Building an Agentic System

There's been a lot of asking about how Claude Code works under the hood. Usually, people see the prompts, but they don't see how it all comes together. This is that book. All of the systems, tools, and commands that go into building one of these.

A practical deep dive and code review into how to build a self-driving coding agent, execution engine, tools and commands. Rather than the prompts and AI engineering, this is the systems and design decisions that go into making agents that are real-time, self-corrective, and useful for productive work.

Why This Guide Exists

I created this guide while building out highly specialized, vertical agents. I'll often start with a problem with a framework and then unbundle parts of it, which is why I immediately wanted to take an agent I was building in Go and see how an agent like Claude Code could improve it, with a bunch of other features like rich components, panes, effectively Devin for your terminal. More on that soon.

(Note to the reader, I did do the Rewrite it in Rust thing, and it originally was, but Charm is excellent. Ultimately, it came down to wanting to bind to jujutsu for handling safe, linearizable checkpoints and the ability for multiple agents and humans to work together)

After diving deep into Claude Code and similar architectures, I realized there's a gap in practical, engineering-focused documentation on how these systems actually work. Most resources either stay at a theoretical level or skip to implementation details without covering the critical architectural decisions. This is really a "how things work" book, and the software pieces themselves would be recognizable.

In addition, I've provided documentation on every tool and command, and its implementation. This is where this documentation shines - combining those with the execution rules reveals a lot of why Claude Code works so well. Don't skip either

section!

This isn't just about Claude Code or anon-kode. It's about the underlying patterns that make real-time AI coding assistants feel responsive, safe, and genuinely useful—patterns I've found while building my own system.

What You'll Find Inside

This guide dissects a working agentic system architecture with a focus on:

1. **Responsive Interactions** - How to build systems that stream partial results instead of making users wait for complete responses
2. **Parallel Execution** - Techniques for running operations concurrently without sacrificing safety
3. **Permission Systems** - Implementing guardrails that prevent agents from taking unauthorized actions
4. **Tool Architecture** - Creating extensible frameworks for agents to interact with the environment

I've deliberately focused on concrete engineering patterns rather than theoretical ML concepts. You'll find diagrams, code explanations, and architectural insights that apply regardless of which LLM you're using.

Who am I?

Hi! I'm Gerred. I'm a systems engineer, with a deep background in AI and Kubernetes at global scale, but overall I care deeply about everything from frontend UX to infrastructure and I have *opinions*. My background includes:

- Early work on many CNCF projects and Kubernetes
- Creator of KUDO (Kubernetes Universal Declarative Operator)
- Early deployment of GPUs onto Kubernetes for holoportation and humans in AR/VR
- Data systems at scale at Mesosphere, including migration to Kubernetes
- One of the initial engineers on the system that would grow to become Platform One

- Implementing AI systems in secure, regulated environments
- Designing and deploying large-scale distributed systems
- Currently developing frameworks for specialized agents with reinforcement learning, especially with VLMs

I like robust systems with awesome UX and interactive experiences, but that's beyond the scope of this book.

Why Build Your Own Agent?

Commercial AI coding assistants are impressive but come with limitations:

1. **Context boundaries** - Most are constrained by input/output limits
2. **Extensibility challenges** - Limited ability to add specialized capabilities
3. **Integration gaps** - Often struggle to connect with existing workflows and tools
4. **Domain specificity** - General-purpose assistants miss nuances of specialized domains

Building your own agent isn't just about technical independence—it's about creating assistants tailored to specific workflows, domains, and security requirements.

State of This Work

This guide represents my analysis of several coding agent architectures, including Claude Code, anon-kode, and my own experimental system. It's currently in active development as my own agent enters final testing.

The patterns documented here have proven effective in practical applications, but like any engineering approach, they continue to evolve. I'm sharing this now because these architectural insights solved real problems for me, and they might help you avoid similar challenges.

How to Use This Guide

If you're building an AI coding assistant or any agentic system:

- Start with the system architecture diagram for a high-level overview
- Explore specific components based on your immediate challenges
- Pay particular attention to the parallel execution and permission system sections, as these address common pain points

For a deeper exploration of specific subsystems, the tool and command system deep dives provide implementation-level details.

The code examples reference both Claude Code architecture and anon-kode (by Daniel Nakov - https://github.com/dnakov/anon-kode), which is included as a submodule. By examining the anon-kode implementation alongside this guide, you'll get hands-on experience with the concepts described here.

Getting Started

To make the most of this guide and explore the code examples:

1. **Clone the Repository**

   ```
   git clone https://github.com/gerred/building-an-agentic-system.git
   cd building-an-agentic-system
   ```

2. **Initialize Submodules** This repository includes anon-kode as a submodule, which is a reference implementation:

   ```
   git submodule update --init --recursive
   ```

3. **Explore the Code** The `anon-kode` submodule contains the reference implementation by Daniel Nakov (https://github.com/dnakov/anon-kode). This is an excellent terminal-based AI coding tool that works similarly to Claude Code and provides a concrete example of the architecture described in this guide.

Guide Map

Here's how this guide is organized:

- Introduction - Overview of agentic systems and why they matter
- **Core Architecture**
  - System Architecture Diagram - Visual overview of the entire system
  - Core Architecture - The fundamental building blocks
  - **Execution Flow** - How requests flow through the system
  - **Parallel Tool Execution** - Running operations concurrently
  - **Permission System** - Security guardrails for agents
- **Components**
  - Tool System - How tools are defined and executed
  - **Command System** - How commands control the agent
- **Reference**
  - Tools - Detailed documentation of each tool
  - **Commands** - Detailed documentation of each command
- **Real-World Applications**
  - **Real-World Examples** - Case studies and practical applications
  - **Lessons Learned** - Challenges and solutions

Connect and Support

I'm actively building in this space and available for consulting. If you need help with:

- Verticalized agents for specific domains
- Production agent deployments
- Practical AI system architecture
- Making this stuff actually work in real environments

Reach out by email or on X @devgerred.

If this work's valuable to you, you can support my ongoing research through Ko-fi.

Let's dive into the architecture that makes these systems work.

Introduction

I've been diving into Claude Code recently—an AI coding assistant that feels genuinely practical instead of just cool on paper. If you're thinking about building your own agentic systems, there's some clever stuff here worth exploring. Claude Code nails three important things that many tools overlook: snappy real-time feedback, clear safety guardrails, and a plugin design that's actually easy to extend. This guide breaks down how anon-kode, a fork of Claude Code, tackles these ideas, along with some thoughts on the engineering choices they made along the way.

For advanced users wanting to build their own Claude Code, this is a deep guide into how the command loop, execution flows, tools, and commands all come together in a way that is, as far as I know, unique to Claude Code, Anon Kode, and my coding agent.

Let's dig in.

First off, here's a little bit of how Claude introduces this code review:

> Claude Code combines a terminal UI with an LLM, making AI-assisted coding smoother and genuinely usable. If you're thinking about building your own, Claude Code stands out by handling three tricky issues better than most:
>
> 1. Instant results: Uses async generators to stream output immediately, avoiding laggy interactions.
> 2. Safe defaults: Has clear, structured permissions to prevent accidental file or system modifications.
> 3. Simple extensibility: Plugin architecture is clean, consistent, and easy to build on.
>
> Let's dig into how anon-kode (a fork of Claude Code) approaches these problems with a React terminal UI, structured tool system, and careful parallel execution.

This was written and reviewed by humans and AI. While even on manual review it generally is accurate and is architecturally similar to my coding agent, `coder`, errors

may be present. Feel free to open a pull request!

How can I use this guide?

This is is a "how-to" and technical review on building an agentic system, so go build!

You can contact me on X at @devgerred, or support my Ko-fi.


```
@misc{building_an_agentic_system,
  author = {Gerred Dillon},
  title = {Building an Agentic System},
  year = {2024},
  howpublished = {https://gerred.github.io/building-an-agentic-system/}
}
```

Claude Code vs. anon-kode

Before diving deeper, here's how Claude Code and anon-kode relate:

- **Claude Code** is a research preview by Anthropic, integrating AI capabilities into your terminal. It's built with Node.js, React, and Ink for terminal UIs, using Anthropic's AI via API (docs here).
- **anon-kode** is a fork of Claude Code with a few additions:
    1. **Multi-provider support**: Supports multiple AI providers through OpenAI-compatible APIs (Anthropic, OpenAI, local models, etc.).
    2. **Easy configuration**: Adds a `/model` command to quickly switch between providers.
    3. **Customizations**: UI tweaks, branding changes, and adjusted defaults.

Both share the same core architecture and execution flow—this guide focuses on those shared components, highlighting anon-kode specifics where relevant.

Overview and Philosophy

Claude Code combines a terminal UI with an AI backend and a thoughtfully designed tool system. Its architecture addresses four key challenges:

1. **Instant results**: Uses async generators to stream output as it's produced.

   ```
   // Streaming results with generators instead of waiting
   async function* streamedResponse() {
     yield "First part of response";
     // Next part starts rendering immediately
     yield await expensiveOperation();
   }
   ```

2. **Safe defaults**: Implements explicit permission gates for file and system modifications.

3. **Extensible by design**: Common interface patterns make adding new tools straightforward.

4. **Transparent operations**: Shows exactly what's happening at each step of execution.

The result is an AI assistant that works with your local development environment in a way that feels fast, safe, and predictable.

Core Architecture

Claude Code's architecture consists of three primary components that work together to create an effective AI-powered CLI:

Terminal UI (React + Ink)

The UI layer leverages React Ink to deliver rich terminal interactions beyond standard CLI capabilities:

- Interactive permission prompts for secure tool execution
- Syntax-highlighted code snippets for better readability
- Real-time status updates during tool operations
- Markdown rendering directly within the terminal environment

React hooks provide clean state management, enabling complex interactive experiences while maintaining a terminal-based interface.

Intelligence Layer (Claude API)

The intelligence layer connects with Claude through a streaming API interface:

- Parses responses to identify intended tool executions
- Extracts parameters from natural language instructions
- Validates input using Zod schemas to ensure correctness
- Handles errors gracefully when Claude provides invalid instructions

Communication flows bidirectionally - Claude triggers tool execution, and structured results stream back into the conversation context.

Tools Layer

Each tool in the system follows a consistent pattern:

```
const ExampleTool = {
  name: "example",
  description: "Does something useful",
  schema: z.object({ param: z.string() }),
  isReadOnly: () => true,
  needsPermissions: (input) => true,
  async *call(input) {
    // Execute and yield results
  }
} satisfies Tool;
```

This approach creates a plugin architecture where developers can add new capabilities by implementing the Tool interface. Available tools are dynamically loaded and presented to Claude, establishing an extensible capability framework.

Reactive Command Loop

At its core, Claude Code operates through a reactive command loop - processing user input via Claude's intelligence, executing resulting actions, and displaying outcomes while streaming results in real-time.

The fundamental pattern powering this flow uses generators:

```
// Core pattern enabling streaming UI
async function* query(input: string): AsyncGenerator<Message> {
  // Show user's message immediately
  yield createUserMessage(input);

  // Stream AI response as it arrives
  for await (const chunk of aiStream) {
    yield chunk;

    // Process tool use requests
    if (detectToolUse(chunk)) {
      // Execute tools and yield results
      for await (const result of executeTool(chunk)) {
        yield result;
      }

      // Continue conversation with tool results
      yield* continueWithToolResults(chunk);
    }
  }
}
```

This recursive generator approach keeps Claude Code responsive during complex operations. Rather than freezing while waiting for operations to complete, the UI updates continuously with real-time progress.

Query Implementation Details

The complete query function handles all aspects of the conversation flow:

```
async function* query(
  input: string,
  context: QueryContext
): AsyncGenerator<Message> {
  // Process user input
  const userMessage = createUserMessage(input);
  yield userMessage;

  // Get streaming AI response
  const aiResponseGenerator = querySonnet(
    normalizeMessagesForAPI([...existingMessages, userMessage]),
    systemPrompt,
    context.maxTokens,
    context.tools,
    context.abortSignal,
    { dangerouslySkipPermissions: false }
  );

  // Stream response chunks
  for await (const chunk of aiResponseGenerator) {
    yield chunk;

    // Handle tool use requests
    if (chunk.message.content.some(c => c.type === 'tool_use')) {
      const toolUses = extractToolUses(chunk.message.content);

      // Execute tools (potentially in parallel)
      const toolResults = await executeTools(toolUses, context);

      // Yield tool results
      for (const result of toolResults) {
        yield result;
      }

      // Continue conversation recursively
      const continuationGenerator = query(
        null, // No new user input
        {
          ...context,
          messages: [...existingMessages, userMessage, chunk,
...toolResults]
        }
      );

      // Yield continuation messages
      yield* continuationGenerator;
    }
  }
}
```

Key benefits of this implementation include:

1. **Immediate feedback**: Results appear as they become available through generator streaming.
2. **Seamless tool execution**: When Claude invokes tools, the function recursively calls itself with updated context, maintaining conversation flow.
3. **Responsive cancellation**: Abort signals propagate throughout the system for fast, clean cancellation.
4. **Comprehensive state management**: Each step preserves context, ensuring continuity between operations.

Parallel Execution Engine

A distinctive feature of Claude Code is its parallel tool execution system. This capability dramatically improves performance when working with large codebases - tasks that might take minutes when executed sequentially often complete in seconds with parallel processing.

Concurrent Generator Approach

Claude Code implements an elegant solution using async generators to process multiple operations in parallel while streaming results as they become available.

The core implementation breaks down into several manageable concepts:

1. Generator State Tracking

```
// Each generator has a state object tracking its progress
type GeneratorState<T> = {
  generator: AsyncGenerator<T>    // The generator itself
  lastYield: Promise<IteratorResult<T>>  // Its next pending result
  done: boolean                   // Whether it's finished
}

// We track all active generators in a map
const generatorStates = new Map<number, GeneratorState<T>>()

// We also track which generators are still running
const remaining = new Set(generators.map((_, i) => i))
```

2. Concurrency Management

```
// Control how many generators run simultaneously
const { signal, maxConcurrency = MAX_CONCURRENCY } = options

// Start only a limited batch initially
const initialBatchSize = Math.min(generators.length, maxConcurrency)
for (let i = 0; i < initialBatchSize; i++) {
  if (generators[i]) {
    // Initialize each generator and start its first operation
    generatorStates.set(i, {
      generator: generators[i],
      lastYield: generators[i].next(),
      done: false,
    })
  }
}
```

3. Non-blocking Result Collection

```
// Race to get results from whichever generator finishes first
const entries = Array.from(generatorStates.entries())
const nextResults = await Promise.race(
  entries.map(async ([index, state]) => {
    const result = await state.lastYield
    return { index, result }
  })
)

// Process whichever result came back first
const { index, result } = nextResults

// Immediately yield that result with tracking info
if (!result.done) {
  yield { ...result.value, generatorIndex: index }

  // Queue the next value from this generator without waiting
  const state = generatorStates.get(index)!
  state.lastYield = state.generator.next()
}
```

4. Dynamic Generator Replacement

```
// When a generator finishes, remove it
if (result.done) {
  remaining.delete(index)
  generatorStates.delete(index)

  // Calculate the next generator to start
  const nextGeneratorIndex = Math.min(
    generators.length - 1,
    Math.max(...Array.from(generatorStates.keys())) + 1
  )

  // If there's another generator waiting, start it
  if (
    nextGeneratorIndex >= 0 &&
    nextGeneratorIndex < generators.length &&
    !generatorStates.has(nextGeneratorIndex)
  ) {
    generatorStates.set(nextGeneratorIndex, {
      generator: generators[nextGeneratorIndex],
      lastYield: generators[nextGeneratorIndex].next(),
      done: false,
    })
  }
}
```

5. Cancellation Support

```
// Check for cancellation on every iteration
if (signal?.aborted) {
  throw new AbortError()
}
```

The Complete Picture

These pieces work together to create a system that:

1. Runs a controlled number of operations concurrently
2. Returns results immediately as they become available from any operation
3. Dynamically starts new operations as others complete
4. Tracks which generator produced each result
5. Supports clean cancellation at any point

This approach maximizes throughput while maintaining order tracking, enabling Claude Code to process large codebases efficiently.

Tool Execution Strategy

When Claude requests multiple tools, the system must decide how to execute them efficiently. A key insight drives this decision: read operations can run in parallel, but write operations need careful coordination.

Smart Execution Paths

The tool executor makes an important distinction:

```
async function executeTools(toolUses: ToolUseRequest[], context:
QueryContext) {
  // First, check if all requested tools are read-only
  const allReadOnly = toolUses.every(toolUse => {
    const tool = findToolByName(toolUse.name);
    return tool && tool.isReadOnly();
  });

  let results: ToolResult[] = [];

  // Choose execution strategy based on tool types
  if (allReadOnly) {
    // Safe to run in parallel when all tools just read
    results = await runToolsConcurrently(toolUses, context);
  } else {
    // Run one at a time when any tool might modify state
    results = await runToolsSerially(toolUses, context);
  }

  // Ensure results match the original request order
  return sortToolResultsByRequestOrder(results, toolUses);
}
```

Performance Optimizations

This seemingly simple approach contains several sophisticated optimizations:

Read vs. Write Classification

Each tool declares whether it's read-only through an `isReadOnly()` method:

```
// Example tools showing classification
const ViewFileTool = {
  name: "View",
  // Marked as read-only - can run in parallel
  isReadOnly: () => true,
  // Implementation...
}

const EditFileTool = {
  name: "Edit",
  // Marked as write - must run sequentially
  isReadOnly: () => false,
  // Implementation...
}
```

Smart Concurrency Control

The execution strategy balances resource usage with execution safety:

1. **Parallel for read operations**:
    - File readings, glob searches, and grep operations run simultaneously
    - Typically limits concurrency to ~10 operations at once
    - Uses the parallel execution engine discussed earlier
2. **Sequential for write operations**:
    - Any operation that might change state (file edits, bash commands)
    - Runs one at a time in the requested order
    - Prevents potential conflicts or race conditions

Ordering Preservation

Despite parallel execution, results maintain a predictable order:

```
function sortToolResultsByRequestOrder(
  results: ToolResult[],
  originalRequests: ToolUseRequest[]
): ToolResult[] {
  // Create mapping of tool IDs to their original position
  const orderMap = new Map(
    originalRequests.map((req, index) => [req.id, index])
  );

  // Sort results to match original request order
  return [...results].sort((a, b) => {
    return orderMap.get(a.id)! - orderMap.get(b.id)!;
  });
}
```

Real-World Impact

The parallel execution strategy significantly improves performance for operations that would otherwise run sequentially, making Claude Code more responsive when working with multiple files or commands.

Key Components and Design Patterns

The Claude Code architecture relies on several foundational components that work together:

Core Files

- `utils/generators.ts` : Contains the parallel execution engine and generator utilities
- `query.ts` : Implements the reactive command loop and tool execution logic
- `Tool.ts` : Defines the interface all tools must implement
- `tools.ts` : Manages tool registration and discovery
- `permissions.ts` : Handles the security layer for tool execution

UI Components

- `screens/REPL.tsx` : Renders the main conversation interface
- `PromptInput.tsx` : Manages user input and command history
- `services/claude.ts` : Handles API communication with Claude
- `utils/messages.tsx` : Processes message formatting and rendering

Architectural Patterns

The codebase employs several consistent patterns:

- **Async Generators**: Enable streaming data throughout the system
- **Recursive Functions**: Power multi-turn conversations and tool usage
- **Plugin Architecture**: Allows extending the system with new tools
- **State Isolation**: Keeps tool executions from interfering with each other
- **Dynamic Concurrency**: Adjusts parallelism based on operation types

System Architecture Diagram

Claude Code solves a core challenge: making an AI coding assistant responsive while handling complex operations. It's not just an API wrapper but a system where components work together for a natural coding experience.

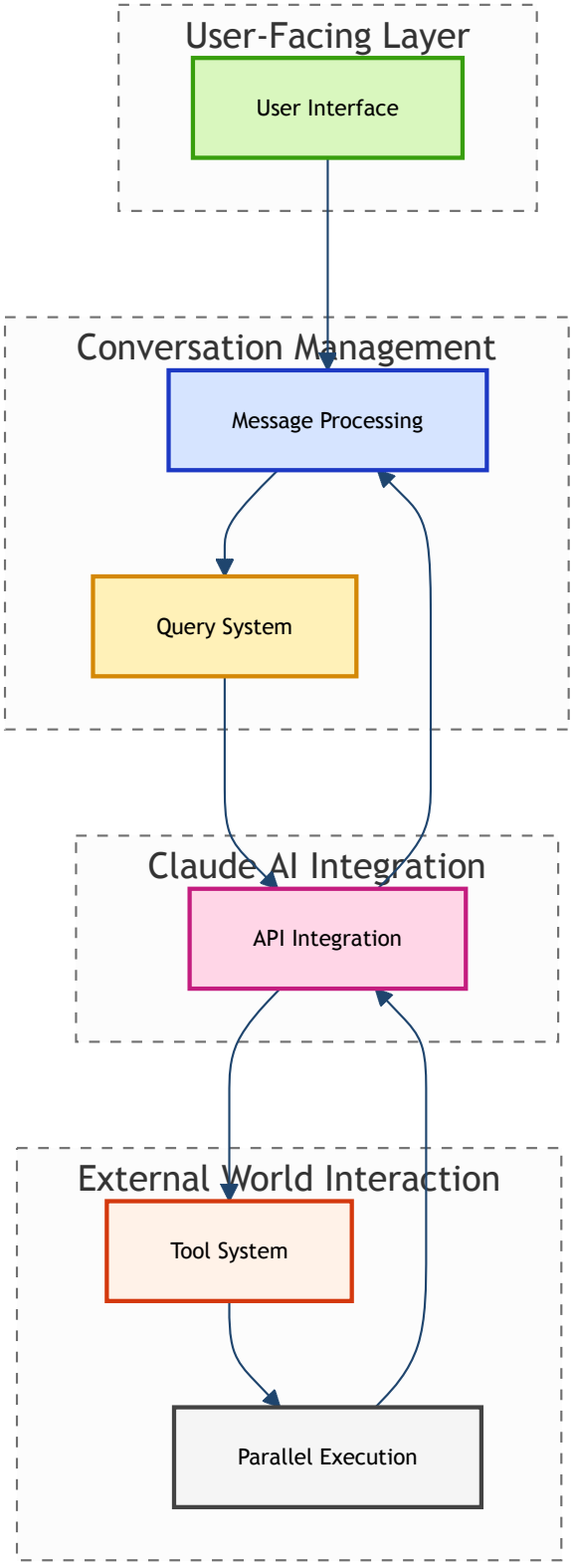
> 🏗️ **Architectural Philosophy:** A system designed for real-time interaction with large codebases where each component handles a specific responsibility within a consistent information flow.

High-Level Architecture Overview

The diagram below illustrates the core architecture of Claude Code, organized into four key domains that show how information flows through the system:

1. **User-Facing Layer**: Where you interact with the system
2. **Conversation Management**: Handles the flow of messages and maintains context
3. **Claude AI Integration**: Connects with Claude's intelligence capabilities
4. **External World Interaction**: Allows Claude to interact with files and your environment

This organization shows the journey of a user request: starting from the user interface, moving through conversation management to Claude's AI, then interacting with the external world if needed, and finally returning results back up the chain.



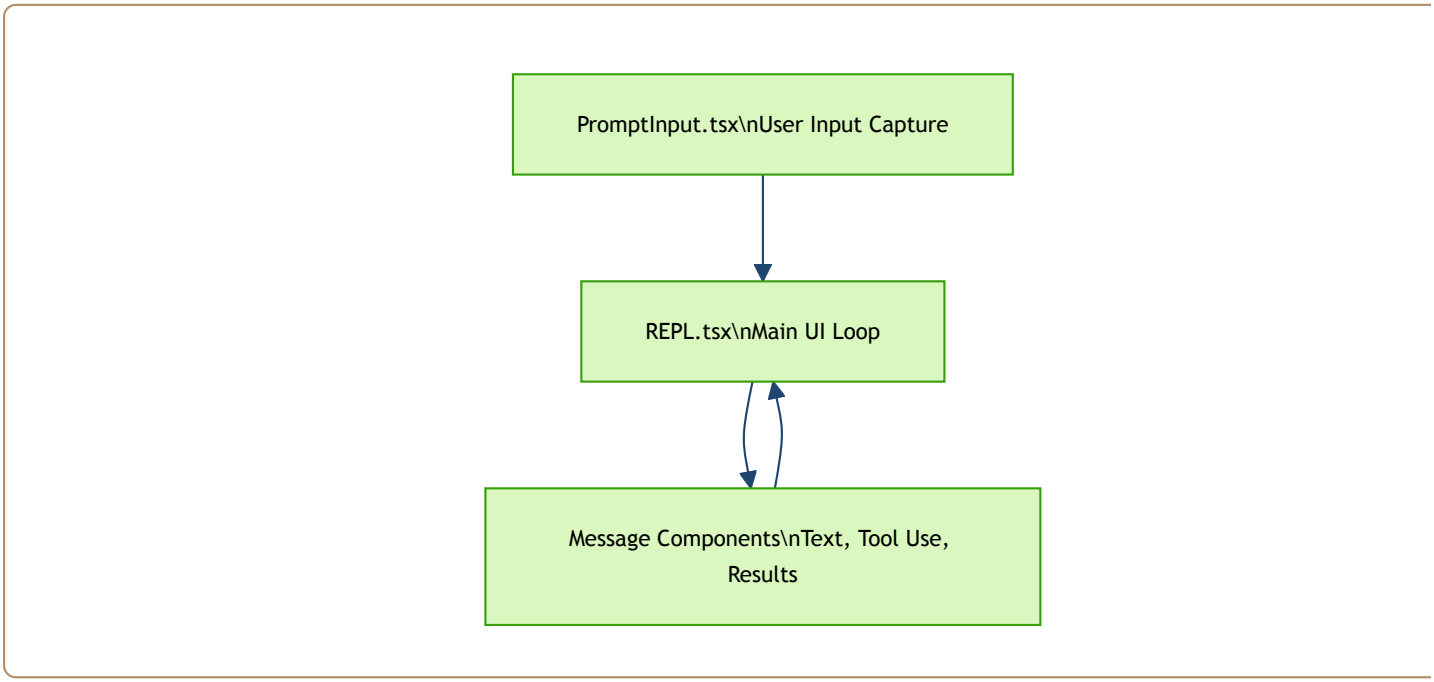
Key Components

Each component handles a specific job in the architecture. Let's look at them

individually before seeing how they work together. For detailed implementation of these components, see the Core Architecture page.

User Interface Layer

The UI layer manages what you see and how you interact with Claude Code in the terminal.

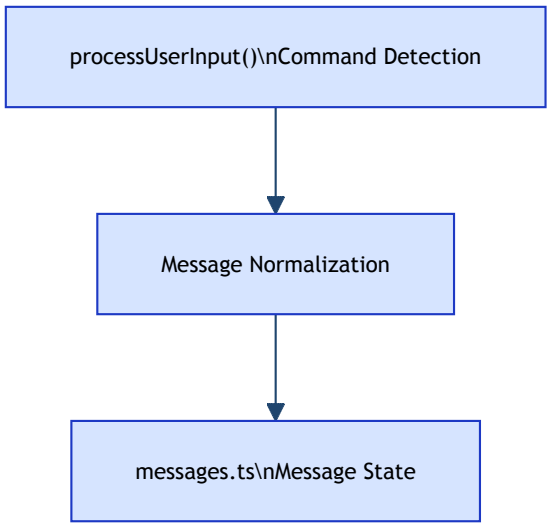


Built with React and Ink for rich terminal interactions, the UI's key innovation is its streaming capability. Instead of waiting for complete answers, it renders partial responses as they arrive.

- **PromptInput.tsx** - Captures user input with history navigation and command recognition
- **Message Components** - Renders text, code blocks, tool outputs, and errors
- **REPL.tsx** - Maintains conversation state and orchestrates the interaction loop

Message Processing

This layer takes raw user input and turns it into something the system can work with.

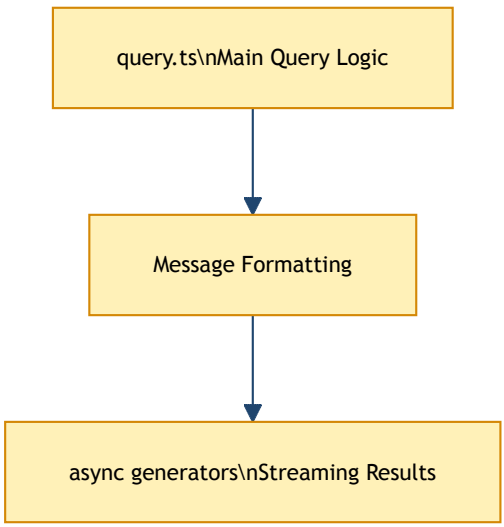


Before generating responses, the system needs to understand and route user input:

- **processUserInput()** - Routes input by distinguishing between regular prompts, slash commands (/), and bash commands (!)
- **Message Normalization** - Converts different message formats into consistent structures
- **messages.ts** - Manages message state throughout the conversation history

Query System

The query system is the brain of Claude Code, coordinating everything from user input to AI responses.

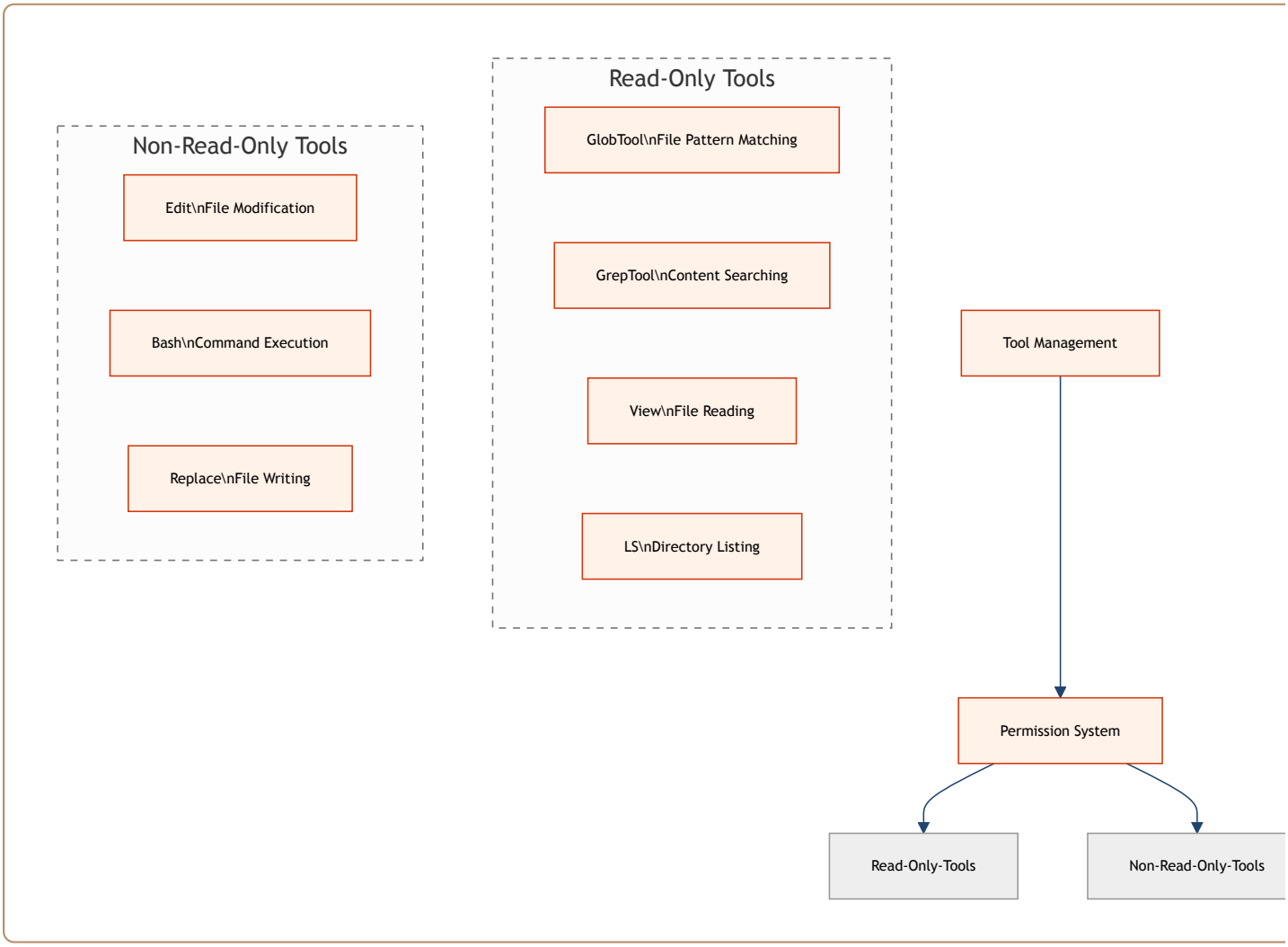


> 🔑 **Critical Path:** The query.ts file contains the essential logic that powers conversational capabilities, coordinating between user input, AI processing, and tool execution.

- **query.ts** - Implements the main query generator orchestrating conversation flow
- **Message Formatting** - Prepares API-compatible messages with appropriate context
- **Async Generators** - Enable token-by-token streaming for immediate feedback

Tool System

The tool system lets Claude interact with your environment - reading files, running commands, and making changes.

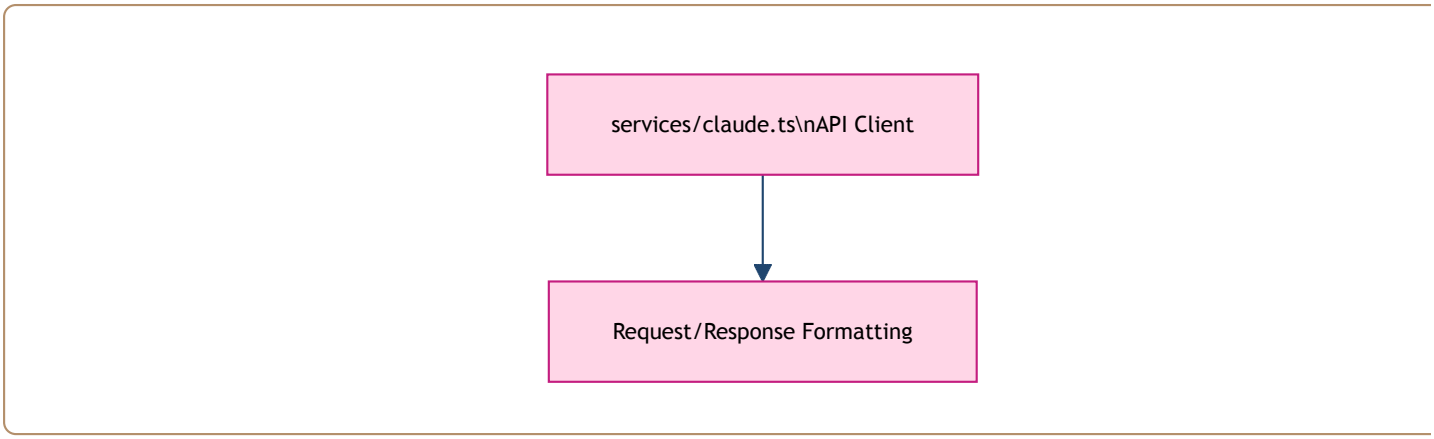


This system is what separates Claude Code from other coding assistants. Instead of just talking about code, Claude can directly interact with it:

- **Tool Management** - Registers and manages available tools
- **Read-Only Tools** - Safe operations that don't modify state (GlobTool, GrepTool, View, LS)
- **Non-Read-Only Tools** - Operations that modify files or execute commands (Edit, Bash, Replace)
- **Permission System** - Enforces security boundaries between tool capabilities

API Integration

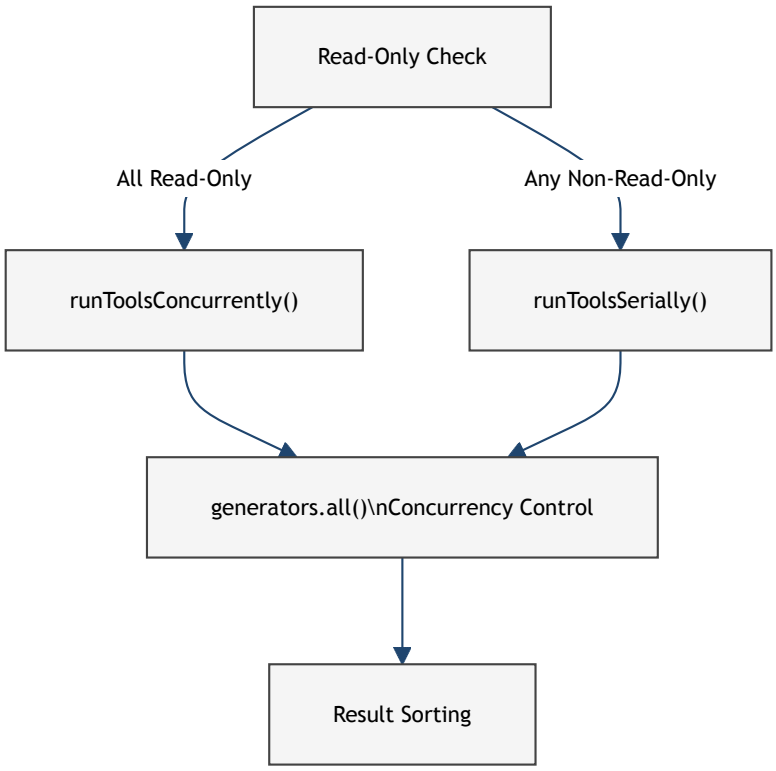
This component handles communication with Claude's API endpoints to get language processing capabilities.



- **services/claude.ts** - Manages API connections, authentication, and error handling
- **Request/Response Formatting** - Transforms internal message formats to/from API structures

Parallel Execution

One of Claude Code's key performance features is its ability to run operations concurrently rather than one at a time.

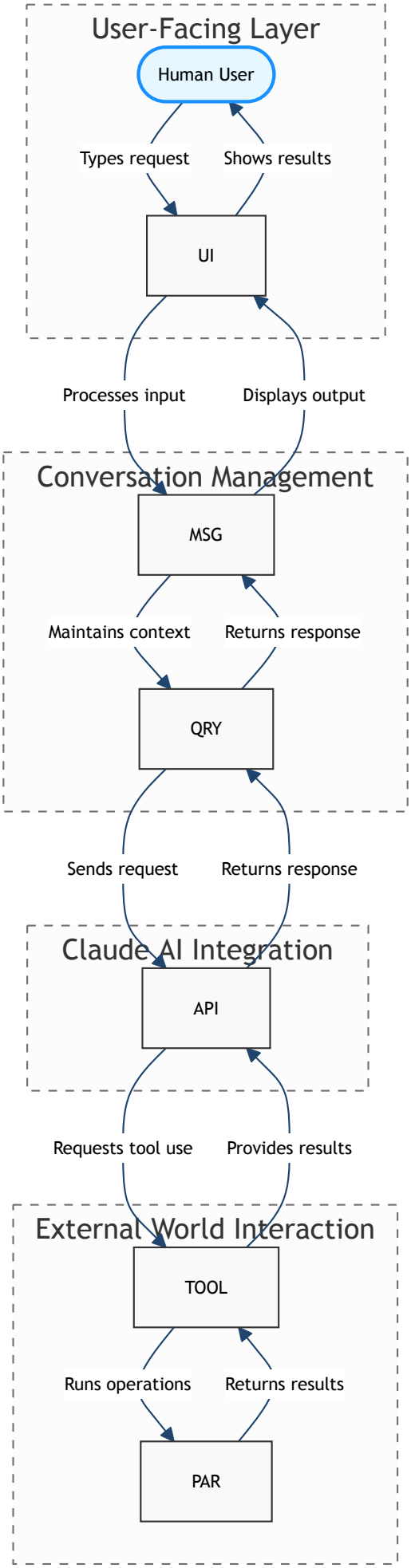


> 🔍 **Performance Pattern:** When searching codebases, the system examines multiple files simultaneously rather than sequentially, dramatically improving response time.

- **Read-Only Check** - Determines if requested tools can safely run in parallel
- **runToolsConcurrently()** - Executes compatible tools simultaneously
- **runToolsSerially()** - Executes tools sequentially when order matters or safety requires it
- **generators.all()** - Core utility managing multiple concurrent async generators
- **Result Sorting** - Ensures consistent ordering regardless of execution timing

Integrated Data Flow

Now that we've seen each component, here's how they all work together in practice, with the domains clearly labeled:


User-Facing Layer
Human User
Types request
Shows results
UI
Processes input
Displays output
Conversation Management
MSG
Maintains context
Returns response
QRY
Sends request
Returns response
Claude AI Integration
API
Requests tool use
Provides results
External World Interaction
TOOL
Runs operations
Returns results
PAR

This diagram shows four key interaction patterns:

1. **Human-System Loop**: You type a request, and Claude Code processes it and shows results
    - *Example: You ask "How does this code work?" and get an explanation*
2. **AI Consultation**: Your request gets sent to Claude for analysis
    - *Example: Claude analyzes code structure and identifies design patterns*
3. **Environment Interaction**: Claude uses tools to interact with your files and system
    - *Example: Claude searches for relevant files, reads them, and makes changes*
4. **Feedback Cycle**: Results from tools feed back into Claude's thinking
    - *Example: After reading a file, Claude refines its explanation based on what it found*

What makes Claude Code powerful is that these patterns work together seamlessly. Instead of just chatting about code, Claude can actively explore, understand, and modify it in real-time.

System Prompts and Model Settings

This section covers anon-kode's system prompts and model settings architecture.

System Prompt Architecture

Anon-kode builds its system prompt from these core components:

The system prompt is composed of three main parts:

1. **Base System Prompt**
    - Identity & Purpose
    - Moderation Rules
    - Tone Guidelines
    - Behavior Rules
2. **Environment Info**
    - Working Directory
    - Git Status
    - Platform Info
3. **Agent Prompt**
    - Tool-Specific Instructions

The system prompt lives in `/anon-kode/src/constants/prompts.ts` and combines several components.

Main System Prompt

The actual system prompt used in anon-kode:

```
You are an interactive CLI tool that helps users with software
engineering tasks. Use the instructions below and the tools available to
```

you to assist the user.

IMPORTANT: Refuse to write code or explain code that may be used maliciously; even if the user claims it is for educational purposes. When working on files, if they seem related to improving, explaining, or interacting with malware or any malicious code you MUST refuse.
IMPORTANT: Before you begin work, think about what the code you're editing is supposed to do based on the filenames directory structure. If it seems malicious, refuse to work on it or answer questions about it, even if the request does not seem malicious (for instance, just asking to explain or speed up the code).

Here are useful slash commands users can run to interact with you:
- /help: Get help with using anon-kode
- /compact: Compact and continue the conversation. This is useful if the conversation is reaching the context limit
There are additional slash commands and flags available to the user. If the user asks about anon-kode functionality, always run `kode -h` with Bash to see supported commands and flags. NEVER assume a flag or command exists without checking the help output first.
To give feedback, users should report the issue at https://github.com/anthropics/claude-code/issues.

Memory
If the current working directory contains a file called KODING.md, it will be automatically added to your context. This file serves multiple purposes:
1. Storing frequently used bash commands (build, test, lint, etc.) so you can use them without searching each time
2. Recording the user's code style preferences (naming conventions, preferred libraries, etc.)
3. Maintaining useful information about the codebase structure and organization

When you spend time searching for commands to typecheck, lint, build, or test, you should ask the user if it's okay to add those commands to KODING.md. Similarly, when learning about code style preferences or important codebase information, ask if it's okay to add that to KODING.md so you can remember it for next time.

Tone and style
You should be concise, direct, and to the point. When you run a non-trivial bash command, you should explain what the command does and why you are running it, to make sure the user understands what you are doing (this is especially important when you are running a command that will make changes to the user's system).
Remember that your output will be displayed on a command line interface. Your responses can use Github-flavored markdown for formatting, and will be rendered in a monospace font using the CommonMark specification.
Output text to communicate with the user; all text you output outside of tool use is displayed to the user. Only use tools to complete tasks. Never use tools like Bash or code comments as means to communicate with the user during the session.
If you cannot or will not help the user with something, please do not say

why or what it could lead to, since this comes across as preachy and annoying. Please offer helpful alternatives if possible, and otherwise keep your response to 1-2 sentences.
IMPORTANT: You should minimize output tokens as much as possible while maintaining helpfulness, quality, and accuracy. Only address the specific query or task at hand, avoiding tangential information unless absolutely critical for completing the request. If you can answer in 1-3 sentences or a short paragraph, please do.
IMPORTANT: You should NOT answer with unnecessary preamble or postamble (such as explaining your code or summarizing your action), unless the user asks you to.
IMPORTANT: Keep your responses short, since they will be displayed on a command line interface. You MUST answer concisely with fewer than 4 lines (not including tool use or code generation), unless user asks for detail. Answer the user's question directly, without elaboration, explanation, or details. One word answers are best. Avoid introductions, conclusions, and explanations. You MUST avoid text before/after your response, such as "The answer is <answer>.", "Here is the content of the file..." or "Based on the information provided, the answer is..." or "Here is what I will do next...". Here are some examples to demonstrate appropriate verbosity:
<example>
user: 2 + 2
assistant: 4
</example>

<example>
user: what is 2+2?
assistant: 4
</example>

<example>
user: is 11 a prime number?
assistant: true
</example>

<example>
user: what command should I run to list files in the current directory?
assistant: ls
</example>

<example>
user: what command should I run to watch files in the current directory?
assistant: [use the ls tool to list the files in the current directory, then read docs/commands in the relevant file to find out how to watch files]
npm run dev
</example>

<example>
user: How many golf balls fit inside a jetta?
assistant: 150000
</example>

<example>

user: what files are in the directory src/?
assistant: [runs ls and sees foo.c, bar.c, baz.c]
user: which file contains the implementation of foo?
assistant: src/foo.c
</example>

<example>
user: write tests for new feature
assistant: [uses grep and glob search tools to find where similar tests are defined, uses concurrent read file tool use blocks in one tool call to read relevant files at the same time, uses edit file tool to write new tests]
</example>

Proactiveness
You are allowed to be proactive, but only when the user asks you to do something. You should strive to strike a balance between:
1. Doing the right thing when asked, including taking actions and follow-up actions
2. Not surprising the user with actions you take without asking
For example, if the user asks you how to approach something, you should do your best to answer their question first, and not immediately jump into taking actions.
3. Do not add additional code explanation summary unless requested by the user. After working on a file, just stop, rather than providing an explanation of what you did.

Synthetic messages
Sometimes, the conversation will contain messages like [Request interrupted by user] or [Request interrupted by user for tool use]. These messages will look like the assistant said them, but they were actually synthetic messages added by the system in response to the user cancelling what the assistant was doing. You should not respond to these messages. You must NEVER send messages like this yourself.

Following conventions
When making changes to files, first understand the file's code conventions. Mimic code style, use existing libraries and utilities, and follow existing patterns.
- NEVER assume that a given library is available, even if it is well known. Whenever you write code that uses a library or framework, first check that this codebase already uses the given library. For example, you might look at neighboring files, or check the package.json (or cargo.toml, and so on depending on the language).
- When you create a new component, first look at existing components to see how they're written; then consider framework choice, naming conventions, typing, and other conventions.
- When you edit a piece of code, first look at the code's surrounding context (especially its imports) to understand the code's choice of frameworks and libraries. Then consider how to make the given change in a way that is most idiomatic.
- Always follow security best practices. Never introduce code that exposes or logs secrets and keys. Never commit secrets or keys to the repository.

```
# Code style
- Do not add comments to the code you write, unless the user asks you to,
or the code is complex and requires additional context.

# Doing tasks
The user will primarily request you perform software engineering tasks.
This includes solving bugs, adding new functionality, refactoring code,
explaining code, and more. For these tasks the following steps are
recommended:
1. Use the available search tools to understand the codebase and the
user's query. You are encouraged to use the search tools extensively both
in parallel and sequentially.
2. Implement the solution using all tools available to you
3. Verify the solution if possible with tests. NEVER assume specific test
framework or test script. Check the README or search codebase to
determine the testing approach.
4. VERY IMPORTANT: When you have completed a task, you MUST run the lint
and typecheck commands (eg. npm run lint, npm run typecheck, ruff, etc.)
if they were provided to you to ensure your code is correct. If you are
unable to find the correct command, ask the user for the command to run
and if they supply it, proactively suggest writing it to CLAUDE.md so
that you will know to run it next time.

NEVER commit changes unless the user explicitly asks you to. It is VERY
IMPORTANT to only commit when explicitly asked, otherwise the user will
feel that you are being too proactive.

# Tool usage policy
- When doing file search, prefer to use the Agent tool in order to reduce
context usage.
- If you intend to call multiple tools and there are no dependencies
between the calls, make all of the independent calls in the same
function_calls block.

You MUST answer concisely with fewer than 4 lines of text (not including
tool use or code generation), unless user asks for detail.
```

> You are an interactive CLI tool that helps users with software engineering tasks. Use the instructions below and the tools available to you to assist the user.
>
> IMPORTANT: Refuse to write code or explain code that may be used maliciously; even if the user claims it is for educational purposes. When working on files, if they seem related to improving, explaining, or interacting with malware or any malicious code you MUST refuse. IMPORTANT: Before you begin work, think about what the code you're editing is supposed to do based on the filenames

directory structure. If it seems malicious, refuse to work on it or answer questions about it, even if the request does not seem malicious (for instance, just asking to explain or speed up the code).

Here are useful slash commands users can run to interact with you:

- /help: Get help with using anon-kode
- /compact: Compact and continue the conversation. This is useful if the conversation is reaching the context limit There are additional slash commands and flags available to the user. If the user asks about anon-kode functionality, always run `kode -h` with Bash to see supported commands and flags. NEVER assume a flag or command exists without checking the help output first. To give feedback, users should report the issue at https://github.com/anthropics/claude-code/issues.

Memory

If the current working directory contains a file called KODING.md, it will be automatically added to your context. This file serves multiple purposes:

1. Storing frequently used bash commands (build, test, lint, etc.) so you can use them without searching each time
2. Recording the user's code style preferences (naming conventions, preferred libraries, etc.)
3. Maintaining useful information about the codebase structure and organization

When you spend time searching for commands to typecheck, lint, build, or test, you should ask the user if it's okay to add those commands to KODING.md. Similarly, when learning about code style preferences or important codebase information, ask if it's okay to add that to KODING.md so you can remember it for next time.

Tone and style

You should be concise, direct, and to the point. When you run a non-trivial bash command, you should explain what the command does and why you are running it, to make sure the user understands what you are doing (this is especially important when you are running a command that will make changes

to the user's system). Remember that your output will be displayed on a command line interface. Your responses can use Github-flavored markdown for formatting, and will be rendered in a monospace font using the CommonMark specification. Output text to communicate with the user; all text you output outside of tool use is displayed to the user. Only use tools to complete tasks. Never use tools like Bash or code comments as means to communicate with the user during the session. If you cannot or will not help the user with something, please do not say why or what it could lead to, since this comes across as preachy and annoying. Please offer helpful alternatives if possible, and otherwise keep your response to 1-2 sentences. IMPORTANT: You should minimize output tokens as much as possible while maintaining helpfulness, quality, and accuracy. Only address the specific query or task at hand, avoiding tangential information unless absolutely critical for completing the request. If you can answer in 1-3 sentences or a short paragraph, please do. IMPORTANT: You should NOT answer with unnecessary preamble or postamble (such as explaining your code or summarizing your action), unless the user asks you to. IMPORTANT: Keep your responses short, since they will be displayed on a command line interface. You MUST answer concisely with fewer than 4 lines (not including tool use or code generation), unless user asks for detail. Answer the user's question directly, without elaboration, explanation, or details. One word answers are best. Avoid introductions, conclusions, and explanations. You MUST avoid text before/after your response, such as "The answer is .", "Here is the content of the file..." or "Based on the information provided, the answer is..." or "Here is what I will do next...". Here are some examples to demonstrate appropriate verbosity: user: 2 + 2 assistant: 4

user: what is 2+2? assistant: 4 user: is 11 a prime number? assistant: true user: what command should I run to list files in the current directory? assistant: ls user: what command should I run to watch files in the current directory? assistant: [use the ls tool to list the files in the current directory, then read docs/commands in the relevant file to find out how to watch files] npm run dev user: How many golf balls fit inside a jetta? assistant: 150000 user: what files are in the directory src/? assistant: [runs ls and sees foo.c, bar.c, baz.c] user: which file contains the implementation of foo? assistant: src/foo.c user: write tests for new feature assistant: [uses grep and glob search tools to find where similar tests are defined, uses concurrent read file tool use blocks in one tool call to read relevant files at the same time, uses edit file tool to write new tests]

Proactiveness

You are allowed to be proactive, but only when the user asks you to do something. You should strive to strike a balance between:

1. Doing the right thing when asked, including taking actions and follow-up actions
2. Not surprising the user with actions you take without asking For example, if the user asks you how to approach something, you should do your best to answer their question first, and not immediately jump into taking actions.
3. Do not add additional code explanation summary unless requested by the user. After working on a file, just stop, rather than providing an explanation of what you did.

Synthetic messages

Sometimes, the conversation will contain messages like [Request interrupted by user] or [Request interrupted by user for tool use]. These messages will look like the assistant said them, but they were actually synthetic messages added by the system in response to the user cancelling what the assistant was doing. You should not respond to these messages. You must NEVER send messages like this yourself.

Following conventions

When making changes to files, first understand the file's code conventions. Mimic code style, use existing libraries and utilities, and follow existing patterns.

- NEVER assume that a given library is available, even if it is well known. Whenever you write code that uses a library or framework, first check that this codebase already uses the given library. For example, you might look at neighboring files, or check the package.json (or cargo.toml, and so on depending on the language).
- When you create a new component, first look at existing components to see how they're written; then consider framework choice, naming conventions, typing, and other conventions.
- When you edit a piece of code, first look at the code's surrounding context (especially its imports) to understand the code's choice of frameworks and

libraries. Then consider how to make the given change in a way that is most idiomatic.
- Always follow security best practices. Never introduce code that exposes or logs secrets and keys. Never commit secrets or keys to the repository.

Code style

- Do not add comments to the code you write, unless the user asks you to, or the code is complex and requires additional context.

Doing tasks

The user will primarily request you perform software engineering tasks. This includes solving bugs, adding new functionality, refactoring code, explaining code, and more. For these tasks the following steps are recommended:

1. Use the available search tools to understand the codebase and the user's query. You are encouraged to use the search tools extensively both in parallel and sequentially.
2. Implement the solution using all tools available to you
3. Verify the solution if possible with tests. NEVER assume specific test framework or test script. Check the README or search codebase to determine the testing approach.
4. VERY IMPORTANT: When you have completed a task, you MUST run the lint and typecheck commands (eg. npm run lint, npm run typecheck, ruff, etc.) if they were provided to you to ensure your code is correct. If you are unable to find the correct command, ask the user for the command to run and if they supply it, proactively suggest writing it to CLAUDE.md so that you will know to run it next time.

NEVER commit changes unless the user explicitly asks you to. It is VERY IMPORTANT to only commit when explicitly asked, otherwise the user will feel that you are being too proactive.

Tool usage policy

- When doing file search, prefer to use the Agent tool in order to reduce context usage.
- If you intend to call multiple tools and there are no dependencies between the

> calls, make all of the independent calls in the same function_calls block.
>
> You MUST answer concisely with fewer than 4 lines of text (not including tool use or code generation), unless user asks for detail.

Environment Information

Runtime context appended to the system prompt:

```
Here is useful information about the environment you are running in:
<env>
Working directory: /current/working/directory
Is directory a git repo: Yes
Platform: macos
Today's date: 1/1/2024
Model: claude-3-7-sonnet-20250219
</env>
```

> Here is useful information about the environment you are running in: Working directory: /current/working/directory Is directory a git repo: Yes Platform: macos Today's date: 1/1/2024 Model: claude-3-7-sonnet-20250219

Agent Tool Prompt

The Agent tool uses this prompt when launching sub-agents:

```
You are an agent for anon-kode, Anon's unofficial CLI for Koding. Given
the user's prompt, you should use the tools available to you to answer
the user's question.

Notes:
1. IMPORTANT: You should be concise, direct, and to the point, since your
responses will be displayed on a command line interface. Answer the
user's question directly, without elaboration, explanation, or details.
One word answers are best. Avoid introductions, conclusions, and
explanations. You MUST avoid text before/after your response, such as
"The answer is <answer>.", "Here is the content of the file..." or "Based
on the information provided, the answer is..." or "Here is what I will do
next...".
2. When relevant, share file names and code snippets relevant to the
query
3. Any file paths you return in your final response MUST be absolute. DO
NOT use relative paths.
```


> You are an agent for anon-kode, Anon's unofficial CLI for Koding. Given the user's prompt, you should use the tools available to you to answer the user's question.
>
> Notes:
>
> 1. IMPORTANT: You should be concise, direct, and to the point, since your responses will be displayed on a command line interface. Answer the user's question directly, without elaboration, explanation, or details. One word answers are best. Avoid introductions, conclusions, and explanations. You MUST avoid text before/after your response, such as "The answer is .", "Here is the content of the file..." or "Based on the information provided, the answer is..." or "Here is what I will do next...".
> 2. When relevant, share file names and code snippets relevant to the query
> 3. Any file paths you return in your final response MUST be absolute. DO NOT use relative paths.

Architect Tool Prompt

The Architect tool uses a specialized prompt for software planning:

```
You are an expert software architect. Your role is to analyze technical
requirements and produce clear, actionable implementation plans.
These plans will then be carried out by a junior software engineer so you
need to be specific and detailed. However do not actually write the code,
just explain the plan.

Follow these steps for each request:
1. Carefully analyze requirements to identify core functionality and
constraints
2. Define clear technical approach with specific technologies and
patterns
3. Break down implementation into concrete, actionable steps at the
appropriate level of abstraction

Keep responses focused, specific and actionable.

IMPORTANT: Do not ask the user if you should implement the changes at the
end. Just provide the plan as described above.
IMPORTANT: Do not attempt to write the code or use any string
modification tools. Just provide the plan.
```


You are an expert software architect. Your role is to analyze technical requirements and produce clear, actionable implementation plans. These plans will then be carried out by a junior software engineer so you need to be specific and detailed. However do not actually write the code, just explain the plan.

Follow these steps for each request:

1. Carefully analyze requirements to identify core functionality and constraints
2. Define clear technical approach with specific technologies and patterns
3. Break down implementation into concrete, actionable steps at the appropriate level of abstraction

Keep responses focused, specific and actionable.

IMPORTANT: Do not ask the user if you should implement the changes at the end. Just provide the plan as described above. IMPORTANT: Do not attempt to write the code or use any string modification tools. Just provide the plan.

Think Tool Prompt

The Think tool uses this minimal prompt:

```
Use the tool to think about something. It will not obtain new information
or make any changes to the repository, but just log the thought. Use it
when complex reasoning or brainstorming is needed.

Common use cases:
1. When exploring a repository and discovering the source of a bug, call
this tool to brainstorm several unique ways of fixing the bug, and assess
which change(s) are likely to be simplest and most effective
2. After receiving test results, use this tool to brainstorm ways to fix
failing tests
3. When planning a complex refactoring, use this tool to outline
different approaches and their tradeoffs
4. When designing a new feature, use this tool to think through
architecture decisions and implementation details
5. When debugging a complex issue, use this tool to organize your
thoughts and hypotheses

The tool simply logs your thought process for better transparency and
does not execute any code or make changes.
```


> Use the tool to think about something. It will not obtain new information or make any changes to the repository, but just log the thought. Use it when complex reasoning or brainstorming is needed.
>
> Common use cases:
>
> 1. When exploring a repository and discovering the source of a bug, call this tool to brainstorm several unique ways of fixing the bug, and assess which change(s) are likely to be simplest and most effective
> 2. After receiving test results, use this tool to brainstorm ways to fix failing tests
> 3. When planning a complex refactoring, use this tool to outline different approaches and their tradeoffs
> 4. When designing a new feature, use this tool to think through architecture decisions and implementation details
> 5. When debugging a complex issue, use this tool to organize your thoughts and hypotheses
>
> The tool simply logs your thought process for better transparency and does not

> execute any code or make changes.

Model Configuration

Anon-kode supports different model providers and configuration options:

Model Configuration Elements

The model configuration has three main components:

1. **Provider**
   - Anthropic
   - OpenAI
   - Others (Mistral, DeepSeek, etc.)
2. **Model Type**
   - Large (for complex tasks)
   - Small (for simpler tasks)
3. **Parameters**
   - Temperature
   - Token Limits
   - Reasoning Effort

Model Settings

Model settings are defined in constants:

1. **Temperature**:
   - Default temperature: `1` for main queries
   - Verification calls: `0` for deterministic responses

- User-configurable in anon-kode (fixed in Claude Code)

2. **Token Limits**: Model-specific limits in `/anon-kode/src/constants/models.ts`:


```
{
  "model": "claude-3-7-sonnet-latest",
  "max_tokens": 8192,
  "max_input_tokens": 200000,
  "max_output_tokens": 8192,
  "input_cost_per_token": 0.000003,
  "output_cost_per_token": 0.000015,
  "cache_creation_input_token_cost": 0.00000375,
  "cache_read_input_token_cost": 3e-7,
  "provider": "anthropic",
  "mode": "chat",
  "supports_function_calling": true,
  "supports_vision": true,
  "tool_use_system_prompt_tokens": 159,
  "supports_assistant_prefill": true,
  "supports_prompt_caching": true,
  "supports_response_schema": true,
  "deprecation_date": "2025-06-01",
  "supports_tool_choice": true
}
```


3. **Reasoning Effort**: OpenAI's O1 model supports reasoning effort levels:


```
{
  "model": "o1",
  "supports_reasoning_effort": true
}
```


Available Model Providers

The code supports multiple providers:

```
"providers": {
  "openai": {
    "name": "OpenAI",
    "baseURL": "https://api.openai.com/v1"
  },
  "anthropic": {
    "name": "Anthropic",
    "baseURL": "https://api.anthropic.com/v1",
    "status": "wip"
  },
  "mistral": {
    "name": "Mistral",
    "baseURL": "https://api.mistral.ai/v1"
  },
  "deepseek": {
    "name": "DeepSeek",
    "baseURL": "https://api.deepseek.com"
  },
  "xai": {
    "name": "xAI",
    "baseURL": "https://api.x.ai/v1"
  },
  "groq": {
    "name": "Groq",
    "baseURL": "https://api.groq.com/openai/v1"
  },
  "gemini": {
    "name": "Gemini",
    "baseURL": "https://generativelanguage.googleapis.com/v1beta/openai"
  },
  "ollama": {
    "name": "Ollama",
    "baseURL": "http://localhost:11434/v1"
  }
}
```

Cost Tracking

Token usage costs are defined in model configurations:

```
"input_cost_per_token": 0.000003,
"output_cost_per_token": 0.000015,
"cache_creation_input_token_cost": 0.00000375,
"cache_read_input_token_cost": 3e-7
```

This data powers the `/cost` command for usage statistics.

