📘 Master Study Notes: Agentic Coding Using Claude Code (CampusX)

Table of Contents

Chapter 1: Learn AI Coding the Right Way (No Vibe Coding)

Part 1: Introduction to the Claude Code Era

The software development landscape is undergoing a massive shift with the introduction of AI coding tools, and Anthropic's **Claude Code** is rapidly emerging as the industry standard [1]. Traditional manual coding workflows are becoming obsolete, as developers

integrating AI tools are experiencing up to a 10x increase in both productivity and output quality [2].

The Mental Model Shift: To succeed in modern software development, you must transition your mindset from being a manual coder who writes every single line of code, to acting as a **Product Manager or System Architect** [2]. In this new dynamic, the human developer maintains high-level system understanding and architectural control, while the AI executes the granular coding tasks [2].

Part 2: Vibe Coding vs. Agentic Coding

A critical distinction in AI programming is understanding the difference between "Vibe Coding" and the superior "Agentic Coding" approach taught in this module.

1. Vibe Coding Vibe coding is an experimental, fast-paced development style where the user relies almost entirely on the AI (LLM) to make decisions, generate, and refine code based on natural language prompts (e.g., "Build me a to-do list website") [3].

The Vibe Coding Workflow: 1. **Input:** User writes a plain English prompt explaining the desired app [3]. 2. **Action:** The AI coding tool generates the entire codebase, making its own architectural decisions [3]. 3. **Validation:** The user tests the application [4]. 4. **Iteration:** If bugs exist, the user prompts the AI to fix them, repeating the loop until the application works [4].

> ⚠ **WARNING**
>
> **The Dangers of Vibe Coding** > While vibe coding is excellent for non-technical users, hackathons, and building quick Minimum Viable Products (MVPs), it fundamentally breaks down in production environments [4]. It is strictly for low-stakes projects [4]. You cannot use vibe coding to build highly scalable systems (e.g., Google-scale traffic) or critical infrastructure (e.g., banking or insurance software) because the developer surrenders architectural control to the AI [4, 5].

2. Agentic Coding (AI-Assisted Coding) Agentic coding is a structured, robust approach to building software. Instead of blindly trusting the AI, the developer maintains strict control over the architecture, utilizes specialized AI sub-agents for specific tasks, and relies on proper engineering workflows to build scalable, production-grade applications [5].

Part 3: Project Architecture: The Expense Tracking Application

To master Agentic Coding, the workflow will be applied to building an end-to-end **Expense Tracking Website** [6].

Core Features: * Landing page with User Registration and Login authentication [6]. * User Dashboard displaying summary statistics (total spent, transaction count, top spending categories) [7]. * Transaction history tables and visual category breakdowns [7]. * Custom date-range filtering (e.g., viewing data strictly from the last two months) [7]. * Full CRUD operations: Add, Edit, and Delete expenses [7].

Why this specific project? The project is intentionally kept simple regarding business logic [8]. A straightforward application ensures that the learner's focus remains entirely on mastering **Claude Code workflows, multi-agent parallel development, and cloud deployment**, rather than getting stuck debugging complex business rules [8].

Part 4: Tool Selection: Why Claude Code?

While there are many AI coding assistants on the market (such as Cursor, Windsurf, OpenAI Codex, Replit, and Lovable), **Claude Code** is explicitly chosen for serious, scalable software development due to four distinct advantages [9, 10]:

Feature	Why it is superior in Claude Code
Raw Coding Intelligence	Powered by Anthropic's flagship **Opus** model, it is highly capable of reasoning through and executing highly complex application features [10].
Long Context Window	It excels at ingesting and holding massive codebases in its context window, making it highly effective for large-scale refactoring and architectural planning [10].
Agentic Capabilities	Claude Code allows developers to spin up specialized "sub-agents" (e.g., a dedicated testing agent or a code review agent) to execute development tasks in parallel [10, 11].
Senior Developer Persona	Interacting with Claude Code's architecture feels remarkably similar to pairing with a highly experienced Senior Software Engineer [11].

Part 5: Overcoming the Fear of AI

A major psychological goal of this methodology is to eliminate the widespread fear that AI will replace developer jobs [12].

The Horror Movie Analogy: The instructor compares the fear of AI to watching a horror movie [13]. In a horror film, the audience is terrified of the ghost as long as it remains hidden in the shadows (the unknown) [13]. However, the moment the ghost's identity and backstory are fully revealed, the fear instantly dissipates [13]. Similarly, developers fear AI because it is an "unknown" entity taking over their tasks. By learning Claude Code fundamentals and actively integrating the tool into a daily workflow, the unknown becomes known. The fear is replaced by practical, educated utilization [13].

Part 6: Target Audience & Prerequisites

This workflow is highly recommended for software developers, data professionals (Data Scientists, Analysts, Engineers), and students preparing to enter the job market [13, 14].

> 💡 **IMPORTANT**
>
> **Strict Prerequisites** > Before utilizing Claude Code to generate project architectures, the developer *must* have foundational knowledge of the underlying technologies. > * **Backend:** Python & Flask [14]. > * **Frontend:** Basic HTML & CSS [14]. > * **Version Control:** Git & GitHub. The agentic workflow relies extensively on branching, committing, and PR merges. A lack of Git knowledge will severely disrupt the AI coding process [14, 15].

> 📝 **NOTE**
>
> **Terminal Commands Notice** > *Because this specific chapter serves as the theoretical foundation and architectural overview of the Agentic Coding methodology, no specific CLI, bash, or Claude Code slash commands are executed in this module. Practical terminal execution begins in subsequent pipeline stages.*

Chapter 2: How to Setup Claude Code on your System

Part 1: Initial Setup & Installing Claude Code (Paid Approach)

The standard, most powerful way to use Claude Code requires a paid subscription to Anthropic's Claude Pro (approx. $24 or ₹2000/month). This grants access to Anthropic's proprietary models (like Opus and Sonnet) which provide the highest level of coding intelligence.

Step-by-Step Workflow: System Installation

1. **Purchase Subscription:** Go to `claude.ai`, create an account, and upgrade to the paid plan.
2. **Copy Install Command:** Navigate to the official Claude Code homepage and copy the provided installation command.
3. **Execute Installation:** Paste the command into your system's command prompt/terminal to install Claude Code globally.
4. **First Run & Authentication:** `bash claude`
5. **What it does:** Initializes the Claude Code interface in your terminal.
6. **Why/When to use it:** Used to start a new agentic coding session in any directory.
7. **Expected Output:** On the first run, Claude will ask if you "trust" the current folder. Select **Yes**. It may also open a browser window to authenticate your Anthropic account.

> 📝 **NOTE**
>
> > **Real-World Simulation Mental Model:** The tutor starts the project using a pre-built skeleton rather than from scratch. This mimics the real-world scenario of a software engineer joining a company—you rarely start from zero; instead, you inherit an existing codebase, understand its architecture, and build features on top of it.

Part 2: Project Workspace & Virtual Environment Configuration

Once Claude is installed, the next phase involves setting up the local workspace for the specific project (a Flask-based expense tracker named **Spendly**).

Step-by-Step Workflow: Environment Setup

1. **Download & Extract:** Download the starter project `.zip` file, extract it, and open the folder in **VS Code**.
2. **Open Terminal:** Open a new terminal instance within VS Code and type `claude` to start the session inside the project root.

3. **Enter Bash Mode:** Press `Shift + 1` (`!`) to enter **Bash Mode** directly inside the Claude Code interface.

> 💡 **IMPORTANT**
>
> > **Why use Bash Mode inside Claude instead of a separate terminal?**

> If you run commands in a separate terminal, Claude is blind to them. By running bash commands *inside* the Claude interface, the commands, their outputs, and any installed libraries become part of Claude's **Context Window**. You can immediately ask Claude questions like, "*What libraries were just installed?*" and it will know the exact answer.

1. **Create a Virtual Environment:** `bash python3 -m venv venv`
2. **What it does:** Creates an isolated Python environment to prevent dependency conflicts.
3. **Activate the Virtual Environment:** `bash source venv/bin/activate`
4. **What it does:** Activates the virtual environment for the current session.
5. **Install Dependencies:** `bash pip install -r requirements.txt`
6. **What it does:** Installs all project-specific libraries (like Flask) listed in the text file.
7. **Run the Application:** `bash python3 app.py`
8. **What it does:** Starts the local Flask development server (running on port 5001).

Part 3: Git & GitHub Integration Workflow

Maintaining version control from day one is an industry-standard best practice. This helps with deployment and tracking changes made by the AI.

Step-by-Step Workflow: Repository Initialization

1. **Create Remote Repo:** Go to `github.com` and create a new repository named `spendly`.
2. **Initialize Local Git (Inside Claude's Bash Mode):** `bash git init`
3. **What it does:** Initializes an empty Git repository locally.
4. **Stage Files:** `bash git add .`
5. **What it does:** Stages all current skeleton code files.
6. **Commit Changes:** `bash git commit -m "initial commit"`
7. **What it does:** Saves the staged changes locally.

8. **Link Remote & Push:** `bash git remote add origin <YOUR_GITHUB_REPO_URL> git push origin main`
9. **What it does:** Links the local code to the GitHub repository and pushes the initial state to the cloud.

Part 4: Exploring a New Codebase Using Claude Code

When inheriting a new or partially built codebase, you must build a mental model of its architecture. You can prompt Claude Code to act as a senior engineer and explain the repository to you.

The 3 Golden Prompts for Codebase Exploration:

1. **Prompt:** `What does this project do?`
2. **Expected Output:** Claude scans the files and deduces the business logic (e.g., "This is a Flask-based personal expense tracker web app...").
3. **Prompt:** `What tech stack does this project use?`
4. **Expected Output:** Claude identifies the specific technologies. In this project:
    - **Backend:** Python 3.11, Flask, SQLite.
    - **Frontend:** Jinja2 templating, Vanilla CSS, Google Fonts.
    - **Testing:** PyTest, PyTest-Flask.
5. **Prompt:** `Explain the project structure to me.`
6. **Expected Output:** Claude generates a detailed hierarchical graph of folders and explains key design decisions.

Part 5: Running Claude Code for Free (The Ollama Alternative)

For students or developers who cannot afford the $24/month subscription, the tutor introduces a "Jugaad" (workaround) using **Ollama** to run open-source Large Language Models (LLMs) as the brain behind the Claude Code harness.

Method A: Using Ollama Cloud Models

1. **Install Ollama:** Download and install Ollama from its official website.
2. **Launch Claude via Ollama:** `bash ollama launch claude`
3. **What it does:** Starts Claude Code but routes the intelligence engine through Ollama instead of Anthropic's paid API.

4. **Select a Cloud Model:** A menu will appear. Select a highly capable coding model (e.g., **Qwen 3.5 Coder**).
5. **Gotcha:** While free, Ollama's cloud models have strict weekly and per-session rate limits. You will hit a usage cap quickly on complex tasks.

Method B: Using Local Offline Models (True Free Tier)

To bypass cloud rate limits entirely, you can download an open-source model directly to your machine's hardware.

1. **Pull the Local Model:** `bash ollama pull qwen2.5:7b`
2. **What it does:** Downloads the 7-billion parameter Qwen 2.5 Coder model to your local machine.
3. **Hardware Warning:** Running local models requires sufficient RAM. A 14-Billion parameter model requires ~16GB RAM. For lower specs, stick to 7B models.
4. **Launch & Select Local Model:** Run `ollama launch claude` again. Scroll down the menu to the "Local Models" section and select the newly downloaded Qwen model.
5. **Trade-off:** Local models are entirely free and have no rate limits, but response times will be noticeably slower depending on your machine's GPU and RAM configuration.

Terminating the Session

```
/exit
```

- **What it does:** A built-in Claude slash command that safely terminates the current agentic coding session and closes the terminal interface. Used when you are done working or need to switch environments.

Chapter 3: Slash Commands in Claude Code

Part 1: Core Fundamentals of Slash Commands

Slash Commands represent one of the most critical structural features of Claude Code, transforming it from a standard conversational AI into a highly efficient agentic coding tool [1].

What are Slash Commands? Slash commands are predefined shortcuts typed directly inside a Claude Code session that begin with a forward slash (`/`). They instantly trigger specific actions, configuration changes, or complex workflows without requiring the developer to type out extensive, repetitive natural language prompts [1].

The "Why" Behind Slash Commands: In traditional interactions with LLMs, programmers often repeat the same lengthy prompts to execute standard workflows (e.g., code reviews, committing code, generating tests). The creators of Claude Code recognized these repeatable patterns and packaged them into single-word commands [2]. This design architectural choice drastically reduces cognitive load, saves keystrokes, and ensures determinism in how standard workflows are executed [2].

There are two primary categories of Slash Commands: 1. **Built-in Commands:** Out-of-the-box commands natively available upon installing Claude Code [2, 3]. 2. **Custom Slash Commands:** User-defined commands tailored to automate specific project workflows (covered practically in advanced chapters) [3].

Part 2: Session Management and Navigation

To understand slash commands, one must first understand **Sessions**. A **Session** is a single, isolated conversational thread with Claude Code [4]. It acts as a dedicated context window that captures the full message history (prompts, tool uses, and AI replies) [4]. Sessions are automatically saved locally and can persist even if the terminal is closed [4, 5].

Starting and Exiting Sessions

```
claude
```

- **What it does:** Initializes a fresh Claude Code session from your command line [4].
- **Why/When to use it:** Use this in your project root directory to start a new feature development task.

```
/exit
```

- **What it does:** Instantly terminates and closes the current Claude Code session [3, 4].

- **Why/When to use it:** Use this when a specific task or feature is complete to prevent context window overflow before moving on to the next task.

Resuming Sessions

```
claude -r
```

- **What it does:** Opens a terminal menu displaying a list of your past conversational sessions, allowing you to select and resume an older session [5, 6].
- **Why/When to use it:** Crucial for multi-day workflows. If you paused working on a feature yesterday, this command restores your exact place with full context intact [5].

```
/resume
```

- **What it does:** Allows you to switch to a different historical session *mid-way* through your current session [6].
- **Why/When to use it:** Use when you realize you are working in the wrong session, or need to quickly jump back to a previous feature's context without exiting the CLI [6].

> 💡 **IMPORTANT**
>
> > **Best Practices for Session Architecture [7, 8]:** > 1. **One Task = One Session:** Never build multiple features in a single session. Isolating tasks keeps the context window clean and prevents the AI from mixing up logic [7]. > 2. **Rename Immediately:** Claude auto-names sessions based on your first prompt, which is often vague. Take control and rename it instantly [7]. > 3. **Commit Frequently:** Commit your Git changes at every meaningful milestone within the session [8].

