — why did it become unused?)
- Flag anything that LOOKS dead but might be used via reflection, dynamic dispatch,

Give me the confidence level for each removal: Certain / Probable / Needs verificat
```

▸ Why It Works

The confidence level is crucial for dead code removal. Some "dead" code is actually accessed dynamically (reflection, metaprogramming, external scripts). The entry points framing gives Claude the context to trace reachability accurately.

▸ Customisation Tips

- Include your test suite — tests that cover dead code are also dead tests
- Mention frameworks that use reflection/convention (Rails, Django, Spring) — they hide usage
- Add "Check if this is used by any cron job or background worker"

CHAPTER 47

Prompt 45: The Error Handling Moderniser

When to use: Upgrading scattered try/catch blocks into a coherent error handling strategy.

▸ The Prompt

```
My {language} codebase has inconsistent error handling. Some functions throw, some

Here's a representative sample:
{paste_code_with_various_error_patterns}

Design a consistent error handling strategy and refactor this code:

1. Classify errors into categories: {e.g., "validation errors (user's fault)", "inf
2. Define a custom error hierarchy for each category
3. Establish clear rules: when to throw, when to return Result/Either types, when t
4. Refactor the code to follow the strategy consistently
5. Add an error boundary/handler at the top level that converts errors to appropria

Rules:
- Every error must be either handled or explicitly propagated (no silent swallowing
- Error messages must be useful for debugging (include context: what was attempted,
- User-facing error messages must be safe (no stack traces, no internal details)
```

▸ Why It Works

Error handling inconsistency is one of the most common codbase issues, and it's usually addressed piecemeal. This prompt takes a strategic approach — designing the error system top-down, then refactoring bottom-up.

▸ Customisation Tips

- Include your logging and monitoring setup — errors should flow into observability
- Specify your API error format (RFC 7807, custom envelope, etc.)
- Add your error tracking service (Sentry, Bugsnag, Rollbar) — integration affects the strategy

Part 7: Performance & Optimisation

Prompts for when speed matters — database queries, algorithms, caching, and profiling.

CHAPTER 48

Prompt 46: The Query Optimiser

When to use: Slow database queries that need tuning.

▸ The Prompt

```
This {database} query is too slow.

Query:
{paste_query}

EXPLAIN/EXPLAIN ANALYSE output:
{paste_explain_output}

Table schema and current indices:
{paste_schema}

Table sizes: {approximate_row_counts}

Optimise this query:
1. Analyse the EXPLAIN output — where is time being spent?
2. Is the query plan doing sequential scans where it shouldn't be?
3. Are there missing indices? (Show the exact CREATE INDEX statement)
4. Can the query be rewritten to be more efficient? (Show the new query)
5. Would denormalisation or materialised views help? (Only if the query fundamental

Show me the expected improvement with reasoning. Don't guess — explain why the opti
```

▸ Why It Works

Including the EXPLAIN output is what separates this from generic “make my query faster” prompts. It gives Claude the actual execution plan to diagnose, not just the SQL to guess about. The “don’t guess” instruction forces reasoned analysis.

▸ Customisation Tips

- Include your PostgreSQL/MySQL configuration (work_mem, shared_buffers, etc.)
- Mention your read/write ratio for the tables involved
- Add “This query runs {n} times per second” for context on how much optimisation is worth

---

CHAPTER 49

Prompt 47: The Caching Strategy Designer

When to use: You need to add caching to improve performance.

▸ The Prompt

```
I need a caching strategy for this {language} application.

What's slow:
{describe_slow_operations}

Current architecture:
{describe_components}

Available caching infrastructure: {Redis/Memcached/CDN/in-process/none}

Design a caching strategy:
1. What to cache (and what explicitly NOT to cache — some things shouldn't be)
2. Cache keys: How to construct them to be unique and predictable
3. TTL strategy for each cached item (and why that TTL was chosen)
4. Invalidation strategy: How does the cache learn the data changed?
5. Cache warming: Should we pre-populate on startup? On deploy?
6. Failure mode: What happens when the cache is down? (Degrade gracefully, don't cr

Address these specific risks:
- Cache stampede (1000 requests all miss at the same time)
- Stale data (how stale is acceptable for each data type?)
- Memory budget (how much RAM does this caching strategy need?)