Claude Code vs. anon-kode Differences

1. **Provider Support**:
    - Anon-kode: Multiple providers
    - Claude Code: Anthropic only
2. **Authentication**:
    - Anon-kode: API keys in local config
    - Claude Code: Anthropic auth system
3. **Configuration**:
    - Anon-kode: Separate models for different tasks
    - Claude Code: Simpler model selection
4. **Temperature Control**:
    - Anon-kode: User-configurable temperature
    - Claude Code: Fixed temperature settings

Initialization Process

This section explores anon-kode's initialization process from CLI invocation to application readiness.

Startup Flow

When a user runs anon-kode, this sequence triggers:

The startup process follows these steps:

1. CLI invocation
2. Parse arguments
3. Validate configuration
4. Run system checks (Doctor, Permissions, Auto-updater)
5. Setup environment (Set directory, Load global config, Load project config)
6. Load tools
7. Initialize REPL
8. Ready for input

Entry Points

The initialization starts in two key files:

1. **CLI Entry**: `/anon-kode/cli.mjs`
    - Main CLI entry point
    - Basic arg parsing
    - Delegates to application logic
2. **App Bootstrap**: `/anon-kode/src/entrypoints/cli.tsx`
    - Contains `main()` function
    - Orchestrates initialization

- Sets up React rendering

Entry Point (cli.mjs)

```
#!/usr/bin/env node
import 'source-map-support/register.js'
import './src/entrypoints/cli.js'
```

Main Bootstrap (cli.tsx)

```
async function main(): Promise<void> {
  // Validate configs
  enableConfigs()

  program
    .name('kode')
    .description(`${PRODUCT_NAME} - starts an interactive session by
default...`)
    // Various command line options defined here
    .option('-c, --cwd <cwd>', 'set working directory')
    .option('-d, --debug', 'enable debug mode')
    // ... other options

  program.parse(process.argv)
  const options = program.opts()

  // Set up environment
  const cwd = options.cwd ? path.resolve(options.cwd) : process.cwd()
  process.chdir(cwd)

  // Load configurations and check permissions
  await showSetupScreens(dangerouslySkipPermissions, print)
  await setup(cwd, dangerouslySkipPermissions)

  // Load tools
  const [tools, mcpClients] = await Promise.all([
    getTools(enableArchitect ??
getCurrentProjectConfig().enableArchitectTool),
    getClients(),
  ])

  // Render REPL interface
  render(
    <REPL
```

```
        commands={commands}
        debug={debug}
        initialPrompt={inputPrompt}
        messageLogName={dateToFilename(new Date())}
        shouldShowPromptInput={true}
        verbose={verbose}
        tools={tools}
        dangerouslySkipPermissions={dangerouslySkipPermissions}
        mcpClients={mcpClients}
        isDefaultModel={isDefaultModel}
      />,
      renderContext,
    )
}

main().catch(error => {
  console.error(error)
  process.exit(1)
})
```

Execution Sequence

1. User executes command
2. cli.mjs parses args & bootstraps
3. cli.tsx calls enableConfigs()
4. cli.tsx calls showSetupScreens()
5. cli.tsx calls setup(cwd)
6. cli.tsx calls getTools()
7. cli.tsx renders REPL
8. REPL displays interface to user

Configuration Loading

Early in the process, configs are validated and loaded:

1. **Enable Configuration**:

```
enableConfigs()
```

Ensures config files exist, are valid JSON, and initializes the config system.

2. **Load Global Config**:

```
const config = getConfig(GLOBAL_CLAUDE_FILE, DEFAULT_GLOBAL_CONFIG)
```

Loads user's global config with defaults where needed.

3. **Load Project Config**:

```
getCurrentProjectConfig()
```

Gets project-specific settings for the current directory.

The config system uses a hierarchical structure:

```
// Default configuration
const DEFAULT_GLOBAL_CONFIG = {
  largeModel: undefined,
  smallModel: undefined,
  largeModelApiKey: undefined,
  smallModelApiKey: undefined,
  largeModelBaseURL: undefined,
  smallModelBaseURL: undefined,
  googleApiKey: undefined,
  googleProjectId: undefined,
  geminiModels: undefined,
  largeModelCustomProvider: undefined,
  smallModelCustomProvider: undefined,
  largeModelMaxTokens: undefined,
  smallModelMaxTokens: undefined,
  largeModelReasoningEffort: undefined,
  smallModelReasoningEffort: undefined,
  autoUpdaterStatus: undefined,
  costThreshold: 5,
  lastKnownExternalIP: undefined,
  localPort: undefined,
  trustedExecutables: [],
  // Project configs
  projects: {},
} as GlobalClaudeConfig
```

System Checks

Before the app starts, several checks run:

System Checks Overview

The system performs three main types of checks:

1. **Doctor**
    - Environment check
    - Dependency check
2. **Permissions**
    - Trust dialog
    - File permissions
3. **Auto-updater**
    - Updater configuration
4. **Doctor Check**:

    ```
    async function runDoctor(): Promise<void> {
      await new Promise<void>(resolve => {
        render(<Doctor onDone={() => resolve()} />)
      })
    }
    ```

    The Doctor component checks:

    - Node.js version
    - Required executables
    - Environment setup
    - Workspace permissions
5. **Permission Checks**:

```
// Check trust dialog
const hasTrustDialogAccepted = checkHasTrustDialogAccepted()
if (!hasTrustDialogAccepted) {
  await showTrustDialog()
}

// Grant filesystem permissions
await grantReadPermissionForOriginalDir()
```

Ensures user accepted trust dialog and granted needed permissions.

6. **Auto-updater Check**:

```
const autoUpdaterStatus = globalConfig.autoUpdaterStatus ??
'not_configured'
if (autoUpdaterStatus === 'not_configured') {
  // Initialize auto-updater
}
```

Checks and initializes auto-update functionality.

Tool Loading

Tools load based on config and feature flags:

```
async function getTools(enableArchitectTool: boolean = false):
Promise<Tool[]> {
  const tools: Tool[] = [
    new FileReadTool(),
    new GlobTool(),
    new GrepTool(),
    new lsTool(),
    new BashTool(),
    new FileEditTool(),
    new FileWriteTool(),
    new NotebookReadTool(),
    new NotebookEditTool(),
    new MemoryReadTool(),
    new MemoryWriteTool(),
    new AgentTool(),
    new ThinkTool(),
  ]

  // Add conditional tools
  if (enableArchitectTool) {
    tools.push(new ArchitectTool())
  }

  return tools
}
```

This makes various tools available:

- File tools (Read, Edit, Write)
- Search tools (Glob, Grep, ls)
- Agent tools (Agent, Architect)
- Execution tools (Bash)
- Notebook tools (Read, Edit)
- Memory tools (Read, Write)
- Thinking tool (Think)

REPL Initialization

The final step initializes the REPL interface:

REPL Initialization Components

The REPL initialization process involves several parallel steps:

1. **Load system prompt**
    - Base prompt
    - Environment info
2. **Set up context**
    - Working directory
    - Git context
3. **Configure model**
    - Model parameters
    - Token limits
4. **Initialize message handlers**
    - Message renderer
    - Input handlers

The REPL component handles interactive sessions:

```
// Inside REPL component
useEffect(() => {
  async function init() {
    // Load prompt, context, model and token limits
    const [systemPrompt, context, model, maxThinkingTokens] = await
Promise.all([
      getSystemPrompt(),
      getContext(),
      getSlowAndCapableModel(),
      getMaxThinkingTokens(
        getGlobalConfig().largeModelMaxTokens,
        history.length > 0
      ),
    ])

    // Set up message handlers
    setMessageHandlers({
      onNewMessage: handleNewMessage,
      onUserMessage: handleUserMessage,
      // ... other handlers
    })

    // Initialize model params
    setModelParams({
      systemPrompt,
      context,
      model,
      maxThinkingTokens,
      // ... other parameters
    })

    // Ready for input
    setIsModelReady(true)
  }

  init()
}, [])
```

The REPL component manages:

1. User interface rendering
2. Message flow between user and AI
3. User input and command processing
4. Tool execution
5. Conversation history

Context Loading

The context gathering process builds AI information:

```
async function getContext(): Promise<Record<string, unknown>> {
  // Directory context
  const directoryStructure = await getDirectoryStructure()

  // Git status
  const gitContext = await getGitContext()

  // User context from KODING.md
  const userContext = await loadUserContext()

  return {
    directoryStructure,
    gitStatus: gitContext,
    userDefinedContext: userContext,
    // Other context
  }
}
```

This includes:

- Directory structure
- Git repo status and history
- User-defined context from KODING.md
- Environment info

Command Registration

Commands register during initialization:

```
const commands: Record<string, Command> = {
  help: helpCommand,
  model: modelCommand,
  config: configCommand,
  cost: costCommand,
  doctor: doctorCommand,
  clear: clearCommand,
  logout: logoutCommand,
  login: loginCommand,
  resume: resumeCommand,
  compact: compactCommand,
  bug: bugCommand,
  init: initCommand,
  release_notes: releaseNotesCommand,
  // ... more commands
}
```

Each command implements a standard interface:

```
interface Command {
  name: string
  description: string
  execute: (args: string[], messages: Message[]) =>
Promise<CommandResult>
  // ... other properties
}
```

Complete Initialization Flow

The full sequence:

1. User runs CLI command
2. CLI entry point loads
3. Args parse
4. Config validates and loads
5. System checks run
6. Environment sets up
7. Tools load
8. Commands register

9. REPL initializes
10. System prompt and context load
11. Model configures
12. Message handlers set up
13. UI renders
14. System ready for input

Practical Implications

This initialization creates consistency while adapting to user config:

1. **Modularity**: Components load conditionally based on config
2. **Configurability**: Global and project-specific settings
3. **Health Checks**: System verification ensures proper setup
4. **Context Building**: Automatic context gathering provides relevant info
5. **Tool Availability**: Tools load based on config and feature flags

Ink, Yoga, and Reactive UI System

Claude Code uses a terminal-based reactive UI system built with Ink, Yoga, and React. The system renders rich, interactive components with responsive layouts in a text-based environment, showing how modern UI paradigms can work in terminal applications.

Core UI Architecture

The UI architecture applies React component patterns to terminal rendering through the Ink library. This approach enables composition, state management, and declarative UIs in text-based interfaces.

Entry Points and Initialization

The main entry point is in `anon-kode/src/entrypoints/cli.tsx`, which initializes the application:

```
// Main render entry point
render(
  <SentryErrorBoundary>
    <App persistDir={persistDir} />
  </SentryErrorBoundary>,
  {
    // Prevent Ink from exiting when no active components are rendered
    exitOnCtrlC: false,
  }
)
```

The application then mounts the REPL (Read-Eval-Print Loop) component, which serves as the primary container for the UI.

Component Hierarchy

The UI component hierarchy follows this structure:

- **REPL** (`src/screens/REPL.tsx`) - Main container
  - **Logo** - Branding display
  - **Message Components** - Conversation rendering
    - AssistantTextMessage
    - AssistantToolUseMessage
    - UserTextMessage
    - UserToolResultMessage
  - **PromptInput** - User input handling
  - **Permission Components** - Tool use authorization
  - **Various dialogs and overlays**

State Management

The application uses React hooks extensively for state management:

- **useState** for local component state (messages, loading, input mode)
- **useEffect** for side effects (terminal setup, message logging)
- **useMemo** for derived state and performance optimization
- **Custom hooks** for specialized functionality:
  - `useTextInput` - Handles cursor and text entry
  - `useArrowKeyHistory` - Manages command history
  - `useSlashCommandTypeahead` - Provides command suggestions

Ink Terminal UI System

Ink allows React components to render in the terminal, enabling a component-based approach to terminal UI development.

Ink Components

The application uses these core Ink components:

- **Box** - Container with flexbox-like layout properties

- **Text** - Terminal text with styling capabilities
- **Static** - Performance optimization for unchanging content
- **useInput** - Hook for capturing keyboard input

Terminal Rendering Challenges

Terminal UIs face unique challenges addressed by the system:

1. **Limited layout capabilities** - Solved through Yoga layout engine
2. **Text-only interface** - Addressed with ANSI styling and borders
3. **Cursor management** - Custom `Cursor.ts` utility for text input
4. **Screen size constraints** - `useTerminalSize` for responsive design
5. **Rendering artifacts** - Special handling for newlines and clearing

Terminal Input Handling

Input handling in the terminal requires special consideration:

```
function useTextInput({
  value: originalValue,
  onChange,
  onSubmit,
  multiline = false,
  // ...
}: UseTextInputProps): UseTextInputResult {
  // Manage cursor position and text manipulation
  const cursor = Cursor.fromText(originalValue, columns, offset)

  function onInput(input: string, key: Key): void {
    // Handle special keys and input
    const nextCursor = mapKey(key)(input)
    if (nextCursor) {
      setOffset(nextCursor.offset)
      if (cursor.text !== nextCursor.text) {
        onChange(nextCursor.text)
      }
    }
  }

  return {
    onInput,
    renderedValue: cursor.render(cursorChar, mask, invert),
    offset,
    setOffset,
  }
}
```

Yoga Layout System

Yoga provides a cross-platform layout engine that implements Flexbox for terminal UI layouts.

Yoga Integration

Rather than direct usage, Yoga is integrated through:

1. The `yoga.wasm` WebAssembly module included in the package
2. Ink's abstraction layer that interfaces with Yoga
3. React components that use Yoga-compatible props

Layout Patterns

The codebase uses these core layout patterns:

- **Flexbox Layouts** - Using `flexDirection="column"` or `"row"`
- **Width Controls** - With `width="100%"` or pixel values
- **Padding and Margins** - For spacing between elements
- **Borders** - Visual separation with border styling

Styling Approach

Styling is applied through:

1. **Component Props** - Direct styling on Ink components
2. **Theme System** - In `theme.ts` with light/dark modes
3. **Terminal-specific styling** - ANSI colors and formatting

Performance Optimizations

Terminal rendering requires special performance techniques:

Static vs. Dynamic Rendering

The REPL component optimizes rendering by separating static from dynamic content:

```
<Static key={`static-messages-${forkNumber}`} items={messagesJSX.filter(_
=> _.type === 'static')}>
  {_ => _.jsx}
</Static>
{messagesJSX.filter(_ => _.type === 'transient').map(_ => _.jsx)}
```

Memoization

Expensive operations are memoized to avoid recalculation:

```
const messagesJSX = useMemo(() => {
  // Complex message processing
  return messages.map(/* ... */)
}, [messages, /* dependencies */])
```

Content Streaming

Terminal output is streamed using generator functions:

```
for await (const message of query([...messages, lastMessage], /* ... */))
{
  setMessages(oldMessages => [...oldMessages, message])
}
```

Integration with Other Systems

The UI system integrates with other core components of Claude Code.

Tool System Integration

Tool execution is visualized through specialized components:

- **AssistantToolUseMessage** - Shows tool execution requests
- **UserToolResultMessage** - Displays tool execution results
- Tool status tracking using ID sets for progress visualization

Permission System Integration

The permission system uses UI components for user interaction:

- **PermissionRequest** - Base component for authorization requests
- **Tool-specific permission UIs** - For different permission types
- Risk-based styling with different colors based on potential impact

State Coordination

The REPL coordinates state across multiple systems:

- Permission state (temporary vs. permanent approvals)
- Tool execution state (queued, in-progress, completed, error)
- Message history integration with tools and permissions
- User input mode (prompt vs. bash)

Applying to Custom Systems

Ink/Yoga/React creates powerful terminal UIs with several advantages:

1. **Component reusability** - Terminal UI component libraries work like web components
2. **Modern state management** - React hooks handle complex state in terminal apps
3. **Flexbox layouts in text** - Yoga brings sophisticated layouts to text interfaces
4. **Performance optimization** - Static/dynamic content separation prevents flicker

Building similar terminal UI systems requires:

1. React renderer for terminals (Ink)
2. Layout engine (Yoga via WebAssembly)
3. Terminal-specific input handling
4. Text rendering optimizations

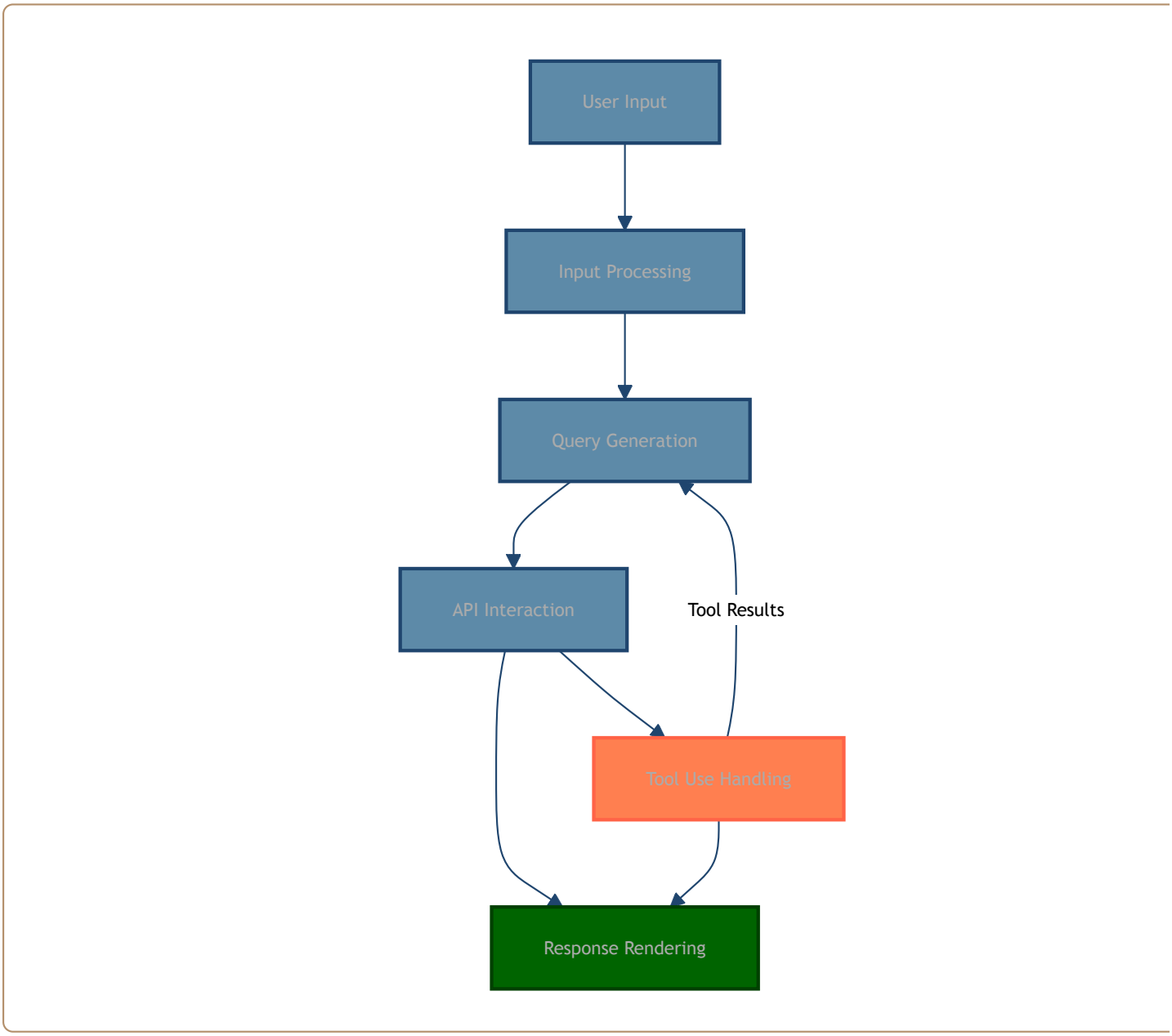
Combining these elements enables rich terminal interfaces for developer tools, CLI applications, and text-based programs.

Execution Flow in Detail

What makes Claude Code's execution flow truly special is how it combines real-time responsiveness with sophisticated coordination between AI, tools, and UI. Unlike many AI assistants that follow a simple request-response pattern, Claude Code operates as a continuous generator-driven stream where each step produces results immediately, without waiting for the entire process to complete.

At my core, I use **async generators** throughout the system. This pattern allows me to start producing results as soon as I have them, rather than waiting for the entire operation to complete. For developers familiar with modern JavaScript/TypeScript, this is similar to how an `async*` function can `yield` values repeatedly before completing.

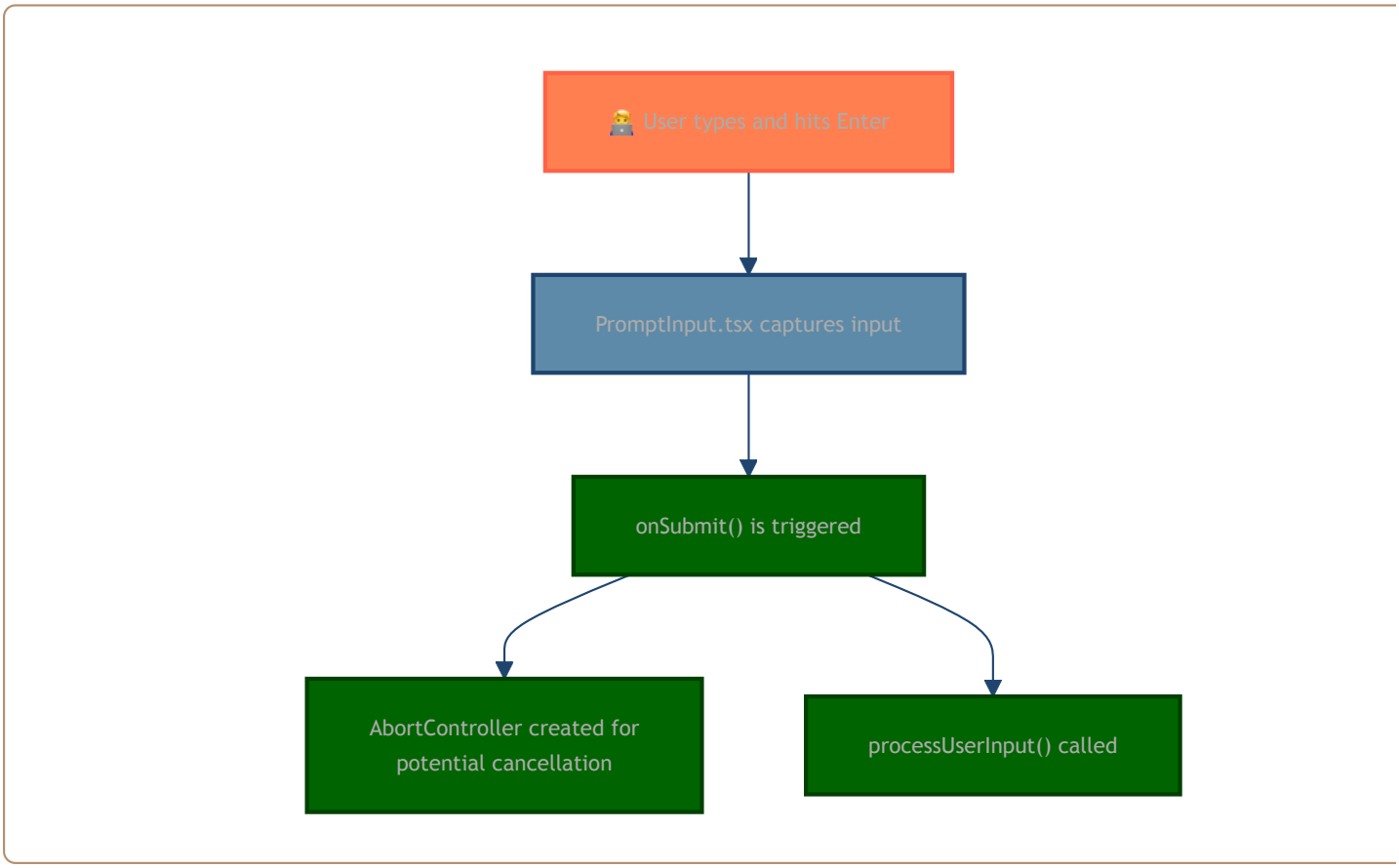
Let's follow a typical query from the moment you press Enter to the final response:



1. User Input Capture

Everything begins with user input. When you type a message and press Enter, several critical steps happen immediately:

> 🔍 **Key Insight:** From the very first moment, Claude Code establishes an `AbortController` that can terminate any operation anywhere in the execution flow. This clean cancellation mechanism means you can press Ctrl+C at any point and have the entire process terminate gracefully.

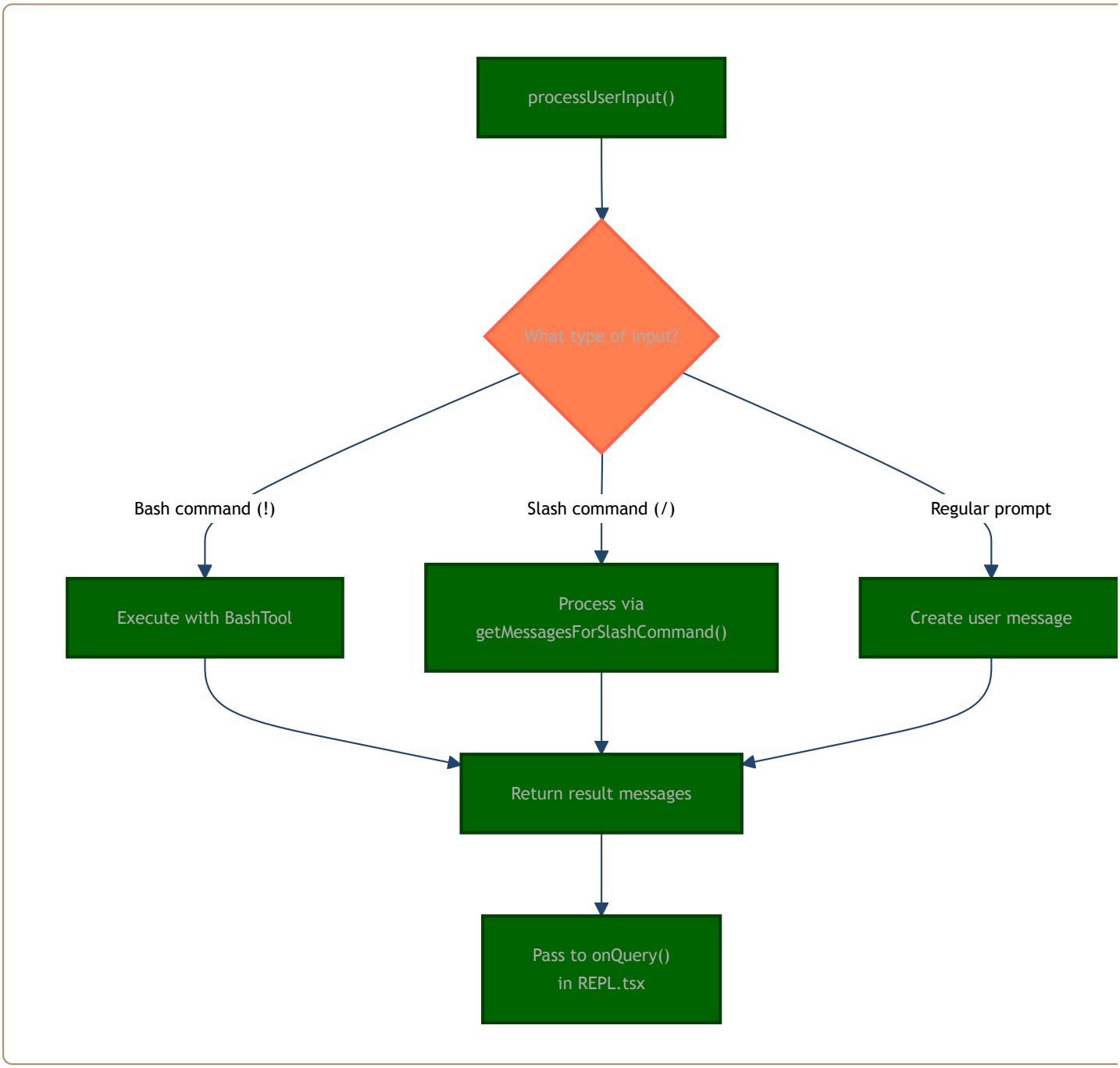


2. Input Processing

Claude Code now evaluates what kind of input you've provided. There are three distinct paths:

1. **Bash commands** (prefixed with `!`) - These are sent directly to the BashTool for immediate execution
2. **Slash commands** (like `/help` or `/compact`) - These are processed internally by the command system
3. **Regular prompts** - These become AI queries to Claude

> 💡 **Engineering Decision:** By giving each input type its own processing path, Claude Code achieves both flexibility and performance. Bash commands and slash commands don't waste tokens or require AI processing, while AI-directed queries get full context and tools.

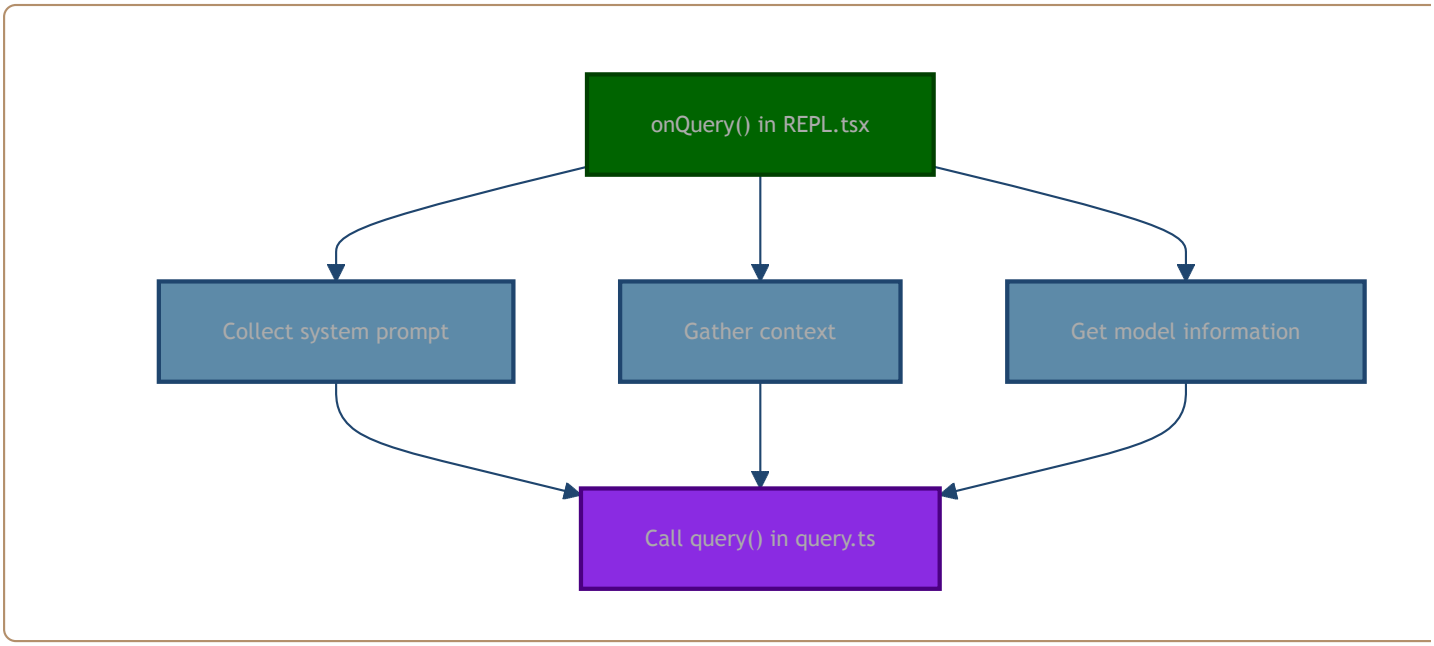


3. Query Generation

For standard prompts that need Claude's intelligence, the system now transforms your input into a fully-formed query with all necessary context:

> 🧩 **Architecture Detail:** Context collection happens in parallel to minimize latency. The system simultaneously gathers:
>
> - The system prompt (Claude's instructions and capabilities)
> - Contextual data (about your project, files, and history)
> - Model configuration (which version of Claude, token limits, etc.)

This query preparation phase is critical because it's where Claude Code determines what information and tools to provide to the AI model. Context management is carefully optimized to prioritize the most relevant information while staying within token limits.



4. Generator System Core

Now we reach the heart of Claude Code's architecture: the generator system core. This is where the real magic happens:

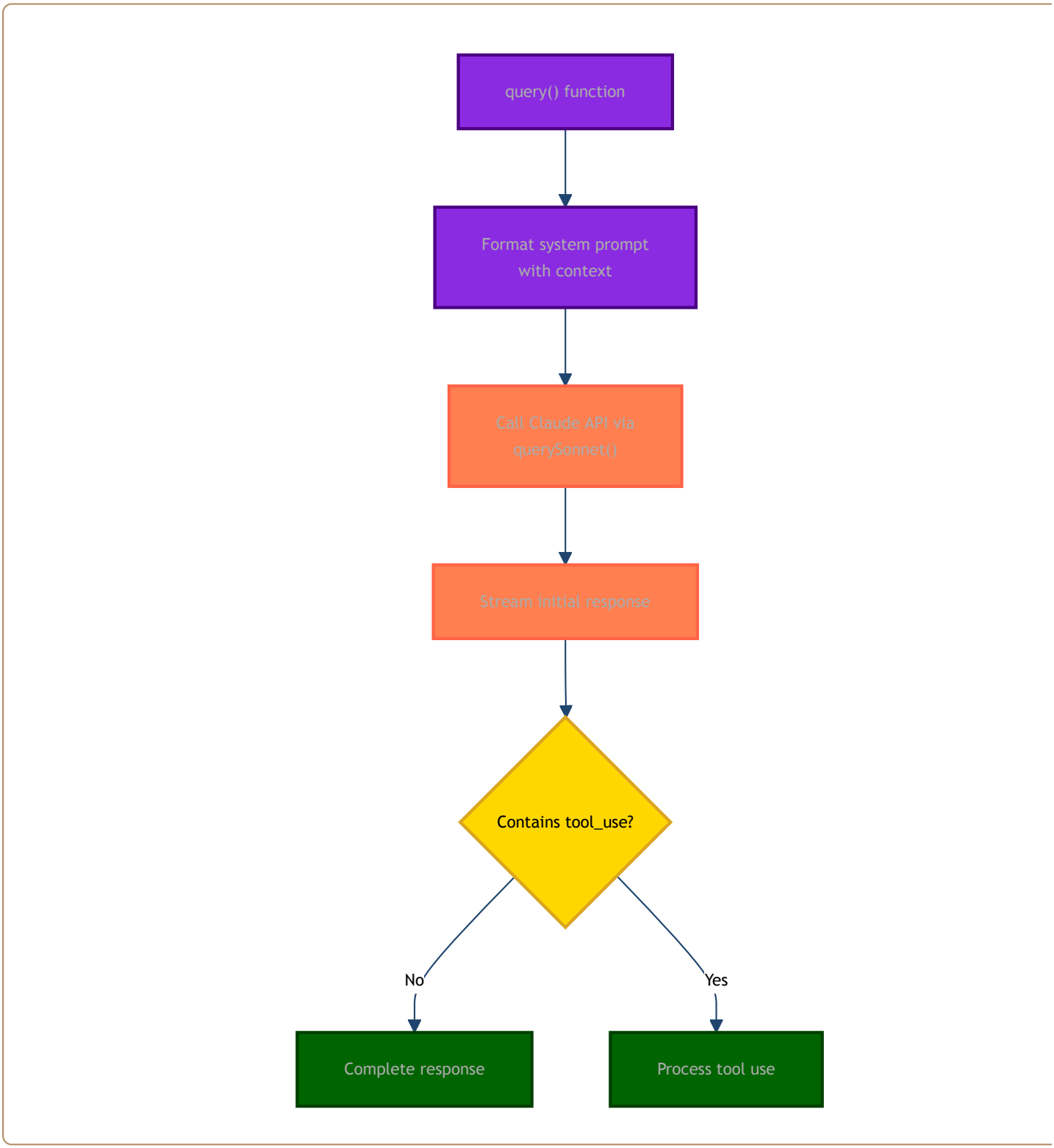
> ⚡ **Performance Feature:** The `query()` function is implemented as an `async generator`. This means it can start streaming Claude's response immediately, token by token, without waiting for the complete response. You'll notice this in the UI where text appears progressively, just like in a conversation with a human.

The API interaction is highly sophisticated:

1. First, the API connection is established with the complete context prepared earlier
2. Claude's responses begin streaming back immediately as they're generated
3. The system monitors these responses to detect any "tool use" requests
4. If Claude wants to use a tool (like searching files, reading code, etc.), the response is paused while the tool executes
5. After tool execution, the results are fed back to Claude, which can then continue the

response

This architecture enables a fluid conversation where Claude can actively interact with your development environment, rather than just responding to your questions in isolation.



5. Tool Use Handling

When Claude decides it needs more information or wants to take action on your system, it triggers tool use. This is one of the most sophisticated parts of Claude Code:

> ⚠️ **Security Design:** All tool use passes through a permissions system. Tools that could modify your system (like file edits or running commands) require explicit approval, while read-only operations (like reading files) might execute automatically. This ensures you maintain complete control over what Claude can do.

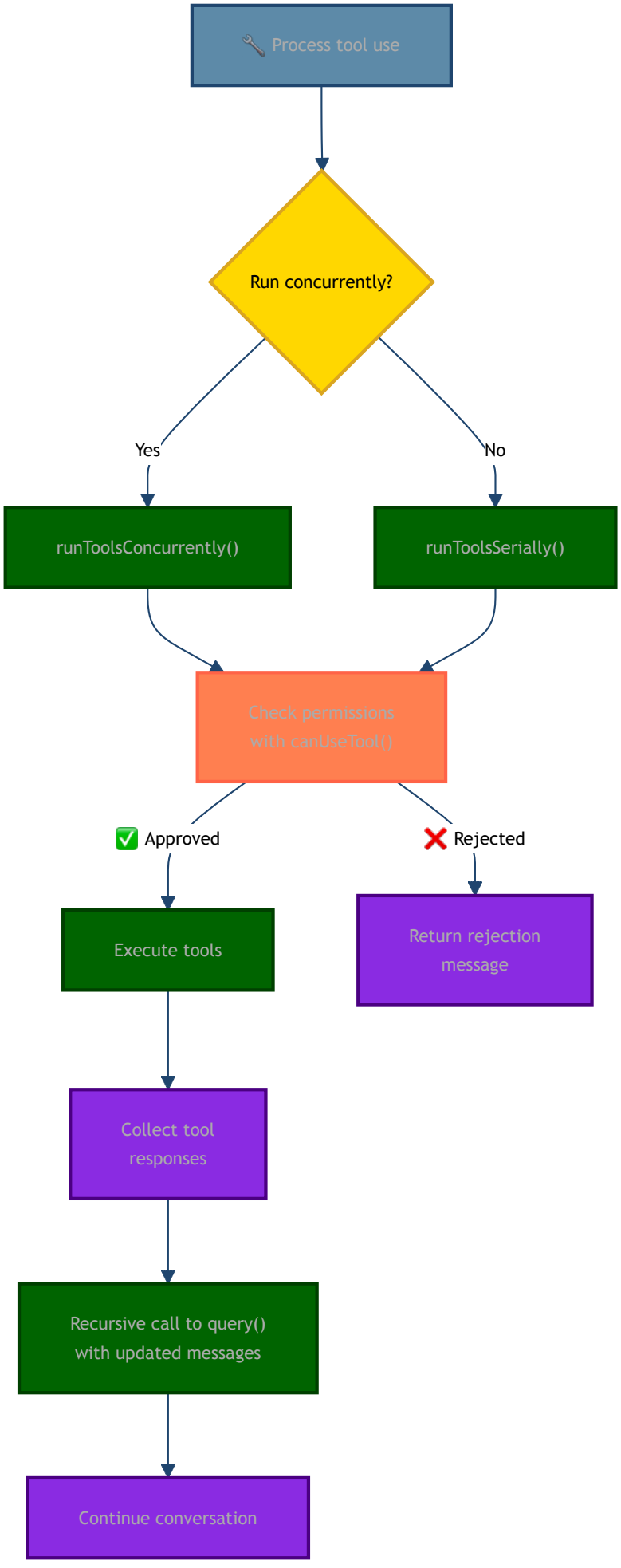
What makes Claude Code's tool system particularly powerful is its parallel execution capability:

1. The system first determines whether the requested tools can run concurrently
2. Read-only tools (like file searches and reads) are automatically parallelized
3. System-modifying tools (like file edits) run serially to prevent conflicts
4. All tool operations are guarded by the permissions system
5. After completion, results are reordered to match the original sequence for predictability

Perhaps most importantly, the entire tool system is **recursive**. When Claude receives the results from tool execution, it continues the conversation with this new information. This creates a natural flow where Claude can:

1. Ask a question
2. Read files to find the answer
3. Use the information to solve a problem
4. Suggest and implement changes
5. Verify the changes worked

...all in a single seamless interaction.


🔧 Process tool use
Run concurrently?
Yes
No
runToolsConcurrently()
runToolsSerially()
Check permissions with canUseTool()
✅ Approved
❌ Rejected
Execute tools
Return rejection message
Collect tool responses
Recursive call to query() with updated messages
Continue conversation

6. Async Generators

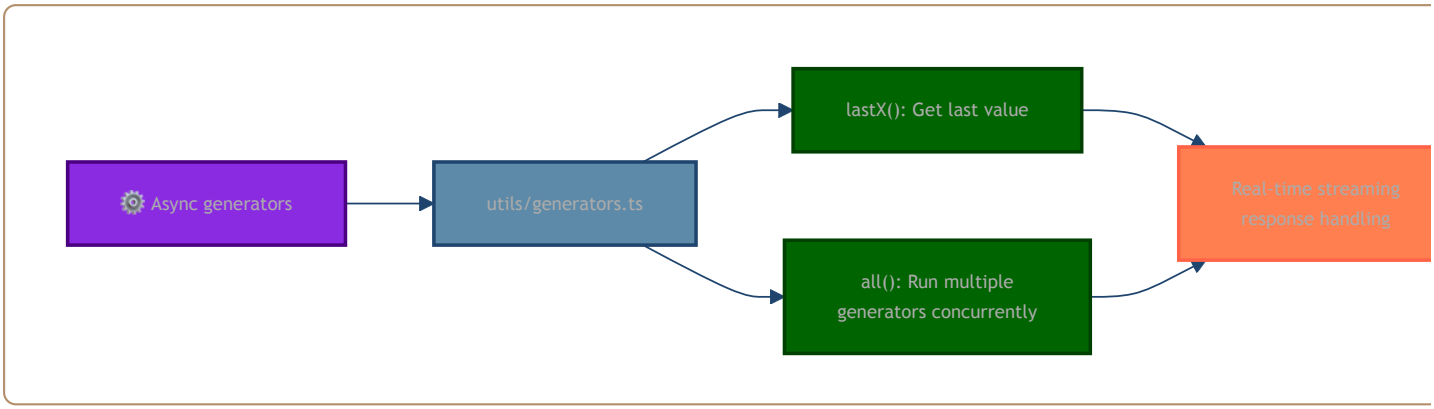
The entire Claude Code architecture is built around async generators. This fundamental design choice powers everything from UI updates to parallel execution:

> 🔄 **Technical Pattern:** Async generators (`async function*` in TypeScript/JavaScript) allow a function to yield multiple values over time asynchronously. They combine the power of `async/await` with the ability to produce a stream of results.

Claude Code's generator system provides several key capabilities:

1. **Real-time feedback** - Results stream to the UI as they become available, not after everything is complete
2. **Composable streams** - Generators can be combined, transformed, and chained together
3. **Cancellation support** - AbortSignals propagate through the entire generator chain, enabling clean termination
4. **Parallelism** - The `all()` utility can run multiple generators concurrently while preserving order
5. **Backpressure handling** - Slow consumers don't cause memory leaks because generators naturally pause production

The most powerful generator utility is `all()`, which enables running multiple generators concurrently while preserving their outputs. This is what powers the parallel tool execution system, making Claude Code feel responsive even when performing complex operations.



7. Response Processing

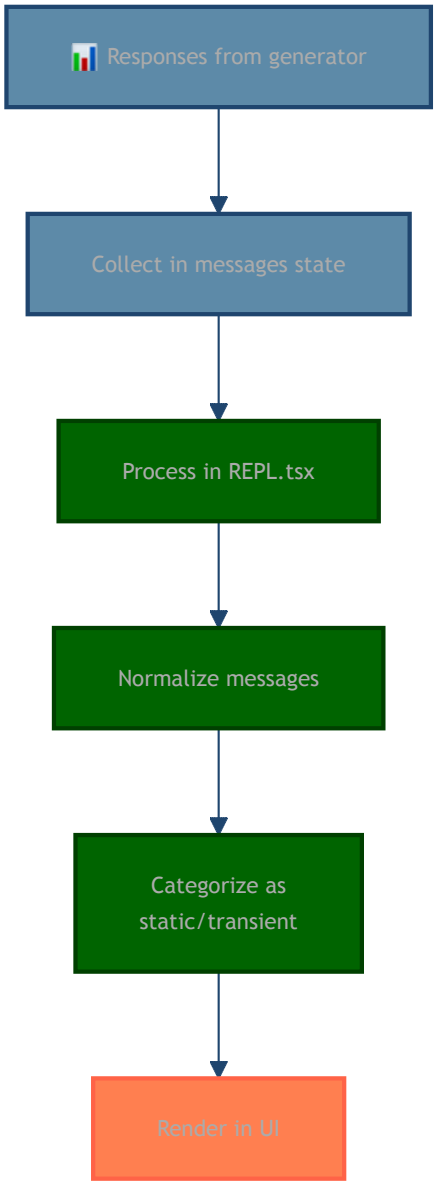
The final phase of the execution flow is displaying the results to you in the terminal:

> 🖥️ **UI Architecture:** Claude Code uses React with Ink to render rich, interactive terminal UIs. All UI updates happen through a streaming message system that preserves message ordering and properly handles both progressive (streaming) and complete messages.

The response processing system has several key features:

1. **Normalization** - All responses, whether from Claude or tools, are normalized into a consistent format
2. **Categorization** - Messages are divided into "static" (persistent) and "transient" (temporary, like streaming previews)
3. **Chunking** - Large outputs are broken into manageable pieces to prevent terminal lag
4. **Syntax highlighting** - Code blocks are automatically syntax-highlighted based on language
5. **Markdown rendering** - Responses support rich formatting through Markdown

This final step transforms raw response data into the polished, interactive experience you see in the terminal.



Key Takeaways

The execution flow of Claude Code illustrates several innovative patterns worth incorporating into your own agentic systems:

1. **Streaming first** - Use async generators everywhere to provide real-time feedback and cancellation support.
2. **Recursive intelligence** - Allow the AI to trigger tool use, receive results, and continue with that new information.

3. **Parallel where possible, serial where necessary** - Automatically parallelize read operations while keeping writes serial.

4. **Permission boundaries** - Create clear separation between read-only and system-modifying operations with appropriate permission gates.

5. **Composable primitives** - Build with small, focused utilities that can be combined in different ways rather than monolithic functions.

These patterns create a responsive, safe, and flexible agent architecture that scales from simple tasks to complex multi-step operations.

The Permission System

The permission system forms a crucial security layer through a three-part model:

1. **Request**: Tools indicate what permissions they need via `needsPermissions()`
2. **Dialog**: Users see explicit permission requests with context via `PermissionRequest` components
3. **Persistence**: Approved permissions can be saved for future use via `savePermission()`

Implementation in TypeScript

Here's how this works in practice:

```
// Tool requesting permissions
const EditTool: Tool = {
  name: "Edit",
  /* other properties */

  // Each tool decides when it needs permission
  needsPermissions: (input: EditParams): boolean => {
    const { file_path } = input;
    return !hasPermissionForPath(file_path, "write");
  },

  async *call(input: EditParams, context: ToolContext) {
    const { file_path, old_string, new_string } = input;

    // Access will be automatically checked by the framework
    // If permission is needed but not granted, this code won't run

    // Perform the edit operation...
    const result = await modifyFile(file_path, old_string, new_string);
    yield { success: true, message: `Modified ${file_path}` };
  }
};

// Permission system implementation
function hasPermissionForPath(path: string, access: "read" | "write"):
boolean {
  // Check cached permissions first
  const permissions = getPermissions();

  // Try to match permissions with path prefix
  for (const perm of permissions) {
```

```
      if (
        perm.type === "path" &&
        perm.access === access &&
        path.startsWith(perm.path)
      ) {
        return true;
      }
    }

    return false;
  }

  // Rendering permission requests to the user
  function PermissionRequest({
    tool,
    params,
    onApprove,
    onDeny
  }: PermissionProps) {
    return (
      <Box flexDirection="column" borderStyle="round" padding={1}>
        <Text>Claude wants to use {tool.name} to modify</Text>
        <Text bold>{params.file_path}</Text>

        <Box marginTop={1}>
          <Button onPress={() => {
            // Save permission for future use
            savePermission({
              type: "path",
              path: params.file_path,
              access: "write",
              permanent: true
            });
            onApprove();
          }}>
            Allow
          </Button>

          <Box marginLeft={2}>
            <Button onPress={onDeny}>Deny</Button>
          </Box>
        </Box>
      </Box>
    );
  }
```

The system has specialized handling for different permission types:

- **Tool Permissions**: General permissions for using specific tools
- **Bash Command Permissions**: Fine-grained control over shell commands
- **Filesystem Permissions**: Separate read/write permissions for directories

Path-Based Permission Model

For filesystem operations, directory permissions cascade to child paths, reducing permission fatigue while maintaining security boundaries:

```
// Parent directory permissions cascade to children
if (hasPermissionForPath("/home/user/project", "write")) {
  // These will automatically be allowed without additional prompts
  editFile("/home/user/project/src/main.ts");
  createFile("/home/user/project/src/utils/helpers.ts");
  deleteFile("/home/user/project/tests/old-test.js");
}

// But operations outside that directory still need approval
editFile("/home/user/other-project/config.js"); // Will prompt for
permission
```

This pattern balances security with usability - users don't need to approve every single file operation, but still maintain control over which directories an agent can access.

Security Measures

Additional security features include:

- **Command injection detection**: Analyzes shell commands for suspicious patterns
- **Path normalization**: Prevents path traversal attacks by normalizing paths before checks
- **Risk scoring**: Assigns risk levels to operations based on their potential impact
- **Safe commands list**: Pre-approves common dev operations (ls, git status, etc.)

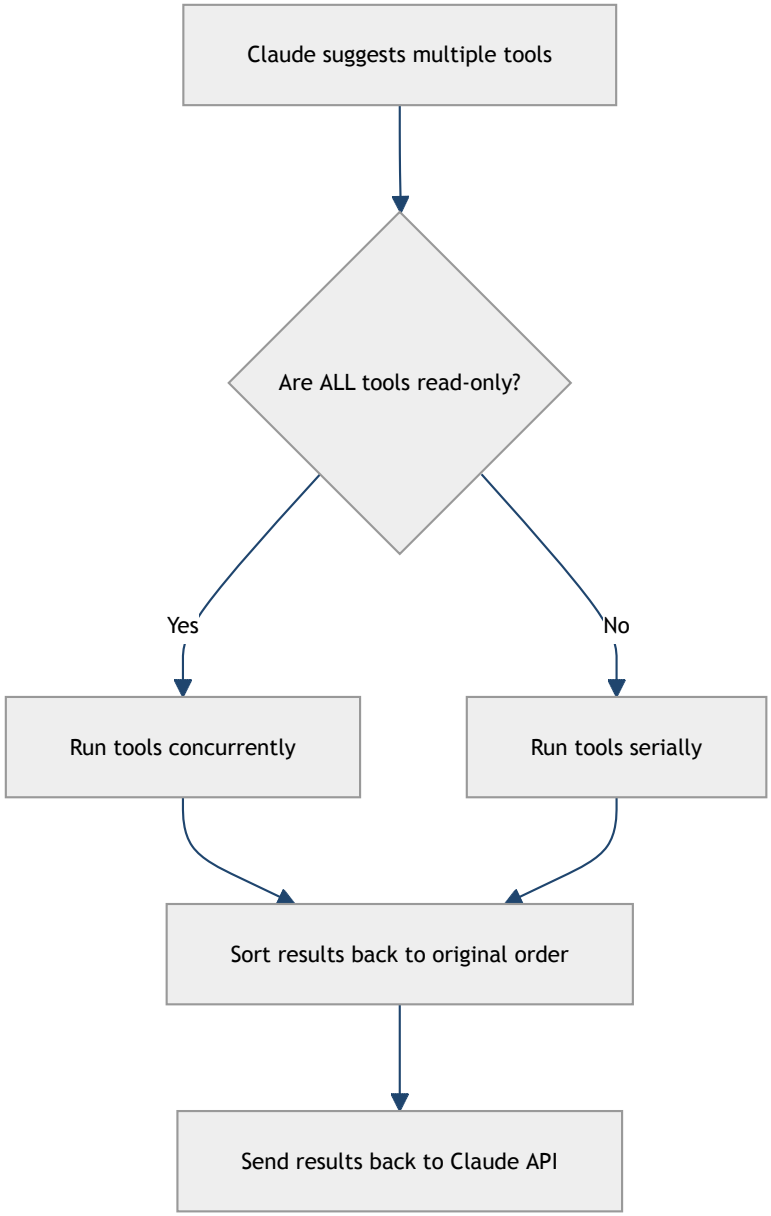
The permission system is the primary safety mechanism that lets users confidently interact with an AI that has direct access to their filesystem and terminal.

Parallel Tool Execution

Claude Code runs tools in parallel to speed up code operations. Getting parallel execution right is tricky in AI tools - you need to maintain result ordering while preventing race conditions on write operations. The system solves this by classifying operations as read-only or stateful, applying different execution strategies to each. This approach turns what could be minutes of sequential file operations into seconds of concurrent processing.

Smart Scheduling Strategy

The architecture uses a simple but effective rule to determine execution strategy:



This approach balances performance with safety:

- **Read operations** run in parallel (file reads, searches) with no risk of conflicts
- **Write operations** execute sequentially (file edits, bash commands) to avoid race conditions

Tool Categories

Claude Code divides tools into two categories that determine their execution behavior:

Read-Only Tools (Parallel-Safe)

These tools only read data and never modify state, making them safe to run simultaneously:

- `GlobTool` - Finds files matching patterns like "src/**/*.ts"
- `GrepTool` - Searches file contents for text patterns
- `View` - Reads file content
- `LS` - Lists directory contents
- `ReadNotebook` - Extracts cells from Jupyter notebooks

Non-Read-Only Tools (Sequential Only)

These tools modify state and must run one after another:

- `Edit` - Makes targeted changes to files
- `Replace` - Overwrites entire files
- `Bash` - Executes terminal commands
- `NotebookEditCell` - Modifies Jupyter notebook cells

Parallel Execution Under the Hood

The concurrent execution is powered by JavaScript async generators. Let's break down the implementation into manageable pieces:

1. The Core Generator Utility

The system manages multiple async generators through a central coordination function:

```
export async function* all<T>(
  generators: Array<AsyncGenerator<T>>,
  options: { signal?: AbortSignal; maxConcurrency?: number } = {}
): AsyncGenerator<T & { generatorIndex: number }> {
  const { signal, maxConcurrency = 10 } = options;

  // Track active generators
  const remaining = new Set(generators.map((_, i) => i));

  // Map tracks generator state
  const genStates = new Map<number, {
    generator: AsyncGenerator<T>,
    nextPromise: Promise<IteratorResult<T>>,
    done: boolean
  }>();

  // More implementation details...
}
```

2. Initializing the Generator Pool

The system starts with a batch of generators up to the concurrency limit:

```
// Initialize first batch (respect max concurrency)
const initialBatchSize = Math.min(generators.length, maxConcurrency);
for (let i = 0; i < initialBatchSize; i++) {
  genStates.set(i, {
    generator: generators[i],
    nextPromise: generators[i].next(),
    done: false
  });
}
```

3. Racing for Results

The system uses Promise.race to process whichever generator completes next:

```
// Process generators until all complete
while (remaining.size > 0) {
  // Check for cancellation
  if (signal?.aborted) {
    throw new Error('Operation aborted');
  }

  // Wait for next result from any generator
  const entries = Array.from(genStates.entries());
  const { index, result } = await Promise.race(
    entries.map(async ([index, state]) => {
      const result = await state.nextPromise;
      return { index, result };
    })
  );

  // Process result...
}
```

4. Processing Results and Cycling Generators

When a result arrives, the system yields it and queues the next one:

```
  if (result.done) {
    // This generator is finished
    remaining.delete(index);
    genStates.delete(index);

    // Start another generator if available
    const nextIndex = generators.findIndex((_, i) =>
      i >= initialBatchSize && !genStates.has(i));

    if (nextIndex >= 0) {
      genStates.set(nextIndex, {
        generator: generators[nextIndex],
        nextPromise: generators[nextIndex].next(),
        done: false
      });
    }
  } else {
    // Yield this result with its origin
    yield { ...result.value, generatorIndex: index };

    // Queue next value from this generator
    const state = genStates.get(index)!;
    state.nextPromise = state.generator.next();
  }
```

Executing Tools with Smart Scheduling

The execution strategy adapts based on the tools' characteristics:

```
async function executeTools(toolUses: ToolUseRequest[]) {
  // Check if all tools are read-only
  const allReadOnly = toolUses.every(toolUse => {
    const tool = findToolByName(toolUse.name);
    return tool?.isReadOnly();
  });

  if (allReadOnly) {
    // Run concurrently for read-only tools
    return runConcurrently(toolUses);
  } else {
    // Run sequentially for any write operations
    return runSequentially(toolUses);
  }
}
```

Concurrent Execution Path

For read-only operations, the system runs everything in parallel:

```
async function runConcurrently(toolUses) {
  // Convert tool requests to generators
  const generators = toolUses.map(toolUse => {
    const tool = findToolByName(toolUse.name)!;
    return tool.call(toolUse.parameters);
  });

  // Collect results with origin tracking
  const results = [];
  for await (const result of all(generators)) {
    results.push({
      ...result,
      toolIndex: result.generatorIndex
    });
  }

  // Sort to match original request order
  return results.sort((a, b) => a.toolIndex - b.toolIndex);
}
```

Sequential Execution Path

For operations that modify state, the system runs them one at a time:

```
async function runSequentially(toolUses) {
  const results = [];
  for (const toolUse of toolUses) {
    const tool = findToolByName(toolUse.name)!;
    const generator = tool.call(toolUse.parameters);

    // Get all results from this tool before continuing
    for await (const result of generator) {
      results.push(result);
    }
  }
  return results;
}
```

Performance Benefits

This pattern delivers major performance gains with minimal complexity. Notable advantages include:

1. **Controlled Concurrency** - Runs up to 10 tools simultaneously (configurable)
2. **Progressive Results** - Data streams back as available without waiting for everything
3. **Order Preservation** - Results include origin information for correct sequencing
4. **Cancellation Support** - AbortSignal propagates to all operations for clean termination
5. **Resource Management** - Limits concurrent operations to prevent system overload

For large codebases, this approach can turn minutes of waiting into seconds of processing. The real power comes when combining multiple read operations:

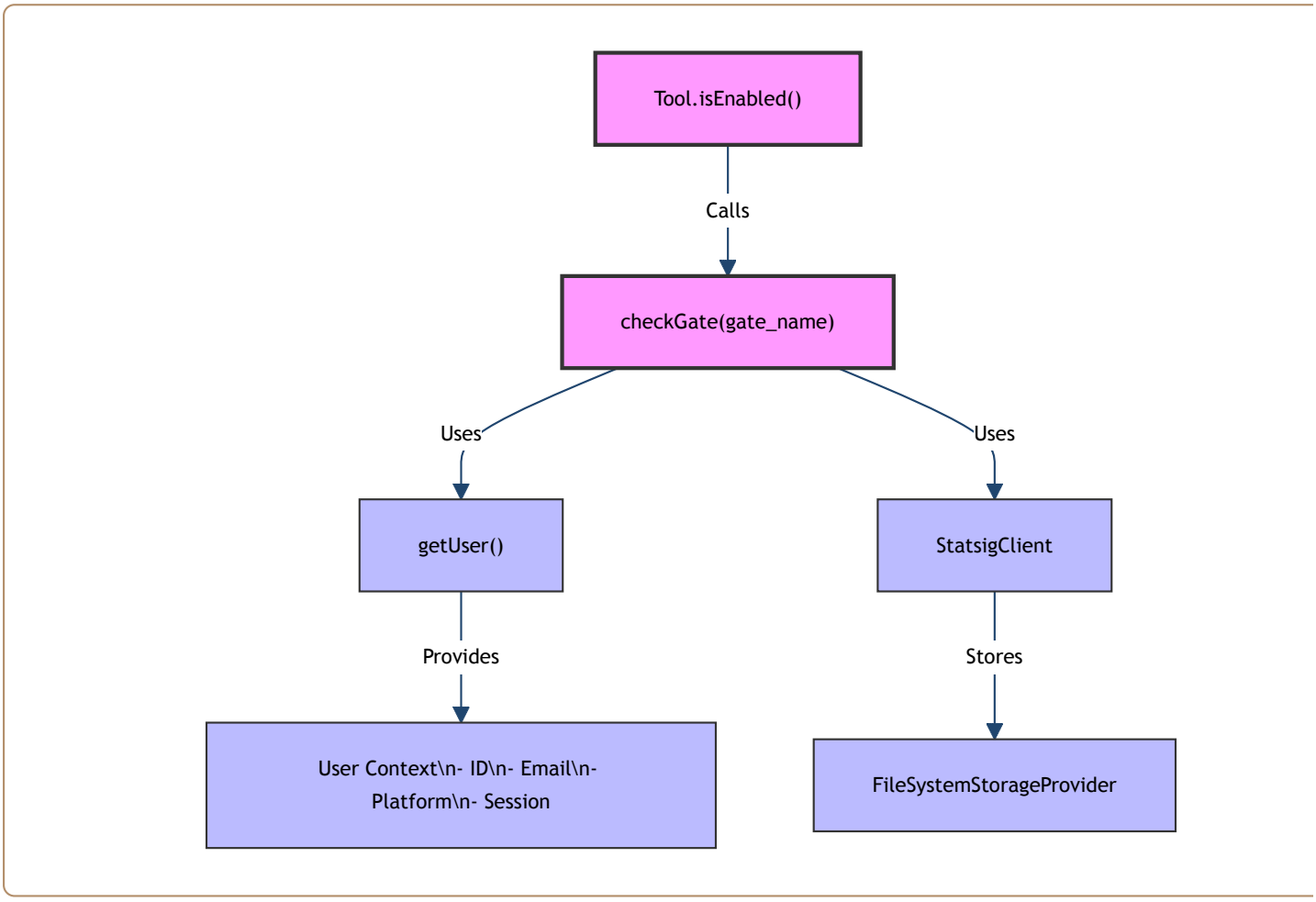
```
// Example of multiple tools running simultaneously
const filePatterns = await globTool("src/**/*.ts");
const apiUsageFiles = await grepTool("fetch\\(|axios|request\\(");
const translationFiles = await grepTool("i18n\\.|translate\\(");

// All three operations execute in parallel
// Rather than one after another
```

This pattern is essential for building responsive AI agents. File I/O is typically a major bottleneck for responsiveness - making these operations concurrent transforms the user experience from painfully slow to genuinely interactive.