Part 3: Core Workflow Optimization Commands

```
/rename <session_name>
```

- **What it does:** Manually overrides the auto-generated name of your active session (e.g., `/rename Login Feature`) [8].
- **Why/When to use it:** Execute this immediately after starting a session to ensure your history remains organized and easy to navigate via `claude -r` [7, 8].

```
/btw <your_question>
```

- **What it does:** "By The Way" evaluates a prompt and answers it *without* adding the question or the answer to the session's permanent context history [9, 10].
- **Why/When to use it:** Use this for tangential reference lookups (e.g., `/btw What is the Jinja templating engine in Flask?`). This allows you to gain knowledge in parallel while Claude is coding, without polluting your feature's context window with unrelated trivia [9, 10]. Pressing the `Space` bar dismisses the answer [10].

```
/export <filename.md>
```

- **What it does:** Takes the entire conversation history of the current session and exports it as a tangible Markdown file in your project directory [11].
- **Why/When to use it:** Highly valuable during major code refactoring. You can export a complex session and feed that `.md` file back into a future session as direct architectural context [11].

Part 4: Account & Model Configuration

```
/logout
```

```
/login
```

- **What they do:** Logs you out of or into your Anthropic/Claude Code account [12].
- **Why/When to use it:** Useful if you are juggling multiple accounts (e.g., switching between a personal subscription and a corporate enterprise account) [12].

```
/model
```

- **What it does:** Opens an interactive menu to switch the underlying Large Language Model powering your Claude Code session [13, 14].

The Claude Code Model Ecosystem [13, 15]:

Model	Capability / Use Case	Price / Speed
Opus (3.0)	Anthropic's flagship model. Used for highly complex reasoning, deep architectural decisions, and writing initial spec documents.	Most expensive, slowest, highest token consumption.
Sonnet (3.5)	The "All-Rounder". The default model offering a perfect balance. Used for reliable code generation and everyday tasks.	Balanced cost, speed, and quality.
Haiku (3.0)	The fastest model. Used for simple, repetitive, or boilerplate programming tasks.	Cheapest and fastest.

> 🌱 **TIP**
>
> > **The Power User Model Workflow:** > A highly effective architectural pattern is to use **Opus** for the planning phase (thinking through architecture, making decisions, writing specs), and then switch to **Sonnet** for the actual implementation phase (writing the code based on the Opus plan) [14].

Part 5: Usage Tracking and Analytics

```
/usage
```

- **What it does:** Displays your real-time token consumption at two levels: Current Session percentage and Weekly Limit percentage [16].
- **Why/When to use it:** Claude Code has strict token limits. Check this frequently (especially if using Opus) to ensure you aren't burning through your token budget unexpectedly [16, 17].

```
/extra_usage
```

- **What it does:** Allows you to mid-way top up or recharge your token limits via billing (e.g., $5, $10 top-ups) [17].
- **Why/When to use it:** Prevents workflow blockers if you hit your usage limit during a critical development sprint [17].

```
/stats
```

- **What it does:** Shows high-level statistics about your Claude Code usage (active days, longest sessions, current streaks, preferred models) [18].

```
/insights
```

- **What it does:** Generates a highly detailed, localized HTML report analyzing your specific usage behavior and providing personalized recommendations [19].
- **Why/When to use it:** Run this after completing 10-15 sessions. It acts as an audit to tell you what you are doing right, what you are doing wrong, and how to improve your agentic workflows [19].

Part 6: Tool Permissions & Security Customization

Claude Code operates as an autonomous agent by utilizing "Tools" (e.g., Read files, Write files, Bash execution, Web Search) [20, 21]. By default, Claude asks for your permission before executing a tool to ensure safety [21].

```
/permissions
```

- **What it does:** Opens a text-based UI containing tabs (`Allow` , `Ask` , `Deny` , `Workspace`) to configure which tools Claude can run autonomously versus which require manual approval [22].
- **Why/When to use it:** Constant permission prompts cause friction. Use this to automate safe tools [21, 22].

Step-by-Step Permissions Workflow [22, 23]: 1. Type `/permissions` and navigate to the **Allow** tab. 2. Select **"Add a new rule"**. 3. Type the tool name (e.g., `Web Search`) to allow Claude to browse the internet without asking [22]. 4. To allow specific bash commands, wrap them in brackets: `bash(git init)` [24]. 5. Select the **Scope** for this permission [22, 23]: * *Local Project Setting:* Saves to the current repository only (`.claude/settings.local.json`). * *Global Project Setting:* Applies to anyone who forks or works on this shared repository. * *User Setting:* Applies globally to every project on your specific laptop.

> 💡 **IMPORTANT**

> **Safety Warning regarding Bash Permissions:** > Never globally allow destructive bash commands. Only whitelist benign commands (like `git init`) that are safe to run autonomously. Deny specific tools if you never want Claude touching them [24].

Part 7: Interface & Accessibility Commands

```
/config
```

- **What it does:** Opens a table of configuration settings (e.g., Toggling Thinking Mode, Verbose outputs, Terminal progress bars) [20].

```
/theme
```

- **What it does:** Toggles the CLI visual theme (Dark Mode, Light Mode, etc.) [25].

```
/voice
```

- **What it does:** Enables Voice Mode. Once enabled, you can hold the `Space` bar to speak your prompts directly into your laptop microphone instead of typing [25, 26].
- **Why/When to use it:** Drastically speeds up prompt generation when explaining complex architectural requirements verbally is easier than typing them out [26].

> 📝 **NOTE**
> **Command Discovery:** > If you ever forget a command, simply type `/` and hit the `Tab` or `Arrow` keys to scroll through a complete, self-documenting list of every available slash command and its description [26].

Chapter 4: Making Code Changes using Claude Code

Part 1: Session Setup and Project Strategy

The primary objective of this chapter is to practically apply Claude Code to an existing codebase (an expense tracker application) by improving its landing page. The workflow introduces three core capabilities: making targeted code changes, generating entirely new pages from scratch, and leveraging multimodal capabilities (image-to-code).

Initial Setup Workflow: 1. Open the project repository in **VS Code**. 2. Launch a new Claude Code session via the terminal. 3. Immediately rename the session to establish a clear context and make future retrieval easier.

```
claude
```

Explanation: Initializes a new, fresh Claude Code session in the terminal within the current project directory.

```
/rename Landing Page Improvements
```

Explanation: Renames the auto-generated session title. Why use this? Claude auto-generates names based on the first prompt, which can be vague. Renaming ensures you can easily identify and resume this specific feature's workspace later using `claude -r`.

Part 2: Modifying Existing Code (Footer Links)

The first development task is to add "Terms and Conditions" and "Privacy Policy" links to the landing page's footer.

> 🌱 **TIP**
>
> > **Best Practice - Pre-written Prompts:** The tutor strongly recommends curating and refining prompts in a separate text file (or via standard Claude/ChatGPT web interfaces) before pasting them into the Claude Code terminal. **Why?** Typing complex prompts manually in the CLI is error-prone and increases the likelihood of omitting crucial architectural constraints.

The `@` Mention Feature To modify an existing file safely, Claude Code utilizes the `@` symbol to bring specific files into the context window deterministically.

```
Add two links to the footer in @base.html. The links should be "Terms and Conditions" and "P
```

Explanation: This prompt uses `@base.html` to explicitly target the file. Why? Without `@` mentions, Claude might hallucinate file paths or edit the wrong component. This ensures deterministic file targeting.

Executing Atomic Commits Following the philosophy of Agentic Coding, the tutor mandates committing code after every meaningful milestone, rather than waiting for the entire feature to be complete.

```
git add .
git commit -m "add footer links"
```

Explanation: Standard Git commands executed directly within the Claude Code terminal (or via Claude's bash mode by pressing `Shift + 1` / `!`). This captures the milestone immediately.

Part 3: Generating New Pages (Terms & Privacy)

The next step is to build out the actual pages for the newly added footer links. This requires Claude to touch multiple files simultaneously (routing logic and frontend templates).

Workflow: Multi-File Generation 1. **Provide a comprehensive prompt** detailing the routes, templates, and content expectations. 2. **Review the execution:** Claude will dynamically read `app.py`, generate a new HTML template, and update the `href` links in `base.html`. 3. **Iterative Refinement:** If the generated page looks unstyled (a common occurrence with AI generation), follow up immediately with a styling constraint.

```
Create a Terms and Condition page for Spendly. Add a new route to @app.py. Create a template
```

Explanation: Notice how multiple files (`@app.py`, `@templates/terms.html`) are mentioned. The prompt enforces that Claude adapts the new template to the existing CSS theme.

After Claude implements and tests the `terms.html` page, the process is repeated identically for `privacy.html`.

```
git add .
git commit -m "added terms and privacy pages"
```

Explanation: Another atomic commit to lock in the successful generation of the new pages and routes.

Part 4: Multimodal Capabilities (Image-to-Code)

This section demonstrates one of Claude Code's most powerful features: passing UI/UX reference images directly into the CLI to generate pixel-perfect code.

Workflow: Spec-Driven UI Generation 1. Obtain the reference design (e.g., a mockup exported from Figma). 2. Paste the image directly into the Claude Code terminal (using `Ctrl+V` or `Cmd+V`). Claude will attach the image as a URL/context object. 3. Provide a strict text prompt to bound Claude's blast radius so it only affects the target UI component.

```
[Pasted Image Reference]
Modify only the hero section in @templates/landing.html and @static/css/landing.css to matc
```

Explanation: This multimodal prompt combines visual context (the image) with strict textual boundaries. Why? Telling Claude "Do not touch any other part of the page" prevents it from unnecessarily refactoring unrelated HTML/CSS, saving tokens and preventing regressions.

```
git add .
git commit -m "redesigned hero section based on mockup"
```

Explanation: Committing the successfully implemented multimodal UI update.

Part 5: Adding Functional Interactivity (Vanilla JS Modal)

The final feature is making a "See how it works" button open a modal overlay containing an embedded YouTube video.

> 💡 **IMPORTANT**
>
> > **Enforcing Tech Stack Constraints:** When asking AI to build interactive components, it will often default to bringing in heavy external libraries (like jQuery, React, or Bootstrap). You must explicitly define your tech stack constraints in the prompt.

```
Add a modal to @templates/landing.html that opens when the user clicks 'See how it works'. F
- Clicking will open a modal overlay.
- Modal contains an embedded YouTube video (Use any placeholder URL).
- Clicking the close button or clicking outside the modal closes it.
- When the modal closes, the video MUST stop playing, not continue in the background.
- NO page libraries or dependencies. Vanilla JavaScript only since we are not using any JS f
- Do not modify any other part of the project.
```

Explanation: A highly defensive prompt. It specifies edge cases (stopping video playback on close so audio doesn't bleed) and enforces the "Vanilla JS only" rule to prevent tech debt.

Part 6: Finalizing, Pushing, and Session Management

Once all local changes for the landing page are complete, tested, and individually committed, the workflow concludes by pushing to the remote repository.

```
git push origin main
```

Explanation: Pushes all local atomic commits to the GitHub repository.

> 📝 **NOTE**
>
> > **Gotcha - Duplicate CSS Files:** The tutor notices a mistake made during the multimodal prompt. Claude generated a new `landing.css` file instead of appending to `style.css` due to a slightly inaccurate prompt instruction. The tutor highlights that this is a normal part of AI coding: you identify the hallucination or file duplication and resolve it iteratively (e.g., by prompting Claude to "merge these two CSS files").

To finish the workflow, the developer should monitor token usage to ensure they haven't blown through their session budget, and then properly terminate the workspace.

```
/exit
```

Explanation: Safely closes the current Claude Code session, freeing up terminal resources while saving the conversation history locally so it can be resumed later if needed.

Chapter 5: Context Window Management in Claude Code

Part 1: Terminal vs. GUI Access Modes in Claude Code

When working with Claude Code, there are multiple access modes available, including the Claude desktop app, web interface, and the VS Code plugin [1]. However, accessing Claude Code directly via the terminal (command prompt) is the preferred "OG" method, especially for power users [1, 2].

Why use the Terminal? The terminal provides the most powerful and unrestrictive environment for Claude Code [2]. If you use GUI-based interfaces (like the VS Code extension), you will be blocked from accessing advanced functionalities [3]. For example, if you attempt to edit memory or use Hooks inside the GUI, the system will prompt you to "Continue in terminal" because those features are unsupported outside the CLI [4]. Learning the terminal method ensures you can harness the maximum potential of the tool without limitations [4].

Part 2: Understanding Context and the Context Window

To manage a context window, you must first understand what "context" is in a programming environment [5].

What is Context? Context represents all the available information required to understand and execute a task correctly [5]. For a software developer, context comes from multiple sources: * The entire project codebase [6]. * PRD (Product Requirement Documents) or Spec documents [6]. * Jira or GitHub issues detailing bugs or tasks [6]. * Slack messages containing team discussions or algorithmic ideas [6]. * Previous chat history with the AI coding tool [6]. * The Git repository [6].

What is a Context Window? A Context Window is the maximum amount of information (measured in tokens) that a Large Language Model (LLM) like Claude can "see" and "remember" at any one time while generating a response [5, 7].

> 📝 **NOTE**
>
> > **Mental Model:** Think of the Context Window as the AI model's "working memory" or RAM [5, 7]. Just as a computer cannot load an infinitely large program

> into its RAM, you cannot feed an infinitely large codebase (e.g., 1 million lines of code) into an LLM at once due to strict token limits [8].

Part 3: Claude Code's Specific Context Window Mechanics

Claude Code operates with specific token limits and behaviors that dictate how memory is consumed during a coding session.

1. The Token Limit Claude Code (specifically using the Sonnet model) provides a context window of **200,000 tokens** [9]. While the Opus model supports up to 1 million tokens, 200k is the standard limit for everyday coding tasks using Sonnet [9].

2. Fresh Sessions = Fresh Memory Every time you initiate a new session in Claude Code, you are granted a completely fresh, empty context window [9].

3. Input and Output Consumption Tokens are consumed by both your inputs (prompts, pasted code, images) and Claude's outputs (code generation, tool executions, explanations) [10].

> 💡 **IMPORTANT**
>
> > **The 6x Rule:** Claude's generated output (responses and tool executions) generally consumes tokens at a rate roughly **6 times faster** than your input messages [10].

4. The Accumulation Problem (Why Context Fills Rapidly) Because LLMs are inherently stateless, they do not remember past conversational turns automatically [11]. To maintain an ongoing conversation, the coding harness must resend the *entire conversation history* to the LLM with every new message [11].

