AI-Assisted Software Development with Claude Code

Analyze, plan, implement, test, and review a real Spring Boot feature in a real Claude Code terminal session: Plan Mode, the permission model, and bugs actually caught along the way.

Intermediate 35 min

Last updated: August 22, 2026
Read online: https://www.learnforgex.com/en/topics/developing-with-claude-code

---

AI-Assisted Software Development with Claude Code

So far in this category, AI has been something you studied from the outside: what it is, how it works, how agents plan and use tools. This lesson flips that around -- you'll use AI as a development tool yourself, in your own terminal, on a real project. Claude Code is an AI coding assistant that lives in your terminal: it can actually read and analyze a project directory, write and modify files, and run commands. This lesson doesn't describe that in the abstract -- it walks through a real Spring Boot feature (adding a Quiz feature to this very platform) being built, start to finish, in a real Claude Code terminal session.

> The screens shown in this lesson (Plan Mode questions, approval screens, file permission prompts) were observed verbatim in a real Claude Code CLI session -- but the CLI's exact wording and flow can change between versions. The point here isn't to memorize a command list, it's to learn the durable workflow underneath it (analyze -> plan -> implement -> test -> review -> git); that workflow holds even if the screens in your own environment look slightly different.

What We'll Build

1. We'll start Claude Code from the terminal, inside this platform's real code base (the `learning-platform` Spring Boot project).
2. We'll give it a real task: adding a 5-question, bilingual (TR+EN) multiple-choice Quiz feature to the `enum` topic.

3. We'll see how Plan Mode works before any code is written -- how it asks clarifying questions and produces a plan file.
4. We'll see why reading a plan before approving it matters, using a real bug that was actually caught this way.
5. We'll follow the implementation through the permission model, approving changes file by file, diff by diff.
6. We'll actually run and test the app -- and catch a real bug (a broken HTML tag) and a real content-quality problem (every correct answer sitting in the same slot) along the way.
7. Finally, we'll cover the review and git steps, and Claude Code's permission/security model.

What Is Claude Code, and Why Is It Different?

You've probably asked an AI chat window for code before: you ask a question, you get a snippet back, and you copy-paste it into your project yourself. Claude Code inverts that: it's a terminal tool that runs inside your project directory, so it can actually **read your files** (without you pasting them in), actually **write changes to files**, and, when needed, actually **run commands** (`mvn test`, `npm install`, `git diff`, and so on). That difference -- "describe what to do and copy-paste it yourself" versus "hand off a task and let it actually work in your project" -- is the reason the rest of this lesson exists: once a tool can genuinely modify your files, permission and review stop being optional habits and become part of how you use it at all.

Installing and Starting It in a Project Directory

Install steps vary by platform and change over time -- check the official Claude Code installation docs for the current, accurate instructions. Once installed, the tool starts with the `claude` command. The critical part is **where** you run it --

```
cd learning-platform
claude
```

-- because Claude Code uses the directory it's running in as its own context: it infers which files exist, how the project is organized, which languages/frameworks are in play. That's why you don't have to manually explain "here's how my project is structured" the way you would to a chat-based AI -- Claude Code starts by reading the files itself, and keeps reading them as needed.

Project Context: The Role of CLAUDE.md

When a project root contains a file named `CLAUDE.md`, the documented behavior is that Claude Code reads it automatically and treats it as persistent, project-specific context -- architectural decisions, "rules that never change," previously-discovered constraints, coding

conventions (the exact reading/prioritization mechanism can change between versions, so check the official docs for the current details). This platform's own `CLAUDE.md` does exactly that: rules like "Flyway migration numbers are sequential and never renumbered retroactively," or "business logic belongs in the Service layer, not the Controller." Without this file, Claude Code would have to guess these conventions from scratch on every task -- sometimes it guesses right, sometimes (as you'll see later in this lesson) it doesn't. `CLAUDE.md` is the AI equivalent of a human telling a new teammate "read this doc first."

Step 1 -- Analyze: Bringing Claude Code Into the Task

The first step in the workflow isn't saying "code this" directly -- it's describing what needs to happen, clearly. This lesson's terminal session started with a prompt close to this (abridged):

```
I want to add a 5-question, bilingual (TR+EN), multiple-choice Quiz feature
to the `enum` topic in the Java Fundamentals category. The DB schema, the
endpoint contract, and the architectural decisions are already settled
(below). Analyze the project first, then propose a plan in Plan Mode --
don't start writing code yet.
```

Two things matter here: the task is **concrete and bounded** (exactly which topic, how many questions, which languages -- not a vague "build a quiz system"), and the prompt explicitly says **"don't start writing code yet."** That second part is what triggers the next section, Plan Mode.

