Claude in VS Code

Pair Programming — From Setup to Shipping a Real Feature

8 Chapters · Setup & Installation · Code Explanation · Refactoring
Code Generation · Debugging · Writing Tests
Cross-File Changes · Real-World Workflow

Claude in VS Code: Pair Programming

Chapter 1 · Setup — Installing Claude Code and Your First Session

Claude Code is Anthropic's official CLI for Claude — and it integrates directly into VS Code as an extension. Once connected, Claude can read your open files, understand your project context, and work alongside you inside the editor rather than in a separate browser tab. This chapter covers installation, sign-in, and the key ways you interact with Claude from inside VS Code.

What You're Installing

There are two distinct things, and it's worth being clear about both:

- **Claude Code CLI** — the command-line tool that runs Claude. This is the engine. It can be used standalone in a terminal, or surfaced through the VS Code extension.
- **Claude Code VS Code extension** — the editor integration. It adds a Claude panel, inline code actions, and keyboard shortcuts. Underneath, it talks to the same CLI.

You need both. The extension alone does nothing without the CLI installed. Most installation flows handle this automatically — but it's useful to know the two layers exist.

Installation

1. **Install Node.js (if not already installed)**
   Claude Code CLI requires Node.js 18 or later. Check with `node --version`. If you need it, download from nodejs.org — the LTS version is fine.

2. **Install the Claude Code CLI**
   Open a terminal and run: `npm install -g @anthropic-ai/claude-code`
   This installs the `claude` command globally. Verify with `claude --version`.

3. **Install the VS Code extension**
   In VS Code: open the Extensions panel (`Ctrl+Shift+X`), search for **Claude Code**, and install the extension published by Anthropic. You'll see the Claude icon appear in the Activity Bar on the left.

4 **Sign in to Anthropic**

Click the Claude icon in the Activity Bar (or press `Ctrl+Shift+P` → *Claude: Sign In*). A browser window opens for OAuth sign-in with your Anthropic account. Once authenticated, return to VS Code — the panel activates.

5 **Open a project folder**

Claude Code is much more useful with a folder open (`Ctrl+K Ctrl+O`) than with a single file. When a folder is open, Claude can see all files in the project and understand the overall structure — not just the active tab.

6 **Verify the connection**

Type a message in the Claude panel: "*What files are in this project?*" — Claude should respond with a summary of the folder contents. If it can, the setup is working correctly.

> **SUBSCRIPTION VS API KEY**
>
> Claude Code can be connected via a Claude.ai subscription (Pro or above) or a direct API key from console.anthropic.com. The subscription route is simpler for individual use. The API key route gives more control over model selection and spend limits — useful if you're building or running Claude in multiple tools.

The VS Code Interface — What's Where

Activity Bar icon	The Claude logo in the left sidebar. Click it to open the Claude panel. This is your primary entry point. The icon shows a badge when Claude is processing a request.
Chat panel	A full conversation interface — type prompts, see responses with syntax-highlighted code blocks, click to insert code directly into the editor. The panel is persistent within a session but resets when VS Code restarts, unless you're using a Project.
Inline code lens	Appears above functions and classes — small links like "Claude: Explain" or "Claude: Generate tests." Click to send that code to Claude without copying and pasting. Only appears when the extension is active. Can be toggled in extension settings.

Quick fix integration	When VS Code flags an error with a lightbulb, "Ask Claude" appears as a quick-fix option. Sends the error + surrounding code to Claude automatically. Works with TypeScript, Python, JavaScript, and any language with a language server.
Command palette	All Claude commands are accessible via `Ctrl+Shift+P` → type "Claude". Includes sign in/out, open panel, clear chat, and any custom commands. Useful for actions you do infrequently — no need to remember a keyboard shortcut.
Terminal integration	The VS Code integrated terminal can run `claude` CLI commands directly. Useful for running slash commands or scripted tasks that go beyond the chat panel. The terminal and the panel share the same Claude installation but run independent sessions.

Key Keyboard Shortcuts

`Ctrl + Shift + C`

Open Claude panel

Toggle the Claude chat sidebar. Works from anywhere in the editor.

`Ctrl + Shift + P → Claude`

Command palette

Access all Claude commands — sign in, clear chat, explain selection, generate tests.

`Select text → right-click`

Context menu actions

"Ask Claude", "Explain with Claude", "Fix with Claude" appear on right-click when text is selected.

`Ctrl + I`

Inline edit (Copilot-style)

Opens a prompt bar inline in the editor. Type an instruction; Claude rewrites the selected code in place.

`Ctrl + Enter`

Send message

Submits the current message in the chat panel. Enter alone adds a newline.

`Esc`

Cancel / stop

Interrupts Claude mid-response. Useful if Claude goes off in the wrong direction.

> **CUSTOMISE SHORTCUTS**
>
> All Claude Code shortcuts can be remapped. Open **Keyboard Shortcuts** (`Ctrl+K Ctrl+S`), search for "Claude", and reassign any binding that conflicts with your existing setup. On macOS, `Ctrl` is typically `Cmd` .

Two Ways to Work with Claude in VS Code

Chat panel

- Full conversation — multi-turn, with history
- Good for questions, explanations, and planning
- Claude can reference multiple files at once
- Responses include code blocks you can insert
- Best for: "explain how this module works" or "help me design this feature"

Inline edit (Ctrl+I)

- Single instruction, applied directly to selected code
- No conversation history — one-shot per invocation
- Result appears as a diff you can accept or reject
- Best for: "rename this variable", "add error handling", "convert to async"
- Fastest path for small, precise changes

In practice you'll use both: the chat panel for anything that needs thought or context, and the inline editor for quick targeted changes. The two modes complement each other — use chat to plan and understand, use inline to execute.

Your First Real Interaction

With the extension installed and a project folder open, try this sequence to get a feel for the tool:

1. Open a source file you know well.
2. In the Claude panel, type: "*Give me a one-paragraph summary of what this file does.*" — Claude reads the open file and responds.
3. Select a function in the editor, right-click, and choose **Explain with Claude**. Read the explanation.
4. Select the same function, press `Ctrl+I`, and type: "*Add a docstring.*" Accept the diff.
5. In the chat panel, ask: "*What files in this project relate to authentication?*" — Claude searches across the project, not just the open file.

These five steps cover the core interaction patterns. Everything in the following chapters builds on them.

> **CONTEXT IS AUTOMATIC IN VS CODE**
>
> Unlike the web interface, the VS Code extension automatically includes your active file in Claude's context — you don't have to paste code. Claude can also see other open files and

> read any file in the project when needed. This is the biggest practical advantage of working inside the editor rather than in a browser tab.

Next — Chapter 2: Code Explanation

Asking Claude to explain a file, function, or unfamiliar codebase. The right questions to ask, how to use Claude to onboard onto new code quickly, and how to make sense of code you didn't write — without reading every line yourself.