The Step-by-Step Accumulation Loop: 1. **Turn 1:** You send a message (1,000 tokens), Claude replies (1,000 tokens). Total used = 2,000 tokens [11, 12]. 2. **Turn 2:** You send a new message. Claude Code resends Turn 1 + the new message. Total used = 4,000 tokens [12]. 3. **Turn 3:** You send another message. Claude Code resends Turn 1 + Turn 2 + the new message. Total used = 6,000 tokens [12]. 4. **Turn 10:** By the 10th turn, you are spending massive amounts of tokens (e.g., 20,000 tokens) just to ask a simple question, purely because the history is attached [12].

Part 4: The "150K Reality" (What Occupies the Context Window)

Although Claude Code advertises a 200,000 token limit, you actually only have about **150,000 usable tokens** [13].

> 🌱 **TIP**
>
> > **Analogy:** Buying a laptop with 8GB of RAM. When you turn it on, the OS consumes 2GB, leaving you with only 6GB of usable RAM [13]. The context window works the exact same way.

Certain tokens are pre-reserved and occupied the moment you start a session [13]. Here is the architectural breakdown of what occupies your context window:

Component	Token Cost (Approx)	Description
System Prompt	~6,000 tokens	Hidden instructions from Anthropic detailing Claude's persona and capabilities [13, 14].
Tool Schemas	~8,000 tokens	Definitions, inputs, and output structures for tools (Read, Write, Bash, etc.) [14].
Auto-Compaction Buffer	~33,000 tokens	Pre-reserved space specifically kept empty to store conversational summaries when the context gets too full [15, 16].
`CLAUDE.md`	Variable	Your project overview file, pre-loaded into every session [14].
Conversation History	Variable	The ongoing chat and tool execution outputs [14, 15].
MCP Tools/Skills	Variable	Definitions for any active Model Context Protocol servers or Skill Markdown files [14, 15].

Monitoring Your Context Window

```
/context
```

- **What it does:** Displays a real-time dashboard of your current context window consumption, showing exact token counts for System Prompts, Tools, Auto-compact buffer, and free space [16].
- **When to use it:** Run this periodically during a long coding session to ensure you are not approaching the 150k usable limit [17, 18].

Part 5: Why Context Window Management is Critical

Failing to manage your context window leads to three severe negative outcomes:

1. **Increased Financial Cost:** Because you pay per token, sending massive conversational histories back and forth in a bloated session burns through your token budget rapidly [19].
2. **Poor Workflow Structure:** Trying to build four different software features in a single session forces the LLM to carry irrelevant context, increasing token usage by 4x compared to using isolated sessions per feature [20].
3. **Degradation of Response Quality:** As you approach the 150k limit (hitting around 120k - 130k tokens), the LLM's reasoning and coding quality severely degrades [20, 21]. It will start making mistakes and ignoring instructions [21].

Part 6: Strategies to Rescue and Manage the Context Window

When a session becomes too long and the context window is near capacity, you must take action to free up space [21].

1. Auto-Compaction (Reactive) When you consume 75% to 92% of your available tokens, Claude automatically triggers "Auto-compaction" [21, 22]. Claude pauses, reads the entire conversation history, generates a compressed summary, and moves this summary into the pre-reserved 33,000-token buffer [22]. It then deletes the raw history, freeing up main context space [22]. * *The Danger:* Auto-compaction is "lossy" (you lose fine details) and can happen randomly in the middle of a complex tool execution or feature build, causing the AI to lose track of crucial code [23].

2. Manual Compaction (Proactive)

```
/compact
```

- **What it does:** Manually triggers the compaction process, summarizing the current history into the buffer and freeing up tokens [24]. You can press `Ctrl+O` to view the generated summary [17].

- **Why to use it:** Proactively running `/compact` when you notice you are at 70%-75% usage ensures the AI doesn't summarize data unpredictably in the middle of writing an important feature [18, 23].

3. Subagent Delegation If your context is full but you have parallel tasks, you can trigger Subagents [25, 26]. * **Why to use it:** Every Subagent is spawned with its own isolated, fresh 200k context window [26]. They do the heavy lifting in a separate space and only return a small summary to the main agent, saving you tens of thousands of tokens [26, 27].

4. Clearing the Session

```
/clear
```

- **What it does:** Deletes the entire conversation history within the current session, returning you to a blank slate [18, 28].
- **Why to use it:** When your auto-compaction buffer (33k) is entirely full and you cannot compact anymore, you must use `/clear` or start a brand-new session to continue working [22, 24, 28].

Part 7: Best Practices for Context Window Optimization

To maximize efficiency and minimize cost, adhere to these architectural best practices:

1. **One Feature = One Session:** Never build multiple features in a single session. Once a feature is complete, close the session (`/exit`) and start a new one. This keeps the context window pure and unpolluted [28].
2. **Proactive Compaction:** Do not rely on Claude to Auto-compact. Use `/context` to monitor usage, and manually execute `/compact` during natural resting points in your workflow [28].
3. **Write Highly Focused Prompts:** Avoid vague instructions. Clear, specific prompts restrict unnecessary token generation and keep the context lean [29].
4. **Delegate Exploratory Work to Subagents:** Use Subagents for code-base exploration or isolated parallel tasks to protect the main agent's context window [29].
5. **Use `.claudeignore` Files:**
6. **What it does:** Functions exactly like a `.gitignore` file, but for Claude [29].
7. **Why to use it:** By adding large, irrelevant files (like build artifacts or `venv` folders) to `.claudeignore` , you physically block Claude from reading them, preventing massive files from accidentally bloating the context window [3, 29].

Chapter 6: Claude.md — The Most Important File

Part 1: The Problem of Stateless LLMs & The `CLAUDE.md` Solution

The Core Problem: LLMs are Stateless By nature, Large Language Models (LLMs) lack memory. When you start a new conversation or session with an AI coding agent, it has no context of your past interactions. In a software development workflow, this means you would have to manually re-explain your entire project architecture, tech stack, coding conventions, and folder structure from scratch every time you start a new session to build a feature.

Why this is problematic: 1. **Cumbersome:** Typing out the same intricate project details repeatedly wastes time. 2. **Error-Prone:** Missing out on minor but critical details leads to inconsistent code generation (e.g., the AI might use `psycopg2` when your project strictly uses `SQLAlchemy`).

The Solution: `CLAUDE.md` To solve this, Claude Code introduces `CLAUDE.md`. * **Mental Model:** Think of `CLAUDE.md` as a **persistent system prompt**. * It is a special project-level instruction file saved in your project directory. Whenever a new Claude Code session starts, the AI automatically pulls and reads this file, instantly understanding how it should behave and the specific constraints of your codebase.

Part 2: Creating the `CLAUDE.md` File

There are two primary methods to create this file:

1. Manual Creation

Create a markdown file explicitly named `CLAUDE.md` (all uppercase) in your project's root directory and write all instructions from scratch.

2. Automated Creation via Slash Command (Preferred)

You can delegate the heavy lifting to Claude Code by using the initialization command.

```
/init
```

What the command does: Automatically scans your entire codebase and generates a foundational `CLAUDE.md` file in your root directory. **Why/When to use it:** Highly recommended when starting a project, onboarding onto someone else's large codebase, or when you are unfamiliar with the optimal structure of a `CLAUDE.md` file. It prevents you from missing subtle project patterns.

Step-by-Step Workflow of `/init`: 1. **Agent Trigger:** Claude starts an internal subagent. 2. **High-Signal Scan:** The agent scans critical configuration files first (`package.json`, `requirements.txt`, `README.md`). 3. **Structure Analysis:** It maps out the directory tree structure to understand where logic lives. 4. **Pattern Recognition:** It identifies the tech stack, naming conventions, and folder layouts. 5. **File Generation:** It compiles this context and writes the `CLAUDE.md` file.

> ⚠ **WARNING**
>
> > **The 30/70 Rule:** The automated `/init` command only generates about **30%** of a truly useful `CLAUDE.md`. The remaining **70%** (specific workflows, business logic constraints, and things to avoid) must be manually curated and added by the programmer.

Part 3: Architecture of an Ideal `CLAUDE.md`

An effective `CLAUDE.md` acts as the single source of truth for the AI. It should structurally contain:

1. **Project Overview:** A concise 1-2 line description so Claude immediately knows what it is building (e.g., "*This is a FastAPI backend for a health tracking app that stores patient records and exposes CRUD APIs.*").
2. **Architecture & Directory Structure:** Clear mapping of what files belong where (e.g., "*Routes live in `/routes`, business logic in `/services`*").
3. **Coding Style:** Uniform conventions the AI must follow (e.g., "*Use Python type hints, prefer Pydantic models, keep functions small and focused*").
4. **Preferred Libraries & Tools:** Strict tech stack boundaries (e.g., "*Use vanilla CSS only, no external styling libraries*").
5. **Commands:** Exact CLI commands used in the project for building, running the dev server, and testing.

6. **Critical Rules & Warnings:** Explicit "Do Nots" (e.g., "*Do not generate Patient IDs automatically, do not modify* `database.py` *unless explicitly asked*").
7. **Development Roadmap (Tutor's Custom Addition):** A list of planned API routes/features and their current completion status. This aligns Claude with your development timeline.

Part 4: The `.claude` Configuration Folder

Mental Model: If `CLAUDE.md` is the brain's instruction manual, the `.claude` folder is the **Toolbox** for Claude Code.

It is a local configuration directory that stores all your Agentic tools, including Custom Slash Commands, Skills, Subagents, and JSON settings.

Comparison of `.claude` Folder Scopes:

Feature	Project-Level `.claude` Folder	Global/User-Level `.claude` Folder
Location	Project Root Directory	Machine's Home Directory (`~/.claude`)
Scope	Applies ONLY to the current project	Applies to ALL projects on the machine
Git / Sharing	Committed to repo; shared with team	**NOT** committed; strictly personal
Use Case	Project-specific commands, settings, and workflows	Personal coding style, global tools, personal defaults

---

Part 5: Types and Locations of `CLAUDE.md` Files

Depending on your needs, `CLAUDE.md` files can be placed in different locations. Claude Code resolves them based on specific behaviors.

1. **Root Directory (`CLAUDE.md`):** The standard project-level instruction file.

2. **Inside Configuration Folder (`.claude/CLAUDE.md`):** Functionally identical to the root file. Used purely if you prefer keeping all configuration files neatly tucked inside the `.claude` folder.
3. **Personal Project Preferences (`.claude.local.md`):**
    - **Why use it:** For adding project instructions that suit your personal workflow but shouldn't be forced on your teammates.
    - > 📝 **NOTE**
      > This file is automatically added to `.gitignore` . It is read alongside the main `CLAUDE.md` but never pushed to the repository.
4. **Global/User Level (`~/.claude/CLAUDE.md`):** Contains your universal coding persona (e.g., preferred languages, broad coding habits). Applies across every project you open on that machine.
5. **Subdirectory Level (e.g., `frontend/CLAUDE.md`):**
    - **Why use it:** Essential for large, monorepo codebases.
    - **Behavior:** It is *not* loaded globally. It is loaded recursively only when Claude is instructed to work inside that specific subdirectory, allowing for "Lazy Loading" of context.

Part 6: Best Practices & Handling Large `CLAUDE.md` Files

Execution Best Practices: * **Treat it as a Living Document:** Update it organically after every newly developed feature. Remove irrelevant, outdated instructions. * **Codify Mistakes:** If Claude repeatedly makes the same error, don't just correct it in the chat. Tell Claude to add the corrected instruction directly into the `CLAUDE.md` file. * **Use "Important" Sparingly:** You can tag lines with the word "Important" to force strict compliance. However, remember the rule: "*If everything is important, nothing is.*" Overusing it dilutes its impact.

> 💡 **IMPORTANT**
> **The 200-Line Limit:** Your `CLAUDE.md` file should ideally not exceed 200-300 lines. As the instruction count grows, the LLM's instruction-following quality degrades heavily. > **Rule of Thumb for Auditing:** Ask yourself, *"If I remove this line, will Claude start making mistakes?"* If the answer is no, delete the line.

Workflows to Handle Oversized `CLAUDE.md` Files: If your project dictates a massive ruleset, use these strategies instead of a single bloated file: 1. **Split into the `.claude/rules/` folder:** Break the file into topic-specific markdown files (e.g., `testing.md`, `security.md`) and place them in the `rules` folder. Claude will lazy-load these specific rule files only when the context demands it. 2. **Use Imports:** Similar to Python imports, instruct the main `CLAUDE.md` to reference other markdown files for specific domains (e.g., API guidelines). 3. **Subdirectory Contexts:** As mentioned earlier, push frontend rules to `frontend/CLAUDE.md` and backend rules to `backend/CLAUDE.md`.

Part 7: Auto Memory & `memory.md`

While the programmer writes `CLAUDE.md`, Claude Code maintains its own internal memory system.

Concept: Auto Memory As you work through various sessions, Claude silently observes patterns, preferences, and project insights. When it detects a meaningful pattern, it automatically saves it to a persistent file called `memory.md`.

- **Location:** `~/.claude/projects/<project_name>/memory/memory.md`
- **Behavior:** In every new session, Claude loads `CLAUDE.md` AND the top 200 lines of `memory.md`.

Managing Memory via Command:

```
/memory
```

What the command does: Opens an interactive menu to view and edit memory files. **Why/When to use it:** Use it when you need to audit what Claude has "learned" about your project or to clean up bloated auto-memory. **Menu Options Provided:** 1. *Project Memory:* Opens `CLAUDE.md` 2. *User Memory:* Opens your global `CLAUDE.md` 3. *Auto Memory Folder:* Opens `memory.md`

Manual Memory Update Workflow: You do not have to wait for Claude to infer a pattern. You can forcefully inject learnings into `memory.md` at the end of a session.

```
Update your memory files: We use INR and not USD
```

What the command does: Instructs Claude via a standard chat prompt to immediately write a specific rule into its `memory.md` file. **Why/When to use it:** When a critical business logic decision is made during a session and you want to ensure Claude remembers it for all future sessions without cluttering your main `CLAUDE.md` file. Output: Claude will confirm with a "Saved" response.

Chapter 7: Spec-Driven Development in Claude Code

Part 1: The Problem with Vibe Coding

The concept of **Spec-Driven Development** serves as one of the core pillars of **Agentic Coding**, directly contrasting with and solving the problems inherent in "Vibe Coding" [1].