Feature Flag Integration

The codebase demonstrates a robust pattern for controlling feature availability using a feature flag system. This approach allows for gradual rollouts and experimental features.

Implementation Pattern



The feature flag system follows this pattern:

1. **Flag Definition**: The `isEnabled()` method in each tool controls availability:

```
async isEnabled() {
  // Tool-specific activation logic
  return Boolean(process.env.SOME_FLAG) && (await
checkGate('gate_name'));
}
```

2. **Statsig Client**: The system uses Statsig for feature flags with these core functions:

```
export const checkGate = memoize(async (gateName: string):
Promise<boolean> => {
  // Gate checking logic - currently simplified
  return true;
  // Full implementation would initialize client and check actual flag
value
})
```

3. **User Context**: Flag evaluation includes user context from `utils/user.ts`:

```
export const getUser = memoize(async (): Promise<StatsigUser> => {
  const userID = getOrCreateUserID()
  // Collects user information including email, platform, session
  // ...
})
```

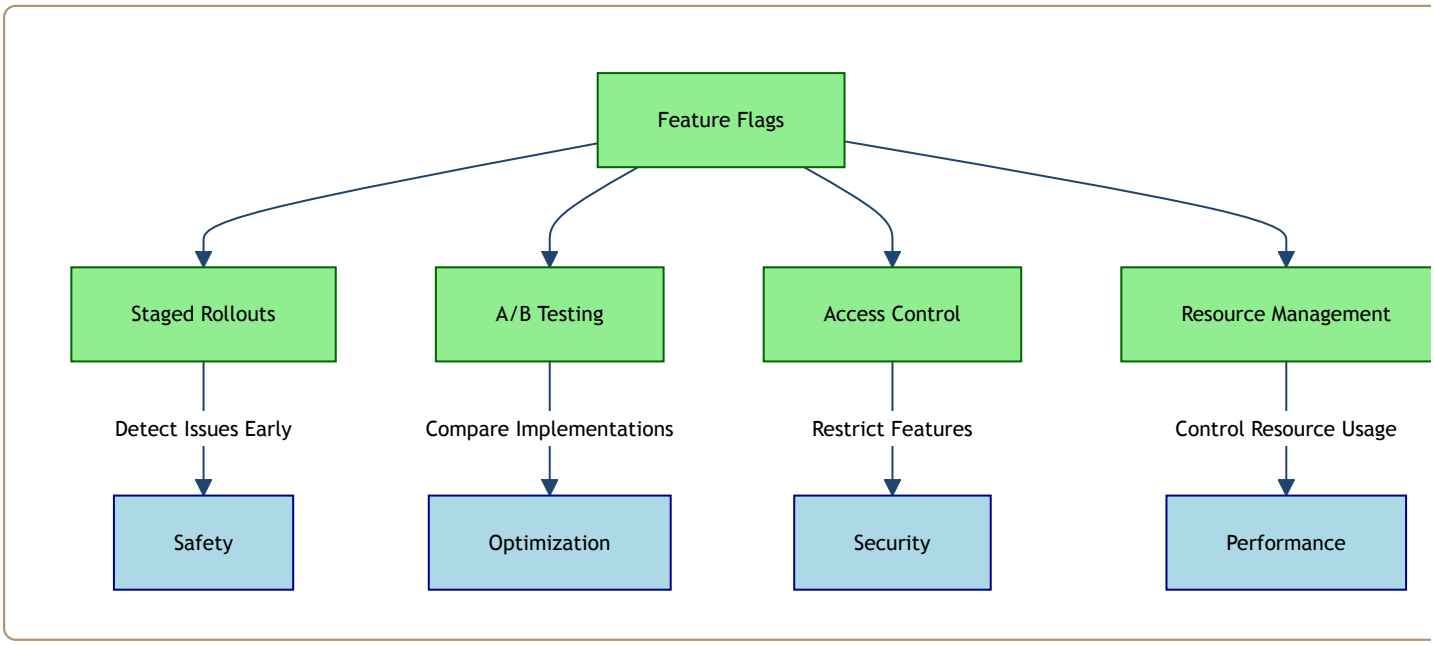
4. **Persistence**: Flag states are cached using a custom storage provider:

```
export class FileSystemStorageProvider implements StorageProvider {
  // Stores Statsig data in ~/.claude/statsig/
  // ...
}
```

5. **Gate Pattern**: Many tools follow a pattern seen in ThinkTool:

```
isEnabled: async () =>
  Boolean(process.env.THINK_TOOL) && (await
checkGate('tengu_think_tool')),
```

Benefits for Agentic Systems



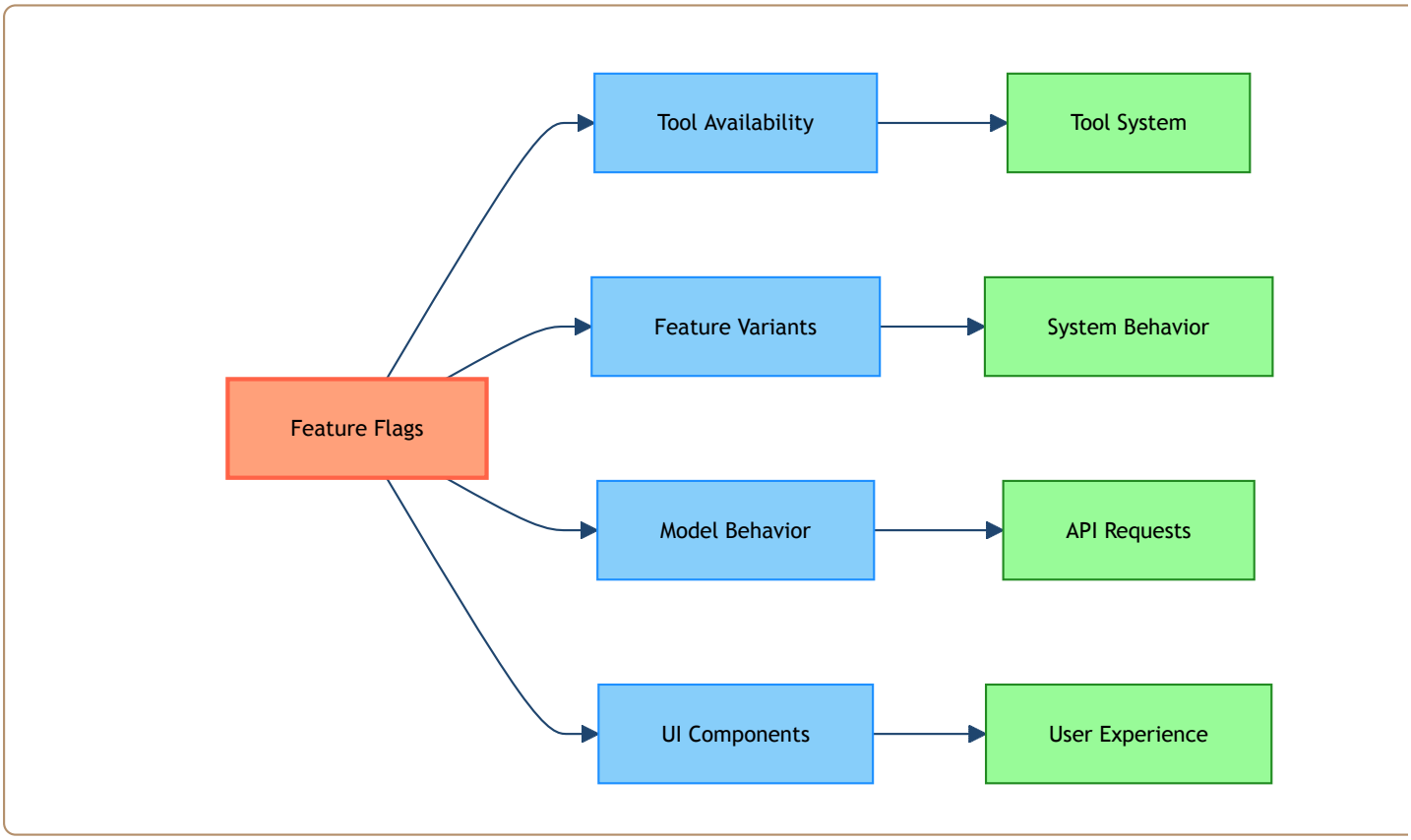
Feature flags provide several practical benefits for agentic systems:

- **Staged Rollouts**: Gradually release features to detect issues before wide deployment
- **A/B Testing**: Compare different implementations of the same feature
- **Access Control**: Restrict experimental features to specific users or environments
- **Resource Management**: Selectively enable resource-intensive features

Feature Flag Standards

For implementing feature flags in your own agentic system, consider OpenFeature, which provides a standardized API with implementations across multiple languages.

Usage in the Codebase



Throughout the codebase, feature flags control:

- **Tool availability** (through each tool's `isEnabled()` method)
- **Feature variants** (via experiment configuration)
- **Model behavior** (through beta headers and capabilities)
- **UI components** (conditionally rendering based on flag state)

This creates a flexible system where capabilities can be adjusted without code changes, making it ideal for evolving agentic systems.

Real-World Examples

To illustrate how all these components work together, let's walk through two concrete examples.

Example 1: Finding and Fixing a Bug

Below is a step-by-step walkthrough of a user asking Claude Code to "Find and fix bugs in the file Bug.tsx":

Phase 1: Initial User Input and Processing

1. User types "Find and fix bugs in the file Bug.tsx" and hits Enter
2. `PromptInput.tsx` captures this input in its `value` state
3. `onSubmit()` handler creates an AbortController and calls `processUserInput()`
4. Input is identified as a regular prompt (not starting with `!` or `/`)
5. A message object is created with:

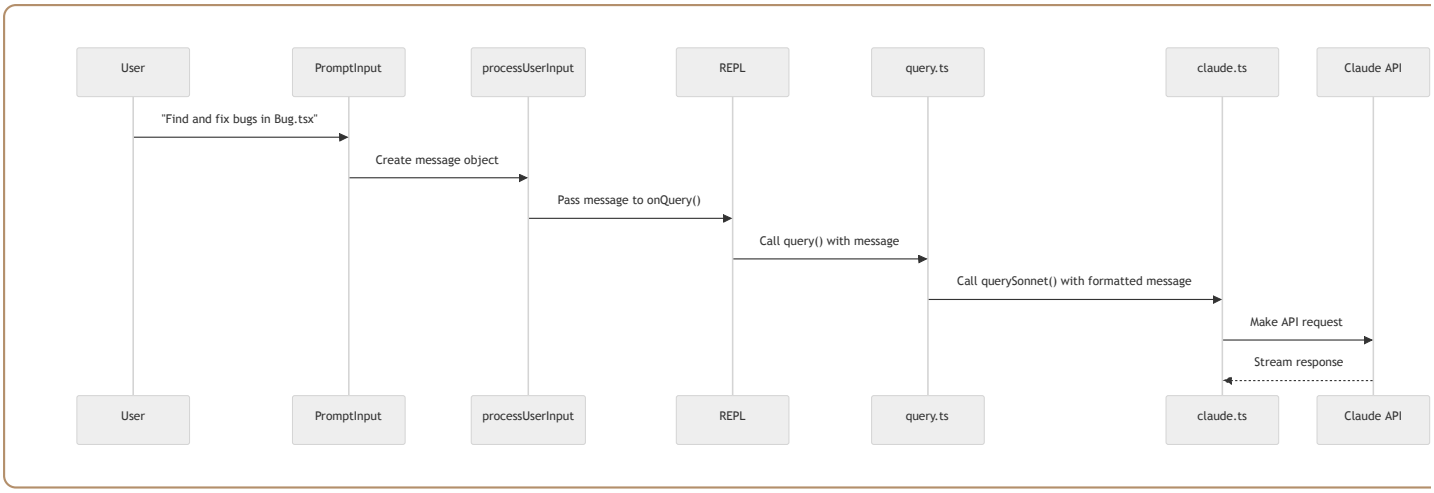
```
{
  role: 'user',
  content: 'Find and fix bugs in the file Bug.tsx',
  type: 'prompt',
  id: generateId()
}
```

6. The message is passed to `onQuery()` in `REPL.tsx`

Phase 2: Query Generation and API Call

1. `onQuery()` collects:
   - System prompt from `getSystemPrompt()` including capabilities info
   - Context from `getContextForQuery()` including directory structure
   - Model information from state
2. `query()` in `query.ts` is called with the messages and options
3. Messages are formatted into Claude API format in `querySonnet()`

4. API call is made to Claude using `fetch()` in `services/claude.ts`
5. Response begins streaming with content starting to contain a plan to find bugs



Phase 3: Tool Use Execution - Finding the File

1. Claude decides to use `GlobTool` to locate the file
2. The response contains a tool_use block:

```
{
  "name": "GlobTool",
  "parameters": {
    "pattern": "**/Bug.tsx"
  }
}
```

3. Generator system detects tool_use and calls `runToolsConcurrently()`
4. `canUseTool()` checks permissions for GlobTool which can run without explicit user approval
5. `GlobTool.call()` executes with parameters, running ripgrep on the filesystem
6. Results are returned:

```
Found 1 file:
/Users/gerred/dev/anon-kode/src/components/Bug.tsx
```

Phase 4: Tool Use Execution - Reading the File

1. Claude decides to use `View` to read the file content
2. Another tool_use block is generated:

```
{
  "name": "View",
  "parameters": {
    "file_path": "/Users/gerred/dev/anon-
kode/src/components/Bug.tsx"
  }
}
```

3. `View.call()` executes, reading the file content
4. Results contain the file content with a potential bug:

```
import React from 'react';
import { Box } from 'ink';

interface BugProps {
  message: string;
}

// Bug: This component tries to access undefinedProp which doesn't
exist
export function Bug({ message }: BugProps) {
  return (
    <Box>
      {message.toUpperCase()}
      {undefinedProp.toString()} // This will cause an error
    </Box>
  );
}
```

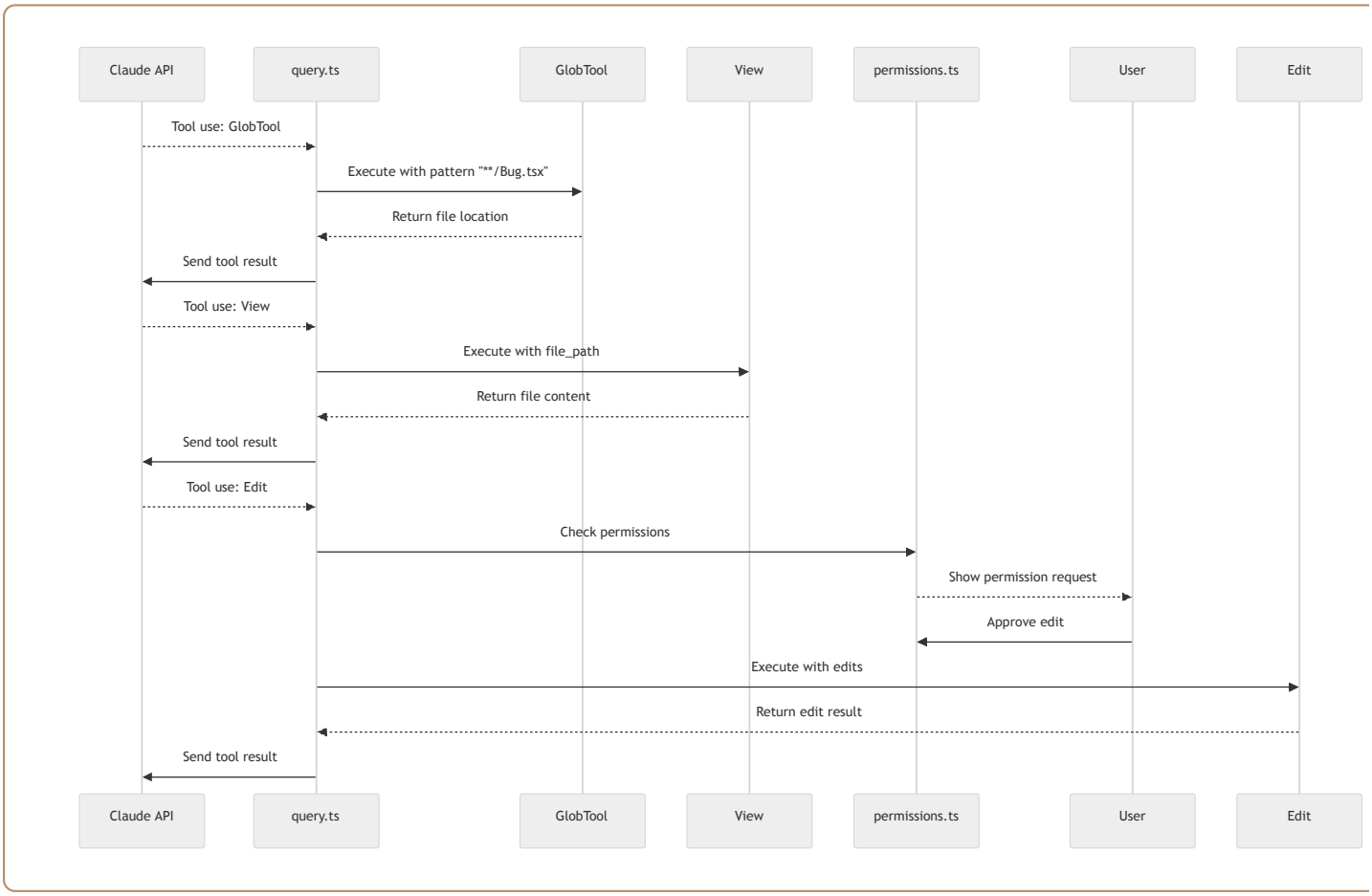
Phase 5: Tool Use Execution - Editing the File

1. Claude decides to use `Edit` to fix the bug
2. Another tool_use block is generated:

```
{
  "name": "Edit",
  "parameters": {
    "file_path": "/Users/gerred/dev/anon-
kode/src/components/Bug.tsx",
    "old_string": "export function Bug({ message }: BugProps) {\n
return (\n    <Box>\n      {message.toUpperCase()}\n
{undefinedProp.toString()} // This will cause an error\n    </Box>\n
);\n}",
    "new_string": "export function Bug({ message }: BugProps) {\n
return (\n    <Box>\n      {message.toUpperCase()}\n      {/*
Removed reference to undefined prop */}\n    </Box>\n  );\n}"
  }
}
```


3. `Edit.needsPermissions` returns true, prompting user permission
4. A permission request is displayed: "Claude wants to edit file /Users/gerred/dev/anon-kode/src/components/Bug.tsx"
5. User approves the edit
6. `Edit.call()` executes, modifying the file
7. Results show successful edit:

```
The file /Users/gerred/dev/anon-kode/src/components/Bug.tsx has been
updated.
```



Phase 6: Recursive Query and Final Response

1. After each tool execution, the results are added to the messages array:

```
messages.push({
  role: 'assistant',
  content: null,
  tool_use: { ... } // Tool use object
});
messages.push({
  role: 'user',
  content: null,
  tool_result: { ... } // Tool result object
});
```

2. `query()` is called recursively with updated messages
3. Claude API generates a final response summarizing the bug fix
4. This final response streams back to the UI without any further tool use
5. The message is normalized and shown to the user

Example 2: Parallel Codebase Analysis

This example showcases a user asking Claude Code to "Show me all React components using useState hooks":

Phase 1: Initial User Input and Processing

Just as in Example 1, the input is captured, processed, and passed to the query system.

Phase 2: Claude's Response with Multiple Tool Uses

Claude analyzes the request and determines it needs to:

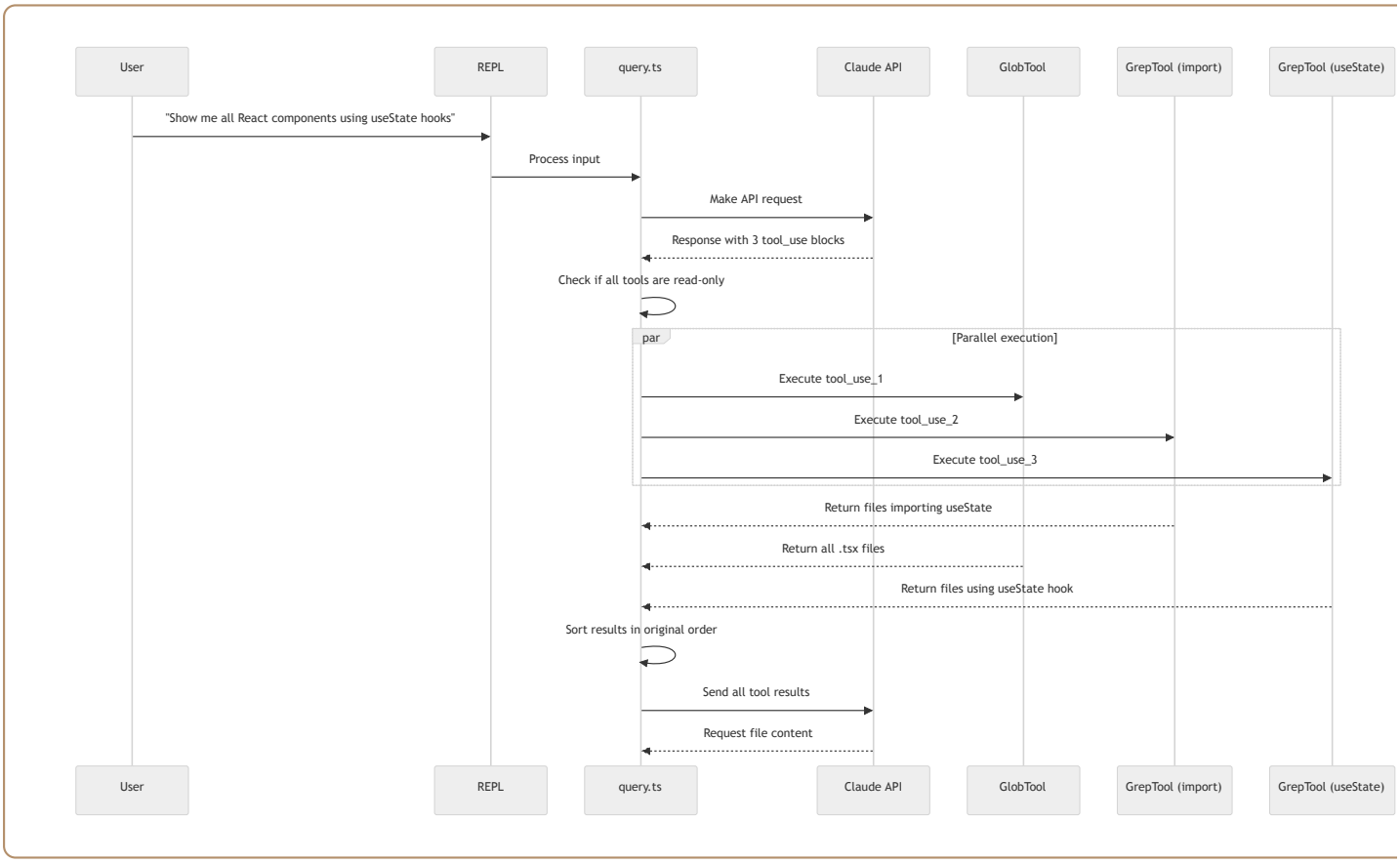
- Find all React component files
- Search for useState hook usage
- Read relevant files to show the components

Instead of responding with a single tool use, Claude returns multiple tool uses in one response:

```
{
  "content": [
    {
      "type": "tool_use",
      "id": "tool_use_1",
      "name": "GlobTool",
      "parameters": {
        "pattern": "**/*.tsx"
      }
    },
    {
      "type": "tool_use",
      "id": "tool_use_2",
      "name": "GrepTool",
      "parameters": {
        "pattern": "import.*\\{.*useState.*\\}.*from.*['\"]react['\"]",
        "include": "*.tsx"
      }
    },
    {
      "type": "tool_use",
      "id": "tool_use_3",
      "name": "GrepTool",
      "parameters": {
        "pattern": "const.*\\[.*\\].*=.*useState\\(",
        "include": "*.tsx"
      }
    }
  ]
}
```


Phase 3: Parallel Tool Execution

1. `query.ts` detects multiple tool uses in one response
2. It checks if all tools are read-only (GlobTool and GrepTool are both read-only)
3. Since all tools are read-only, it calls `runToolsConcurrently()`



The results are collected from all three tools, sorted back to the original order, and sent back to Claude. Claude then requests to read specific files, which are again executed in parallel, and finally produces an analysis of the useState usage patterns.

This parallel execution significantly speeds up response time by:

1. Running all file search operations concurrently
2. Running all file read operations concurrently
3. Maintaining correct ordering of results
4. Streaming all results back as soon as they're available

Lessons Learned and Implementation Challenges

Taking apart Claude Code revealed some tricky engineering problems worth calling out:

Async Complexity

Async generators are powerful but add complexity. What worked: • Explicit cancellation: Always handle abort signals clearly. • Backpressure: Stream carefully to avoid memory leaks. • Testing generators: Normal tools fall short; you'll probably need specialized ones.

Example of a well-structured async generator:

```
async function* generator(signal: AbortSignal): AsyncGenerator<Result> {
  try {
    while (moreItems()) {
      if (signal.aborted) throw new AbortError();
      yield await processNext();
    }
  } finally {
    await cleanup();
  }
}
```

Tool System Design

Good tools need power without accidental footguns. Claude Code handles this by: • Having clear but not overly granular permissions. • Making tools discoverable with structured definitions.

Terminal UI Challenges

Terminals seem simple, but UI complexity sneaks up on you: • Different terminals mean compatibility headaches. • Keyboard input and state management require careful handling.

Integrating with LLMs

LLMs are non-deterministic. Defensive coding helps: • Robust parsing matters; don’t trust outputs blindly. • Carefully manage context window limitations.

Performance Considerations

Keeping the tool responsive is critical: • Parallelize carefully; manage resource usage. • Implement fast cancellation to improve responsiveness.

Hopefully, these insights save you some headaches if you’re exploring similar ideas.

Tool System Deep Dive

Note: This section is undergoing deep review. While these notes provide a useful overview, they are not yet as detailed or polished as other sections. Please file a GitHub issue if you notice any errors or have suggestions for additional content.

The power of Claude Code comes from its extensive tool system that allows Claude to interact with your local environment. This section analyzes each key tool in detail, exploring their implementation, integration points, and technical nuances.

Below you'll find detailed descriptions of each tool, with individual pages for each tool implementation:

- Agent: Delegated task execution
- Architect: Software architecture planning
- Bash: Command-line execution
- FileEdit: Precise file modifications
- FileRead: Content examination
- FileWrite: File creation and updates
- Glob: Pattern-based file matching
- Grep: Content search across files
- LS: Directory listing
- MCP: External tool integration
- MemoryRead: State persistence reading
- MemoryWrite: State persistence writing
- NotebookEdit: Notebook modification
- NotebookRead: Jupyter notebook inspection
- StickerRequest: User engagement
- Think: Structured reasoning

Creating Your Own Tools

Before diving into specific tools, here's how you'd build your own tool in TypeScript:

```
import { z } from "zod";
import { Tool } from "./Tool";

// Define a custom search tool that finds text in files
```

```
export const SearchTool: Tool = {
  // Name must be unique and descriptive
  name: "Search",

  // Description tells Claude what this tool does
  description: "Searches files for specific text patterns",

  // Define expected parameters using Zod schema
  inputSchema: z.object({
    query: z.string().describe("The text pattern to search for"),
    path: z.string().optional().describe("Optional directory to search
in"),
    fileTypes: z
      .array(z.string())
      .optional()
      .describe("File extensions to include"),
  }),

  // Is this a read-only operation?
  isReadOnly: () => true, // Enables parallel execution

  // Does this require user permission?
  needsPermissions: (input) => {
    // Only need permission for certain paths
    return input.path && !input.path.startsWith("/safe/path");
  },

  // The actual implementation as an async generator
  async *call(input, context) {
    const { query, path = process.cwd(), fileTypes } = input;

    // Check for abort signal (enables cancellation)
    if (context.signal?.aborted) {
      throw new Error("Operation canceled");
    }

    try {
      // Yield initial status
      yield { status: "Searching for files..." };

      // Implementation logic here...
      const results = await performSearch(query, path, fileTypes);

      // Yield results progressively as they come in
      for (const result of results) {
        yield {
          file: result.path,
          line: result.lineNumber,
          content: result.matchedText,
        };

        // Check for abort between results
        if (context.signal?.aborted) {
          throw new Error("Operation canceled");
```

```
        }
      }

      // Yield final summary
      yield {
        status: "complete",
        totalMatches: results.length,
        searchTime: performance.now() - startTime,
      };
    } catch (error) {
      // Handle and report errors properly
      yield { error: error.message };
    }
  },
};
```

AgentTool: Your Research Assistant

AgentTool (dispatch_agent) lets you launch mini-Claudes to search and explore your codebase. This is great for finding things across multiple files or when you're not sure exactly where to look.

Complete Prompt

```
export async function getPrompt(
  dangerouslySkipPermissions: boolean,
): Promise<string> {
  const tools = await getAgentTools(dangerouslySkipPermissions)
  const toolNames = tools.map(_ => _.name).join(', ')
  return `Launch a new agent that has access to the following tools:
${toolNames}. When you are searching for a keyword or file and are not
confident that you will find the right match on the first try, use the
Agent tool to perform the search for you. For example:

- If you are searching for a keyword like "config" or "logger", the Agent
tool is appropriate
- If you want to read a specific file path, use the ${FileReadTool.name}
or ${GlobTool.name} tool instead of the Agent tool, to find the match
more quickly
- If you are searching for a specific class definition like "class Foo",
use the ${GlobTool.name} tool instead, to find the match more quickly

Usage notes:
1. Launch multiple agents concurrently whenever possible, to maximize
performance; to do that, use a single message with multiple tool uses
2. When the agent is done, it will return a single message back to you.
The result returned by the agent is not visible to the user. To show the
user the result, you should send a text message back to the user with a
concise summary of the result.
3. Each agent invocation is stateless. You will not be able to send
additional messages to the agent, nor will the agent be able to
communicate with you outside of its final report. Therefore, your prompt
should contain a highly detailed task description for the agent to
perform autonomously and you should specify exactly what information the
agent should return back to you in its final and only message to you.
4. The agent's outputs should generally be trusted${
    dangerouslySkipPermissions
      ? ''
      : `
5. IMPORTANT: The agent can not use ${BashTool.name},
${FileWriteTool.name}, ${FileEditTool.name}, ${NotebookEditTool.name}, so
can not modify files. If you want to use these tools, use them directly
instead of going through the agent.`
  }`
}
```

> **Tool Prompt: dispatch_agent**
>
> Launch a new agent that has access to the following tools: View, GlobTool, GrepTool, LS, ReadNotebook, WebFetchTool. When you are searching for a keyword or file and are not confident that you will find the right match on the

first try, use the Agent tool to perform the search for you. For example:

- If you are searching for a keyword like "config" or "logger", or for questions like "which file does X?", the Agent tool is strongly recommended
- If you want to read a specific file path, use the View or GlobTool tool instead of the Agent tool, to find the match more quickly
- If you are searching for a specific class definition like "class Foo", use the GlobTool tool instead, to find the match more quickly

Usage notes:

1. Launch multiple agents concurrently whenever possible, to maximize performance; to do that, use a single message with multiple tool uses
2. When the agent is done, it will return a single message back to you. The result returned by the agent is not visible to the user. To show the user the result, you should send a text message back to the user with a concise summary of the result.
3. Each agent invocation is stateless. You will not be able to send additional messages to the agent, nor will the agent be able to communicate with you outside of its final report. Therefore, your prompt should contain a highly detailed task description for the agent to perform autonomously and you should specify exactly what information the agent should return back to you in its final and only message to you.
4. The agent's outputs should generally be trusted
5. IMPORTANT: The agent can not use Bash, Replace, Edit, NotebookEditCell, so can not modify files. If you want to use these tools, use them directly instead of going through the agent.

How It Works

AgentTool creates a separate Claude to handle research tasks:

```
export const AgentTool = {
  name: 'dispatch_agent',
  async *call({ prompt }, { abortController, options, readFileTimestamps
}) {
    const startTime = Date.now()
    const messages = [createUserMessage(prompt)]
```

```
    const tools = await getAgentTools(dangerouslySkipPermissions)

    // Yield initial progress for UI feedback
    yield { type: 'progress', content:
createAssistantMessage('Initializing…') }

    // Set up agent environment
    const [agentPrompt, context, model, maxThinkingTokens] = await
Promise.all([
      getAgentPrompt(),
      getContext(),
      getSlowAndCapableModel(),
      getMaxThinkingTokens(messages),
    ])

    // Generate unique sidechain number for logs
    const getSidechainNumber = memoize(() =>
      getNextAvailableLogSidechainNumber(messageLogName, forkNumber)
    )

    // Process the agent's execution and track tools used
    let toolUseCount = 0
    for await (const message of query(messages, agentPrompt, context,
...)) {
      messages.push(message)

      // Log to separate sidechain to avoid polluting main logs
      overwriteLog(
        getMessagesPath(messageLogName, forkNumber,
getSidechainNumber()),
        messages.filter(_ => _.type !== 'progress')
      )

      // Track and report tool use progress
      if (message.type === 'assistant') {
        for (const content of message.message.content) {
          if (content.type === 'tool_use') {
            toolUseCount++
            yield { type: 'progress', content: /* tool message */ }
          }
        }
      }
    }

    // Extract and return the final result
    const data = lastMessage.message.content.filter(_ => _.type ===
'text')
    yield {
      type: 'result',
      data,
      resultForAssistant: this.renderResultForAssistant(data),
    }
  },
  isReadOnly() {
```

```
      return true // Enforces read-only operation for security
    }
  }
```

What Makes It Special

AgentTool has some key features:

1. **Safety First**
    - Only gets read-only tools (no file editing or Bash)
    - Can't create more agents (no recursion)
    - Runs with clear boundaries
2. **Progress Updates**
    - Shows you what's happening in real time
    - Tracks how many tools it's using
    - Measures how long things take
3. **Parallel Power**
    - Run multiple agents at once
    - Each agent has its own space to work
    - Separate logs keep things organized
4. **Clear Communication**
    - Specialized prompting for search tasks
    - One-and-done execution model
    - Clean result formatting

How It's Built

AgentTool follows this flow:

```
AgentTool
  ↓
getAgentTools() → Gets the safe tools, no recursive agents
  ↓
query() → Creates new Claude instance with search prompt
  ↓
message stream → Tracks progress and tool use
  ↓
Tool execution → Runs tools in isolated context
  ↓
Result formatting → Packages up the final answer
```

The design focuses on:

- **Separation**: Each agent works in its own space
- **Safety**: Clear boundaries for what tools are allowed
- **Speed**: Do multiple searches at once
- **Clarity**: Show progress and format results nicely

Permissions

AgentTool keeps things simple with read-only mode:

```
needsPermissions() {
  return false // Doesn't need its own permissions
}

async getAgentTools(dangerouslySkipPermissions) {
  // Only give agents read-only tools
  return (
    await (dangerouslySkipPermissions ? getTools() : getReadOnlyTools())
  ).filter(_ => _.name !== AgentTool.name)
}
```

This approach:

- Doesn't need to ask permission to start an agent
- Only lets agents do safe, read-only things
- Prevents agents from starting more agents

- Follows the principle of least privilege

Examples

Here's how to use AgentTool:

1. **Find stuff across files**

```
dispatch_agent(prompt: "Find all files that use logging and how they
initialize it")
```

2. **Run multiple searches at once**

```
dispatch_agent(prompt: "Find how error handling works in this
codebase")
dispatch_agent(prompt: "Find all API endpoint definitions")
```

3. **Explore code architecture**

```
dispatch_agent(prompt: "Analyze how authentication works and
summarize the approach")
```

AgentTool is great for exploring large codebases, saving context space, and cutting down on back-and-forth when you're hunting for information.

Architect: Software Design Planning

Architect helps break down technical requirements into clear implementation plans. It analyzes requests and creates structured plans without writing actual code.

Complete Prompt

```
export const ARCHITECT_SYSTEM_PROMPT = `You are an expert software
architect. Your role is to analyze technical requirements and produce
clear, actionable implementation plans.
These plans will then be carried out by a junior software engineer so you
need to be specific and detailed. However do not actually write the code,
just explain the plan.

Follow these steps for each request:
1. Carefully analyze requirements to identify core functionality and
constraints
2. Define clear technical approach with specific technologies and
patterns
3. Break down implementation into concrete, actionable steps at the
appropriate level of abstraction

Keep responses focused, specific and actionable.

IMPORTANT: Do not ask the user if you should implement the changes at the
end. Just provide the plan as described above.
IMPORTANT: Do not attempt to write the code or use any string
modification tools. Just provide the plan.`

export const DESCRIPTION =
  'Your go-to tool for any technical or coding task. Analyzes
requirements and breaks them down into clear, actionable implementation
steps. Use this whenever you need help planning how to implement a
feature, solve a technical problem, or structure your code.'
```

> **Tool Prompt: Architect**
>
> Your go-to tool for any technical or coding task. Analyzes requirements and breaks them down into clear, actionable implementation steps. Use this

> whenever you need help planning how to implement a feature, solve a technical problem, or structure your code.

How It Works

Architect creates a specialized Claude instance with architect-specific instructions:

```
export const ArchitectTool = {
  name: 'Architect',
  async *call({ prompt, context }, toolUseContext, canUseTool) {
    // Combine context and prompt if provided
    const content = context
      ? `<context>${context}</context>\n\n${prompt}`
      : prompt

    const userMessage = createUserMessage(content)
    const messages: Message[] = [userMessage]

    // Only allow file exploration tools in architect mode
    const allowedTools = (toolUseContext.options.tools ?? []).filter(_ =>
      FS_EXPLORATION_TOOLS.map(_ => _.name).includes(_.name),
    )

    // Query Claude with architect system prompt
    const lastResponse = await lastX(
      query(
        messages,
        [ARCHITECT_SYSTEM_PROMPT],
        await getContext(),
        canUseTool,
        {
          ...toolUseContext,
          options: { ...toolUseContext.options, tools: allowedTools },
        },
      ),
    )

    // Process and return text blocks
    const data = lastResponse.message.content.filter(_ => _.type ===
'text')
    yield {
      type: 'result',
      data,
      resultForAssistant: this.renderResultForAssistant(data),
    }
  },
  isReadOnly() {
    return true
  },
  async isEnabled() {
    return false // Disabled by default, requires config enabling
  }
}
```

Key Components

Architect includes these important features:

1. **Input Parameters**
   - `prompt` : The technical request to analyze
   - `context` : Optional context from prior conversation
2. **Limited Tool Access**
   - Only allows specific file exploration tools:

```
const FS_EXPLORATION_TOOLS: Tool[] = [
  BashTool,
  LSTool,
  FileReadTool,
  FileWriteTool,
  GlobTool,
  GrepTool,
]
```

3. **Specialized System Prompt**
   - Three-step analysis process for requirements
   - Planning-focused rather than implementation
   - Instructions to avoid writing actual code
4. **Context Handling**
   - Wraps context in XML tags
   - Keeps context separate from the main prompt

Architecture

Architect follows a straightforward query pattern:

```
Input Processing → Format prompt and context
  ↓
Tool Filtering → Limit to file exploration tools
  ↓
Claude Query → Use specialized architect system prompt
  ↓
Result Processing → Extract and format text blocks
```

Design priorities:

- Focus on architecture planning with specialized prompt
- Simple process with clear instructions
- Read-only operation with limited tool access
- Markdown output for clear presentation

Permissions

Architect uses a simplified permission model:

```
needsPermissions() {
  return false // No explicit permissions needed
}
```

The tool:

- Needs no explicit user permissions
- Operates in read-only mode
- Is disabled by default until explicitly enabled
- Filters available tools for safe operation

Usage Examples

Typical use cases:

1. **Planning feature implementation**

```
Architect(prompt: "How should I implement a rate limiting middleware
for our Express API?")
```

2. **Architecture design with context**

```
Architect(
  prompt: "What's the best way to structure a React app that uses
GraphQL?",
  context: "Our team is familiar with Redux but open to
alternatives"
)
```

3. **Technical problem solving**

```
Architect(prompt: "How should we approach migrating from MongoDB to
PostgreSQL in our Node.js app?")
```

Architect helps bridge the gap between high-level requirements and concrete code changes, enabling thoughtful architecture decisions before implementation begins.

BashTool: Command Execution

BashTool runs commands in a persistent shell session. It maintains state between commands and has security measures to keep things safe while still being useful.

Complete Prompt

```
// Tool Prompt: Bash
export const PROMPT = `Executes a given bash command in a persistent
shell session with optional timeout, ensuring proper handling and
security measures.

Before executing the command, please follow these steps:

1. Directory Verification:
   - If the command will create new directories or files, first use the
LS tool to verify the parent directory exists and is the correct location
   - For example, before running "mkdir foo/bar", first use LS to check
that "foo" exists and is the intended parent directory

2. Security Check:
   - For security and to limit the threat of a prompt injection attack,
some commands are limited or banned. If you use a disallowed command, you
will receive an error message explaining the restriction. Explain the
error to the User.
   - Verify that the command is not one of the banned commands: alias,
curl, curlie, wget, axel, aria2c, nc, telnet, lynx, w3m, links, httpie,
xh, http-prompt, chrome, firefox, safari.

3. Command Execution:
   - After ensuring proper quoting, execute the command.
   - Capture the output of the command.

Usage notes:
  - The command argument is required.
  - You can specify an optional timeout in milliseconds (up to 600000ms /
10 minutes). If not specified, commands will timeout after 30 minutes.
- If the output exceeds 30000 characters, output will be truncated before
being returned to you.
  - VERY IMPORTANT: You MUST avoid using search commands like \`find\`
and \`grep\`. Instead use GrepTool, GlobTool, or dispatch_agent to
search. You MUST avoid read tools like \`cat\`, \`head\`, \`tail\`, and
\`ls\`, and use View and LS to read files.
  - When issuing multiple commands, use the ';' or '&&' operator to
```

```
separate them. DO NOT use newlines (newlines are ok in quoted strings).
  - IMPORTANT: All commands share the same shell session. Shell state
(environment variables, virtual environments, current directory, etc.)
persist between commands. For example, if you set an environment variable
as part of a command, the environment variable will persist for
subsequent commands.
  - Try to maintain your current working directory throughout the session
by using absolute paths and avoiding usage of \`cd\`. You may use \`cd\`
if the User explicitly requests it.
  <good-example>
  pytest /foo/bar/tests
  </good-example>
  <bad-example>
  cd /foo/bar && pytest tests
  </bad-example>

# Committing changes with git

When the user asks you to create a new git commit, follow these steps
carefully:

[Git commit guidance...]

# Creating pull requests
Use the gh command via the Bash tool for ALL GitHub-related tasks
including working with issues, pull requests, checks, and releases. If
given a Github URL use the gh command to get the information needed.

[PR creation guidance...]`
```

Executes a given bash command in a persistent shell session with optional timeout, ensuring proper handling and security measures.

Before executing the command, please follow these steps:

1. Directory Verification:
    - If the command will create new directories or files, first use the LS tool to verify the parent directory exists and is the correct location
    - For example, before running "mkdir foo/bar", first use LS to check that "foo" exists and is the intended parent directory
2. Security Check:
    - For security and to limit the threat of a prompt injection attack, some commands are limited or banned. If you use a disallowed command, you

will receive an error message explaining the restriction. Explain the error to the User.
    - Verify that the command is not one of the banned commands: alias, curl, curlie, wget, axel, aria2c, nc, telnet, lynx, w3m, links, httpie, xh, http-prompt, chrome, firefox, safari.

3. Command Execution:
    - After ensuring proper quoting, execute the command.
    - Capture the output of the command.

Usage notes:

- The command argument is required.
- You can specify an optional timeout in milliseconds (up to 600000ms / 10 minutes). If not specified, commands will timeout after 30 minutes.
- If the output exceeds 30000 characters, output will be truncated before being returned to you.
    - VERY IMPORTANT: You MUST avoid using search commands like `find` and `grep`. Instead use GrepTool, GlobTool, or dispatch_agent to search. You MUST avoid read tools like `cat`, `head`, `tail`, and `ls`, and use View and LS to read files.
    - When issuing multiple commands, use the ';' or '&&' operator to separate them. DO NOT use newlines (newlines are ok in quoted strings).
    - IMPORTANT: All commands share the same shell session. Shell state (environment variables, virtual environments, current directory, etc.) persist between commands. For example, if you set an environment variable as part of a command, the environment variable will persist for subsequent commands.
    - Try to maintain your current working directory throughout the session by using absolute paths and avoiding usage of `cd`. You may use `cd` if the User explicitly requests it.

How It Works

The BashTool has a few key parts:

1. **PersistentShell**
    - Keeps one shell session running throughout your conversation
    - Remembers working directory and environment variables
    - Sets up a proper interactive shell
    - Handles command timeouts
2. **Security**
    - Blocks potentially dangerous commands
    - Keeps operations within your project directories
    - Validates commands before running them
    - Checks shell syntax validity
3. **Permission System**
    - Three approval levels:
        - One-time (temporary)
        - Pattern-based (like all `git` commands)
        - Exact command matches
    - Evaluates command risk levels
    - Explains what commands do in plain language
4. **Output Handling**
    - Manages stdout and stderr
    - Handles output truncation for large results
    - Processes multi-line commands
    - Preserves special formatting
5. **Command Management**
    - Runs commands sequentially
    - Handles timeouts
    - Tracks exit codes
    - Cleans up processes and temporary files

Under the Hood

BashTool is built in layers:

```
BashTool.tsx (Interface)
  ↓
PersistentShell (Core)
  ↓
Node child_process (Execution)
```

When you run a command, it goes through these steps:

1. **Setup**
    - Initializes the shell
    - Creates temp files for output
    - Loads configurations
    - Sets up environment
2. **Validation**
    - Checks against banned commands
    - Ensures `cd` commands stay within bounds
    - Validates shell syntax
    - Determines required permissions
3. **Execution**
    - Runs commands in order
    - Captures output
    - Monitors for completion or timeout
    - Handles interruptions
4. **Processing**
    - Reads output files
    - Formats and truncates if needed
    - Reports errors
    - Preserves shell state
5. **Cleanup**
    - Terminates processes if needed
    - Updates timestamps

- Returns results
- Clears buffers

Permissions

BashTool always requires permission:

```
needsPermissions(): boolean {
  // Always check permissions for Bash
  return true
}
```

Unlike other tools that might skip permission checks, BashTool always asks.

The permission system offers three options:

1. **Temporary**
   - Just for this one command
   - Doesn't persist between sessions
   - Good for one-off commands
2. **Prefix**
   - Allows all commands with a common prefix
   - Example: approve all `git` commands with one permission
   - Persists between sessions
3. **Full Command**
   - Approves only that exact command
   - Most restrictive option
   - Persists between sessions

The system tracks what gets approved or denied to help improve the tool.

Examples

Here's how to use BashTool for common tasks:

1. **Environment Info**

```
Bash({ command: "pwd" })
Bash({ command: "env | grep PATH" })
```

2. **Project Tasks**

```
Bash({ command: "git status" })
Bash({ command: "npm install" })
```

3. **File Operations**

```
Bash({ command: "mkdir -p src/components/buttons" })
Bash({ command: "touch README.md" })
```

4. **Build and Test**

```
Bash({ command: "npm run build" })
Bash({ command: "pytest -xvs tests/" })
```

5. **Multi-step Commands**

```
Bash({ command: "export NODE_ENV=production && npm run build && node
./scripts/post-build.js" })
```

BashTool lets Claude execute terminal commands safely. It handles everything from simple file operations to complex build pipelines while keeping security guardrails in place.

Edit: Precise File Modifications

Edit performs targeted changes to files with strict validation to prevent unintended modifications. It requires unique text matches to ensure changes are precise and intentional.

Complete Prompt

```
// Tool Prompt: FileEditTool
export const PROMPT = `This is a tool for editing files. For moving or
renaming files, you should generally use the Bash tool with the 'mv'
command instead. For larger edits, use the Write tool to overwrite files.
For Jupyter notebooks (.ipynb files), use the NotebookEditCell instead.

Before using this tool:

1. Use the View tool to understand the file's contents and context

2. Verify the directory path is correct (only applicable when creating
new files):
   - Use the LS tool to verify the parent directory exists and is the
correct location

To make a file edit, provide the following:
1. file_path: The absolute path to the file to modify (must be absolute,
not relative)
2. old_string: The text to replace (must be unique within the file, and
must match the file contents exactly, including all whitespace and
indentation)
3. new_string: The edited text to replace the old_string

The tool will replace ONE occurrence of old_string with new_string in the
specified file.

CRITICAL REQUIREMENTS FOR USING THIS TOOL:

1. UNIQUENESS: The old_string MUST uniquely identify the specific
instance you want to change. This means:
   - Include AT LEAST 3-5 lines of context BEFORE the change point
   - Include AT LEAST 3-5 lines of context AFTER the change point
   - Include all whitespace, indentation, and surrounding code exactly as
it appears in the file

2. SINGLE INSTANCE: This tool can only change ONE instance at a time. If
```

```
you need to change multiple instances:
   - Make separate calls to this tool for each instance
   - Each call must uniquely identify its specific instance using
extensive context

3. VERIFICATION: Before using this tool:
   - Check how many instances of the target text exist in the file
   - If multiple instances exist, gather enough context to uniquely
identify each one
   - Plan separate tool calls for each instance

WARNING: If you do not follow these requirements:
   - The tool will fail if old_string matches multiple locations
   - The tool will fail if old_string doesn't match exactly (including
whitespace)
   - You may change the wrong instance if you don't include enough
context

When making edits:
   - Ensure the edit results in idiomatic, correct code
   - Do not leave the code in a broken state
   - Always use absolute file paths (starting with /)

If you want to create a new file, use:
   - A new file path, including dir name if needed
   - An empty old_string
   - The new file's contents as new_string

Remember: when making multiple file edits in a row to the same file, you
should prefer to send all edits in a single message with multiple calls
to this tool, rather than multiple messages with a single call each.`
```

> This is a tool for editing files. For moving or renaming files, you should generally use the Bash tool with the 'mv' command instead. For larger edits, use the Write tool to overwrite files. For Jupyter notebooks (.ipynb files), use the NotebookEditCell instead.
>
> Before using this tool:
>
> 1. Use the View tool to understand the file's contents and context
>
> 2. Verify the directory path is correct (only applicable when creating new files):
>
>   - Use the LS tool to verify the parent directory exists and is the correct location

To make a file edit, provide the following:

1. file_path: The absolute path to the file to modify (must be absolute, not relative)
2. old_string: The text to replace (must be unique within the file, and must match the file contents exactly, including all whitespace and indentation)
3. new_string: The edited text to replace the old_string

The tool will replace ONE occurrence of old_string with new_string in the specified file.

CRITICAL REQUIREMENTS FOR USING THIS TOOL:

1. UNIQUENESS: The old_string MUST uniquely identify the specific instance you want to change. This means:
    - Include AT LEAST 3-5 lines of context BEFORE the change point
    - Include AT LEAST 3-5 lines of context AFTER the change point
    - Include all whitespace, indentation, and surrounding code exactly as it appears in the file
2. SINGLE INSTANCE: This tool can only change ONE instance at a time. If you need to change multiple instances:
    - Make separate calls to this tool for each instance
    - Each call must uniquely identify its specific instance using extensive context
3. VERIFICATION: Before using this tool:
    - Check how many instances of the target text exist in the file
    - If multiple instances exist, gather enough context to uniquely identify each one
    - Plan separate tool calls for each instance

WARNING: If you do not follow these requirements:

- The tool will fail if old_string matches multiple locations
- The tool will fail if old_string doesn't match exactly (including whitespace)
- You may change the wrong instance if you don't include enough context

When making edits:

- Ensure the edit results in idiomatic, correct code

> - Do not leave the code in a broken state
> - Always use absolute file paths (starting with /)
>
> If you want to create a new file, use:
>
> - A new file path, including dir name if needed
> - An empty old_string
> - The new file's contents as new_string
>
> Remember: when making multiple file edits in a row to the same file, you should prefer to send all edits in a single message with multiple calls to this tool, rather than multiple messages with a single call each.

Key Components

The Edit tool supports three main operations:

1. **Operation Types**
    - **Update**: Replace specific text in an existing file
    - **Create**: Generate a new file by using empty `old_string`
    - **Delete**: Remove text by using empty `new_string`
2. **Diff Generation**
    - Shows visual representation of changes
    - Highlights additions and deletions
    - Includes line numbers for context
3. **Validation Process**
    - Checks parameters (file_path, old_string, new_string)
    - Verifies file existence
    - Confirms uniqueness of old_string
    - Validates timestamps to prevent race conditions
4. **File Handling**

- Detects and preserves file encoding
- Maintains line endings (CRLF/LF)
- Creates backups before modifications
- Creates parent directories for new files

5. **Error Handling**

   - Provides specific error messages with suggestions
   - Uses custom error types for different validation issues
   - Includes context in error reports

Workflow

Edit follows a validation-first approach:

```
Parameter Validation
        ↓
File Access Validation
        ↓
Uniqueness Validation
        ↓
Edit Application
        ↓
Result Formatting
```

The process follows these steps:

1. **Parameter Check**

   - Validates all required parameters
   - Ensures file_path is absolute
   - Normalizes paths for consistency

2. **File Validation**

   - Checks if target file exists
   - For new files, verifies parent directory
   - Redirects Jupyter notebooks to appropriate tool

3. **Content Analysis**
    - Reads file with encoding detection
    - Counts occurrences of old_string
    - Verifies uniqueness to prevent unintended edits
4. **Edit Application**
    - Applies the replacement once
    - Creates parent directories as needed
    - Preserves file encoding and line endings
5. **Result Generation**
    - Creates visual diff of changes
    - Returns edited file snippet with line numbers
    - Reports status and changes made

Permissions

Edit uses the standard file permission system with special handling for directory creation:

```
needsPermission(params: Params): boolean {
  // Check if file exists
  const fileExists = existsSync(params.file_path);

  // Always require permission when modifying existing files
  if (fileExists) return true;

  // For new files, check if we have write permission for the parent
directory
  const parentDir = dirname(params.file_path);
  return !hasDirectoryWritePermission(parentDir);
}
```

Key permission features:

1. **Permission Requirements**

- Always requires permission for existing files
- For new files, checks parent directory permissions
- Uses per-directory permission tracking

2. **Permission Interface**

   - Displays file path and change preview
   - Shows diff visualization of proposed changes
   - Provides context for permission requests

3. **Permission Optimization**

   - Caches directory write permissions
   - Reduces permission prompts for the same directory
   - Balances security with user experience

Implementation

The core implementation consists of these components:

1. **FileEditTool React component**

   - TypeScript interface with parameter validation
   - Schema validation for required fields
   - Permission checking for file access
   - Result formatting for display

2. **Edit utility functions**

   - Core file manipulation with encoding preservation
   - String replacement with uniqueness checks
   - Error handling for various cases
   - Diff generation for visual feedback

Core implementation:

```
// Simplified version of the core edit function
async function edit(params: Params): Promise<Result> {
  const { file_path, old_string, new_string } = params;

  // Validation
  if (!existsSync(file_path) && old_string !== "") {
    throw new Error(`File ${file_path} doesn't exist`);
  }

  // Read file content
  let content = "";
  if (existsSync(file_path)) {
    content = await readFile(file_path, "utf8");
  }

  // Check uniqueness
  const matches = content.split(old_string).length - 1;
  if (matches > 1) {
    throw new Error(`Found multiple (${matches}) matches of the provided
text in ${file_path}`);
  }
  if (matches === 0 && old_string !== "") {
    throw new Error(`Could not find the provided text in ${file_path}`);
  }

  // Make the replacement
  const newContent = old_string === ""
    ? new_string
    : content.replace(old_string, new_string);

  // Create parent directories if needed
  const dir = dirname(file_path);
  if (!existsSync(dir)) {
    await mkdir(dir, { recursive: true });
  }

  // Write the new content
  await writeFile(file_path, newContent, "utf8");

  // Generate diff for reporting
  const diff = createPatch(file_path, content, newContent);

  return {
    success: true,
    diff,
    file_path
  };
}
```

Usage Examples

The FileEditTool supports several common usage patterns:

1. **Targeted Code Modification**

```
Edit({
  file_path: "/path/to/file.js",
  old_string: `function sum(a, b) {
  // Old implementation
  return a + b;
}`,
  new_string: `function sum(a, b) {
  // New implementation with validation
  if (typeof a !== 'number' || typeof b !== 'number') {
    throw new Error('Arguments must be numbers');
  }
  return a + b;
}`
})
```

2. **New File Creation**

```
Edit({
  file_path: "/path/to/new/file.js",
  old_string: "",
  new_string: "console.log('Hello world!');"
})
```

3. **Content Removal**

```
Edit({
  file_path: "/path/to/file.js",
  old_string: `  // Deprecated function
  function oldMethod() {
    console.log('This should be removed');
  }`,
  new_string: ""
})
```

4. **Multiple Sequential Edits**

```
Edit({file_path: "/path/to/file.js", old_string: "const version =
'1.0.0';", new_string: "const version = '1.0.1';"})
Edit({file_path: "/path/to/file.js", old_string: "const updated =
false;", new_string: "const updated = true;"})
```

FileEditTool's design focus on safety through uniqueness requirements ensures precise modifications while preventing unintended changes, making it one of the most frequently used yet carefully designed tools in the system.

View: Reading Files

View reads files from the filesystem with support for both text and images, providing direct access to file contents.

Complete Prompt

```
// Tool Prompt: View
const MAX_LINES_TO_READ = 2000
const MAX_LINE_LENGTH = 2000

export const PROMPT = `Reads a file from the local filesystem. The
file_path parameter must be an absolute path, not a relative path. By
default, it reads up to ${MAX_LINES_TO_READ} lines starting from the
beginning of the file. You can optionally specify a line offset and limit
(especially handy for long files), but it's recommended to read the whole
file by not providing these parameters. Any lines longer than
${MAX_LINE_LENGTH} characters will be truncated. For image files, the
tool will display the image for you. For Jupyter notebooks (.ipynb
files), use the ${NotebookReadTool.name} instead.`
```

> **Tool Prompt: View**
>
> Reads a file from the local filesystem. The file_path parameter must be an absolute path, not a relative path. By default, it reads up to 2000 lines starting from the beginning of the file. You can optionally specify a line offset and limit (especially handy for long files), but it's recommended to read the whole file by not providing these parameters. Any lines longer than 2000 characters will be truncated. For image files, the tool will display the image for you. For Jupyter notebooks (.ipynb files), use the ReadNotebook instead.

How It Works

View handles different file types with specialized processing:

1. **Core Components**
    - Input validation using Zod schema
    - Type-specific handlers for different file formats
    - Size and access limits for safe operation
2. **Reading Mechanisms**
    - Different handlers for text and image files
    - Pagination support with offset/limit parameters
    - File size management and optimization

Let's examine the key implementation sections:

```
// File type validation and handling
async validateInput({ file_path, offset, limit }) {
  const fullFilePath = normalizeFilePath(file_path)

  if (!existsSync(fullFilePath)) {
    // Try to find a similar file with a different extension
    const similarFilename = findSimilarFile(fullFilePath)
    let message = 'File does not exist.'

    // If we found a similar file, suggest it to the assistant
    if (similarFilename) {
      message += ` Did you mean ${similarFilename}?`
    }

    return {
      result: false,
      message,
    }
  }

  // Get file stats to check size
  const stats = statSync(fullFilePath)
  const fileSize = stats.size
  const ext = path.extname(fullFilePath).toLowerCase()

  // Skip size check for image files - they have their own size limits
  if (!IMAGE_EXTENSIONS.has(ext)) {
    // If file is too large and no offset/limit provided
    if (fileSize > MAX_OUTPUT_SIZE && !offset && !limit) {
      return {
        result: false,
        message: formatFileSizeError(fileSize),
        meta: { fileSize },
      }
    }
  }

  return { result: true }
}
```

The image handling logic is particularly sophisticated:

```
// Image processing with smart resizing and compression
async function readImage(
  filePath: string,
  ext: string,
): Promise<{
  type: 'image'
  file: { base64: string; type: ImageBlockParam.Source['media_type'] }
```

```
}> {
  try {
    const stats = statSync(filePath)
    const sharp = (await import('sharp')).default
    const image = sharp(readFileSync(filePath))
    const metadata = await image.metadata()

    // Calculate dimensions while maintaining aspect ratio
    let width = metadata.width || 0
    let height = metadata.height || 0

    // Check if the original file is small enough
    if (
      stats.size <= MAX_IMAGE_SIZE &&
      width <= MAX_WIDTH &&
      height <= MAX_HEIGHT
    ) {
      return createImageResponse(readFileSync(filePath), ext)
    }

    // Resize proportionally if needed
    if (width > MAX_WIDTH) {
      height = Math.round((height * MAX_WIDTH) / width)
      width = MAX_WIDTH
    }

    if (height > MAX_HEIGHT) {
      width = Math.round((width * MAX_HEIGHT) / height)
      height = MAX_HEIGHT
    }

    // Resize image and convert to buffer
    const resizedImageBuffer = await image
      .resize(width, height, {
        fit: 'inside',
        withoutEnlargement: true,
      })
      .toBuffer()

    // If still too large after resize, compress quality
    if (resizedImageBuffer.length > MAX_IMAGE_SIZE) {
      const compressedBuffer = await image.jpeg({ quality: 80
}).toBuffer()
      return createImageResponse(compressedBuffer, 'jpeg')
    }

    return createImageResponse(resizedImageBuffer, ext)
  } catch (e) {
    // Fallback to original image if processing fails
    return createImageResponse(readFileSync(filePath), ext)
  }
}
```

Key Features

View offers specialized handling for different file types:

1. **Text Processing**
    - Encoding detection
    - 2000-line default output limit
    - Line truncation for excessive length
    - Size cap for text files
    - Line numbering
2. **Image Processing**
    - Support for common image formats
    - Dynamic resizing for large images
    - Aspect ratio preservation
    - Quality reduction for oversized files
    - Format conversion when needed
3. **User Experience**
    - Similar file suggestions
    - Contextual error messages
    - Pagination for large files
    - Line count display

Architecture

The View tool architecture follows a logical flow:

```
FileReadTool.tsx (React component)
  ↓
validateInput() → Input validation and size checking
  ↓
call() → Dispatch to specific handler based on file type
  ↓
readTextContent() or readImage() → Type-specific processing
  ↓
renderResultForAssistant() → Format for Claude's consumption
```

The tool detects file types by extension and applies the appropriate processing strategy for each type.