Claude in VS Code: Pair Programming

Chapter 2 · Code Explanation — Understanding Code You Didn't Write

One of the most immediately useful things Claude can do inside VS Code is explain code — a function you inherited, a library you've never used, a codebase you've just joined. This chapter covers how to ask for explanations at the right level of detail, how to use Claude to onboard onto an unfamiliar project quickly, and which questions actually produce useful answers versus which ones waste your time.

Why Explanation in the Editor Is Different

Asking Claude to explain code in VS Code is fundamentally different from pasting code into a browser chat. In the editor, Claude already has the file open — and can reach out to read other files in the project when the explanation requires it. You don't have to copy anything; you don't have to provide imports or class definitions separately; Claude can follow a function call into the file that defines it.

This changes the questions you can ask. "What does this function do?" is fine. But so is "What calls this function, and does anything depend on the value it returns?" — a question that requires reading multiple files, which Claude can do automatically.

Levels of Explanation

The right explanation depends on what you already know and what you're trying to decide. Match the prompt to the level:

OVERVIEW	**What does this file/module/class do?** A paragraph-level summary. Use when you're orienting — skimming a codebase to understand its structure before reading any code in detail. Ask: `"Give me a one-paragraph summary of what this file does and why it exists."`
FUNCTION	**What does this function do, and how?** Detailed explanation of logic, parameters, return values, and side effects. Use when you need to

understand a specific piece of behaviour before modifying it. Ask: `"Explain what this function does step by step, including what each parameter controls."`

LINE

What does this specific line / pattern do?
For unfamiliar syntax, idioms, or patterns you don't recognise. Select the exact text and ask. Ask: `"What does this line do and why is it written this way?"`

Prompt Patterns That Work Well

FILE OVERVIEW

```
Give me a one-paragraph summary of what this file does and where it fits in the project.
```

Best sent from the chat panel with the file open — Claude reads it automatically.

FUNCTION WALKTHROUGH

```
Walk me through this function step by step. What does each section do, and what does it return?
```

Select the function first, then right-click → Explain with Claude, or paste into chat.

DEPENDENCY TRACE

```
What calls this function? Does anything in the project depend on the value it returns?
```

Claude reads across files to answer this. More accurate than a manual grep.

DATA FLOW

```
Trace what happens to the `user` object from when it enters this function to when it leaves. What gets modified?
```

Especially useful for understanding side effects and mutation.

UNFAMILIAR PATTERN

```
I don't recognise this pattern. What is it, why would someone use it, and are there any gotchas?
```

Select the pattern before asking. Works great for decorators, metaclasses, generator expressions, async patterns.

INTENT VS IMPLEMENTATION

```
What was this code trying to achieve? Is the implementation a straightforward way to achieve it, or is there something unusual going on?
```

Surfaces hidden complexity and workarounds that aren't obvious from reading the code.

Before and After: Vague vs Specific

The specificity of your question directly determines the usefulness of the answer. These examples show the same request — one vague, one specific:

LESS USEFUL

```
Explain this code.
```

MORE USEFUL

```
Explain what this function does when the `retries` parameter is 0. Does it still attempt the request, or does it return immediately?
```

LESS USEFUL

```
What does this file do?
```

MORE USEFUL

```
What does this file do, and what would break if it were removed? I'm trying to decide if it's safe to refactor.
```

LESS USEFUL

```
I don't understand this.
```

MORE USEFUL

```
I don't understand why this uses a closure here instead of passing the value as a parameter. What does the closure buy us?
```

The pattern: replace "explain this" with a specific question about behaviour, edge cases, or intent. You'll get a targeted answer instead of a generic walkthrough.

Onboarding onto an Unfamiliar Codebase

When you're new to a project — joining a team, picking up someone else's code, or returning to your own code after six months — Claude can dramatically compress the time it takes to get oriented. Here's a sequence that works well:

1. **Start at the entry point.** Open the main file (e.g. `main.py`, `index.ts`, `App.jsx`) and ask: *"What is the entry point of this application and what does it set up?"* Claude explains the bootstrapping sequence.

2. **Map the major components.** Ask: "*What are the main modules or directories in this project and what does each one do?*" Claude reads the file tree and summarises the structure.

3. **Trace a request end to end.** Ask: "*Walk me through what happens when a user logs in — from the HTTP request to the response.*" Claude follows the code path across files.

4. **Find the data layer.** Ask: "*Where is the database accessed? What ORM or query pattern is used?*" Gets you to the data layer quickly without reading every file.

5. **Identify non-obvious conventions.** Ask: "*Is there anything unusual or non-standard in how this codebase is structured that I should know about?*" Surfaces patterns that would confuse you if you discovered them mid-task.

6. **Locate your task area.** Ask: "*I need to add [feature]. Which files would I most likely need to touch?*" Claude uses its codebase understanding to point you to the right area.

> **LET CLAUDE READ THE WHOLE PROJECT FIRST**
>
> Before starting an onboarding sequence, ask Claude: "*Take a look at the project structure and let me know when you have a feel for what's here.*" This gives Claude a chance to read key files before you start asking specific questions — the answers will be more accurate.

Asking About Third-Party Code

Claude is also useful for explaining library and framework code — not just code you wrote. If you're using a library you don't fully understand:

- **Paste a usage example** from the library and ask what each argument does
- Ask: "*What does this middleware actually do under the hood?*" — Claude draws on its training knowledge of common libraries
- Ask: "*Are there any gotchas with using this pattern in production?*" — surfaces known pitfalls
- Ask: "*What's the difference between [method A] and [method B] in this library?*" — faster than reading docs

> **VERIFY LIBRARY KNOWLEDGE AGAINST DOCS**

> Claude's knowledge of libraries has a training cutoff. For libraries that release frequently (React, Next.js, FastAPI, etc.), Claude may describe an older API. Always cross-check against the current documentation when an API detail matters — especially for breaking changes.

When Explanation Falls Short

Situation	What happens	What to do instead
Highly domain-specific logic	Claude explains the code structure correctly but misses business intent (e.g. tax calculation rules, medical protocols)	Provide context: "*This is a VAT calculation for EU digital goods — explain with that in mind.*"
Generated or minified code	Claude can explain it but the explanation is rarely useful — the code isn't meant to be read	Ask Claude to rewrite it as readable code first, then explain the rewrite
Macro-level architecture	Claude can describe files and directories but can't draw architectural diagrams without help	Ask Claude to produce a plain-text description of the architecture, then turn that into a diagram yourself
Runtime behaviour	Claude reads static code — it can't tell you what actually happened at runtime (memory usage, actual values, timing)	Use the debugger for runtime questions; use Claude for static analysis questions

Next — Chapter 3: Refactoring

Asking Claude to improve, rename, and restructure existing code. How to scope a refactor so Claude doesn't change more than you intended, how to handle multi-file refactors, and how to use the diff view to stay in control of every change.

Claude in VS Code: Pair Programming

Chapter 3 · Refactoring — Improving Code with Claude

Refactoring is where Claude in VS Code earns its keep. It can rename, restructure, simplify, and modernise code — but only if you give it the right scope. Too broad and it changes more than you intended. Too narrow and it misses dependencies. This chapter covers how to scope refactors correctly, how to use the diff view to stay in control, and how to handle changes that span multiple files.