Vibe Coding is a modern, fast, conversational, and experimental style of software development where a programmer relies heavily on AI tools (like Cursor, Lovable, or Replit) to generate, refine, and debug code based purely on natural language prompts without upfront planning [2]. For example, a user might simply prompt, "Build a To-Do application for me," and iterate through corrections until the app works [2].

> ⚠ **WARNING**
>
> > **The Core Flaw of Vibe Coding: Loss of Control** > In Vibe Coding, you surrender crucial architectural decision-making to the AI [3]. For example, if you prompt an AI to "build a user authentication system," the AI autonomously decides the framework, authentication method (JWT vs. sessions), password complexity rules, and rate-limiting [3, 4]. If these AI-made decisions do not align with your actual needs, you get stuck in a frustrating loop of patches and corrections [4, 5].

While Vibe Coding is excellent for non-technical users, exploring ideas, building Minimum Viable Products (MVPs), or hackathons, it completely fails for building scalable, high-stakes, or critical infrastructure systems (like banking or insurance software) [6, 7].

Part 2: Introduction to Spec-Driven Development (SDD)

Spec-Driven Development (SDD) is a software development approach where a detailed specification document (the "Spec Document") is written *before* any code is generated [8].

Why use SDD? The Spec Document acts as a **Single Source of Truth** for what the system should do [8]. Instead of the AI making ambiguous architectural decisions, the programmer retains full control, defines the precise parameters upfront, and the AI acts purely as the executor [5, 8]. This results in highly predictable, scalable, and consistent code [8, 9].

Part 3: Anatomy of an Ideal Spec Document

A standard Spec Document bridges the gap between human requirements and AI execution. It typically contains six core sections [10]:

1. **Problem Statement (The "Why"):** Clearly establishes the necessity of the feature [10].
2. *Example:* "Users cannot revisit past chats, making it difficult to find previous discussions." [11]
3. **Functional Requirements (The "What"):** Details exactly what will be delivered [10].
4. *Example:* "The system should display a sidebar showing a list of past chats with a short readable title." [12]
5. **Input/Output Behavior:** Defines how the system reacts to specific inputs [10].
6. *Example:* "Input: User clicks a past conversation. Output: The new chat loads in the main area." [12]
7. **System Constraints:** Technical or UX limitations [10].
8. *Example:* "The sidebar must load within 1 second and work properly on standard laptop screens." [12]
9. **Edge Cases & Error Handling:** Anticipates points of failure and dictates fallback behavior [10].
10. *Example:* "If no chat history exists, display 'No chat history yet'." [13]
11. **Acceptance Criteria (Definition of Done):** The checklist used to validate if the AI successfully wrote the code [10].
12. *Example:* "Feature is complete if the correct conversation is displayed every time it is clicked." [13]

Part 4: The Spec-Driven Development Workflow

The traditional SDD workflow in professional software teams follows a rigid, multi-step pipeline [14]:

1. **Create the Spec Document:** Draft the high-level, non-technical requirements ("What" and "Why") [14].
2. **Review the Spec Document:** Identify missing elements or errors before proceeding [14, 15].
3. **Create the Technical Design Plan:** Translate the non-technical Spec into a "How" document [15]. This includes the chosen tech stack (e.g., React, FastAPI), high-level architecture, data models, and functional flows [15, 16].
4. **Review the Technical Design Plan:** Ensure technical accuracy and correct architectural decisions [17].
5. **Extract Tasks:** Break the Technical Design Plan down into actionable engineering tasks (e.g., Database setup -> Backend Endpoints -> Frontend UI -> Integration) [17, 18].
6. **Build/Code:** Execute the tasks sequentially [18].
7. **Validate:** Compare the final coded feature against the Acceptance Criteria defined in step 1 [18, 19].

> 📝 **NOTE**
>
> > **Why separate the Spec Document from the Technical Design Plan?** > The Spec Document is product-focused, while the Technical Design Plan is engineering-focused [20]. Separating them allows a team to completely change the technology stack (e.g., migrating from Flask to Next.js) without needing to rewrite the core product requirements and logic stored in the Spec Document [17, 20].

Part 5: Comparing Vibe Coding vs. Spec-Driven Development

Feature	Vibe Coding	Spec-Driven Development (Agentic)
Starting Point	A rough idea defined via prompt [19].	A detailed, written specification document [19].
Requirements	AI decides based on ambiguity [19].	Programmer defines explicitly [19].
Control	Low. AI leads decision-making [9, 19].	High. Programmer leads the process [9].

Feature	Vibe Coding	Spec-Driven Development (Agentic)
Speed	Extremely fast upfront code generation [9].	Slower upfront (planning phase), fast execution [9].
Code Quality	Unpredictable; may veer off track [9].	Consistent, traceable, and exact [9].
Best Use Case	MVPs, prototypes, side projects [9, 21].	Production systems, scalable software [21].
Failure Mode	Incomprehensible, tangled "spaghetti" code [21].	AI might "over-engineer" simple features [21].
Debugging	Ask AI to fix observed errors [21].	Validate code directly against the Spec [21, 22].
Required Skill	Minimal programming knowledge needed [22].	Strong grasp of the programming language required [22].

Part 6: Adapting SDD for Claude Code (The Agentic Workflow)

Instead of manually writing all documentation, the AI-assisted Agentic Coding workflow delegates document *creation* to **Claude Code**, while the programmer handles *review and validation* [23].

Step-by-Step Agentic Workflow: 1. **Automated Spec Creation:** A custom slash command is used to prompt Claude to generate the Spec Document for a new feature [23, 24]. The programmer manually reviews it [24]. 2. **Automated Technical Design Plan:** Using Claude's internal `Plan Mode`, the AI studies the generated Spec Document and creates a Technical Design Plan [24]. The programmer manually reviews it [24]. 3. **Automated Task Extraction:** Claude automatically creates sequential tasks based on its Technical Design Plan [24]. 4. **Code Execution:** Claude writes the code. This can be done by a single agent, or delegated to multiple parallel sub-agents handling different files simultaneously [25]. 5. **Validation & Testing:** The programmer validates the code against the Spec Document's Acceptance Criteria and generates automated tests [25].

Part 7: Git & GitHub Integration in the SDD Workflow

To maintain a clean, production-grade codebase, the SDD workflow in Claude Code is tightly coupled with Git and GitHub. The operational loop for every single feature follows this strict Git sequence [25, 26]:

1. **Pull Latest Code:** Ensure you are starting with the most recent codebase from the remote repository.

```
git pull origin main
```

Why: Prevents merge conflicts by syncing local files with the team's central repository before starting new work [25].

1. **Create and Switch to Feature Branch:** Isolate the new feature development.

```
git checkout -b feature/<feature-name>
```

Why: Keeps the `main` branch clean. All SDD steps (Spec generation, Planning, Coding) happen safely inside this isolated feature branch [25, 26].

1. **Commit Changes:** Once the code is written by Claude and validated by the programmer, stage and commit the work.

```
git add .
git commit -m "Add <feature-name> via SDD workflow"
```

Why: Saves the working state of the successfully validated feature locally [26].

1. **Push to Remote:** Push the branch to GitHub.

```
git push origin feature/<feature-name>
```

Why: Uploads the isolated feature to the remote repository for review [26].

1. **Pull Request & Merge:** Create a Pull Request (PR) on GitHub and merge it into the `main` branch [26]. *Why:* This integrates the newly developed feature into the production codebase [26].
2. **Cleanup:** Delete the feature branch locally and switch back to `main`.

```
git branch -d feature/<feature-name>
git checkout main
```

Why: Maintains branch hygiene and resets the developer's environment to the updated `main` branch, ready for the next Spec-Driven cycle [26].

Chapter 8: Plan Mode & Ultraplan Mode in Claude Code

Part 1: Project Kickoff & Spec-Driven Database Setup

This section introduces the practical application of Spec-Driven Development to build the foundational database for the Expense Tracker application [1, 2].

Step-by-Step Workflow: Pre-Development Git Initialization Before writing any code or generating plans, a strict Git workflow is followed to ensure isolation of the new feature [3, 4].

1. **Start a new Claude Code Session & Rename it:** `bash claude` `markdown /rename Database Setup` *Explanation:* Initializes a fresh context window. Renaming the session immediately prevents Claude from auto-generating a vague name and helps maintain an organized session history [4].
2. **Pull the latest code from the remote repository:** `bash git pull origin main` *Explanation:* Ensures the local `main` branch is perfectly synchronized with the remote repository to prevent merge conflicts later [4].
3. **Create and checkout a new feature branch:** `bash git checkout -b feature/database-setup` *Explanation:* Isolates the database setup work from the main codebase. Standard naming convention is `feature/[feature-name]` [5].
4. **Create the Spec Document:** A highly detailed specification document is manually created (or generated via AI) and saved in a dedicated project folder (e.g., `.claude/specs/database-setup.md`). This document explicitly defines the database schema (`users` and `expenses` tables), required functions (`get_db` , `init_db` , `seed_db`), constraints (e.g., no ORMs, parameterize queries), and strict acceptance criteria [6-8].

Part 2: Introduction to Plan Mode

Plan Mode is a dedicated operational state within Claude Code designed explicitly for analyzing requirements and formulating a technical design plan before writing any actual code [9].

The "Why" Behind Plan Mode: Jumping straight into code generation for complex features often leads to hallucination, unstructured architecture, or missing dependencies. Plan Mode separates the "architecting" phase from the "coding" phase, allowing Claude to comprehensively read the existing codebase and the spec document to generate a structured implementation roadmap [9, 10].

> 💡 **IMPORTANT**
>
> > **Read-Only Constraint:** While in Plan Mode, Claude Code has **zero write permissions**. It can explore the codebase and read files, but it is strictly prohibited from modifying files or executing write operations until the plan is finalized and approved [10].

Triggering Plan Mode You can enter Plan Mode using one of two commands:

```
Shift + Tab (Press twice)
```

or

```
/plan
```

Explanation: Instantly switches the Claude Code environment into Plan Mode, notifying the AI that the upcoming prompt is for planning, not execution [10].

Step-by-Step Workflow: Executing Plan Mode 1. Enter Plan Mode using `/plan` [10]. 2. Provide a prompt directing Claude to read the spec and output a plan: `text Read the file at .claude/specs/database-setup.md and existing files. Generate an implementation plan and save it to the .claude/plans/ folder.` 3. Claude automatically spawns background subagents (like the Explore agent) to analyze the project structure, template files, and testing framework [11, 12]. 4. Claude outputs a detailed, step-by-step Technical Design Plan [12].

Part 3: Implementation, Validation, and Post-Development Workflow

Once the plan is generated, it must be implemented and validated against the original Spec Document [12].

Step-by-Step Workflow: Implementation & Git Teardown 1. **Approve the Plan:** Review the generated plan and instruct Claude to implement it by selecting the option to **Manually Approve Edits** [12]. 2. **Execute Code Changes:** Claude writes the code file-by-file (e.g., creating `db.py` and modifying `app.py`). Approve each diff (red lines removed, green lines added) [12, 13]. 3. **Manual Validation:** Run the application locally to verify the Acceptance Criteria defined in the spec document. `bash python 3 app.py` *Explanation:* Verifies the database file (`spendly.db`) is created, schemas are correct, dummy data is seeded, and parameterized queries are utilized [14, 15]. 4. **Stage and Commit:** `bash git add . git commit -m "Create database setup"` *Explanation:* Saves the validated feature state to local version control [16]. 5. **Push to Remote:** `bash git push origin feature/database-setup` *Explanation:* Pushes the isolated branch to GitHub for Pull Request creation [16]. 6. **Merge & Cleanup:** After creating and merging the PR on GitHub, synchronize the local environment: `bash git checkout main git pull origin main git branch -d feature/database-setup` *Explanation:* Switches back to `main`, pulls the newly merged feature from the remote, and deletes the local, now-redundant feature branch to maintain repository hygiene [17].

Part 4: Best Practices & Advanced Settings for Plan Mode

To maximize the quality of technical plans, Claude Code offers several configurations that adjust its internal reasoning and computational budget [18].

1. Strategic Model Selection

You should dynamically switch between LLM models depending on the phase of the Software Development Life Cycle (SDLC) [18]. * **Planning Phase (Opus):** For deep architectural decisions, complex refactoring, or multi-file reasoning, switch to the **Opus** model. It consumes more tokens but produces highly robust plans [18, 19]. * **Implementation Phase (Sonnet):** Once the plan is solid, switch back to **Sonnet**. It provides the perfect balance of speed, cost-efficiency, and reliable code generation [19].

```
/model
```

Explanation: Opens an interactive menu to switch the active underlying model (e.g., Sonnet 3.5, Opus, Haiku) for the current session [19].

2. Extended Thinking (Thinking Mode)

> 🌱 **TIP**
> > **Analogy:** Imagine being asked a highly complex algorithmic question in a technical interview. If you start talking immediately, you might stumble or code yourself into a corner. A better approach is to ask for a whiteboard, silently reason through the logic, and *then* speak the answer. Extended Thinking is Claude's "whiteboard" [20, 21].

- **What it is:** Prevents the LLM's default reflex of immediately streaming answer tokens. Instead, Claude pauses, opens an internal "scratchpad," and reasons through the problem step-by-step before generating the final response [21].
- **Why use it:** Massively improves the quality of plans for complex tasks by reducing logic errors and hallucinations [22].

```
/config
```

Explanation: Opens the configuration menu where you can toggle **Thinking Mode** to `true` or `false` [22].

3. Effort Level (Token Budgeting)

Effort Level dictates the computational budget allocated to the Extended Thinking scratchpad. Since "thinking" burns tokens, you cannot allow the model to think infinitely [23]. * **Options:** `Low`, `Medium`, `High`, `Max` (Opus only), `Auto` [24]. * **Recommendation:** Set to `Medium` or `High` for Planning Mode. Setting it to `Low` restricts Claude's reasoning depth, while `Max` gives it unlimited tokens (which can be unnecessarily expensive) [24].

```
/effort
```

Explanation: Allows the developer to explicitly set the token budgeting level for Claude's internal reasoning phase [24].

Part 5: Ultraplan Mode

For highly complex, enterprise-scale features where local Plan Mode yields unsatisfactory results, Claude offers a specialized escalation tool: **Ultraplan** [25].

What is Ultraplan? It is a remote, cloud-based alternative to Plan Mode. Instead of processing the plan locally in your terminal, it spins up an isolated, dedicated container running the **Opus 4.6** model on Claude Code for Web [26].

Why use it? It provides a superior, rich web-editing GUI for reviewing and modifying architectural plans generated by the most capable model available. It should be reserved *only* for complex features, as it is a heavy, token-expensive operation ("an overkill" for simple tasks) [26, 27].

