ChatGPT

# Terminal Coding Agents: Claude Code, Gemini CLI, and the Future of Development

## Slide 1: What Are Terminal Coding Agents?

- **Definition:** Terminal-based AI coding agents are CLI tools that use large language models to generate, modify, and debug code via natural language commands. They run in your terminal with access to shell tools, making them more autonomous than editor plugins [1] [2].
- **How They Work:** You converse with the agent (like ChatGPT in a REPL), or pipe inputs to it, and it executes multi-step actions (reading files, writing code, running tests) to fulfill high-level requests.
- **Examples:** Leading examples include **Claude Code** (Anthropic), **Gemini CLI** (Google's open-source agent using Gemini 2.5 Pro) [3] [4], **Sourcegraph Amp**, and **OpenCode** (open-source model-agnostic agent) [5]. *(Others like GitHub Copilot Chat, Replit Ghostwriter, Cursor, and Windsurf's Cascade agent also pioneered aspects in IDEs.)*

## Slide 2: Why They Matter – Impact and Current State

- **Accelerating Development:** These agents can turn specs into code on demand, shifting the bottleneck from coding speed to **specification clarity** [6]. Developers report faster PR merges, quicker bug fixes, and more ambitious refactors when using AI coding agents [7]. Solopreneurs can build functional MVPs rapidly, delaying the need for big engineering teams [8].
- **Productivity vs. Pitfalls:** The experience, however, can be **uneven**. While some sessions feel like "this is the future of everything," others result in *"prompt churn"* and mistrust of the output [7]. In fact, ~40% of devs have struggled with high iteration overhead (reviewing AI code, re-prompting) and confidence gaps in agent-written code [9].
- **Key Capabilities:** Modern agents handle entire features from a description, automatically debug failing code, answer questions about large codebases, and even interpret other modalities (e.g. generating code from a diagram or UI image) [10] [11]. The **context window** of advanced models is huge (Claude 2 and Gemini offer up to 100K–1M tokens), letting the agent ingest your whole project if needed for context [4].

## Slide 3: Evolution of AI Coding – From Autocomplete to Autonomous Agents

- **Gen 1 – Code Autocomplete:** Early tools (OpenAI Codex in GitHub Copilot, etc.) predicted the next few lines of code. They sped up typing but required the developer to drive every step.
- **Gen 2 – Interactive Assistants:** IDE-integrated chats like Copilot Chat, Sourcegraph Cody, or Replit Ghostwriter let you ask questions and modify code via conversation. This **"vibe coding"** style is human-in-the-loop: you prompt, review diffs, and guide the AI continuously [12] [11].
- **Gen 3 – Agentic IDE Tools:** Advanced IDEs (Cursor's agent mode, Windsurf's "Cascade" agent) started chaining steps: the AI could run build/test commands or multi-file edits automatically. These

were early attempts at autonomy, though often limited and sometimes erratic ("longer leash" experimentation).

- **Gen 4 – Terminal Agents (Today):** Fully agentic CLI assistants like Claude Code and Gemini CLI represent this stage. They autonomously plan and execute multi-step coding tasks (read code, write code, run shell commands) with minimal intervention [13] [14]. Multiple agents can even run in parallel to tackle different tasks concurrently [15]. We're **"here" now – on the cusp of widespread agent-driven development.
- **Gen 5 – The Next Frontier:** Coming soon are more **multi-agent orchestration** and **context-aware** workflows. Agents might coordinate with each other on sub-tasks and maintain more persistent memory of project goals. Some developers already claim "I barely open my IDE anymore" as agents handle most coding; this hints at a near-future where AI could implement features from high-level intent with very little human coding at all.