The Core Risk: Scope Creep

The most common problem with AI-assisted refactoring is that Claude improves things you didn't ask it to improve. You say "rename this variable" and get back a rewritten function. You say "simplify this method" and Claude also changes the calling code.

This isn't necessarily wrong — Claude may be right that the related code should change. But it means you're reviewing more than you planned, and changes you didn't request can introduce bugs you won't notice until later.

The solution is explicit scope constraints in every refactoring prompt.

Scoping a Refactor

INLINE — SINGLE EXPRESSION

Rename / simplify

```
Rename `usr` to
`current_user`
throughout this
function only. Don't
change anything else.
```

Use `Ctrl+I` with selection. Fastest for single-expression changes.

FUNCTION SCOPE

Restructure logic

```
Refactor this function
to use early returns
instead of nested ifs.
Keep the same inputs,
outputs, and
behaviour.
```

Select the function, send to chat. Ask Claude to confirm behaviour is unchanged.

FILE SCOPE

Extract / reorganise

```
Extract the database
logic in this file
into a separate class.
Keep the public
interface identical —
don't change how
callers use it.
```

Use chat panel. Ask Claude to list every change it's making before

doing it.

MULTI-FILE

Rename across project

```
Rename the `UserController` class to `AccountController` everywhere in the project. Show me every file you'll touch before making changes.
```

Always preview first. Review each file diff before accepting.

Constraint Phrases That Prevent Over-reach

These phrases, added to any refactoring prompt, keep Claude within the scope you intended:

Phrase	What it prevents
`Don't change anything outside this function`	Claude modifying callers or related functions it thinks "should" also change
`Keep the public interface identical`	Parameter names, return types, or method signatures being altered silently
`Preserve existing behaviour exactly`	"Improvements" that change edge-case handling or error behaviour
`Show me every file you'll touch before making changes`	Surprise edits to files you didn't know were affected
`Only rename — don't restructure the logic`	Claude taking a rename request as an invitation to also clean up the surrounding code
`Make the minimum change to achieve this`	Over-engineering or adding abstractions you didn't ask for

Using the Diff View

When Claude makes changes via the inline editor (`Ctrl+I`), VS Code shows the result as a diff — additions in green, removals in red — before anything is applied. This is your main safety mechanism for refactoring.

auth.py — inline refactor diff ✓ Accept ✕ Reject

```
def authenticate(usr, pwd):
- if usr == None or pwd == None:
+ if usr is None or pwd is None:
return False
- result = check_credentials(usr, pwd)
- if result == True:
+ if check_credentials(usr, pwd):
return True
return False
```

Read the diff carefully before accepting. Ask yourself:

- Did Claude change anything I didn't ask it to?
- Is every removed line intentionally removed, or did something slip?
- Are the added lines functionally equivalent to what they replace?
- Does the diff touch code outside my selected scope?

> **ACCEPT PARTIAL DIFFS**
>
> If Claude's diff is 90% right but one change is wrong, don't reject the whole thing — accept it, then use `Ctrl+Z` to undo just the specific line, or make a manual edit. Rejecting a large diff because of one line wastes all the good work.

Common Refactoring Requests

VAGUE

```
Clean this up.
```

SPECIFIC

```
Replace the nested if-else chain with
early returns. Don't change the logic,
```

just the structure.

VAGUE

```
Make this more Pythonic.
```

SPECIFIC

```
Replace the manual list-building loop
with a list comprehension. Keep the
filter logic identical.
```

VAGUE

```
Refactor this class.
```

SPECIFIC

```
Extract the three private `_parse_*`
methods into a separate `Parser` class
in the same file. The public API of
`DataLoader` must not change.
```

Multi-File Refactors

Some refactors touch more than one file — renaming a class, changing a function signature, moving a module. These need more care because the diff view shows one file at a time and it's easy to miss an affected file.

📋 **Plan before executing.** Ask Claude: "*If I rename `UserService` to `AccountService`, which files in this project would need to change?*" Get the list first — don't start editing until you know the full scope.

🔍 **Verify the list.** Use VS Code's built-in **Find in Files** (`Ctrl+Shift+F`) to search for the old name. Compare Claude's list against the search results — if they differ, Claude missed something or the search found false positives.

✏️ **Do one file at a time.** Ask Claude to make the change in a single file, review the diff, accept it, then move to the next. Don't let Claude batch-edit multiple files in one response — you lose visibility.

🧪 **Run tests after each file.** If the project has tests, run them after each file is updated. A failing test immediately after a specific file tells you exactly where the problem is — much easier to debug than a failing test suite at the end.

Final check. After all files are done, search for the old name one more time with `Ctrl+Shift+F`. Any remaining hits that shouldn't be there are missed updates.

> **VS CODE RENAME SYMBOL VS CLAUDE**
>
> For pure renames — a variable, function, or class name — VS Code's built-in **Rename Symbol** (`F2`) is often faster and more reliable than asking Claude. It uses the language server to find every reference automatically. Use Claude for renaming when the change also involves restructuring, or when the language server doesn't cover the file type.

Refactoring Patterns Claude Does Well

- **Early returns** — flattening deeply nested conditionals
- **Extract method/class** — pulling a coherent block of logic into its own function or class
- **Modernise syntax** — updating old patterns to current language idioms (e.g. f-strings, optional chaining, pattern matching)
- **Remove duplication** — identifying repeated logic and centralising it
- **Clarify naming** — renaming cryptic variables and functions to descriptive names
- **Convert callback to async/await** — restructuring callback-based code to async equivalents
- **Add type annotations** — inferring and adding types to untyped Python or TypeScript code

Refactoring patterns Claude does poorly

- **Performance optimisation** — Claude can suggest patterns but rarely knows your runtime bottleneck. Profile first, then ask Claude to apply a specific optimisation.
- **Architecture-level restructuring** — moving entire modules, changing dependency direction, introducing new layers. These benefit more from a planning conversation than a direct refactoring request.
- **Database schema changes** — Claude can write the migration but doesn't know your data volume, index strategy, or production constraints. Treat its output as a starting point.

Next — Chapter 4: Generating Code

Asking Claude to write new features, boilerplate, and schema from a description. How to give Claude

enough context to generate code that fits your existing patterns, what to check in generated code before committing, and how to iterate when the first output isn't quite right.

Claude in VS Code: Pair Programming

Chapter 4 · Generating Code — Features, Boilerplate, and Schema

Generating new code is where Claude feels closest to having a second developer next to you. Describe what you need, and Claude writes it — functions, classes, API routes, database migrations, configuration files. But generated code that doesn't fit your existing patterns creates more work than it saves. This chapter is about giving Claude enough context to generate code that slots in cleanly, and what to check before you commit it.

The Generation Problem: Context Mismatch

Claude's biggest weakness when generating code isn't correctness — it's fit. It can write a perfectly valid function that uses the wrong naming convention, imports a library you don't use, handles errors in a style inconsistent with the rest of the codebase, or reinvents a utility you already have.