Step-by-Step Workflow: 1. **Trigger Ultraplan:** `markdown /ultraplan Plan the registration feature for this website` *Explanation:* Pushes the prompt to the cloud environment and provides a URL to track progress [28]. 2. **Review in Web GUI:** Click the provided link to open Claude Code for Web. The Opus agent explores the codebase remotely and drafts a high-level plan [28, 29]. 3. **Teleport the Plan:** Once satisfied with the plan in the web GUI, select **"Approve plan and teleport back to terminal"** [29]. 4. **Local Implementation:** The approved plan is sent back to your local CLI session, where you can instruct your local Sonnet model to execute the code implementation [29].

Feature	Plan Mode (`/plan`)	Ultraplan Mode (`/ultraplan`)
Execution Environment	Local Terminal	Remote Cloud Container (Web GUI)
Default Model	Current Session Model (Usually Sonnet)	Opus 4.6
Cost & Token Usage	Standard	High / Expensive
Best Used For	Standard feature specs, straightforward	Complex refactoring, highly ambiguous architecture, massive

Feature	Plan Mode (`/plan`)	Ultraplan Mode (`/ultraplan`)
	architecture	features

Chapter 9: Custom Slash Commands

Part 1: Foundation of Custom Slash Commands

Custom Slash Commands are user-defined shortcuts within a Claude Code session that trigger predefined actions, prompts, or workflows without the need to write out full, lengthy prompts repeatedly.

The "Why" behind Custom Slash Commands: In standard AI-assisted coding, developers find themselves repeating the same lengthy instructions for routine tasks (e.g., code review, test generation, generating commits). Writing these repeatedly is inefficient and prone to errors. Custom Slash Commands solve this by acting as **saved prompts** that automatically invoke complex behaviors via a simple `/command_name` syntax.

Scope and Types of Custom Slash Commands

> 📝 **NOTE**
>
> > Custom Slash Commands are stored as markdown (`.md`) files. The name of the file exactly dictates the name of the slash command (e.g., `seed_user.md` becomes `/seed_user`).

Command Type	Storage Location	Accessibility / Scope	Use Case
Project-Scoped	`<project_root>/.claude/commands/`	Accessible *only* within the current project.	Project-specific tasks (e.g., populating a specific database schema).

Command Type	Storage Location	Accessibility / Scope	Use Case
User-Scoped (Global)	`~/.claude/commands/` (Home Directory)	Accessible across *all* projects on the machine.	Universal developer preferences (e.g., universal code review rules, global security scans).

Common Real-World Use Cases (Mental Models)

- `/review`: Automatically reads the newly modified file and runs a code quality review.
- `/commit`: Analyzes the uncommitted `git diff` and generates a standardized Git commit message.
- `/test`: Runs the test suite specifically for the newly generated feature.
- `/security_scan`: Scans the codebase for vulnerabilities (like exposed API keys or SQL injections).

Part 2: Building Database Seeding Commands (Step-by-Step)

To develop the UI for the Expense Tracker, the database requires realistic dummy data. Instead of manually entering data, the tutor creates two Custom Slash Commands to automate **Database Seeding**.

Workflow 1: Creating a Static Command (`/seed_user`)

This command creates a single dummy user in the database automatically.

1. **Create the Command File**: Navigate to your project directory. `bash mkdir -p .claude/commands touch .claude/commands/seed_user.md`

2. **Define the Command Specification**: Inside `seed_user.md`, define the description, tool access, and exact behavior. ```markdown # Description: Create a single dummy user in the database # Allowed Tools: Read files, Bash (Python 3 only)

Read database.py to understand the users table. Write and run a Python script using Bash that generates a realistic random Indian user. Fields to generate: Name, Email, Password (always "123" encoded), Created_At. Check if the email already exists; if so, generate a new one. Insert the user using the get_db function and print details. ``` 3. **Restart Claude Code**:

> 💡 **IMPORTANT**
>
> > Claude Code must be restarted to register newly created custom slash commands.

`text /exit claude` *Explanation*: Exits the current session and launches a new one, loading the new `.claude` configurations. 4. **Execute the Command**: `text /seed_user` *Explanation*: Claude executes the markdown instructions, writes the Python seeding script, and inserts the user into the SQLite database.

Workflow 2: Creating a Dynamic Command with Inputs (`/seed_expense`)

This command populates dummy expenses but requires dynamic user input (User ID, Number of Expenses, Time span).

1. **Create the File**: `bash touch .claude/commands/seed_expense.md`
2. **Handle User Arguments**: Use the special `$arguments` variable to capture inputs passed after the slash command. ```markdown # Description: Seed realistic dummy expenses for a specific user. (Usage: /seed_expense) # Allowed Tools: Read files, Bash (Python 3 only)

User Input: $arguments

Step 1: Extract from arguments: user_id (int), count (int), months (int). If missing, show correct format. Step 2: Verify user exists in the database. Step 3: Generate and insert expenses. Spread dates randomly across the specified months. Follow these realistic price ranges: Food (50-800), Health (100-2000). Distribute categories proportionally. `3. **Restart and Execute**:` text /exit claude ` text /seed_expense 2 5 3 ``*Explanation*: Instructs Claude to parse the` $arguments `as` user_id=2 `,` count=5 `,` months=3`. Claude generates 5 dummy expenses spread across 3 months for user ID 2.

Part 3: Automating Spec-Driven Development (`/create_spec`)

In the previous workflow, Spec Documents were created manually. To achieve true **Agentic Coding**, the tutor creates a custom command to automate the generation of Spec Documents based on feature requests.

Workflow 3: Building the `/create_spec` Command

1. **Create the File**: `bash touch .claude/commands/create_spec.md`
2. **Define the Prompt Logic**: ```markdown # Description: Create a spec file for the next Spendly feature. (Usage: /create_spec)

User Input: $arguments

Step 1: Parse arguments for Step Number, Feature Title, and Feature Slug. Step 2: Research codebase (read claude.md, app.py, database.py, and existing files in .claude/specs/). Step 3: Generate a spec document following this exact structure: Title, Overview, Dependencies, Routes, Database Changes, Templates, Rules of Implementation, Acceptance Criteria. Step 4: Save it to .claude/specs/_.md `3. **Commit the setup**:` bash git status git add . git commit -m "create spec slash command" ```

Workflow 4: Implementing the Registration Feature using `/create_spec`

1. **Create a Feature Branch**: `bash git checkout -b feature/registration` *Explanation*: Isolates feature development from the main codebase.
2. **Invoke the Auto-Spec Command**: `text /create_spec 2 Registration` *Explanation*: Claude reads the codebase and automatically generates a highly structured Spec Document (`.claude/specs/02_registration.md`).
3. **Generate an Implementation Plan (Plan Mode)**: `text /plan Read .claude/specs/02_registration.md and create a detailed implementation plan for the same. Don't write any code.` *Explanation*: Transitions Claude into Plan Mode to create technical architecture without altering files.
4. **Execute and Validate**: Accept the plan edits. Manually test the application (e.g., checking password mismatch validation, duplicate email handling, hash storage).
5. **Git Merge & Cleanup Workflow**: `bash git add . git commit -m "add registration feature" git push origin feature/registration` *Action*: Navigate to GitHub, create a Pull Request, and merge it. `bash git checkout main git pull origin main git branch -d feature/registration` *Explanation*: Switches back to the main branch, pulls the newly merged remote code, and deletes the local feature branch to maintain a clean Git tree.

Part 4: Advanced Workflow Enhancements & Login/Logout Feature

To stop repeating the manual Git branch creation steps, the tutor refines the `/create_spec` command to handle Git operations autonomously.

Upgrading `/create_spec` to handle Git

Modify `create_spec.md` to inject Git automation before generating the spec:

```
Step 1: Check if the current working directory is clean (`git status`). If uncommitted chang
Step 2: Fetch arguments. Check if the branch name already exists (append numbers if occupied
Step 3: Switch to `main` branch and pull the latest code.
Step 4: Create a new branch `feature/<slug>` and switch to it.
Step 5: Proceed with Research and Spec Document generation.
```

> 🌱 **TIP**
>
> > **Why this matters:** By automating Git checkouts inside the AI prompt, you remove friction. The AI becomes a complete co-pilot that manages version control scaffolding alongside documentation.

Workflow 5: Implementing Login & Logout (Fully Automated Start)

1. **Commit the Command Updates**: `bash git add . git commit -m "modify create_spec.md file"`
2. **Trigger the Upgraded Workflow**: `text /create_spec 3 Login and Logout` *Explanation*: Claude automatically checks Git status, pulls `main`, creates branch `feature/login_and_logout`, checks out the branch, and writes the spec document.
3. **Generate Plan**: `text /plan Read .claude/specs/03_login_and_logout.md and generate an implementation plan. Don't write any code.`
4. **Implement and Debug**: Allow Claude to write the code. Manually test the app. *Bug Discovered:* A logged-in user can still access the `/login` route. *Action:* Simply prompt Claude to fix it: "I am able to access /login and /register even when I am logged in. This should not happen." Claude applies the fix iteratively.
5. **Final Git Pipeline**: `bash git add . git commit -m "add login functionality" git push origin feature/login_and_logout` *Action*: Create

PR on GitHub, merge, and clean up local repo. `bash git checkout main git pull origin main git branch -d feature/login_and_logout`

Chapter 10: Claude Code Skills — Full Guide

Part 1: The Core Problem: General-Purpose LLMs vs. Specialized Tasks

While large language models (LLMs) like Claude, ChatGPT, or Gemini possess excellent general reasoning capabilities, they struggle to produce reliable, high-quality outputs for highly specialized or domain-specific tasks [1].

The Mental Model: The PPT Generation Analogy If you ask an LLM to generate a PowerPoint presentation, it understands the general structure of a PPT and the Python libraries needed to build one [2]. However, it does not know your specific company's design guidelines, preferred typography, layout rules, or when to use specific charts versus tables [2, 3]. It has *general* capability, but lacks the *specialized* skill required for your specific workflow [3]. This gap applies across various domains: web development (company themes), data analysis (domain-specific metrics), document writing (personal tone), and code review styles [3, 4].

Why Detailed Prompts Are Not the Solution

To solve this, developers often try writing massive, highly detailed instructions within their prompts [4]. However, this approach introduces severe workflow problems [5-7]: 1. **Repetitive & Error-Prone:** You must manually re-type or copy-paste these instructions repeatedly across different sessions [5]. 2. **Context Window Burn:** Storing a 20,000-token instruction in a system prompt eats up valuable context window space permanently, even when that specific task isn't being performed [5]. 3. **Resource Bundling Issues:** You cannot easily attach supporting resources (like reference images, templates, or external Python scripts) directly into a standard text prompt [5, 6]. 4. **Lack of Versioning:** Prompts are highly personal. They are difficult to share across a team, version-control, or iteratively improve [6]. 5. **Poor Composability:** If you combine multiple complex prompts (e.g., read a PDF, extract tables, generate a PPT) into one massive prompt, the LLM often misreads instructions, gets confused, and degrades in performance [6, 7].

Part 2: Introduction to Claude Code Skills

Skills are reusable, file-based resources that provide Claude with domain-specific expertise—such as workflows, context, and best practices—transforming a general-purpose agent into a specialist [8].

> 📝 **NOTE**
>
> > Skills act as "Just-In-Time" knowledge. Unlike system prompts that are always active, Skills are folder-based resources that load on-demand only when Claude detects they are needed [8, 9].

The Architecture of a Skill

A Skill is essentially a directory residing inside your project [9]. * `skill.md` **(Mandatory):** The core instruction file required for the skill to function [9]. * **Resource Folders (Optional):** You can include supporting folders like `/scripts/` (for Python scripts) or `/templates/` (for design guidelines or images) [10].

Anatomy of `skill.md`

The `skill.md` file is divided into two distinct sections [11, 12]: 1. **YAML Front Matter:** Located at the very top. It contains the skill's **Name** and a **Description**. * *Why this is critical:* The description acts as a trigger. Claude reads this description to understand exactly *when* to pull this skill into the context window (e.g., "Load this skill when the user asks to generate a PowerPoint") [11, 12]. 2. **Markdown Body:** Contains the exhaustive, step-by-step instructions, coding patterns, validation rules, and file path links to external resources (like the scripts in your `/scripts/` folder) [12, 13].

Progressive Disclosure (Memory Management)

To protect the context window, Skills operate on a principle called **Progressive Disclosure** ("don't present information until the moment it's needed") [13]. * **Level 1 (Session Start):** Claude only loads the YAML Front Matter (Name and Description) into the context window for all available skills [14]. * **Level 2 (Trigger):** When the user makes a request (e.g., "Create a PPT"), Claude matches the request to the skill's description and loads the Markdown Body on demand [14, 15]. * **Level 3 (Resource Fetching):** If the Markdown Body instructs Claude to use a specific Python script or reference template, those external resources are fetched and loaded into context *only* at that exact execution step [15].

Part 3: Types of Skills & Creation Methods

Types of Skills (Based on Scope)

Skill Type	Storage Location	Scope & Applicability
Personal Skills	`~/.claude/skills/` (Home Directory)	Applies globally across **all** your projects. Ideal for personal coding styles, writing tones, or generic preferences [16].
Project-Based Skills	`.claude/skills/` (Project Root)	Scoped strictly to the current project. Can be committed to Git, version-controlled, and shared with teammates [16, 17].

Methods to Create a Skill

1. **Manual Creation:** Manually creating the folder and `skill.md`. Not recommended for beginners due to the learning curve of exact formatting [17].
2. **Using Claude's "Skill Creator":** The most recommended approach. Claude has a built-in skill (the "Skill Creator") specifically designed to help you generate other skills [17, 18].
3. **Community Marketplaces:** Downloading pre-made skills from places like Anthropic's public repository or third-party marketplaces (e.g., Skills MP) [18, 19].

> ⚠ **WARNING**
>
> > Exercise caution with third-party community skills. If you do not read the `skill.md` carefully, malicious skills could potentially expose your project's API keys or introduce security flaws [19].

Part 4: Workflow for Creating & Implementing a Skill

Step-by-Step Methodology: Building a Skill via Claude 1. **Identify the Need:** Ensure the task is a highly specialized, repetitive workflow (e.g., Frontend Design for Profile

Pages) [19-21]. 2. **Invoke the Skill Creator:** Open Claude Code, navigate to Skills, and launch the Skill Creator [18, 22]. 3. **Define Parameters:** Answer Claude's prompt questions. Define what the skill does, the exact trigger conditions, and what a successful output looks like [22, 23]. 4. **Generate & Save:** Claude will generate the `skill.md` code. Copy this code, create a new folder (e.g., `.claude/skills/frontend_design/`), and save it as `skill.md` inside that folder [24]. 5. **Restart Claude:** You must exit and restart the Claude session for the new skill to be registered into Level 1 memory [25]. 6. **Test, Evaluate, and Iterate:** Run the skill. Skills are rarely perfect on the first try. You will likely need 4-5 iterative improvements to the `skill.md` instructions before it becomes highly reliable [20, 26].