## Slide 4: How Terminal Agents Operate – Interactive vs. Batch Modes

- **Interactive REPL:** You can run the agent in a shell session (e.g. just type `claude` or `amp`). You converse by entering natural-language instructions, and the agent responds with plans or code changes, executing tools as needed in between. The agent remembers context within the session (up to its token limit), giving a conversational coding experience.
- **Batch & Unix Pipeline:** These agents can also work as one-off command-line tools. For example, you can pipe data in and get a result out: `git diff | claude -p "Explain this diff"` will output an explanation. In CI or scripts, you might run a fixed prompt from a file with the CLI. *This Unix-like usage is evolving:* e.g., Claude Code's SDK allows running a prompt with restricted tools for automation [16]. Google's Gemini CLI similarly supports quick uses inside other dev tools [2].
- **Example:** *Interactive mode:* Ask Claude Code *"How is the* `FooBar` *class used across the repo?"* and it will search and summarize occurrences across files [10]. *Batch mode:* Pipe an error log into the agent with a prompt to get an instant analysis in the terminal (no chat history maintained).

## Slide 5: Getting Started – Installation and Setup

- **Install the Agent:** Most agents are a single CLI install. For Claude Code: `npm install -g @anthropic-ai/claude-code` (requires Node 18+). For Google's Gemini CLI: use npx to run the package from GitHub [17]. Sourcegraph's Amp can be installed via npm (`@sourcegraph/amp` CLI) [18]. These tools work on Mac, Linux, and Windows (Windows Terminal supported for Gemini CLI) [19].
- **First Run Configuration:** On first launch, you may be prompted to select a theme (dark/light) and log in to the model service (Claude or Gemini) [17]. For Claude, Pro users just run `claude /terminal-setup` to set things like Shift+Enter for newlines [20]. Gemini CLI requires linking a Google account to unlock the free quota (1M-token context, 1000 requests/day) [4].
- **Project Initialization:** Navigate to your code repository and start the agent there. Claude Code will automatically index the project structure (it reads files on-demand, no manual indexing needed) [10]. A great first step is asking broad questions about the codebase ("What does this repo do? How is X implemented?") to warm up context. *Tip:* Claude Code can even inspect git history or GitHub issues if asked, without extra setup (it uses your local Git data and can call APIs via tools).

## Slide 6: Context and Memory – Guiding the Agent Effectively

- **Context Engineering:** Provide the agent with sufficient *context* so it understands your project and requirements. Claude Code supports special files like **CLAUDE.md** – a markdown file where you write project-specific guidance (design decisions, coding style, API keys to ignore, etc.). This file is auto-loaded into the agent's context every session [21]. Keep it concise (a few KB) and well-structured (bullet points, sections) to maximize its utility [22].
- **Global vs Project Memory:** Claude Code checks `~/.claude/CLAUDE.md` for user-wide preferences (your personal "memory") and the repo's own `CLAUDE.md` for project rules [23]. It even supports nested CLAUDE.md files in subfolders (useful for monorepos or distinct modules). Other agents have similar concept of **system prompts or config files** – e.g., OpenCode loads a global config and project-specific settings.
- **Long Context Windows:** Advanced models let you stuff a lot into context (hundreds of pages). But more isn't always better – irrelevant details can distract the AI (*"context poisoning"*). A best practice is to curate what you feed: summarize or trim outdated sections, and use references (like pointing the agent to specific files with an `@filename` syntax) rather than dumping everything. *Example:* Claude Code's `@` references let it pull in file content on demand without permanently eating up context [24].
- **Memory Commands:** Use the agent's built-in commands to manage context. For instance, `claude /memory` shows which files/notes are currently loaded [21]. You can prune or reset context if the conversation goes off track. It's often better to start a fresh session (or use a new agent thread) than to let too much irrelevant text accumulate – avoid using any "/compact" or auto-summarize feature that may distort memory (better to manually summarize or restart for clarity).