This happens because Claude is generating from a description rather than from deep knowledge of your specific codebase. The fix is always the same: give Claude more context about your patterns before asking it to generate.

What Context to Provide

MUST HAVE	**An example of a similar existing thing.** "Here's a function that does something similar — follow the same pattern." This single piece of context eliminates most style and convention mismatches. Open the example file before generating.
MUST HAVE	**The exact signature or interface.** If you know what parameters the function should accept and what it should return, say so explicitly. Don't leave it for Claude to invent.
GOOD TO HAVE	**The tech stack and version.** "This is a FastAPI 0.111 project using SQLAlchemy 2.0 async." Claude knows many frameworks but defaults change between versions — specificity avoids deprecated patterns.

GOOD TO HAVE	**Error handling style.** "We raise custom exceptions; we never return None on failure" or "errors are returned as result tuples." Claude will otherwise pick whatever convention seems most common for the language.
OPTIONAL	**What not to use.** "Don't add logging — we have a decorator for that." "Don't use requests — we use httpx." Negative constraints prevent Claude from adding things you'll just have to remove.

Anatomy of a Good Generation Prompt

Example: generating a new API endpoint

```
GOAL Write a POST /users/password-reset endpoint.

CONTEXT This is a FastAPI project. See routes/users.py for the pattern — all routes use the `get_db` dependency, return Pydantic response models, and raise HTTPException on error. The User model is in models/user.py.

PATTERN Follow the same structure as the existing POST /users/verify-email route.

CONSTRAINTS Accept `email: str` in the request body. Return a 200 with `{"message": "reset email sent"}` whether or not the email exists (don't leak user enumeration). Don't add rate limiting — that's handled by middleware.
```

Four components: what to build, where it lives and what patterns to follow, a concrete example to mirror, and constraints that prevent common additions you don't want. Every generation prompt benefits from all four.

Common Generation Use Cases

New function from description

```
Write a function `calculate_late_fee(due_date, paid_date, daily_rate)` that returns the total fee as a Decimal. Return 0 if paid on time. Follow the style in billing/utils.py.
```

Boilerplate / scaffolding

```
Create a new service class for sending SMS notifications. Follow the same pattern as services/email_service.py — same constructor, same error handling, same logging approach.
```

Always open a related file first so Claude can see your conventions.

Point Claude at an existing service to mirror. Boilerplate is where "follow this pattern" saves the most time.

Database schema / migration

```
Write an Alembic migration to add a `password_reset_tokens` table. Columns: id (UUID pk), user_id (FK to users.id), token (string, unique), expires_at (datetime), used_at (nullable datetime).
```

Specify every column and constraint. Claude will fill in the Alembic boilerplate — you specify the schema.

Configuration / infra files

```
Write a docker-compose.yml for this project. Services: FastAPI app (port 8000), PostgreSQL 16, Redis. Include a .env reference for DB credentials. Don't expose Redis externally.
```

Config generation is one of Claude's strongest areas — it knows the formats well and rarely hallucinates field names.

Data transformation / mapper

```
Write a function that maps a raw Stripe webhook payload dict to our internal `PaymentEvent` dataclass. Only extract the fields we need: id, type, amount, currency, customer_id, created_at.
```

Give Claude both the input shape and the target type. It handles the mapping; you verify the field names are right.

CLI / script

```
Write a Python script that reads a CSV of user emails from stdin and calls our `/admin/users/bulk-deactivate` API endpoint in batches of 50. Use httpx. Print progress to stderr.
```

Scripts are low-risk to generate — they're standalone, easy to test, and the blast radius of a bug is limited.

Reviewing Generated Code Before Committing

Generated code should always be read before it's committed — not just run. Claude is confident and fluent, which means bugs read as naturally as correct code. A review pass catches the categories of problem that tests often miss:

⚠ **Security:** does it handle untrusted input? SQL injection, XSS, path traversal, insecure defaults — Claude sometimes omits these unless you asked for them explicitly.

⚠ **Error handling:** what happens when the external call fails, the DB is unavailable, or the input is malformed? Verify Claude's error paths match your project's approach.

- ⚠ **Imports:** does the generated code import libraries you actually have? Claude occasionally imports a package that would solve the problem but isn't in your project.
- ⚠ **Naming:** do the identifiers match your conventions? Variable names, function names, and class names should be consistent with the surrounding code.
- ⚠ **Edge cases:** does it handle empty input, None values, zero, negative numbers, or whatever the relevant boundary conditions are for this function?
- ⚠ **Hallucinated APIs:** verify any method or attribute calls against documentation or the actual library source. Claude occasionally invents plausible-sounding method names that don't exist.
- ⚠ **Hardcoded values:** config values, magic numbers, and hardcoded strings that should be constants or environment variables.

> **RUN IT, DON'T JUST READ IT**
>
> Reading the code is necessary but not sufficient. Run the generated code against real inputs before merging. Type checking and linting catch a class of bugs that reading misses; tests catch behavioural bugs that the linter misses. All three layers together give you reasonable confidence.

Iterating When the First Output Isn't Right

The first generation is rarely perfect. The efficient approach is targeted follow-up, not regenerating from scratch:

1. **Identify the specific problem.** Don't say "this isn't right." Say "the error handling on line 12 returns None — it should raise a ValueError with the message 'invalid date range'."
2. **Lock in what's correct.** "The function signature and the happy-path logic are correct — only fix the error handling." Prevents Claude from helpfully rewriting parts that were fine.
3. **Add the missing constraint.** If Claude got something wrong, it's usually because a constraint was absent from the original prompt. Add it explicitly: "We never return None from this layer — always raise."

4. **Ask for the minimal change.** "Make the minimum edit to fix this — don't restructure anything else." Keeps the diff small and the review fast.

5. **If it keeps going wrong, provide an example.** "Here's how we handle this in the payment module — do the same." A concrete example beats any amount of description for style and convention alignment.

> **GENERATE, REVIEW, TEST IN SMALL UNITS**
>
> Generate one function at a time rather than asking Claude to write an entire feature in one go. Small generations are faster to review, easier to test, and when something is wrong you know exactly which piece caused it. Long generations that are 80% correct leave you doing a line-by-line review of everything to find the 20%.

Next — Chapter 5: Debugging with Claude

Pasting errors and stack traces, reading runtime failures, and finding root causes. How to give Claude the right diagnostic information, the difference between asking Claude to fix a bug versus explain what caused it, and when to reach for the debugger instead.

Claude in VS Code: Pair Programming

Chapter 5 · Debugging with Claude

Debugging is where having a knowledgeable second pair of eyes pays off immediately. Claude can read a stack trace, explain what went wrong, suggest root causes, and propose a fix — often faster than you could work through it alone. But there's a discipline to debugging with AI: the quality of Claude's diagnosis is directly proportional to the quality of the information you give it. This chapter covers what to send, how to ask, and when to reach for the debugger instead.

