READ MODE · ANSWER FIRST

How Claude Code Works

An interactive, visual explanation of How Claude Code Works, from setup through the complete system.

CHEAT SHEET · 7 ESSENTIAL IDEAS

The whole story in 7 lines

How Claude Code Works becomes easier to reason about when every stage is connected as one system.

1. Claude Code is not just a prompt box. The first fork is whether the request should be executed by command handlers or by the query loop.
2. A slash command is not just text with a slash prefix. It resolves into a structured command object with metadata, enablement checks, and...
3. The model is never operating on the raw latest prompt alone. Claude Code assembles a structured context packet first, then drives the...
4. The loop is deliberate: model proposes a tool call, Claude Code validates it, executes the tool, maps the result, and only then lets the...
5. Hooks are not just lifecycle decorations. They can block, annotate, or observe parts of the loop such as `SessionStart`, `PreToolUse`...
6. Plugin activation is not magic. Claude Code explicitly reads plugin manifests, installs their command/skill markdown, loads hook config...
7. The terminal UI is the front door, but the runtime is a broker. App state sits in the middle while bridge, MCP, and LSP services extend...

STAGE 1 OF 9

Setup

Welcome. Before we start, let us cover the big picture. Claude Code is a command-line AI agent that can read files, write code, run commands, and talk to external services. Think of it like giving a skilled developer direct access to your terminal, but with safety rails built in.

The agent loop is the heartbeat of Claude Code. The model reads your request, decides what to do, uses a tool, sees the result, and then decides again. This cycle repeats until the task is done.

Tool use is how Claude Code takes action. Instead of only writing text, the model can call tools that read files, run shell commands, search code, and more. Each tool returns a result that feeds back into the loop.

MCP (Model Context Protocol) lets Claude Code connect to external servers that provide extra tools and data. Hooks are custom checks that run at key moments, like before a tool runs, to enforce rules or add safety logic.

The context window is the total text the model can see at once, like a desk that can only hold so many papers. The system prompt is a set of hidden instructions that shape how the model behaves before it even reads your message.

Over the next seven stages we will walk through the full Claude Code runtime: how input arrives, how commands get routed, how context is built, how the tool loop works, how hooks enforce policy, how plugins extend the system, and how external edges connect to IDEs and servers. Now let us start with where every interaction begins.

Entry Surfaces

02 ENTRY SURFACES

Now that we know the key terms, let us see where every interaction begins. A user request arrives at the CLI entrypoint. Try switching the Entry control in the sidebar between Prompt and Slash to see how the input type changes what happens next.

`main.tsx` boots the runtime and hands your input to the top-level router. Think of it like a receptionist who greets you and decides which department you need.

Commander separates slash commands from ordinary conversational turns. If you typed a slash command, it goes straight to the command handler. If you typed a question, it falls through to the REPL path.

The REPL path is where ongoing agent work lives. Message history and the terminal UI stay alive here so the model can carry on a full conversation with tool use.

The key takeaway is that the very first fork decides everything. Slash commands run immediately while prompts enter the agent loop. Next, we will look at what happens when a slash command reaches the command router.

STAGE 3 OF 9

Command Router

We just saw how `main.tsx` routes input at the entry layer. Now let us zoom into what happens when a slash command reaches the command router. The normalized input arrives and needs to find a matching handler.

`commands.ts` matches the slash name against a live registry. This registry is not a simple list. It merges built-in commands, plugin commands, and skill commands into one lookup space.

Built-in handlers like `/commit` and `/review` are available immediately. Other commands may sit behind feature gates or lazy imports that load only when called. Switch the Source control to Plugin to see where external commands come from.

Plugin commands and skill commands get merged into the same lookup space as built-ins. From the router's perspective, there is no difference between a built-in command and one that came from a plugin.

If the input is not a slash command at all, the router yields to the freeform query path. That query path is exactly what we will explore next: how Claude Code builds context and talks to the model.

STAGE 4 OF 9

Query + Context

Now that we understand how commands get routed, let us look at what happens with freeform prompts. The first thing the engine does is gather recent messages into the active conversation window. Toggle the Memory control on and off to see how the context packet changes.

System prompt state and project context are assembled around that message history. Remember the system prompt from Setup? This is where it gets wired in, along with CLAUDE.md files and environment details.

`QueryEngine.ts` takes all of that and turns it into a structured model request. It attaches tool metadata, retry rules, and streaming configuration. The model never sees just your latest line of text.

The model samples against the full context packet. Think of it like a student reading an entire assignment brief, not just the last question. The richer the context, the better the answer.