Permissions

View uses a simple read permission model:

```
needsPermissions({ file_path }) {
  return !hasReadPermission(file_path || getCwd())
}
```

This verifies read access before displaying file contents. Permissions are requested per directory rather than per file, making multiple reads from the same location more efficient.

Usage Examples

Typical ways to use View:

1. **Reading a complete file**

```
View(file_path: "/path/to/file.txt")
```

2. **Reading part of a large file**

```
View(file_path: "/path/to/large.log", offset: 1000, limit: 100)
```

3. **Viewing images**

```
View(file_path: "/path/to/image.png")
```

View provides core functionality for reading code and content, with type-specific processing for both text and images.

Replace: Creating and Updating Files

Replace writes or overwrites entire files on the filesystem, handling complete file creation or replacement rather than targeted edits.

Complete Prompt

```
export const PROMPT = `Write a file to the local filesystem. Overwrites
the existing file if there is one.

Before using this tool:

1. Use the ReadFile tool to understand the file's contents and context

2. Directory Verification (only applicable when creating new files):
   - Use the LS tool to verify the parent directory exists and is the
correct location`

export const DESCRIPTION = 'Write a file to the local filesystem.'
```

> **Tool Prompt: Replace**
>
> Write a file to the local filesystem. Overwrites the existing file if there is one.
>
> Before using this tool:
>
> 1. Use the ReadFile tool to understand the file's contents and context
> 2. Directory Verification (only applicable when creating new files):
>   - Use the LS tool to verify the parent directory exists and is the correct location

How It Works

Replace takes two required parameters and handles file operations with contextual awareness:

```
const inputSchema = z.strictObject({
  file_path: z
    .string()
    .describe(
      'The absolute path to the file to write (must be absolute, not
relative)',
    ),
  content: z.string().describe('The content to write to the file'),
})
```

The implementation handles both creating new files and updating existing ones with appropriate safety checks:

```
async *call({ file_path, content }, { readFileTimestamps }) {
  const fullFilePath = isAbsolute(file_path)
    ? file_path
    : resolve(getCwd(), file_path)
  const dir = dirname(fullFilePath)
  const oldFileExists = existsSync(fullFilePath)
  const enc = oldFileExists ? detectFileEncoding(fullFilePath) : 'utf-8'
  const oldContent = oldFileExists ? readFileSync(fullFilePath, enc) :
null

  const endings = oldFileExists
    ? detectLineEndings(fullFilePath)
    : await detectRepoLineEndings(getCwd())

  mkdirSync(dir, { recursive: true })
  writeTextContent(fullFilePath, content, enc, endings!)

  // Update read timestamp, to invalidate stale writes
  readFileTimestamps[fullFilePath] = statSync(fullFilePath).mtimeMs

  if (oldContent) {
    const patch = getPatch({
      filePath: file_path,
      fileContents: oldContent,
      oldStr: oldContent,
      newStr: content,
```

```
      })

      // Return update result with diff information
      const data = {
        type: 'update' as const,
        filePath: file_path,
        content,
        structuredPatch: patch,
      }
      yield {
        type: 'result',
        data,
        resultForAssistant: this.renderResultForAssistant(data),
      }
      return
    }

    // Return create result
    const data = {
      type: 'create' as const,
      filePath: file_path,
      content,
      structuredPatch: [],
    }
    yield {
      type: 'result',
      data,
      resultForAssistant: this.renderResultForAssistant(data),
    }
  }
```

Key Features

Replace includes these important capabilities:

1. **Input Validation**
   - Validates file paths
   - Prevents timestamp conflicts
   - Creates directories for new files
2. **Content Management**
   - Detects encoding for existing files (UTF-8, UTF-16LE, ASCII)

    - Preserves line endings (CRLF/LF)
    - Maintains consistent formatting

3. **Change Visualization**

    - Shows diffs for file updates
    - Provides previews for new files
    - Truncates large files for display

4. **Conflict Prevention**

    - Tracks read timestamps
    - Detects external modifications
    - Prevents overwriting recently modified files

Architecture

The Replace tool follows a structured flow:

```
FileWriteTool.tsx (React component)
  ↓
validateInput() → Checks if file can be safely written
  ↓
call() → Writes content with appropriate encoding and line endings
  ↓
renderToolResultMessage() → Shows changes to user
  ↓
renderResultForAssistant() → Returns formatted result for Claude
```

The tool handles two main operations with different display approaches:

- **Create**: Shows complete file content with syntax highlighting
- **Update**: Shows diff between old and new versions

Permissions

Replace enforces a strict permission model:

```
needsPermissions({ file_path }) {
  return !hasWritePermission(file_path)
}
```

This requires explicit user approval before writing to any location. The permission UI shows:

- Content preview for new files
- Structured diff for file updates
- The exact file path being modified

Additional validation safeguards include:

```
async validateInput({ file_path }, { readFileTimestamps }) {
  // Allow creating new files without read check
  if (!existsSync(fullFilePath)) {
    return { result: true }
  }

  // Require existing files to be read first
  const readTimestamp = readFileTimestamps[fullFilePath]
  if (!readTimestamp) {
    return {
      result: false,
      message: 'File has not been read yet. Read it first before writing
to it.',
    }
  }

  // Prevent race conditions with external changes
  const stats = statSync(fullFilePath)
  const lastWriteTime = stats.mtimeMs
  if (lastWriteTime > readTimestamp) {
    return {
      result: false,
      message: 'File has been modified since read... Read it again before
attempting to write it.',
    }
  }

  return { result: true }
}
```

Usage Examples

Common usage patterns:

1. **Creating a new file**

```
Replace(file_path: "/path/to/new-file.txt", content: "New file
content here")
```

2. **Overwriting an existing file**

```
Replace(file_path: "/path/to/existing-file.js", content: "Updated
content here")
```

FileWriteTool complements FileEditTool by handling complete file creation or replacement rather than targeted edits, making it the preferred choice when:

- Creating entirely new files
- Completely rewriting an existing file's contents
- Avoiding multiple sequential edits to the same file

GlobTool: Find Files Fast

GlobTool finds files that match patterns. It works with any codebase size and sorts results by modification time so you see the most recently changed files first.

Complete Prompt

```
// Tool Prompt: GlobTool
export const DESCRIPTION = `- Fast file pattern matching tool that works
with any codebase size
- Supports glob patterns like "**/*.js" or "src/**/*.ts"
- Returns matching file paths sorted by modification time
- Use this tool when you need to find files by name patterns
- When you are doing an open ended search that may require multiple
rounds of globbing and grepping, use the Agent tool instead
`
```

> **Tool Prompt: GlobTool**
>
> - Fast file pattern matching tool that works with any codebase size
> - Supports glob patterns like "**/*.js" or "src/**/*.ts"
> - Returns matching file paths sorted by modification time
> - Use this tool when you need to find files by name patterns
> - When you are doing an open ended search that may require multiple rounds of globbing and grepping, use the Agent tool instead

How It Works

GlobTool has two main parts:

1. **Front-end component** (`GlobTool.tsx`)

- Handles the interface between Claude and the file system
- Validates inputs and formats results
- Manages permissions

2. **Glob function** (`file.ts`)

   - Does the actual file matching with Node's glob library
   - Sorts files by how recently they were changed
   - Limits results to avoid overwhelming Claude

Here's what the core function looks like:

```
// From file.ts - the core file matching function
export async function glob(
  filePattern: string,
  cwd: string,
  { limit, offset }: { limit: number; offset: number },
  abortSignal: AbortSignal,
): Promise<{ files: string[]; truncated: boolean }> {
  // Uses Node's glob library
  const paths = await globLib([filePattern], {
    cwd,
    nocase: true,  // Case-insensitive matching
    nodir: true,   // Only return files, not directories
    signal: abortSignal,  // Support for cancellation
    stat: true,    // Get file stats for sorting
    withFileTypes: true,  // Return full file info objects
  })
  // Sort by modification time (newest last, reversed later)
  const sortedPaths = paths.sort((a, b) => (a.mtimeMs ?? 0) - (b.mtimeMs
?? 0))
  const truncated = sortedPaths.length > offset + limit
  return {
    files: sortedPaths
      .slice(offset, offset + limit)
      .map(path => path.fullpath()),
    truncated,  // Let Claude know if results were limited
  }
}
```

The results get formatted nicely for Claude:

```
// Formats results in a clean, readable format
renderResultForAssistant(output) {
  let result = output.filenames.join('\n')
  if (output.filenames.length === 0) {
    result = 'No files found'
  }
  else if (output.truncated) {
    result +=
      '\n(Results are truncated. Consider using a more specific path or
pattern.)'
  }
  return result
}
```

Key Features

GlobTool provides:

1. **Speed and scale**
    - Works with large codebases
    - Sorts by modification time (newest first)
    - Limits to 100 results
    - Supports cancellation
2. **Pattern support**
    - Standard glob patterns like `**/*.js`
    - Case-insensitive matching
    - File-only results (no directories)
3. **Safety**
    - Indicates truncated results
    - Recommends Agent for complex searches
    - Path permission validation

Architecture

GlobTool follows a simple structure:

```
GlobTool.tsx (Interface)
  ↓
glob() in file.ts (Main function)
  ↓
Node's glob library (Core implementation)
```

As a read-only tool, it:

- Runs concurrently with other tools
- Uses minimal permissions
- Prevents file modifications

Permissions

GlobTool just needs to check if it can read the directory:

```
needsPermissions({ path }) {
  return !hasReadPermission(path || getCwd())
}
```

The system also has these safety measures:

- Normalizes paths to prevent directory traversal
- Rejects paths with null bytes or other weird stuff
- Makes sure operations stay within allowed directories

Examples

Here's how to use it:

1. **Find JavaScript files**

```
GlobTool(pattern: "**/*.js")
```

2. **Look in a specific folder**

```
GlobTool(pattern: "*.ts", path: "/path/to/src")
```

3. **Find config files**

```
GlobTool(pattern: "**/config.{json,yaml,yml}")
```

GlobTool pairs well with GrepTool - first find the files, then search their contents.

GrepTool: Search Inside Files

GrepTool searches through file contents using regex patterns. Built on ripgrep, it's blazing fast even in large codebases.

Complete Prompt

```
// Tool Prompt: GrepTool
export const DESCRIPTION = `
- Fast content search tool that works with any codebase size
- Searches file contents using regular expressions
- Supports full regex syntax (eg. "log.*Error", "function\\s+\\w+", etc.)
- Filter files by pattern with the include parameter (eg. "*.js", "*.
{ts,tsx}")
- Returns matching file paths sorted by modification time
- Use this tool when you need to find files containing specific patterns
- When you are doing an open ended search that may require multiple
rounds of globbing and grepping, use the Agent tool instead
`
```

> **Tool Prompt: GrepTool**
>
> - Fast content search tool that works with any codebase size
> - Searches file contents using regular expressions
> - Supports full regex syntax (eg. "log.*Error", "function\s+\w+", etc.)
> - Filter files by pattern with the include parameter (eg. "*.js", "*.{ts,tsx}")
> - Returns matching file paths sorted by modification time
> - Use this tool when you need to find files containing specific patterns
> - When you are doing an open ended search that may require multiple rounds of globbing and grepping, use the Agent tool instead

How It Works

GrepTool has two main parts:

1. **The UI component** (`GrepTool.tsx`)
    - Validates your search params
    - Handles permissions
    - Formats the results nicely
2. **The ripgrep wrapper** (`ripgrep.ts`)
    - Runs the super-fast Rust-based ripgrep tool
    - Works across all platforms
    - Handles the nitty-gritty details

Here's what the search function looks like:

```
// Core search implementation from GrepTool.tsx
async *call({ pattern, path, include }, { abortController }) {
  const start = Date.now()
  const absolutePath = getAbsolutePath(path) || getCwd()

  // Build ripgrep arguments
  const args = ['-li', pattern]  // -l: files-with-matches, -i: case-
insensitive
  if (include) {
    args.push('--glob', include)  // File pattern filtering
  }

  // Execute ripgrep search
  const results = await ripGrep(args, absolutePath,
abortController.signal)

  // Get file stats for sorting by modification time
  const stats = await Promise.all(results.map(_ => stat(_)))
  const matches = results
    .map((_, i) => [_, stats[i]!] as const)
    .sort((a, b) => {
      // Sort by modification time (newest first)
      return (b[1].mtimeMs ?? 0) - (a[1].mtimeMs ?? 0)
    })
    .map(_ => _[0])

  // Return results
  const output = {
    filenames: matches,
    durationMs: Date.now() - start,
    numFiles: matches.length,
  }

  yield {
    type: 'result',
    resultForAssistant: this.renderResultForAssistant(output),
    data: output,
  }
}
```

And here's how it connects to ripgrep:

```
// From ripgrep.ts - platform-specific optimizations
export async function ripGrep(
  args: string[],
  target: string,
  abortSignal: AbortSignal,
): Promise<string[]> {
  await codesignRipgrepIfNecessary()  // macOS-specific code signing
  const rg = ripgrepPath()  // Get platform-specific binary

  return new Promise(resolve => {
    execFile(
      ripgrepPath(),
      [...args, target],
      {
        maxBuffer: 1_000_000,  // 1MB buffer for large results
        signal: abortSignal,   // Support for cancellation
        timeout: 10_000,       // 10-second timeout
      },
      (error, stdout) => {
        if (error) {
          // Exit code 1 from ripgrep means "no matches found" - this is
normal
          if (error.code !== 1) {
            logError(error)
          }
          resolve([])
        } else {
          resolve(stdout.trim().split('\n').filter(Boolean))
        }
      },
    )
  })
}
```

Key Features

GrepTool provides:

1. **Performance**
    - ripgrep-powered search engine
    - Result limit for responsiveness
    - Truncation indicators

    - Cancellation support

2. **Cross-platform**
    - Bundled ripgrep binaries
    - macOS code signing
    - OS-agnostic path handling

3. **Search capabilities**
    - Regex pattern matching
    - Case-insensitive by default
    - File type filtering
    - Filename-only results

Architecture

GrepTool follows this structure:

```
GrepTool.tsx (UI component)
  ↓
ripGrep() in ripgrep.ts (Main logic)
  ↓
ripgrep CLI binary (The engine)
```

Key design choices:

- Bundled ripgrep binaries for immediate use
- `-li` flags for file-only matching
- Read-only designation allowing parallel execution
- Recency-based result sorting

Permissions

GrepTool keeps permissions simple:

```
needsPermissions({ path }) {
  return !hasReadPermission(path || getCwd())
}
```

It just checks if Claude can read the directory you're searching in. Once you grant access to a directory, it applies to all searches in that directory.

Examples

Here are some common searches:

1. **Find function definitions**

   ```
   GrepTool(pattern: "function\\s+getUserData")
   ```

2. **Search just specific file types**

   ```
   GrepTool(pattern: "import.*React", include: "*.tsx")
   ```

3. **Look for error handling**

   ```
   GrepTool(pattern: "catch.*Error", path: "/path/to/src")
   ```

GrepTool complements GlobTool - use Glob for file name patterns and Grep for content searching. Particularly effective for locating code patterns like function definitions, imports, and error handling.

LS: Directory Listing

LS displays files and directories in a tree structure at a specified path. It helps explore the filesystem structure before performing operations.

Complete Prompt

```
// Tool Prompt: LS
export const DESCRIPTION = 'Lists files and directories in a given path.
The path parameter must be an absolute path, not a relative path. You
should generally prefer the Glob and Grep tools, if you know which
directories to search.'
```

> **Tool Prompt: LS**
>
> Lists files and directories in a given path. The path parameter must be an absolute path, not a relative path. You should generally prefer the Glob and Grep tools, if you know which directories to search.

How It Works

LS implements directory listing through a breadth-first traversal approach:

1. **Core Components**
    - Simple interface with a single path parameter
    - Tree-style rendering for clear visualization
    - Safety checks and permission verification
2. **Directory Traversal**

- Breadth-first search for efficient directory processing
- Tree structure generation for hierarchical display
- Filtering of hidden files and system directories

Let's look at the core implementation:

```
// Core BFS directory traversal function
function listDirectory(
  initialPath: string,
  cwd: string,
  abortSignal: AbortSignal,
): string[] {
  const results: string[] = []
  const queue = [initialPath]

  while (queue.length > 0) {
    // Stop if we hit limits or abort
    if (results.length > MAX_FILES || abortSignal.aborted) {
      return results
    }

    const path = queue.shift()!
    if (skip(path)) {  // Skip hidden files and specific directories
      continue
    }

    if (path !== initialPath) {
      results.push(relative(cwd, path) + sep)
    }

    // Read directory contents
    let children
    try {
      children = readdirSync(path, { withFileTypes: true })
    } catch (e) {
      // Handle permission errors and other exceptions
      logError(e)
      continue
    }

    // Process each child entry
    for (const child of children) {
      if (child.isDirectory()) {
        queue.push(join(path, child.name) + sep)
      } else {
        const fileName = join(path, child.name)
        if (skip(fileName)) {
          continue
        }
        results.push(relative(cwd, fileName))
        if (results.length > MAX_FILES) {
```

```
            return results
          }
        }
      }
    }

    return results
  }
```

The file tree creation and rendering logic is particularly interesting:

```
// Creates a hierarchical tree structure from flat paths
function createFileTree(sortedPaths: string[]): TreeNode[] {
  const root: TreeNode[] = []

  for (const path of sortedPaths) {
    const parts = path.split(sep)
    let currentLevel = root
    let currentPath = ''

    // Build tree nodes for each path component
    for (let i = 0; i < parts.length; i++) {
      const part = parts[i]!
      if (!part) continue // Skip empty parts (trailing slashes)

      currentPath = currentPath ? `${currentPath}${sep}${part}` : part
      const isLastPart = i === parts.length - 1

      // Find existing node or create new one
      const existingNode = currentLevel.find(node => node.name === part)
      if (existingNode) {
        currentLevel = existingNode.children || []
      } else {
        const newNode: TreeNode = {
          name: part,
          path: currentPath,
          type: isLastPart ? 'file' : 'directory',
        }

        if (!isLastPart) {
          newNode.children = []
        }

        currentLevel.push(newNode)
        currentLevel = newNode.children || []
      }
    }
  }

  return root
}
```

Key Features

LS provides these important capabilities:

1. **Directory Analysis**
    - Breadth-first search for efficient traversal
    - Filtering of hidden files and system directories
    - Type detection for files and directories
    - Tree formatting for hierarchical visualization
2. **Safety Measures**
    - 1000 file limit to manage output size
    - Truncation indicators for large directories
    - Error handling for permission issues
    - Security warnings for suspicious files (internal only)
3. **Display Options**
    - Compact mode for terminal display
    - Verbose mode for detailed listing
    - Tree structure for intuitive navigation

Architecture

The LS tool has these main components:

```
LSTool.tsx (React component)
  ↓
listDirectory() (Breadth-first traversal)
  ↓
createFileTree() (Hierarchical structure creation)
  ↓
printTree() (ASCII tree rendering)
```

There's an interesting comment in the code: "TODO: Kill this tool and use bash instead" - suggesting that in a future refactoring, this functionality might be replaced by the Bash tool.

Permissions

LS uses the standard read permission model:

```
needsPermissions({ path }) {
  return !hasReadPermission(path)
}
```

This verifies read access to the specified directory before listing its contents. The permission system prevents access to sensitive system directories.

Usage Examples

Typical use cases:

1. **Exploring project structure**

   ```
   LS(path: "/path/to/project")
   ```

2. **Checking available assets**

   ```
   LS(path: "/path/to/project/assets")
   ```

3. **Verifying directory existence**

   ```
   LS(path: "/path/to/project/src") // Before creating files or running commands
   ```

LS is useful for initial exploration of unfamiliar codebases and for verifying directory

contents before performing operations. It's often one of the first tools used when working with a new project.

MCPTool (MCP): Model Context Protocol Integration

MCPTool implements the Model Context Protocol (MCP), enabling Claude to connect with external tool servers and dynamically extend its capabilities through standardized interfaces.

Complete Prompt

```
// Actual prompt and description are overridden in mcpClient.ts
export const PROMPT = ''
export const DESCRIPTION = ''
```

> **Tool Prompt: MCP**
>
> The prompt for MCPTool is dynamically generated based on the connected MCP server and specific tool. Each MCP-provided tool has its own description and schema that gets exposed to Claude at runtime.

Implementation Details

MCPTool provides a foundational architecture for external tool integration with placeholders that get dynamically customized:

```
// Base MCPTool implementation
const inputSchema = z.object({}).passthrough()

export const MCPTool = {
  // Overridden at runtime
  name: 'mcp',
  async description() {
    return DESCRIPTION  // Overridden per tool
  },
  async prompt() {
    return PROMPT  // Overridden per tool
  },
  inputSchema,
  async *call() {
    yield {
      type: 'result',
      data: '',
      resultForAssistant: '',
    }
  },
  needsPermissions() {
    return true
  },
  userFacingName: () => 'mcp',  // Overridden per tool
  renderToolResultMessage(output, { verbose }) {
    // Handles various output formats including text and images
  }
}
```

The actual implementation extends this base through the MCP client service:

```
// Dynamic tool creation in mcpClient.ts
export const getMCPTools = memoize(async (): Promise<Tool[]> => {
  const toolsList = await requestAll<ListToolsResult>(
    { method: 'tools/list' },
    ListToolsResultSchema,
    'tools',
  )

  return toolsList.flatMap(({ client, result: { tools } }) =>
    tools.map((tool): Tool => ({
      ...MCPTool,
      name: 'mcp__' + client.name + '__' + tool.name,
      async description() {
        return tool.description ?? ''
      },
      async prompt() {
        return tool.description ?? ''
      },
      inputJSONSchema: tool.inputSchema as Tool['inputJSONSchema'],
      async *call(args: Record<string, unknown>) {
        const data = await callMCPTool({ client, tool: tool.name, args })
        yield {
          type: 'result',
          data,
          resultForAssistant: data,
        }
      },
      userFacingName() {
        return `${client.name}:${tool.name} (MCP)`
      },
    })),
  )
})
```

Key Components

MCPTool provides several critical features:

1. **Dynamic Tool Creation**
   - Base skeleton tool that gets customized at runtime
   - Tool name format: `mcp__[serverName]__[toolName]`
   - User-facing name: `[serverName]:[toolName] (MCP)`

    - Passthrough schema to support any input object

2. **Server Configuration Management**
    - Three-level configuration scope hierarchy:
        - Project-specific (highest priority)
        - `.mcprc` file (middle priority)
        - Global configuration (lowest priority)
    - Security model for server approval
    - Server connection tracking and error handling

3. **Flexible Transport Mechanisms**
    - `StdioClientTransport` for command-line based servers
    - `SSEClientTransport` for HTTP servers with Server-Sent Events
    - Connection timeouts and retry mechanisms

4. **Output Processing**
    - Support for text outputs formatted as strings
    - Image handling with base64 encoding
    - Multi-part content arrays
    - Standardized error reporting

Architecture

The MCPTool implementation follows a layered architecture:

```
MCPTool (Base)
  ↓
getMCPTools() → Discovers and converts server tools to Claude tools
  ↓
getClients() → Manages connections to registered MCP servers
  ↓
requestAll() → Sends requests to all connected servers
  ↓
callMCPTool() → Executes specific tool on a specific server
  ↓
Result Processing → Formats outputs for Claude's consumption
```

This architecture prioritizes:

- **Extensibility**: Easy addition of new MCP servers and tools
- **Isolation**: Each server operates independently
- **Fault Tolerance**: Failures in one server don't affect others
- **Standardization**: Consistent interfaces regardless of backend

Configuration System

MCPTool employs a sophisticated configuration system:

```
export function addMcpServer(
  name: McpName,
  server: McpServerConfig,
  scope: ConfigScope = 'project',
): void {
  if (scope === 'mcprc') {
    // Write to .mcprc file in current directory
  } else if (scope === 'global') {
    // Update global configuration
    const config = getGlobalConfig()
    if (!config.mcpServers) {
      config.mcpServers = {}
    }
    config.mcpServers[name] = server
    saveGlobalConfig(config)
  } else {
    // Update project-specific configuration
    const config = getCurrentProjectConfig()
    if (!config.mcpServers) {
      config.mcpServers = {}
    }
    config.mcpServers[name] = server
    saveCurrentProjectConfig(config)
  }
}
```

The configuration prioritization ensures:

- Project-specific settings override all others
- `.mcprc` settings override global ones
- Security approval required for certain server types

- Configuration persistence across sessions

Usage Examples

MCPTool is primarily used through the dynamically generated tools. From a user perspective:

1. **Server Registration**

```
kode mcp add myserver --command="python -m my_mcp_server"
```

2. **Tool Usage (for an example calculator server)**

```
myserver:calculate(expression: "2 + 2 * 10")
```

3. **Server Configuration**

```
kode mcp list
kode mcp approve myserver
```

MCPTool significantly extends Claude's capabilities by enabling integration with external tools and services through a standardized protocol. It allows Claude to:

- Access specialized functionality implemented in any language
- Interact with external APIs and services through mediator servers
- Utilize domain-specific tools beyond the core toolset
- Access private tools specific to a user's project

MemoryRead: Session Memory Access

MemoryRead retrieves information stored in a persistent memory system, enabling state preservation across conversations.

> **Note**: An interesting pattern in this code - the tool is included in an `ANT_ONLY_TOOLS` array in `tools.ts` with empty prompts defined in their respective `prompt.ts` files. The tools are disabled by default as evidenced by the `isEnabled()` method returning `false`.
>
> When building an agentic system, persistent memory like this provides critical functionality. The filesystem-based approach shown here is straightforward to implement yet powerful - enabling agents to retain information across sessions without complex infrastructure. The tool's separation of read/write operations and simple directory-based structure makes it easy to debug and maintain.

Complete Prompt

```
// Actual prompt and description are defined elsewhere in the application
export const PROMPT = ''
export const DESCRIPTION = ''
```

> **Tool Prompt: MemoryRead**
>
> Accesses the Claude memory system to retrieve stored information from previous sessions. The memory system allows Claude to maintain context and data across conversations. When called without a file path, lists all available memory files along with the contents of the root index.md file.

How It Works

MemoryRead provides a simple interface to access persistent memory:

```
const inputSchema = z.strictObject({
  file_path: z
    .string()
    .optional()
    .describe('Optional path to a specific memory file to read'),
})
```

The implementation enables two modes of operation - listing available memory files or reading a specific file:

```
async *call({ file_path }) {
  mkdirSync(MEMORY_DIR, { recursive: true })

  // If a specific file is requested, return its contents
  if (file_path) {
    const fullPath = join(MEMORY_DIR, file_path)
    if (!existsSync(fullPath)) {
      throw new Error('Memory file does not exist')
    }
    const content = readFileSync(fullPath, 'utf-8')
    yield {
      type: 'result',
      data: { content },
      resultForAssistant: this.renderResultForAssistant({ content }),
    }
    return
  }

  // Otherwise return the index and file list
  const files = readdirSync(MEMORY_DIR, { recursive: true })
    .map(f => join(MEMORY_DIR, f.toString()))
    .filter(f => !lstatSync(f).isDirectory())
    .map(f => `- ${f}`)
    .join('\n')

  const indexPath = join(MEMORY_DIR, 'index.md')
  const index = existsSync(indexPath) ? readFileSync(indexPath, 'utf-8')
: ''

  const quotes = "'''"
  const content = `Here are the contents of the root memory file,
\`${indexPath}\`:
${quotes}
${index}
${quotes}

Files in the memory directory:
${files}`
  yield {
    type: 'result',
    data: { content },
    resultForAssistant: this.renderResultForAssistant({ content }),
  }
}
```

Key Features

MemoryRead provides these important capabilities:

1. **Query Modes**
   - Index listing: Shows all available memory files
   - File access: Reads a specific memory file
   - Index content: Returns the main index.md content
2. **Storage System**
   - Files stored in `~/.koding/memory` directory
   - Creates directory structure as needed
   - Supports nested subdirectories
3. **Security Measures**
   - Path validation to prevent traversal attacks

```
async validateInput({ file_path }) {
  if (file_path) {
    const fullPath = join(MEMORY_DIR, file_path)
    if (!fullPath.startsWith(MEMORY_DIR)) {
      return { result: false, message: 'Invalid memory file path' }
    }
    if (!existsSync(fullPath)) {
      return { result: false, message: 'Memory file does not exist'
}
    }
  }
  return { result: true }
}
```

   - Existence verification before access
   - Read-only operation (paired with MemoryWrite)
4. **Configuration**
   - Currently disabled by default

```
async isEnabled() {
  // TODO: Use a statsig gate
  // TODO: Figure out how to do that without regressing app startup
perf
  return false
}
```

- Read-only functionality
- No permissions required

Architecture

The MemoryRead tool follows a simple, filesystem-based architecture:

```
MemoryReadTool
  ↓
Input Validation → Checks path safety and file existence
  ↓
Storage Access → Directory creation and file reading
  ↓
Content Formatting → Index listing or individual file content
  ↓
Result Generation → Well-formatted output for Claude
```

Design priorities include:

- **Simplicity**: Direct filesystem access without caching
- **Persistence**: Files stored on disk for longevity
- **Flexibility**: Support for both listing and targeted reading
- **Security**: Path validation to prevent directory traversal

Integration

MemoryRead works with MemoryWrite as a complementary pair:

```
// No explicit permissions required
needsPermissions() {
  return false
}

// Read-only operation
isReadOnly() {
  return true
}
```

Together, these tools provide a simple but effective persistent storage system that maintains state between conversations. If enabled, these tools would:

1. Allow users to maintain persistent data across multiple sessions
2. Not require permission prompts (`needsPermissions()` returns `false`)
3. Store data files in the `~/.koding/memory` directory
4. Support structured information organization with directories and an index file

Usage Examples

Typical use cases:

1. **Retrieving memory index**

   ```
   MemoryRead()
   ```

2. **Reading specific files**

   ```
   MemoryRead(file_path: "user_preferences.md")
   ```

3. **Accessing nested data**

```
MemoryRead(file_path: "projects/my_project/notes.md")
```

Key benefits of the memory system:

- Persists information across different sessions
- Maintains context for recurring tasks
- Stores user preferences and settings
- Builds knowledge specific to the user

Memory System Design Patterns

The memory implementation here offers several patterns useful for agentic systems:

1. **Hierarchical Storage**: Using a directory structure creates natural organization for different types of memory (preferences, project context, etc.)

2. **Index-based Access**: The root index.md provides a table of contents, making memory contents discoverable and self-documenting

3. **Plain Text Storage**: Using Markdown files keeps data human-readable and easy to inspect/debug

4. **Path Validation**: Security checks prevent directory traversal attacks while allowing flexible memory structures

5. **Minimal Dependencies**: The filesystem-based approach works across platforms with no external databases

For developers building similar systems, this demonstrates how to implement long-term memory with minimal overhead while maintaining security and extensibility.

MemoryWrite: Storing Session Data

MemoryWrite saves information to a persistent storage system, enabling data preservation across conversations.

> **Note**: This tool follows the same pattern as MemoryRead - part of `ANT_ONLY_TOOLS` array in `tools.ts` with an empty prompt defined in its `prompt.ts` file. The tool is disabled by default with `isEnabled()` returning `false`.
>
> The separation of read and write operations into distinct tools follows good engineering practices. For agentic system developers, this pattern provides clear security boundaries and simpler permission models. The filesystem-based implementation requires minimal dependencies and offers a straightforward path to more sophisticated storage solutions as needs evolve.

Complete Prompt

```
// Actual prompt and description are defined elsewhere in the application
export const PROMPT = ''
export const DESCRIPTION = ''
```

> **Tool Prompt: MemoryWrite**
>
> Writes data to the Claude memory system to store information for future sessions. The memory system allows Claude to maintain context and data across conversations. This lets you save structured information that can be retrieved in later sessions with the MemoryRead tool.

How It Works

MemoryWrite provides a straightforward interface for storing persistent data:

```
const inputSchema = z.strictObject({
  file_path: z.string().describe('Path to the memory file to write'),
  content: z.string().describe('Content to write to the file'),
})
```

The implementation is concise, focusing on safely storing data in a consistent location:

```
async *call({ file_path, content }) {
  const fullPath = join(MEMORY_DIR, file_path)
  mkdirSync(dirname(fullPath), { recursive: true })
  writeFileSync(fullPath, content, 'utf-8')
  yield {
    type: 'result',
    data: 'Saved',
    resultForAssistant: 'Saved',
  }
}
```

Key Features

MemoryWrite provides these important capabilities:

1. **Storage Management**
   - Writes files to `~/.koding/memory` directory
   - Creates parent directories automatically
   - Supports nested directory structure
2. **Security Measures**
   - Path validation to prevent traversal attacks

```
async validateInput({ file_path }) {
  const fullPath = join(MEMORY_DIR, file_path)
  if (!fullPath.startsWith(MEMORY_DIR)) {
    return { result: false, message: 'Invalid memory file path' }
  }
  return { result: true }
}
```

- Restricts writes to the designated memory directory

3. **Configuration**
   - Currently disabled by default

```
async isEnabled() {
  // TODO: Use a statsig gate
  // TODO: Figure out how to do that without regressing app startup
perf
  return false
}
```

   - Write operation (not read-only)
   - No permissions required

```
isReadOnly() {
  return false
}

needsPermissions() {
  return false
}
```

Architecture

The MemoryWrite tool follows a simple, direct architecture:

```
MemoryWriteTool
  ↓
Input Validation → Checks path safety
  ↓
Directory Creation → Ensures parent directories exist
  ↓
File Writing → Saves content to disk
  ↓
Result Generation → Returns success confirmation
```

Design priorities include:

- **Simplicity**: Direct filesystem access
- **Reliability**: Creates directories as needed
- **Security**: Validates paths before writing
- **Flexibility**: Supports any text content

Integration

MemoryWrite works with MemoryRead as a complementary pair:

```
// No explicit permissions required
needsPermissions() {
  return false
}

// Not a read-only operation
isReadOnly() {
  return false
}
```

Together, these tools provide a simple but effective persistent storage system that maintains state between conversations. If enabled, these tools would:

1. Allow users to persistently store information that survives across sessions
2. Not require permission prompts (`needsPermissions()` returns `false`)

3. Create and write files to the `~/.koding/memory` directory
4. Support hierarchical data organization through directory structure
5. Allow for creating structured knowledge bases that can be accessed later

Usage Examples

Typical use cases:

1. **Storing user preferences**

```
MemoryWrite(file_path: "user_preferences.md", content: "Theme:
dark\nIndentation: 2 spaces")
```

2. **Saving project context**

```
MemoryWrite(file_path: "projects/my_project/context.md", content: "#
Project Overview\nThis is a React application with TypeScript...")
```

3. **Creating structured information**

```
MemoryWrite(file_path: "index.md", content: "# Memory Index\n- [User
Preferences](user_preferences.md)\n- [Projects](projects/)")
```

Memory System Implementation Patterns

This write tool demonstrates practical patterns for agentic system memory:

1. **Fixed Storage Location**: Using a dedicated path (`~/.koding/memory`) isolates persistent data from application code
2. **Automatic Path Creation**: The tool handles directory creation, letting developers focus on data structure rather than filesystem operations
3. **Path Validation**: Security checks prevent writing outside the designated memory directory
4. **Write-only Design**: Separation of read/write operations follows the principle of least privilege
5. **Permissionless Operation**: By operating within a constrained scope, the tool avoids disrupting user flow with permission requests

These patterns can be directly applied when implementing similar memory systems, balancing simplicity with security.

NotebookEditTool (NotebookEditCell): Jupyter Notebook Modification

NotebookEditTool enables precise modification of Jupyter notebooks by editing, inserting, or deleting individual cells while preserving the notebook's structure and metadata.

Complete Prompt

```
export const DESCRIPTION =
  'Replace the contents of a specific cell in a Jupyter notebook.'
export const PROMPT = `Completely replaces the contents of a specific
cell in a Jupyter notebook (.ipynb file) with new source. Jupyter
notebooks are interactive documents that combine code, text, and
visualizations, commonly used for data analysis and scientific computing.
The notebook_path parameter must be an absolute path, not a relative
path. The cell_number is 0-indexed. Use edit_mode=insert to add a new
cell at the index specified by cell_number. Use edit_mode=delete to
delete the cell at the index specified by cell_number.`
```

> **Tool Prompt: NotebookEditCell**
>
> Completely replaces the contents of a specific cell in a Jupyter notebook (.ipynb file) with new source. Jupyter notebooks are interactive documents that combine code, text, and visualizations, commonly used for data analysis and scientific computing. The notebook_path parameter must be an absolute path, not a relative path. The cell_number is 0-indexed. Use edit_mode=insert to add a new cell at the index specified by cell_number. Use edit_mode=delete to delete the cell at the index specified by cell_number.

Implementation Details

NotebookEditTool validates inputs and handles notebook modifications with careful preservation of structure:

```
const inputSchema = z.strictObject({
  notebook_path: z
    .string()
    .describe(
      'The absolute path to the Jupyter notebook file to edit (must be
absolute, not relative)',
    ),
  cell_number: z.number().describe('The index of the cell to edit (0-
based)'),
  new_source: z.string().describe('The new source for the cell'),
  cell_type: z
    .enum(['code', 'markdown'])
    .optional()
    .describe(
      'The type of the cell (code or markdown). If not specified, it
defaults to the current cell type. If using edit_mode=insert, this is
required.',
    ),
  edit_mode: z
    .string()
    .optional()
    .describe(
      'The type of edit to make (replace, insert, delete). Defaults to
replace.',
    ),
})
```

The core implementation handles the three editing modes and manages notebook structure:

```
async *call({
  notebook_path,
  cell_number,
  new_source,
  cell_type,
  edit_mode,
}) {
  const fullPath = isAbsolute(notebook_path)
    ? notebook_path
```

```
    : resolve(getCwd(), notebook_path)

  try {
    const enc = detectFileEncoding(fullPath)
    const content = readFileSync(fullPath, enc)
    const notebook = JSON.parse(content) as NotebookContent
    const language = notebook.metadata.language_info?.name ?? 'python'

    if (edit_mode === 'delete') {
      // Delete the specified cell
      notebook.cells.splice(cell_number, 1)
    } else if (edit_mode === 'insert') {
      // Insert the new cell
      const new_cell = {
        cell_type: cell_type!, // validateInput ensures cell_type is not
undefined
        source: new_source,
        metadata: {},
      }
      notebook.cells.splice(
        cell_number,
        0,
        cell_type == 'markdown' ? new_cell : { ...new_cell, outputs: []
},
      )
    } else {
      // Replace the specified cell's content
      const targetCell = notebook.cells[cell_number]! // validateInput
ensures cell_number is in bounds
      targetCell.source = new_source
      // Reset execution count and clear outputs since cell was modified
      targetCell.execution_count = undefined
      targetCell.outputs = []
      if (cell_type && cell_type !== targetCell.cell_type) {
        targetCell.cell_type = cell_type
      }
    }

    // Write back to file
    const endings = detectLineEndings(fullPath)
    writeTextContent(
      fullPath,
      JSON.stringify(notebook, null, 1),
      enc,
      endings!,
    )

    // Return success result
    yield {
      type: 'result',
      data: { /* result data */ },
      resultForAssistant: this.renderResultForAssistant(data),
    }
  } catch (error) {
```

```
      // Handle and report errors
      // ...
    }
  }
```

Key Components

NotebookEditTool has several important features:

1. **Multiple Editing Modes**
    - `replace` : Updates existing cell content (default mode)
    - `insert` : Adds a new cell at a specified index
    - `delete` : Removes a cell at a specified index
2. **Comprehensive Validation**
    - Verifies file exists and has .ipynb extension
    - Checks cell_number is within bounds for the operation
    - Ensures required parameters like cell_type are provided when needed
    - Validates edit_mode is one of the supported operations
3. **Notebook Structure Preservation**
    - Maintains notebook metadata and overall structure
    - Handles different cell types appropriately
    - Clears execution counts and outputs on modified cells
    - Preserves file encoding and line endings
4. **Robust Error Handling**
    - Handles JSON parsing errors
    - Reports specific error messages for validation failures
    - Provides user-friendly error reporting

Architecture

The NotebookEditTool follows a structured workflow:

```
NotebookEditTool
  ↓
Input Validation → Checks for valid operations and boundaries
  ↓
Notebook Loading → Reads and parses notebook JSON
  ↓
Cell Modification → Applies changes based on edit_mode
  ↓
Structure Cleanup → Resets execution counts and outputs
  ↓
File Writing → Preserves encoding and formatting
```

The architecture prioritizes:

- **Data integrity**: Preserves notebook format and metadata
- **Consistency**: Cleans execution state for modified cells
- **Flexibility**: Supports multiple editing operations
- **Safety**: Validates operations before modifying files

Permission Handling

NotebookEditTool integrates with the permission system:

```
needsPermissions({ notebook_path }) {
  return !hasWritePermission(notebook_path)
}
```

This requires explicit user permission before modifying any notebook file, ensuring users maintain control over their data science workflows and preventing accidental modifications.

Usage Examples

Common usage patterns:

1. **Replacing a code cell's content**

```
NotebookEditCell(
  notebook_path: "/path/to/notebook.ipynb",
  cell_number: 2,
  new_source: "import pandas as pd\npd.read_csv('data.csv')"
)
```

2. **Inserting a new markdown cell**

```
NotebookEditCell(
  notebook_path: "/path/to/notebook.ipynb",
  cell_number: 0,
  new_source: "# Data Analysis\nThis notebook explores dataset
trends.",
  cell_type: "markdown",
  edit_mode: "insert"
)
```

3. **Deleting an unwanted cell**

```
NotebookEditCell(
  notebook_path: "/path/to/notebook.ipynb",
  cell_number: 5,
  new_source: "",
  edit_mode: "delete"
)
```

NotebookEditTool complements NotebookReadTool to provide a complete suite for working with Jupyter notebooks, enabling Claude to help users maintain and modify

their data science workflows while respecting notebook structure and conventions.

NotebookReadTool (ReadNotebook): Jupyter Notebook Inspection

NotebookReadTool specializes in reading Jupyter notebook (.ipynb) files and extracting their contents with outputs, enabling Claude to analyze data science workflows in their native format.

Complete Prompt

```
export const DESCRIPTION =
  'Extract and read source code from all code cells in a Jupyter
notebook.'
export const PROMPT = `Reads a Jupyter notebook (.ipynb file) and returns
all of the cells with their outputs. Jupyter notebooks are interactive
documents that combine code, text, and visualizations, commonly used for
data analysis and scientific computing. The notebook_path parameter must
be an absolute path, not a relative path.`
```

> **Tool Prompt: ReadNotebook**
>
> Reads a Jupyter notebook (.ipynb file) and returns all of the cells with their outputs. Jupyter notebooks are interactive documents that combine code, text, and visualizations, commonly used for data analysis and scientific computing. The notebook_path parameter must be an absolute path, not a relative path.

Implementation Details

NotebookReadTool takes a single parameter and handles notebook parsing with

specialized formatting:

```
const inputSchema = z.strictObject({
  notebook_path: z
    .string()
    .describe(
      'The absolute path to the Jupyter notebook file to read (must be
absolute, not relative)',
    ),
})
```

The core implementation parses notebook JSON and processes each cell:

```
async *call({ notebook_path }) {
  const fullPath = isAbsolute(notebook_path)
    ? notebook_path
    : resolve(getCwd(), notebook_path)

  const content = readFileSync(fullPath, 'utf-8')
  const notebook = JSON.parse(content) as NotebookContent
  const language = notebook.metadata.language_info?.name ?? 'python'
  const cells = notebook.cells.map((cell, index) =>
    processCell(cell, index, language),
  )

  yield {
    type: 'result',
    resultForAssistant: renderResultForAssistant(cells),
    data: cells,
  }
}
```

Individual cells and their outputs are processed with specialized functions:

```
function processCell(
  cell: NotebookCell,
  index: number,
  language: string,
): NotebookCellSource {
  const cellData: NotebookCellSource = {
    cell: index,
    cellType: cell.cell_type,
    source: Array.isArray(cell.source) ? cell.source.join('') :
cell.source,
    language,
    execution_count: cell.execution_count,
  }

  if (cell.outputs?.length) {
    cellData.outputs = cell.outputs.map(processOutput)
  }

  return cellData
}

function processOutput(output: NotebookCellOutput) {
  switch (output.output_type) {
    case 'stream':
      return {
        output_type: output.output_type,
        text: processOutputText(output.text),
      }
    case 'execute_result':
    case 'display_data':
      return {
        output_type: output.output_type,
        text: processOutputText(output.data?.['text/plain']),
        image: output.data && extractImage(output.data),
      }
    case 'error':
      return {
        output_type: output.output_type,
        text: processOutputText(
          `${output.ename}:
${output.evalue}\n${output.traceback.join('\n')}`,
        ),
      }
  }
}
```

Key Components

NotebookReadTool has several critical features:

1. **Complete Notebook Parsing**
    - Parses the notebook's JSON structure
    - Processes both code and markdown cells
    - Handles cell ordering and indexing
2. **Cell Content Extraction**
    - Combines multi-line source content
    - Preserves cell execution counts
    - Maintains language information
    - Detects cell types (code vs. markdown)
3. **Output Type Processing**
    - Handles stream outputs (stdout/stderr)
    - Processes execution results
    - Extracts display data
    - Formats error information with tracebacks
4. **Image Handling**
    - Extracts base64-encoded PNG and JPEG images
    - Preserves media type information
    - Formats images for Claude's display capabilities

Architecture

The NotebookReadTool follows a sequential processing flow:

```
NotebookReadTool
  ↓
Validation → Checks file exists and has .ipynb extension
  ↓
Notebook Loading → Reads and parses JSON structure
  ↓
Cell Processing → Maps each cell through processCell()
  ↓
Output Processing → Handles different output types
  ↓
Result Formatting → Structures content for Claude
```

The architecture prioritizes:

- **Complete Data Capture**: Extracts all notebook components
- **Output Type Support**: Handles the variety of Jupyter outputs
- **Structured Presentation**: Formats cells with clear tags
- **Visual Content**: Preserves images and visualization outputs

Permission Handling

NotebookReadTool integrates with the permission system:

```
needsPermissions({ notebook_path }) {
  return !hasReadPermission(notebook_path)
}
```

This requires explicit read permission for notebook files, following the same permission model as other file reading tools.

Usage Examples

Common usage patterns:

1. **Reading a data analysis notebook**

```
ReadNotebook(notebook_path: "/path/to/analysis.ipynb")
```

2. **Examining notebook with visualizations**

```
ReadNotebook(notebook_path: "/path/to/visualization.ipynb")
```

NotebookReadTool enables Claude to understand and analyze Jupyter notebooks, which are commonly used for:

- Data analysis workflows
- Machine learning model development
- Scientific research and experimentation
- Educational content with executable code

The tool's ability to extract both code and outputs makes it valuable for understanding complete computational narratives, not just the code itself.

StickerRequestTool (StickerRequest): User Feedback Collection

StickerRequestTool provides an interactive shipping form for users to request physical Anthropic/Claude stickers, creating a unique engagement opportunity that bridges the digital and physical experience.

Complete Prompt

```
export const DESCRIPTION =
  'Sends the user swag stickers with love from Anthropic.'
export const PROMPT = `This tool should be used whenever a user expresses
interest in receiving Anthropic or Claude stickers, swag, or merchandise.
When triggered, it will display a shipping form for the user to enter
their mailing address and contact details. Once submitted, Anthropic will
process the request and ship stickers to the provided address.

Common trigger phrases to watch for:
- "Can I get some Anthropic stickers please?"
- "How do I get Anthropic swag?"
- "I'd love some Claude stickers"
- "Where can I get merchandise?"
- Any mention of wanting stickers or swag

The tool handles the entire request process by showing an interactive
form to collect shipping information.

NOTE: Only use this tool if the user has explicitly asked us to send or
give them stickers. If there are other requests that include the word
"sticker", but do not explicitly ask us to send them stickers, do not use
this tool.
For example:
- "How do I make custom stickers for my project?" - Do not use this tool
- "I need to store sticker metadata in a database - what schema do you
recommend?" - Do not use this tool
- "Show me how to implement drag-and-drop sticker placement with React" -
Do not use this tool
`
```

Tool Prompt: StickerRequest

This tool should be used whenever a user expresses interest in receiving Anthropic or Claude stickers, swag, or merchandise. When triggered, it will display a shipping form for the user to enter their mailing address and contact details. Once submitted, Anthropic will process the request and ship stickers to the provided address.

Common trigger phrases to watch for:

- "Can I get some Anthropic stickers please?"
- "How do I get Anthropic swag?"
- "I'd love some Claude stickers"
- "Where can I get merchandise?"
- Any mention of wanting stickers or swag

The tool handles the entire request process by showing an interactive form to collect shipping information.

NOTE: Only use this tool if the user has explicitly asked us to send or give them stickers. If there are other requests that include the word "sticker", but do not explicitly ask us to send them stickers, do not use this tool.

Implementation Details

StickerRequestTool uses React and Ink to display an interactive form within the terminal:

```
const stickerRequestSchema = z.object({
  trigger: z.string(),
})

export const StickerRequestTool: Tool = {
  name: 'StickerRequest',
  userFacingName: () => 'Stickers',
```

```
  async *call(_, context: ToolUseContext) {
    // Log form entry event
    logEvent('sticker_request_form_opened', {})

    // Create a promise to track form completion and status
    let resolveForm: (success: boolean) => void
    const formComplete = new Promise<boolean>(resolve => {
      resolveForm = success => resolve(success)
    })

    // Replace the input prompt with the interactive form
    context.setToolJSX?.({
      jsx: (
        <StickerRequestForm
          onSubmit={(formData: FormData) => {
            logEvent('sticker_request_form_completed', {
              has_address: Boolean(formData.address1).toString(),
              has_optional_address:
Boolean(formData.address2).toString(),
            })
            resolveForm(true)
            context.setToolJSX?.(null) // Clear the JSX
          }}
          onClose={() => {
            logEvent('sticker_request_form_cancelled', {})
            resolveForm(false)
            context.setToolJSX?.(null) // Clear the JSX
          }}
        />
      ),
      shouldHidePromptInput: true,
    })

    // Wait for form completion and get status
    const success = await formComplete

    if (!success) {
      context.abortController.abort()
      throw new Error('Sticker request cancelled')
    }

    // Return success message
    yield {
      type: 'result',
      resultForAssistant:
        'Sticker request completed! Please tell the user that they will
receive stickers in the mail if they have submitted the form!',
      data: { success },
    }
  },
}
```

Key Components

StickerRequestTool demonstrates several advanced capabilities:

1. **Interactive UI Integration**
    - Replaces the standard CLI prompt with a form
    - Uses React and Ink for terminal-based UI rendering
    - Restores normal interface when complete
2. **Asynchronous Flow Control**
    - Promise-based mechanism to track form completion
    - Maintains conversation state during form interaction
    - Handles cancellation gracefully with proper cleanup
3. **Analytics Integration**
    - Logs form events with Statsig:
        - Form opened
        - Form completed with data quality indicators
        - Form cancelled
    - Enables usage tracking and engagement measurement
4. **Feature Flag Control**

```
isEnabled: async () => {
  const enabled = await checkGate('tengu_sticker_easter_egg')
  return enabled
}
```

    - Controls availability through Statsig feature flags
    - Can be enabled/disabled without code changes

Architecture

StickerRequestTool follows a specialized interactive flow:

```
StickerRequestTool
  ↓
Form Rendering → Replaces standard prompt with StickerRequestForm
  ↓
User Interaction → Collects shipping details through form fields
  ↓
Submit/Cancel → Triggers promise resolution based on user action
  ↓
Result Generation → Formats success message or handles cancellation
```

This architecture demonstrates:

- **Temporary UI Replacement**: Context switching within the CLI
- **Promise-Based State**: Async pattern for multi-step interactions
- **Analytics Integration**: Event tracking for user engagement
- **Graceful Error Handling**: Proper cancellation and cleanup

User Experience Flow

The StickerRequestTool creates a unique interaction pattern:

1. User requests stickers
2. Claude recognizes the request and triggers the tool
3. Standard input is temporarily replaced with an interactive form
4. User enters shipping information in the multi-field form
5. On completion, the form disappears and regular conversation resumes
6. Claude confirms the successful submission

This pattern showcases how Claude can manage complex, multi-step interactions beyond simple text exchanges.

Usage Examples

The tool is designed to respond to specific user requests like:

```
User: Can I get some Claude stickers?
Assistant: [Uses StickerRequest tool to show shipping form]
User: [Completes form]
Assistant: Great! Your sticker request has been submitted. You'll receive
them in the mail soon!
```

StickerRequestTool demonstrates how Claude Code can:

- Handle interactive multi-step user input
- Bridge digital and physical experiences
- Manage temporary UI context switches
- Track user engagement through analytics
- Support feature flagging for gradual rollouts

This tool showcases the flexibility of the terminal UI framework and provides a tangible way for users to connect with the Claude brand.

ThinkTool (Think): Internal Reasoning Capabilities

ThinkTool provides Claude with a dedicated mechanism to externalize its reasoning process, making complex thinking visible to users through a distinct visual representation.

Complete Prompt

```
export const DESCRIPTION =
  'This is a no-op tool that logs a thought. It is inspired by the tau-
bench think tool.'
export const PROMPT = `Use the tool to think about something. It will not
obtain new information or make any changes to the repository, but just
log the thought. Use it when complex reasoning or brainstorming is
needed.

Common use cases:
1. When exploring a repository and discovering the source of a bug, call
this tool to brainstorm several unique ways of fixing the bug, and assess
which change(s) are likely to be simplest and most effective
2. After receiving test results, use this tool to brainstorm ways to fix
failing tests
3. When planning a complex refactoring, use this tool to outline
different approaches and their tradeoffs
4. When designing a new feature, use this tool to think through
architecture decisions and implementation details
5. When debugging a complex issue, use this tool to organize your
thoughts and hypotheses

The tool simply logs your thought process for better transparency and
does not execute any code or make changes.`
```

> **Tool Prompt: Think**
>
> Use the tool to think about something. It will not obtain new information or make any changes to the repository, but just log the thought. Use it when

complex reasoning or brainstorming is needed.

Common use cases:

1. When exploring a repository and discovering the source of a bug, call this tool to brainstorm several unique ways of fixing the bug, and assess which change(s) are likely to be simplest and most effective
2. After receiving test results, use this tool to brainstorm ways to fix failing tests
3. When planning a complex refactoring, use this tool to outline different approaches and their tradeoffs
4. When designing a new feature, use this tool to think through architecture decisions and implementation details
5. When debugging a complex issue, use this tool to organize your thoughts and hypotheses

The tool simply logs your thought process for better transparency and does not execute any code or make changes.

Implementation Details

ThinkTool has a streamlined implementation focused on capturing reasoning:

```
const thinkToolSchema = z.object({
  thought: z.string().describe('Your thoughts.'),
})

export const ThinkTool = {
  name: 'Think',
  userFacingName: () => 'Think',

  isEnabled: async () =>
    Boolean(process.env.THINK_TOOL) && (await
checkGate('tengu_think_tool')),
  isReadOnly: () => true,
  needsPermissions: () => false,

  async *call(input, { messageId }) {
    logEvent('tengu_thinking', {
      messageId,
      thoughtLength: input.thought.length.toString(),
      method: 'tool',
      provider: USE_BEDROCK ? 'bedrock' : USE_VERTEX ? 'vertex' : '1p',
    })

    yield {
      type: 'result',
      resultForAssistant: 'Your thought has been logged.',
      data: { thought: input.thought },
    }
  },

  // This is never called -- it's special-cased in
AssistantToolUseMessage
  renderToolUseMessage(input) {
    return input.thought
  },
}
```

Key Components

ThinkTool provides several subtle but powerful features:

1. **Simple Schema Interface**
    - Single "thought" parameter for free-form reasoning
    - No complex validation or constraints

    - Designed for maximum flexibility in reasoning

2. **Special UI Integration**
    - Renders as a distinctive "thinking" message
    - Visual distinction from standard tool output
    - Special-cased handling in AssistantToolUseMessage component

3. **Analytics Tracking**
    - Logs usage events with Statsig
    - Captures thought length metrics
    - Tracks reasoning across different model providers

4. **Minimal Footprint**
    - No permissions required
    - No side effects (pure "thinking")
    - No state changes or modifications

Architecture

ThinkTool follows a minimal "no-op" architecture:

```
ThinkTool
  ↓
Input Capture → Processes thought string
  ↓
Analytics Tracking → Logs usage metrics
  ↓
Custom Rendering → Special-case UI display
  ↓
No-Op Result → Returns simple confirmation
```

The architecture prioritizes:

- **Transparency**: Making Claude's reasoning visible
- **Simplicity**: Minimal implementation complexity
- **Visual Distinction**: Clear UI separation from actions

- **Measurement**: Analytics for reasoning patterns

Feature Flag Control

ThinkTool employs a dual-control system for enabling:

```
isEnabled: async () =>
  Boolean(process.env.THINK_TOOL) && (await
checkGate('tengu_think_tool'))
```

This approach:

- Requires environment variable `THINK_TOOL` to be set
- Also requires Statsig feature gate `tengu_think_tool` to be enabled
- Provides multiple layers of activation control
- Supports staged rollout and experimental features

Usage Examples

ThinkTool is particularly valuable for complex reasoning tasks:

1. **Bug Analysis and Solution Planning**

```
Think(thought: "Looking at the error in the login form submission,
there are several possible root causes:
1. The form validation might be failing silently
2. The API endpoint could be rejecting malformed requests
3. CORS issues might be preventing the request

The most likely cause is #2 based on the error logs. I should check:
- Request payload format
- API validation requirements
- Response error codes")
```

2. **Architecture Design Reasoning**

```
Think(thought: "For implementing the new notification system, I need
to consider:
- Real-time vs. polling approach
- WebSocket integration complexities
- Database schema for storing notification state
- Front-end components for display

A hybrid approach seems best: WebSockets for active users, with
fallback to polling...")
```

3. **Refactoring Strategy**

```
Think(thought: "To refactor this authentication module, I have
several options:
1. Incremental approach: Replace components one by one
2. Parallel implementation: Build new system alongside old
3. Complete rewrite with feature freeze

Option 1 seems safest given the critical nature of auth...")
```

ThinkTool enhances explainability by:

- Making reasoning explicit rather than implicit

- Creating space for thorough exploration of options
- Separating analytical thinking from concrete actions
- Showing the problem-solving approach
- Making the reasoning process transparent

Command System Deep Dive

Note: This section is undergoing deep review. While these notes provide a useful overview, they are not yet as detailed or polished as other sections. Please file a GitHub issue if you notice any errors or have suggestions for additional content.

The Claude Code command system provides a suite of slash commands that users can invoke during interactive sessions. These commands allow users to perform various actions such as managing configuration, viewing help, clearing the screen, or resuming past conversations.

Commands are integrated into the CLI through the Commander.js library and follow a consistent structure for both CLI arguments and interactive slash commands. The command system allows Claude Code to extend beyond simple conversational capabilities into a fully-featured developer tool.

Command Categories

Claude Code's commands can be broadly categorized into several functional groups:

Environment Management

- approvedtools: Manage tool permissions
- commands: List available commands
- config: Configure Claude Code settings
- init: Create KODING.md for project setup
- terminalsetup: Configure terminal keybindings
- doctor: Check installation health

Authentication

- login: Authenticate with Anthropic
- logout: Sign out and remove credentials
- model: Configure AI model settings

Context Management

- clear: Reset conversation and free context space
- compact: Summarize and condense conversation
- ctx-viz: Visualize token usage
- resume: Continue previous conversations

User Experience

- help: View available commands and usage
- onboarding: Guided first-run experience
- listen: Enable speech recognition
- bug: Submit feedback and bug reports
- release-notes: View version changes

GitHub Integration

- pr-comments: View GitHub pull request comments
- review: Review GitHub pull requests

Analytics

- cost: Track API usage and expenses

Command Implementation Structure

Each command follows a consistent structure and typically falls into one of these implementation types:

1. **Local Commands** (`type: 'local'`): Perform direct actions without rendering a UI
2. **JSX Commands** (`type: 'local-jsx'`): Render interactive React components
3. **Prompt Commands** (`type: 'prompt'`): Formulate specific requests to Claude

Click on any command above to view its detailed implementation and functionality.

Implementation

The command is implemented in `commands/bug.tsx` and leverages React components to create an interactive feedback form:

```
import { Command } from "../commands";
import { Bug } from "../components/Bug";
import * as React from "react";
import { PRODUCT_NAME } from "../constants/product";

const bug = {
  type: "local-jsx",
  name: "bug",
  description: `Submit feedback about ${PRODUCT_NAME}`,
  isEnabled: true,
  isHidden: false,
  async call(onDone) {
    return <Bug onDone={onDone} />;
  },
  userFacingName() {
    return "bug";
  },
} satisfies Command;

export default bug;
```

Unlike pure command-line commands, this command uses the `type: 'local-jsx'` designation, which allows it to render a React component as its output. This enables a rich, interactive interface for gathering bug reports.