## Slide 7: Workflow Basics – Planning, Coding, and Verification

- **"Plan-then-Code" Strategy:** Don't jump straight into code generation; ask the agent to draft a plan first. For example: "Add logging to `router.ts`. **Before writing code, brainstorm a 3-step plan and ask for my approval.**" Claude will output a step-by-step plan (e.g. which function to modify, how to log, how to test) [13]. Review this plan to catch misunderstandings early. Once you say "Looks good," the agent proceeds to implement it. This avoids 3000-line surprise diffs and tends to yield higher-quality results [25].
- **Autonomous Execution:** After approval, the agent chains actions automatically. It might do a file search to find where to insert code, use the *edit tool* to apply changes, then run tests with *bash tool*, etc. [13]. For example, with one prompt you can have Claude Code create a new React component, write Jest tests, run `npm test`, notice failures, fix the code, and repeat until tests pass [14]. The agent stops when the acceptance criteria are met (all tests green, or whatever goal you set).
- **Iterative Refinement:** Treat the agent like a junior pair-programmer. You can ask it to self-check its work: *"Run the tests and ensure everything passes."* It will run them and, if failures occur, immediately attempt fixes. Always keep an eye on the diffs it produces. If something looks off, you can interrupt (hit Esc to stop mid-action [26]) and clarify the requirement or fix manually.
- **Verification and Testing:** Leverage the agent to generate tests for any new code. It's good practice to say, *"Write a unit test for X before you implement, then make it pass."* This makes the agent validate its output. Claude Code, for instance, can be told to continuously loop on a prompt like *"Implement feature X per spec, run tests, and keep fixing until tests pass."* This **loopback** workflow often yields working code in one go [14]. Just be sure to review the final tests to ensure they truly cover the spec (no cheating).

## Slide 8: Safety, Control, and Best Practices

- **Permissioning:** By default, most agents will *ask* before performing potentially destructive actions (like running a database migration or installing a package). In Claude Code, risky shell commands prompt for confirmation unless whitelisted [27]. You can streamline trusted operations by creating a `permissions.json` (for common commands like tests, git pulls, etc.) [28]. If you fully trust the agent or are working in a disposable environment, you might run in a "dangerously accept all" mode – but use with caution. *Recommendation:* Keep version control on – commit after each agent action, so you can revert easily if it goes off-script.
- **Undo and Interrupt:** Agents can make mistakes. Know how to stop or roll back. In Claude Code: pressing `Esc` once interrupts the current action; pressing `Esc` twice steps back one turn in the conversation if you need to undo recent context [26]. If the agent starts doing something crazy, you can always terminate the session and restart. Nothing persistently bad can happen if you have your code in Git – worst case, you reset to last good commit.
- **Using Sandboxes/Branches:** It's wise to let the agent work in an isolated git branch or a clean environment. Claude Code will infer branch names and commit messages for you when asked to commit [29]. Example: *"Refactor* `dateUtils` *and open a PR"* – it will create a new branch, commit changes, and even initiate a GitHub pull request with a title [30]. Still, review those changes via PR; do not directly push to production without human eyes.
- **Watch Out for Edge Cases:** Agents lack true understanding and may introduce subtle bugs (e.g., off-by-one errors, security issues). Use your normal code review and testing practices on AI-written code. Specifically, inspect how it handled any ambiguous requirement – since agents won't ask clarifying questions, *ambiguity in prompt = ambiguity in code*. Always clarify specifications in your prompt (e.g. "use O(1) memory" or "ensure thread-safety") to align the agent's output with expectations.