How to Give an Effective Task

The prompt above wasn't written casually -- it's worth thinking of a task description at three levels:

- **Bad:** "Build a quiz system." This forces Claude Code to guess which topic, how many questions, which languages, and what constraints apply -- the result usually scopes itself up (attempt history, a leaderboard, a timer, an admin UI, none of which anyone asked for).
- **Better:** "Add a 5-question, bilingual multiple-choice quiz to the `enum` topic. Analyze first, produce a plan, don't write code yet." Concrete and bounded, but still leaves the DB schema and architectural decisions to Claude Code.
- **Best:** a task that hands over **existing architecture + scope + acceptance criteria + constraints** together, the way the real prompt above did. It opened with "the DB schema, the endpoint contract, and the architectural decisions are already settled" -- a real schema, a real endpoint contract, a real layering decision, and an explicit list of what was out of scope (attempt history, a leaderboard, randomization, a timer, an admin UI) were all handed over up front. Claude Code was only asked to implement that, not to decide the schema or architecture on its own.

That progression is, in fact, how this very lesson -- and the plan behind its hands-on task -- came together: scope and constraints were nailed down by a human first, and Claude Code was given "implement exactly this scope, under these constraints," not "design a quiz system."

Step 2 -- Plan Mode: Planning Before Writing Code

Plan Mode is one of Claude Code's important features: it reads and analyzes files, but produces a plan **before changing anything** -- and waits for you to approve it. This is the concrete version of adding a human-approval checkpoint to the observe-decide-act loop from "What Is an AI Agent?".

While in Plan Mode, Claude Code can ask you **clarifying questions** at points that are genuinely ambiguous before it continues. Two questions were actually asked in this session:

```
Who decides the content of the 5 (TR+EN) quiz questions?
  1. I'll write them (Recommended)
  2. You (the user) write them, I'll just wire them into the schema
  3. ...
```

```
After the user submits the quiz, should they be able to change their
answers and resubmit?
  1. No, one attempt only (Recommended)
  2. Yes, unlimited
  3. ...
```

Two things are worth noticing. First, the questions are about genuinely **open** points -- things like the DB schema and endpoint contract, which were already settled, are never re-asked; only details the plan hasn't covered yet are. Second, every option carries a "(Recommended)" tag -- but that tag doesn't mean **"accept without thinking."** It's Claude Code's own suggestion, nothing more. Both recommendations here were evaluated independently before being accepted (the first with the explicit caveat that "even if I write the content, it must be reviewed before it's published"; the second with the reasoning that "we don't keep any persistent attempt history anyway, so a page reload already gives the same effect as unlimited retries -- one attempt is simpler") -- not a blind yes.

Once the questions are answered, Claude Code updates the plan, and you can preview it with `/plan`:

```
❯ /plan
#  Current Plan
   /Users/.../.claude/plans/{auto-generated-name}.md

   Quiz Feature -- for the enum Topic (Java Course, java-basics Category)
   ...
```

The plan is stored on disk as a real Markdown file (under `~/.claude/plans/`, with a name auto-generated from the task) -- it isn't ephemeral text that disappears from the chat, it's a persistent document you can open in your editor with a shortcut like `ctrl+g`.

Human Approval: The Ready-to-Code Gate

Here's the behavior we observed in this session: when Claude Code finished writing its plan, it didn't move on to writing code automatically -- it asked for explicit approval:

```
Claude has written up a plan and is ready to execute.
Would you like to proceed?

  1. Yes, and use auto mode
> 2. Yes, manually approve edits
  3. Tell Claude what to change

ctrl+g to edit in VS Code
```

This screen is a live example of "Human-in-the-Loop: Approval Before Risky Actions" from "Controlling Agent Behavior" -- an agent (Claude Code, here) pausing before a risky action (writing files, running commands) to ask a human for approval. The three options really do offer three different autonomy levels: **auto mode** runs every step of the approved plan back to back without asking again; **manually approve edits** asks you to approve every file change/command one at a time; **tell Claude what to change** lets you correct the plan before any code is written at all.

Throughout this lesson, **"manually approve edits"** was deliberately chosen -- not auto mode. The reason: it's the safer default on a task you're not yet familiar with (see "Security and Permissions"). Two real problems came up in this session: a bug in the plan itself (next section), and an HTML bug during implementation (a few sections later). The first -- the migration number mistake -- was caught by reading the plan **before** approving it, which happens regardless of which mode you pick next. The second -- the HTML bug -- wasn't caught in the file diff even under manual approval; it only surfaced once the app was actually tested. Manual approval still paid off, though: it gave the chance to verify every remaining file against the known architectural decisions, one at a time.