Reading a Stack Trace with Claude

A stack trace tells a story — but reading it bottom-up (where the error originates) is different from reading it top-down (how you got there). Claude is good at translating the trace into plain language and pointing at the line that matters. The key is sending the *whole* trace, not just the error message.

Python traceback — annotated

```
Traceback (most recent call last):
  File "app/api/routes/orders.py", line 47, in create_order   ENTRY POINT
    result = order_service.process(payload)
  File "app/services/order_service.py", line 112, in process
    user = db.query(User).filter_by(id=payload.user_id).one()
  File "sqlalchemy/orm/query.py", line 2691, in one   READ FROM HERE DOWN
    raise NoResultFound()
sqlalchemy.orm.exc.NoResultFound: No row was found for one()   ACTUAL ERROR
```

When sending a trace to Claude:

- **Include the full trace** — not just the final error line. The frames above it show how the code got there.
- **Include the relevant source lines** — the trace shows line numbers, but not the surrounding context. Open the file and paste the function Claude is being asked to diagnose.

- **Describe what you were doing** — "this happens when I try to create an order for a user who signed up yesterday." Conditions that trigger the bug are often the key to understanding it.

What Information to Send

Always include

→ The complete error message and stack trace

→ The function or block where the error occurs

→ What you were doing / what input triggered it

→ What you expected to happen

Include if relevant

→ Recent changes to the code ("I refactored this yesterday")

→ The environment it fails in vs works in

→ Any related function that calls or is called by the failing code

→ Data shape or sample values that trigger the error

Helpful extras

→ What you've already tried and ruled out

→ Whether it's intermittent or consistent

→ Library / framework versions if version-specific

→ Logs from immediately before the error

Don't omit

→ The inner-most exception (sometimes wrapped)

→ The file and line number from your code (not the library)

→ Type information if it's a type error

→ The actual vs expected value if it's an assertion

Explain vs Fix — Know Which to Ask For

There's an important difference between asking Claude to explain a bug and asking it to fix a bug. Both are useful, but they serve different purposes and the one you choose affects how much you learn.

Ask Claude to explain

You understand the root cause before any code changes. Better for: bugs in code you're unfamiliar

Ask Claude to fix

Fastest path to a working change. Better for: well-understood errors (typo, missing import, off-by-

with, subtle logic errors, or anything where you want to stay in control of the fix.

```
What's causing this error? Explain the root cause — don't write a fix yet, I want to understand what went wrong first.
```

one), boilerplate fixes, or when you're under time pressure.

```
Fix this bug. The error is a NoResultFound when user_id doesn't exist in the DB — handle that case by returning a 404 HTTPException.
```

EXPLAIN FIRST FOR UNFAMILIAR CODE

If you're debugging code you didn't write and don't fully understand, always ask for the explanation first. Accepting a fix you don't understand means you'll hit a variant of the same bug again — and Claude won't be there to fix it in production.

Bug Categories and How to Ask

Exception / crash

```
Here's the full traceback and the function at line 112. What's causing the NoResultFound and how should it be handled?
```

Paste the full trace + the relevant function. Don't summarise the trace — send it verbatim.

Wrong output

```
This function returns 42 when I call it with input X, but I expect 38. Here's the function — can you trace through what it computes for that input?
```

Give Claude both the actual and expected value, plus a concrete input to trace through.

Intermittent failure

```
This fails maybe 1 in 10 times with a race condition. Here's the code path. What could cause non-deterministic behaviour here?
```

Claude will point at shared state, async timing, and unguarded reads. Follow up with "how would I reproduce this reliably?"

Performance regression

```
This query ran in 40ms yesterday. After I added the LEFT JOIN it takes 4 seconds. Here's the query and the EXPLAIN output. What's causing the slowdown?
```

Always include EXPLAIN / EXPLAIN ANALYZE output for query performance issues. Without it Claude is guessing.

Silent failure

```
This function returns successfully but the record isn't being saved. No exception is raised. Here's the
```

Type / import error

```
I'm getting `AttributeError: 'NoneType' object has no attribute 'email'`. Here's the line and the function that
```

```
function — what could cause a silent
no-op?
```

Silent failures are often un-awaited coroutines, swallowed exceptions, or a session that isn't being committed.

```
creates the object. When would `user`
be None here?
```

Ask Claude when the value could be None rather than how to guard against it — understanding the cause prevents the same bug downstream.

Claude vs the Debugger — When to Use Which

Reach for Claude when…	Reach for the debugger when…
You have a stack trace and want it explained	You need to inspect actual runtime values at a specific line
The bug is in logic you can read statically	The bug only manifests under specific timing or load conditions
You want to understand why the code fails, not just where	You need to step through a complex execution path frame by frame
The error message is cryptic and needs translation	You're dealing with a memory issue, thread race, or async timing bug
You want to brainstorm possible root causes	You need to watch a variable change across many iterations
The bug is in a third-party library and you want to understand its behaviour	The execution path is long and branching and you can't mentally model it

The two tools complement each other. A common workflow: use Claude to identify the likely cause and the suspect lines, then set a breakpoint at exactly those lines in the debugger to confirm. This is faster than stepping through from the start.

When Claude's Diagnosis Is Wrong

Claude can misdiagnose a bug — especially when the cause is environmental (a specific data state, a platform difference, a version incompatibility) rather than visible in the static code. If Claude's suggested fix doesn't resolve the problem:

- **Tell Claude what happened** — "I tried that fix and still get the same error. Here's the new trace." This is the most important thing you can do — don't start over, iterate.
- **Add more context** — if Claude missed something, it's usually because you didn't include it. What's in the database? What's the actual value of the variable? What changed recently?
- **Ask for alternative hypotheses** — "If that's not the cause, what else could explain this?" Claude will generate other candidates to investigate.
- **Try rubber duck debugging** — ask Claude to ask *you* questions: "Ask me questions to help diagnose this." Often the act of answering surfaces what you'd forgotten to include.

> **THE QUICK-FIX TRAP**
>
> When Claude confidently proposes a fix, it's tempting to apply it and move on if the error disappears — without understanding what caused it. Resist this. A fix you don't understand is a bug that will return in a different form. Take thirty seconds to read Claude's explanation even when you're under pressure.

Next — Chapter 6: Writing Tests

Asking Claude to write unit tests, integration tests, and edge cases. How to brief Claude on your testing framework and style, how to get tests that actually cover the right behaviour, and how to use Claude to find the edge cases you haven't thought of yet.

Claude in VS Code: Pair Programming

Chapter 6 · Writing Tests with Claude

Writing tests is one of the most mechanical parts of software development — and one of the best fits for Claude. Given a function to test, Claude can generate a suite of cases covering the happy path, error paths, and boundary conditions faster than you can type them. The skill is briefing Claude correctly on your testing framework and style, and knowing how to prompt for the edge cases you haven't thought of yet.