UI Component

The core functionality is housed in the `Bug` component in `components/Bug.tsx`. Key aspects of this component include:

1. **Multi-Step Form**: The UI guides users through a multi-step process:
    - User input (description of the bug)
    - Consent for information collection
    - Submission
    - Completion
2. **Information Collection**: The component collects:
    - User-provided bug description
    - Environment information (platform, terminal, version)
    - Git repository metadata (disabled in the current implementation)
    - Model settings (without API keys)

3. **GitHub Integration**: After submission, users can create a GitHub issue with pre-filled information including:
    - Bug description
    - Environment info
    - Model settings
4. **Privacy Considerations**: The component has been carefully designed to avoid collecting sensitive information:
    - No API keys are included
    - Personal identifiers have been removed
    - Direct submission to Anthropic's feedback endpoint has been commented out

Technical Features

Several technical aspects of the implementation are worth noting:

1. **Stateful Form Management**: Uses React's `useState` to manage the form state through multiple steps.
2. **Terminal Adaptation**: Adapts to terminal size using the `useTerminalSize` hook to ensure a good experience regardless of window size.
3. **Keyboard Navigation**: Implements customized keyboard handling with `useInput` from Ink to enable intuitive navigation.
4. **Error Handling**: Includes robust error handling for submission failures.
5. **Title Generation**: Originally included a capability to generate a concise title for the bug report using Claude's Haiku endpoint (currently commented out).
6. **Browser Integration**: Uses the `openBrowser` utility to open the user's default web browser to the GitHub issues page with pre-filled information.

Design Considerations

The bug command exemplifies several design principles of Claude Code's command system:

1. **Rich Terminal UI**: Unlike traditional CLI tools that might just accept arguments, the command provides a fully interactive experience with visual feedback.
2. **Progressive Disclosure**: Information about what will be collected is clearly shown before submission.
3. **Simple Escape Paths**: Users can easily cancel at any point using Escape or Ctrl+C/D.
4. **Clear Status Indicators**: The UI clearly shows the current step and available actions at all times.

This command demonstrates how Claude Code effectively leverages React and Ink to create sophisticated terminal user interfaces for commands that require complex interaction.

clear Command

The `clear` command provides a way to reset the current conversation, clear the message history, and free up context space.

Implementation

The command is implemented in `commands/clear.ts` as a type: 'local' command, which means it doesn't render a UI component but rather performs an operation directly:

```
import { Command } from "../commands";
import { getMessagesSetter } from "../messages";
import { getContext } from "../context";
import { getCodeStyle } from "../utils/style";
import { clearTerminal } from "../utils/terminal";
import { getOriginalCwd, setCwd } from "../utils/state";
import { Message } from "../query";

export async function clearConversation(context: {
  setForkConvoWithMessagesOnTheNextRender: (
    forkConvoWithMessages: Message[]
  ) => void;
}) {
  await clearTerminal();
  getMessagesSetter()([]);
  context.setForkConvoWithMessagesOnTheNextRender([]);
  getContext.cache.clear?.();
  getCodeStyle.cache.clear?.();
  await setCwd(getOriginalCwd());
}

const clear = {
  type: "local",
  name: "clear",
  description: "Clear conversation history and free up context",
  isEnabled: true,
  isHidden: false,
  async call(_, context) {
    clearConversation(context);
    return "";
  },
  userFacingName() {
    return "clear";
  },
} satisfies Command;

export default clear;
```

Functionality

The `clear` command performs several key operations:

1. **Terminal Cleaning**: Uses `clearTerminal()` to visually reset the terminal display.
2. **Message History Reset**: Sets the message array to empty using `getMessagesSetter()([])`.

3. **Context Clearing**: Clears the cached context information using `getContext.cache.clear()`.
4. **Style Cache Reset**: Clears the cached code style information with `getCodeStyle.cache.clear()`.
5. **Working Directory Reset**: Resets the current working directory to the original one using `setCwd(getOriginalCwd())`.
6. **Conversation Forking Reset**: Clears any pending conversation forks via the context parameter.

Technical Implementation Notes

The `clear` command makes use of several architectural patterns in Claude Code:

1. **Getter/Setter Pattern**: Uses message setter functions obtained through `getMessagesSetter()` rather than directly manipulating a message store, allowing for changes to be reactive across the UI.
2. **Cache Invalidation**: Explicitly clears caches for context and code style information to ensure fresh data when the user continues.
3. **State Management**: Demonstrates how state (like current working directory) is reset when clearing a conversation.
4. **Context Parameter**: Receives a context object from the command system that allows it to interact with the component rendering the REPL.
5. **Separate Function**: The core functionality is extracted into a separate `clearConversation` function, which allows it to be used by other parts of the system if needed.

User Experience Considerations

From a UX perspective, the `clear` command provides several benefits:

1. **Context Space Management**: Allows users to free up context space when they hit limitations with model context windows.

2. **Fresh Start**: Provides a clean slate for starting a new conversation without entirely restarting the CLI.
3. **Visual Reset**: The terminal clearing provides immediate visual feedback that the conversation has been reset.

The `clear` command is simple but vital for long-running CLI sessions, where context management becomes crucial for effective use of the LLM.

compact Command

The `compact` command offers a sophisticated solution to context management by summarizing the conversation history before clearing it, thereby retaining essential context while freeing up token space.

Implementation

The command is implemented in `commands/compact.ts` as a type: 'local' command:

```
import { Command } from "../commands";
import { getContext } from "../context";
import { getMessagesGetter, getMessagesSetter } from "../messages";
import { API_ERROR_MESSAGE_PREFIX, querySonnet } from
"../services/claude";
import {
  createUserMessage,
  normalizeMessagesForAPI,
} from "../utils/messages.js";
import { getCodeStyle } from "../utils/style";
import { clearTerminal } from "../utils/terminal";

const compact = {
  type: "local",
  name: "compact",
  description: "Clear conversation history but keep a summary in
context",
  isEnabled: true,
  isHidden: false,
  async call(
    _,
    {
      options: { tools, slowAndCapableModel },
      abortController,
```

```
      setForkConvoWithMessagesOnTheNextRender,
    }
  ) {
    // Get existing messages before clearing
    const messages = getMessagesGetter()();

    // Add summary request as a new message
    const summaryRequest = createUserMessage(
      "Provide a detailed but concise summary of our conversation above.
Focus on information that would be helpful for continuing the
conversation, including what we did, what we're doing, which files we're
working on, and what we're going to do next."
    );

    const summaryResponse = await querySonnet(
      normalizeMessagesForAPI([...messages, summaryRequest]),
      ["You are a helpful AI assistant tasked with summarizing
conversations."],
      0,
      tools,
      abortController.signal,
      {
        dangerouslySkipPermissions: false,
        model: slowAndCapableModel,
        prependCLISysprompt: true,
      }
    );

    // Extract summary from response, throw if we can't get it
    const content = summaryResponse.message.content;
    const summary =
      typeof content === "string"
        ? content
        : content.length > 0 && content[0]?.type === "text"
        ? content[0].text
        : null;

    if (!summary) {
      throw new Error(
        `Failed to generate conversation summary - response did not
contain valid text content - ${summaryResponse}`
      );
    } else if (summary.startsWith(API_ERROR_MESSAGE_PREFIX)) {
      throw new Error(summary);
    }

    // Substitute low token usage info so that the context-size UI
warning goes
    // away. The actual numbers don't matter too much: `countTokens`
checks the
    // most recent assistant message for usage numbers, so this estimate
will
    // be overridden quickly.
    summaryResponse.message.usage = {
```

```
        input_tokens: 0,
        output_tokens: summaryResponse.message.usage.output_tokens,
        cache_creation_input_tokens: 0,
        cache_read_input_tokens: 0,
      };

      // Clear screen and messages
      await clearTerminal();
      getMessagesSetter()([]);
      setForkConvoWithMessagesOnTheNextRender([
        createUserMessage(
          `Use the /compact command to clear the conversation history, and
start a new conversation with the summary in context.`
        ),
        summaryResponse,
      ]);
      getContext.cache.clear?.();
      getCodeStyle.cache.clear?.();

      return ""; // not used, just for typesafety. TODO: avoid this hack
    },
    userFacingName() {
      return "compact";
    },
  } satisfies Command;

  export default compact;
```

Functionality

The `compact` command implements a sophisticated workflow:

1. **Conversation Summary Generation**:
   - Retrieves the current message history
   - Creates a user message requesting a conversation summary
   - Uses Claude API to generate a summary through the `querySonnet` function
   - Validates the summary to ensure it was successfully generated
2. **Token Usage Management**:
   - Manipulates the usage data to prevent context-size warnings
   - Sets input tokens to 0 to indicate the conversation has been compacted
3. **Context Reset with Summary**:
   - Clears the terminal display

- Resets message history
- Creates a new conversation "fork" containing only:
  - A user message indicating a compact operation occurred
  - The generated summary response
- Clears context and code style caches

Technical Implementation Notes

The `compact` command demonstrates several advanced patterns:

1. **Meta-Conversation**: The command uses Claude to talk about the conversation itself, leveraging the model's summarization abilities.
2. **Model Selection**: Explicitly uses the `slowAndCapableModel` option to ensure high-quality summarization.
3. **Content Extraction Logic**: Implements robust parsing of the response content, handling different content formats (string vs. structured content).
4. **Error Handling**: Provides clear error messages for when summarization fails or when the API returns an error.
5. **Token Manipulation**: Intelligently manipulates token usage information to maintain a good user experience after compaction.
6. **Conversation Forking**: Uses the `setForkConvoWithMessagesOnTheNextRender` mechanism to create a new conversation branch with only the summary.

User Experience Benefits

The `compact` command addresses several key pain points in AI assistant interactions:

1. **Context Window Management**: Helps users stay within token limits while preserving the essence of the conversation.
2. **Conversation Continuity**: Unlike a complete clear, it maintains the thread of discussion through the summary.
3. **Work Session Persistence**: Preserves information about files being edited and tasks in progress.

4. **Smart Reset**: Performs a targeted reset that balances clearing space with maintaining context.

The command is particularly valuable for long development sessions where context limits become an issue but completely starting over would lose important progress information.

config Command

The `config` command provides an interactive terminal interface for viewing and editing Claude Code's configuration settings, including model settings, UI preferences, and API keys.

Implementation

The command is implemented in `commands/config.tsx` as a type: 'local-jsx' command that renders a React component:

```
import { Command } from "../commands";
import { Config } from "../components/Config";
import * as React from "react";

const config = {
  type: "local-jsx",
  name: "config",
  description: "Open config panel",
  isEnabled: true,
  isHidden: false,
  async call(onDone) {
    return <Config onClose={onDone} />;
  },
  userFacingName() {
    return "config";
  },
} satisfies Command;

export default config;
```

Like the `bug` command, this command uses JSX to render an interactive UI component. The actual functionality is implemented in the `Config` component

located in `components/Config.tsx`.

UI Component

The `Config` component implements a rich terminal-based settings interface with the following features:

1. **Settings Management**: Displays a list of configurable settings with their current values.
2. **Multiple Setting Types**: Supports various setting types:
   - `boolean`: Toggle settings (true/false)
   - `enum`: Options from a predefined list
   - `string`: Text input values
   - `number`: Numeric values
3. **Interactive Editing**: Allows users to:
   - Navigate settings with arrow keys
   - Toggle boolean and enum settings with Enter/Space
   - Edit string and number settings with a text input mode
   - Exit configuration with Escape
4. **Configuration Persistence**: Saves settings to a configuration file using `saveGlobalConfig`.

Configuration Options

The component exposes numerous configuration options, including:

1. **Model Configuration**:
   - AI Provider selection (anthropic, openai, custom)
   - API keys for small and large models
   - Model names for both small and large models
   - Base URLs for API endpoints
   - Max token settings
   - Reasoning effort levels
2. **User Interface**:

- Theme selection (light, dark, light-daltonized, dark-daltonized)
- Verbose output toggle

3. **System Settings**:
   - Notification preferences
   - HTTP proxy configuration

Technical Implementation Notes

The `Config` component demonstrates several advanced patterns:

1. **State Management**: Uses React's `useState` to track:
   - Current configuration state
   - Selected setting index
   - Editing mode state
   - Current input text
   - Input validation errors
2. **Reference Comparison**: Maintains a reference to the initial configuration using `useRef` to track changes.
3. **Keyboard Input Handling**: Implements sophisticated keyboard handling for navigation and editing:
   - Arrow keys for selection
   - Enter/Space for toggling/editing
   - Escape for cancellation
   - Input handling with proper validation
4. **Input Sanitization**: Cleans input text to prevent control characters and other problematic input.
5. **Visual Feedback**: Provides clear visual indication of:
   - Currently selected item
   - Editing state
   - Input errors
   - Available actions
6. **Change Tracking**: Tracks and logs configuration changes when exiting.

User Experience Design

The config component showcases several UI/UX design principles for terminal applications:

1. **Modal Interface**: Creates a focused settings panel that temporarily takes over the terminal.
2. **Progressive Disclosure**: Shows relevant controls and options based on the current state.
3. **Clear Instructions**: Displays context-sensitive help text at the bottom of the interface.
4. **Visual Highlighting**: Uses color and indicators to show the current selection and editing state.
5. **Immediate Feedback**: Changes take effect immediately, with visual confirmation.
6. **Multiple Input Methods**: Supports keyboard navigation, toggling, and text input in a unified interface.
7. **Safe Editing**: Provides validation and escape routes for configuration editing.

The `config` command demonstrates how Claude Code effectively combines the simplicity of terminal interfaces with the rich interaction capabilities typically associated with graphical applications, creating a powerful yet accessible configuration experience.

compact Command

The `compact` command offers a sophisticated solution to context management by summarizing the conversation history before clearing it, thereby retaining essential context while freeing up token space.

Implementation

The command is implemented in `commands/compact.ts` as a type: 'local' command:

```
import { Command } from "../commands";
import { getContext } from "../context";
import { getMessagesGetter, getMessagesSetter } from "../messages";
import { API_ERROR_MESSAGE_PREFIX, querySonnet } from
"../services/claude";
import {
  createUserMessage,
  normalizeMessagesForAPI,
} from "../utils/messages.js";
import { getCodeStyle } from "../utils/style";
import { clearTerminal } from "../utils/terminal";

const compact = {
  type: "local",
  name: "compact",
  description: "Clear conversation history but keep a summary in
context",
  isEnabled: true,
  isHidden: false,
  async call(
    _,
    {
      options: { tools, slowAndCapableModel },
      abortController,
      setForkConvoWithMessagesOnTheNextRender,
    }
  ) {
    // Get existing messages before clearing
    const messages = getMessagesGetter()();

    // Add summary request as a new message
    const summaryRequest = createUserMessage(
      "Provide a detailed but concise summary of our conversation above.
Focus on information that would be helpful for continuing the
conversation, including what we did, what we're doing, which files we're
working on, and what we're going to do next."
    );

    const summaryResponse = await querySonnet(
      normalizeMessagesForAPI([...messages, summaryRequest]),
      ["You are a helpful AI assistant tasked with summarizing
conversations."],
      0,
      tools,
      abortController.signal,
      {
        dangerouslySkipPermissions: false,
        model: slowAndCapableModel,
        prependCLISysprompt: true,
      }
```

```
    );

    // Extract summary from response, throw if we can't get it
    const content = summaryResponse.message.content;
    const summary =
      typeof content === "string"
        ? content
        : content.length > 0 && content[0]?.type === "text"
        ? content[0].text
        : null;

    if (!summary) {
      throw new Error(
        `Failed to generate conversation summary - response did not
contain valid text content - ${summaryResponse}`
      );
    } else if (summary.startsWith(API_ERROR_MESSAGE_PREFIX)) {
      throw new Error(summary);
    }

    // Substitute low token usage info so that the context-size UI
warning goes
    // away. The actual numbers don't matter too much: `countTokens`
checks the
    // most recent assistant message for usage numbers, so this estimate
will
    // be overridden quickly.
    summaryResponse.message.usage = {
      input_tokens: 0,
      output_tokens: summaryResponse.message.usage.output_tokens,
      cache_creation_input_tokens: 0,
      cache_read_input_tokens: 0,
    };

    // Clear screen and messages
    await clearTerminal();
    getMessagesSetter()([]);
    setForkConvoWithMessagesOnTheNextRender([
      createUserMessage(
        `Use the /compact command to clear the conversation history, and
start a new conversation with the summary in context.`
      ),
      summaryResponse,
    ]);
    getContext.cache.clear?.();
    getCodeStyle.cache.clear?.();

    return ""; // not used, just for typesafety. TODO: avoid this hack
  },
  userFacingName() {
    return "compact";
  },
} satisfies Command;
```

```
export default compact;
```

Functionality

The `compact` command implements a sophisticated workflow:

1. **Conversation Summary Generation**:
   - Retrieves the current message history
   - Creates a user message requesting a conversation summary
   - Uses Claude API to generate a summary through the `querySonnet` function
   - Validates the summary to ensure it was successfully generated
2. **Token Usage Management**:
   - Manipulates the usage data to prevent context-size warnings
   - Sets input tokens to 0 to indicate the conversation has been compacted
3. **Context Reset with Summary**:
   - Clears the terminal display
   - Resets message history
   - Creates a new conversation "fork" containing only:
     - A user message indicating a compact operation occurred
     - The generated summary response
   - Clears context and code style caches

Technical Implementation Notes

The `compact` command demonstrates several advanced patterns:

1. **Meta-Conversation**: The command uses Claude to talk about the conversation itself, leveraging the model's summarization abilities.
2. **Model Selection**: Explicitly uses the `slowAndCapableModel` option to ensure high-quality summarization.
3. **Content Extraction Logic**: Implements robust parsing of the response content, handling different content formats (string vs. structured content).

4. **Error Handling**: Provides clear error messages for when summarization fails or when the API returns an error.
5. **Token Manipulation**: Intelligently manipulates token usage information to maintain a good user experience after compaction.
6. **Conversation Forking**: Uses the `setForkConvoWithMessagesOnTheNextRender` mechanism to create a new conversation branch with only the summary.

User Experience Benefits

The `compact` command addresses several key pain points in AI assistant interactions:

1. **Context Window Management**: Helps users stay within token limits while preserving the essence of the conversation.
2. **Conversation Continuity**: Unlike a complete clear, it maintains the thread of discussion through the summary.
3. **Work Session Persistence**: Preserves information about files being edited and tasks in progress.
4. **Smart Reset**: Performs a targeted reset that balances clearing space with maintaining context.

The command is particularly valuable for long development sessions where context limits become an issue but completely starting over would lose important progress information.

ctx_viz Command

The `ctx_viz` command provides a detailed visualization of token usage across different components of the Claude conversation context, helping users understand how their context window is being utilized.

Implementation

The command is implemented in `commands/ctx_viz.ts` as a type: 'local' command that generates a formatted table of token usage:

```
import type { Command } from "../commands";
import type { Tool } from "../Tool";
import Table from "cli-table3";
import { getSystemPrompt } from "../constants/prompts";
import { getContext } from "../context";
import { zodToJsonSchema } from "zod-to-json-schema";
import { getMessagesGetter } from "../messages";

// Quick and dirty estimate of bytes per token for rough token counts
const BYTES_PER_TOKEN = 4;

interface Section {
  title: string;
  content: string;
}

interface ToolSummary {
  name: string;
  description: string;
}

function getContextSections(text: string): Section[] {
  const sections: Section[] = [];

  // Find first <context> tag
  const firstContextIndex = text.indexOf("<context");

  // Everything before first tag is Core Sysprompt
  if (firstContextIndex > 0) {
    const coreSysprompt = text.slice(0, firstContextIndex).trim();
    if (coreSysprompt) {
      sections.push({
        title: "Core Sysprompt",
        content: coreSysprompt,
      });
    }
  }

  let currentPos = firstContextIndex;
  let nonContextContent = "";

  const regex = /<context\s+name="([^"]*)">([\s\S]*?)<\/context>/g;
  let match: RegExpExecArray | null;

  while ((match = regex.exec(text)) !== null) {
    // Collect text between context tags
    if (match.index > currentPos) {
      nonContextContent += text.slice(currentPos, match.index);
```

```
    }

    const [, name = "Unnamed Section", content = ""] = match;
    sections.push({
      title: name === "codeStyle" ? "CodeStyle + KODING.md's" : name,
      content: content.trim(),
    });

    currentPos = match.index + match[0].length;
  }

  // Collect remaining text after last tag
  if (currentPos < text.length) {
    nonContextContent += text.slice(currentPos);
  }

  // Add non-contextualized content if present
  const trimmedNonContext = nonContextContent.trim();
  if (trimmedNonContext) {
    sections.push({
      title: "Non-contextualized Content",
      content: trimmedNonContext,
    });
  }

  return sections;
}

function formatTokenCount(bytes: number): string {
  const tokens = bytes / BYTES_PER_TOKEN;
  const k = tokens / 1000;
  return `${Math.round(k * 10) / 10}k`;
}

function formatByteCount(bytes: number): string {
  const kb = bytes / 1024;
  return `${Math.round(kb * 10) / 10}kb`;
}

function createSummaryTable(
  systemText: string,
  systemSections: Section[],
  tools: ToolSummary[],
  messages: unknown
): string {
  const table = new Table({
    head: ["Component", "Tokens", "Size", "% Used"],
    style: { head: ["bold"] },
    chars: {
      mid: "─",
      "left-mid": "├",
      "mid-mid": "┼",
      "right-mid": "┤",
    },
```

```
  });

  const messagesStr = JSON.stringify(messages);
  const toolsStr = JSON.stringify(tools);

  // Calculate total for percentages
  const total = systemText.length + toolsStr.length + messagesStr.length;
  const getPercentage = (n: number) => `${Math.round((n / total) *
100)}%`;

  // System prompt and its sections
  table.push([
    "System prompt",
    formatTokenCount(systemText.length),
    formatByteCount(systemText.length),
    getPercentage(systemText.length),
  ]);
  for (const section of systemSections) {
    table.push([
      `  ${section.title}`,
      formatTokenCount(section.content.length),
      formatByteCount(section.content.length),
      getPercentage(section.content.length),
    ]);
  }

  // Tools
  table.push([
    "Tool definitions",
    formatTokenCount(toolsStr.length),
    formatByteCount(toolsStr.length),
    getPercentage(toolsStr.length),
  ]);
  for (const tool of tools) {
    table.push([
      `  ${tool.name}`,
      formatTokenCount(tool.description.length),
      formatByteCount(tool.description.length),
      getPercentage(tool.description.length),
    ]);
  }

  // Messages and total
  table.push(
    [
      "Messages",
      formatTokenCount(messagesStr.length),
      formatByteCount(messagesStr.length),
      getPercentage(messagesStr.length),
    ],
    ["Total", formatTokenCount(total), formatByteCount(total), "100%"]
  );

  return table.toString();
```

```
}

const command: Command = {
  name: "ctx-viz",
  description:
    "[ANT-ONLY] Show token usage breakdown for the current conversation
context",
  isEnabled: true,
  isHidden: false,
  type: "local",

  userFacingName() {
    return this.name;
  },

  async call(_args: string, cmdContext: { options: { tools: Tool[] } }) {
    // Get tools and system prompt with injected context
    const [systemPromptRaw, sysContext] = await Promise.all([
      getSystemPrompt(),
      getContext(),
    ]);

    const rawTools = cmdContext.options.tools;

    // Full system prompt with context sections injected
    let systemPrompt = systemPromptRaw.join("\n");
    for (const [name, content] of Object.entries(sysContext)) {
      systemPrompt += `\n<context name="${name}">${content}</context>`;
    }

    // Get full tool definitions including prompts and schemas
    const tools = rawTools.map((t) => {
      // Get full prompt and schema
      const fullPrompt = t.prompt({ dangerouslySkipPermissions: false });
      const schema = JSON.stringify(
        "inputJSONSchema" in t && t.inputJSONSchema
          ? t.inputJSONSchema
          : zodToJsonSchema(t.inputSchema)
      );

      return {
        name: t.name,
        description: `${fullPrompt}\n\nSchema:\n${schema}`,
      };
    });

    // Get current messages from REPL
    const messages = getMessagesGetter()();

    const sections = getContextSections(systemPrompt);
    return createSummaryTable(systemPrompt, sections, tools, messages);
  },
};
```

```
export default command;
```

Functionality

The `ctx_viz` command provides a detailed breakdown of token usage across different components of the conversation context:

1. **System Prompt Analysis**:
    - Parses the system prompt to identify its separate sections
    - Extracts `<context>` tags and their contents for individual analysis
    - Identifies core system prompt sections vs. injected context
2. **Token Usage Calculation**:
    - Estimates token usage based on a bytes-per-token approximation
    - Presents data in kilobytes and estimated token counts
    - Calculates percentage usage for each component of the context
3. **Tool Definitions Analysis**:
    - Extracts complete tool definitions including prompts and JSON schemas
    - Calculates token usage per tool
    - Shows the total footprint of tool definitions in the context
4. **Conversation Message Analysis**:
    - Includes the current message history in the analysis
    - Shows what portion of the context window is used by the conversation
5. **Structured Presentation**:
    - Outputs a formatted ASCII table with columns for component, tokens, size, and percentage
    - Uses hierarchical indentation to show the structure of the context
    - Includes totals for complete context usage

Technical Implementation Notes

The `ctx_viz` command demonstrates several sophisticated implementation patterns:

1. **Regex-Based Context Parsing**: Uses regular expressions to parse the context sections from the system prompt, handling nested tags and multi-line content.
2. **Parallel Resource Loading**: Uses `Promise.all` to concurrently fetch system prompt and context data for efficiency.
3. **Tool Schema Introspection**: Extracts JSON schemas from tool definitions using either explicit schemas or by converting Zod schemas to JSON Schema format.
4. **Token Approximation**: Implements a simple but effective token estimation approach based on byte length, which provides a reasonable approximation without requiring a tokenizer.
5. **Table Formatting**: Uses the `cli-table3` library to create a formatted ASCII table with custom styling, making the output readable in a terminal environment.
6. **Context Section Management**: Special-cases certain context sections like "codeStyle" with custom labeling for clarity.

User Experience Benefits

The `ctx_viz` command addresses several important needs for developers using Claude Code:

1. **Context Window Transparency**: Gives users insight into how their limited context window is being utilized.
2. **Optimization Opportunities**: Helps identify large components that might be consuming excessive context, enabling targeted optimization.
3. **Debugging Aid**: Provides a debugging tool for situations where context limitations are affecting Claude's performance.
4. **System Prompt Visibility**: Makes the usually hidden system prompt and context visible to users for better understanding of Claude's behavior.

The command is particularly valuable for advanced users and developers who need to understand and optimize their context usage to get the most out of Claude's

capabilities within token limitations.

doctor Command

The `doctor` command provides a diagnostic tool for checking the health of the Claude Code installation, with a focus on npm permissions required for proper auto-updating functionality.

Implementation

The command is implemented in `commands/doctor.ts` as a type: 'local-jsx' command that renders a React component:

```
import React from "react";
import type { Command } from "../commands";
import { Doctor } from "../screens/Doctor";

const doctor: Command = {
  name: "doctor",
  description: "Checks the health of your Claude Code installation",
  isEnabled: true,
  isHidden: false,
  userFacingName() {
    return "doctor";
  },
  type: "local-jsx",
  call(onDone) {
    const element = React.createElement(Doctor, {
      onDone,
      doctorMode: true,
    });
    return Promise.resolve(element);
  },
};

export default doctor;
```

The command uses the `Doctor` screen component defined in `screens/Doctor.tsx`, passing the special `doctorMode` flag to indicate it's being used as a diagnostic tool rather than during initialization.

Functionality

The `doctor` command implements a comprehensive installation health check focused on npm permissions:

1. **Permissions Verification**:
    - Checks if the npm global installation directory has correct write permissions
    - Determines the current npm prefix path
    - Validates if auto-updates can function correctly
2. **Status Reporting**:
    - Provides clear success or failure messages about installation health
    - Shows a green checkmark and confirmation message for healthy installations
    - Presents an interactive dialog for resolving permission issues
3. **Problem Resolution**:
    - Offers three remediation options when permission problems are detected:
        1. **Manual Fix**: Provides a sudo/icacls command to fix permissions on the current npm prefix
        2. **New Prefix**: Guides the user through creating a new npm prefix in their home directory
        3. **Skip**: Allows deferring the fix until later
4. **Installation Repair**:
    - For the "new prefix" option, provides a guided, step-by-step process to:
        - Create a new directory for npm global packages
        - Configure npm to use the new location
        - Update shell PATH configurations
        - Reinstall Claude Code globally

Technical Implementation Notes

The command demonstrates several sophisticated implementation patterns:

1. **Component-Based Architecture**: Uses React components for the UI, allowing for a rich interactive experience.
2. **Platform-Aware Logic**: Implements different permission-fixing approaches for

Windows vs. Unix-like systems:

   - Windows: Uses `icacls` for permission management and `setx` for PATH updates
   - Unix: Uses `sudo chown/chmod` for permission fixes and shell config file updates

3. **Shell Detection**: For Unix platforms, identifies and updates multiple possible shell configuration files:

   - .bashrc and .bash_profile for Bash
   - .zshrc for Zsh
   - config.fish for Fish shell

4. **Status Management**: Uses React state to track:

   - Permission check status
   - Selected remediation option
   - Custom prefix path (if chosen)
   - Step-by-step installation progress

5. **Lock-Based Concurrency Control**: Implements a file-based locking mechanism to prevent multiple processes from attempting auto-updates simultaneously.

6. **Error Handling**: Provides detailed error reporting and recovery options:

   - Shows precisely which operation failed
   - Offers alternative approaches when errors occur
   - Logs error details for debugging

User Experience Benefits

The `doctor` command addresses several important pain points for CLI tool users:

1. **Installation Troubleshooting**: Provides clear diagnostics and fixes for common problems:

   - Permission issues that prevent updates
   - Global npm configuration problems
   - PATH configuration issues

2. **Security-Conscious Design**: Offers both privileged (sudo) and non-privileged (new prefix) solutions, allowing users to choose based on their security

preferences.

3. **Multi-Platform Support**: Works identically across Windows, macOS, and Linux with platform-appropriate solutions.
4. **Shell Environment Enhancement**: Automatically updates shell configuration files to ensure the installation works correctly across terminal sessions.
5. **Visual Progress Feedback**: Uses spinners and checkmarks to keep users informed about long-running operations.

The command provides a comprehensive diagnostics and repair tool that helps maintain a healthy Claude Code installation, particularly focusing on the auto-update capability which is crucial for keeping the tool current with new features and improvements.

help Command

The `help` command provides users with information about Claude Code's usage, available commands, and general guidance for effective interaction with the tool.

Implementation

The command is implemented in `commands/help.tsx` as a type: 'local-jsx' command that renders a React component:

```
import { Command } from "../commands";
import { Help } from "../components/Help";
import * as React from "react";

const help = {
  type: "local-jsx",
  name: "help",
  description: "Show help and available commands",
  isEnabled: true,
  isHidden: false,
  async call(onDone, { options: { commands } }) {
    return <Help commands={commands} onClose={onDone} />;
  },
  userFacingName() {
    return "help";
  },
} satisfies Command;

export default help;
```

The command delegates to the `Help` component in `components/Help.tsx`, passing the list of available commands and an `onClose` callback to handle when the help screen should be dismissed.

Functionality

The `Help` component implements a progressive disclosure pattern for displaying information:

1. **Basic Information**:
    - Shows the product name and version
    - Displays a brief description of Claude Code's capabilities and limitations
    - Presents a disclaimer about the beta nature of the product
2. **Usage Modes** (shown after a brief delay):
    - Explains the two primary ways to use Claude Code:
        - REPL (interactive session)
        - Non-interactive mode with the `-p` flag for one-off questions
    - Mentions that users can run `claude -h` for additional command-line options
3. **Common Tasks** (shown after a longer delay):

- Lists examples of typical use cases:
  - Asking questions about the codebase
  - Editing files
  - Fixing errors
  - Running commands
  - Running bash commands

4. **Available Commands** (shown after the longest delay):
   - Displays a comprehensive list of all enabled slash commands
   - Shows command names and descriptions
   - Filters out hidden commands

5. **Additional Resources**:
   - Provides links for getting more help
   - Shows different resources based on user type (internal vs. external)

Technical Implementation Notes

The `help` command demonstrates several effective patterns:

1. **Progressive Disclosure**: Uses a time-based mechanism to gradually reveal information:

   ```
   const [count, setCount] = React.useState(0);

   React.useEffect(() => {
     const timer = setTimeout(() => {
       if (count < 3) {
         setCount(count + 1);
       }
     }, 250);

     return () => clearTimeout(timer);
   }, [count]);
   ```

   This approach avoids overwhelming users with too much information at once, showing more details as they spend time on the help screen.

2. **Filtering System**: Uses `filter` to show only non-hidden commands:

```
const filteredCommands = commands.filter((cmd) => !cmd.isHidden);
```

   This keeps the help screen focused on commands relevant to users.

3. **Dynamic Resource Links**: Changes help resources based on user type:

```
const isInternal = process.env.USER_TYPE === "ant";
const moreHelp = isInternal
  ? "[ANT-ONLY] For more help: go/claude-cli or #claude-cli-
feedback"
  : `Learn more at: ${MACRO.README_URL}`;
```

   This customizes the experience for different user populations.

4. **Input Handling**: Uses Ink's `useInput` hook to detect when the user presses Enter:

```
useInput((_, key) => {
  if (key.return) onClose();
});
```

   This allows for clean dismissal of the help screen.

5. **Conditional Rendering**: Uses the `count` state to progressively show different sections:

```
{
  count >= 1 && (
    <Box flexDirection="column" marginTop={1}>
      <Text bold>Usage Modes:</Text>
      {/* Usage content */}
    </Box>
  );
}
```

This creates the staggered reveal effect for information groups.

User Experience Benefits

The `help` command addresses several important needs for CLI tool users:

1. **Onboarding Assistance**: Provides new users with immediate guidance on how to use the tool effectively.
2. **Command Discovery**: Makes it easy to see what slash commands are available without having to memorize them.
3. **Progressive Learning**: The staggered reveal of information allows users to absorb basics first before seeing more advanced options.
4. **Usage Examples**: Shows concrete examples of common use cases, helping users understand practical applications.
5. **Quick Reference**: Serves as a compact reference for command options during regular use.

The help command exemplifies Claude Code's approach to user experience: it's focused on being informative while remaining concise and unobtrusive, with multiple levels of detail available as users need them.

init Command

The `init` command helps users initialize a new project for use with Claude Code by

creating a KODING.md file that contains key information about the codebase and project.

Implementation

The command is implemented in `commands/init.ts` as a type: 'prompt' command that passes a specific request to Claude:

```
import type { Command } from "../commands";
import { markProjectOnboardingComplete } from "../ProjectOnboarding";

const command = {
  type: "prompt",
  name: "init",
  description: "Initialize a new KODING.md file with codebase
documentation",
  isEnabled: true,
  isHidden: false,
  progressMessage: "analyzing your codebase",
  userFacingName() {
    return "init";
  },
  async getPromptForCommand(_args: string) {
    // Mark onboarding as complete when init command is run
    markProjectOnboardingComplete();
    return [
      {
        role: "user",
        content: [
          {
            type: "text",
            text: `Please analyze this codebase and create a KODING.md
file containing:
1. Build/lint/test commands - especially for running a single test
2. Code style guidelines including imports, formatting, types, naming
conventions, error handling, etc.

The file you create will be given to agentic coding agents (such as
yourself) that operate in this repository. Make it about 20 lines long.
If there's already a KODING.md, improve it.
If there are Cursor rules (in .cursor/rules/ or .cursorrules) or Copilot
rules (in .github/copilot-instructions.md), make sure to include them.`,
          },
        ],
      },
    ];
  },
} satisfies Command;

export default command;
```

Unlike many other commands that directly perform actions, the `init` command is implemented as a 'prompt' type command that simply formulates a specific request to Claude to analyze the codebase and generate a KODING.md file.

Functionality

The `init` command serves several important functions:

1. **Project Configuration Generation**:
    - Instructs Claude to analyze the codebase structure, conventions, and patterns
    - Generates a KODING.md file with essential project information
    - Focuses on capturing build/lint/test commands and code style guidelines
2. **Existing File Enhancement**:
    - Checks if a KODING.md file already exists
    - Improves the existing file with additional information if present
    - Preserves existing content while enhancing where needed
3. **Integration with Other Tools**:
    - Looks for Cursor rules (in .cursor/rules/ or .cursorrules)
    - Looks for Copilot instructions (in .github/copilot-instructions.md)
    - Incorporates these rules into the KODING.md file for a unified guidance document
4. **Onboarding Status Tracking**:
    - Calls `markProjectOnboardingComplete()` to update the project configuration
    - This flags the project as having completed onboarding steps
    - Prevents repeated onboarding prompts in future sessions

Technical Implementation Notes

The `init` command demonstrates several interesting technical aspects:

1. **Command Type**: Uses the 'prompt' type rather than 'local' or 'local-jsx', which means it sets up a specific request to Claude rather than executing custom logic or rendering a UI component.
2. **Progress Message**: Includes a `progressMessage` property ("analyzing your codebase") that's displayed to users while Claude processes the request, providing feedback during what could be a longer operation.
3. **Project Onboarding Integration**: The command is part of the project onboarding workflow, tracking completion state in the project configuration:

```
export function markProjectOnboardingComplete(): void {
  const projectConfig = getCurrentProjectConfig();
  if (!projectConfig.hasCompletedProjectOnboarding) {
    saveCurrentProjectConfig({
      ...projectConfig,
      hasCompletedProjectOnboarding: true,
    });
  }
}
```

4. **External Tool Recognition**: Explicitly looks for configuration files from other AI coding tools (Cursor, Copilot) to create a unified guidance document, showing awareness of the broader AI coding tool ecosystem.

5. **Conciseness Guidance**: Explicitly requests a document of around 20 lines, balancing comprehensive information with brevity for practical usage.

User Experience Benefits

The `init` command provides several important benefits for Claude Code users:

1. **Simplified Project Setup**: Makes it easy to prepare a codebase for effective use with Claude Code with a single command.

2. **Consistent Memory**: Creates a standardized document that Claude Code will access in future sessions, providing consistent context.

3. **Command Discovery**: By capturing build/test/lint commands, it helps Claude remember the correct commands for the project, reducing the need for users to repeatedly provide them.

4. **Code Style Guidance**: Helps Claude generate code that matches the project's conventions, improving integration of AI-generated code.

5. **Onboarding Pathway**: Serves as a key step in the onboarding flow for new projects, guiding users toward the most effective usage patterns.

The `init` command exemplifies Claude Code's overall approach to becoming more

effective over time by creating persistent context that improves future interactions. By capturing project-specific information in a standardized format, it enables Claude to provide more tailored assistance for each unique codebase.

listen Command

The `listen` command provides macOS users with the ability to dictate to Claude Code using speech recognition, enhancing accessibility and offering an alternative input method.

Implementation

The command is implemented in `commands/listen.ts` as a type: 'local' command that invokes macOS's built-in dictation feature:

```
import { Command } from "../commands";
import { logError } from "../utils/log";
import { execFileNoThrow } from "../utils/execFileNoThrow";

const isEnabled =
  process.platform === "darwin" &&
  ["iTerm.app", "Apple_Terminal"].includes(process.env.TERM_PROGRAM ||
"");

const listen: Command = {
  type: "local",
  name: "listen",
  description: "Activates speech recognition and transcribes speech to
text",
  isEnabled: isEnabled,
  isHidden: isEnabled,
  userFacingName() {
    return "listen";
  },
  async call(_, { abortController }) {
    // Start dictation using AppleScript
    const script = `tell application "System Events" to tell ¬
(the first process whose frontmost is true) to tell ¬
menu bar 1 to tell ¬
menu bar item "Edit" to tell ¬
menu "Edit" to tell ¬
menu item "Start Dictation" to ¬
if exists then click it`;

    const { stderr, code } = await execFileNoThrow(
      "osascript",
      ["-e", script],
      abortController.signal
    );

    if (code !== 0) {
      logError(`Failed to start dictation: ${stderr}`);
      return "Failed to start dictation";
    }
    return "Dictation started. Press esc to stop.";
  },
};

export default listen;
```

The command uses AppleScript to trigger macOS's built-in dictation feature, automating the process of selecting "Start Dictation" from the Edit menu.

Functionality

The `listen` command provides a simple but effective accessibility enhancement:

1. **Platform Integration**:
    - Invokes macOS's built-in dictation feature
    - Uses AppleScript to navigate the application's menu structure
    - Triggers the "Start Dictation" menu item in the Edit menu
2. **Status Reporting**:
    - Returns clear success or error messages
    - Provides user instructions on how to stop dictation
    - Logs detailed error information when dictation fails to start
3. **Platform-Specific Availability**:
    - Only enabled on macOS platforms
    - Further restricted to specific terminal applications (iTerm.app and Apple_Terminal)
    - Hidden from command listings on unsupported platforms

Technical Implementation Notes

The `listen` command demonstrates several interesting technical approaches:

1. **Platform Detection**: Uses environment variables and platform checks to determine when the command should be available:

```
const isEnabled =
  process.platform === "darwin" &&
  ["iTerm.app", "Apple_Terminal"].includes(process.env.TERM_PROGRAM
|| "");
```

2. **AppleScript Integration**: Uses a complex AppleScript command to navigate through application menus:

```
const script = `tell application "System Events" to tell ¬
(the first process whose frontmost is true) to tell ¬
menu bar 1 to tell ¬
menu bar item "Edit" to tell ¬
menu "Edit" to tell ¬
menu item "Start Dictation" to ¬
if exists then click it`;
```

This script navigates the UI hierarchy to find and click the dictation menu item.

3. **Command Visibility Control**: Uses the same conditions for both `isEnabled` and `isHidden`, ensuring the command is simultaneously enabled and hidden on supported platforms:

```
isEnabled: isEnabled,
isHidden: isEnabled,
```

This unusual pattern makes the command available but not visible in help listings, suggesting it's an internal or experimental feature.

4. **Error Handling**: Implements robust error handling with detailed logging:

```
if (code !== 0) {
  logError(`Failed to start dictation: ${stderr}`);
  return "Failed to start dictation";
}
```

User Experience Considerations

While simple in implementation, the `listen` command addresses several important user needs:

1. **Accessibility Enhancement**: Provides an alternative input method for users who prefer or require speech recognition.

2. **Workflow Efficiency**: Eliminates the need to manually navigate menus to start dictation, streamlining the process.
3. **Integrated Experience**: Keeps users within the Claude Code interface rather than requiring them to use separate dictation tools.
4. **Platform Integration**: Leverages native OS capabilities rather than implementing custom speech recognition, ensuring high-quality transcription.

The `listen` command demonstrates Claude Code's commitment to accessibility and platform integration, even if the implementation is relatively simple. By leveraging existing OS capabilities rather than reinventing them, it provides a valuable feature with minimal code complexity.

login Command

The `login` command provides users with a secure OAuth 2.0 authentication flow to connect Claude Code with their Anthropic Console account, enabling API access and proper billing.

Implementation

The command is implemented in `commands/login.tsx` as a type: 'local-jsx' command that renders a React component for the authentication flow:

```tsx
import * as React from "react";
import type { Command } from "../commands";
import { ConsoleOAuthFlow } from "../components/ConsoleOAuthFlow";
import { clearTerminal } from "../utils/terminal";
import { isLoggedInToAnthropic } from "../utils/auth";
import { useExitOnCtrlCD } from "../hooks/useExitOnCtrlCD";
import { Box, Text } from "ink";
import { clearConversation } from "./clear";

export default () =>
  ({
    type: "local-jsx",
    name: "login",
    description: isLoggedInToAnthropic()
      ? "Switch Anthropic accounts"
      : "Sign in with your Anthropic account",
    isEnabled: true,
    isHidden: false,
    async call(onDone, context) {
      await clearTerminal();
      return (
        <Login
          onDone={async () => {
            clearConversation(context);
            onDone();
          }}
        />
      );
    },
    userFacingName() {
      return "login";
    },
  } satisfies Command);

function Login(props: { onDone: () => void }) {
  const exitState = useExitOnCtrlCD(props.onDone);
  return (
    <Box flexDirection="column">
      <ConsoleOAuthFlow onDone={props.onDone} />
      <Box marginLeft={3}>
        <Text dimColor>
          {exitState.pending ? (
            <>Press {exitState.keyName} again to exit</>
          ) : (
            ""
          )}
        </Text>
      </Box>
    </Box>
  );
}
```

The command uses a factory function to dynamically create a command object that adapts its description based on the current login state. The main logic is delegated to the `ConsoleOAuthFlow` component, which handles the OAuth flow with the Anthropic API.

Functionality

The `login` command implements a comprehensive authentication flow:

1. **OAuth 2.0 Integration**:
    - Uses OAuth 2.0 with PKCE (Proof Key for Code Exchange) for security
    - Opens a browser for the user to log in to Anthropic Console
    - Handles both automatic browser redirect and manual code entry flows
    - Securely exchanges authorization codes for access tokens
2. **Account Management**:
    - Creates and stores API keys for authenticated users
    - Normalizes and securely saves keys in the global configuration
    - Tracks account information including account UUID and organization details
    - Provides context-aware descriptions ("Sign in" vs "Switch accounts")
3. **Security Features**:
    - Implements state verification to prevent CSRF attacks
    - Uses code verifiers and challenges for secure code exchange
    - Validates all tokens and responses from the authentication server
    - Provides clear error messages for authentication failures
4. **UI Experience**:
    - Clears the terminal for a clean login experience
    - Provides a progressive disclosure flow with appropriate status messages
    - Offers fallback mechanisms for cases where automatic browser opening fails
    - Shows loading indicators during asynchronous operations

Technical Implementation Notes

The login command demonstrates several sophisticated technical approaches:

1. **Local HTTP Server**: Creates a temporary HTTP server for OAuth callback handling:

```
this.server = http.createServer(
  (req: IncomingMessage, res: ServerResponse) => {
    const parsedUrl = url.parse(req.url || "", true);
    if (parsedUrl.pathname === "/callback") {
      // Handle OAuth callback
    }
  }
);
```

2. **PKCE Implementation**: Implements the Proof Key for Code Exchange extension to OAuth:

```
function generateCodeVerifier(): string {
  return base64URLEncode(crypto.randomBytes(32));
}

async function generateCodeChallenge(verifier: string):
Promise<string> {
  const encoder = new TextEncoder();
  const data = encoder.encode(verifier);
  const digest = await crypto.subtle.digest("SHA-256", data);
  return base64URLEncode(Buffer.from(digest));
}
```

3. **Parallel Authentication Paths**: Supports both automatic and manual authentication flows:

```
const { autoUrl, manualUrl } = this.generateAuthUrls(codeChallenge,
state);
await authURLHandler(manualUrl); // Show manual URL in UI
await openBrowser(autoUrl); // Try automatic browser opening
```

4. **Promise-Based Flow Control**: Uses promises to coordinate the asynchronous authentication flow:

```
const { authorizationCode, useManualRedirect } = await new Promise<{
  authorizationCode: string;
  useManualRedirect: boolean;
}>((resolve, reject) => {
  this.pendingCodePromise = { resolve, reject };
  this.startLocalServer(state, onReady);
});
```

5. **State Management with React**: Uses React state and hooks for UI management:

```
const [oauthStatus, setOAuthStatus] = useState<OAuthStatus>({
  state: "idle",
});
```

6. **Error Recovery**: Implements sophisticated error handling with retry mechanisms:

```
if (oauthStatus.state === "error" && oauthStatus.toRetry) {
  setPastedCode("");
  setOAuthStatus({
    state: "about_to_retry",
    nextState: oauthStatus.toRetry,
  });
}
```

User Experience Benefits

The `login` command addresses several important user needs:

1. **Seamless Authentication**: Provides a smooth authentication experience

without requiring manual API key creation or copying.

2. **Cross-Platform Compatibility**: Works across different operating systems and browsers.
3. **Fallback Mechanisms**: Offers manual code entry when automatic browser redirection fails.
4. **Clear Progress Indicators**: Shows detailed status messages throughout the authentication process.
5. **Error Resilience**: Provides helpful error messages and retry options when authentication issues occur.
6. **Account Switching**: Allows users to easily switch between different Anthropic accounts.

The login command exemplifies Claude Code's approach to security and user experience, implementing a complex authentication flow with attention to both security best practices and ease of use.

logout Command

The `logout` command provides users with the ability to sign out from their Anthropic account, removing stored authentication credentials and API keys from the local configuration.

Implementation

The command is implemented in `commands/logout.tsx` as a type: 'local-jsx' command that handles the logout process and renders a confirmation message:

```
import * as React from "react";
import type { Command } from "../commands";
import { getGlobalConfig, saveGlobalConfig } from "../utils/config";
import { clearTerminal } from "../utils/terminal";
import { Text } from "ink";

export default {
  type: "local-jsx",
  name: "logout",
  description: "Sign out from your Anthropic account",
  isEnabled: true,
  isHidden: false,
  async call() {
    await clearTerminal();

    const config = getGlobalConfig();

    config.oauthAccount = undefined;
    config.primaryApiKey = undefined;
    config.hasCompletedOnboarding = false;

    if (config.customApiKeyResponses?.approved) {
      config.customApiKeyResponses.approved = [];
    }

    saveGlobalConfig(config);

    const message = (
      <Text>Successfully logged out from your Anthropic account.</Text>
    );

    setTimeout(() => {
      process.exit(0);
    }, 200);

    return message;
  },
  userFacingName() {
    return "logout";
  },
} satisfies Command;
```

Unlike the more complex `login` command, the `logout` command is relatively straightforward, focusing on removing authentication data from the configuration and providing a clean exit.

Functionality

The `logout` command performs several critical operations:

1. **Credential Removal**:
    - Clears the OAuth account information from the global configuration
    - Removes the primary API key used for authentication
    - Erases the list of approved API keys from storage
    - Resets the onboarding completion status
2. **User Experience**:
    - Clears the terminal before displaying the logout message
    - Provides a clear confirmation message about successful logout
    - Exits the application completely after a short delay
    - Ensures a clean break with the authenticated session
3. **Security Focus**:
    - Removes all sensitive authentication data from the local configuration
    - Ensures the next application start will require re-authentication
    - Prevents accidental API usage with old credentials
    - Provides a clean slate for a new login if desired

Technical Implementation Notes

Despite its relative simplicity, the `logout` command demonstrates several interesting implementation details:

1. **Configuration Management**: Uses the global configuration system to handle persistent state:

```
const config = getGlobalConfig();

config.oauthAccount = undefined;
config.primaryApiKey = undefined;
config.hasCompletedOnboarding = false;

if (config.customApiKeyResponses?.approved) {
  config.customApiKeyResponses.approved = [];
}

saveGlobalConfig(config);
```

2. **Graceful Exit Strategy**: Uses a timeout to allow the message to be displayed before exiting:

```
setTimeout(() => {
  process.exit(0);
}, 200);
```

   This ensures the user sees confirmation before the application closes.

3. **Type Safety**: Uses the `satisfies Command` pattern to ensure type correctness:

```
export default {
  // Command implementation
} satisfies Command;
```

4. **Terminal Management**: Clears the terminal before displaying the logout confirmation:

```
await clearTerminal();
```

   This creates a clean visual experience for the logout process.

5. **Optional Field Handling**: Carefully checks for the existence of optional configuration fields:

```
if (config.customApiKeyResponses?.approved) {
  config.customApiKeyResponses.approved = [];
}
```

User Experience Benefits

The `logout` command addresses several important user needs:

1. **Account Security**: Provides a clear way to remove credentials when sharing devices or ending a session.
2. **User Confidence**: Confirms successful logout with a clear message, reassuring users their credentials have been removed.
3. **Clean Exit**: Exits the application completely, avoiding any state confusion in the current process.
4. **Simplicity**: Keeps the logout process straightforward and quick, with minimal user interaction required.
5. **Fresh Start**: Resets the onboarding status, ensuring a proper re-onboarding flow on next login.

The `logout` command provides a necessary counterpart to the `login` command, completing the authentication lifecycle with a secure, clean way to end a session. While much simpler than its login counterpart, it maintains the same attention to security and user experience that characterizes Claude Code's approach to authentication management.

model Command

> [!WARNING] This command implementation is specific to the anon-kode fork of

> Claude Code and is not part of the original Claude Code codebase. The analysis below pertains to this specific implementation rather than standard Claude Code functionality.

The `model` command provides users with a comprehensive interface to configure and customize the AI models used by Claude Code, enabling fine-grained control over model selection, parameters, and provider settings.

Implementation

The command is implemented in `commands/model.tsx` as a type: 'local-jsx' command that renders a React component for model configuration:

```
import React from "react";
import { render } from "ink";
import { ModelSelector } from "../components/ModelSelector";
import { enableConfigs } from "../utils/config";

export const help = "Change your AI provider and model settings";
export const description = "Change your AI provider and model settings";
export const isEnabled = true;
export const isHidden = false;
export const name = "model";
export const type = "local-jsx";

export function userFacingName(): string {
  return name;
}

export async function call(
  onDone: (result?: string) => void,
  { abortController }: { abortController?: AbortController }
): Promise<React.ReactNode> {
  enableConfigs();
  abortController?.abort?.();
  return (
    <ModelSelector
      onDone={() => {
        onDone();
      }}
    />
  );
}
```

The command uses a different export style than other commands, directly exporting properties and functions rather than a single object. The main functionality is handled by the `ModelSelector` component, which provides an interactive UI for configuring model settings.

Functionality

The `model` command provides a sophisticated model selection and configuration workflow:

1. **Multi-Model Management**:
    - Allows configuring both "large" and "small" models separately or together
    - Provides different models for different task complexities for optimal cost/performance
    - Shows current configuration information for reference
2. **Provider Selection**:
    - Supports multiple AI providers (Anthropic, OpenAI, Gemini, etc.)
    - Dynamically fetches available models from the selected provider's API
    - Handles provider-specific API requirements and authentication
3. **Model Parameters**:
    - Configures maximum token settings for response length control
    - Offers reasoning effort controls for supported models (low/medium/high)
    - Preserves provider-specific configuration options
4. **Search and Filtering**:
    - Provides search functionality to filter large model lists
    - Displays model capabilities including token limits and feature support
    - Organizes models with sensible sorting and grouping
5. **API Key Management**:
    - Securely handles API keys for different providers
    - Masks sensitive information during input and display
    - Stores keys securely in the local configuration

Technical Implementation Notes

The `model` command demonstrates several sophisticated technical approaches:

1. **Multi-Step Navigation**: Implements a screen stack pattern for intuitive flow navigation:

```
const [screenStack, setScreenStack] = useState<
  Array<
    | "modelType"
    | "provider"
    | "apiKey"
    | "model"
    | "modelParams"
    | "confirmation"
  >
>(["modelType"]);

// Current screen is always the last item in the stack
const currentScreen = screenStack[screenStack.length - 1];

// Function to navigate to a new screen
const navigateTo = (
  screen:
    | "modelType"
    | "provider"
    | "apiKey"
    | "model"
    | "modelParams"
    | "confirmation"
) => {
  setScreenStack((prev) => [...prev, screen]);
};

// Function to go back to the previous screen
const goBack = () => {
  if (screenStack.length > 1) {
    // Remove the current screen from the stack
    setScreenStack((prev) => prev.slice(0, -1));
  } else {
    // If we're at the first screen, call onDone to exit
    onDone();
  }
};
```

2. **Dynamic Model Loading**: Fetches available models directly from provider APIs:

```
async function fetchModels() {
  setIsLoadingModels(true);
  setModelLoadError(null);

  try {
    // Provider-specific logic...
    const openai = new OpenAI({
      apiKey: apiKey,
      baseURL: baseURL,
      dangerouslyAllowBrowser: true,
    });

    // Fetch the models
    const response = await openai.models.list();

    // Transform the response into our ModelInfo format
    const fetchedModels = [];
    // Process models...

    return fetchedModels;
  } catch (error) {
    setModelLoadError(`Failed to load models: ${error.message}`);
    throw error;
  } finally {
    setIsLoadingModels(false);
  }
}
```

3. **Form Focus Management**: Implements sophisticated form navigation with keyboard support:

```
// Handle Tab key for form navigation in model params screen
useInput((input, key) => {
  if (currentScreen === "modelParams" && key.tab) {
    const formFields = getFormFieldsForModelParams();
    // Move to next field
    setActiveFieldIndex((current) => (current + 1) %
formFields.length);
    return;
  }

  // Handle Enter key for form submission in model params screen
  if (currentScreen === "modelParams" && key.return) {
    const formFields = getFormFieldsForModelParams();

    if (activeFieldIndex === formFields.length - 1) {
      // If on the Continue button, submit the form
      handleModelParamsSubmit();
    }
    return;
  }
});
```

4. **Provider-Specific Handling**: Implements custom logic for different AI providers:

```
// For Gemini, use the separate fetchGeminiModels function
if (selectedProvider === "gemini") {
  const geminiModels = await fetchGeminiModels();
  setAvailableModels(geminiModels);
  navigateTo("model");
  return geminiModels;
}
```

5. **Configuration Persistence**: Carefully updates global configuration with new model settings:

```
function saveConfiguration(provider: ProviderType, model: string) {
  const baseURL = providers[provider]?.baseURL || "";

  // Create a new config object based on the existing one
  const newConfig = { ...config };

  // Update the primary provider regardless of which model we're
changing
  newConfig.primaryProvider = provider;

  // Update the appropriate model based on the selection
  if (modelTypeToChange === "both" || modelTypeToChange === "large")
{
    newConfig.largeModelName = model;
    newConfig.largeModelBaseURL = baseURL;
    newConfig.largeModelApiKey = apiKey || config.largeModelApiKey;
    newConfig.largeModelMaxTokens = parseInt(maxTokens);

    // Save reasoning effort for large model if supported
    if (supportsReasoningEffort) {
      newConfig.largeModelReasoningEffort = reasoningEffort;
    } else {
      newConfig.largeModelReasoningEffort = undefined;
    }
  }

  // Similar handling for small model...

  // Save the updated configuration
  saveGlobalConfig(newConfig);
}
```

User Experience Benefits

The `model` command provides several important benefits for Claude Code users:

1. **Customization Control**: Gives users fine-grained control over the AI models powering their interaction.

2. **Cost Optimization**: Allows setting different models for different complexity tasks, optimizing for cost and speed.
3. **Provider Flexibility**: Enables users to choose from multiple AI providers based on preference, cost, or feature needs.
4. **Parameter Tuning**: Offers advanced users the ability to tune model parameters for optimal performance.
5. **Progressive Disclosure**: Uses a step-by-step flow that makes configuration accessible to both novice and advanced users.
6. **Intuitive Navigation**: Implements keyboard navigation with clear indicators for a smooth configuration experience.

The `model` command exemplifies Claude Code's approach to giving users control and flexibility while maintaining an accessible interface, keeping advanced configuration options available but not overwhelming.

cost Command

The `cost` command provides users with visibility into the cost and duration of their current Claude Code session, helping them monitor their API usage and expenses.

Implementation

The command is implemented in `commands/cost.ts` as a simple type: 'local' command that calls a formatting function:

```
import type { Command } from "../commands";
import { formatTotalCost } from "../cost-tracker";

const cost = {
  type: "local",
  name: "cost",
  description: "Show the total cost and duration of the current session",
  isEnabled: true,
  isHidden: false,
  async call() {
    return formatTotalCost();
  },
  userFacingName() {
    return "cost";
  },
} satisfies Command;

export default cost;
```

This command relies on the cost tracking system implemented in `cost-tracker.ts`, which maintains a running total of API costs and session duration.

Cost Tracking System

The cost tracking system is implemented in `cost-tracker.ts` and consists of several key components:

1. **State Management**: Maintains a simple singleton state object tracking:
   - `totalCost`: Running total of API costs in USD
   - `totalAPIDuration`: Cumulative time spent waiting for API responses
   - `startTime`: Timestamp when the session began
2. **Cost Accumulation**: Provides a function to add costs as they occur:

```
export function addToTotalCost(cost: number, duration: number): void
{
  STATE.totalCost += cost;
  STATE.totalAPIDuration += duration;
}
```

3. **Reporting**: Formats cost information in a human-readable format:

```
export function formatTotalCost(): string {
  return chalk.grey(
    `Total cost: ${formatCost(STATE.totalCost)}
    Total duration (API): ${formatDuration(STATE.totalAPIDuration)}
    Total duration (wall): ${formatDuration(getTotalDuration())}`
  );
}
```

4. **Persistence**: Uses a React hook to save session cost data when the process exits:

```
export function useCostSummary(): void {
  useEffect(() => {
    const f = () => {
      process.stdout.write("\n" + formatTotalCost() + "\n");

      // Save last cost and duration to project config
      const projectConfig = getCurrentProjectConfig();
      saveCurrentProjectConfig({
        ...projectConfig,
        lastCost: STATE.totalCost,
        lastAPIDuration: STATE.totalAPIDuration,
        lastDuration: getTotalDuration(),
        lastSessionId: SESSION_ID,
      });
    };
    process.on("exit", f);
    return () => {
      process.off("exit", f);
    };
  }, []);
}
```

UI Components

The cost tracking system is complemented by two UI components:

1. **Cost Component**: A simple display component used in the debug panel to show the most recent API call cost:

```
export function Cost({
  costUSD,
  durationMs,
  debug,
}: Props): React.ReactNode {
  if (!debug) {
    return null;
  }

  const durationInSeconds = (durationMs / 1000).toFixed(1);
  return (
    <Box flexDirection="column" minWidth={23} width={23}>
      <Text dimColor>
        Cost: ${costUSD.toFixed(4)} ({durationInSeconds}s)
      </Text>
    </Box>
  );
}
```

2. **CostThresholdDialog**: A warning dialog shown when users exceed a certain cost threshold:

```
export function CostThresholdDialog({ onDone }: Props):
React.ReactNode {
  // Handle Ctrl+C, Ctrl+D and Esc
  useInput((input, key) => {
    if ((key.ctrl && (input === "c" || input === "d")) ||
key.escape) {
      onDone();
    }
  });

  return (
    <Box
      flexDirection="column"
      borderStyle="round"
      padding={1}
      borderColor={getTheme().secondaryBorder}
    >
      <Box marginBottom={1} flexDirection="column">
        <Text bold>You've spent $5 on the Anthropic API this
session.</Text>
        <Text>Learn more about how to monitor your spending:</Text>
        <Link url="https://docs.anthropic.com/s/claude-code-cost" />
      </Box>
      <Box>
        <Select
          options={[
            {
              value: "ok",
              label: "Got it, thanks!",
            },
          ]}
          onChange={onDone}
        />
      </Box>
    </Box>
  );
}
```

Technical Implementation Notes

The cost tracking system demonstrates several design considerations:

1. **Singleton State**: Uses a single state object with a clear comment warning against adding more state.
2. **Persistence Across Sessions**: Saves cost data to the project configuration, allowing for tracking across sessions.
3. **Formatting Flexibility**: Uses different decimal precision based on the cost amount (4 decimal places for small amounts, 2 for larger ones).
4. **Multiple Time Metrics**: Tracks both wall clock time and API request time separately.
5. **Environment-Aware Testing**: Includes a reset function that's only available in test environments.
6. **Exit Hooks**: Uses process exit hooks to ensure cost data is saved and displayed even if the application exits unexpectedly.