The response either becomes plain text output or triggers a tool call that enters the tool loop. That tool loop is the real engine of Claude Code, and we will explore it next.

STAGE 5 OF 9

Tool Loop

05 TOOL LOOP

We just saw how the query engine builds context and calls the model. That raises a question: what happens when the model needs to take action, not just write text? The query engine recognizes a tool call and enters the tool loop.

`tools.ts` exposes the full tool registry. This includes BashTool, FileEditTool, MCPTool, and more. Use the Breadth stepper in the sidebar to see more or fewer tool categories on the board at once.

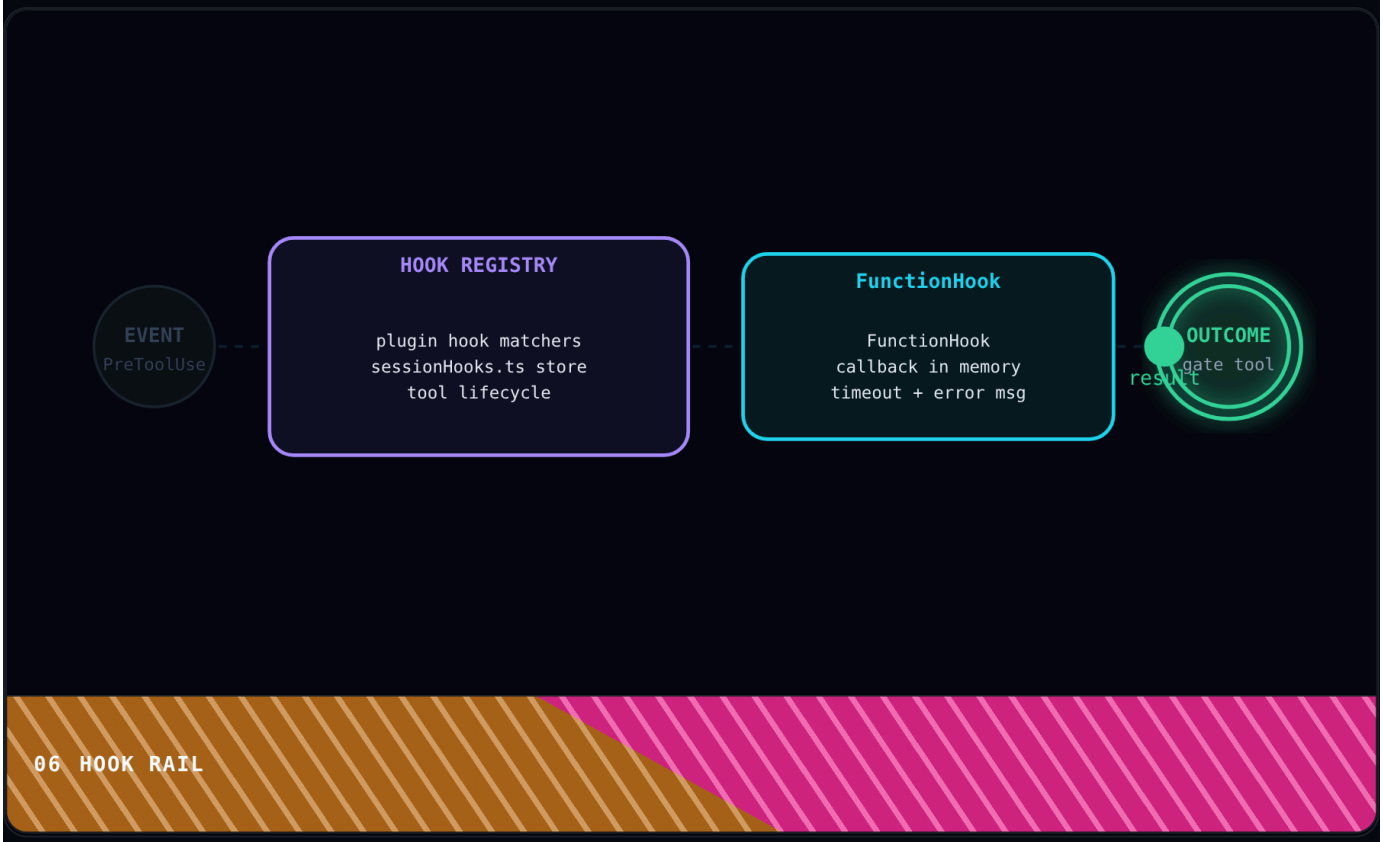
Before any tool runs, the permission layer checks whether it is allowed. Switch the Approval control to Auto to see what happens when policy auto-approves. In default mode, Claude Code may pause and ask you for permission first.