## Slide 9: Advanced Features – Customization and Parallelism

- **Slash Commands & Macros:** Terminal agents often come with **built-in commands** and allow user-defined ones. Claude Code has 60+ slash commands like `/theme`, `/memory`, `/plan` etc. [31] and you can add your own. For example, you might define a custom command `/deploy` that contains a series of instructions (build, containerize, etc.). In Claude, you can save these as `.md` files under `.claude/commands/` for project-specific or `~/.claude/commands/` for global reuse. This is a powerful way to encapsulate common tasks so you just invoke `/fix-lint` instead of typing the whole prompt each time.
- **Integrating Team Tools (MCP):** Claude Code introduced an **MCP (Multi-Channel Protocol)** to let the agent interface with external services. By editing an `mcp.json`, you can connect the AI to custom tools or servers. E.g., you can give Claude a Puppeteer server to control a headless browser [32] – now it can render a React component, screenshot it, and verify UI output as part of a test loop. Teams can similarly plug in internal APIs or devops commands. This extends the agent beyond code editing: it can become a full DevOps or QA assistant when configured [33].
- **Parallel Agents:** Why use one AI agent when you can use several? Advanced users run multiple agent sessions in parallel to speed up large tasks. For instance, you might split a big refactor into backend and frontend and run two Claude Code instances simultaneously (each in a separate tmux pane or terminal) [15]. Sourcegraph's Amp supports parallel thread limits (you can raise `maxParallelTasksCount` if your tasks don't conflict). This "multi-boxing" of LLMs can drastically

cut down waiting time, as long as the tasks are independent. Some IDEs like Zed even demo'd running 3 agents in parallel to handle different file sets concurrently.

- **Session Persistence:** Tools like Claude and Amp allow resuming sessions (`claude --resume`) to pick up where you left off, which is handy for long-running tasks or if you accidentally close the terminal [15]. Just be mindful of context length – very long sessions may need a summary reset or handoff to a fresh agent to avoid confusion.

## Slide 10: Case Study Example – "Fix the Bug and Add a Feature"

*(Illustrative walkthrough of using a terminal agent for a real task.)* - **Task:** *"There's a bug in the signup flow and we need to add email verification."* – In a traditional flow, a dev would search the codebase, write a fix, add a new module for verification, test, etc. With a terminal agent, we can delegate much of this: - **Step 1: Ask Q&A** – *"Why are new users not receiving welcome emails?"* The agent searches the code and finds an exception in the email sender logic [10]. It explains which module is failing and why (e.g. misconfigured API key). - **Step 2: Debug/Fix** – *"Fix the bug in the email sender (it's failing auth)."* The agent locates the code, applies a fix (perhaps updating an API endpoint or key), and even suggests adding a unit test for this scenario. You approve the changes after review. - **Step 3: Implement Feature** – *"Now add an email verification step to the signup. Plan it out first."* The agent produces a plan: e.g. *1) create a verification token model, 2) email template, 3) verification API endpoint, 4) tests*. You tweak the plan if needed, then say go. Claude Code writes the new files, updates config, sends test emails via a stub (if you configured an SMTP tool or a mock), and writes tests. It runs `npm test` – maybe a couple fail – it adjusts code, and soon all tests pass [14]. - **Step 4: Review & Merge** – You inspect the git diff of all changes. The agent already prepared a commit on a new branch and even opened a PR [30]. The diff looks good (thanks to your guidance in the planning stage and the agent's adherence to project style via CLAUDE.md). Finally, you merge the PR. **Outcome:** bug fixed and feature shipped in a fraction of the time. - **Result:** This example shows how a developer's role shifts to high-level oversight: you specified the goal, provided clarifications, and vetted the output, but the agent did the heavy lifting in code and test writing. The faster iteration loop and ability to operate on multiple files at once is a game-changer.

## Slide 11: Challenges in the Agent Era

- **Breaking Traditional Workflows:** Our familiar dev processes (Waterfall specs, Agile sprints, code review, GitFlow) assume a human is writing the code. Agent autonomy breaks these assumptions. For example, an agent won't ask a product manager to clarify ambiguous instructions – it will *guess* a solution, which might be misaligned [34]. This means any ambiguity in a spec or ticket can lead the agent down a wrong path silently. Teams must adapt by writing much clearer, testable requirements up front.
- **Multi-Developer Coordination:** When multiple people and agents collaborate, context fragmentation is a problem. Each agent session might have a different view unless they share the same spec and memory. There's no implicit knowledge – everything must be explicit. This can make **integration** of work tricky, as agents might make conflicting changes if not guided by a unified vision. New conventions (like keeping a single source-of-truth spec file that agents reference [35], or trunk-based development with small batches) are emerging to mitigate this.
- **Trust and Verification:** Letting an agent commit code directly raises the question: can you trust it? Tests help, but tests can have gaps. Code review is still necessary, but reviewing AI-generated code can be tedious if it's large. Developers report **"review drag"** where checking the AI's work takes as long as writing it would have. To combat this, keep agent changes small and self-contained (use the