User Experience Considerations

The cost tracking system addresses several user needs:

1. **Transparency**: Provides clear visibility into API usage costs.
2. **Usage Monitoring**: Helps users track and manage their API spending.
3. **Efficiency Insights**: Shows both total runtime and API time, helping identify bottlenecks.
4. **Threshold Warnings**: Alerts users when they've spent significant amounts.
5. **Documentation Links**: Provides resources for learning more about cost management.

The `/cost` command and associated systems represent Claude Code's approach to transparent cost management, giving users control over their API usage while maintaining a simple, unobtrusive interface.

onboarding Command

The `onboarding` command provides a guided first-run experience for new users, helping them configure Claude Code to their preferences and introducing them to the tool's capabilities.

Implementation

The command is implemented in `commands/onboarding.tsx` as a type: 'local-jsx' command that renders a React component:

```
import * as React from "react";
import type { Command } from "../commands";
import { Onboarding } from "../components/Onboarding";
import { clearTerminal } from "../utils/terminal";
import { getGlobalConfig, saveGlobalConfig } from "../utils/config";
import { clearConversation } from "./clear";

export default {
  type: "local-jsx",
  name: "onboarding",
  description: "[ANT-ONLY] Run through the onboarding flow",
  isEnabled: true,
  isHidden: false,
  async call(onDone, context) {
    await clearTerminal();
    const config = getGlobalConfig();
    saveGlobalConfig({
      ...config,
      theme: "dark",
    });

    return (
      <Onboarding
        onDone={async () => {
          clearConversation(context);
          onDone();
        }}
      />
    );
  },
  userFacingName() {
    return "onboarding";
  },
} satisfies Command;
```

The command delegates to the `Onboarding` component in `components/Onboarding.tsx`, which handles the multi-step onboarding flow.

Functionality

The `onboarding` command implements a comprehensive first-run experience:

1. **Multi-Step Flow**:
    - Walks users through a series of configuration steps with a smooth, guided experience
    - Includes theme selection, usage guidance, and model selection
    - Uses a stack-based navigation system for intuitive flow between steps
2. **Theme Configuration**:
    - Allows users to choose between light and dark themes
    - Includes colorblind-friendly theme options for accessibility
    - Provides a live preview of the selected theme using a code diff example
3. **Usage Guidelines**:
    - Introduces users to effective ways of using Claude Code
    - Explains how to provide clear context and work with the tool
    - Sets appropriate expectations for the tool's capabilities
4. **Model Selection**:
    - Guides users through configuring AI provider and model settings
    - Uses the `ModelSelector` component for a consistent model selection experience
    - Allows configuration of both small and large models for different tasks
5. **Configuration Persistence**:
    - Saves user preferences to the global configuration
    - Marks onboarding as complete to prevent repeat runs
    - Clears the conversation after onboarding to provide a clean start

Technical Implementation Notes

The `onboarding` command demonstrates several sophisticated patterns:

1. **Screen Navigation Stack**: Implements a stack-based navigation system for multi-step flow:

```
const [screenStack, setScreenStack] = useState<
  Array<
    | "modelType"
    | "provider"
    | "apiKey"
    | "model"
    | "modelParams"
    | "confirmation"
  >
>(["modelType"]);

// Current screen is always the last item in the stack
const currentScreen = screenStack[screenStack.length - 1];

// Function to navigate to a new screen
const navigateTo = (screen) => {
  setScreenStack((prev) => [...prev, screen]);
};

// Function to go back to the previous screen
const goBack = () => {
  if (screenStack.length > 1) {
    setScreenStack((prev) => prev.slice(0, -1));
  } else {
    onDone();
  }
};
```

2. **Progressive Disclosure**: Presents information in digestible chunks across multiple steps.

3. **Terminal UI Adaptation**: Uses Ink components optimized for terminal rendering:

```
<Box flexDirection="column" gap={1} paddingLeft={1}>
  <Text bold>Using {PRODUCT_NAME} effectively:</Text>
  <Box flexDirection="column" width={70}>
    <OrderedList>
      <OrderedList.Item>
        <Text>
          Start in your project directory
          <Newline />
          <Text color={theme.secondaryText}>
            Files are automatically added to context when needed.
          </Text>
          <Newline />
        </Text>
      </OrderedList.Item>
      {/* Additional list items */}
    </OrderedList>
  </Box>
  <PressEnterToContinue />
</Box>
```

4. **Interactive Components**: Uses custom select components for theme selection with previews:

```
<Select
  options={[
    { label: "Light text", value: "dark" },
    { label: "Dark text", value: "light" },
    {
      label: "Light text (colorblind-friendly)",
      value: "dark-daltonized",
    },
    {
      label: "Dark text (colorblind-friendly)",
      value: "light-daltonized",
    },
  ]}
  onFocus={handleThemePreview}
  onChange={handleThemeSelection}
/>
```

5. **Exit Handling**: Implements `useExitOnCtrlCD` to provide users with a clear way to exit the flow:

```
const exitState = useExitOnCtrlCD(() => process.exit(0));
```

6. **Conditional Rendering**: Uses state to conditionally show different screens:

```
// If we're showing the model selector screen, render it directly
if (showModelSelector) {
  return <ModelSelector onDone={handleModelSelectionDone} />;
}
```

User Experience Benefits

The `onboarding` command addresses several key needs for new users:

1. **Guided Setup**: Provides a structured introduction to Claude Code rather than dropping users into a blank interface.

2. **Preference Customization**: Allows users to set their preferences immediately, increasing comfort with the tool.
3. **Learning Opportunity**: Teaches best practices for using the tool effectively from the start.
4. **Accessibility Awareness**: Explicitly offers colorblind-friendly themes, demonstrating attention to accessibility.
5. **Progressive Complexity**: Introduces features gradually, avoiding overwhelming new users.

The `onboarding` command exemplifies Claude Code's attention to user experience, ensuring new users can quickly set up the tool according to their preferences and learn how to use it effectively from the beginning.

pr_comments Command

The `pr_comments` command provides a specialized interface for retrieving and displaying GitHub pull request comments, helping users review feedback on their code without leaving the terminal.

Implementation

The command is implemented in `commands/pr_comments.ts` as a type: 'prompt' command that formulates a specialized request to Claude:

```
import { Command } from '../commands'

export default {
  type: 'prompt',
  name: 'pr-comments',
  description: 'Get comments from a GitHub pull request',
  progressMessage: 'fetching PR comments',
  isEnabled: true,
  isHidden: false,
  userFacingName() {
    return 'pr-comments'
  },
  async getPromptForCommand(args: string) {
    return [
      {
        role: 'user',
        content: [
          {
            type: 'text',
            text: `You are an AI assistant integrated into a git-based
version control system. Your task is to fetch and display comments from a
GitHub pull request.

Follow these steps:

1. Use \`gh pr view --json number,headRepository\` to get the PR number
and repository info
2. Use \`gh api /repos/{owner}/{repo}/issues/{number}/comments\` to get
PR-level comments
3. Use \`gh api /repos/{owner}/{repo}/pulls/{number}/comments\` to get
review comments. Pay particular attention to the following fields:
\`body\`, \`diff_hunk\`, \`path\`, \`line\`, etc. If the comment
references some code, consider fetching it using eg \`gh api
/repos/{owner}/{repo}/contents/{path}?ref={branch} | jq .content -r |
base64 -d\`
4. Parse and format all comments in a readable way
5. Return ONLY the formatted comments, with no additional text

Format the comments as:
```

approvedTools Command

The `approvedTools` command manages which tools Claude has permission to use in a given project.

Implementation

This command is implemented in `commands/approvedTools.ts` and provides functionality to list and remove tools from the approved list. The implementation uses a simple handler pattern with two key functions:

1. `handleListApprovedTools` : Lists all tools that are approved for use in the current directory.
2. `handleRemoveApprovedTool` : Removes a specific tool from the approved list.

The command follows a configurable pattern that supports dependency injection:

```
export type ProjectConfigHandler = {
  getCurrentProjectConfig: () => ProjectConfig;
  saveCurrentProjectConfig: (config: ProjectConfig) => void;
};

// Default config handler using the real implementation
const defaultConfigHandler: ProjectConfigHandler = {
  getCurrentProjectConfig: getCurrentProjectConfigDefault,
  saveCurrentProjectConfig: saveCurrentProjectConfigDefault,
};
```

This design makes the command testable by allowing the injection of mock config handlers.

CLI Integration

In `cli.tsx` , the command is registered as a subcommand under the main `claude` command:

```
const allowedTools = program
  .command("approved-tools")
  .description("Manage approved tools");

allowedTools
  .command("list")
  .description("List all approved tools")
  .action(async () => {
    const result = handleListApprovedTools(getCwd());
    console.log(result);
    process.exit(0);
  });

allowedTools
  .command("remove <tool>")
  .description("Remove a tool from the list of approved tools")
  .action(async (tool: string) => {
    const result = handleRemoveApprovedTool(tool);
    logEvent("tengu_approved_tool_remove", {
      tool,
      success: String(result.success),
    });
    console.log(result.message);
    process.exit(result.success ? 0 : 1);
  });
```

This allows users to use commands like:

- `claude approved-tools list` - To list all approved tools
- `claude approved-tools remove <tool>` - To remove a specific tool from the approved list

Security Implications

The `approvedTools` command plays an important security role in Claude Code's permission system. It allows users to revoke permissions for specific tools, providing a mechanism to limit what Claude can do in a project. This is particularly important for tools that have the potential to modify files or execute commands.

bug Command

The `bug` command provides users with a way to submit bug reports and feedback directly from the CLI interface.

Implementation

The command is implemented in `commands/bug.tsx` and leverages React components to create an interactive feedback form:

```
import { Command } from "../commands";
import { Bug } from "../components/Bug";
import * as React from "react";
import { PRODUCT_NAME } from "../constants/product";

const bug = {
  type: "local-jsx",
  name: "bug",
  description: `Submit feedback about ${PRODUCT_NAME}`,
  isEnabled: true,
  isHidden: false,
  async call(onDone) {
    return <Bug onDone={onDone} />;
  },
  userFacingName() {
    return "bug";
  },
} satisfies Command;

export default bug;
```

Unlike pure command-line commands, this command uses the `type: 'local-jsx'` designation, which allows it to render a React component as its output. This enables a rich, interactive interface for gathering bug reports.

UI Component

The core functionality is housed in the `Bug` component in `components/Bug.tsx`. Key aspects of this component include:

1. **Multi-Step Form**: The UI guides users through a multi-step process:

    - User input (description of the bug)
    - Consent for information collection
    - Submission
    - Completion

2. **Information Collection**: The component collects:

    - User-provided bug description
    - Environment information (platform, terminal, version)
    - Git repository metadata (disabled in the current implementation)
    - Model settings (without API keys)

3. **GitHub Integration**: After submission, users can create a GitHub issue with pre-filled information including:

    - Bug description
    - Environment info
    - Model settings

4. **Privacy Considerations**: The component has been carefully designed to avoid collecting sensitive information:

    - No API keys are included
    - Personal identifiers have been removed
    - Direct submission to Anthropic's feedback endpoint has been commented out

Technical Features

Several technical aspects of the implementation are worth noting:

1. **Stateful Form Management**: Uses React's `useState` to manage the form state through multiple steps.

2. **Terminal Adaptation**: Adapts to terminal size using the `useTerminalSize` hook to ensure a good experience regardless of window size.

3. **Keyboard Navigation**: Implements customized keyboard handling with `useInput` from Ink to enable intuitive navigation.

4. **Error Handling**: Includes robust error handling for submission failures.

5. **Title Generation**: Originally included a capability to generate a concise title for

the bug report using Claude's Haiku endpoint (currently commented out).

6. **Browser Integration**: Uses the `openBrowser` utility to open the user's default web browser to the GitHub issues page with pre-filled information.

Design Considerations

The bug command exemplifies several design principles of Claude Code's command system:

1. **Rich Terminal UI**: Unlike traditional CLI tools that might just accept arguments, the command provides a fully interactive experience with visual feedback.
2. **Progressive Disclosure**: Information about what will be collected is clearly shown before submission.
3. **Simple Escape Paths**: Users can easily cancel at any point using Escape or Ctrl+C/D.
4. **Clear Status Indicators**: The UI clearly shows the current step and available actions at all times.

This command demonstrates how Claude Code effectively leverages React and Ink to create sophisticated terminal user interfaces for commands that require complex interaction.

clear Command

The `clear` command provides a way to reset the current conversation, clear the message history, and free up context space.

Implementation

The command is implemented in `commands/clear.ts` as a type: 'local' command, which means it doesn't render a UI component but rather performs an operation directly:

```
import { Command } from "../commands";
import { getMessagesSetter } from "../messages";
import { getContext } from "../context";
import { getCodeStyle } from "../utils/style";
import { clearTerminal } from "../utils/terminal";
import { getOriginalCwd, setCwd } from "../utils/state";
import { Message } from "../query";

export async function clearConversation(context: {
  setForkConvoWithMessagesOnTheNextRender: (
    forkConvoWithMessages: Message[]
  ) => void;
}) {
  await clearTerminal();
  getMessagesSetter()([]);
  context.setForkConvoWithMessagesOnTheNextRender([]);
  getContext.cache.clear?.();
  getCodeStyle.cache.clear?.();
  await setCwd(getOriginalCwd());
}

const clear = {
  type: "local",
  name: "clear",
  description: "Clear conversation history and free up context",
  isEnabled: true,
  isHidden: false,
  async call(_, context) {
    clearConversation(context);
    return "";
  },
  userFacingName() {
    return "clear";
  },
} satisfies Command;

export default clear;
```

Functionality

The `clear` command performs several key operations:

1. **Terminal Cleaning**: Uses `clearTerminal()` to visually reset the terminal display.
2. **Message History Reset**: Sets the message array to empty using `getMessagesSetter()([])`.

3. **Context Clearing**: Clears the cached context information using `getContext.cache.clear()`.
4. **Style Cache Reset**: Clears the cached code style information with `getCodeStyle.cache.clear()`.
5. **Working Directory Reset**: Resets the current working directory to the original one using `setCwd(getOriginalCwd())`.
6. **Conversation Forking Reset**: Clears any pending conversation forks via the context parameter.

Technical Implementation Notes

The `clear` command makes use of several architectural patterns in Claude Code:

1. **Getter/Setter Pattern**: Uses message setter functions obtained through `getMessagesSetter()` rather than directly manipulating a message store, allowing for changes to be reactive across the UI.
2. **Cache Invalidation**: Explicitly clears caches for context and code style information to ensure fresh data when the user continues.
3. **State Management**: Demonstrates how state (like current working directory) is reset when clearing a conversation.
4. **Context Parameter**: Receives a context object from the command system that allows it to interact with the component rendering the REPL.
5. **Separate Function**: The core functionality is extracted into a separate `clearConversation` function, which allows it to be used by other parts of the system if needed.

User Experience Considerations

From a UX perspective, the `clear` command provides several benefits:

1. **Context Space Management**: Allows users to free up context space when they hit limitations with model context windows.

2. **Fresh Start**: Provides a clean slate for starting a new conversation without entirely restarting the CLI.
3. **Visual Reset**: The terminal clearing provides immediate visual feedback that the conversation has been reset.

The `clear` command is simple but vital for long-running CLI sessions, where context management becomes crucial for effective use of the LLM.

approvedTools Command

The `approvedTools` command manages which tools Claude has permission to use in a given project.

Implementation

This command is implemented in `commands/approvedTools.ts` and provides functionality to list and remove tools from the approved list. The implementation uses a simple handler pattern with two key functions:

1. `handleListApprovedTools` : Lists all tools that are approved for use in the current directory.
2. `handleRemoveApprovedTool` : Removes a specific tool from the approved list.

The command follows a configurable pattern that supports dependency injection:

```
export type ProjectConfigHandler = {
  getCurrentProjectConfig: () => ProjectConfig;
  saveCurrentProjectConfig: (config: ProjectConfig) => void;
};

// Default config handler using the real implementation
const defaultConfigHandler: ProjectConfigHandler = {
  getCurrentProjectConfig: getCurrentProjectConfigDefault,
  saveCurrentProjectConfig: saveCurrentProjectConfigDefault,
};
```

This design makes the command testable by allowing the injection of mock config handlers.

CLI Integration

In `cli.tsx` , the command is registered as a subcommand under the main `claude` command:

```
const allowedTools = program
  .command("approved-tools")
  .description("Manage approved tools");

allowedTools
  .command("list")
  .description("List all approved tools")
  .action(async () => {
    const result = handleListApprovedTools(getCwd());
    console.log(result);
    process.exit(0);
  });

allowedTools
  .command("remove <tool>")
  .description("Remove a tool from the list of approved tools")
  .action(async (tool: string) => {
    const result = handleRemoveApprovedTool(tool);
    logEvent("tengu_approved_tool_remove", {
      tool,
      success: String(result.success),
    });
    console.log(result.message);
    process.exit(result.success ? 0 : 1);
  });
```

This allows users to use commands like:

- `claude approved-tools list` - To list all approved tools
- `claude approved-tools remove <tool>` - To remove a specific tool from the approved list

Security Implications

The `approvedTools` command plays an important security role in Claude Code's permission system. It allows users to revoke permissions for specific tools, providing a mechanism to limit what Claude can do in a project. This is particularly important for tools that have the potential to modify files or execute commands.

bug Command

The `bug` command provides users with a way to submit bug reports and feedback directly from the CLI interface.

Implementation

The command is implemented in `commands/bug.tsx` and leverages React components to create an interactive feedback form:

```
import { Command } from "../commands";
import { Bug } from "../components/Bug";
import * as React from "react";
import { PRODUCT_NAME } from "../constants/product";

const bug = {
  type: "local-jsx",
  name: "bug",
  description: `Submit feedback about ${PRODUCT_NAME}`,
  isEnabled: true,
  isHidden: false,
  async call(onDone) {
    return <Bug onDone={onDone} />;
  },
  userFacingName() {
    return "bug";
  },
} satisfies Command;

export default bug;
```

Unlike pure command-line commands, this command uses the `type: 'local-jsx'` designation, which allows it to render a React component as its output. This enables a rich, interactive interface for gathering bug reports.

UI Component

The core functionality is housed in the `Bug` component in `components/Bug.tsx`. Key aspects of this component include:

1. **Multi-Step Form**: The UI guides users through a multi-step process:
    - User input (description of the bug)
    - Consent for information collection
    - Submission
    - Completion
2. **Information Collection**: The component collects:

    - User-provided bug description
    - Environment information (platform, terminal, version)
    - Git repository metadata (disabled in the current implementation)
    - Model settings (without API keys)

3. **GitHub Integration**: After submission, users can create a GitHub issue with pre-filled information including:

    - Bug description
    - Environment info
    - Model settings

4. **Privacy Considerations**: The component has been carefully designed to avoid collecting sensitive information:

    - No API keys are included
    - Personal identifiers have been removed
    - Direct submission to Anthropic's feedback endpoint has been commented out

Technical Features

Several technical aspects of the implementation are worth noting:

1. **Stateful Form Management**: Uses React's `useState` to manage the form state through multiple steps.

2. **Terminal Adaptation**: Adapts to terminal size using the `useTerminalSize` hook to ensure a good experience regardless of window size.

3. **Keyboard Navigation**: Implements customized keyboard handling with `useInput` from Ink to enable intuitive navigation.

4. **Error Handling**: Includes robust error handling for submission failures.

5. **Title Generation**: Originally included a capability to generate a concise title for the bug report using Claude's Haiku endpoint (currently commented out).

6. **Browser Integration**: Uses the `openBrowser` utility to open the user's default web browser to the GitHub issues page with pre-filled information.

Design Considerations

The bug command exemplifies several design principles of Claude Code's command system:

1. **Rich Terminal UI**: Unlike traditional CLI tools that might just accept arguments, the command provides a fully interactive experience with visual feedback.
2. **Progressive Disclosure**: Information about what will be collected is clearly shown before submission.
3. **Simple Escape Paths**: Users can easily cancel at any point using Escape or Ctrl+C/D.
4. **Clear Status Indicators**: The UI clearly shows the current step and available actions at all times.

This command demonstrates how Claude Code effectively leverages React and Ink to create sophisticated terminal user interfaces for commands that require complex interaction.

clear Command

The `clear` command provides a way to reset the current conversation, clear the message history, and free up context space.

Implementation

The command is implemented in `commands/clear.ts` as a type: 'local' command, which means it doesn't render a UI component but rather performs an operation directly:

```
import { Command } from "../commands";
import { getMessagesSetter } from "../messages";
import { getContext } from "../context";
import { getCodeStyle } from "../utils/style";
import { clearTerminal } from "../utils/terminal";
import { getOriginalCwd, setCwd } from "../utils/state";
import { Message } from "../query";

export async function clearConversation(context: {
  setForkConvoWithMessagesOnTheNextRender: (
    forkConvoWithMessages: Message[]
  ) => void;
}) {
  await clearTerminal();
  getMessagesSetter()([]);
  context.setForkConvoWithMessagesOnTheNextRender([]);
  getContext.cache.clear?.();
  getCodeStyle.cache.clear?.();
  await setCwd(getOriginalCwd());
}

const clear = {
  type: "local",
  name: "clear",
  description: "Clear conversation history and free up context",
  isEnabled: true,
  isHidden: false,
  async call(_, context) {
    clearConversation(context);
    return "";
  },
  userFacingName() {
    return "clear";
  },
} satisfies Command;

export default clear;
```

Functionality

The `clear` command performs several key operations:

1. **Terminal Cleaning**: Uses `clearTerminal()` to visually reset the terminal display.
2. **Message History Reset**: Sets the message array to empty using `getMessagesSetter()([])`.

3. **Context Clearing**: Clears the cached context information using `getContext.cache.clear()`.
4. **Style Cache Reset**: Clears the cached code style information with `getCodeStyle.cache.clear()`.
5. **Working Directory Reset**: Resets the current working directory to the original one using `setCwd(getOriginalCwd())`.
6. **Conversation Forking Reset**: Clears any pending conversation forks via the context parameter.

Technical Implementation Notes

The `clear` command makes use of several architectural patterns in Claude Code:

1. **Getter/Setter Pattern**: Uses message setter functions obtained through `getMessagesSetter()` rather than directly manipulating a message store, allowing for changes to be reactive across the UI.
2. **Cache Invalidation**: Explicitly clears caches for context and code style information to ensure fresh data when the user continues.
3. **State Management**: Demonstrates how state (like current working directory) is reset when clearing a conversation.
4. **Context Parameter**: Receives a context object from the command system that allows it to interact with the component rendering the REPL.
5. **Separate Function**: The core functionality is extracted into a separate `clearConversation` function, which allows it to be used by other parts of the system if needed.

User Experience Considerations

From a UX perspective, the `clear` command provides several benefits:

1. **Context Space Management**: Allows users to free up context space when they hit limitations with model context windows.

2. **Fresh Start**: Provides a clean slate for starting a new conversation without entirely restarting the CLI.

3. **Visual Reset**: The terminal clearing provides immediate visual feedback that the conversation has been reset.

The `clear` command is simple but vital for long-running CLI sessions, where context management becomes crucial for effective use of the LLM.

compact Command

The `compact` command offers a sophisticated solution to context management by summarizing the conversation history before clearing it, thereby retaining essential context while freeing up token space.

Implementation

The command is implemented in `commands/compact.ts` as a type: 'local' command:

```
import { Command } from "../commands";
import { getContext } from "../context";
import { getMessagesGetter, getMessagesSetter } from "../messages";
import { API_ERROR_MESSAGE_PREFIX, querySonnet } from
"../services/claude";
import {
  createUserMessage,
  normalizeMessagesForAPI,
} from "../utils/messages.js";
import { getCodeStyle } from "../utils/style";
import { clearTerminal } from "../utils/terminal";

const compact = {
  type: "local",
  name: "compact",
  description: "Clear conversation history but keep a summary in
context",
  isEnabled: true,
  isHidden: false,
  async call(
    _,
    {
      options: { tools, slowAndCapableModel },
      abortController,
```

```
      setForkConvoWithMessagesOnTheNextRender,
    }
  ) {
    // Get existing messages before clearing
    const messages = getMessagesGetter()();

    // Add summary request as a new message
    const summaryRequest = createUserMessage(
      "Provide a detailed but concise summary of our conversation above.
Focus on information that would be helpful for continuing the
conversation, including what we did, what we're doing, which files we're
working on, and what we're going to do next."
    );

    const summaryResponse = await querySonnet(
      normalizeMessagesForAPI([...messages, summaryRequest]),
      ["You are a helpful AI assistant tasked with summarizing
conversations."],
      0,
      tools,
      abortController.signal,
      {
        dangerouslySkipPermissions: false,
        model: slowAndCapableModel,
        prependCLISysprompt: true,
      }
    );

    // Extract summary from response, throw if we can't get it
    const content = summaryResponse.message.content;
    const summary =
      typeof content === "string"
        ? content
        : content.length > 0 && content[0]?.type === "text"
        ? content[0].text
        : null;

    if (!summary) {
      throw new Error(
        `Failed to generate conversation summary - response did not
contain valid text content - ${summaryResponse}`
      );
    } else if (summary.startsWith(API_ERROR_MESSAGE_PREFIX)) {
      throw new Error(summary);
    }

    // Substitute low token usage info so that the context-size UI
warning goes
    // away. The actual numbers don't matter too much: `countTokens`
checks the
    // most recent assistant message for usage numbers, so this estimate
will
    // be overridden quickly.
    summaryResponse.message.usage = {
```

```
        input_tokens: 0,
        output_tokens: summaryResponse.message.usage.output_tokens,
        cache_creation_input_tokens: 0,
        cache_read_input_tokens: 0,
      };

      // Clear screen and messages
      await clearTerminal();
      getMessagesSetter()([]);
      setForkConvoWithMessagesOnTheNextRender([
        createUserMessage(
          `Use the /compact command to clear the conversation history, and
start a new conversation with the summary in context.`
        ),
        summaryResponse,
      ]);
      getContext.cache.clear?.();
      getCodeStyle.cache.clear?.();

      return ""; // not used, just for typesafety. TODO: avoid this hack
    },
    userFacingName() {
      return "compact";
    },
  } satisfies Command;

  export default compact;
```

Functionality

The `compact` command implements a sophisticated workflow:

1. **Conversation Summary Generation**:
   - Retrieves the current message history
   - Creates a user message requesting a conversation summary
   - Uses Claude API to generate a summary through the `querySonnet` function
   - Validates the summary to ensure it was successfully generated
2. **Token Usage Management**:
   - Manipulates the usage data to prevent context-size warnings
   - Sets input tokens to 0 to indicate the conversation has been compacted
3. **Context Reset with Summary**:
   - Clears the terminal display

- Resets message history
- Creates a new conversation "fork" containing only:
  - A user message indicating a compact operation occurred
  - The generated summary response
- Clears context and code style caches

Technical Implementation Notes

The `compact` command demonstrates several advanced patterns:

1. **Meta-Conversation**: The command uses Claude to talk about the conversation itself, leveraging the model's summarization abilities.
2. **Model Selection**: Explicitly uses the `slowAndCapableModel` option to ensure high-quality summarization.
3. **Content Extraction Logic**: Implements robust parsing of the response content, handling different content formats (string vs. structured content).
4. **Error Handling**: Provides clear error messages for when summarization fails or when the API returns an error.
5. **Token Manipulation**: Intelligently manipulates token usage information to maintain a good user experience after compaction.
6. **Conversation Forking**: Uses the `setForkConvoWithMessagesOnTheNextRender` mechanism to create a new conversation branch with only the summary.

User Experience Benefits

The `compact` command addresses several key pain points in AI assistant interactions:

1. **Context Window Management**: Helps users stay within token limits while preserving the essence of the conversation.
2. **Conversation Continuity**: Unlike a complete clear, it maintains the thread of discussion through the summary.
3. **Work Session Persistence**: Preserves information about files being edited and tasks in progress.

4. **Smart Reset**: Performs a targeted reset that balances clearing space with maintaining context.

The command is particularly valuable for long development sessions where context limits become an issue but completely starting over would lose important progress information.

config Command

The `config` command provides an interactive terminal interface for viewing and editing Claude Code's configuration settings, including model settings, UI preferences, and API keys.

Implementation

The command is implemented in `commands/config.tsx` as a type: 'local-jsx' command that renders a React component:

```
import { Command } from "../commands";
import { Config } from "../components/Config";
import * as React from "react";

const config = {
  type: "local-jsx",
  name: "config",
  description: "Open config panel",
  isEnabled: true,
  isHidden: false,
  async call(onDone) {
    return <Config onClose={onDone} />;
  },
  userFacingName() {
    return "config";
  },
} satisfies Command;

export default config;
```

Like the `bug` command, this command uses JSX to render an interactive UI component. The actual functionality is implemented in the `Config` component

located in `components/Config.tsx`.

UI Component

The `Config` component implements a rich terminal-based settings interface with the following features:

1. **Settings Management**: Displays a list of configurable settings with their current values.
2. **Multiple Setting Types**: Supports various setting types:
    - `boolean`: Toggle settings (true/false)
    - `enum`: Options from a predefined list
    - `string`: Text input values
    - `number`: Numeric values
3. **Interactive Editing**: Allows users to:
    - Navigate settings with arrow keys
    - Toggle boolean and enum settings with Enter/Space
    - Edit string and number settings with a text input mode
    - Exit configuration with Escape
4. **Configuration Persistence**: Saves settings to a configuration file using `saveGlobalConfig`.

Configuration Options

The component exposes numerous configuration options, including:

1. **Model Configuration**:
    - AI Provider selection (anthropic, openai, custom)
    - API keys for small and large models
    - Model names for both small and large models
    - Base URLs for API endpoints
    - Max token settings
    - Reasoning effort levels
2. **User Interface**:

- Theme selection (light, dark, light-daltonized, dark-daltonized)
- Verbose output toggle

3. **System Settings**:
   - Notification preferences
   - HTTP proxy configuration

Technical Implementation Notes

The `Config` component demonstrates several advanced patterns:

1. **State Management**: Uses React's `useState` to track:
   - Current configuration state
   - Selected setting index
   - Editing mode state
   - Current input text
   - Input validation errors
2. **Reference Comparison**: Maintains a reference to the initial configuration using `useRef` to track changes.
3. **Keyboard Input Handling**: Implements sophisticated keyboard handling for navigation and editing:
   - Arrow keys for selection
   - Enter/Space for toggling/editing
   - Escape for cancellation
   - Input handling with proper validation
4. **Input Sanitization**: Cleans input text to prevent control characters and other problematic input.
5. **Visual Feedback**: Provides clear visual indication of:
   - Currently selected item
   - Editing state
   - Input errors
   - Available actions
6. **Change Tracking**: Tracks and logs configuration changes when exiting.

User Experience Design

The config component showcases several UI/UX design principles for terminal applications:

1. **Modal Interface**: Creates a focused settings panel that temporarily takes over the terminal.
2. **Progressive Disclosure**: Shows relevant controls and options based on the current state.
3. **Clear Instructions**: Displays context-sensitive help text at the bottom of the interface.
4. **Visual Highlighting**: Uses color and indicators to show the current selection and editing state.
5. **Immediate Feedback**: Changes take effect immediately, with visual confirmation.
6. **Multiple Input Methods**: Supports keyboard navigation, toggling, and text input in a unified interface.
7. **Safe Editing**: Provides validation and escape routes for configuration editing.

The `config` command demonstrates how Claude Code effectively combines the simplicity of terminal interfaces with the rich interaction capabilities typically associated with graphical applications, creating a powerful yet accessible configuration experience.

compact Command

The `compact` command offers a sophisticated solution to context management by summarizing the conversation history before clearing it, thereby retaining essential context while freeing up token space.

Implementation

The command is implemented in `commands/compact.ts` as a type: 'local' command:

```
import { Command } from "../commands";
import { getContext } from "../context";
import { getMessagesGetter, getMessagesSetter } from "../messages";
import { API_ERROR_MESSAGE_PREFIX, querySonnet } from
"../services/claude";
import {
  createUserMessage,
  normalizeMessagesForAPI,
} from "../utils/messages.js";
import { getCodeStyle } from "../utils/style";
import { clearTerminal } from "../utils/terminal";

const compact = {
  type: "local",
  name: "compact",
  description: "Clear conversation history but keep a summary in
context",
  isEnabled: true,
  isHidden: false,
  async call(
    _,
    {
      options: { tools, slowAndCapableModel },
      abortController,
      setForkConvoWithMessagesOnTheNextRender,
    }
  ) {
    // Get existing messages before clearing
    const messages = getMessagesGetter()();

    // Add summary request as a new message
    const summaryRequest = createUserMessage(
      "Provide a detailed but concise summary of our conversation above.
Focus on information that would be helpful for continuing the
conversation, including what we did, what we're doing, which files we're
working on, and what we're going to do next."
    );

    const summaryResponse = await querySonnet(
      normalizeMessagesForAPI([...messages, summaryRequest]),
      ["You are a helpful AI assistant tasked with summarizing
conversations."],
      0,
      tools,
      abortController.signal,
      {
        dangerouslySkipPermissions: false,
        model: slowAndCapableModel,
        prependCLISysprompt: true,
      }
```

```
    );

    // Extract summary from response, throw if we can't get it
    const content = summaryResponse.message.content;
    const summary =
      typeof content === "string"
        ? content
        : content.length > 0 && content[0]?.type === "text"
        ? content[0].text
        : null;

    if (!summary) {
      throw new Error(
        `Failed to generate conversation summary - response did not
contain valid text content - ${summaryResponse}`
      );
    } else if (summary.startsWith(API_ERROR_MESSAGE_PREFIX)) {
      throw new Error(summary);
    }

    // Substitute low token usage info so that the context-size UI
warning goes
    // away. The actual numbers don't matter too much: `countTokens`
checks the
    // most recent assistant message for usage numbers, so this estimate
will
    // be overridden quickly.
    summaryResponse.message.usage = {
      input_tokens: 0,
      output_tokens: summaryResponse.message.usage.output_tokens,
      cache_creation_input_tokens: 0,
      cache_read_input_tokens: 0,
    };

    // Clear screen and messages
    await clearTerminal();
    getMessagesSetter()([]);
    setForkConvoWithMessagesOnTheNextRender([
      createUserMessage(
        `Use the /compact command to clear the conversation history, and
start a new conversation with the summary in context.`
      ),
      summaryResponse,
    ]);
    getContext.cache.clear?.();
    getCodeStyle.cache.clear?.();

    return ""; // not used, just for typesafety. TODO: avoid this hack
  },
  userFacingName() {
    return "compact";
  },
} satisfies Command;
```

```
export default compact;
```

Functionality

The `compact` command implements a sophisticated workflow:

1. **Conversation Summary Generation**:
   - Retrieves the current message history
   - Creates a user message requesting a conversation summary
   - Uses Claude API to generate a summary through the `querySonnet` function
   - Validates the summary to ensure it was successfully generated
2. **Token Usage Management**:
   - Manipulates the usage data to prevent context-size warnings
   - Sets input tokens to 0 to indicate the conversation has been compacted
3. **Context Reset with Summary**:
   - Clears the terminal display
   - Resets message history
   - Creates a new conversation "fork" containing only:
     - A user message indicating a compact operation occurred
     - The generated summary response
   - Clears context and code style caches

Technical Implementation Notes

The `compact` command demonstrates several advanced patterns:

1. **Meta-Conversation**: The command uses Claude to talk about the conversation itself, leveraging the model's summarization abilities.
2. **Model Selection**: Explicitly uses the `slowAndCapableModel` option to ensure high-quality summarization.
3. **Content Extraction Logic**: Implements robust parsing of the response content, handling different content formats (string vs. structured content).

4. **Error Handling**: Provides clear error messages for when summarization fails or when the API returns an error.
5. **Token Manipulation**: Intelligently manipulates token usage information to maintain a good user experience after compaction.
6. **Conversation Forking**: Uses the `setForkConvoWithMessagesOnTheNextRender` mechanism to create a new conversation branch with only the summary.

User Experience Benefits

The `compact` command addresses several key pain points in AI assistant interactions:

1. **Context Window Management**: Helps users stay within token limits while preserving the essence of the conversation.
2. **Conversation Continuity**: Unlike a complete clear, it maintains the thread of discussion through the summary.
3. **Work Session Persistence**: Preserves information about files being edited and tasks in progress.
4. **Smart Reset**: Performs a targeted reset that balances clearing space with maintaining context.

The command is particularly valuable for long development sessions where context limits become an issue but completely starting over would lose important progress information.

ctx_viz Command

The `ctx_viz` command provides a detailed visualization of token usage across different components of the Claude conversation context, helping users understand how their context window is being utilized.

Implementation

The command is implemented in `commands/ctx_viz.ts` as a type: 'local' command that generates a formatted table of token usage:

```
import type { Command } from "../commands";
import type { Tool } from "../Tool";
import Table from "cli-table3";
import { getSystemPrompt } from "../constants/prompts";
import { getContext } from "../context";
import { zodToJsonSchema } from "zod-to-json-schema";
import { getMessagesGetter } from "../messages";

// Quick and dirty estimate of bytes per token for rough token counts
const BYTES_PER_TOKEN = 4;

interface Section {
  title: string;
  content: string;
}

interface ToolSummary {
  name: string;
  description: string;
}

function getContextSections(text: string): Section[] {
  const sections: Section[] = [];

  // Find first <context> tag
  const firstContextIndex = text.indexOf("<context");

  // Everything before first tag is Core Sysprompt
  if (firstContextIndex > 0) {
    const coreSysprompt = text.slice(0, firstContextIndex).trim();
    if (coreSysprompt) {
      sections.push({
        title: "Core Sysprompt",
        content: coreSysprompt,
      });
    }
  }

  let currentPos = firstContextIndex;
  let nonContextContent = "";

  const regex = /<context\s+name="([^"]*)">([\s\S]*?)<\/context>/g;
  let match: RegExpExecArray | null;

  while ((match = regex.exec(text)) !== null) {
    // Collect text between context tags
    if (match.index > currentPos) {
      nonContextContent += text.slice(currentPos, match.index);
```

```
    }

    const [, name = "Unnamed Section", content = ""] = match;
    sections.push({
      title: name === "codeStyle" ? "CodeStyle + KODING.md's" : name,
      content: content.trim(),
    });

    currentPos = match.index + match[0].length;
  }

  // Collect remaining text after last tag
  if (currentPos < text.length) {
    nonContextContent += text.slice(currentPos);
  }

  // Add non-contextualized content if present
  const trimmedNonContext = nonContextContent.trim();
  if (trimmedNonContext) {
    sections.push({
      title: "Non-contextualized Content",
      content: trimmedNonContext,
    });
  }

  return sections;
}

function formatTokenCount(bytes: number): string {
  const tokens = bytes / BYTES_PER_TOKEN;
  const k = tokens / 1000;
  return `${Math.round(k * 10) / 10}k`;
}

function formatByteCount(bytes: number): string {
  const kb = bytes / 1024;
  return `${Math.round(kb * 10) / 10}kb`;
}

function createSummaryTable(
  systemText: string,
  systemSections: Section[],
  tools: ToolSummary[],
  messages: unknown
): string {
  const table = new Table({
    head: ["Component", "Tokens", "Size", "% Used"],
    style: { head: ["bold"] },
    chars: {
      mid: "─",
      "left-mid": "├",
      "mid-mid": "┼",
      "right-mid": "┤",
    },
```

```
  });

  const messagesStr = JSON.stringify(messages);
  const toolsStr = JSON.stringify(tools);

  // Calculate total for percentages
  const total = systemText.length + toolsStr.length + messagesStr.length;
  const getPercentage = (n: number) => `${Math.round((n / total) *
100)}%`;

  // System prompt and its sections
  table.push([
    "System prompt",
    formatTokenCount(systemText.length),
    formatByteCount(systemText.length),
    getPercentage(systemText.length),
  ]);
  for (const section of systemSections) {
    table.push([
      `  ${section.title}`,
      formatTokenCount(section.content.length),
      formatByteCount(section.content.length),
      getPercentage(section.content.length),
    ]);
  }

  // Tools
  table.push([
    "Tool definitions",
    formatTokenCount(toolsStr.length),
    formatByteCount(toolsStr.length),
    getPercentage(toolsStr.length),
  ]);
  for (const tool of tools) {
    table.push([
      `  ${tool.name}`,
      formatTokenCount(tool.description.length),
      formatByteCount(tool.description.length),
      getPercentage(tool.description.length),
    ]);
  }

  // Messages and total
  table.push(
    [
      "Messages",
      formatTokenCount(messagesStr.length),
      formatByteCount(messagesStr.length),
      getPercentage(messagesStr.length),
    ],
    ["Total", formatTokenCount(total), formatByteCount(total), "100%"]
  );

  return table.toString();
```

```
}

const command: Command = {
  name: "ctx-viz",
  description:
    "[ANT-ONLY] Show token usage breakdown for the current conversation
context",
  isEnabled: true,
  isHidden: false,
  type: "local",

  userFacingName() {
    return this.name;
  },

  async call(_args: string, cmdContext: { options: { tools: Tool[] } }) {
    // Get tools and system prompt with injected context
    const [systemPromptRaw, sysContext] = await Promise.all([
      getSystemPrompt(),
      getContext(),
    ]);

    const rawTools = cmdContext.options.tools;

    // Full system prompt with context sections injected
    let systemPrompt = systemPromptRaw.join("\n");
    for (const [name, content] of Object.entries(sysContext)) {
      systemPrompt += `\n<context name="${name}">${content}</context>`;
    }

    // Get full tool definitions including prompts and schemas
    const tools = rawTools.map((t) => {
      // Get full prompt and schema
      const fullPrompt = t.prompt({ dangerouslySkipPermissions: false });
      const schema = JSON.stringify(
        "inputJSONSchema" in t && t.inputJSONSchema
          ? t.inputJSONSchema
          : zodToJsonSchema(t.inputSchema)
      );

      return {
        name: t.name,
        description: `${fullPrompt}\n\nSchema:\n${schema}`,
      };
    });

    // Get current messages from REPL
    const messages = getMessagesGetter()();

    const sections = getContextSections(systemPrompt);
    return createSummaryTable(systemPrompt, sections, tools, messages);
  },
};
```

```
export default command;
```

Functionality

The `ctx_viz` command provides a detailed breakdown of token usage across different components of the conversation context:

1. **System Prompt Analysis**:
    - Parses the system prompt to identify its separate sections
    - Extracts `<context>` tags and their contents for individual analysis
    - Identifies core system prompt sections vs. injected context
2. **Token Usage Calculation**:
    - Estimates token usage based on a bytes-per-token approximation
    - Presents data in kilobytes and estimated token counts
    - Calculates percentage usage for each component of the context
3. **Tool Definitions Analysis**:
    - Extracts complete tool definitions including prompts and JSON schemas
    - Calculates token usage per tool
    - Shows the total footprint of tool definitions in the context
4. **Conversation Message Analysis**:
    - Includes the current message history in the analysis
    - Shows what portion of the context window is used by the conversation
5. **Structured Presentation**:
    - Outputs a formatted ASCII table with columns for component, tokens, size, and percentage
    - Uses hierarchical indentation to show the structure of the context
    - Includes totals for complete context usage

Technical Implementation Notes

The `ctx_viz` command demonstrates several sophisticated implementation patterns:

1. **Regex-Based Context Parsing**: Uses regular expressions to parse the context sections from the system prompt, handling nested tags and multi-line content.
2. **Parallel Resource Loading**: Uses `Promise.all` to concurrently fetch system prompt and context data for efficiency.
3. **Tool Schema Introspection**: Extracts JSON schemas from tool definitions using either explicit schemas or by converting Zod schemas to JSON Schema format.
4. **Token Approximation**: Implements a simple but effective token estimation approach based on byte length, which provides a reasonable approximation without requiring a tokenizer.
5. **Table Formatting**: Uses the `cli-table3` library to create a formatted ASCII table with custom styling, making the output readable in a terminal environment.
6. **Context Section Management**: Special-cases certain context sections like "codeStyle" with custom labeling for clarity.

User Experience Benefits

The `ctx_viz` command addresses several important needs for developers using Claude Code:

1. **Context Window Transparency**: Gives users insight into how their limited context window is being utilized.
2. **Optimization Opportunities**: Helps identify large components that might be consuming excessive context, enabling targeted optimization.
3. **Debugging Aid**: Provides a debugging tool for situations where context limitations are affecting Claude's performance.
4. **System Prompt Visibility**: Makes the usually hidden system prompt and context visible to users for better understanding of Claude's behavior.

The command is particularly valuable for advanced users and developers who need to understand and optimize their context usage to get the most out of Claude's

capabilities within token limitations.

doctor Command

The `doctor` command provides a diagnostic tool for checking the health of the Claude Code installation, with a focus on npm permissions required for proper auto-updating functionality.

Implementation

The command is implemented in `commands/doctor.ts` as a type: 'local-jsx' command that renders a React component:

```
import React from "react";
import type { Command } from "../commands";
import { Doctor } from "../screens/Doctor";

const doctor: Command = {
  name: "doctor",
  description: "Checks the health of your Claude Code installation",
  isEnabled: true,
  isHidden: false,
  userFacingName() {
    return "doctor";
  },
  type: "local-jsx",
  call(onDone) {
    const element = React.createElement(Doctor, {
      onDone,
      doctorMode: true,
    });
    return Promise.resolve(element);
  },
};

export default doctor;
```

The command uses the `Doctor` screen component defined in `screens/Doctor.tsx`, passing the special `doctorMode` flag to indicate it's being used as a diagnostic tool rather than during initialization.

Functionality

The `doctor` command implements a comprehensive installation health check focused on npm permissions:

1. **Permissions Verification**:
    - Checks if the npm global installation directory has correct write permissions
    - Determines the current npm prefix path
    - Validates if auto-updates can function correctly
2. **Status Reporting**:
    - Provides clear success or failure messages about installation health
    - Shows a green checkmark and confirmation message for healthy installations
    - Presents an interactive dialog for resolving permission issues
3. **Problem Resolution**:
    - Offers three remediation options when permission problems are detected:
        1. **Manual Fix**: Provides a sudo/icacls command to fix permissions on the current npm prefix
        2. **New Prefix**: Guides the user through creating a new npm prefix in their home directory
        3. **Skip**: Allows deferring the fix until later
4. **Installation Repair**:
    - For the "new prefix" option, provides a guided, step-by-step process to:
        - Create a new directory for npm global packages
        - Configure npm to use the new location
        - Update shell PATH configurations
        - Reinstall Claude Code globally

Technical Implementation Notes

The command demonstrates several sophisticated implementation patterns:

1. **Component-Based Architecture**: Uses React components for the UI, allowing for a rich interactive experience.
2. **Platform-Aware Logic**: Implements different permission-fixing approaches for

Windows vs. Unix-like systems:

   - Windows: Uses `icacls` for permission management and `setx` for PATH updates
   - Unix: Uses `sudo chown/chmod` for permission fixes and shell config file updates

3. **Shell Detection**: For Unix platforms, identifies and updates multiple possible shell configuration files:

   - .bashrc and .bash_profile for Bash
   - .zshrc for Zsh
   - config.fish for Fish shell

4. **Status Management**: Uses React state to track:

   - Permission check status
   - Selected remediation option
   - Custom prefix path (if chosen)
   - Step-by-step installation progress

5. **Lock-Based Concurrency Control**: Implements a file-based locking mechanism to prevent multiple processes from attempting auto-updates simultaneously.

6. **Error Handling**: Provides detailed error reporting and recovery options:

   - Shows precisely which operation failed
   - Offers alternative approaches when errors occur
   - Logs error details for debugging

User Experience Benefits

The `doctor` command addresses several important pain points for CLI tool users:

1. **Installation Troubleshooting**: Provides clear diagnostics and fixes for common problems:

   - Permission issues that prevent updates
   - Global npm configuration problems
   - PATH configuration issues

2. **Security-Conscious Design**: Offers both privileged (sudo) and non-privileged (new prefix) solutions, allowing users to choose based on their security

preferences.

3. **Multi-Platform Support**: Works identically across Windows, macOS, and Linux with platform-appropriate solutions.
4. **Shell Environment Enhancement**: Automatically updates shell configuration files to ensure the installation works correctly across terminal sessions.
5. **Visual Progress Feedback**: Uses spinners and checkmarks to keep users informed about long-running operations.

The command provides a comprehensive diagnostics and repair tool that helps maintain a healthy Claude Code installation, particularly focusing on the auto-update capability which is crucial for keeping the tool current with new features and improvements.

help Command

The `help` command provides users with information about Claude Code's usage, available commands, and general guidance for effective interaction with the tool.

Implementation

The command is implemented in `commands/help.tsx` as a type: 'local-jsx' command that renders a React component:

```
import { Command } from "../commands";
import { Help } from "../components/Help";
import * as React from "react";

const help = {
  type: "local-jsx",
  name: "help",
  description: "Show help and available commands",
  isEnabled: true,
  isHidden: false,
  async call(onDone, { options: { commands } }) {
    return <Help commands={commands} onClose={onDone} />;
  },
  userFacingName() {
    return "help";
  },
} satisfies Command;

export default help;
```

The command delegates to the `Help` component in `components/Help.tsx`, passing the list of available commands and an `onClose` callback to handle when the help screen should be dismissed.

Functionality

The `Help` component implements a progressive disclosure pattern for displaying information:

1. **Basic Information**:
    - Shows the product name and version
    - Displays a brief description of Claude Code's capabilities and limitations
    - Presents a disclaimer about the beta nature of the product
2. **Usage Modes** (shown after a brief delay):
    - Explains the two primary ways to use Claude Code:
        - REPL (interactive session)
        - Non-interactive mode with the `-p` flag for one-off questions
    - Mentions that users can run `claude -h` for additional command-line options
3. **Common Tasks** (shown after a longer delay):

- Lists examples of typical use cases:
  - Asking questions about the codebase
  - Editing files
  - Fixing errors
  - Running commands
  - Running bash commands

4. **Available Commands** (shown after the longest delay):
   - Displays a comprehensive list of all enabled slash commands
   - Shows command names and descriptions
   - Filters out hidden commands

5. **Additional Resources**:
   - Provides links for getting more help
   - Shows different resources based on user type (internal vs. external)

Technical Implementation Notes

The `help` command demonstrates several effective patterns:

1. **Progressive Disclosure**: Uses a time-based mechanism to gradually reveal information:

   ```
   const [count, setCount] = React.useState(0);

   React.useEffect(() => {
     const timer = setTimeout(() => {
       if (count < 3) {
         setCount(count + 1);
       }
     }, 250);

     return () => clearTimeout(timer);
   }, [count]);
   ```

   This approach avoids overwhelming users with too much information at once, showing more details as they spend time on the help screen.

2. **Filtering System**: Uses `filter` to show only non-hidden commands:

```
const filteredCommands = commands.filter((cmd) => !cmd.isHidden);
```

   This keeps the help screen focused on commands relevant to users.

3. **Dynamic Resource Links**: Changes help resources based on user type:

```
const isInternal = process.env.USER_TYPE === "ant";
const moreHelp = isInternal
  ? "[ANT-ONLY] For more help: go/claude-cli or #claude-cli-
feedback"
  : `Learn more at: ${MACRO.README_URL}`;
```

   This customizes the experience for different user populations.

4. **Input Handling**: Uses Ink's `useInput` hook to detect when the user presses Enter:

```
useInput((_, key) => {
  if (key.return) onClose();
});
```

   This allows for clean dismissal of the help screen.

5. **Conditional Rendering**: Uses the `count` state to progressively show different sections:

```
{
  count >= 1 && (
    <Box flexDirection="column" marginTop={1}>
      <Text bold>Usage Modes:</Text>
      {/* Usage content */}
    </Box>
  );
}
```

This creates the staggered reveal effect for information groups.

User Experience Benefits

The `help` command addresses several important needs for CLI tool users:

1. **Onboarding Assistance**: Provides new users with immediate guidance on how to use the tool effectively.
2. **Command Discovery**: Makes it easy to see what slash commands are available without having to memorize them.
3. **Progressive Learning**: The staggered reveal of information allows users to absorb basics first before seeing more advanced options.
4. **Usage Examples**: Shows concrete examples of common use cases, helping users understand practical applications.
5. **Quick Reference**: Serves as a compact reference for command options during regular use.

The help command exemplifies Claude Code's approach to user experience: it's focused on being informative while remaining concise and unobtrusive, with multiple levels of detail available as users need them.

init Command

The `init` command helps users initialize a new project for use with Claude Code by

creating a KODING.md file that contains key information about the codebase and project.

Implementation

The command is implemented in `commands/init.ts` as a type: 'prompt' command that passes a specific request to Claude:

```
import type { Command } from "../commands";
import { markProjectOnboardingComplete } from "../ProjectOnboarding";

const command = {
  type: "prompt",
  name: "init",
  description: "Initialize a new KODING.md file with codebase
documentation",
  isEnabled: true,
  isHidden: false,
  progressMessage: "analyzing your codebase",
  userFacingName() {
    return "init";
  },
  async getPromptForCommand(_args: string) {
    // Mark onboarding as complete when init command is run
    markProjectOnboardingComplete();
    return [
      {
        role: "user",
        content: [
          {
            type: "text",
            text: `Please analyze this codebase and create a KODING.md
file containing:
1. Build/lint/test commands - especially for running a single test
2. Code style guidelines including imports, formatting, types, naming
conventions, error handling, etc.

The file you create will be given to agentic coding agents (such as
yourself) that operate in this repository. Make it about 20 lines long.
If there's already a KODING.md, improve it.
If there are Cursor rules (in .cursor/rules/ or .cursorrules) or Copilot
rules (in .github/copilot-instructions.md), make sure to include them.`,
          },
        ],
      },
    ];
  },
} satisfies Command;

export default command;
```

Unlike many other commands that directly perform actions, the `init` command is implemented as a 'prompt' type command that simply formulates a specific request to Claude to analyze the codebase and generate a KODING.md file.

Functionality

The `init` command serves several important functions:

1. **Project Configuration Generation**:
    - Instructs Claude to analyze the codebase structure, conventions, and patterns
    - Generates a KODING.md file with essential project information
    - Focuses on capturing build/lint/test commands and code style guidelines
2. **Existing File Enhancement**:
    - Checks if a KODING.md file already exists
    - Improves the existing file with additional information if present
    - Preserves existing content while enhancing where needed
3. **Integration with Other Tools**:
    - Looks for Cursor rules (in .cursor/rules/ or .cursorrules)
    - Looks for Copilot instructions (in .github/copilot-instructions.md)
    - Incorporates these rules into the KODING.md file for a unified guidance document
4. **Onboarding Status Tracking**:
    - Calls `markProjectOnboardingComplete()` to update the project configuration
    - This flags the project as having completed onboarding steps
    - Prevents repeated onboarding prompts in future sessions

Technical Implementation Notes

The `init` command demonstrates several interesting technical aspects:

1. **Command Type**: Uses the 'prompt' type rather than 'local' or 'local-jsx', which means it sets up a specific request to Claude rather than executing custom logic or rendering a UI component.
2. **Progress Message**: Includes a `progressMessage` property ("analyzing your codebase") that's displayed to users while Claude processes the request, providing feedback during what could be a longer operation.
3. **Project Onboarding Integration**: The command is part of the project onboarding workflow, tracking completion state in the project configuration:

```
export function markProjectOnboardingComplete(): void {
  const projectConfig = getCurrentProjectConfig();
  if (!projectConfig.hasCompletedProjectOnboarding) {
    saveCurrentProjectConfig({
      ...projectConfig,
      hasCompletedProjectOnboarding: true,
    });
  }
}
```

4. **External Tool Recognition**: Explicitly looks for configuration files from other AI coding tools (Cursor, Copilot) to create a unified guidance document, showing awareness of the broader AI coding tool ecosystem.

5. **Conciseness Guidance**: Explicitly requests a document of around 20 lines, balancing comprehensive information with brevity for practical usage.

User Experience Benefits

The `init` command provides several important benefits for Claude Code users:

1. **Simplified Project Setup**: Makes it easy to prepare a codebase for effective use with Claude Code with a single command.

2. **Consistent Memory**: Creates a standardized document that Claude Code will access in future sessions, providing consistent context.

3. **Command Discovery**: By capturing build/test/lint commands, it helps Claude remember the correct commands for the project, reducing the need for users to repeatedly provide them.

4. **Code Style Guidance**: Helps Claude generate code that matches the project's conventions, improving integration of AI-generated code.

5. **Onboarding Pathway**: Serves as a key step in the onboarding flow for new projects, guiding users toward the most effective usage patterns.

The `init` command exemplifies Claude Code's overall approach to becoming more

effective over time by creating persistent context that improves future interactions. By capturing project-specific information in a standardized format, it enables Claude to provide more tailored assistance for each unique codebase.

listen Command

The `listen` command provides macOS users with the ability to dictate to Claude Code using speech recognition, enhancing accessibility and offering an alternative input method.

Implementation

The command is implemented in `commands/listen.ts` as a type: 'local' command that invokes macOS's built-in dictation feature:

```
import { Command } from "../commands";
import { logError } from "../utils/log";
import { execFileNoThrow } from "../utils/execFileNoThrow";

const isEnabled =
  process.platform === "darwin" &&
  ["iTerm.app", "Apple_Terminal"].includes(process.env.TERM_PROGRAM ||
"");

const listen: Command = {
  type: "local",
  name: "listen",
  description: "Activates speech recognition and transcribes speech to
text",
  isEnabled: isEnabled,
  isHidden: isEnabled,
  userFacingName() {
    return "listen";
  },
  async call(_, { abortController }) {
    // Start dictation using AppleScript
    const script = `tell application "System Events" to tell ¬
(the first process whose frontmost is true) to tell ¬
menu bar 1 to tell ¬
menu bar item "Edit" to tell ¬
menu "Edit" to tell ¬
menu item "Start Dictation" to ¬
if exists then click it`;

    const { stderr, code } = await execFileNoThrow(
      "osascript",
      ["-e", script],
      abortController.signal
    );

    if (code !== 0) {
      logError(`Failed to start dictation: ${stderr}`);
      return "Failed to start dictation";
    }
    return "Dictation started. Press esc to stop.";
  },
};

export default listen;
```

The command uses AppleScript to trigger macOS's built-in dictation feature, automating the process of selecting "Start Dictation" from the Edit menu.

Functionality

The `listen` command provides a simple but effective accessibility enhancement:

1. **Platform Integration**:
    - Invokes macOS's built-in dictation feature
    - Uses AppleScript to navigate the application's menu structure
    - Triggers the "Start Dictation" menu item in the Edit menu
2. **Status Reporting**:
    - Returns clear success or error messages
    - Provides user instructions on how to stop dictation
    - Logs detailed error information when dictation fails to start
3. **Platform-Specific Availability**:
    - Only enabled on macOS platforms
    - Further restricted to specific terminal applications (iTerm.app and Apple_Terminal)
    - Hidden from command listings on unsupported platforms

Technical Implementation Notes

The `listen` command demonstrates several interesting technical approaches:

1. **Platform Detection**: Uses environment variables and platform checks to determine when the command should be available:

```
const isEnabled =
  process.platform === "darwin" &&
  ["iTerm.app", "Apple_Terminal"].includes(process.env.TERM_PROGRAM
|| "");
```

2. **AppleScript Integration**: Uses a complex AppleScript command to navigate through application menus:

```
const script = `tell application "System Events" to tell ¬
(the first process whose frontmost is true) to tell ¬
menu bar 1 to tell ¬
menu bar item "Edit" to tell ¬
menu "Edit" to tell ¬
menu item "Start Dictation" to ¬
if exists then click it`;
```

This script navigates the UI hierarchy to find and click the dictation menu item.

3. **Command Visibility Control**: Uses the same conditions for both `isEnabled` and `isHidden`, ensuring the command is simultaneously enabled and hidden on supported platforms:

```
isEnabled: isEnabled,
isHidden: isEnabled,
```

This unusual pattern makes the command available but not visible in help listings, suggesting it's an internal or experimental feature.

4. **Error Handling**: Implements robust error handling with detailed logging:

```
if (code !== 0) {
  logError(`Failed to start dictation: ${stderr}`);
  return "Failed to start dictation";
}
```

User Experience Considerations

While simple in implementation, the `listen` command addresses several important user needs:

1. **Accessibility Enhancement**: Provides an alternative input method for users who prefer or require speech recognition.

2. **Workflow Efficiency**: Eliminates the need to manually navigate menus to start dictation, streamlining the process.
3. **Integrated Experience**: Keeps users within the Claude Code interface rather than requiring them to use separate dictation tools.
4. **Platform Integration**: Leverages native OS capabilities rather than implementing custom speech recognition, ensuring high-quality transcription.

The `listen` command demonstrates Claude Code's commitment to accessibility and platform integration, even if the implementation is relatively simple. By leveraging existing OS capabilities rather than reinventing them, it provides a valuable feature with minimal code complexity.

login Command

The `login` command provides users with a secure OAuth 2.0 authentication flow to connect Claude Code with their Anthropic Console account, enabling API access and proper billing.