Show me the implementation code for the two most impactful caches.
```

▸ Why It Works

Caching is easy to add and hard to get right. This prompt addresses the full lifecycle — not just “put it in Redis” but invalidation, failure modes, and stampede protection. The “what NOT to cache” question prevents the common mistake of caching mutable, high-consistency data.

▸ Customisation Tips

- Include your data update frequency: “Prices change every 5 minutes” vs “User profiles change monthly”
- Specify your consistency requirements: “Eventual consistency is fine” vs “Must be strongly consistent”
- Add your memory budget: “We have 2GB Redis with 500MB available”

CHAPTER 50

Prompt 48: The N+1 Query Detector

When to use: Your ORM is generating too many database queries.

▸ The Prompt

```
Analyse this {language} code using {ORM} for N+1 query problems.

{paste_code_here}

Models/relationships:
{paste_model_definitions_or_describe_relationships}

For each N+1 issue found:
1. Show the code that causes it
2. Show the SQL queries it generates (approximate)
3. Calculate how many queries it generates for N={realistic_number} records
4. Show the fix (eager loading, batch loading, query rewrite)
5. Show the SQL the fix generates (should be 1-2 queries regardless of N)

Also check for:
- Lazy loading in loops (the classic N+1)
- Serialisation-triggered loads (accessing relationships in JSON serialisation)
- Count queries in loops (checking `has_many` count for each record)

Use {ORM_specific_eager_loading — e.g., "select_related/prefetch_related", "include
```

▸ Why It Works

N+1 queries are the most common ORM performance problem, and they're invisible until you check. This prompt not only finds them but quantifies the impact and shows the ORM-idiomatic fix.

▸ Customisation Tips

- Include your query logging output if you have it
- Mention your ORM version — eager loading APIs change between versions
- Add "We use a GraphQL layer" — GraphQL adds unique N+1 challenges (DataLoader pattern)

CHAPTER 51

Prompt 49: The Bundle Size Reducer

When to use: Your frontend bundle is too large and you need to slim it down.

▸ The Prompt

```
My frontend bundle is {current_size} and needs to be under {target_size}.

Here's my package.json:
{paste_package_json}

Here's my webpack/vite/rollup config:
{paste_config}

Bundle analysis output (if available):
{paste_bundle_analysis}

Reduce the bundle size:
1. Identify the largest dependencies and suggest smaller alternatives
2. Find code that's included but never used (tree-shaking failures)
3. Identify dependencies that should be lazy-loaded instead of included in the mair
4. Suggest code-splitting opportunities (route-based, feature-based)
5. Check for duplicate packages (different versions of the same library)

For each suggestion:
- Expected size reduction (be specific — "removes 45KB gzipped")
- Migration effort (trivial / moderate / significant)
- Risk level (drop-in replacement / needs testing / behaviour change)

Priority order: biggest size reduction with lowest effort first.
```

▸ Why It Works

Bundle size is a concrete, measurable target. The priority ordering (biggest reduction, lowest effort) ensures you tackle the high-impact wins first. Including the bundle analysis output gives Claude real data instead of requiring guesswork.

▸ Customisation Tips

- Include your target devices: “Must work on 3G mobile connections” changes acceptable sizes
- Specify your code-splitting framework (React.lazy, dynamic imports, etc.)
- Add your CDN caching strategy — it affects the tradeoffs of splitting

---

Part 8: DevOps & CI/CD

Prompts for pipelines, deployments, infrastructure, and the scripts that keep everything running.

CHAPTER 52

Prompt 50: The CI Pipeline Designer

When to use: Setting up or improving your CI/CD pipeline.

▸ The Prompt

```
Design a CI/CD pipeline for my {language} {framework} project.