plan feature to limit scope), and invest in comprehensive automated tests so you can trust green CI runs.

- **Cost and Performance:** Running these models isn't free. With large context windows and long sessions, the token usage (which translates to API cost) can spike. There are also speed considerations – executing a complex multi-step fix might take a couple of minutes of AI "thinking". It's easy to burn through tokens when an agent gets stuck in a loop. Monitoring tools (OpenTelemetry metrics for Claude Code, for example) can track tokens consumed, number of edits, etc. [36]. Expect tooling and pricing models to evolve (ideas like outcome-based pricing or local model offloading are being explored [37] [7]).
- **Ethical & Security Concerns:** An autonomous coding agent has the power of your terminal – which means it can delete files, make web requests, etc. Ensure it runs with appropriate safeguards: e.g., use sandboxed credentials (maybe a dummy database for testing, not production keys), and review any code it wants to run. There's also the aspect of data privacy: check whether the agent uploads any code to a cloud service. (Claude Code keeps code local and doesn't train on it [38], and Gemini CLI is open source, but always double-check these guarantees for each tool.)

## Slide 12: Mastering Your AI Pair Programmer – Tips & Tricks

- **Be the "Conductor":** Approach coding with an agent like conducting an orchestra [11]. You provide high-level direction and timing, the agent plays the instruments (writes the code). Maintain control by explicitly stating what you want: break tasks into smaller prompts, and give frequent feedback. If the output isn't what you hoped, clarify or show an example of the format you expect (few-shot examples can steer it better than abstract instructions).
- **Explicitness is Everything:** Remember, the agent is **literal** and has no intuition beyond its training. Always specify requirements that a human might assume. Instead of "optimize this function," say "optimize for memory usage and clarify the algorithm with comments." Instead of "make it user-friendly," describe what "user-friendly" means (e.g. "add input validation and more descriptive error messages"). Treat the AI like a very smart but forgetful junior dev who needs an exact definition of "done" each time.
- **Leverage Chain-of-Thought:** If the agent offers a reasoning or plan (some allow a special mode to output `<thinking>` steps), pay attention to it. You can often spot a flawed assumption in its chain-of-thought before it writes code. Don't hesitate to ask the agent *"explain why you chose this approach"* – a good agent will articulate its reasoning, and you can catch misunderstandings early.
- **Use Iterative Prompts:** You don't have to get the perfect prompt the first time. It's often more effective to have a back-and-forth: *"Here is my spec... What do you think is the best approach?"* -> agent suggests approach -> *"Okay, implement step 1."* -> agent codes -> *"Now step 2..."* This incremental guidance prevents the agent from wandering too far off course. Many Claude Code users start with a brainstorming prompt to let the AI propose solutions, then iterate [13].
- **Continuously Update Your** CLAUDE.md**** (or equivalent): After each significant agent misstep or learning, encode that knowledge into your project's instructions. If the agent made a wrong assumption once ("e.g., assumed X library when we use Y"), put a note in CLAUDE.md about it ("Always use Y for PDF generation"). This way each new session learns from past mistakes. Over time, your custom memory makes the agent more aligned to your project's unique quirks.