Brief Claude on Your Testing Setup First

Before asking Claude to write any tests, give it a one-time brief on your testing environment. Without this, Claude will make reasonable guesses — and those guesses may not match your project. Do this once at the start of a testing session:

```
Testing brief — send this before the first test request

FRAMEWORK We use pytest with pytest-asyncio for async tests.

STYLE Tests live in tests/ mirroring the app structure. Test files are named
test_[module].py. Test functions are named test_[what]_[condition] (e.g.
test_create_order_missing_user_returns_404).

COVERAGE We aim to cover: happy path, each distinct error case, and at least one
boundary value per numeric input.

EXCEPTIONS We don't mock the database — tests run against a real test DB. We do mock
external HTTP calls using respx.
```

Once Claude has this brief, all test requests in the session will use the right framework, naming conventions, and mocking approach without you repeating it.

Types of Tests Claude Generates Well

UNIT

Pure function tests

```
Write tests for the `calculate_late_fee` function in billing/utils.py. Cover: no late fee (paid on time), one day late, exactly 30 days late, and a negative due date input.
```

Claude's strongest area. Pure functions with defined inputs and outputs are trivial to test and Claude generates comprehensive cases quickly.

UNIT

Class / method tests

```
Write tests for the `OrderProcessor.process()` method. Mock the payment gateway. Test: successful order, payment declined, and inventory unavailable.
```

Tell Claude exactly what to mock. If you leave it unspecified, Claude will choose — and may choose differently from your project's conventions.

INTEGRATION

API endpoint tests

```
Write integration tests for POST /orders using the test client. Test: valid order creation (201), missing required fields (422), unauthenticated request (401), and duplicate order (409).
```

Specify the expected status code for every case. Claude will generate the fixture setup if you open the existing conftest.py first.

PARAMETRISED

Data-driven tests

```
Write a parametrised test for `parse_date_range(start, end)` covering: valid range, reversed dates, same start and end, start in the past, and end more than 1 year from now.
```

Claude writes clean pytest.mark.parametrize tables. Give it the cases; it handles the structure.

Anatomy of a Well-Generated Test

Example — pytest test for a billing function

```python
def test_calculate_late_fee_one_day_late():  ARRANGE
    # Arrange
    due_date = date(2026, 1, 10)
    paid_date = date(2026, 1, 11)
    daily_rate = Decimal("2.50")

    # Act  ACT
    result = calculate_late_fee(due_date, paid_date, daily_rate)
```

```
    # Assert  ASSERT
    assert result == Decimal("2.50")
```

A well-structured test has three clear sections: **Arrange** (set up inputs), **Act** (call the thing being tested), **Assert** (verify the result). If you ask Claude to follow this structure explicitly, the output is easier to read and easier to modify. Add `Follow the Arrange-Act-Assert pattern` to your brief.

Using Claude to Find Edge Cases You Haven't Thought Of

This is one of the highest-value ways to use Claude for testing. Rather than asking Claude to write tests for cases you already know about, ask it to find the ones you missed:

Edge case discovery prompts

```
// Discover first, write second
What edge cases and error conditions should I test for `calculate_late_fee`? List
them before writing any code.

// Security-focused
What inputs to this function could cause unexpected behaviour from a security
perspective? I want to make sure we're testing those.

// Boundary focus
What are the boundary values for this function? I want parametrised tests that cover
exactly at, just below, and just above each boundary.

// Combination cases
What combinations of inputs are most likely to cause problems? I'm thinking about
cases where multiple conditions interact.
```

Ask Claude to list the cases first — before writing any code. Review the list, add or remove cases, then ask Claude to write tests for the approved list. This gives you control over coverage without having to think of every case yourself.

Edge Case Categories to Always Cover

Null / empty

None, empty string, empty list, empty dict, zero-length file

Boundary values

Min, max, exactly at limit, one below limit, one above limit

Type coercion

Integer where float expected, string "0", "false", "null", numeric string

Special characters

Unicode, emoji, newlines in strings, SQL metacharacters, HTML tags

Negative / zero

Negative numbers, zero, negative zero, very large numbers, overflow

Time and dates

Leap day, DST transition, midnight, timezone mismatch, far-future dates

Concurrency

Duplicate requests, race on shared state, timeout during operation

External failure

DB unavailable, HTTP 500, network timeout, malformed response body

Common Pitfalls in Claude-Generated Tests

Pitfall	What it looks like	How to avoid it
Tests that always pass	An assertion like `assert result is not None` — technically passes but verifies almost nothing	Ask Claude: "Assert the exact expected value, not just that a result exists."
Over-mocking	Claude mocks the function being tested, or mocks so much that the test doesn't exercise real code	Specify what to mock and what not to. "Mock the HTTP call only — use the real service layer."
Wrong framework syntax	Claude uses `unittest.mock` when you use `pytest-mock`, or uses `self.assert` in a non-class test	Include framework and library specifics in your brief. Open an existing test file before asking.
Tests coupled to implementation	Tests that break when you refactor internal details, even if the behaviour didn't change	Ask Claude: "Test the observable behaviour, not the internal implementation."

Pitfall	What it looks like	How to avoid it
Missing cleanup	Tests that leave state (DB rows, files, env vars) that affect other tests	Ask Claude to add fixtures or teardown. "Make sure each test is independent and cleans up after itself."

> **RUN THE TESTS BEFORE COMMITTING THEM**
>
> Claude-generated tests can contain subtle errors — wrong import paths, a fixture that doesn't exist yet, an assertion that passes vacuously. Always run the full test file after generation and fix any failures before committing. A test suite with broken tests is worse than no test suite — it trains the team to ignore failures.

Tests for Code That Already Has Bugs

An underused pattern: ask Claude to write a test that *reproduces a bug you're about to fix*. Write the test first (it should fail), then fix the code (the test passes), then commit both. This is lightweight TDD with Claude doing the test-writing:

> **BUG-FIRST TEST PROMPT**
>
> "Write a test that demonstrates this bug: when `calculate_late_fee` is called with a `paid_date` before the `due_date`, it returns a negative fee instead of zero. The test should fail against the current code and pass once the bug is fixed."

Next — Chapter 7: Working Across Files

Project-wide changes, asking Claude about architecture, and how Claude's cross-file awareness works. How to make changes that span modules, how to ask architectural questions that require reading multiple files, and the limits of Claude's project-wide understanding.

Claude in VS Code: Pair Programming

Chapter 7 · Working Across Files — Project-Wide Changes and Architecture

Previous chapters focused on single files or functions. Real work rarely stays that contained — a feature touches multiple modules, a rename ripples across the project, an architecture question requires reading a dozen files before you can answer it. This chapter covers how Claude's cross-file awareness works, how to make project-wide changes safely, and how to ask architectural questions that produce useful answers.

How Claude's Cross-File Awareness Actually Works

Claude doesn't automatically read your entire project into memory. It reads files on demand — either because you have them open, because you reference them explicitly, or because Claude decides it needs to read them to answer your question. Understanding this model prevents both over-reliance and under-utilisation.