Repository structure: {monorepo_or_standard}
Team size: {number_of_developers}
Deployment target: {where_it_runs}
Current CI: {what_you_have_now_or_starting_from_scratch}

Design a pipeline with these stages:
1. **Fast feedback** (< 2 min): What runs on every push to catch obvious issues?
2. **Full validation** (< 10 min): What runs on every PR before merge?
3. **Pre-deploy** (< 5 min): What validates the build before deployment?
4. **Deployment**: How does code get to production? (Blue/green, rolling, canary?)
5. **Post-deploy**: How do we verify the deployment worked? (Smoke tests, health ch

For each stage, specify:
- Exact steps (with commands)
- What triggers it
- What blocks if it fails
- Caching strategy (to keep it fast)

Write the pipeline config for {ci_platform — e.g., GitHub Actions, GitLab CI, Circl

Include:
- Branch protection rules
- Required checks before merge
- Automatic rollback triggers
```

▸ Why It Works

The time budgets per stage are the key constraint. Without them, pipelines grow until they take 45 minutes and everyone ignores them. The five-stage structure matches how production-grade pipelines actually work.

▸ Customisation Tips

- Include your test suite timing breakdown
- Specify your deployment frequency target: “We want to deploy 5x per day”
- Add your compliance requirements: “We need approval gates for production”

CHAPTER 53

Prompt 51: The Dockerfile Optimiser

When to use: Your Docker builds are slow or your images are too large.

▸ The Prompt

```
Optimise this Dockerfile for build speed and image size.

Current Dockerfile:
{paste_dockerfile}

Current image size: {size}
Current build time: {time}
Target image size: {target_size}
Target build time: {target_time}

Optimise:
1. Layer ordering: What should change order to maximise cache hits?
2. Multi-stage build: What can be built in a separate stage and not included in the
3. Base image: Is there a smaller base that works? (Alpine, distroless, slim?)
4. Dependencies: What's installed but not needed at runtime?
5. COPY strategy: What files are copied that trigger unnecessary rebuilds?
6. Security: Is anything in the image that shouldn't be? (Secrets, build tools, tes

Show me the optimised Dockerfile with comments explaining each decision.
Also provide a .dockerignore that prevents unnecessary context from being sent.
```

▸ Why It Works

Docker optimisation has well-known best practices, but applying them to a specific Dockerfile requires understanding the build context. This prompt addresses the six most impactful optimisation categories in order of typical impact.

▸ Customisation Tips

- Include your CI caching setup — it affects the layer ordering strategy
- Mention your target platform: “Must run on ARM64” or “x86 only”

- Add security requirements: “This runs in production with a read-only filesystem”

CHAPTER 54

Prompt 52: The Infrastructure as Code Reviewer

When to use: Reviewing Terraform, Pulumi, CloudFormation, or similar.

▸ The Prompt

```
Review this infrastructure code for production readiness.

{paste_iac_code}

Cloud provider: {AWS/GCP/Azure}
Environment: {staging/production}

Check for:
1. **Security:** Overly permissive IAM roles, public resources that should be priva
2. **Reliability:** Single points of failure, missing health checks, no auto-scalin
3. **Cost:** Over-provisioned resources, always-on resources that could be on-demar
4. **Operability:** Missing tags/labels, no monitoring, unclear resource naming, mi
5. **Drift risk:** Hardcoded values that should be variables, resources that might

For each issue:
- Severity (Critical / Important / Nice to have)
- The fix (show the corrected code)
- Cost impact if applicable

Also: Is there anything this code SHOULD be managing that it isn't? (DNS, SSL certs
```

▸ Why It Works

IaC reviews are different from code reviews because the consequences of mistakes are immediate and expensive (data exposure, unexpected bills, outages). The five-category checklist covers the full spectrum of IaC concerns.

▸ Customisation Tips

- Include your compliance framework (SOC 2, HIPAA, GDPR) for security-specific checks
- Specify your cost budget: “Monthly spend should stay under £500”
- Add your existing infrastructure context: “We already have a VPC set up in a separate stack”

CHAPTER 55

Prompt 53: The Incident Post-Mortem Writer

When to use: After an incident, you need to write a blameless post-mortem.

▸ The Prompt

```
Help me write a blameless post-mortem for this incident.