Implementation

The command is implemented in `commands/login.tsx` as a type: 'local-jsx' command that renders a React component for the authentication flow:

```tsx
import * as React from "react";
import type { Command } from "../commands";
import { ConsoleOAuthFlow } from "../components/ConsoleOAuthFlow";
import { clearTerminal } from "../utils/terminal";
import { isLoggedInToAnthropic } from "../utils/auth";
import { useExitOnCtrlCD } from "../hooks/useExitOnCtrlCD";
import { Box, Text } from "ink";
import { clearConversation } from "./clear";

export default () =>
  ({
    type: "local-jsx",
    name: "login",
    description: isLoggedInToAnthropic()
      ? "Switch Anthropic accounts"
      : "Sign in with your Anthropic account",
    isEnabled: true,
    isHidden: false,
    async call(onDone, context) {
      await clearTerminal();
      return (
        <Login
          onDone={async () => {
            clearConversation(context);
            onDone();
          }}
        />
      );
    },
    userFacingName() {
      return "login";
    },
  } satisfies Command);

function Login(props: { onDone: () => void }) {
  const exitState = useExitOnCtrlCD(props.onDone);
  return (
    <Box flexDirection="column">
      <ConsoleOAuthFlow onDone={props.onDone} />
      <Box marginLeft={3}>
        <Text dimColor>
          {exitState.pending ? (
            <>Press {exitState.keyName} again to exit</>
          ) : (
            ""
          )}
        </Text>
      </Box>
    </Box>
  );
}
```

The command uses a factory function to dynamically create a command object that adapts its description based on the current login state. The main logic is delegated to the `ConsoleOAuthFlow` component, which handles the OAuth flow with the Anthropic API.

Functionality

The `login` command implements a comprehensive authentication flow:

1. **OAuth 2.0 Integration**:
    - Uses OAuth 2.0 with PKCE (Proof Key for Code Exchange) for security
    - Opens a browser for the user to log in to Anthropic Console
    - Handles both automatic browser redirect and manual code entry flows
    - Securely exchanges authorization codes for access tokens
2. **Account Management**:
    - Creates and stores API keys for authenticated users
    - Normalizes and securely saves keys in the global configuration
    - Tracks account information including account UUID and organization details
    - Provides context-aware descriptions ("Sign in" vs "Switch accounts")
3. **Security Features**:
    - Implements state verification to prevent CSRF attacks
    - Uses code verifiers and challenges for secure code exchange
    - Validates all tokens and responses from the authentication server
    - Provides clear error messages for authentication failures
4. **UI Experience**:
    - Clears the terminal for a clean login experience
    - Provides a progressive disclosure flow with appropriate status messages
    - Offers fallback mechanisms for cases where automatic browser opening fails
    - Shows loading indicators during asynchronous operations

Technical Implementation Notes

The login command demonstrates several sophisticated technical approaches:

1. **Local HTTP Server**: Creates a temporary HTTP server for OAuth callback handling:

```
this.server = http.createServer(
  (req: IncomingMessage, res: ServerResponse) => {
    const parsedUrl = url.parse(req.url || "", true);
    if (parsedUrl.pathname === "/callback") {
      // Handle OAuth callback
    }
  }
);
```

2. **PKCE Implementation**: Implements the Proof Key for Code Exchange extension to OAuth:

```
function generateCodeVerifier(): string {
  return base64URLEncode(crypto.randomBytes(32));
}

async function generateCodeChallenge(verifier: string):
Promise<string> {
  const encoder = new TextEncoder();
  const data = encoder.encode(verifier);
  const digest = await crypto.subtle.digest("SHA-256", data);
  return base64URLEncode(Buffer.from(digest));
}
```

3. **Parallel Authentication Paths**: Supports both automatic and manual authentication flows:

```
const { autoUrl, manualUrl } = this.generateAuthUrls(codeChallenge,
state);
await authURLHandler(manualUrl); // Show manual URL in UI
await openBrowser(autoUrl); // Try automatic browser opening
```

4. **Promise-Based Flow Control**: Uses promises to coordinate the asynchronous authentication flow:

```
const { authorizationCode, useManualRedirect } = await new Promise<{
  authorizationCode: string;
  useManualRedirect: boolean;
}>((resolve, reject) => {
  this.pendingCodePromise = { resolve, reject };
  this.startLocalServer(state, onReady);
});
```

5. **State Management with React**: Uses React state and hooks for UI management:

```
const [oauthStatus, setOAuthStatus] = useState<OAuthStatus>({
  state: "idle",
});
```

6. **Error Recovery**: Implements sophisticated error handling with retry mechanisms:

```
if (oauthStatus.state === "error" && oauthStatus.toRetry) {
  setPastedCode("");
  setOAuthStatus({
    state: "about_to_retry",
    nextState: oauthStatus.toRetry,
  });
}
```

User Experience Benefits

The `login` command addresses several important user needs:

1. **Seamless Authentication**: Provides a smooth authentication experience

without requiring manual API key creation or copying.

2. **Cross-Platform Compatibility**: Works across different operating systems and browsers.
3. **Fallback Mechanisms**: Offers manual code entry when automatic browser redirection fails.
4. **Clear Progress Indicators**: Shows detailed status messages throughout the authentication process.
5. **Error Resilience**: Provides helpful error messages and retry options when authentication issues occur.
6. **Account Switching**: Allows users to easily switch between different Anthropic accounts.

The login command exemplifies Claude Code's approach to security and user experience, implementing a complex authentication flow with attention to both security best practices and ease of use.

logout Command

The `logout` command provides users with the ability to sign out from their Anthropic account, removing stored authentication credentials and API keys from the local configuration.

Implementation

The command is implemented in `commands/logout.tsx` as a type: 'local-jsx' command that handles the logout process and renders a confirmation message:

```
import * as React from "react";
import type { Command } from "../commands";
import { getGlobalConfig, saveGlobalConfig } from "../utils/config";
import { clearTerminal } from "../utils/terminal";
import { Text } from "ink";

export default {
  type: "local-jsx",
  name: "logout",
  description: "Sign out from your Anthropic account",
  isEnabled: true,
  isHidden: false,
  async call() {
    await clearTerminal();

    const config = getGlobalConfig();

    config.oauthAccount = undefined;
    config.primaryApiKey = undefined;
    config.hasCompletedOnboarding = false;

    if (config.customApiKeyResponses?.approved) {
      config.customApiKeyResponses.approved = [];
    }

    saveGlobalConfig(config);

    const message = (
      <Text>Successfully logged out from your Anthropic account.</Text>
    );

    setTimeout(() => {
      process.exit(0);
    }, 200);

    return message;
  },
  userFacingName() {
    return "logout";
  },
} satisfies Command;
```

Unlike the more complex `login` command, the `logout` command is relatively straightforward, focusing on removing authentication data from the configuration and providing a clean exit.

Functionality

The `logout` command performs several critical operations:

1. **Credential Removal**:
    - Clears the OAuth account information from the global configuration
    - Removes the primary API key used for authentication
    - Erases the list of approved API keys from storage
    - Resets the onboarding completion status
2. **User Experience**:
    - Clears the terminal before displaying the logout message
    - Provides a clear confirmation message about successful logout
    - Exits the application completely after a short delay
    - Ensures a clean break with the authenticated session
3. **Security Focus**:
    - Removes all sensitive authentication data from the local configuration
    - Ensures the next application start will require re-authentication
    - Prevents accidental API usage with old credentials
    - Provides a clean slate for a new login if desired

Technical Implementation Notes

Despite its relative simplicity, the `logout` command demonstrates several interesting implementation details:

1. **Configuration Management**: Uses the global configuration system to handle persistent state:

```
const config = getGlobalConfig();

config.oauthAccount = undefined;
config.primaryApiKey = undefined;
config.hasCompletedOnboarding = false;

if (config.customApiKeyResponses?.approved) {
  config.customApiKeyResponses.approved = [];
}

saveGlobalConfig(config);
```

2. **Graceful Exit Strategy**: Uses a timeout to allow the message to be displayed before exiting:

```
setTimeout(() => {
  process.exit(0);
}, 200);
```

   This ensures the user sees confirmation before the application closes.

3. **Type Safety**: Uses the `satisfies Command` pattern to ensure type correctness:

```
export default {
  // Command implementation
} satisfies Command;
```

4. **Terminal Management**: Clears the terminal before displaying the logout confirmation:

```
await clearTerminal();
```

   This creates a clean visual experience for the logout process.

5. **Optional Field Handling**: Carefully checks for the existence of optional configuration fields:

```
if (config.customApiKeyResponses?.approved) {
  config.customApiKeyResponses.approved = [];
}
```

User Experience Benefits

The `logout` command addresses several important user needs:

1. **Account Security**: Provides a clear way to remove credentials when sharing devices or ending a session.
2. **User Confidence**: Confirms successful logout with a clear message, reassuring users their credentials have been removed.
3. **Clean Exit**: Exits the application completely, avoiding any state confusion in the current process.
4. **Simplicity**: Keeps the logout process straightforward and quick, with minimal user interaction required.
5. **Fresh Start**: Resets the onboarding status, ensuring a proper re-onboarding flow on next login.

The `logout` command provides a necessary counterpart to the `login` command, completing the authentication lifecycle with a secure, clean way to end a session. While much simpler than its login counterpart, it maintains the same attention to security and user experience that characterizes Claude Code's approach to authentication management.

model Command

> [!WARNING] This command implementation is specific to the anon-kode fork of

> Claude Code and is not part of the original Claude Code codebase. The analysis below pertains to this specific implementation rather than standard Claude Code functionality.

The `model` command provides users with a comprehensive interface to configure and customize the AI models used by Claude Code, enabling fine-grained control over model selection, parameters, and provider settings.

Implementation

The command is implemented in `commands/model.tsx` as a type: 'local-jsx' command that renders a React component for model configuration:

```
import React from "react";
import { render } from "ink";
import { ModelSelector } from "../components/ModelSelector";
import { enableConfigs } from "../utils/config";

export const help = "Change your AI provider and model settings";
export const description = "Change your AI provider and model settings";
export const isEnabled = true;
export const isHidden = false;
export const name = "model";
export const type = "local-jsx";

export function userFacingName(): string {
  return name;
}

export async function call(
  onDone: (result?: string) => void,
  { abortController }: { abortController?: AbortController }
): Promise<React.ReactNode> {
  enableConfigs();
  abortController?.abort?.();
  return (
    <ModelSelector
      onDone={() => {
        onDone();
      }}
    />
  );
}
```

The command uses a different export style than other commands, directly exporting properties and functions rather than a single object. The main functionality is handled by the `ModelSelector` component, which provides an interactive UI for configuring model settings.

Functionality

The `model` command provides a sophisticated model selection and configuration workflow:

1. **Multi-Model Management**:
    - Allows configuring both "large" and "small" models separately or together
    - Provides different models for different task complexities for optimal cost/performance
    - Shows current configuration information for reference
2. **Provider Selection**:
    - Supports multiple AI providers (Anthropic, OpenAI, Gemini, etc.)
    - Dynamically fetches available models from the selected provider's API
    - Handles provider-specific API requirements and authentication
3. **Model Parameters**:
    - Configures maximum token settings for response length control
    - Offers reasoning effort controls for supported models (low/medium/high)
    - Preserves provider-specific configuration options
4. **Search and Filtering**:
    - Provides search functionality to filter large model lists
    - Displays model capabilities including token limits and feature support
    - Organizes models with sensible sorting and grouping
5. **API Key Management**:
    - Securely handles API keys for different providers
    - Masks sensitive information during input and display
    - Stores keys securely in the local configuration

Technical Implementation Notes

The `model` command demonstrates several sophisticated technical approaches:

1. **Multi-Step Navigation**: Implements a screen stack pattern for intuitive flow navigation:

```
const [screenStack, setScreenStack] = useState<
  Array<
    | "modelType"
    | "provider"
    | "apiKey"
    | "model"
    | "modelParams"
    | "confirmation"
  >
>(["modelType"]);

// Current screen is always the last item in the stack
const currentScreen = screenStack[screenStack.length - 1];

// Function to navigate to a new screen
const navigateTo = (
  screen:
    | "modelType"
    | "provider"
    | "apiKey"
    | "model"
    | "modelParams"
    | "confirmation"
) => {
  setScreenStack((prev) => [...prev, screen]);
};

// Function to go back to the previous screen
const goBack = () => {
  if (screenStack.length > 1) {
    // Remove the current screen from the stack
    setScreenStack((prev) => prev.slice(0, -1));
  } else {
    // If we're at the first screen, call onDone to exit
    onDone();
  }
};
```

2. **Dynamic Model Loading**: Fetches available models directly from provider APIs:

```
async function fetchModels() {
  setIsLoadingModels(true);
  setModelLoadError(null);

  try {
    // Provider-specific logic...
    const openai = new OpenAI({
      apiKey: apiKey,
      baseURL: baseURL,
      dangerouslyAllowBrowser: true,
    });

    // Fetch the models
    const response = await openai.models.list();

    // Transform the response into our ModelInfo format
    const fetchedModels = [];
    // Process models...

    return fetchedModels;
  } catch (error) {
    setModelLoadError(`Failed to load models: ${error.message}`);
    throw error;
  } finally {
    setIsLoadingModels(false);
  }
}
```

3. **Form Focus Management**: Implements sophisticated form navigation with keyboard support:

```
// Handle Tab key for form navigation in model params screen
useInput((input, key) => {
  if (currentScreen === "modelParams" && key.tab) {
    const formFields = getFormFieldsForModelParams();
    // Move to next field
    setActiveFieldIndex((current) => (current + 1) %
formFields.length);
    return;
  }

  // Handle Enter key for form submission in model params screen
  if (currentScreen === "modelParams" && key.return) {
    const formFields = getFormFieldsForModelParams();

    if (activeFieldIndex === formFields.length - 1) {
      // If on the Continue button, submit the form
      handleModelParamsSubmit();
    }
    return;
  }
});
```

4. **Provider-Specific Handling**: Implements custom logic for different AI providers:

```
// For Gemini, use the separate fetchGeminiModels function
if (selectedProvider === "gemini") {
  const geminiModels = await fetchGeminiModels();
  setAvailableModels(geminiModels);
  navigateTo("model");
  return geminiModels;
}
```

5. **Configuration Persistence**: Carefully updates global configuration with new model settings:

```
function saveConfiguration(provider: ProviderType, model: string) {
  const baseURL = providers[provider]?.baseURL || "";

  // Create a new config object based on the existing one
  const newConfig = { ...config };

  // Update the primary provider regardless of which model we're
changing
  newConfig.primaryProvider = provider;

  // Update the appropriate model based on the selection
  if (modelTypeToChange === "both" || modelTypeToChange === "large")
{
    newConfig.largeModelName = model;
    newConfig.largeModelBaseURL = baseURL;
    newConfig.largeModelApiKey = apiKey || config.largeModelApiKey;
    newConfig.largeModelMaxTokens = parseInt(maxTokens);

    // Save reasoning effort for large model if supported
    if (supportsReasoningEffort) {
      newConfig.largeModelReasoningEffort = reasoningEffort;
    } else {
      newConfig.largeModelReasoningEffort = undefined;
    }
  }

  // Similar handling for small model...

  // Save the updated configuration
  saveGlobalConfig(newConfig);
}
```

User Experience Benefits

The `model` command provides several important benefits for Claude Code users:

1. **Customization Control**: Gives users fine-grained control over the AI models powering their interaction.

2. **Cost Optimization**: Allows setting different models for different complexity tasks, optimizing for cost and speed.
3. **Provider Flexibility**: Enables users to choose from multiple AI providers based on preference, cost, or feature needs.
4. **Parameter Tuning**: Offers advanced users the ability to tune model parameters for optimal performance.
5. **Progressive Disclosure**: Uses a step-by-step flow that makes configuration accessible to both novice and advanced users.
6. **Intuitive Navigation**: Implements keyboard navigation with clear indicators for a smooth configuration experience.

The `model` command exemplifies Claude Code's approach to giving users control and flexibility while maintaining an accessible interface, keeping advanced configuration options available but not overwhelming.

cost Command

The `cost` command provides users with visibility into the cost and duration of their current Claude Code session, helping them monitor their API usage and expenses.

Implementation

The command is implemented in `commands/cost.ts` as a simple type: 'local' command that calls a formatting function:

```
import type { Command } from "../commands";
import { formatTotalCost } from "../cost-tracker";

const cost = {
  type: "local",
  name: "cost",
  description: "Show the total cost and duration of the current session",
  isEnabled: true,
  isHidden: false,
  async call() {
    return formatTotalCost();
  },
  userFacingName() {
    return "cost";
  },
} satisfies Command;

export default cost;
```

This command relies on the cost tracking system implemented in `cost-tracker.ts`, which maintains a running total of API costs and session duration.

Cost Tracking System

The cost tracking system is implemented in `cost-tracker.ts` and consists of several key components:

1. **State Management**: Maintains a simple singleton state object tracking:
   - `totalCost`: Running total of API costs in USD
   - `totalAPIDuration`: Cumulative time spent waiting for API responses
   - `startTime`: Timestamp when the session began
2. **Cost Accumulation**: Provides a function to add costs as they occur:

```
export function addToTotalCost(cost: number, duration: number): void
{
  STATE.totalCost += cost;
  STATE.totalAPIDuration += duration;
}
```

3. **Reporting**: Formats cost information in a human-readable format:

```
export function formatTotalCost(): string {
  return chalk.grey(
    `Total cost: ${formatCost(STATE.totalCost)}
    Total duration (API): ${formatDuration(STATE.totalAPIDuration)}
    Total duration (wall): ${formatDuration(getTotalDuration())}`
  );
}
```

4. **Persistence**: Uses a React hook to save session cost data when the process exits:

```
export function useCostSummary(): void {
  useEffect(() => {
    const f = () => {
      process.stdout.write("\n" + formatTotalCost() + "\n");

      // Save last cost and duration to project config
      const projectConfig = getCurrentProjectConfig();
      saveCurrentProjectConfig({
        ...projectConfig,
        lastCost: STATE.totalCost,
        lastAPIDuration: STATE.totalAPIDuration,
        lastDuration: getTotalDuration(),
        lastSessionId: SESSION_ID,
      });
    };
    process.on("exit", f);
    return () => {
      process.off("exit", f);
    };
  }, []);
}
```

UI Components

The cost tracking system is complemented by two UI components:

1. **Cost Component**: A simple display component used in the debug panel to show the most recent API call cost:

```
export function Cost({
  costUSD,
  durationMs,
  debug,
}: Props): React.ReactNode {
  if (!debug) {
    return null;
  }

  const durationInSeconds = (durationMs / 1000).toFixed(1);
  return (
    <Box flexDirection="column" minWidth={23} width={23}>
      <Text dimColor>
        Cost: ${costUSD.toFixed(4)} ({durationInSeconds}s)
      </Text>
    </Box>
  );
}
```

2. **CostThresholdDialog**: A warning dialog shown when users exceed a certain cost threshold:

```
export function CostThresholdDialog({ onDone }: Props):
React.ReactNode {
  // Handle Ctrl+C, Ctrl+D and Esc
  useInput((input, key) => {
    if ((key.ctrl && (input === "c" || input === "d")) ||
key.escape) {
      onDone();
    }
  });

  return (
    <Box
      flexDirection="column"
      borderStyle="round"
      padding={1}
      borderColor={getTheme().secondaryBorder}
    >
      <Box marginBottom={1} flexDirection="column">
        <Text bold>You've spent $5 on the Anthropic API this
session.</Text>
        <Text>Learn more about how to monitor your spending:</Text>
        <Link url="https://docs.anthropic.com/s/claude-code-cost" />
      </Box>
      <Box>
        <Select
          options={[
            {
              value: "ok",
              label: "Got it, thanks!",
            },
          ]}
          onChange={onDone}
        />
      </Box>
    </Box>
  );
}
```

Technical Implementation Notes

The cost tracking system demonstrates several design considerations:

1. **Singleton State**: Uses a single state object with a clear comment warning against adding more state.
2. **Persistence Across Sessions**: Saves cost data to the project configuration, allowing for tracking across sessions.
3. **Formatting Flexibility**: Uses different decimal precision based on the cost amount (4 decimal places for small amounts, 2 for larger ones).
4. **Multiple Time Metrics**: Tracks both wall clock time and API request time separately.
5. **Environment-Aware Testing**: Includes a reset function that's only available in test environments.
6. **Exit Hooks**: Uses process exit hooks to ensure cost data is saved and displayed even if the application exits unexpectedly.

User Experience Considerations

The cost tracking system addresses several user needs:

1. **Transparency**: Provides clear visibility into API usage costs.
2. **Usage Monitoring**: Helps users track and manage their API spending.
3. **Efficiency Insights**: Shows both total runtime and API time, helping identify bottlenecks.
4. **Threshold Warnings**: Alerts users when they've spent significant amounts.
5. **Documentation Links**: Provides resources for learning more about cost management.

The `/cost` command and associated systems represent Claude Code's approach to transparent cost management, giving users control over their API usage while maintaining a simple, unobtrusive interface.

onboarding Command

The `onboarding` command provides a guided first-run experience for new users, helping them configure Claude Code to their preferences and introducing them to the tool's capabilities.

Implementation

The command is implemented in `commands/onboarding.tsx` as a type: 'local-jsx' command that renders a React component:

```
import * as React from "react";
import type { Command } from "../commands";
import { Onboarding } from "../components/Onboarding";
import { clearTerminal } from "../utils/terminal";
import { getGlobalConfig, saveGlobalConfig } from "../utils/config";
import { clearConversation } from "./clear";

export default {
  type: "local-jsx",
  name: "onboarding",
  description: "[ANT-ONLY] Run through the onboarding flow",
  isEnabled: true,
  isHidden: false,
  async call(onDone, context) {
    await clearTerminal();
    const config = getGlobalConfig();
    saveGlobalConfig({
      ...config,
      theme: "dark",
    });

    return (
      <Onboarding
        onDone={async () => {
          clearConversation(context);
          onDone();
        }}
      />
    );
  },
  userFacingName() {
    return "onboarding";
  },
} satisfies Command;
```

The command delegates to the `Onboarding` component in `components/Onboarding.tsx`, which handles the multi-step onboarding flow.

Functionality

The `onboarding` command implements a comprehensive first-run experience:

1. **Multi-Step Flow**:
    - Walks users through a series of configuration steps with a smooth, guided experience
    - Includes theme selection, usage guidance, and model selection
    - Uses a stack-based navigation system for intuitive flow between steps
2. **Theme Configuration**:
    - Allows users to choose between light and dark themes
    - Includes colorblind-friendly theme options for accessibility
    - Provides a live preview of the selected theme using a code diff example
3. **Usage Guidelines**:
    - Introduces users to effective ways of using Claude Code
    - Explains how to provide clear context and work with the tool
    - Sets appropriate expectations for the tool's capabilities
4. **Model Selection**:
    - Guides users through configuring AI provider and model settings
    - Uses the `ModelSelector` component for a consistent model selection experience
    - Allows configuration of both small and large models for different tasks
5. **Configuration Persistence**:
    - Saves user preferences to the global configuration
    - Marks onboarding as complete to prevent repeat runs
    - Clears the conversation after onboarding to provide a clean start

Technical Implementation Notes

The `onboarding` command demonstrates several sophisticated patterns:

1. **Screen Navigation Stack**: Implements a stack-based navigation system for multi-step flow:

```
const [screenStack, setScreenStack] = useState<
  Array<
    | "modelType"
    | "provider"
    | "apiKey"
    | "model"
    | "modelParams"
    | "confirmation"
  >
>(["modelType"]);

// Current screen is always the last item in the stack
const currentScreen = screenStack[screenStack.length - 1];

// Function to navigate to a new screen
const navigateTo = (screen) => {
  setScreenStack((prev) => [...prev, screen]);
};

// Function to go back to the previous screen
const goBack = () => {
  if (screenStack.length > 1) {
    setScreenStack((prev) => prev.slice(0, -1));
  } else {
    onDone();
  }
};
```

2. **Progressive Disclosure**: Presents information in digestible chunks across multiple steps.

3. **Terminal UI Adaptation**: Uses Ink components optimized for terminal rendering:

```
<Box flexDirection="column" gap={1} paddingLeft={1}>
  <Text bold>Using {PRODUCT_NAME} effectively:</Text>
  <Box flexDirection="column" width={70}>
    <OrderedList>
      <OrderedList.Item>
        <Text>
          Start in your project directory
          <Newline />
          <Text color={theme.secondaryText}>
            Files are automatically added to context when needed.
          </Text>
          <Newline />
        </Text>
      </OrderedList.Item>
      {/* Additional list items */}
    </OrderedList>
  </Box>
  <PressEnterToContinue />
</Box>
```

4. **Interactive Components**: Uses custom select components for theme selection with previews:

```
<Select
  options={[
    { label: "Light text", value: "dark" },
    { label: "Dark text", value: "light" },
    {
      label: "Light text (colorblind-friendly)",
      value: "dark-daltonized",
    },
    {
      label: "Dark text (colorblind-friendly)",
      value: "light-daltonized",
    },
  ]}
  onFocus={handleThemePreview}
  onChange={handleThemeSelection}
/>
```

5. **Exit Handling**: Implements `useExitOnCtrlCD` to provide users with a clear way to exit the flow:

```
const exitState = useExitOnCtrlCD(() => process.exit(0));
```

6. **Conditional Rendering**: Uses state to conditionally show different screens:

```
// If we're showing the model selector screen, render it directly
if (showModelSelector) {
  return <ModelSelector onDone={handleModelSelectionDone} />;
}
```

User Experience Benefits

The `onboarding` command addresses several key needs for new users:

1. **Guided Setup**: Provides a structured introduction to Claude Code rather than dropping users into a blank interface.

2. **Preference Customization**: Allows users to set their preferences immediately, increasing comfort with the tool.
3. **Learning Opportunity**: Teaches best practices for using the tool effectively from the start.
4. **Accessibility Awareness**: Explicitly offers colorblind-friendly themes, demonstrating attention to accessibility.
5. **Progressive Complexity**: Introduces features gradually, avoiding overwhelming new users.

The `onboarding` command exemplifies Claude Code's attention to user experience, ensuring new users can quickly set up the tool according to their preferences and learn how to use it effectively from the beginning.

pr_comments Command

The `pr_comments` command provides a specialized interface for retrieving and displaying GitHub pull request comments, helping users review feedback on their code without leaving the terminal.

Implementation

The command is implemented in `commands/pr_comments.ts` as a type: 'prompt' command that formulates a specialized request to Claude:

```
import { Command } from "../commands";

export default {
  type: "prompt",
  name: "pr-comments",
  description: "Get comments from a GitHub pull request",
  progressMessage: "fetching PR comments",
  isEnabled: true,
  isHidden: false,
  userFacingName() {
    return "pr-comments";
  },
  async getPromptForCommand(args: string) {
```

```
    return [
      {
        role: "user",
        content: [
          {
            type: "text",
            text: `You are an AI assistant integrated into a git-based
version control system. Your task is to fetch and display comments from a
GitHub pull request.

Follow these steps:

1. Use \`gh pr view --json number,headRepository\` to get the PR number
and repository info
2. Use \`gh api /repos/{owner}/{repo}/issues/{number}/comments\` to get
PR-level comments
3. Use \`gh api /repos/{owner}/{repo}/pulls/{number}/comments\` to get
review comments. Pay particular attention to the following fields:
\`body\`, \`diff_hunk\`, \`path\`, \`line\`, etc. If the comment
references some code, consider fetching it using eg \`gh api
/repos/{owner}/{repo}/contents/{path}?ref={branch} | jq .content -r |
base64 -d\`
4. Parse and format all comments in a readable way
5. Return ONLY the formatted comments, with no additional text

Format the comments as:

## Comments

[For each comment thread:]
- @author file.ts#line:
  \`\`\`diff
  [diff_hunk from the API response]
  \`\`\`
  > quoted comment text

  [any replies indented]

If there are no comments, return "No comments found."

Remember:
1. Only show the actual comments, no explanatory text
2. Include both PR-level and code review comments
3. Preserve the threading/nesting of comment replies
4. Show the file and line number context for code review comments
5. Use jq to parse the JSON responses from the GitHub API

${args ? "Additional user input: " + args : ""}
`,
          },
        ],
      },
    ];
  },
```

```
} satisfies Command;
```

Unlike commands that directly render UI components, the `pr_comments` command is a 'prompt' type command that formulates a specific request to Claude, instructing it to perform a complex sequence of operations using the GitHub CLI and API.

Functionality

The `pr_comments` command provides several key capabilities:

1. **GitHub Integration**:
    - Uses the GitHub CLI (`gh`) to interact with GitHub's API
    - Retrieves both PR-level comments (issue comments) and review comments (inline code comments)
    - Handles authentication and API access through the existing GitHub CLI configuration
2. **Comment Retrieval**:
    - Fetches PR metadata to determine repository and PR information
    - Makes API calls to retrieve different types of comments
    - Handles pagination and JSON parsing using `jq` utility
3. **Context Preservation**:
    - Retrieves code context for review comments using the `diff_hunk` field
    - Shows file paths and line numbers for specific comments
    - Preserves the hierarchical structure of comment threads and replies
4. **Formatted Output**:
    - Creates a well-structured, readable display of comments
    - Uses Markdown formatting for readability
    - Shows comments in a hierarchical, threaded view
5. **Progress Indication**:
    - Shows a progress message ("fetching PR comments") while the operation is in progress
    - Provides clear indication when no comments are found

Technical Implementation Notes

The `pr_comments` command demonstrates several sophisticated approaches:

1. **Prompt-Based Implementation**: Unlike UI commands, this command uses Claude itself to execute a complex sequence of operations through a carefully crafted prompt.

2. **GitHub CLI Utilization**: Leverages the GitHub CLI's capabilities to interact with GitHub's API, taking advantage of existing authentication and configuration.

3. **API Interaction Patterns**: Provides a detailed workflow for accessing and processing data from different GitHub API endpoints:

   ```
   gh api /repos/{owner}/{repo}/issues/{number}/comments
   gh api /repos/{owner}/{repo}/pulls/{number}/comments
   ```

4. **JSON Processing with JQ**: Uses the `jq` command-line JSON processor for parsing complex API responses.

5. **Complex Data Formatting**: Provides explicit formatting instructions to ensure consistent, readable output:

   ```
   - @author file.ts#line:
     ```diff
     [diff_hunk from the API response]
   ```

   > quoted comment text

   [any replies indented]

   ```
   ```
   ```

6. **Arguments Passthrough**: Allows users to provide additional arguments that are appended to the prompt, enabling refinement of the command's behavior.

User Experience Benefits

The `pr_comments` command addresses several important needs for developers:

1. **Context Switching Reduction**: Allows reviewing PR comments without leaving the terminal or switching to a browser.
2. **Comment Aggregation**: Brings together comments from different locations (PR-level and code-specific) in a single view.
3. **Reading Optimization**: Formats comments with proper context, improving readability compared to raw API responses.
4. **Workflow Integration**: Enables PR review activities to be part of the normal development workflow in the terminal.
5. **GitHub Integration**: Takes advantage of existing GitHub CLI authentication and configuration.

The `pr_comments` command exemplifies how Claude Code can leverage Claude's capabilities to implement complex workflows that would otherwise require significant custom code. By using a prompt-based approach, it achieves powerful GitHub integration with minimal implementation complexity.

release-notes Command

The `release-notes` command provides users with a simple way to view the changes and new features introduced in each version of Claude Code, helping them stay informed about updates.

Implementation

The command is implemented in `commands/release-notes.ts` as a type: 'local' command that formats and displays release notes from a constant:

```
import { MACRO } from "../constants/macros.js";
import type { Command } from "../commands";
import { RELEASE_NOTES } from "../constants/releaseNotes";

const releaseNotes: Command = {
  description: "Show release notes for the current or specified version",
  isEnabled: false,
  isHidden: false,
  name: "release-notes",
  userFacingName() {
    return "release-notes";
  },
  type: "local",
  async call(args) {
    const currentVersion = MACRO.VERSION;

    // If a specific version is requested, show that version's notes
    const requestedVersion = args ? args.trim() : currentVersion;

    // Get the requested version's notes
    const notes = RELEASE_NOTES[requestedVersion];

    if (!notes || notes.length === 0) {
      return `No release notes available for version
${requestedVersion}.`;
    }

    const header = `Release notes for version ${requestedVersion}:`;
    const formattedNotes = notes.map((note) => `• ${note}`).join("\n");

    return `${header}\n\n${formattedNotes}`;
  },
};

export default releaseNotes;
```

This command demonstrates a simple but effective approach to displaying versioned information, using a constant defined in `constants/releaseNotes.ts` as the data source.

Functionality

The `release-notes` command provides a straightforward information display:

1. **Version Selection**:
    - Defaults to showing notes for the current version

    - Allows explicit version selection through command arguments
    - Handles cases where notes are unavailable for a version

2. **Note Formatting**:
    - Displays a clear header indicating the version
    - Formats individual notes as bullet points
    - Presents notes in a readable, consistent format

3. **No Dependencies**:
    - Doesn't require external API calls or complex processing
    - Works entirely with data embedded in the application

4. **Error Handling**:
    - Provides a friendly message when notes aren't available
    - Gracefully handles empty or missing note arrays

Technical Implementation Notes

While simpler than many other commands, the `release-notes` command demonstrates several effective patterns:

1. **Constant-Based Data Store**: Uses a predefined constant for storing release notes data:

```
// Example from constants/releaseNotes.ts
export const RELEASE_NOTES: Record<string, string[]> = {
  "0.0.22": [
    "Added proxy configuration support via --proxy flag or
HTTP_PROXY environment variable",
    "Improved error handling for API connectivity issues",
    "Fixed issue with terminal output formatting in certain
environments",
  ],
  "0.0.21": [
    "Enhanced model selection interface with provider-specific
options",
    "Improved documentation and help text for common commands",
    "Fixed bug with conversation state persistence",
  ],
};
```

2. **Default Value Pattern**: Uses the current version as a default if no specific version is requested:

```
const currentVersion = MACRO.VERSION;
const requestedVersion = args ? args.trim() : currentVersion;
```

3. **Array Transformation**: Uses map and join for clean formatting of bullet points:

```
const formattedNotes = notes.map((note) => `• ${note}`).join("\n");
```

4. **Null-Safe Access**: Checks for undefined or empty notes before attempting to display them:

```
if (!notes || notes.length === 0) {
  return `No release notes available for version
${requestedVersion}.`;
}
```

5. **Command Availability Control**: The command is marked as disabled with `isEnabled: false`, suggesting it may be under development or available only in certain builds.

User Experience Benefits

Despite its simplicity, the `release-notes` command provides several valuable benefits:

1. **Update Awareness**: Helps users understand what has changed between versions.
2. **Feature Discovery**: Introduces users to new capabilities they might not otherwise notice.
3. **Version Verification**: Allows users to confirm which version they're currently using.
4. **Historical Context**: Provides access to past release information for reference.
5. **Simplified Format**: Presents notes in a clean, readable format without requiring browser access.

The `release-notes` command exemplifies how even simple commands can provide valuable functionality through clear organization and presentation of information. It serves as part of Claude Code's approach to transparency and user communication.

resume Command

The `resume` command allows users to continue previous conversations with Claude, providing a way to revisit and extend past interactions without losing context.

Implementation

The command is implemented in `commands/resume.tsx` as a type: 'local-jsx' command that renders a React component for selecting and resuming past conversations:

```
import * as React from "react";
import type { Command } from "../commands";
import { ResumeConversation } from "../screens/ResumeConversation";
import { render } from "ink";
import { CACHE_PATHS, loadLogList } from "../utils/log";

export default {
  type: "local-jsx",
  name: "resume",
  description: "[ANT-ONLY] Resume a previous conversation",
  isEnabled: true,
  isHidden: false,
  userFacingName() {
    return "resume";
  },
  async call(onDone, { options: { commands, tools, verbose } }) {
    const logs = await loadLogList(CACHE_PATHS.messages());
    render(
      <ResumeConversation
        commands={commands}
        context={{ unmount: onDone }}
        logs={logs}
        tools={tools}
        verbose={verbose}
      />
    );
    // This return is here for type only
    return null;
  },
} satisfies Command;
```

The command delegates to the `ResumeConversation` component in `screens/ResumeConversation.tsx`, passing the list of previous conversations and necessary context.

Functionality

The `resume` command implements a sophisticated conversation history management system:

1. **Conversation Retrieval**:
    - Loads previously saved conversation logs from the cache directory
    - Uses the `loadLogList` utility to parse and organize log files
    - Presents a navigable list of past conversations
2. **Context Restoration**:
    - Allows users to select a specific past conversation to resume
    - Loads the complete message history for the selected conversation
    - Reinstates the context of the previous interaction
3. **Selection Interface**:
    - Provides a visual interface for browsing conversation history
    - Shows metadata about each conversation (date, duration, topic)
    - Enables keyboard navigation through the history list
4. **Seamless Transition**:
    - Integrates resumed conversations into the current CLI session
    - Passes necessary tooling and command context to the resumed conversation
    - Maintains configuration settings across resumed sessions

Technical Implementation Notes

The `resume` command demonstrates several sophisticated implementation patterns:

1. **Custom Rendering Approach**: Unlike most commands that return a React element, this command uses a direct render call:

```
render(
  <ResumeConversation
    commands={commands}
    context={{ unmount: onDone }}
    logs={logs}
    tools={tools}
    verbose={verbose}
  />
);
```

This approach gives it more direct control over the rendering lifecycle.

2. **Log Loading and Parsing**: Implements careful log handling with the loadLogList utility:

```
// From utils/log.ts
export async function loadLogList(directory: string):
Promise<LogFile[]> {
  // Reads log directory
  // Parses log files
  // Extracts metadata
  // Sorts by recency
  // Returns formatted log list
}
```

3. **Context Passing**: Passes the full command and tool context to ensure the resumed conversation has access to all capabilities:

```
<ResumeConversation
  commands={commands}
  context={{ unmount: onDone }}
  logs={logs}
  tools={tools}
  verbose={verbose}
/>
```

4. **Path Management**: Uses a centralized path management system for log files:

```
// From utils/log.ts
export const CACHE_PATHS = {
  messages: () => path.join(getConfigRoot(), "messages"),
  // Other path definitions...
};
```

5. **ANT-ONLY Flag**: Uses the `[ANT-ONLY]` prefix in the description, indicating it's a

feature specific to internal Anthropic usage and potentially not available in all distributions.

User Experience Benefits

The `resume` command addresses several important user needs:

1. **Conversation Continuity**: Allows users to pick up where they left off in previous sessions.
2. **Context Preservation**: Maintains the full context of previous interactions, reducing repetition.
3. **Work Session Management**: Enables users to organize work across multiple sessions.
4. **History Access**: Provides a browsable interface to previous conversations.
5. **Interrupted Work Recovery**: Helps recover from interrupted work sessions or system crashes.

The `resume` command exemplifies Claude Code's approach to persistent user experience, ensuring that valuable conversation context isn't lost between sessions. This is particularly important for complex coding tasks that may span multiple work sessions.

review Command

The `review` command provides a specialized workflow for reviewing GitHub pull requests, leveraging Claude's code analysis capabilities to generate comprehensive code reviews.

Implementation

The command is implemented in `commands/review.ts` as a type: 'prompt' command that formulates a specific request to Claude:

```
import { Command } from "../commands";
import { BashTool } from "../tools/BashTool/BashTool";

export default {
  type: "prompt",
  name: "review",
  description: "Review a pull request",
  isEnabled: true,
  isHidden: false,
  progressMessage: "reviewing pull request",
  userFacingName() {
    return "review";
  },
  async getPromptForCommand(args) {
    return [
      {
        role: "user",
        content: [
          {
            type: "text",
            text: `
      You are an expert code reviewer. Follow these steps:

      1. If no PR number is provided in the args, use ${BashTool.name}
("gh pr list") to show open PRs
      2. If a PR number is provided, use ${BashTool.name}("gh pr view
<number>") to get PR details
      3. Use ${BashTool.name}("gh pr diff <number>") to get the diff
      4. Analyze the changes and provide a thorough code review that
includes:
         - Overview of what the PR does
         - Analysis of code quality and style
         - Specific suggestions for improvements
         - Any potential issues or risks

      Keep your review concise but thorough. Focus on:
      - Code correctness
      - Following project conventions
      - Performance implications
      - Test coverage
      - Security considerations

      Format your review with clear sections and bullet points.

      PR number: ${args}
    `,
          },
        ],
      },
    ];
  },
} satisfies Command;
```

Like the `pr_comments` command, the `review` command is a 'prompt' type command that formulates a specific request to Claude, instructing it to perform a complex sequence of operations using the GitHub CLI.

Functionality

The `review` command implements a comprehensive PR review workflow:

1. **PR Discovery and Selection**:
    - Lists open pull requests if no PR number is specified
    - Retrieves details for a specific PR when a number is provided
    - Presents options for users to select which PR to review
2. **Code Analysis**:
    - Fetches the PR diff to understand code changes
    - Analyzes code quality, style, and potential issues
    - Evaluates changes in the context of the project's conventions
3. **Review Generation**:
    - Creates a structured code review with logical sections
    - Highlights potential improvements and issues
    - Provides specific, actionable feedback on code changes
4. **Focus Areas**:
    - Analyzes code correctness and logical issues
    - Evaluates adherence to project conventions
    - Assesses performance implications of changes
    - Reviews test coverage of new code
    - Identifies potential security considerations
5. **Progress Indication**:
    - Shows a "reviewing pull request" message during analysis
    - Provides a clear visual indication that processing is underway

Technical Implementation Notes

The `review` command demonstrates several important implementation patterns:

1. **Tool Reference in Prompt**: Explicitly references the BashTool by name in the prompt to ensure proper tool usage:

   ```
   use ${BashTool.name}("gh pr list")
   ```

   This ensures the correct tool is used regardless of how Claude might interpret the instruction.

2. **GitHub CLI Integration**: Leverages the GitHub CLI's capabilities for PR interaction:

   ```
   gh pr list - Show open PRs
   gh pr view <number> - Get PR details
   gh pr diff <number> - Get the diff
   ```

3. **Comprehensive Analysis Instructions**: Provides Claude with a detailed framework for analysis:

   ```
   Analyze the changes and provide a thorough code review that
   includes:
   - Overview of what the PR does
   - Analysis of code quality and style
   - Specific suggestions for improvements
   - Any potential issues or risks
   ```

4. **Focus Guidance**: Directs Claude's analysis toward specific aspects of code quality:

```
Focus on:
- Code correctness
- Following project conventions
- Performance implications
- Test coverage
- Security considerations
```

5. **Formatting Guidance**: Ensures consistent output format with clear instructions:

```
Format your review with clear sections and bullet points.
```

User Experience Benefits

The `review` command provides several valuable benefits for developers:

1. **Code Review Automation**: Reduces the manual effort required for initial PR reviews.
2. **Consistent Review Quality**: Ensures all PRs receive a thorough analysis covering key areas.
3. **Learning Opportunity**: Exposes developers to alternative perspectives on their code.
4. **Workflow Integration**: Fits seamlessly into GitHub-based development workflows.
5. **Time Efficiency**: Quickly generates comprehensive reviews that serve as a starting point for human reviewers.

The `review` command exemplifies how Claude Code can leverage Claude's code analysis capabilities to add value to existing development workflows. It transforms Claude from a conversational assistant into an active participant in the code review process.

terminalSetup Command

The `terminalSetup` command enhances the user experience by configuring terminal-specific keyboard shortcuts, specifically implementing Shift+Enter for newlines in terminals like iTerm2 and VS Code.

Implementation

The command is implemented in `commands/terminalSetup.ts` as a type: 'local' command that configures terminal-specific settings:

```
import { Command } from "../commands";
import { EOL, platform, homedir } from "os";
import { execFileNoThrow } from "../utils/execFileNoThrow";
import chalk from "chalk";
import { getTheme } from "../utils/theme";
import { env } from "../utils/env";
import { getGlobalConfig, saveGlobalConfig } from "../utils/config";
import { markProjectOnboardingComplete } from "../ProjectOnboarding";
import { readFileSync, writeFileSync } from "fs";
import { join } from "path";
import { safeParseJSON } from "../utils/json";
import { logError } from "../utils/log";

const terminalSetup: Command = {
  type: "local",
  name: "terminal-setup",
  userFacingName() {
    return "terminal-setup";
  },
  description:
    "Install Shift+Enter key binding for newlines (iTerm2 and VSCode
only)",
  isEnabled:
    (platform() === "darwin" && env.terminal === "iTerm.app") ||
    env.terminal === "vscode",
  isHidden: false,
  async call() {
    let result = "";

    switch (env.terminal) {
      case "iTerm.app":
        result = await installBindingsForITerm2();
        break;
      case "vscode":
        result = installBindingsForVSCodeTerminal();
        break;
```

```
    }

    // Update global config to indicate Shift+Enter key binding is
installed
    const config = getGlobalConfig();
    config.shiftEnterKeyBindingInstalled = true;
    saveGlobalConfig(config);

    // Mark onboarding as complete
    markProjectOnboardingComplete();

    return result;
  },
};

export function isShiftEnterKeyBindingInstalled(): boolean {
  return getGlobalConfig().shiftEnterKeyBindingInstalled === true;
}

export default terminalSetup;
```

The command includes specialized implementation functions for different terminal types, with unique approaches for iTerm2 and VS Code.

Functionality

The `terminalSetup` command provides a targeted terminal enhancement:

1. **Terminal Detection**:
    - Determines the current terminal environment automatically
    - Supports iTerm2 on macOS and VS Code's integrated terminal
    - Only enables the command when in compatible terminals
2. **Keyboard Shortcut Configuration**:
    - Configures Shift+Enter to insert a newline without submitting input
    - Uses terminal-specific mechanisms for each supported terminal
    - For iTerm2, uses the defaults command to modify keybindings
    - For VS Code, modifies the keybindings.json configuration file
3. **Configuration Status Tracking**:
    - Updates global configuration to track installation state
    - Provides a utility function to check if bindings are installed

- Avoids redundant installations by checking state

4. **Onboarding Integration**:
   - Marks project onboarding as complete after setup
   - Integrates with the broader onboarding workflow
   - Provides a smooth setup experience for new users

Technical Implementation Notes

The `terminalSetup` command demonstrates several sophisticated technical approaches:

1. **Platform-Specific Implementations**: Implements different strategies based on the terminal type:

```
switch (env.terminal) {
  case "iTerm.app":
    result = await installBindingsForITerm2();
    break;
  case "vscode":
    result = installBindingsForVSCodeTerminal();
    break;
}
```

2. **macOS Defaults System**: For iTerm2, uses the macOS defaults system to modify key bindings:

```
async function installBindingsForITerm2(): Promise<string> {
  const { code } = await execFileNoThrow("defaults", [
    "write",
    "com.googlecode.iterm2",
    "GlobalKeyMap",
    "-dict-add",
    "0xd-0x20000-0x24",
    `<dict>
      <key>Text</key>
      <string>\\n</string>
      <key>Action</key>
      <integer>12</integer>
      <key>Version</key>
      <integer>1</integer>
      <key>Keycode</key>
      <integer>13</integer>
      <key>Modifiers</key>
      <integer>131072</integer>
    </dict>`,
  ]);
  // Error handling and success message...
}
```

3. **VS Code Configuration File Handling**: For VS Code, directly modifies the keybindings.json file:

```
function installBindingsForVSCodeTerminal(): string {
  const vscodeKeybindingsPath = join(
    homedir(),
    platform() === "win32"
      ? join("AppData", "Roaming", "Code", "User")
      : platform() === "darwin"
      ? join("Library", "Application Support", "Code", "User")
      : join(".config", "Code", "User"),
    "keybindings.json"
  );

  try {
    const content = readFileSync(vscodeKeybindingsPath, "utf-8");
    const keybindings: VSCodeKeybinding[] =
      (safeParseJSON(content) as VSCodeKeybinding[]) ?? [];

    // Check for existing bindings...
    // Add new binding...
    // Write updated file...
  } catch (e) {
    // Error handling...
  }
}
```

4. **Cross-Platform Path Handling**: Uses the path.join utility along with platform detection to handle OS-specific paths:

```
const vscodeKeybindingsPath = join(
  homedir(),
  platform() === "win32"
    ? join("AppData", "Roaming", "Code", "User")
    : platform() === "darwin"
    ? join("Library", "Application Support", "Code", "User")
    : join(".config", "Code", "User"),
  "keybindings.json"
);
```

5. **Safe JSON Parsing**: Uses a utility function for safe JSON parsing to handle potential errors:

```
const keybindings: VSCodeKeybinding[] =
  (safeParseJSON(content) as VSCodeKeybinding[]) ?? [];
```

User Experience Benefits

The `terminalSetup` command addresses a specific pain point in terminal interaction:

1. **Improved Input Experience**: Allows multi-line input with Shift+Enter without submitting prematurely.
2. **Workflow Enhancement**: Makes it easier to compose complex prompts or code snippets.
3. **Consistency Across Environments**: Provides similar behavior in different terminal environments.
4. **Seamless Integration**: Configures the terminal without requiring users to understand terminal-specific configuration files.
5. **Visual Feedback**: Provides clear success messages when binding installation completes.

The `terminalSetup` command exemplifies Claude Code's attention to detail in the user experience, addressing a specific friction point in terminal interaction to create a more seamless interaction pattern for complex inputs.

compact Command

The `compact` command offers a sophisticated solution to context management by summarizing the conversation history before clearing it, thereby retaining essential context while freeing up token space.

Implementation

The command is implemented in `commands/compact.ts` as a type: 'local' command:

```
import { Command } from "../commands";
import { getContext } from "../context";
import { getMessagesGetter, getMessagesSetter } from "../messages";
import { API_ERROR_MESSAGE_PREFIX, querySonnet } from
"../services/claude";
import {
  createUserMessage,
  normalizeMessagesForAPI,
} from "../utils/messages.js";
import { getCodeStyle } from "../utils/style";
import { clearTerminal } from "../utils/terminal";

const compact = {
  type: "local",
  name: "compact",
  description: "Clear conversation history but keep a summary in
context",
  isEnabled: true,
  isHidden: false,
  async call(
    _,
    {
      options: { tools, slowAndCapableModel },
      abortController,
      setForkConvoWithMessagesOnTheNextRender,
    }
  ) {
    // Get existing messages before clearing
    const messages = getMessagesGetter()();

    // Add summary request as a new message
    const summaryRequest = createUserMessage(
      "Provide a detailed but concise summary of our conversation above.
Focus on information that would be helpful for continuing the
conversation, including what we did, what we're doing, which files we're
working on, and what we're going to do next."
    );
```

```
    const summaryResponse = await querySonnet(
      normalizeMessagesForAPI([...messages, summaryRequest]),
      ["You are a helpful AI assistant tasked with summarizing
conversations."],
      0,
      tools,
      abortController.signal,
      {
        dangerouslySkipPermissions: false,
        model: slowAndCapableModel,
        prependCLISysprompt: true,
      }
    );

    // Extract summary from response, throw if we can't get it
    const content = summaryResponse.message.content;
    const summary =
      typeof content === "string"
        ? content
        : content.length > 0 && content[0]?.type === "text"
        ? content[0].text
        : null;

    if (!summary) {
      throw new Error(
        `Failed to generate conversation summary - response did not
contain valid text content - ${summaryResponse}`
      );
    } else if (summary.startsWith(API_ERROR_MESSAGE_PREFIX)) {
      throw new Error(summary);
    }

    // Substitute low token usage info so that the context-size UI
warning goes
    // away. The actual numbers don't matter too much: `countTokens`
checks the
    // most recent assistant message for usage numbers, so this estimate
will
    // be overridden quickly.
    summaryResponse.message.usage = {
      input_tokens: 0,
      output_tokens: summaryResponse.message.usage.output_tokens,
      cache_creation_input_tokens: 0,
      cache_read_input_tokens: 0,
    };

    // Clear screen and messages
    await clearTerminal();
    getMessagesSetter()([]);
    setForkConvoWithMessagesOnTheNextRender([
      createUserMessage(
        `Use the /compact command to clear the conversation history, and
start a new conversation with the summary in context.`
      ),
```

```
        summaryResponse,
      ]);
      getContext.cache.clear?.();
      getCodeStyle.cache.clear?.();

      return ""; // not used, just for typesafety. TODO: avoid this hack
    },
    userFacingName() {
      return "compact";
    },
  } satisfies Command;

  export default compact;
```

Functionality

The `compact` command implements a sophisticated workflow:

1. **Conversation Summary Generation**:
   - Retrieves the current message history
   - Creates a user message requesting a conversation summary
   - Uses Claude API to generate a summary through the `querySonnet` function
   - Validates the summary to ensure it was successfully generated
2. **Token Usage Management**:
   - Manipulates the usage data to prevent context-size warnings
   - Sets input tokens to 0 to indicate the conversation has been compacted
3. **Context Reset with Summary**:
   - Clears the terminal display
   - Resets message history
   - Creates a new conversation "fork" containing only:
     - A user message indicating a compact operation occurred
     - The generated summary response
   - Clears context and code style caches

Technical Implementation Notes

The `compact` command demonstrates several advanced patterns:

1. **Meta-Conversation**: The command uses Claude to talk about the conversation

itself, leveraging the model's summarization abilities.

2. **Model Selection**: Explicitly uses the `slowAndCapableModel` option to ensure high-quality summarization.
3. **Content Extraction Logic**: Implements robust parsing of the response content, handling different content formats (string vs. structured content).
4. **Error Handling**: Provides clear error messages for when summarization fails or when the API returns an error.
5. **Token Manipulation**: Intelligently manipulates token usage information to maintain a good user experience after compaction.
6. **Conversation Forking**: Uses the `setForkConvoWithMessagesOnTheNextRender` mechanism to create a new conversation branch with only the summary.

User Experience Benefits

The `compact` command addresses several key pain points in AI assistant interactions:

1. **Context Window Management**: Helps users stay within token limits while preserving the essence of the conversation.
2. **Conversation Continuity**: Unlike a complete clear, it maintains the thread of discussion through the summary.
3. **Work Session Persistence**: Preserves information about files being edited and tasks in progress.
4. **Smart Reset**: Performs a targeted reset that balances clearing space with maintaining context.

The command is particularly valuable for long development sessions where context limits become an issue but completely starting over would lose important progress information.

config Command

The `config` command provides an interactive terminal interface for viewing and editing Claude Code's configuration settings, including model settings, UI preferences, and API keys.

Implementation

The command is implemented in `commands/config.tsx` as a type: 'local-jsx' command that renders a React component:

```
import { Command } from "../commands";
import { Config } from "../components/Config";
import * as React from "react";

const config = {
  type: "local-jsx",
  name: "config",
  description: "Open config panel",
  isEnabled: true,
  isHidden: false,
  async call(onDone) {
    return <Config onClose={onDone} />;
  },
  userFacingName() {
    return "config";
  },
} satisfies Command;

export default config;
```

Like the `bug` command, this command uses JSX to render an interactive UI component. The actual functionality is implemented in the `Config` component located in `components/Config.tsx` .

UI Component

The `Config` component implements a rich terminal-based settings interface with the following features:

1. **Settings Management**: Displays a list of configurable settings with their current

values.

2. **Multiple Setting Types**: Supports various setting types:
   - `boolean` : Toggle settings (true/false)
   - `enum` : Options from a predefined list
   - `string` : Text input values
   - `number` : Numeric values
3. **Interactive Editing**: Allows users to:
   - Navigate settings with arrow keys
   - Toggle boolean and enum settings with Enter/Space
   - Edit string and number settings with a text input mode
   - Exit configuration with Escape
4. **Configuration Persistence**: Saves settings to a configuration file using `saveGlobalConfig` .

Configuration Options

The component exposes numerous configuration options, including:

1. **Model Configuration**:
   - AI Provider selection (anthropic, openai, custom)
   - API keys for small and large models
   - Model names for both small and large models
   - Base URLs for API endpoints
   - Max token settings
   - Reasoning effort levels
2. **User Interface**:
   - Theme selection (light, dark, light-daltonized, dark-daltonized)
   - Verbose output toggle
3. **System Settings**:
   - Notification preferences
   - HTTP proxy configuration

Technical Implementation Notes

The `Config` component demonstrates several advanced patterns:

1. **State Management**: Uses React's `useState` to track:
    - Current configuration state
    - Selected setting index
    - Editing mode state
    - Current input text
    - Input validation errors
2. **Reference Comparison**: Maintains a reference to the initial configuration using `useRef` to track changes.
3. **Keyboard Input Handling**: Implements sophisticated keyboard handling for navigation and editing:
    - Arrow keys for selection
    - Enter/Space for toggling/editing
    - Escape for cancellation
    - Input handling with proper validation
4. **Input Sanitization**: Cleans input text to prevent control characters and other problematic input.
5. **Visual Feedback**: Provides clear visual indication of:
    - Currently selected item
    - Editing state
    - Input errors
    - Available actions
6. **Change Tracking**: Tracks and logs configuration changes when exiting.

User Experience Design

The config component showcases several UI/UX design principles for terminal applications:

1. **Modal Interface**: Creates a focused settings panel that temporarily takes over

the terminal.

2. **Progressive Disclosure**: Shows relevant controls and options based on the current state.
3. **Clear Instructions**: Displays context-sensitive help text at the bottom of the interface.
4. **Visual Highlighting**: Uses color and indicators to show the current selection and editing state.
5. **Immediate Feedback**: Changes take effect immediately, with visual confirmation.
6. **Multiple Input Methods**: Supports keyboard navigation, toggling, and text input in a unified interface.
7. **Safe Editing**: Provides validation and escape routes for configuration editing.

The `config` command demonstrates how Claude Code effectively combines the simplicity of terminal interfaces with the rich interaction capabilities typically associated with graphical applications, creating a powerful yet accessible configuration experience.

cost Command

The `cost` command provides users with visibility into the cost and duration of their current Claude Code session, helping them monitor their API usage and expenses.

Implementation

The command is implemented in `commands/cost.ts` as a simple type: 'local' command that calls a formatting function:

```
import type { Command } from "../commands";
import { formatTotalCost } from "../cost-tracker";

const cost = {
  type: "local",
  name: "cost",
  description: "Show the total cost and duration of the current session",
  isEnabled: true,
  isHidden: false,
  async call() {
    return formatTotalCost();
  },
  userFacingName() {
    return "cost";
  },
} satisfies Command;

export default cost;
```

This command relies on the cost tracking system implemented in `cost-tracker.ts`, which maintains a running total of API costs and session duration.

Cost Tracking System

The cost tracking system is implemented in `cost-tracker.ts` and consists of several key components:

1. **State Management**: Maintains a simple singleton state object tracking:
    - `totalCost`: Running total of API costs in USD
    - `totalAPIDuration`: Cumulative time spent waiting for API responses
    - `startTime`: Timestamp when the session began

2. **Cost Accumulation**: Provides a function to add costs as they occur:

```
export function addToTotalCost(cost: number, duration: number): void
{
  STATE.totalCost += cost;
  STATE.totalAPIDuration += duration;
}
```

3. **Reporting**: Formats cost information in a human-readable format:

```
export function formatTotalCost(): string {
  return chalk.grey(
    `Total cost: ${formatCost(STATE.totalCost)}
    Total duration (API): ${formatDuration(STATE.totalAPIDuration)}
    Total duration (wall): ${formatDuration(getTotalDuration())}`
  );
}
```

4. **Persistence**: Uses a React hook to save session cost data when the process exits:

```
export function useCostSummary(): void {
  useEffect(() => {
    const f = () => {
      process.stdout.write("\n" + formatTotalCost() + "\n");

      // Save last cost and duration to project config
      const projectConfig = getCurrentProjectConfig();
      saveCurrentProjectConfig({
        ...projectConfig,
        lastCost: STATE.totalCost,
        lastAPIDuration: STATE.totalAPIDuration,
        lastDuration: getTotalDuration(),
        lastSessionId: SESSION_ID,
      });
    };
    process.on("exit", f);
    return () => {
      process.off("exit", f);
    };
  }, []);
}
```

UI Components

The cost tracking system is complemented by two UI components:

1. **Cost Component**: A simple display component used in the debug panel to show the most recent API call cost:

```
export function Cost({
  costUSD,
  durationMs,
  debug,
}: Props): React.ReactNode {
  if (!debug) {
    return null;
  }

  const durationInSeconds = (durationMs / 1000).toFixed(1);
  return (
    <Box flexDirection="column" minWidth={23} width={23}>
      <Text dimColor>
        Cost: ${costUSD.toFixed(4)} ({durationInSeconds}s)
      </Text>
    </Box>
  );
}
```

2. **CostThresholdDialog**: A warning dialog shown when users exceed a certain cost threshold:

```
export function CostThresholdDialog({ onDone }: Props):
React.ReactNode {
  // Handle Ctrl+C, Ctrl+D and Esc
  useInput((input, key) => {
    if ((key.ctrl && (input === "c" || input === "d")) ||
key.escape) {
      onDone();
    }
  });

  return (
    <Box
      flexDirection="column"
      borderStyle="round"
      padding={1}
      borderColor={getTheme().secondaryBorder}
    >
      <Box marginBottom={1} flexDirection="column">
        <Text bold>You've spent $5 on the Anthropic API this
session.</Text>
        <Text>Learn more about how to monitor your spending:</Text>
        <Link url="https://docs.anthropic.com/s/claude-code-cost" />
      </Box>
      <Box>
        <Select
          options={[
            {
              value: "ok",
              label: "Got it, thanks!",
            },
          ]}
          onChange={onDone}
        />
      </Box>
    </Box>
  );
}
```

Technical Implementation Notes

The cost tracking system demonstrates several design considerations:

1. **Singleton State**: Uses a single state object with a clear comment warning against adding more state.
2. **Persistence Across Sessions**: Saves cost data to the project configuration, allowing for tracking across sessions.
3. **Formatting Flexibility**: Uses different decimal precision based on the cost amount (4 decimal places for small amounts, 2 for larger ones).
4. **Multiple Time Metrics**: Tracks both wall clock time and API request time separately.
5. **Environment-Aware Testing**: Includes a reset function that's only available in test environments.
6. **Exit Hooks**: Uses process exit hooks to ensure cost data is saved and displayed even if the application exits unexpectedly.

User Experience Considerations

The cost tracking system addresses several user needs:

1. **Transparency**: Provides clear visibility into API usage costs.
2. **Usage Monitoring**: Helps users track and manage their API spending.
3. **Efficiency Insights**: Shows both total runtime and API time, helping identify bottlenecks.
4. **Threshold Warnings**: Alerts users when they've spent significant amounts.
5. **Documentation Links**: Provides resources for learning more about cost management.

The `/cost` command and associated systems represent Claude Code's approach to transparent cost management, giving users control over their API usage while maintaining a simple, unobtrusive interface.

ctx_viz Command

The `ctx_viz` command provides a detailed visualization of token usage across different components of the Claude conversation context, helping users understand how their context window is being utilized.

Implementation

The command is implemented in `commands/ctx_viz.ts` as a type: 'local' command that generates a formatted table of token usage:

```
import type { Command } from "../commands";
import type { Tool } from "../Tool";
import Table from "cli-table3";
import { getSystemPrompt } from "../constants/prompts";
import { getContext } from "../context";
import { zodToJsonSchema } from "zod-to-json-schema";
import { getMessagesGetter } from "../messages";