AUTO	**Currently open files** — always in context. Claude reads the active editor tab and any other open tabs automatically. The more relevant files you have open, the better Claude's cross-file answers will be. Action: open the files relevant to your task before starting a cross-file conversation.
ON DEMAND	**Files Claude reads when needed** — when you ask a question that requires a specific file, Claude reads it. It can also proactively read files it identifies as relevant from imports or references in code you've shown it. Action: name the file explicitly if Claude needs something you haven't opened — "see how this is done in services/auth.py".
LIMITED	**Project-wide search** — Claude can search across the project using Grep and Glob, but this is bounded by what the search returns. A search for a function name finds all references; it doesn't read every file in the project. Action: for broad questions, ask Claude to search first and tell you what it found before drawing conclusions.

NOT AVAILABLE

Runtime state, git history, external systems — Claude reads static files. It cannot see what's currently running, what values exist in the database, or what changed between commits.

Action: provide this context manually when it's relevant — paste the relevant git diff, the DB query result, or the runtime log.

Cross-File Prompt Patterns

DEPENDENCY TRACE

```
Which files in this project import or depend on `UserService`? I'm planning to rename it and need to know the full blast radius.
```

Claude searches the project and returns the list. Verify with Ctrl+Shift+F as a second source of truth.

DATA FLOW ACROSS MODULES

```
Trace what happens to `order.total` from the moment it's calculated in order_service.py to when it's stored. What files touch it along the way?
```

Open the entry and exit files before asking. Claude follows the data through imports and function calls.

CONSISTENCY CHECK

```
I've added a new field `cancelled_at` to the Order model. What other files might need updating — serialisers, schemas, tests, migrations?
```

Claude reads the model and scans the project for related patterns. Works best after you've made the model change so Claude can see it.

PATTERN SURVEY

```
How does this project handle authentication across different endpoints? Look at a few route files and tell me if the pattern is consistent.
```

Useful for onboarding and for spotting divergence before it becomes a security problem.

DEAD CODE DETECTION

```
Is `LegacyPaymentAdapter` referenced anywhere other than its own file? I want to delete it if nothing uses it.
```

Combine with a manual search. Claude's answer is a strong indicator but may miss dynamic references (e.g. string-based imports).

INTERFACE ALIGNMENT

```
The `process()` method on `OrderService` changed its return type. Show me every place it's called and whether the caller handles the old or new return value.
```

Open both the service and the key callers first. Claude checks each call site and flags mismatches.

Making a Project-Wide Change Safely

1. **State the change and ask for a plan first.** "I want to change the `User.username` field to `User.handle` everywhere in the project. Before we start, list every file and type of change that will be needed." Claude produces a change plan you can review.

2. **Cross-check Claude's list with a project search.** Use `Ctrl+Shift+F` to search for `username`. Compare against Claude's list — any file Claude missed needs to be added to the plan.

3. **Start with the definition, not the callers.** Make the change at the source (the model or interface) first. This makes the compiler or type checker flag all the broken callers immediately, giving you a definitive list.

4. **Work file by file, reviewing each diff.** Ask Claude to update one file at a time. Read the diff before accepting. Batch changes across files make it hard to isolate the source of a newly introduced bug.

5. **Run the test suite after each file.** A test failure immediately after updating a specific file tells you exactly where the problem is. Waiting until all files are done makes attribution much harder.

6. **Do a final search sweep.** After all changes are accepted, search the project for the old name one more time. Any remaining hits that aren't comments or string literals are missed updates.

> **DYNAMIC REFERENCES SLIP THROUGH**
>
> Claude and `Ctrl+Shift+F` both find *static* references. If your codebase uses dynamic dispatch, string-based imports (`importlib` , `require(variableName)`), or config-driven class loading, those references won't appear in a text search. Flag these areas explicitly when planning a rename.

Asking Architecture Questions

Claude can answer questions about how your system is designed — not just what individual functions do. These questions require reading multiple files and synthesising an answer, which Claude handles well as long as you give it the right starting points.

> *"Where should I add the new notification logic — in the service layer or the controller?"*

Opens a discussion about your existing layering. Claude reads the service and controller pattern, then makes a recommendation consistent with what's already there.

> *"Is there a single place where all external API calls are made, or are they scattered across the codebase?"*

Claude surveys import patterns and call sites. Useful before deciding whether to centralise HTTP logic.

> *"What would break if I removed the caching layer from the user lookup?"*

Claude traces what depends on the cached value. Gives you the blast radius before you experiment.

> *"Does this project follow a consistent error handling strategy, or are there different approaches in different modules?"*

Claude reads several modules and compares patterns. Surfaces inconsistency before it becomes a maintenance burden.

> *"I need to add rate limiting. Where in the existing request pipeline should it go?"*

Claude reads the middleware stack and routing, then suggests an insertion point consistent with existing middleware.

> *"How tightly coupled is the payment module to the order module? Could they be separated?"*

Claude examines imports and shared types between the modules. Gives you a coupling assessment before a refactor decision.

> **GROUND ARCHITECTURE QUESTIONS IN CONCRETE DECISIONS**
>
> Vague architecture questions ("is this codebase well-structured?") produce vague answers. Anchor every architecture question to a concrete decision you're facing ("I need to add X — where should it go and why?"). Claude gives more useful answers when the question has a practical outcome.

The Limits of Cross-File Understanding

Limitation	Why it happens	Workaround
Large projects exceed context	A project with hundreds of files can't all be read into context simultaneously. Claude reads what it can and may miss relevant files.	Open the most relevant files manually before asking. Name the files Claude should read explicitly.

Limitation	Why it happens	Workaround
Generated or compiled code	Auto-generated files (protobuf outputs, ORM migrations, bundled JS) are readable but rarely useful for architecture questions.	Direct Claude to the source definitions, not the generated outputs.
Implicit conventions	If a pattern exists in your codebase but isn't written down and the example files aren't open, Claude won't know about it.	Open a representative example or describe the convention before asking Claude to follow it.
Runtime vs static divergence	What the code says and what actually runs can differ (feature flags, environment config, monkey-patching). Claude only sees the static code.	Provide runtime context manually: "in production, this flag is always false" or paste the relevant config values.

Next — Chapter 8: Real-World Workflow

The final chapter — a complete feature built from start to finish with Claude as pair programmer. From reading the ticket to committing tested code: planning, generating, refactoring, debugging, and writing tests as a connected workflow rather than isolated operations.

Claude in VS Code: Pair Programming

Chapter 8 · Real-World Workflow — A Feature Built Start to Finish

The previous seven chapters covered individual techniques in isolation: explanation, refactoring, code generation, debugging, testing, cross-file changes. This final chapter puts them together — a complete feature from ticket to commit, with Claude as the pair programmer throughout. The goal is to show how the techniques connect in practice, and what the rhythm of real AI-assisted development looks like.

The Feature: Password Reset via Email

We'll build a password reset flow for a FastAPI web application. It's a realistic, self-contained feature that touches several layers: database schema, service logic, API endpoint, email dispatch, and tests. Here's the ticket:

PROJ-247 FEATURE

Add password reset via email

DESCRIPTION

Users should be able to reset their password via a tokenised email link. The flow: request reset → receive email → click link → set new password.