Part 5: Practical Implementation & Workflow Commands

During the video, the tutor demonstrates building a Profile Page UI first *without* a skill (yielding a generic, unstyled page) and then *with* the newly created "Frontend Design" skill to show the architectural improvement [27-29].

To reset the project state and try again with the skill, a crucial slash command is utilized:

```
/rewind
```

- **What it does:** Allows you to revert your code and conversation history backward in time to a specific state [24, 30].
- **Why/When to use it:** Use it when an AI code generation goes wrong, or when you want to erase a specific implementation (like a poorly designed UI) and attempt it again from an earlier branching point. It prompts you to select whether you want to restore *only* the conversation or *both* the code and the conversation [30].

After saving the new skill, Claude must be restarted to recognize it:

```
claude -r
```

- **What it does:** Resumes a specific past Claude Code conversation session [25].
- **Why/When to use it:** Because new skills require a Claude restart to be parsed, you exit the terminal, and use `claude -r` to launch the menu, allowing you to jump right back into the "Profile Page Design" session with your new skill successfully loaded into the progressive disclosure architecture [25].

Part 6: The Convergence of Skills and Slash Commands

> 💡 **IMPORTANT**
>
> > **Architectural Update:** Anthropic has officially merged the concepts of Custom Slash Commands and Skills [31, 32].

Historically, Custom Slash Commands (triggered manually by the user) and Skills (triggered automatically by the model) were separate features [31, 32]. Going forward: * The `commands/` folder architecture is deprecated. Everything will reside in the `skills/` folder [33, 34]. * Every Skill automatically acts as a slash command. If your skill is named `frontend_design`, you can manually invoke it by typing `/frontend_design` in the chat interface [33]. * Both use the exact same file structure (YAML Front Matter + Markdown Body) [32].

Disabling Auto-Invocation (Creating Pure Commands)

If you want to create a workflow that Claude *never* auto-invokes (meaning it strictly acts as a manual Custom Slash Command), you must explicitly block the model's ability to trigger it [33, 34].

```
disable_model_invocation: true
```

- **What it does:** A flag placed directly inside the YAML Front Matter of the `skill.md` file [33].
- **Why/When to use it:** Use this when building sensitive or highly specific workflows (like deploying to production, running specific destructive tests, or generating commit messages) where you only want the workflow to run when you manually type the `/skill_name` command, preventing Claude from accidentally running it autonomously during an Agentic loop [33, 34].

Chapter 11: Claude SubAgents — Solve Context & Token Cost Problems

Part 1: The Core Problem: LLM Statelessness and Context Window Overload

To understand why SubAgents are necessary, one must first understand the fundamental limitation of Large Language Models (LLMs): **Statelessness** [1].

Because LLMs do not inherently possess memory, they cannot remember past interactions [1]. If you ask, "What is the capital of France?" and then follow up with, "What about Germany?", the model will fail to understand the context of the second question unless the entire conversation history is sent with every single API call [1, 2].

While this brute-force method of passing history works for simple chat applications, it catastrophically fails in **Agentic Coding** due to the sheer size of codebases [3, 4].

The Context Bloat Scenario: If you load a 30,000-token codebase into Claude to build an authentication system, here is what happens without SubAgents: 1. **Turn 1:** You send the 30,000-token codebase + your prompt. Claude returns a 2,000-token plan [4, 5]. 2. **Turn 2:** You ask Claude to implement a JWT middleware. You must now send the original 30,000 tokens + the 2,000-token plan + the new prompt [5]. You are now consuming 32,000+ tokens for a single turn [5]. 3. **Turn 3:** You ask Claude to add rate limiting. You send the 30,000 tokens + the past conversation + the newly generated code. You are now passing 39,000 tokens [6]. 4. **Turn 8:** By the eighth interaction, your single request is passing over **76,000 tokens**, costing more than $1 per turn [7].

> 💡 **IMPORTANT**
>
> > **The Two Major Consequences of Context Bloat:** > 1. **Context Window Overflow:** The context window fills up incredibly fast, unnecessarily burning tokens and money on every turn [7, 8]. > 2. **"Lost in the Middle" Effect:** As the context window becomes saturated, LLMs tend to focus heavily on the very beginning and the very end of the prompt, completely forgetting or hallucinating the instructions hidden in the middle of the context [8].

Part 2: What Are SubAgents?

SubAgents are specialized AI assistants spawned by the main Claude agent that run in their own completely isolated, fresh context windows [8]. They perform heavy analytical or generative lifting in a separate space and only return the distilled, relevant output back to the main agent [8].

The Mental Model: SubAgents as Programming Functions

> The tutor uses a powerful analogy: SubAgents operate exactly like **Functions** in traditional programming [9]. As a programmer, you pass inputs to a function, the function processes the data independently, and it returns an output. You do not need to keep the function's internal execution steps in your main memory. SubAgents do exactly this for the LLM's context window [9].

How SubAgents Solve the Context Problem: Instead of dragging a 30,000-token codebase through every conversation turn, the workflow becomes: 1. The Main Agent spawns a **Planning SubAgent** [10]. 2. The SubAgent reads the 30,000-token codebase, analyzes it, and generates a 500-token implementation plan [11]. 3. The SubAgent passes *only* the 500-token plan back to the Main Agent [11]. 4. The SubAgent's heavy context window is destroyed [12]. 5. The Main Agent continues the conversation using only the 500-token plan, saving roughly 28,000 tokens per subsequent turn [9, 12].

Part 3: Core Advantages & Top Industry Use Cases

By utilizing SubAgents, developers unlock architectures that are impossible with a single-agent setup:

Advantage	Explanation
Context Isolation	Heavy tasks happen in disposable context windows, keeping the main session lightweight, cheap, and focused [13].
Specialization	Each SubAgent can be tailored. You can give a specific agent access to Web Search, while restricting another agent strictly to Read-Only file access for safety [13, 14].
Modularity	The software lifecycle is split. One agent writes code, another tests it, and a third reviews it [14].
Parallelism	Multiple SubAgents can run simultaneously. If you need to perform Exploratory Data Analysis (EDA) on three separate datasets, you can spawn three EDA SubAgents in parallel [15].

Top Industry Use Cases for SubAgents: 1. **Codebase Exploration:** Summarizing large repositories without polluting the main context window [16]. 2. **Code Review:** The agent

that writes the code inherently has bias and blind spots regarding its own logic. A separate, fresh SubAgent provides a superior, unbiased code review [17, 18]. 3. **Testing:** Similar to reviews, isolating the test-generation and test-execution to distinct SubAgents ensures higher code quality [18]. 4. **Multi-Stage Pipelines:** Stringing together sequential hand-offs (e.g., Agent 1 writes API contracts -> Agent 2 builds the backend -> Agent 3 writes the tests) [18, 19]. 5. **Parallel Independent Tasks:** Building an Auth service, Payment service, and User service simultaneously in separate files [19]. 6. **Security Auditing:** A dedicated agent acting as a red-team hacker to check for SQL injections or exposed API keys [19].

Part 4: Types of SubAgents

Claude Code categorizes SubAgents into two primary buckets: **Built-in** and **Custom**.

1. Built-in SubAgents

These come pre-packaged with Claude Code and handle fundamental operational tasks [20]. * **Explore SubAgent:** Triggered when the user asks Claude to summarize or explore the codebase [21]. * **Plan SubAgent:** Triggered during Plan Mode to read specs and generate technical implementation plans [21]. * **General Purpose SubAgent:** Handed generic read/write tasks when the main agent decides to delegate work [21].

Trigger Mechanisms: * **Implicitly:** Claude reads your prompt and autonomously decides a SubAgent is the most efficient way to handle the task [22]. * **Explicitly:** The programmer explicitly instructs Claude to spin up a SubAgent for a task [22].

2. Custom SubAgents

Custom SubAgents are tailored by the user to execute highly specialized workflows (e.g., a "Security Reviewer" or "Research Agent") [23]. * **Project-Level:** Saved within the project's `.claude/agents` directory and shared with the team [23]. * **User-Level (Global):** Saved in the machine's root home directory, accessible across all local projects [23]. * **Configuration Parameters:** When defining a Custom SubAgent, the developer explicitly configures its Tools (Read, Write, Bash), System Prompt, LLM Model (Opus, Sonnet, Haiku), Hooks, and Skills [24, 25].

Part 5: Practical Workflow - Implementing & Observing SubAgents

To view SubAgents in action (which run invisibly in the background by default), the tutor introduces an open-source observability dashboard [26].

```
observe start
```

- **What it does:** Starts the `agents-observe` library server, providing a local web dashboard that visually tracks every SubAgent spawned, its tool calls, and its execution state in real-time [27].
- **When to use:** Used during development to debug agent loops and understand how Claude is delegating tasks [27].

Workflow 1: Codebase Exploration (Implicit SubAgent)

```
/context
```

- **What it does:** Displays the current available token space in the Context Window [28].
- **Why to use:** Crucial to run before and after heavy commands to verify that SubAgents successfully prevented context bloat [28, 29].

Step 1: The user provides an exploratory prompt: "*Explore the codebase and tell me what this is all about...* " [28]. **Step 2:** On the `agents-observe` dashboard, a `Claude Explore` SubAgent instantly spawns [29]. **Step 3:** The SubAgent utilizes Read Tools to analyze files [29]. **Step 4:** The SubAgent stops, passes the summary to the Main Agent, and its context is destroyed [29]. Running `/context` again confirms the main context window remains largely empty [29].

Workflow 2: Parallel Development via Explicit Prompting

In this workflow, the tutor demonstrates forcing Claude to use multiple SubAgents in parallel to build features on a profile page [30, 31].

```
/create spec 05 backend routes for profile page
```

- **What it does:** A custom slash command (created in previous videos) that spins up a new git branch and generates a Spec Document [30].

> ⚠ **WARNING**

> **Architectural Anti-Pattern:** The tutor explicitly notes that commanding three SubAgents to write code in the *exact same file* simultaneously is a bad idea due to merge conflicts [31]. Parallel SubAgents should ideally be assigned to completely independent files or services [32, 33].

Step 1 (Prompting Plan Mode): The user enters Plan Mode and explicitly requests parallel delegation: "*Read this file... come up with an implementation plan. While implementing the plan into code split the work across three parallel sub agents. Sub agent 1's job is X, Sub agent 2's job is Y...*" [30, 31].

Step 2 (The Orchestration Loop): 1. **Exploration Phase:** Two `Explore` SubAgents spawn in parallel (one analyzes the template, one analyzes test structures) to gather context without bloating the main window [32, 34]. 2. **Planning Phase:** A `Plan` SubAgent spawns to synthesize the exploration data into a technical implementation plan [33, 34]. 3. **Execution Phase:** Upon plan approval, three `General Purpose` SubAgents spawn simultaneously [33]. SubAgent 1, SubAgent 2, and SubAgent 3 write their respective code blocks in parallel [33]. 4. **Integration Phase:** Once all three SubAgents stop, the Main Agent compiles their outputs and resolves any conflicts to finalize the file [33, 35].

Workflow 3: Data Seeding and Git Version Control

To test the newly built profile page features, the database must be seeded with dummy data [35].

```
/seed expense user id 3 count 3 months 6
```

- **What it does:** A custom slash command (built in previous chapters) that connects to the SQLite database and populates dummy expenses for a specific user ID over a specified timeframe [35].
- **When to use:** Used immediately after database/route creation to visually validate frontend features [35].

Once the feature is validated, standard Git commands are used to commit and push the SubAgent-generated code:

```
git add .
git commit -m "add dynamic routes to profile page"
git push origin feature/backend-routes-profile-page
```

- **What it does:** Stages, commits, and pushes the local feature branch to the remote GitHub repository [36].

```
git checkout main
git pull origin main
git branch -d feature/backend-routes-profile-page
```

- **What it does:** Switches back to the main branch, pulls the newly merged PR code from remote, and deletes the local disposable feature branch to keep the workspace clean [37].

Chapter 12: Custom Subagents (Build Your Own AI Workers)

Part 1: Introduction to Custom Subagents & The "Why"

Custom Subagents are specialized, user-defined AI workers built within Claude Code. While Claude possesses Built-in Subagents (like Explore, Plan, and General Purpose), custom subagents solve the problem of **specialization and strict adherence to specific guidelines.**

The Security Audit Analogy: Imagine you want to run a security audit on your codebase. You could use a built-in subagent, which would scan for general vulnerabilities. However, if your company has a highly specific security checklist or compliance standard, the generic agent will fail to follow it perfectly. By building a **Custom Subagent**, you can create a "tailor-made" worker equipped with: * A custom system prompt strictly enforcing your company's checklist. * Access to only the specific tools it needs (e.g., read-only access to prevent accidental code deletion). * Specific skills and context tailored to your codebase.

> 💡 **IMPORTANT**
>
> > **Core Rule of Custom Subagents:** Whenever a task requires domain specialization, strict adherence to custom rules, or specific tool limitations, you should build a Custom Subagent rather than relying on generic built-in agents.

Part 2: Architecture & Triggering Mechanisms of Custom Subagents

Custom subagents are structurally very simple: they are just markdown files.

File Structure: To create a project-scoped subagent, you create a markdown file inside the `.claude/agents/` directory of your project root. The file consists of two main parts: 1. **YAML Front Matter:** Located at the very top. It configures the agent's identity and permissions. * **Name:** The identifier for the agent. * **Description:** A detailed explanation of what the agent does. * **Tools:** Which specific tools the agent can use (Read, Write, Bash, etc.). * **Model:** Which LLM powers it (e.g., Opus, Sonnet). * **Color:** The UI color displayed in the terminal when the agent is active. * **Memory / Skills / Hooks:** Access permissions for other Claude features. 2. **Main Body:** Contains the system prompt and step-by-step instructions on how the agent should carry out its specific task.

How Subagents are Triggered: * **Implicitly (Automatic):** Claude's main agent reads the descriptions of all available subagents. If a user's prompt matches a subagent's description, Claude automatically deploys that subagent. * **Explicitly (Manual):** The programmer triggers the subagent manually, usually by wrapping it inside a Custom Slash Command (e.g., `/test-feature`), explicitly orchestrating the workflow.