What happened: {describe_the_incident}
Timeline: {list_key_events_with_timestamps}
Impact: {who_was_affected_and_how}
Resolution: {how_it_was_fixed}
Duration: {how_long_it_lasted}

Write the post-mortem with:
1. **Summary** (3 sentences: what happened, how long, who was affected)
2. **Timeline** (formatted table of events)
3. **Root cause** (the actual root cause, not "human error")
4. **Contributing factors** (what made this possible — missing monitoring, inadequa
5. **What went well** (detection time, response, communication)
6. **What went poorly** (where the process broke down)
7. **Action items** (specific, assigned, with deadlines — not vague "improve monito

Rules:
- No blame. No names in the narrative (use roles: "the on-call engineer")
- Focus on systemic issues, not individual mistakes
- Every action item must be concrete enough to create a ticket from it
```

▸ Why It Works

The “no names, focus on systems” rules enforce blameless culture. The action item quality requirement — “concrete enough to create a ticket” — is what separates useful post-mortems from performative ones. “Improve monitoring” is not a ticket; “Add p99 latency alert for /api/payments endpoint with 2-second threshold” is.

▸ Customisation Tips

- Include your incident severity framework for proper classification
- Add your SLA/SLO context: “We have a 99.9% uptime SLA”
- Specify your audience: “This goes to the whole engineering org” vs “Just our team”

Part 9: Bonus — Power Prompts

Advanced techniques for getting the most out of Claude in development workflows.

CHAPTER 56

Prompt 54: The Chain-of-Thought Debugger

When to use: For complex logic errors where you need Claude to show its reasoning.

▸ The Prompt

```
I need you to think through this problem step by step. Show ALL your reasoning, eve

Problem: {describe_the_issue}
Code: {paste_code_here}

At each step:
- State what you're looking at
- State what you expect to find
- State what you actually find
- If they differ, explain why

Don't jump to the answer. Walk me through it like you're pair programming with me a
```

▸ Why It Works

This prompt turns Claude from an answer machine into a teaching tool. The explicit “show ALL reasoning” instruction produces a mental model you can learn from, not just a fix you can paste.

▸ Customisation Tips

- Add “Assume I have {n} years of experience” to calibrate the explanation depth
- Use this when the first-attempt prompt didn’t produce a satisfying answer
- Chain it with Prompt 1 for the most thorough debugging

CHAPTER 57

Prompt 55: The Rubber Duck Pro

When to use: You’re stuck and need a structured thinking partner.

▸ The Prompt

```
I'm stuck on this problem. I'm going to explain what I'm doing, and I need you to b

Here's my current understanding:
{explain_what_you_think_is_happening}

Here's what I've tried:
{list_what_you_ve_tried}

Here's where I'm stuck:
{describe_the_sticking_point}

Your job:
1. Identify my unstated assumptions (things I think are true but haven't verified)
2. For each assumption, suggest how to verify it (a specific command, test, or expe
3. Suggest at least one completely different approach I haven't considered
4. Ask me 3 questions that would narrow the problem down if I can answer them

Don't try to solve it yet. Help me think about it better.
```

▸ Why It Works

"Don't solve it yet" is the key instruction. Most developers don't need someone to solve the problem — they need help thinking about it differently. The assumption-challenging approach replicates what the best technical mentors do.

▸ Customisation Tips

- This prompt works particularly well for design problems, not just bugs
- Follow up with answers to the 3 questions for a much more targeted second attempt
- Use this when you've been stuck for more than 30 minutes

CHAPTER 58

Prompt 56: The Code Review Reviewer

When to use: You want to improve your own code review skills.

▸ The Prompt

```
I reviewed this code and here are my comments:

Code:
{paste_code_under_review}

My review comments:
{paste_your_review_comments}

Evaluate my review:
1. What did I catch that was important? (Confirm I'm right, or correct me if I'm wr
2. What did I miss that I should have caught? (With explanations)
3. Which of my comments are nit-picks that I should have skipped?
4. How's my communication? (Am I being constructive? Specific? Actionable?)
5. Give me a score out of 10 for review thoroughness

Help me become a better code reviewer.
```

▸ Why It Works

Code review is a skill that improves with feedback, but reviewers rarely get feedback on their reviews. This prompt creates a meta-review loop that systematically improves your review abilities.

▸ Customisation Tips

- Include your team's code review guidelines for context
- Specify your relationship to the author: "I'm reviewing a junior's code" vs "I'm reviewing a senior's code"
- Use this regularly to track your improvement over time

CHAPTER 59

Prompt 57: The Specification to Implementation

When to use: You have clear requirements and want production-quality implementation.

▸ The Prompt

```
Here's a specification. Implement it in {language} using {framework}.