Reading a Plan Before Approving It: A Real Bug

We said above not to trust the "(Recommended)" tag blindly -- here's exactly why. The plan Claude Code produced named its migration files `enum/V7__enum_quiz_questions.sql` and `enum/V8__enum_quiz_options.sql`.

Checked against the project's actual file system, that was wrong. In this project, Flyway migration numbers follow **one single global sequence, independent of folder** (as `CLAUDE.md` itself states, subfolders are purely a filesystem-level organization) -- and the project had already reached V258 by that point. The `enum/` folder itself only went up to V6,

but **V7 and V8 were already taken by other migrations elsewhere in the project** (`core/V7__category_sort_order.sql` and `record/V8__records_topic.sql`). Had the plan been applied as written, Flyway would most likely have failed on startup with a duplicate-version error.

This doesn't mean Claude Code was "bad" -- quite the opposite, the plan was otherwise flawless (schema, permission model, endpoint contract, layering). What it does show is that an AI can misjudge a project's **local convention** (here: that version numbers are project-wide, not per-folder) even when that convention is written down explicitly in `CLAUDE.md` , if it hasn't been fully internalized from context. So the plan was sent back with a correction request using the third option ("Tell Claude what to change") -- before a single line of code was written.

> Before approving a plan, checking the file names/numbers it claims against the real project state (`ls` , `find` , `grep`) is a much cheaper and much more effective verification step than reading the rest of the plan -- exactly as it was here.

The Permission Model: Approving File by File

Once the corrected plan was approved, here's what we observed in "manually approve edits" mode: Claude Code asked for permission on each file separately -- showing the file's full content and offering three options:

```
Do you want to create V259__quiz_schema.sql?
> 1. Yes
  2. Yes, and switch to accept edits (auto-approve file edits and common
     file commands) for this session (shift+tab)
  3. No
```

Every file in this session was deliberately approved with **"1. Yes"** -- "2. auto-approve" was never selected. Every remaining file (entities, repositories, service, controller, template, JS, migrations) was reviewed and approved individually; as you'll see in "Step 3 -- Implementation: Following the Diffs", each file was actually checked against the known architectural decisions (FK pattern, the language converter, column mapping) before it was approved.

But be careful: manual approval doesn't catch everything. The missing `>` character you'll see in "A Real Debugging Session: A Broken HTML Tag" wasn't noticed while the file diff was being approved -- it only surfaced once the app was actually run and tested. That's exactly why "Step 4 -- Test: Running It and Verifying It by Hand" is a separate, non-skippable discipline: reading diffs and actually running the app catch different classes of mistakes, and neither substitutes for the other.

Step 3 -- Implementation: Following the Diffs

While approving file by file, it makes sense to check each diff against known architectural decisions -- rather than saying "Yes" out of habit. When `QuizQuestion.java` came up for approval, for example, the things actually checked were: whether the `topic` field matched the project's `CodeExample` entity's FK pattern (`@ManyToOne(fetch = FetchType.LAZY)` + `@JoinColumn(name = "topic_id", nullable = false)`) exactly; whether `@Enumerated` had mistakenly been added to the `language` field (this project converts language automatically via a separate `LanguageAttributeConverter`, and adding `@Enumerated` would break that); and whether `sort_order` mapped to the right column name. All of it was correct, so the file was approved -- but only **after** those three points were actually checked for this new entity, not because the file simply existed.

A Real Debugging Session: A Broken HTML Tag

Once the migrations and the Java side were approved and the app restarted, the `/en/topics/enum` page crashed with:

```
org.attoparser.ParseException: (Line = 208, Column = 24) Malformed markup:
Attribute "class" appears more than once in element
```

The error message itself was a little misleading -- "class defined twice" didn't quite describe the real problem. The actual root cause was that the `<nav>` opening tag on line 206 of `topic.html` was **missing its closing `>`**:

```
<nav class="d-flex justify-content-between mt-5 pt-3 border-top"
     aria-label="Topic navigation"
<a th:if="${previousTopic != null}"
   class="btn btn-outline-secondary text-start"
   ...
```

Because the `>` was missing, the parser assumed `<nav ...` was still open, and kept treating the following `<a>` tag's own `class` attribute as part of the same (still-unclosed) element -- two separate `class` attributes (one belonging to `nav`, one to `a`) collided inside what the parser saw as a single tag. The fix was one character: adding the missing `>` to the end of line 206.