ACCEPTANCE CRITERIA

- POST /auth/password-reset-request accepts an email; always returns 200 (no user enumeration)
- A unique, time-limited token is stored against the user record
- A reset email is sent with a link containing the token
- POST /auth/password-reset accepts token + new password; invalidates the token on use
- Expired or already-used tokens return 400

Phase 1 — Orient and Plan

PHASE 1 · EXPLANATION

Understand the existing codebase before writing anything

```
I'm about to add a password reset feature (PROJ-247). Before I start, help me understand the existing auth setup. Open files: models/user.py, routes/auth.py, services/auth_service.py. What does the current auth flow look like, and what patterns should I follow for the new feature?
```

Claude reads the three files and summarises: the User model fields, how existing routes are structured, how the auth service handles tokens, and what email-sending utilities already exist. This takes two minutes and prevents an hour of wrong-direction work.

PHASE 1 · PLANNING

Ask Claude to produce a change plan

```
Based on what you've seen, what changes will this feature require? List: new DB columns or tables, new service methods, new routes, new tests, and any existing code that needs updating. Don't write any code yet.
```

Claude produces a list: add `reset_token` and `reset_token_expires` to User, write `AuthService.generate_reset_token()` and `AuthService.consume_reset_token()`, add two new routes, update the email service, write tests. Review and adjust the plan before proceeding.

Phase 2 — Schema and Migration

PHASE 2 · GENERATION

Generate the database migration

```
Write an Alembic migration to add `reset_token` (String, nullable, unique) and `reset_token_expires` (DateTime, nullable) to the users table. Follow the style of the existing migrations in alembic/versions/. Include both upgrade() and downgrade().
```

Claude generates the migration file. Open an existing migration first so Claude mirrors your exact Alembic style. Review: check the column types match your DB, the constraint name doesn't clash, and downgrade() reverses the change cleanly.

Phase 3 — Service Logic

PHASE 3 · GENERATION

Generate the service methods

```
Add two methods to AuthService in services/auth_service.py: 1. `generate_reset_token(email: str) → None` — looks up the user, generates a secure token (secrets.token_urlsafe), sets reset_token and reset_token_expires (now + 1 hour), saves, then calls email_service.send_reset_email(). Returns None regardless of whether the email exists. 2. `consume_reset_token(token: str, new_password: str) → None` — finds the user by token, raises ValueError("invalid or expired token") if not found or expired or already used, hashes and sets the new password, clears both token fields, saves. Follow the existing method style in the file.
```

Providing the full method spec eliminates ambiguity. Claude writes both methods. Review: check the token expiry comparison uses timezone-aware datetimes if your project does, and that the password hashing uses the same utility as the rest of the auth service.

Phase 4 — API Routes

PHASE 4 · GENERATION

Generate the two new endpoints

```
Add two routes to routes/auth.py following the existing pattern: - POST /auth/password-reset-request: accepts `{"email": str}`, calls auth_service.generate_reset_token(), always returns 200 with `{"message": "If that email exists, a reset link has been sent"}`. - POST /auth/password-reset: accepts `{"token": str, "new_password": str}`, calls auth_service.consume_reset_token(), returns 200 on success, catches ValueError and returns 400 with the error message. Don't add rate limiting — that's handled by middleware.
```

The constraint about rate limiting is important — Claude would otherwise add it. After generation, manually verify the routes are registered in the router and the request/response models match your Pydantic conventions.

Phase 5 — Debugging a Problem

After wiring the routes up and running the app, the first manual test fails. The reset email sends but the token lookup returns 404.

PHASE 5 · DEBUGGING

Diagnose the failure with Claude

```
I'm getting a 404 when calling /auth/password-reset with a valid token. The token was definitely saved — I can see it in the DB. Here's the consume_reset_token method and the full traceback. What could cause the token lookup to fail even when the token exists in the database?
```

Claude diagnoses: the token was saved with a trailing newline from `secrets.token_urlsafe()` being stored via a form field that didn't strip whitespace. The fix: `.strip()` on both store and lookup. A five-minute diagnosis that would have taken much longer without the static code analysis.

Phase 6 — Tests

PHASE 6 · TESTING

Generate the test suite

```
Write pytest integration tests for the two new endpoints using the test client. We use a real test DB (see conftest.py for fixtures). Test cases needed: - reset-request: valid email returns 200, non-existent email also returns 200 (no enumeration) - reset: valid token sets new password and clears token fields, expired token returns 400, already-used token returns 400, malformed token returns 400 Name tests as test_[endpoint]_[condition]. Follow the pattern in tests/test_auth.py.
```

Six test cases specified explicitly. Claude generates them using the existing fixture pattern. Run the suite immediately — fix any import errors or fixture mismatches before declaring done.

ASK CLAUDE FOR THE EDGE CASES YOU MISSED

After writing the specified tests, ask: "*What edge cases for this feature haven't I tested yet?*" Claude flagged two more: a token generated for a deleted user, and a race condition where two reset requests are made in quick succession. Both worth adding.

Phase 7 — Review Before Commit

PHASE 7 · REVIEW

Final review pass with Claude

```
Before I commit this feature, review the new code for: security issues, missing error handling, anything inconsistent with the existing codebase, and anything I might have missed from the ticket requirements.
```

Claude flags one issue: the token is logged at DEBUG level in a middleware, which would expose it in log files. Adds a redaction. This is the kind of subtle security issue that code review catches — and Claude catches it here before it ever reaches a reviewer.

The Rhythm of AI-Assisted Development

Looking back at the full workflow, a pattern emerges:

Orient first

Read before writing. Ask Claude to explain the existing code and plan the change before generating anything.

Spec before generate

Tell Claude exactly what you want — signature, behaviour, constraints. Vague prompts produce vague code.

Review every diff

Read every change before accepting it. The diff view is your safety net — use it every time.

Diagnose, then fix

When something breaks, ask Claude to explain the cause before applying a fix. Understanding prevents recurrence.

Test as you go

Run tests after each phase, not just at the end. Failures are easier to attribute to a specific change.

Always review security

Ask Claude for a security pass before committing anything that handles auth, input, or external data.

What Claude Does Not Replace

⚠ **Your judgement.** Claude generates options and suggestions. You decide what to accept, what to reject, and what to change. Never commit code you haven't read and understood.

⚠ **Domain knowledge.** Claude doesn't know your business rules, your users, or your production constraints. You bring that; Claude handles the mechanical implementation.

⚠ **Runtime verification.** Tests pass, code reads correctly — but the only way to know the feature works end-to-end is to run it. Claude can't do that for you.

⚠ **Architecture decisions.** Claude can tell you where a change fits within existing patterns. Deciding whether those patterns are right for the next stage of the product is a human call.

⚠ **Accountability.** When the code ships and something goes wrong, it's your name on the commit. Claude is the tool; you are the engineer.

🎓

Claude in VS Code: Pair Programming — Complete

8 chapters · From setup to shipping a real feature with Claude

8	6	1
CHAPTERS	CORE TECHNIQUES	COMPLETE FEATURE