Part 3: Architecting an Agentic Workflow (Sequential vs. Parallel)

To demonstrate the power of custom subagents, the tutor introduces a new, robust software development pipeline before pushing any code to Git. This involves creating two new stages powered by custom subagents:

1. The Testing Stage (Sequential Execution) A workflow that requires tests to be written *before* they are run. * **Subagent A (Test Writer):** Reads the specification document and writes PyTest cases. * **Subagent B (Test Runner):** Waits for Subagent A to finish, then executes the tests and generates a report.

2. The Code Review Stage (Parallel Execution) A workflow where multiple independent reviews happen simultaneously to save time and context. * **Subagent C (Security Reviewer):** Scans the new code strictly for security vulnerabilities (SQL injections, exposed keys). * **Subagent D (Code Quality Reviewer):** Analyzes the code for best practices and formatting.

> 🌱 **TIP**

> **Why separate the agent that writes the code from the agent that tests/reviews it? ** > An AI agent that writes a feature inherently develops a "bias" toward its own code, knowing exactly what workarounds it used. A fresh, isolated subagent with an independent context window will review the code objectively, just like a real-world Senior Developer would.

Part 4: Building the Testing Subagents (Sequential)

Step 1: Create the Test Writer Subagent

The tutor uses Claude's built-in creation tool to generate the first agent.

```
/agents
```

- **What it does:** Opens the agent management UI in Claude Code.
- **Why/When to use:** Used to view active agents, access the agent library, or generate a new custom agent file interactively.

Configuration choices made during creation: * **Scope:** Project-level. * **Method:** "Create using Claude". * **Description provided:** "Use this agent to write pytest test cases for Spendly features. Invoke after implementing any feature to generate tests based on feature specs, not the implementation." * **Tools Allowed:** Read-only tools + Edit tools (needs to write the test files). * **Model:** Sonnet.

> 💡 **IMPORTANT**
>
> **Testing Best Practice:** The subagent is explicitly instructed to write test cases based on the **Spec Document**, NOT the generated code. If it reads the generated code to write tests, it will subconsciously write tests that pass the code's bugs.

Step 2: Create the Test Runner Subagent

Instead of using Claude, this agent is created manually by placing a markdown file directly into the directory. * **Filepath:** `.claude/agents/spendly_test_runner.md` * **Tools Allowed:** Read tools + Bash tools (to execute `pytest`). *Write access is intentionally restricted so it cannot alter the code.* * **Workflow:** Verifies tests exist -> Runs them via Bash -> Generates a structured Pass/Fail report.

Step 3: Orchestrate via Custom Slash Command

To link these two agents, a custom slash command is created at `.claude/commands/test-feature.md`.

```
/test-feature [spec_name]
```

- **What it does:** Triggers the sequential testing pipeline.
- **Why/When to use:** Used immediately after a feature's code is written. It tells Claude to first invoke the Test Writer agent, wait for it to finish, and then invoke the Test Runner agent.

Part 5: Building the Code Review Subagents (Parallel)

Step 1: Create the Review Agents

Two new markdown files are placed in `.claude/agents/`: 1. `spendly_security_reviewer.md` 2. `spendly_quality_reviewer.md`

Step 2: Orchestrate via Custom Slash Command

A custom slash command is created at `.claude/commands/code-review-feature.md`.

```
/code-review-feature [spec_name]
```

- **What it does:** Triggers the parallel code review pipeline.
- **Why/When to use:** Run after the testing phase passes. The markdown file specifically instructs Claude to launch **both** the Security and Quality subagents simultaneously. Once both finish, the main agent unifies their findings into a single report and asks the user for permission to implement the fixes.

Part 6: End-to-End Practical Execution

The tutor demonstrates building a "Date Filter" feature using the newly created agentic workflow.

1. Setup & Pre-requisites

```
git add .
git commit -m "add sub agents"
```

- **What it does:** Stages and commits the newly created subagent and command markdown files.
- **Why/When to use:** Ensures the workspace is clean before spinning up a new feature branch.

2. Spec & Code Generation

```
/create-spec 06 "Date Filter for Profile Page"
```

- **What it does:** Custom command (built in previous videos) that automatically creates a new Git branch, explores the codebase, and generates a Product Spec Document.

(The user then enters Plan Mode to generate the technical plan based on the spec, and implements the code).

3. Execution of the Testing Pipeline

```
/test-feature 06-date-filter-profile
```

- **What it does:** Triggers the custom testing subagents.
- **Workflow Observed:**
  1. Main agent spins up **Spendly Test Writer**.
  2. Test Writer reads the spec and writes `tests/test_06_date_filter.py`.
  3. Test Writer stops.
  4. Main agent spins up **Spendly Test Runner**.
  5. Test Runner executes the tests in bash and outputs the summary (e.g., "76 cases run, 73 passed, 3 failed").

4. Execution of the Code Review Pipeline

```
/code-review-feature 06-date-filter-profile
```

- **What it does:** Triggers the parallel review subagents.
- **Workflow Observed:**
  1. Main agent launches Security Reviewer and Quality Reviewer side-by-side.

2. Both analyze the newly written feature code.
3. Security agent identifies a vulnerability: *An f-string was used for a SQL query instead of a parameterized query.*
4. Main agent compiles the report and asks: "Do you want me to implement the action plan now?"
5. User approves, and Claude automatically patches the security vulnerability.

5. Committing and Pushing the Feature

```
git add .
git commit -m "add date filter to profile page"
```

- **What it does:** Stages and commits the completed, tested, and reviewed feature.

```
git branch
```

- **What it does:** Lists current branches to verify the exact name of the active feature branch.

```
git push origin feature/date-filter-profile
```

- **What it does:** Pushes the local feature branch to the remote GitHub repository.

(The tutor then manually creates a Pull Request on GitHub, merges it, and deletes the remote branch).

6. Cleaning Up the Local Environment

```
git checkout main
```

- **What it does:** Switches the local repository back to the main branch.

```
git pull origin main
```

- **What it does:** Fetches the newly merged code from the remote main branch down to the local machine.

```
git branch -d feature/date-filter-profile
```

- **What it does:** Deletes the local feature branch now that the code is safely merged into main, keeping the local workspace clean.

Chapter 13: Claude + MCP Explained

Part 1: Model Context Protocol (MCP) Foundations

Concept Overview: The **Model Context Protocol (MCP)** is an open standard created by Anthropic roughly 1.5 years ago. It acts as a **Universal Connector** between Large Language Models (LLMs) like Claude and external tools, services, and data sources.

The "Why" Behind MCP: Before MCP, integrating an LLM with external tools required writing custom, non-standardized API connection code. If the service provider (e.g., GitHub, Jira) updated their API, your connection code would break, creating severe maintenance overhead. MCP standardizes this connection.

Mental Model: Think of standard Claude Code as having only three basic "senses" (tools): 1. **Read Tool** (Reading file systems) 2. **Write Tool** (Creating/editing files) 3. **Bash Tool** (Executing command-line instructions)

MCP acts as "sensory upgrades." Without MCP, Claude cannot independently fetch a Google Drive document, read pending Jira tickets, or summarize active Slack conversations. By adding MCP servers, you equip Claude with specialized senses to pull real-time context from your entire development ecosystem without manual copy-pasting.

Part 2: Integrating the Database MCP Server

Concept Overview: Connecting your local or remote database directly to Claude allows you to query schema, relationships, and data using natural language instead of writing manual Python or SQL scripts.

Architectural Decision & Gotcha: The tutor initially attempted to use **DB Hub** (a zero-dependency, token-efficient MCP server capable of handling multiple databases simultaneously). However, DB Hub failed to connect to the project's local **SQLite** database (`spendly.db`). *Alternative chosen:* The tutor switched to the standard **SQLite MCP Server** utilizing the `stdio` transport mechanism (ideal for local setups).

Workflow: Database Querying via MCP 1. Initialize the SQLite MCP server in the terminal, pointing it to the local `spendly.db` path. 2. Restart the Claude Code session to ensure the server loads into the context. 3. Verify connection using the `/mcp` command. 4. Query the database using natural English.

```
/mcp
```

- **What it does:** Opens the Claude Code MCP interface to list all configured MCP servers and their current connection status (e.g., "Connected" or "Failed").
- **Why/When to use it:** Use this to verify that an external service is actively talking to your Claude instance or to debug connection failures.

Example Natural Language Queries Executed: * "*List all the tables in spendly database.*" * "*Can you tell me the schema of expenses table?*" * "*Show total spending grouped by category.*"

> 🌱 **TIP**
>
> > **Why this matters for scaling:** While querying a 2-table database manually is easy, production environments often have 15-20+ interconnected tables. An MCP Database server allows you to instantly understand complex table relationships on the fly without breaking your coding flow.

Part 3: Automating UI/UX with Figma MCP Server

Concept Overview: Typically, a UI/UX team designs wireframes in **Figma**, hands them off to developers, and developers manually translate visual designs into CSS/HTML. The **Figma MCP Server** automates this translation layer.

Workflow: Spec-Driven UI Development via Figma MCP 1. **Design Extraction:** The designer creates a UI component in Figma (e.g., an "Analytics Coming Soon" page). The developer right-clicks the design in Figma and copies the URL. 2. **MCP Authorization:** The developer installs the Figma MCP Plugin via the terminal. In the Claude session, they authenticate their Figma account via browser. 3. **Prompt Execution:** The developer provides the Figma link alongside a specialized prompt. 4. **Automated Implementation:** Claude uses the Figma MCP to read the design elements (fonts, spacing, layout), explores the existing codebase's CSS, and generates the exact replica in code.

```
Read the Figma design and convert it to a Jinja 2 HTML template. Add an analytics menu item
```

- **What it does:** Instructs Claude to use the Figma MCP tool to read the design URL, while simultaneously acting as a backend developer (Flask routing) and front-end developer (Jinja2/CSS).
- **Why/When to use it:** To achieve pixel-perfect UI replication directly from design tools without writing boilerplate HTML/CSS.

Part 4: Agentic GitHub Workflows with GitHub MCP Server

Concept Overview: The **GitHub MCP Server** allows Claude to interact directly with your repositories, issues, and pull requests.

Setup Requirements: You must generate a **Personal Access Token (PAT)** from GitHub (`Developer Settings` -> `Fine-grained tokens`).

> 💡 **IMPORTANT**
>
> > **Permissions Gotcha:** The token *must* explicitly include **Read/Write access for Pull Requests, Issues, and Code**. In the video, the tutor forgot to grant PR Write access, causing the automated MCP PR-creation workflow to fail, forcing a manual fallback.

Workflow: Fully Automated Feature Delivery (Attempted) The goal was to automate the entire post-development lifecycle (Commit -> Push -> PR -> Merge -> Cleanup) using the GitHub MCP.

1. **Development & Review:** Feature is coded, then validated via subagents (Test Runner, Security Audit).
2. **Agentic GitHub Prompt:**

```
Commit all changes with an appropriate conventional commit message. Push to the current feat
```

- **What it does:** Orchestrates a 7-step Git and GitHub workflow autonomously.
- **Why/When to use it:** Once a feature passes local testing, this single prompt delegates the entire CI/CD and version control handoff to Claude.

Manual Fallback Commands (Due to Token Error): Because the MCP token lacked PR write permissions, the following manual Git workflow was executed to recover:

```
git add .
git commit -m "add coming soon page"
```

- **What it does:** Stages and commits the uncommitted work preventing branch creation.

```
git checkout main
git branch -d feature-add-expense
git pull origin main
```

- **What it does:** Switches back to the primary branch, deletes the local feature branch after manual GitHub merging, and syncs the local repository with the remote.

Part 5: Top 7 Highly Recommended MCP Servers

The tutor outlined seven additional industry-standard MCP servers to enhance agentic coding:

MCP Server	Primary Use Case & "Why" It's Necessary
Context7	Pulls live, up-to-date documentation for libraries/frameworks. **Why:** Bypasses the LLM's knowledge cutoff date, preventing hallucinations regarding deprecated library syntax.
Jira	Project management integration. **Why:** Allows Claude to autonomously read assigned bug tickets, fetch their context, and write the fix without the developer manually opening Jira.
Notion	Workspace knowledge base integration. **Why:** Claude can read Product Requirement Docs (PRDs) or API design guidelines directly from Notion to scaffold compliant code.
Slack	Team communication integration. **Why:** Automates status updates (e.g., posting PR links to a `#code-reviews` channel) or allows Claude

MCP Server	Primary Use Case & "Why" It's Necessary
	to read the `#incidents` channel to actively debug live production errors.
AWS	Cloud infrastructure management. **Why:** Empowers Claude to deploy builds to EC2 instances or fetch and analyze CloudWatch logs for 500-errors directly in the IDE.
Docker	Containerization management. **Why:** Claude can autonomously read an application, generate an optimized Dockerfile, or analyze an existing 2GB image and reduce its size.

Part 6: Managing and Optimizing MCP Servers

Architectural Warning regarding Context Windows:

> 💡 **IMPORTANT**
>
> > **Context Window Bloat:** Do **not** blindly install every MCP server available. Every installed MCP server automatically injects its tool descriptions and schemas into Claude's context window at the start of *every* session. If you have too many servers, you will fill your context window with unnecessary tokens, severely degrading the model's performance and reasoning capabilities.

Workflow: Removing Unnecessary MCP Servers To maintain a lean context window, you must remove MCP servers you are not actively using.

```
claude mcp remove <server_name>
```

- *Example:* `claude mcp remove sqlite`
- **What it does:** Detaches and uninstalls the specified MCP server from your Claude Code configuration.
- **Why/When to use it:** Use this to free up context window tokens and improve model performance by removing idle integrations.

Viewing MCP Tool Capabilities: To understand exactly what a specific MCP server can do, navigate to `/mcp`, highlight the server, and select **View Tools**. This provides a full description of the exact capabilities that server grants to the LLM.

Chapter 14: Hooks in Claude Code — Full Theory + Practical Use

Part 1: System Design Perspective of Claude Code (The "Why")

To understand Hooks, one must first understand how Claude Code operates at a system design level. Most users view Claude Code simply as a terminal-based AI coding agent. However, from an engineering perspective, Claude Code is defined as a **Coding Harness**.

What is a Coding Harness? A piece of software used to convert a raw Large Language Model (LLM) into a reliable software engineering system [1].