This shows two separate skills. First, **reading a real stack trace** and tracing an error (the line/column attoparser reported) back to its actual source -- looking at what the error message means, not just what it literally says. Second, feeding the fix back to Claude Code as a **complete, specific** instruction ("something's broken here, fix it" versus "the `<nav>` tag on line 206 is missing its closing `>`, add it") -- which gets you a faster, more reliable fix.

Step 4 -- Test: Running It and Verifying It by Hand

Once the HTML was fixed, the app ran successfully, and the 5-question quiz actually rendered on `/en/topics/enum`: each question with 4 radio-button options, and a "Submit" button that stayed disabled until every question was answered. Once all five were answered and submitted, the score, the correct/incorrect marking, and each question's explanation genuinely appeared.

The gap between a feature "appearing to work" and actually being "correct" is exactly the subject of the next section.

Step 5 -- Review: Working and Correct Aren't the Same Thing

In the first test, the quiz worked flawlessly at a technical level -- the score was computed correctly, the partial unique index held, every question had exactly one correct option. But a closer look at the content itself turned up a real problem: **the correct option for all 5 questions sat in the same, first slot (option A)** on screen. Technically flawless, pedagogically pointless -- a student could mark "A" on every question without reading any of them and still score 5/5.

This repeats the same lesson taught earlier by the migration-number bug in "Reading a Plan Before Approving It: A Real Bug": an AI's suggested content/plan **working** doesn't mean it's **correct** -- and that distinction matters even more once content authorship ("I'll write them (Recommended)") has been deliberately handed to the AI.

The fix required a careful distinction: **which option was correct** shouldn't change (each option's text makes a specific factual claim; moving "correct" onto a false statement would make the lesson itself wrong) -- only **where the correct option appeared on screen** should.

The Fix: A New Migration, or Editing the Existing One?

Since the app was already running and rendering the quiz, the problematic migrations (`V260` , `V261`) had already been applied by Flyway and recorded in `flyway_schema_history`. `CLAUDE.md` 's rule that "Flyway migration numbers are sequential and never renumbered retroactively" answers this exactly: instead of editing an already-applied migration in place, the fix was added as a **new migration** (`V262`) -- a set of `UPDATE` statements that reshuffled each question's four options' `sort_order` values, never touching `is_correct` .

After restarting the app and retrying the quiz, the correct answers now sat in different positions (one question's correct option at B, another's at C, for example) -- and the score still computed correctly as 5/5, because only the display order had changed; the scoring logic (comparing against `is_correct`) was never touched.

Step 6 -- Git: Reviewing Changes and Committing

The last step in the workflow isn't committing the moment files "work." `git status` and `git diff` show which files changed or were added -- your IDE's (e.g. IntelliJ) "Changes" view does the same thing visually -- and that's your last review opportunity before a commit. Scanning that list yourself before telling Claude Code to "commit the changes" is a natural continuation of the same file-by-file approval discipline you've been practicing throughout: a commit message an agent proposes is, just like a plan it proposes, a draft worth reviewing. A good commit message summarizes **why** you made a change, not what changed (the diff already shows that); and splitting a large feature into logical commits (e.g. schema + seed data, entity/repository/service, controller + template + JS) instead of one giant commit makes it much easier to trace a bug later with `git bisect`.

Security and Permissions

Because Claude Code can write files and run commands, using its permission model deliberately is central to using the tool safely. "Scoping Tool Access: The Principle of Least Privilege" from "Controlling Agent Behavior" applies here too: if you're new to a directory, or the task is risky (database schema, deletion operations, requests to external services), starting with **"manually approve edits"** is safer than trusting "auto mode." Auto mode saves time on low-risk tasks you've run many times and know the outcome of -- but as this lesson itself showed, plan mistakes and implementation bugs really do happen on a new task, and manual approval is precisely the mechanism that catches them. A rough rule: if a command's or file change's outcome is hard or expensive to undo (applying a migration to production, deleting a file, a `git push --force`), never leave that step to automatic approval.

Other Claude Code Features, Briefly

The analyze-plan-approve-implement-test-review-git flow used throughout this lesson is Claude Code's core -- but the tool offers a few more capabilities, all extensions of the same underlying idea (extend the context, keep a human in the loop on risky steps): files like `CLAUDE.md` at a project root provide persistent context (as seen in "Project Context: The Role of CLAUDE.md"); **Subagents** split a large task into independent subtasks that can run in parallel; **MCP (Model Context Protocol)** lets Claude Code connect to external tools /services with the same protocol seen in "Building an MCP Server"; **Skills** package repeated task patterns into something reusable. This lesson isn't the place to go deep on any of these -- the point is that they all repeat, at different scales, the same discipline you just practiced by hand: give it context, review the plan, approve the risky step.