Once approved, the tool executes and returns a structured result block. That result goes right back into the message stream as new evidence for the model to read.

This is the agent loop in action. The model proposes, the runtime validates and runs, the result feeds back, and the model decides again. But who enforces the rules around this loop? Next, we will look at the hook rail.

STAGE 6 OF 9

Hook Rail



Now that we understand the tool loop, let us see how Claude Code enforces safety around it. A lifecycle event fires at key moments. Switch the Event control to see the difference between PreToolUse, PostToolUse, and SessionStart.

When an event fires, registered matchers scan for hooks that care about this specific event and pattern. Think of hooks like hall monitors stationed at doorways. They watch for specific events and decide whether to let things pass.

The matched hook runs as either a shell command or an in-memory function callback. Try switching the Type control between Command and Function to see the two execution styles. Function hooks run faster because they stay in memory.

The hook outcome can block the action, annotate it with extra information, or simply log what happened. A PreToolUse hook might reject a dangerous shell command before it ever runs.

Hooks are the guardrails of the entire runtime. They wrap the tool loop, the session lifecycle, and more. Next, we will look at how plugins bring in new hooks, commands, and servers from outside the core codebase.

STAGE 7 OF 9

Plugin Loader

07 PLUGIN LOADER

We just saw how hooks enforce policy around the tool loop. That raises a question: where do all those extra hooks, commands, and servers come from? The answer is plugins. The loader starts by scanning disk for plugin manifests. Use the Plugins stepper to add more plugin roots and watch the diagram grow.

`useManagePlugins` loads each plugin and isolates failures. If one plugin is broken, it does not take down the whole registry. This is like a power strip with individual fuses for each outlet.

Plugin commands, skills, agents, and hooks all get merged into the live runtime state. Remember the command router from Stage 2? This is how plugin commands end up in that same lookup space.

Plugins can also contribute MCP and LSP server configs. The loader warms those caches so connections are ready when the tool loop needs them. Switch Refresh to Reload to see how the reload path differs from initial startup.

After loading, the REPL and AppState see one expanded capability surface. Plugins are not bolted on. They are woven in. Next, we will look at the external edges where Claude Code connects to the world beyond your terminal.

STAGE 8 OF 9

External Edges

Now that we have seen how plugins extend the runtime from inside, let us look at how Claude Code connects outward. The core app state sits at the center and keeps one coherent picture of the live session. Switch the Edge control to explore each integration.

The Bridge edge lets Claude Code mirror a session into an IDE. When you use Claude Code inside VS Code or another editor, the bridge is what keeps the terminal agent and the IDE in sync. Try switching the Client control to IDE to see how the session changes.

MCP connections make external server tools look like native capabilities. Remember the tool loop from Stage 4? MCP tools show up in that same registry alongside BashTool and FileEditTool.

LSP services add code intelligence like go-to-definition and hover info. These are things that plain shell tools cannot provide on their own. The LSP layer fills that gap.

All three edges feed back into the same agent loop, so you still experience one unified Claude Code session. Now let us step back and see the whole picture together.

STAGE 9 OF 9

Recap

We started at the Entry Surfaces, where every interaction begins. The CLI boot path and Commander decide whether your input is a slash command or a conversational turn for the agent loop.

Then we saw the Command Router, where slash commands resolve against a registry of built-in, plugin, and skill handlers. If the input is not a command, it falls through to the query path.

Next came Query + Context, where the runtime assembles message history, system prompts, and project memory into one structured packet before calling the model.

The Tool Loop is where real work happens. The model proposes a tool call, Claude Code validates and runs it, then feeds the result back so the model can decide what to do next.

The Hook Rail wraps the loop with safety and policy. Hooks can block, annotate, or observe events like tool use and session start, acting as guardrails for the entire runtime.

The Plugin Loader brings in external capabilities. Plugins contribute commands, skills, hooks, MCP servers, and LSP servers, all merged into the live runtime without changing core code.

External Edges connect Claude Code to the world beyond the terminal. Bridge links to IDEs, MCP adds remote tool servers, and LSP provides code intelligence that plain shell tools cannot offer.

Put it all together and you get one unified agent runtime. Input arrives, gets routed, enriched with context, processed through a tool loop with policy hooks, extended by plugins, and connected outward through bridges and servers. That is how Claude Code works.