The tutor uses two powerful mental models to explain this: 1. **The Horse and Harness Analogy:** An LLM is like a horse—it possesses immense raw power, speed, and intelligence. However, a horse is also unpredictable and cannot pull a carriage safely without a structured system. The **Harness** (straps and equipment) controls and directs that raw power safely. Similarly, a Coding Harness controls the raw, unpredictable power of the LLM [1-3]. 2. **The Mind, Brain, and Body Analogy:** - **LLM = Brain/Mind:** Responsible for generating thoughts, reasoning, and deciding the next action (e.g., "I need to read `app.py`") [4]. - **Coding Harness = Body/Nervous System:** Executes the physical actions. It takes the LLM's thought, opens the file system, copies the content, and feeds it back to the LLM [4].

What does the Coding Harness handle? - Reading file systems to provide context. - Managing context window usage and conversation history. - Handling API requests to Anthropic's servers. - Spawning parallel subagents. - Safety modules, memory management, and executing terminal outputs [5-7].

> 📝 **NOTE**
>
> > **Harness Engineering** is emerging as a distinct field. Examples of other harnesses include **Open Devin/Open Clau** (Personal Agent Harnesses for long-

> running tasks), **Hermes Agent** (Self-learning personal harness), and **Pie** (Lightweight coding harness) [4, 8, 9].

Part 2: The Core Problem (Boss-Employee Relationship)

The need for Hooks arises from the relationship between the LLM and the Coding Harness, which mirrors a **Boss-Employee Relationship** [9]. - **The LLM (Boss) is Probabilistic:** It is moody, non-deterministic, and prone to hallucination. It might randomly decide to delete critical files or overwrite `.env` API keys [10]. - **The Coding Harness (Employee) is Deterministic:** It is faithfully obedient. If the LLM issues a command to delete the database, the Harness will blindly execute it [11].

> 💡 **IMPORTANT**
>
> > **Why not just write safety rules in `claude.md`?** > System prompts and `claude.md` instructions are followed ~98% of the time. However, during complex refactoring or when the context window is heavily loaded, the LLM might override instructions and hallucinate dangerous commands (the 2% failure rate). Hooks are required to enforce **100% deterministic safety** [12, 13].

Part 3: Foundational Concepts — Agent Loops & Session Lifecycles

To understand how Hooks intercept commands, you must understand the execution flow.

1. The Agent Loop [13-16] An Agent Loop is the multi-step process the Harness executes to complete a single prompt. 1. **Input:** User prompts the AI (e.g., "Add a `/delete` endpoint"). 2. **Execution:** LLM requests to read `app.py` -> Harness executes -> LLM requests to read schema -> Harness executes -> LLM writes code -> Harness pastes code -> LLM runs tests. 3. **Termination:** LLM observes tests passed and returns "Done."

2. The Session Lifecycle [16-19] The Session Lifecycle wraps the Agent Loop. It is the full lifespan of a session from typing `claude` to typing `/exit`. The Coding Harness defines specific **Events** throughout this lifecycle. **Key Lifecycle Events:** - `Session Start` - `User Submits Prompt` - `Pre Tool Use` (Right before a tool is executed) - `Post Tool Use` (Right after a tool is executed) - `Sub Agent Start` / `Sub Agent Stop` - `Stop` / `Session End`

Part 4: What Are Hooks?

Hooks are custom scripts written by the programmer that the Coding Harness automatically executes at specific events during a session's lifecycle [20, 21].

They intercept the probabilistic LLM behavior and inject deterministic programmer control. If the LLM tries to run `rm spendly.db`, a **Pre Tool Use** hook can intercept the bash command, realize it targets a sensitive file, and abort the operation before the Harness executes it [21-23].

Part 5: Top Practical Use Cases for Hooks

Use Case	Lifecycle Event	Description & Benefits
Auto-Formatting	`Post Tool Use`	Running formatters like `black` automatically after Claude writes code to ensure consistent styling across multiple sessions [24-26].
Linting	`Post Tool Use`	Catching unused imports, undefined variables, or bad patterns (e.g., broad `except:` clauses) without running the code [27, 28].
Blocking Dangerous Commands	`Pre Tool Use`	Preventing `rm` commands on critical files (e.g., production databases) [28].
Protecting Sensitive Files	`Pre Tool Use`	Preventing the AI from reading or modifying `.env` or credential files [29].
Push Notifications	`Stop`	Triggering a push notification (e.g., via Notify.sh) to your phone when a long-running refactor is complete [29].
Telemetry Dashboards	`Sub Agent Start/Stop`	Pinging an external dashboard to monitor which subagents are running in real-time [30].

Use Case	Lifecycle Event	Description & Benefits
Custom Workflows	`Session Start`	Automatically running a script to generate a project summary of completed/pending features every time you start Claude [31].

Part 6: How to Implement Hooks (Methodology)

Hooks are configured inside your project's `.claude/settings.json` file [32, 33].

Step 1: Define the Hook Structure

A hook requires three configuration components: 1. **Event:** When it triggers (e.g., `preToolUse`). 2. **Matcher (Filter):** Narrows down the trigger condition so the script doesn't run unnecessarily. (e.g., Only trigger on `bash` or `write` tools, not `read` tools). 3. **Action:** The script/command the Harness must execute.

Step 2: Write the Hook Script Logic

When a hook triggers (e.g., on a Bash command), the Harness sends a JSON object to the script via standard input (`stdin`). The script must: 1. Parse the JSON to extract the `"tool_input"` . 2. Check the command for forbidden actions (e.g., targeting `spendly.db`). 3. Output an **Exit Code** back to the Harness: - **`Exit 0`** : Proceed. Operation is safe. - **`Exit 2`** : Abort. Stop the operation and notify the LLM the command was blocked [34-36].

Part 7: Practical Project Workflow — "Edit Expense" & Custom Commands

The tutor demonstrates applying hooks and automating the Spec-Driven Development workflow by building an "Edit Expense" feature.

Step 1: Session Initiation

```
/rename Edit Expense
```

Explanation: Renames the active Claude Code session to maintain clean conversation history and context organization [37].

Step 2: Fixing GitHub MCP Token Permissions

To automate PR creation and merging natively from Claude Code, the GitHub MCP Token requires fine-grained permissions [38, 39]: - **Repository Access:** Only selected repositories (e.g., `spendly`). - **Permissions:** Read & Write access for *Pull Requests*, *Issues*, *Contents*, and *Commit Statuses*.

Step 3: The `/ship-feature` Custom Slash Command

Instead of manually typing git commands after validating the feature, the tutor introduces a custom slash command (`/ship-feature`) which acts as an orchestration workflow [39-41].

Workflow sequence executed by `/ship-feature`: 1. Commits all current progress with an auto-generated appropriate message. 2. Pushes the current feature branch to the remote repository. 3. Creates a Pull Request against the `main` branch via GitHub MCP. 4. Merges the Pull Request (Squash merge). 5. Deletes the remote feature branch. 6. Switches back to the local `main` branch. 7. Pulls the latest updated code from `main`. 8. Deletes the local feature branch.

```
/ship-feature
```

Explanation: A custom slash command that automates the entire version control and deployment handover process. It transforms an 8-step manual Git/GitHub GUI process into a single-word terminal command, proving the power of Agentic Coding workflows [41, 42].

Chapter 15: Plugins in Claude Code + Claude Code Notes

Part 1: The Problem — Scaling Agentic Workflows

To understand the core necessity of **Plugins** in Claude Code, the tutor introduces a real-world persona and scenario: **Rahul, a Senior Data Scientist at a Fintech company** building critical Credit Risk Models.

Rahul has completely transformed his coding workflow by heavily integrating Claude Code to achieve 10x productivity. He built a highly personalized "Agentic Workflow" using several Claude Code features: - **Custom Skills:** - **EDA Skill:** Instructs Claude to automatically check dataset shapes, data types, generate missing value heatmaps, flag skewness (if >1.5 or <-1.5), detect target leakage, and compute correlation matrices. - **Feature Engineering Skill:** Instructs Claude to specifically engineer date-time features (e.g., "days since account opened"), apply Target Encoding for high-cardinality features (explicitly avoiding One-Hot Encoding), and flag multicollinearity if VIF (Variance Inflation Factor) is > 10. - **Custom Slash Commands:** Created a `/model_eval` command that instantly generates a confusion matrix formatted in the company's theme, a classification report, ROC curves, SHAP values, and a 1-page summary. - **Hooks:** Implemented a post-tool-use hook that acts as a guardrail. It flags Claude if it tries to use `df.dropna()` without specifying columns, applies scalers to the entire dataset (preventing data leakage), uses hardcoded file paths, or uses generic "accuracy" metrics for imbalanced credit risk datasets. - **MCP (Model Context Protocol):** Connected his Claude Code environment to the company’s internal experiment tracker to seamlessly log models and view past team metrics.

> 📝 **NOTE**
>
> > **The Bottleneck (The "Why"):** While Rahul's personal workflow is highly optimized, onboarding junior data scientists to this exact workflow is a nightmare. Sharing this setup manually requires junior devs to copy the `.claude` folder, manually inject skills, update `settings.json` for hooks, and configure `mcp.json`. This manual distribution is tedious, highly error-prone, and difficult to maintain or version-control.

Part 2: Introduction to Plugins & Architecture

Plugins solve the distribution bottleneck. A plugin allows you to package an entire custom workflow—skills, hooks, custom slash commands, and MCP tools—into a single, installable entity. When another developer installs this package, they get an exact replica of your custom Claude Code environment.

Plugin File Structure

A plugin is fundamentally a structured folder. To be recognized as a valid plugin by Claude Code, it **must** contain a specific configuration directory and manifest file.

```
my_custom_plugin/
├── skills/               # Markdown files for custom skills
├── hooks/                # Python scripts for lifecycle hooks
├── commands/             # Custom slash command markdown files
├── mcp.json              # MCP tool configurations
└── .claude_plugin/       # REQUIRED: Configuration directory
    └── plugin.json       # REQUIRED: The manifest file
```

> 💡 **IMPORTANT**
>
> > **The Manifest File (`plugin.json`):** If the `.claude_plugin/plugin.json` file is missing, Claude Code will not recognize the folder as a plugin. This JSON file contains the metadata for the plugin, including the `name`, `version`, `description`, `author`, `repository` link, and `license`.

Part 3: Marketplaces Explained

If Plugins are the individual packages, **Marketplaces** are where these packages are hosted and distributed.

Mental Model / Analogy: Think of Marketplaces as the **"App Store"** (like Apple App Store or Google Play Store), and Plugins as the individual **"Apps"** you download from them.

Technical Definition of a Marketplace

On a technical level, a Marketplace is simply a GitHub repository that contains a `marketplace.json` file. - The `marketplace.json` holds the owner's information and an array list of all the plugins hosted within that repository.

Types of Marketplaces

Marketplace Type	Description	Installation
Official Marketplace	Maintained directly by Anthropic. Contains highly vetted plugins from reputed companies (e.g., Vercel, Supabase, Figma).	Pre-installed in Claude Code by default.
Third-Party Marketplaces	Custom repositories created by independent developers or specific companies (like Rahul's team).	Must be manually added via a GitHub URL before their plugins can be downloaded.

Part 4: Managing and Installing Plugins

You can interact with plugins directly from the terminal using a dedicated slash command.

```
/plugin
```

- **What it does:** Opens the interactive Plugin Menu in Claude Code.
- **Why use it:** To discover new plugins, view currently installed plugins, or add third-party marketplaces.

Step-by-Step Workflow: Installing a Third-Party Plugin 1. Type `/plugin` and hit `Enter`. 2. Navigate to the **Marketplaces** tab in the UI. 3. Select **"Add Marketplaces"**. 4. Paste the specific GitHub Repository URL of the third-party marketplace. 5. Once added, navigate to the **Discover** tab. You will now see the total plugin count increase (combining Official + Third-Party plugins). 6. Select your desired plugin and hit `Enter` to install it. 7. Choose the installation scope when prompted (either **User Scope** for all your projects, or **Project Scope** for the current repository only).

> 🌱 **TIP**
>
> > **Recommended Popular Plugins:** The tutor recommends exploring plugins like `Super powers` (workflow enhancements), `Frontend Design` (UI/UX guidance), `Context 7` (up-to-date documentation for libraries), `Code Simplifier` (readability), and `Playwright` (browser automation).

Part 5: Project Conclusion & Deployment Workflow

Before deploying, the final project feature ("Delete Expense") is implemented using the automated Agentic workflow established in previous chapters.

1. Final Feature Implementation

```
/create_spec 09 Delete Expense
```

- **What it does:** Triggers a custom slash command to create a feature branch and generate a specification document for the "Delete Expense" feature.
- **Subsequent Steps:**
- Enter **Plan Mode** to generate an implementation plan.
- Approve the plan to let Claude execute the code changes.
- Manually test the UI (verify the delete button works).

```
/ship_feature
```

- **What it does:** Triggers a custom slash command that automates the Git workflow: commits code, pushes to origin, uses GitHub MCP to open a Pull Request, merges the PR via squash, switches back to the `main` branch, pulls latest changes, and deletes the local feature branch.

2. Deployment using the Railway Plugin

The initial plan was to deploy the Flask application using the **Vercel** plugin.

> 💡 **IMPORTANT**
>
> > **Architectural Decision (Vercel vs. Railway):** Vercel is highly optimized for frontend frameworks like Next.js because it treats backend code as ephemeral Serverless Functions. A standard Flask application running on Serverless Functions will frequently spin up and shut down per request, breaking stateful application logic. Therefore, the architectural decision was made to switch to **Railway**, which is much better suited for persistent Flask server deployments.

Step-by-Step Workflow: Deploying via Railway Plugin 1. Create an account on Railway (preferably signing in via GitHub). 2. Install the Railway CLI tool on your local machine. 3.

Log in to the CLI: `bash railway login` * *Explanation:* Opens a browser window to authenticate your local terminal session with your Railway account. 4. Verify authentication: `bash railway whoami` * *Explanation:* Confirms the terminal is successfully connected to your Railway account. 5. Install the **Railway Plugin** via the `/plugin` menu in Claude Code (choose "Project Scope"). 6. Trigger the automated deployment via a natural language prompt: `markdown Deploy this Flask application to Railway and give me a public URL` 7. Claude Code (powered by the Railway Plugin) autonomously analyzes the project, creates required deployment files (like `Procfile`), adds necessary WSGI servers (`gunicorn` to `requirements.txt`), and pushes the deployment to Railway servers.

> ⚠ **WARNING**
>
> > **Production Database Gotcha:** The deployed application uses **SQLite**. Because platforms like Railway use ephemeral file systems, every time the application is re-deployed, the SQLite database file will be wiped, resulting in data loss. For actual production deployments, you **must** migrate to a hosted database solution like PostgreSQL or MySQL.