## Slide 13: Outlook – The Future of Software Development with AI Agents

- **Developers' New Role:** As coding agents handle more of the grunt work, human developers will focus on higher-level skills: architecture design, writing precise specifications/tests, and guiding the AI to make the right trade-offs [39] [40]. The term *"software composer"* or *"software conductor"* is apt [39] – we'll assemble and orchestrate modules built by AI, ensuring the final product meets real-world needs.
- **The New Stack (Specs, Code, Tests):** In the past, code was the central artifact and specs were secondary. Now we're moving to **spec-driven development**: the specification (user stories, acceptance criteria, design docs) is a first-class artifact that might even be executed by agents [41]. Code becomes a by-product of the spec (one possible implementation), and tests verify the spec. Maintaining clear, version-controlled specs and up-to-date tests will be as crucial as the code itself.
- **Increasing Autonomy:** Today's agents can maybe build a medium feature in a few hours of guidance. Future models (like Google's upcoming *Gemini "Kingfall"* or OpenAI's next-gen agents) aim to work **longer autonomously**, handling projects overnight or taking on whole epics with minimal input [42]. We may see agents that debug entire systems during off-hours or collaborate as a team of sub-agents specialized in frontend, backend, data, etc., coordinating via natural language.
- **Multi-Agent Collaboration:** Research is ongoing into agents that talk to other agents. For coding, this could mean a "planner" agent breaks a project into tasks and assigns them to "coder" agents, then a "integrator" agent assembles and tests the result. Early signs of this are visible in some workflows (using one agent to generate specs or review code written by another). This could drastically speed up development, but also demands robust coordination protocols to avoid chaos.
- **Human-in-the-Loop Forever:** Despite the advancements, human judgment remains irreplaceable. AI can generate solutions, but deciding *which* solution is optimal, ethical, and aligned with business goals is a human job. The best outcomes arise when developers treat AI agents as powerful assistants – not infallible oracles. The near future will still have us pairing with our AI "interns," just at a much higher level of abstraction. Embrace the change, but keep your critical thinking and software engineering fundamentals sharp – those will never go out of style.

---

1 3 Gemini CLI vs Claude Code vs Cursor: Which AI Coding Agent Works Best for Devs? | by Kittikawin L. | Jul, 2025 | Medium
https://medium.com/@kittikawin_ball/gemini-cli-vs-claude-code-vs-cursor-which-ai-coding-agent-works-best-for-devs-2917428aa066

2 4 17 19 42 Google releases Gemini CLI with free Gemini 2.5 Pro
https://www.bleepingcomputer.com/news/artificial-intelligence/google-releases-gemini-cli-with-free-gemini-25-pro/

5 opencode: Your terminal's AI agent, with any model you want | Product Hunt
https://www.producthunt.com/products/opencode

6 "(AI coding agents) moved the primary software bottleneck from development speed to specification clarity (which one could argue has _always been the problem_)." | John Gannon
https://www.linkedin.com/posts/jgannon_beyond-code-centric-activity-7342204657638825985-ul91

7 8 9 11 34 35 39 40 41 Beyond Code-Centric: Agents Code but the Problem of Clear Specification Remains
https://www.gregceccarelli.com/writing/beyond-code-centric

10 13 14 15 16 20 21 22 23 25 26 27 28 29 30 31 32 33 36 38 claude-code-cheatsheet-TRIBE.pdf
https://drive.google.com/file/d/1Wz4oHzoMEqXhpLH-kN-mCHQkyioMxKnN

12 Vibe Coding vs. Agentic Coding: 2025 Beginner’s Guide to AI-Driven Development - DEV Community
https://dev.to/dumebii/vibe-coding-vs-agentic-coding-2025-beginners-guide-to-ai-driven-development-2oao

18 Amp CLI - @sourcegraph/amp - npm
https://www.npmjs.com/package/@sourcegraph/amp/v/0.0.1745121793-7a0270

24 Using Claude Code with your Pro or Max plan | Anthropic Help Center
https://support.anthropic.com/en/articles/11145838-using-claude-code-with-your-pro-or-max-plan

37 nibzard - Home
https://www.nibzard.com/