Best Practices

- Before starting a task, tell Claude Code to "analyze first and propose a plan in Plan Mode, don't start writing code yet" -- see "Step 1 -- Analyze: Bringing Claude Code Into the Task".
- Hand over existing architecture, scope, acceptance criteria, and constraints together, not just a bare instruction -- see "How to Give an Effective Task".
- Before approving a plan, check the file names/assumptions it claims against the real project state -- see "Reading a Plan Before Approving It: A Real Bug".
- Start with "manually approve edits" on a new or risky task, and only move to auto mode on low-risk tasks you're already familiar with -- see "Security and Permissions".
- Give Claude Code a complete, specific diagnosis before asking it to fix a bug (which file, which line, root cause) -- see "A Real Debugging Session: A Broken HTML Tag".
- Do a separate "is it correct" review after the "does it work" test, especially when content /logic was AI-generated -- see "Step 5 -- Review: Working and Correct Aren't the Same Thing".
- Never edit an already-applied migration in place -- always add the fix as a new migration -- see "The Fix: A New Migration, or Editing the Existing One?".

Common Mistakes

- **Trusting a "(Recommended)" tag, or a plan that "looks reasonable," blindly.** The migration-number collision in "Reading a Plan Before Approving It: A Real Bug" would have crashed the app on startup had it been let through exactly this way.
- **Treating a feature as done just because it ran without errors.** As "Step 5 -- Review: Working and Correct Aren't the Same Thing" shows, the quiz worked flawlessly at a technical level while its content was pedagogically meaningless -- a separate "is it correct" review was the only thing that caught it.
- **Substituting approving a file diff for actually running and testing the app.** As "The Permission Model: Approving File by File" shows, the missing `>` in `topic.html` wasn't noticed while the file was being approved -- it only surfaced in "Step 4 -- Test: Running It and Verifying It by Hand" once the app was actually run. Reading diffs and actually running/testing the app are two disciplines, and neither replaces the other.
- **Editing an already-applied migration in place the moment you find a problem.** That breaks Flyway's checksum verification and violates `CLAUDE.md`'s "never renumbered retroactively" rule -- the right move is always a new migration.

Summary, Cheat Sheet, and Glossary

Summary

- Claude Code is a terminal AI coding assistant that runs in your project directory, can genuinely read and write files, and can run commands -- fundamentally different from copy-pasting into a chat window.
- The durable workflow is six steps: analyze, plan in Plan Mode, pass through human approval, follow the implementation with file-by-file approval, test, review and commit.
- A good task description hands over existing architecture, scope, acceptance criteria, and constraints together -- a vague task like "build a quiz system" lets Claude Code guess the scope itself, usually wider than intended.
- Plan Mode produces a plan before any code is written, and asks clarifying questions at genuinely open points -- the "(Recommended)" tag is a suggestion to evaluate, not a blind-approval signal.
- The "Ready to code?" approval gate offers three autonomy levels (auto mode / manual approval / change the plan) -- a live example of the human-in-the-loop idea from "Controlling Agent Behavior".
- Three real problems were actually caught in this lesson -- a migration number collision (a plan bug), a broken HTML tag (an implementation bug), every correct answer sitting in the same slot (a content-quality bug) -- each demonstrating a different review discipline: reading the plan, reading error output, and separating "it works" from "it's correct."
- An already-applied migration is never edited in place -- the fix is always a new migration.

Cheat Sheet

- Start: `claude`, inside the project directory.
- Context: `CLAUDE.md` at the project root, read automatically.
- Workflow: analyze -> Plan Mode -> "Ready to code?" approval -> file-by-file implementation approval -> test -> review -> git.
- Approval levels: "Yes" (this file/step) / "Yes, auto-approve" (everything for this session) / "No" (reject) / "Tell Claude what to change" (revise the plan).
- Safe default: manual approval on a new/risky task; auto mode only on a low-risk task you're already familiar with.
- Migration rule: sequential, global, never renumbered retroactively -- the fix is always a new file.

Glossary

- **Claude Code:** a terminal AI coding assistant that uses a project directory as its context, can read/write files, and can run commands.
- **Plan Mode:** the mode where Claude Code produces a plan and waits for approval before changing any file.
- **CLAUDE.md:** a file at a project root that Claude Code reads automatically as persistent, project-specific context/rules.
- **Permission model:** the approval options Claude Code offers before a file change /command (single step, auto for the rest of the session, reject, revise the plan).

- **Auto mode:** the autonomy level where every step of an approved plan runs back to back without asking again.
- **Review:** the separate step of verifying a feature is not just "working" but "correct" (content, logic, security) -- distinct from testing, and comes after it.