Specification:
{paste_spec_or_user_story_or_acceptance_criteria}

Implementation requirements:
1. Production-quality code (not a prototype — include error handling, validation, l
2. Tests for every acceptance criterion
3. Database migrations if needed
4. API documentation if it's an endpoint
5. Configuration for any tuneable parameters (don't hardcode)

Architecture constraints:
{your_patterns — e.g., "We use repository pattern for data access", "All async code

Before writing code, confirm your understanding:
- List the acceptance criteria as you understand them
- List any ambiguities in the spec and how you plan to resolve them
- List any edge cases the spec doesn't address

Then implement.
```

▸ Why It Works

The “confirm your understanding” step catches misinterpretations before they become wasted code. Most spec-to-code prompts skip this and end up with an implementation that solves the wrong problem. The “list ambiguities” instruction surfaces decisions that need human input.

▸ Customisation Tips

- Include your project’s existing code style for consistent implementation
- Paste your model definitions so Claude uses the correct data structures
- Add “Follow our existing patterns — here’s an example of a similar feature: {paste_example}”

Prompt 58: The Git History Archaeologist

When to use: You need to understand why code evolved to its current state.

▸ The Prompt

```
I'm trying to understand why this code exists in its current form. Here's the code

Current code:
{paste_current_code}

Git log (relevant commits):
{paste_git_log_with_messages_and_dates}

Key diffs:
{paste_relevant_diffs}

Reconstruct the evolution:
1. What was the original intent? (Based on the earliest version)
2. Why was each significant change made? (Infer from commit messages and diffs)
3. Were any changes fixing bugs from previous changes? (Identify the chain)
4. Is the current version the "right" architecture, or has it drifted from the orig
5. Are there any changes that contradicted earlier changes? (A changed, then B char

Based on this history, should this code be:
a) Left as is (it's evolved to a good state)
b) Refactored to match a clearer design (the evolution introduced inconsistencies)
c) Rewritten (the accumulated changes no longer make a coherent whole)
```

▸ Why It Works

Understanding code history is crucial for making good refactoring decisions. This prompt turns git archaeology from a manual, time-consuming process into a structured analysis. The a/b/c recommendation at the end provides actionable guidance.

▸ Customisation Tips

- Include `git blame` output for the most confusing sections
- Mention if original authors are available: “I can ask the original author about commit {hash}”
- Add related issues/tickets if they’re linked in commit messages

Appendix A: Prompt Chaining Recipes

These recipes combine multiple prompts from this toolkit for complex workflows.

CHAPTER 61

Recipe 1: The Full Code Review Pipeline

1. **Start with Prompt 9 (Senior Engineer Review)** → Get the issue list
2. **For any security issues, use Prompt 10 (Security Audit)** → Deep-dive on vulnerabilities
3. **Use Prompt 11 (Performance Review)** → Check scalability
4. **Use Prompt 14 (Test Gap Finder)** → Ensure tests cover the changes
5. **Finish with Prompt 15 (New Hire Review)** → Verify readability

CHAPTER 62

Recipe 2: The Bug Investigation Flow

1. **Start with Prompt 3 (Stack Trace Decoder)** → Understand the error
2. **Use Prompt 1 (Root Cause Analyst)** → Find the cause
3. **Use Prompt 4 (Reproduction Builder)** → Create a repro case
4. **Use Prompt 32 (Unit Test Writer)** → Write a test that catches the bug
5. **Fix the bug, verify the test passes**

CHAPTER 63

Recipe 3: The System Design Pipeline

1. **Start with Prompt 18 (System Design Collaborator)** → Design the architecture
2. **Use Prompt 19 (Database Schema Designer)** → Design the data model

3. **Use Prompt 20 (API Design Workshop)** → Design the API
4. **Use Prompt 17 (ADR)** → Document the key decisions
5. **Use Prompt 31 (RFC Writer)** → Propose to the team

CHAPTER 64

Recipe 4: The Legacy Code Rescue

1. **Start with Prompt 40 (Code Smell Identifier)** → Diagnose the problems
2. **Use Prompt 44 (Dead Code Eliminator)** → Remove the noise
3. **Use Prompt 43 (Legacy Code Moderniser)** → Update the language/patterns
4. **Use Prompt 21 (Refactoring Strategy)** → Plan the restructuring
5. **Use Prompt 41 (Extract and Inject)** → Make it testable
6. **Use Prompt 32 (Unit Test Writer)** → Add tests before further changes

---

CHAPTER 65

Recipe 5: The Performance Rescue

1. **Start with Prompt 11 (Performance Review)** → Identify bottlenecks
2. **Use Prompt 48 (N+1 Detector)** → Fix query problems
3. **Use Prompt 46 (Query Optimiser)** → Tune remaining slow queries
4. **Use Prompt 47 (Caching Strategy)** → Add caching where needed
5. **Use Prompt 42 (Performance Refactorer)** → Optimise hot code paths

---

Appendix B: Customising Prompts for Your Stack

Here are specific additions for popular technology stacks:

▸ Python (Django/FastAPI/Flask)

Add to any prompt:

```
Python version: {version}
Framework: {Django 5.x / FastAPI / Flask}
ORM: {Django ORM / SQLAlchemy / Tortoise}
Type checking: {mypy strict / loose / none}
Linter: {ruff / flake8 / pylint}
```

▸ TypeScript (React/Next.js/Node)

Add to any prompt:

```
TypeScript version: {version} with {strict mode / loose}
Runtime: {Node / Deno / Bun}
Framework: {Next.js / Express / Fastify / NestJS}
State management: {Redux / Zustand / Jotai / none}
Styling: {Tailwind / CSS Modules / styled-components}
```

▸ Go

Add to any prompt:

```
Go version: {version}
Key libraries: {gin / echo / chi / standard library}
Error handling: {stdlib errors / pkg/errors / custom}
Concurrency model: {goroutines + channels / errgroup / worker pool}
```

▸ Rust

Add to any prompt:

```
Rust edition: {2021 / 2024}
Async runtime: {tokio / async-std / none}
Error handling: {anyhow / thiserror / custom}
Serialisation: {serde}
Web framework: {axum / actix-web / rocket}
```

▸ Java/Kotlin (Spring Boot)

Add to any prompt:

```
Java/Kotlin version: {version}
Spring Boot version: {version}
Build tool: {Gradle / Maven}
ORM: {JPA/Hibernate / jOOQ / exposed}
Testing: {JUnit 5 / Kotest / TestNG}
```

Appendix C: Quick Reference Card

Print this page and keep it next to your monitor.

SITUATION	PROMPT #	KEY INSTRUCTION
Error I don't understand	1	"Root cause, not symptom"
No error, just wrong	2	"Walk through line by line"
Wall of stack trace	3	"My code vs library code"
Works locally, fails in prod	5	"Ranked hypotheses"
Need code review	9	"🔴 BLOCKER / 🟡 SHOULD FIX / 🟢 NIT"
Security check	10	"Step-by-step exploit scenario"
Slow code	11	"At what scale does it break?"
Design decision	17	"Be opinionated, not 'it depends'"
New system design	18	"Simplest first, then evolve"
Database schema	19	"Start with query patterns"
Need tests	32	"Happy path / edges / errors / state"
Test quality check	36	"Mutation testing analysis"

SITUATION	PROMPT #	KEY INSTRUCTION
Code smells	40	“Concrete consequences, not principles”
Legacy modernisation	43	“Don’t change logic, just modernise”
Slow queries	46	“Include EXPLAIN output”
CI/CD setup	50	“Time budget per stage”
Post-mortem	53	“No blame, concrete action items”
Stuck on a problem	55	“Don’t solve — help me think”

About This Toolkit

This toolkit was built for developers who use Claude (or similar AI assistants) daily in professional software development. Every prompt has been tested in real-world codebases across multiple languages and frameworks.

The prompts work because they follow a simple principle: **the more specific your input, the more useful the output.** Generic prompts produce generic answers. These prompts are designed to extract specific, actionable, expert-level analysis.

Version: 1.0 **Last updated:** February 2026

Built with ☕ and extensive iteration. If a prompt saved you more than 30 minutes of debugging, it paid for itself.