// Quick and dirty estimate of bytes per token for rough token counts
const BYTES_PER_TOKEN = 4;

interface Section {
  title: string;
  content: string;
}

interface ToolSummary {
  name: string;
  description: string;
}

function getContextSections(text: string): Section[] {
  const sections: Section[] = [];

  // Find first <context> tag
  const firstContextIndex = text.indexOf("<context");

  // Everything before first tag is Core Sysprompt
  if (firstContextIndex > 0) {
    const coreSysprompt = text.slice(0, firstContextIndex).trim();
    if (coreSysprompt) {
      sections.push({
        title: "Core Sysprompt",
        content: coreSysprompt,
      });
    }
```

```
  }

  let currentPos = firstContextIndex;
  let nonContextContent = "";

  const regex = /<context\s+name="([^"]*)">([\s\S]*?)<\/context>/g;
  let match: RegExpExecArray | null;

  while ((match = regex.exec(text)) !== null) {
    // Collect text between context tags
    if (match.index > currentPos) {
      nonContextContent += text.slice(currentPos, match.index);
    }

    const [, name = "Unnamed Section", content = ""] = match;
    sections.push({
      title: name === "codeStyle" ? "CodeStyle + KODING.md's" : name,
      content: content.trim(),
    });

    currentPos = match.index + match[0].length;
  }

  // Collect remaining text after last tag
  if (currentPos < text.length) {
    nonContextContent += text.slice(currentPos);
  }

  // Add non-contextualized content if present
  const trimmedNonContext = nonContextContent.trim();
  if (trimmedNonContext) {
    sections.push({
      title: "Non-contextualized Content",
      content: trimmedNonContext,
    });
  }

  return sections;
}

function formatTokenCount(bytes: number): string {
  const tokens = bytes / BYTES_PER_TOKEN;
  const k = tokens / 1000;
  return `${Math.round(k * 10) / 10}k`;
}

function formatByteCount(bytes: number): string {
  const kb = bytes / 1024;
  return `${Math.round(kb * 10) / 10}kb`;
}

function createSummaryTable(
  systemText: string,
  systemSections: Section[],
```

```
  tools: ToolSummary[],
  messages: unknown
): string {
  const table = new Table({
    head: ["Component", "Tokens", "Size", "% Used"],
    style: { head: ["bold"] },
    chars: {
      mid: "─",
      "left-mid": "├",
      "mid-mid": "┼",
      "right-mid": "┤",
    },
  });

  const messagesStr = JSON.stringify(messages);
  const toolsStr = JSON.stringify(tools);

  // Calculate total for percentages
  const total = systemText.length + toolsStr.length + messagesStr.length;
  const getPercentage = (n: number) => `${Math.round((n / total) *
100)}%`;

  // System prompt and its sections
  table.push([
    "System prompt",
    formatTokenCount(systemText.length),
    formatByteCount(systemText.length),
    getPercentage(systemText.length),
  ]);
  for (const section of systemSections) {
    table.push([
      `  ${section.title}`,
      formatTokenCount(section.content.length),
      formatByteCount(section.content.length),
      getPercentage(section.content.length),
    ]);
  }

  // Tools
  table.push([
    "Tool definitions",
    formatTokenCount(toolsStr.length),
    formatByteCount(toolsStr.length),
    getPercentage(toolsStr.length),
  ]);
  for (const tool of tools) {
    table.push([
      `  ${tool.name}`,
      formatTokenCount(tool.description.length),
      formatByteCount(tool.description.length),
      getPercentage(tool.description.length),
    ]);
  }
```

```
  // Messages and total
  table.push(
    [
      "Messages",
      formatTokenCount(messagesStr.length),
      formatByteCount(messagesStr.length),
      getPercentage(messagesStr.length),
    ],
    ["Total", formatTokenCount(total), formatByteCount(total), "100%"]
  );

  return table.toString();
}

const command: Command = {
  name: "ctx-viz",
  description:
    "[ANT-ONLY] Show token usage breakdown for the current conversation
context",
  isEnabled: true,
  isHidden: false,
  type: "local",

  userFacingName() {
    return this.name;
  },

  async call(_args: string, cmdContext: { options: { tools: Tool[] } }) {
    // Get tools and system prompt with injected context
    const [systemPromptRaw, sysContext] = await Promise.all([
      getSystemPrompt(),
      getContext(),
    ]);

    const rawTools = cmdContext.options.tools;

    // Full system prompt with context sections injected
    let systemPrompt = systemPromptRaw.join("\n");
    for (const [name, content] of Object.entries(sysContext)) {
      systemPrompt += `\n<context name="${name}">${content}</context>`;
    }

    // Get full tool definitions including prompts and schemas
    const tools = rawTools.map((t) => {
      // Get full prompt and schema
      const fullPrompt = t.prompt({ dangerouslySkipPermissions: false });
      const schema = JSON.stringify(
        "inputJSONSchema" in t && t.inputJSONSchema
          ? t.inputJSONSchema
          : zodToJsonSchema(t.inputSchema)
      );

      return {
        name: t.name,
```

```
          description: `${fullPrompt}\n\nSchema:\n${schema}`,
        };
      });

      // Get current messages from REPL
      const messages = getMessagesGetter()();

      const sections = getContextSections(systemPrompt);
      return createSummaryTable(systemPrompt, sections, tools, messages);
    },
  };

  export default command;
```

Functionality

The `ctx_viz` command provides a detailed breakdown of token usage across different components of the conversation context:

1. **System Prompt Analysis**:
   - Parses the system prompt to identify its separate sections
   - Extracts `<context>` tags and their contents for individual analysis
   - Identifies core system prompt sections vs. injected context
2. **Token Usage Calculation**:
   - Estimates token usage based on a bytes-per-token approximation
   - Presents data in kilobytes and estimated token counts
   - Calculates percentage usage for each component of the context
3. **Tool Definitions Analysis**:
   - Extracts complete tool definitions including prompts and JSON schemas
   - Calculates token usage per tool
   - Shows the total footprint of tool definitions in the context
4. **Conversation Message Analysis**:
   - Includes the current message history in the analysis
   - Shows what portion of the context window is used by the conversation
5. **Structured Presentation**:

- Outputs a formatted ASCII table with columns for component, tokens, size, and percentage
- Uses hierarchical indentation to show the structure of the context
- Includes totals for complete context usage

Technical Implementation Notes

The `ctx_viz` command demonstrates several sophisticated implementation patterns:

1. **Regex-Based Context Parsing**: Uses regular expressions to parse the context sections from the system prompt, handling nested tags and multi-line content.

2. **Parallel Resource Loading**: Uses `Promise.all` to concurrently fetch system prompt and context data for efficiency.

3. **Tool Schema Introspection**: Extracts JSON schemas from tool definitions using either explicit schemas or by converting Zod schemas to JSON Schema format.

4. **Token Approximation**: Implements a simple but effective token estimation approach based on byte length, which provides a reasonable approximation without requiring a tokenizer.

5. **Table Formatting**: Uses the `cli-table3` library to create a formatted ASCII table with custom styling, making the output readable in a terminal environment.

6. **Context Section Management**: Special-cases certain context sections like "codeStyle" with custom labeling for clarity.

User Experience Benefits

The `ctx_viz` command addresses several important needs for developers using Claude Code:

1. **Context Window Transparency**: Gives users insight into how their limited context window is being utilized.

2. **Optimization Opportunities**: Helps identify large components that might be

consuming excessive context, enabling targeted optimization.

3. **Debugging Aid**: Provides a debugging tool for situations where context limitations are affecting Claude's performance.

4. **System Prompt Visibility**: Makes the usually hidden system prompt and context visible to users for better understanding of Claude's behavior.

The command is particularly valuable for advanced users and developers who need to understand and optimize their context usage to get the most out of Claude's capabilities within token limitations.

doctor Command

The `doctor` command provides a diagnostic tool for checking the health of the Claude Code installation, with a focus on npm permissions required for proper auto-updating functionality.

Implementation

The command is implemented in `commands/doctor.ts` as a type: 'local-jsx' command that renders a React component:

```
import React from "react";
import type { Command } from "../commands";
import { Doctor } from "../screens/Doctor";

const doctor: Command = {
  name: "doctor",
  description: "Checks the health of your Claude Code installation",
  isEnabled: true,
  isHidden: false,
  userFacingName() {
    return "doctor";
  },
  type: "local-jsx",
  call(onDone) {
    const element = React.createElement(Doctor, {
      onDone,
      doctorMode: true,
    });
    return Promise.resolve(element);
  },
};

export default doctor;
```

The command uses the `Doctor` screen component defined in `screens/Doctor.tsx`, passing the special `doctorMode` flag to indicate it's being used as a diagnostic tool rather than during initialization.

Functionality

The `doctor` command implements a comprehensive installation health check

focused on npm permissions:

1. **Permissions Verification**:
    - Checks if the npm global installation directory has correct write permissions
    - Determines the current npm prefix path
    - Validates if auto-updates can function correctly
2. **Status Reporting**:
    - Provides clear success or failure messages about installation health
    - Shows a green checkmark and confirmation message for healthy installations
    - Presents an interactive dialog for resolving permission issues
3. **Problem Resolution**:
    - Offers three remediation options when permission problems are detected:
        1. **Manual Fix**: Provides a sudo/icacls command to fix permissions on the current npm prefix
        2. **New Prefix**: Guides the user through creating a new npm prefix in their home directory
        3. **Skip**: Allows deferring the fix until later
4. **Installation Repair**:
    - For the "new prefix" option, provides a guided, step-by-step process to:
        - Create a new directory for npm global packages
        - Configure npm to use the new location
        - Update shell PATH configurations
        - Reinstall Claude Code globally

Technical Implementation Notes

The command demonstrates several sophisticated implementation patterns:

1. **Component-Based Architecture**: Uses React components for the UI, allowing for a rich interactive experience.
2. **Platform-Aware Logic**: Implements different permission-fixing approaches for Windows vs. Unix-like systems:
    - Windows: Uses `icacls` for permission management and `setx` for PATH

updates
    - Unix: Uses `sudo chown/chmod` for permission fixes and shell config file updates

3. **Shell Detection**: For Unix platforms, identifies and updates multiple possible shell configuration files:
    - .bashrc and .bash_profile for Bash
    - .zshrc for Zsh
    - config.fish for Fish shell

4. **Status Management**: Uses React state to track:
    - Permission check status
    - Selected remediation option
    - Custom prefix path (if chosen)
    - Step-by-step installation progress

5. **Lock-Based Concurrency Control**: Implements a file-based locking mechanism to prevent multiple processes from attempting auto-updates simultaneously.

6. **Error Handling**: Provides detailed error reporting and recovery options:
    - Shows precisely which operation failed
    - Offers alternative approaches when errors occur
    - Logs error details for debugging

User Experience Benefits

The `doctor` command addresses several important pain points for CLI tool users:

1. **Installation Troubleshooting**: Provides clear diagnostics and fixes for common problems:
    - Permission issues that prevent updates
    - Global npm configuration problems
    - PATH configuration issues

2. **Security-Conscious Design**: Offers both privileged (sudo) and non-privileged (new prefix) solutions, allowing users to choose based on their security preferences.

3. **Multi-Platform Support**: Works identically across Windows, macOS, and Linux

with platform-appropriate solutions.

4. **Shell Environment Enhancement**: Automatically updates shell configuration files to ensure the installation works correctly across terminal sessions.

5. **Visual Progress Feedback**: Uses spinners and checkmarks to keep users informed about long-running operations.

The command provides a comprehensive diagnostics and repair tool that helps maintain a healthy Claude Code installation, particularly focusing on the auto-update capability which is crucial for keeping the tool current with new features and improvements.

help Command

The `help` command provides users with information about Claude Code's usage, available commands, and general guidance for effective interaction with the tool.

Implementation

The command is implemented in `commands/help.tsx` as a type: 'local-jsx' command that renders a React component:

```
import { Command } from "../commands";
import { Help } from "../components/Help";
import * as React from "react";

const help = {
  type: "local-jsx",
  name: "help",
  description: "Show help and available commands",
  isEnabled: true,
  isHidden: false,
  async call(onDone, { options: { commands } }) {
    return <Help commands={commands} onClose={onDone} />;
  },
  userFacingName() {
    return "help";
  },
} satisfies Command;

export default help;
```

The command delegates to the `Help` component in `components/Help.tsx`, passing the list of available commands and an `onClose` callback to handle when the help screen should be dismissed.

Functionality

The `Help` component implements a progressive disclosure pattern for displaying information:

1. **Basic Information**:
    - Shows the product name and version

    - Displays a brief description of Claude Code's capabilities and limitations
    - Presents a disclaimer about the beta nature of the product

2. **Usage Modes** (shown after a brief delay):
    - Explains the two primary ways to use Claude Code:
        - REPL (interactive session)
        - Non-interactive mode with the `-p` flag for one-off questions
    - Mentions that users can run `claude -h` for additional command-line options

3. **Common Tasks** (shown after a longer delay):
    - Lists examples of typical use cases:
        - Asking questions about the codebase
        - Editing files
        - Fixing errors
        - Running commands
        - Running bash commands

4. **Available Commands** (shown after the longest delay):
    - Displays a comprehensive list of all enabled slash commands
    - Shows command names and descriptions
    - Filters out hidden commands

5. **Additional Resources**:
    - Provides links for getting more help
    - Shows different resources based on user type (internal vs. external)

Technical Implementation Notes

The `help` command demonstrates several effective patterns:

1. **Progressive Disclosure**: Uses a time-based mechanism to gradually reveal information:

```
const [count, setCount] = React.useState(0);

React.useEffect(() => {
  const timer = setTimeout(() => {
    if (count < 3) {
      setCount(count + 1);
    }
  }, 250);

  return () => clearTimeout(timer);
}, [count]);
```

This approach avoids overwhelming users with too much information at once, showing more details as they spend time on the help screen.

2. **Filtering System**: Uses `filter` to show only non-hidden commands:

```
const filteredCommands = commands.filter((cmd) => !cmd.isHidden);
```

This keeps the help screen focused on commands relevant to users.

3. **Dynamic Resource Links**: Changes help resources based on user type:

```
const isInternal = process.env.USER_TYPE === "ant";
const moreHelp = isInternal
  ? "[ANT-ONLY] For more help: go/claude-cli or #claude-cli-feedback"
  : `Learn more at: ${MACRO.README_URL}`;
```

This customizes the experience for different user populations.

4. **Input Handling**: Uses Ink's `useInput` hook to detect when the user presses Enter:

```
useInput((_, key) => {
  if (key.return) onClose();
});
```

This allows for clean dismissal of the help screen.

5. **Conditional Rendering**: Uses the `count` state to progressively show different sections:

```
{
  count >= 1 && (
    <Box flexDirection="column" marginTop={1}>
      <Text bold>Usage Modes:</Text>
      {/* Usage content */}
    </Box>
  );
}
```

This creates the staggered reveal effect for information groups.

User Experience Benefits

The `help` command addresses several important needs for CLI tool users:

1. **Onboarding Assistance**: Provides new users with immediate guidance on how to use the tool effectively.
2. **Command Discovery**: Makes it easy to see what slash commands are available without having to memorize them.
3. **Progressive Learning**: The staggered reveal of information allows users to absorb basics first before seeing more advanced options.
4. **Usage Examples**: Shows concrete examples of common use cases, helping users understand practical applications.

5. **Quick Reference**: Serves as a compact reference for command options during regular use.

The help command exemplifies Claude Code's approach to user experience: it's focused on being informative while remaining concise and unobtrusive, with multiple levels of detail available as users need them.

init Command

The `init` command helps users initialize a new project for use with Claude Code by creating a KODING.md file that contains key information about the codebase and project.

Implementation

The command is implemented in `commands/init.ts` as a type: 'prompt' command that passes a specific request to Claude:

```
import type { Command } from "../commands";
import { markProjectOnboardingComplete } from "../ProjectOnboarding";

const command = {
  type: "prompt",
  name: "init",
  description: "Initialize a new KODING.md file with codebase
documentation",
  isEnabled: true,
  isHidden: false,
  progressMessage: "analyzing your codebase",
  userFacingName() {
    return "init";
  },
  async getPromptForCommand(_args: string) {
    // Mark onboarding as complete when init command is run
    markProjectOnboardingComplete();
    return [
      {
        role: "user",
        content: [
          {
            type: "text",
            text: `Please analyze this codebase and create a KODING.md
file containing:
1. Build/lint/test commands - especially for running a single test
2. Code style guidelines including imports, formatting, types, naming
conventions, error handling, etc.

The file you create will be given to agentic coding agents (such as
yourself) that operate in this repository. Make it about 20 lines long.
If there's already a KODING.md, improve it.
If there are Cursor rules (in .cursor/rules/ or .cursorrules) or Copilot
rules (in .github/copilot-instructions.md), make sure to include them.`,
          },
        ],
      },
    ];
  },
} satisfies Command;

export default command;
```

Unlike many other commands that directly perform actions, the `init` command is implemented as a 'prompt' type command that simply formulates a specific request to Claude to analyze the codebase and generate a KODING.md file.

Functionality

The `init` command serves several important functions:

1. **Project Configuration Generation**:
    - Instructs Claude to analyze the codebase structure, conventions, and patterns
    - Generates a KODING.md file with essential project information
    - Focuses on capturing build/lint/test commands and code style guidelines
2. **Existing File Enhancement**:
    - Checks if a KODING.md file already exists
    - Improves the existing file with additional information if present
    - Preserves existing content while enhancing where needed
3. **Integration with Other Tools**:
    - Looks for Cursor rules (in .cursor/rules/ or .cursorrules)
    - Looks for Copilot instructions (in .github/copilot-instructions.md)
    - Incorporates these rules into the KODING.md file for a unified guidance document
4. **Onboarding Status Tracking**:
    - Calls `markProjectOnboardingComplete()` to update the project configuration
    - This flags the project as having completed onboarding steps
    - Prevents repeated onboarding prompts in future sessions

Technical Implementation Notes

The `init` command demonstrates several interesting technical aspects:

1. **Command Type**: Uses the 'prompt' type rather than 'local' or 'local-jsx', which means it sets up a specific request to Claude rather than executing custom logic or rendering a UI component.
2. **Progress Message**: Includes a `progressMessage` property ("analyzing your codebase") that's displayed to users while Claude processes the request, providing feedback during what could be a longer operation.
3. **Project Onboarding Integration**: The command is part of the project onboarding workflow, tracking completion state in the project configuration:

```
export function markProjectOnboardingComplete(): void {
  const projectConfig = getCurrentProjectConfig();
  if (!projectConfig.hasCompletedProjectOnboarding) {
    saveCurrentProjectConfig({
      ...projectConfig,
      hasCompletedProjectOnboarding: true,
    });
  }
}
```

4. **External Tool Recognition**: Explicitly looks for configuration files from other AI coding tools (Cursor, Copilot) to create a unified guidance document, showing awareness of the broader AI coding tool ecosystem.

5. **Conciseness Guidance**: Explicitly requests a document of around 20 lines, balancing comprehensive information with brevity for practical usage.

User Experience Benefits

The `init` command provides several important benefits for Claude Code users:

1. **Simplified Project Setup**: Makes it easy to prepare a codebase for effective use with Claude Code with a single command.

2. **Consistent Memory**: Creates a standardized document that Claude Code will access in future sessions, providing consistent context.

3. **Command Discovery**: By capturing build/test/lint commands, it helps Claude remember the correct commands for the project, reducing the need for users to repeatedly provide them.

4. **Code Style Guidance**: Helps Claude generate code that matches the project's conventions, improving integration of AI-generated code.

5. **Onboarding Pathway**: Serves as a key step in the onboarding flow for new projects, guiding users toward the most effective usage patterns.

The `init` command exemplifies Claude Code's overall approach to becoming more

effective over time by creating persistent context that improves future interactions. By capturing project-specific information in a standardized format, it enables Claude to provide more tailored assistance for each unique codebase.

listen Command

The `listen` command provides macOS users with the ability to dictate to Claude Code using speech recognition, enhancing accessibility and offering an alternative input method.

Implementation

The command is implemented in `commands/listen.ts` as a type: 'local' command that invokes macOS's built-in dictation feature:

```
import { Command } from "../commands";
import { logError } from "../utils/log";
import { execFileNoThrow } from "../utils/execFileNoThrow";

const isEnabled =
  process.platform === "darwin" &&
  ["iTerm.app", "Apple_Terminal"].includes(process.env.TERM_PROGRAM ||
"");

const listen: Command = {
  type: "local",
  name: "listen",
  description: "Activates speech recognition and transcribes speech to
text",
  isEnabled: isEnabled,
  isHidden: isEnabled,
  userFacingName() {
    return "listen";
  },
  async call(_, { abortController }) {
    // Start dictation using AppleScript
    const script = `tell application "System Events" to tell ¬
(the first process whose frontmost is true) to tell ¬
menu bar 1 to tell ¬
menu bar item "Edit" to tell ¬
menu "Edit" to tell ¬
menu item "Start Dictation" to ¬
if exists then click it`;

    const { stderr, code } = await execFileNoThrow(
      "osascript",
      ["-e", script],
      abortController.signal
    );

    if (code !== 0) {
      logError(`Failed to start dictation: ${stderr}`);
      return "Failed to start dictation";
    }
    return "Dictation started. Press esc to stop.";
  },
};

export default listen;
```

The command uses AppleScript to trigger macOS's built-in dictation feature, automating the process of selecting "Start Dictation" from the Edit menu.

Functionality

The `listen` command provides a simple but effective accessibility enhancement:

1. **Platform Integration**:
    - Invokes macOS's built-in dictation feature
    - Uses AppleScript to navigate the application's menu structure
    - Triggers the "Start Dictation" menu item in the Edit menu
2. **Status Reporting**:
    - Returns clear success or error messages
    - Provides user instructions on how to stop dictation
    - Logs detailed error information when dictation fails to start
3. **Platform-Specific Availability**:
    - Only enabled on macOS platforms
    - Further restricted to specific terminal applications (iTerm.app and Apple_Terminal)
    - Hidden from command listings on unsupported platforms

Technical Implementation Notes

The `listen` command demonstrates several interesting technical approaches:

1. **Platform Detection**: Uses environment variables and platform checks to determine when the command should be available:

```
const isEnabled =
  process.platform === "darwin" &&
  ["iTerm.app", "Apple_Terminal"].includes(process.env.TERM_PROGRAM
|| "");
```

2. **AppleScript Integration**: Uses a complex AppleScript command to navigate through application menus:

```
const script = `tell application "System Events" to tell ¬
(the first process whose frontmost is true) to tell ¬
menu bar 1 to tell ¬
menu bar item "Edit" to tell ¬
menu "Edit" to tell ¬
menu item "Start Dictation" to ¬
if exists then click it`;
```

This script navigates the UI hierarchy to find and click the dictation menu item.

3. **Command Visibility Control**: Uses the same conditions for both `isEnabled` and `isHidden`, ensuring the command is simultaneously enabled and hidden on supported platforms:

```
isEnabled: isEnabled,
isHidden: isEnabled,
```

This unusual pattern makes the command available but not visible in help listings, suggesting it's an internal or experimental feature.

4. **Error Handling**: Implements robust error handling with detailed logging:

```
if (code !== 0) {
  logError(`Failed to start dictation: ${stderr}`);
  return "Failed to start dictation";
}
```

User Experience Considerations

While simple in implementation, the `listen` command addresses several important user needs:

1. **Accessibility Enhancement**: Provides an alternative input method for users who prefer or require speech recognition.

2. **Workflow Efficiency**: Eliminates the need to manually navigate menus to start dictation, streamlining the process.

3. **Integrated Experience**: Keeps users within the Claude Code interface rather than requiring them to use separate dictation tools.

4. **Platform Integration**: Leverages native OS capabilities rather than implementing custom speech recognition, ensuring high-quality transcription.

The `listen` command demonstrates Claude Code's commitment to accessibility and platform integration, even if the implementation is relatively simple. By leveraging existing OS capabilities rather than reinventing them, it provides a valuable feature with minimal code complexity.

login Command

The `login` command provides users with a secure OAuth 2.0 authentication flow to connect Claude Code with their Anthropic Console account, enabling API access and proper billing.

Implementation

The command is implemented in `commands/login.tsx` as a type: 'local-jsx' command that renders a React component for the authentication flow:

```tsx
import * as React from "react";
import type { Command } from "../commands";
import { ConsoleOAuthFlow } from "../components/ConsoleOAuthFlow";
import { clearTerminal } from "../utils/terminal";
import { isLoggedInToAnthropic } from "../utils/auth";
import { useExitOnCtrlCD } from "../hooks/useExitOnCtrlCD";
import { Box, Text } from "ink";
import { clearConversation } from "./clear";

export default () =>
  ({
    type: "local-jsx",
    name: "login",
    description: isLoggedInToAnthropic()
      ? "Switch Anthropic accounts"
      : "Sign in with your Anthropic account",
    isEnabled: true,
    isHidden: false,
    async call(onDone, context) {
      await clearTerminal();
      return (
        <Login
          onDone={async () => {
            clearConversation(context);
            onDone();
          }}
        />
      );
    },
    userFacingName() {
      return "login";
    },
  } satisfies Command);

function Login(props: { onDone: () => void }) {
  const exitState = useExitOnCtrlCD(props.onDone);
  return (
    <Box flexDirection="column">
      <ConsoleOAuthFlow onDone={props.onDone} />
      <Box marginLeft={3}>
        <Text dimColor>
          {exitState.pending ? (
            <>Press {exitState.keyName} again to exit</>
          ) : (
            ""
          )}
        </Text>
      </Box>
    </Box>
  );
}
```

The command uses a factory function to dynamically create a command object that adapts its description based on the current login state. The main logic is delegated to the `ConsoleOAuthFlow` component, which handles the OAuth flow with the Anthropic API.

Functionality

The `login` command implements a comprehensive authentication flow:

1. **OAuth 2.0 Integration**:
    - Uses OAuth 2.0 with PKCE (Proof Key for Code Exchange) for security
    - Opens a browser for the user to log in to Anthropic Console
    - Handles both automatic browser redirect and manual code entry flows
    - Securely exchanges authorization codes for access tokens
2. **Account Management**:
    - Creates and stores API keys for authenticated users
    - Normalizes and securely saves keys in the global configuration
    - Tracks account information including account UUID and organization details
    - Provides context-aware descriptions ("Sign in" vs "Switch accounts")
3. **Security Features**:
    - Implements state verification to prevent CSRF attacks
    - Uses code verifiers and challenges for secure code exchange
    - Validates all tokens and responses from the authentication server
    - Provides clear error messages for authentication failures
4. **UI Experience**:
    - Clears the terminal for a clean login experience
    - Provides a progressive disclosure flow with appropriate status messages
    - Offers fallback mechanisms for cases where automatic browser opening fails
    - Shows loading indicators during asynchronous operations

Technical Implementation Notes

The login command demonstrates several sophisticated technical approaches:

1. **Local HTTP Server**: Creates a temporary HTTP server for OAuth callback handling:

```
this.server = http.createServer(
  (req: IncomingMessage, res: ServerResponse) => {
    const parsedUrl = url.parse(req.url || "", true);
    if (parsedUrl.pathname === "/callback") {
      // Handle OAuth callback
    }
  }
);
```

2. **PKCE Implementation**: Implements the Proof Key for Code Exchange extension to OAuth:

```
function generateCodeVerifier(): string {
  return base64URLEncode(crypto.randomBytes(32));
}

async function generateCodeChallenge(verifier: string):
Promise<string> {
  const encoder = new TextEncoder();
  const data = encoder.encode(verifier);
  const digest = await crypto.subtle.digest("SHA-256", data);
  return base64URLEncode(Buffer.from(digest));
}
```

3. **Parallel Authentication Paths**: Supports both automatic and manual authentication flows:

```
const { autoUrl, manualUrl } = this.generateAuthUrls(codeChallenge,
state);
await authURLHandler(manualUrl); // Show manual URL in UI
await openBrowser(autoUrl); // Try automatic browser opening
```

4. **Promise-Based Flow Control**: Uses promises to coordinate the asynchronous authentication flow:

```
const { authorizationCode, useManualRedirect } = await new Promise<{
  authorizationCode: string;
  useManualRedirect: boolean;
}>((resolve, reject) => {
  this.pendingCodePromise = { resolve, reject };
  this.startLocalServer(state, onReady);
});
```

5. **State Management with React**: Uses React state and hooks for UI management:

```
const [oauthStatus, setOAuthStatus] = useState<OAuthStatus>({
  state: "idle",
});
```

6. **Error Recovery**: Implements sophisticated error handling with retry mechanisms:

```
if (oauthStatus.state === "error" && oauthStatus.toRetry) {
  setPastedCode("");
  setOAuthStatus({
    state: "about_to_retry",
    nextState: oauthStatus.toRetry,
  });
}
```

User Experience Benefits

The `login` command addresses several important user needs:

1. **Seamless Authentication**: Provides a smooth authentication experience

without requiring manual API key creation or copying.

2. **Cross-Platform Compatibility**: Works across different operating systems and browsers.
3. **Fallback Mechanisms**: Offers manual code entry when automatic browser redirection fails.
4. **Clear Progress Indicators**: Shows detailed status messages throughout the authentication process.
5. **Error Resilience**: Provides helpful error messages and retry options when authentication issues occur.
6. **Account Switching**: Allows users to easily switch between different Anthropic accounts.

The login command exemplifies Claude Code's approach to security and user experience, implementing a complex authentication flow with attention to both security best practices and ease of use.

logout Command

The `logout` command provides users with the ability to sign out from their Anthropic account, removing stored authentication credentials and API keys from the local configuration.

Implementation

The command is implemented in `commands/logout.tsx` as a type: 'local-jsx' command that handles the logout process and renders a confirmation message:

```
import * as React from "react";
import type { Command } from "../commands";
import { getGlobalConfig, saveGlobalConfig } from "../utils/config";
import { clearTerminal } from "../utils/terminal";
import { Text } from "ink";

export default {
  type: "local-jsx",
  name: "logout",
  description: "Sign out from your Anthropic account",
  isEnabled: true,
  isHidden: false,
  async call() {
    await clearTerminal();

    const config = getGlobalConfig();

    config.oauthAccount = undefined;
    config.primaryApiKey = undefined;
    config.hasCompletedOnboarding = false;

    if (config.customApiKeyResponses?.approved) {
      config.customApiKeyResponses.approved = [];
    }

    saveGlobalConfig(config);

    const message = (
      <Text>Successfully logged out from your Anthropic account.</Text>
    );

    setTimeout(() => {
      process.exit(0);
    }, 200);

    return message;
  },
  userFacingName() {
    return "logout";
  },
} satisfies Command;
```

Unlike the more complex `login` command, the `logout` command is relatively straightforward, focusing on removing authentication data from the configuration and providing a clean exit.

Functionality

The `logout` command performs several critical operations:

1. **Credential Removal**:
    - Clears the OAuth account information from the global configuration
    - Removes the primary API key used for authentication
    - Erases the list of approved API keys from storage
    - Resets the onboarding completion status
2. **User Experience**:
    - Clears the terminal before displaying the logout message
    - Provides a clear confirmation message about successful logout
    - Exits the application completely after a short delay
    - Ensures a clean break with the authenticated session
3. **Security Focus**:
    - Removes all sensitive authentication data from the local configuration
    - Ensures the next application start will require re-authentication
    - Prevents accidental API usage with old credentials
    - Provides a clean slate for a new login if desired

Technical Implementation Notes

Despite its relative simplicity, the `logout` command demonstrates several interesting implementation details:

1. **Configuration Management**: Uses the global configuration system to handle persistent state:

```
const config = getGlobalConfig();

config.oauthAccount = undefined;
config.primaryApiKey = undefined;
config.hasCompletedOnboarding = false;

if (config.customApiKeyResponses?.approved) {
  config.customApiKeyResponses.approved = [];
}

saveGlobalConfig(config);
```

2. **Graceful Exit Strategy**: Uses a timeout to allow the message to be displayed before exiting:

```
setTimeout(() => {
  process.exit(0);
}, 200);
```

   This ensures the user sees confirmation before the application closes.

3. **Type Safety**: Uses the `satisfies Command` pattern to ensure type correctness:

```
export default {
  // Command implementation
} satisfies Command;
```

4. **Terminal Management**: Clears the terminal before displaying the logout confirmation:

```
await clearTerminal();
```

   This creates a clean visual experience for the logout process.

5. **Optional Field Handling**: Carefully checks for the existence of optional configuration fields:

```
if (config.customApiKeyResponses?.approved) {
  config.customApiKeyResponses.approved = [];
}
```

User Experience Benefits

The `logout` command addresses several important user needs:

1. **Account Security**: Provides a clear way to remove credentials when sharing devices or ending a session.
2. **User Confidence**: Confirms successful logout with a clear message, reassuring users their credentials have been removed.
3. **Clean Exit**: Exits the application completely, avoiding any state confusion in the current process.
4. **Simplicity**: Keeps the logout process straightforward and quick, with minimal user interaction required.
5. **Fresh Start**: Resets the onboarding status, ensuring a proper re-onboarding flow on next login.

The `logout` command provides a necessary counterpart to the `login` command, completing the authentication lifecycle with a secure, clean way to end a session. While much simpler than its login counterpart, it maintains the same attention to security and user experience that characterizes Claude Code's approach to authentication management.

model Command

> [!WARNING] This command implementation is specific to the anon-kode fork of Claude Code and is not part of the original Claude Code codebase. The analysis below pertains to this specific implementation rather than standard Claude Code functionality.

The `model` command provides users with a comprehensive interface to configure and customize the AI models used by Claude Code, enabling fine-grained control over model selection, parameters, and provider settings.

Implementation

The command is implemented in `commands/model.tsx` as a type: 'local-jsx' command that renders a React component for model configuration:

```
import React from "react";
import { render } from "ink";
import { ModelSelector } from "../components/ModelSelector";
import { enableConfigs } from "../utils/config";

export const help = "Change your AI provider and model settings";
export const description = "Change your AI provider and model settings";
export const isEnabled = true;
export const isHidden = false;
export const name = "model";
export const type = "local-jsx";

export function userFacingName(): string {
  return name;
}

export async function call(
  onDone: (result?: string) => void,
  { abortController }: { abortController?: AbortController }
): Promise<React.ReactNode> {
  enableConfigs();
  abortController?.abort?.();
  return (
    <ModelSelector
      onDone={() => {
        onDone();
      }}
    />
  );
}
```

The command uses a different export style than other commands, directly exporting properties and functions rather than a single object. The main functionality is handled by the `ModelSelector` component, which provides an interactive UI for configuring model settings.

Functionality

The `model` command provides a sophisticated model selection and configuration workflow:

1. **Multi-Model Management**:
    - Allows configuring both "large" and "small" models separately or together
    - Provides different models for different task complexities for optimal

cost/performance
    - Shows current configuration information for reference

2. **Provider Selection**:
    - Supports multiple AI providers (Anthropic, OpenAI, Gemini, etc.)
    - Dynamically fetches available models from the selected provider's API
    - Handles provider-specific API requirements and authentication

3. **Model Parameters**:
    - Configures maximum token settings for response length control
    - Offers reasoning effort controls for supported models (low/medium/high)
    - Preserves provider-specific configuration options

4. **Search and Filtering**:
    - Provides search functionality to filter large model lists
    - Displays model capabilities including token limits and feature support
    - Organizes models with sensible sorting and grouping

5. **API Key Management**:
    - Securely handles API keys for different providers
    - Masks sensitive information during input and display
    - Stores keys securely in the local configuration

Technical Implementation Notes

The `model` command demonstrates several sophisticated technical approaches:

1. **Multi-Step Navigation**: Implements a screen stack pattern for intuitive flow navigation:

```
const [screenStack, setScreenStack] = useState<
  Array<
    | "modelType"
    | "provider"
    | "apiKey"
    | "model"
    | "modelParams"
    | "confirmation"
  >
>(["modelType"]);

// Current screen is always the last item in the stack
const currentScreen = screenStack[screenStack.length - 1];

// Function to navigate to a new screen
const navigateTo = (
  screen:
    | "modelType"
    | "provider"
    | "apiKey"
    | "model"
    | "modelParams"
    | "confirmation"
) => {
  setScreenStack((prev) => [...prev, screen]);
};

// Function to go back to the previous screen
const goBack = () => {
  if (screenStack.length > 1) {
    // Remove the current screen from the stack
    setScreenStack((prev) => prev.slice(0, -1));
  } else {
    // If we're at the first screen, call onDone to exit
    onDone();
  }
};
```

2. **Dynamic Model Loading**: Fetches available models directly from provider APIs:

```
async function fetchModels() {
  setIsLoadingModels(true);
  setModelLoadError(null);

  try {
    // Provider-specific logic...
    const openai = new OpenAI({
      apiKey: apiKey,
      baseURL: baseURL,
      dangerouslyAllowBrowser: true,
    });

    // Fetch the models
    const response = await openai.models.list();

    // Transform the response into our ModelInfo format
    const fetchedModels = [];
    // Process models...

    return fetchedModels;
  } catch (error) {
    setModelLoadError(`Failed to load models: ${error.message}`);
    throw error;
  } finally {
    setIsLoadingModels(false);
  }
}
```

3. **Form Focus Management**: Implements sophisticated form navigation with keyboard support:

```
// Handle Tab key for form navigation in model params screen
useInput((input, key) => {
  if (currentScreen === "modelParams" && key.tab) {
    const formFields = getFormFieldsForModelParams();
    // Move to next field
    setActiveFieldIndex((current) => (current + 1) %
formFields.length);
    return;
  }

  // Handle Enter key for form submission in model params screen
  if (currentScreen === "modelParams" && key.return) {
    const formFields = getFormFieldsForModelParams();

    if (activeFieldIndex === formFields.length - 1) {
      // If on the Continue button, submit the form
      handleModelParamsSubmit();
    }
    return;
  }
});
```

4. **Provider-Specific Handling**: Implements custom logic for different AI providers:

```
// For Gemini, use the separate fetchGeminiModels function
if (selectedProvider === "gemini") {
  const geminiModels = await fetchGeminiModels();
  setAvailableModels(geminiModels);
  navigateTo("model");
  return geminiModels;
}
```

5. **Configuration Persistence**: Carefully updates global configuration with new model settings:

```
function saveConfiguration(provider: ProviderType, model: string) {
  const baseURL = providers[provider]?.baseURL || "";

  // Create a new config object based on the existing one
  const newConfig = { ...config };

  // Update the primary provider regardless of which model we're
changing
  newConfig.primaryProvider = provider;

  // Update the appropriate model based on the selection
  if (modelTypeToChange === "both" || modelTypeToChange === "large")
{
    newConfig.largeModelName = model;
    newConfig.largeModelBaseURL = baseURL;
    newConfig.largeModelApiKey = apiKey || config.largeModelApiKey;
    newConfig.largeModelMaxTokens = parseInt(maxTokens);

    // Save reasoning effort for large model if supported
    if (supportsReasoningEffort) {
      newConfig.largeModelReasoningEffort = reasoningEffort;
    } else {
      newConfig.largeModelReasoningEffort = undefined;
    }
  }

  // Similar handling for small model...

  // Save the updated configuration
  saveGlobalConfig(newConfig);
}
```

User Experience Benefits

The `model` command provides several important benefits for Claude Code users:

1. **Customization Control**: Gives users fine-grained control over the AI models powering their interaction.

2. **Cost Optimization**: Allows setting different models for different complexity tasks, optimizing for cost and speed.
3. **Provider Flexibility**: Enables users to choose from multiple AI providers based on preference, cost, or feature needs.
4. **Parameter Tuning**: Offers advanced users the ability to tune model parameters for optimal performance.
5. **Progressive Disclosure**: Uses a step-by-step flow that makes configuration accessible to both novice and advanced users.
6. **Intuitive Navigation**: Implements keyboard navigation with clear indicators for a smooth configuration experience.

The `model` command exemplifies Claude Code's approach to giving users control and flexibility while maintaining an accessible interface, keeping advanced configuration options available but not overwhelming.

onboarding Command

The `onboarding` command provides a guided first-run experience for new users, helping them configure Claude Code to their preferences and introducing them to the tool's capabilities.

Implementation

The command is implemented in `commands/onboarding.tsx` as a type: 'local-jsx' command that renders a React component:

```
import * as React from "react";
import type { Command } from "../commands";
import { Onboarding } from "../components/Onboarding";
import { clearTerminal } from "../utils/terminal";
import { getGlobalConfig, saveGlobalConfig } from "../utils/config";
import { clearConversation } from "./clear";

export default {
  type: "local-jsx",
  name: "onboarding",
  description: "[ANT-ONLY] Run through the onboarding flow",
  isEnabled: true,
  isHidden: false,
  async call(onDone, context) {
    await clearTerminal();
    const config = getGlobalConfig();
    saveGlobalConfig({
      ...config,
      theme: "dark",
    });

    return (
      <Onboarding
        onDone={async () => {
          clearConversation(context);
          onDone();
        }}
      />
    );
  },
  userFacingName() {
    return "onboarding";
  },
} satisfies Command;
```

The command delegates to the `Onboarding` component in `components/Onboarding.tsx`, which handles the multi-step onboarding flow.

Functionality

The `onboarding` command implements a comprehensive first-run experience:

1. **Multi-Step Flow**:
    - Walks users through a series of configuration steps with a smooth, guided experience
    - Includes theme selection, usage guidance, and model selection
    - Uses a stack-based navigation system for intuitive flow between steps
2. **Theme Configuration**:
    - Allows users to choose between light and dark themes
    - Includes colorblind-friendly theme options for accessibility
    - Provides a live preview of the selected theme using a code diff example
3. **Usage Guidelines**:
    - Introduces users to effective ways of using Claude Code
    - Explains how to provide clear context and work with the tool
    - Sets appropriate expectations for the tool's capabilities
4. **Model Selection**:
    - Guides users through configuring AI provider and model settings
    - Uses the `ModelSelector` component for a consistent model selection experience
    - Allows configuration of both small and large models for different tasks
5. **Configuration Persistence**:
    - Saves user preferences to the global configuration
    - Marks onboarding as complete to prevent repeat runs
    - Clears the conversation after onboarding to provide a clean start

Technical Implementation Notes

The `onboarding` command demonstrates several sophisticated patterns:

1. **Screen Navigation Stack**: Implements a stack-based navigation system for multi-step flow:

```
const [screenStack, setScreenStack] = useState<
  Array<
    | "modelType"
    | "provider"
    | "apiKey"
    | "model"
    | "modelParams"
    | "confirmation"
  >
>(["modelType"]);

// Current screen is always the last item in the stack
const currentScreen = screenStack[screenStack.length - 1];

// Function to navigate to a new screen
const navigateTo = (screen) => {
  setScreenStack((prev) => [...prev, screen]);
};

// Function to go back to the previous screen
const goBack = () => {
  if (screenStack.length > 1) {
    setScreenStack((prev) => prev.slice(0, -1));
  } else {
    onDone();
  }
};
```

2. **Progressive Disclosure**: Presents information in digestible chunks across multiple steps.

3. **Terminal UI Adaptation**: Uses Ink components optimized for terminal rendering:

```
<Box flexDirection="column" gap={1} paddingLeft={1}>
  <Text bold>Using {PRODUCT_NAME} effectively:</Text>
  <Box flexDirection="column" width={70}>
    <OrderedList>
      <OrderedList.Item>
        <Text>
          Start in your project directory
          <Newline />
          <Text color={theme.secondaryText}>
            Files are automatically added to context when needed.
          </Text>
          <Newline />
        </Text>
      </OrderedList.Item>
      {/* Additional list items */}
    </OrderedList>
  </Box>
  <PressEnterToContinue />
</Box>
```

4. **Interactive Components**: Uses custom select components for theme selection with previews:

```
<Select
  options={[
    { label: "Light text", value: "dark" },
    { label: "Dark text", value: "light" },
    {
      label: "Light text (colorblind-friendly)",
      value: "dark-daltonized",
    },
    {
      label: "Dark text (colorblind-friendly)",
      value: "light-daltonized",
    },
  ]}
  onFocus={handleThemePreview}
  onChange={handleThemeSelection}
/>
```

5. **Exit Handling**: Implements `useExitOnCtrlCD` to provide users with a clear way to exit the flow:

```
const exitState = useExitOnCtrlCD(() => process.exit(0));
```

6. **Conditional Rendering**: Uses state to conditionally show different screens:

```
// If we're showing the model selector screen, render it directly
if (showModelSelector) {
  return <ModelSelector onDone={handleModelSelectionDone} />;
}
```

User Experience Benefits

The `onboarding` command addresses several key needs for new users:

1. **Guided Setup**: Provides a structured introduction to Claude Code rather than dropping users into a blank interface.

2. **Preference Customization**: Allows users to set their preferences immediately, increasing comfort with the tool.
3. **Learning Opportunity**: Teaches best practices for using the tool effectively from the start.
4. **Accessibility Awareness**: Explicitly offers colorblind-friendly themes, demonstrating attention to accessibility.
5. **Progressive Complexity**: Introduces features gradually, avoiding overwhelming new users.

The `onboarding` command exemplifies Claude Code's attention to user experience, ensuring new users can quickly set up the tool according to their preferences and learn how to use it effectively from the beginning.

pr_comments Command

The `pr_comments` command provides a specialized interface for retrieving and displaying GitHub pull request comments, helping users review feedback on their code without leaving the terminal.

Implementation

The command is implemented in `commands/pr_comments.ts` as a type: 'prompt' command that formulates a specialized request to Claude:

```
import { Command } from "../commands";

export default {
  type: "prompt",
  name: "pr-comments",
  description: "Get comments from a GitHub pull request",
  progressMessage: "fetching PR comments",
  isEnabled: true,
  isHidden: false,
  userFacingName() {
    return "pr-comments";
  },
  async getPromptForCommand(args: string) {
    return [
      {
        role: "user",
        content: [
          {
            type: "text",
            text: `You are an AI assistant integrated into a git-based
version control system. Your task is to fetch and display comments from a
GitHub pull request.

Follow these steps:

1. Use \`gh pr view --json number,headRepository\` to get the PR number
and repository info
2. Use \`gh api /repos/{owner}/{repo}/issues/{number}/comments\` to get
PR-level comments
3. Use \`gh api /repos/{owner}/{repo}/pulls/{number}/comments\` to get
review comments. Pay particular attention to the following fields:
\`body\`, \`diff_hunk\`, \`path\`, \`line\`, etc. If the comment
references some code, consider fetching it using eg \`gh api
/repos/{owner}/{repo}/contents/{path}?ref={branch} | jq .content -r |
base64 -d\`
4. Parse and format all comments in a readable way
```

```
5. Return ONLY the formatted comments, with no additional text

Format the comments as:

## Comments

[For each comment thread:]
- @author file.ts#line:
  \`\`\`diff
  [diff_hunk from the API response]
  \`\`\`
  > quoted comment text

  [any replies indented]

If there are no comments, return "No comments found."

Remember:
1. Only show the actual comments, no explanatory text
2. Include both PR-level and code review comments
3. Preserve the threading/nesting of comment replies
4. Show the file and line number context for code review comments
5. Use jq to parse the JSON responses from the GitHub API

${args ? "Additional user input: " + args : ""}
`,
          },
        ],
      },
    ];
  },
} satisfies Command;
```

Unlike commands that directly render UI components, the `pr_comments` command is a 'prompt' type command that formulates a specific request to Claude, instructing it to perform a complex sequence of operations using the GitHub CLI and API.

Functionality

The `pr_comments` command provides several key capabilities:

1. **GitHub Integration**:
   - Uses the GitHub CLI (`gh`) to interact with GitHub's API
   - Retrieves both PR-level comments (issue comments) and review comments (inline code comments)
   - Handles authentication and API access through the existing GitHub CLI

configuration

2. **Comment Retrieval**:
    - Fetches PR metadata to determine repository and PR information
    - Makes API calls to retrieve different types of comments
    - Handles pagination and JSON parsing using `jq` utility
3. **Context Preservation**:
    - Retrieves code context for review comments using the `diff_hunk` field
    - Shows file paths and line numbers for specific comments
    - Preserves the hierarchical structure of comment threads and replies
4. **Formatted Output**:
    - Creates a well-structured, readable display of comments
    - Uses Markdown formatting for readability
    - Shows comments in a hierarchical, threaded view
5. **Progress Indication**:
    - Shows a progress message ("fetching PR comments") while the operation is in progress
    - Provides clear indication when no comments are found

Technical Implementation Notes

The `pr_comments` command demonstrates several sophisticated approaches:

1. **Prompt-Based Implementation**: Unlike UI commands, this command uses Claude itself to execute a complex sequence of operations through a carefully crafted prompt.
2. **GitHub CLI Utilization**: Leverages the GitHub CLI's capabilities to interact with GitHub's API, taking advantage of existing authentication and configuration.
3. **API Interaction Patterns**: Provides a detailed workflow for accessing and processing data from different GitHub API endpoints:

```
gh api /repos/{owner}/{repo}/issues/{number}/comments
gh api /repos/{owner}/{repo}/pulls/{number}/comments
```

4. **JSON Processing with JQ**: Uses the `jq` command-line JSON processor for parsing complex API responses.

5. **Complex Data Formatting**: Provides explicit formatting instructions to ensure consistent, readable output:

   ```
   - @author file.ts#line:
     ```diff
     [diff_hunk from the API response]
   ```

   > quoted comment text

   [any replies indented]

   ```
   ```

6. **Arguments Passthrough**: Allows users to provide additional arguments that are appended to the prompt, enabling refinement of the command's behavior.

User Experience Benefits

The `pr_comments` command addresses several important needs for developers:

1. **Context Switching Reduction**: Allows reviewing PR comments without leaving the terminal or switching to a browser.

2. **Comment Aggregation**: Brings together comments from different locations (PR-level and code-specific) in a single view.
   ```

3. **Reading Optimization**: Formats comments with proper context, improving readability compared to raw API responses.

4. **Workflow Integration**: Enables PR review activities to be part of the normal development workflow in the terminal.

5. **GitHub Integration**: Takes advantage of existing GitHub CLI authentication and configuration.

The `pr_comments` command exemplifies how Claude Code can leverage Claude's capabilities to implement complex workflows that would otherwise require significant custom code. By using a prompt-based approach, it achieves powerful GitHub integration with minimal implementation complexity.

release-notes Command

The `release-notes` command provides users with a simple way to view the changes and new features introduced in each version of Claude Code, helping them stay informed about updates.

Implementation

The command is implemented in `commands/release-notes.ts` as a type: 'local' command that formats and displays release notes from a constant:

```
import { MACRO } from "../constants/macros.js";
import type { Command } from "../commands";
import { RELEASE_NOTES } from "../constants/releaseNotes";

const releaseNotes: Command = {
  description: "Show release notes for the current or specified version",
  isEnabled: false,
  isHidden: false,
  name: "release-notes",
  userFacingName() {
    return "release-notes";
  },
  type: "local",
  async call(args) {
    const currentVersion = MACRO.VERSION;

    // If a specific version is requested, show that version's notes
    const requestedVersion = args ? args.trim() : currentVersion;

    // Get the requested version's notes
    const notes = RELEASE_NOTES[requestedVersion];

    if (!notes || notes.length === 0) {
      return `No release notes available for version
${requestedVersion}.`;
    }

    const header = `Release notes for version ${requestedVersion}:`;
    const formattedNotes = notes.map((note) => `• ${note}`).join("\n");

    return `${header}\n\n${formattedNotes}`;
  },
};

export default releaseNotes;
```

This command demonstrates a simple but effective approach to displaying versioned information, using a constant defined in `constants/releaseNotes.ts` as the data source.

Functionality

The `release-notes` command provides a straightforward information display:

1. **Version Selection**:
    - Defaults to showing notes for the current version
    - Allows explicit version selection through command arguments
    - Handles cases where notes are unavailable for a version
2. **Note Formatting**:
    - Displays a clear header indicating the version
    - Formats individual notes as bullet points
    - Presents notes in a readable, consistent format
3. **No Dependencies**:
    - Doesn't require external API calls or complex processing
    - Works entirely with data embedded in the application
4. **Error Handling**:
    - Provides a friendly message when notes aren't available
    - Gracefully handles empty or missing note arrays

Technical Implementation Notes

While simpler than many other commands, the `release-notes` command demonstrates several effective patterns:

1. **Constant-Based Data Store**: Uses a predefined constant for storing release notes data:

```
// Example from constants/releaseNotes.ts
export const RELEASE_NOTES: Record<string, string[]> = {
  "0.0.22": [
    "Added proxy configuration support via --proxy flag or
HTTP_PROXY environment variable",
    "Improved error handling for API connectivity issues",
    "Fixed issue with terminal output formatting in certain
environments",
  ],
  "0.0.21": [
    "Enhanced model selection interface with provider-specific
options",
    "Improved documentation and help text for common commands",
    "Fixed bug with conversation state persistence",
  ],
};
```

2. **Default Value Pattern**: Uses the current version as a default if no specific version is requested:

```
const currentVersion = MACRO.VERSION;
const requestedVersion = args ? args.trim() : currentVersion;
```

3. **Array Transformation**: Uses map and join for clean formatting of bullet points:

```
const formattedNotes = notes.map((note) => `• ${note}`).join("\n");
```

4. **Null-Safe Access**: Checks for undefined or empty notes before attempting to display them:

```
if (!notes || notes.length === 0) {
  return `No release notes available for version
${requestedVersion}.`;
}
```

5. **Command Availability Control**: The command is marked as disabled with `isEnabled: false`, suggesting it may be under development or available only in certain builds.

User Experience Benefits

Despite its simplicity, the `release-notes` command provides several valuable benefits:

1. **Update Awareness**: Helps users understand what has changed between versions.
2. **Feature Discovery**: Introduces users to new capabilities they might not otherwise notice.
3. **Version Verification**: Allows users to confirm which version they're currently using.
4. **Historical Context**: Provides access to past release information for reference.
5. **Simplified Format**: Presents notes in a clean, readable format without requiring browser access.

The `release-notes` command exemplifies how even simple commands can provide valuable functionality through clear organization and presentation of information. It serves as part of Claude Code's approach to transparency and user communication.

resume Command

The `resume` command allows users to continue previous conversations with Claude, providing a way to revisit and extend past interactions without losing context.

Implementation

The command is implemented in `commands/resume.tsx` as a type: 'local-jsx' command that renders a React component for selecting and resuming past conversations:

```
import * as React from "react";
import type { Command } from "../commands";
import { ResumeConversation } from "../screens/ResumeConversation";
import { render } from "ink";
import { CACHE_PATHS, loadLogList } from "../utils/log";

export default {
  type: "local-jsx",
  name: "resume",
  description: "[ANT-ONLY] Resume a previous conversation",
  isEnabled: true,
  isHidden: false,
  userFacingName() {
    return "resume";
  },
  async call(onDone, { options: { commands, tools, verbose } }) {
    const logs = await loadLogList(CACHE_PATHS.messages());
    render(
      <ResumeConversation
        commands={commands}
        context={{ unmount: onDone }}
        logs={logs}
        tools={tools}
        verbose={verbose}
      />
    );
    // This return is here for type only
    return null;
  },
} satisfies Command;
```

The command delegates to the `ResumeConversation` component in `screens/ResumeConversation.tsx`, passing the list of previous conversations and

necessary context.

Functionality

The `resume` command implements a sophisticated conversation history management system:

1. **Conversation Retrieval**:
    - Loads previously saved conversation logs from the cache directory
    - Uses the `loadLogList` utility to parse and organize log files
    - Presents a navigable list of past conversations
2. **Context Restoration**:
    - Allows users to select a specific past conversation to resume
    - Loads the complete message history for the selected conversation
    - Reinstates the context of the previous interaction
3. **Selection Interface**:
    - Provides a visual interface for browsing conversation history
    - Shows metadata about each conversation (date, duration, topic)
    - Enables keyboard navigation through the history list
4. **Seamless Transition**:
    - Integrates resumed conversations into the current CLI session
    - Passes necessary tooling and command context to the resumed conversation
    - Maintains configuration settings across resumed sessions

Technical Implementation Notes

The `resume` command demonstrates several sophisticated implementation patterns:

1. **Custom Rendering Approach**: Unlike most commands that return a React element, this command uses a direct render call:

```
render(
  <ResumeConversation
    commands={commands}
    context={{ unmount: onDone }}
    logs={logs}
    tools={tools}
    verbose={verbose}
  />
);
```

This approach gives it more direct control over the rendering lifecycle.

2. **Log Loading and Parsing**: Implements careful log handling with the loadLogList utility:

```
// From utils/log.ts
export async function loadLogList(directory: string):
Promise<LogFile[]> {
  // Reads log directory
  // Parses log files
  // Extracts metadata
  // Sorts by recency
  // Returns formatted log list
}
```

3. **Context Passing**: Passes the full command and tool context to ensure the resumed conversation has access to all capabilities:

```
<ResumeConversation
  commands={commands}
  context={{ unmount: onDone }}
  logs={logs}
  tools={tools}
  verbose={verbose}
/>
```

4. **Path Management**: Uses a centralized path management system for log files:

```
// From utils/log.ts
export const CACHE_PATHS = {
  messages: () => path.join(getConfigRoot(), "messages"),
  // Other path definitions...
};
```

5. **ANT-ONLY Flag**: Uses the `[ANT-ONLY]` prefix in the description, indicating it's a feature specific to internal Anthropic usage and potentially not available in all distributions.

User Experience Benefits

The `resume` command addresses several important user needs:

1. **Conversation Continuity**: Allows users to pick up where they left off in previous sessions.
2. **Context Preservation**: Maintains the full context of previous interactions, reducing repetition.
3. **Work Session Management**: Enables users to organize work across multiple sessions.
4. **History Access**: Provides a browsable interface to previous conversations.
5. **Interrupted Work Recovery**: Helps recover from interrupted work sessions or system crashes.

The `resume` command exemplifies Claude Code's approach to persistent user experience, ensuring that valuable conversation context isn't lost between sessions. This is particularly important for complex coding tasks that may span multiple work sessions.

review Command

The `review` command provides a specialized workflow for reviewing GitHub pull requests, leveraging Claude's code analysis capabilities to generate comprehensive code reviews.

Implementation

The command is implemented in `commands/review.ts` as a type: 'prompt' command that formulates a specific request to Claude:

```
import { Command } from "../commands";
import { BashTool } from "../tools/BashTool/BashTool";

export default {
  type: "prompt",
  name: "review",
  description: "Review a pull request",
  isEnabled: true,
  isHidden: false,
  progressMessage: "reviewing pull request",
  userFacingName() {
    return "review";
  },
  async getPromptForCommand(args) {
    return [
      {
        role: "user",
        content: [
          {
            type: "text",
            text: `
      You are an expert code reviewer. Follow these steps:

      1. If no PR number is provided in the args, use ${BashTool.name}
("gh pr list") to show open PRs
      2. If a PR number is provided, use ${BashTool.name}("gh pr view
<number>") to get PR details
      3. Use ${BashTool.name}("gh pr diff <number>") to get the diff
      4. Analyze the changes and provide a thorough code review that
includes:
         - Overview of what the PR does
         - Analysis of code quality and style
         - Specific suggestions for improvements
         - Any potential issues or risks

      Keep your review concise but thorough. Focus on:
      - Code correctness
      - Following project conventions
      - Performance implications
      - Test coverage
      - Security considerations

      Format your review with clear sections and bullet points.

      PR number: ${args}
    `,
          },
        ],
      },
    ];
  },
} satisfies Command;
```

Like the `pr_comments` command, the `review` command is a 'prompt' type command that formulates a specific request to Claude, instructing it to perform a complex sequence of operations using the GitHub CLI.

Functionality

The `review` command implements a comprehensive PR review workflow:

1. **PR Discovery and Selection**:
    - Lists open pull requests if no PR number is specified
    - Retrieves details for a specific PR when a number is provided
    - Presents options for users to select which PR to review
2. **Code Analysis**:
    - Fetches the PR diff to understand code changes
    - Analyzes code quality, style, and potential issues
    - Evaluates changes in the context of the project's conventions
3. **Review Generation**:
    - Creates a structured code review with logical sections
    - Highlights potential improvements and issues
    - Provides specific, actionable feedback on code changes
4. **Focus Areas**:
    - Analyzes code correctness and logical issues
    - Evaluates adherence to project conventions
    - Assesses performance implications of changes
    - Reviews test coverage of new code
    - Identifies potential security considerations
5. **Progress Indication**:
    - Shows a "reviewing pull request" message during analysis
    - Provides a clear visual indication that processing is underway

Technical Implementation Notes

The `review` command demonstrates several important implementation patterns:

1. **Tool Reference in Prompt**: Explicitly references the BashTool by name in the prompt to ensure proper tool usage:

   ```
   use ${BashTool.name}("gh pr list")
   ```

   This ensures the correct tool is used regardless of how Claude might interpret the instruction.

2. **GitHub CLI Integration**: Leverages the GitHub CLI's capabilities for PR interaction:

   ```
   gh pr list - Show open PRs
   gh pr view <number> - Get PR details
   gh pr diff <number> - Get the diff
   ```

3. **Comprehensive Analysis Instructions**: Provides Claude with a detailed framework for analysis:

   ```
   Analyze the changes and provide a thorough code review that
   includes:
   - Overview of what the PR does
   - Analysis of code quality and style
   - Specific suggestions for improvements
   - Any potential issues or risks
   ```

4. **Focus Guidance**: Directs Claude's analysis toward specific aspects of code quality:

```
Focus on:
- Code correctness
- Following project conventions
- Performance implications
- Test coverage
- Security considerations
```

5. **Formatting Guidance**: Ensures consistent output format with clear instructions:

```
Format your review with clear sections and bullet points.
```

User Experience Benefits

The `review` command provides several valuable benefits for developers:

1. **Code Review Automation**: Reduces the manual effort required for initial PR reviews.
2. **Consistent Review Quality**: Ensures all PRs receive a thorough analysis covering key areas.
3. **Learning Opportunity**: Exposes developers to alternative perspectives on their code.
4. **Workflow Integration**: Fits seamlessly into GitHub-based development workflows.
5. **Time Efficiency**: Quickly generates comprehensive reviews that serve as a starting point for human reviewers.

The `review` command exemplifies how Claude Code can leverage Claude's code analysis capabilities to add value to existing development workflows. It transforms Claude from a conversational assistant into an active participant in the code review process.

terminalSetup Command

The `terminalSetup` command enhances the user experience by configuring terminal-specific keyboard shortcuts, specifically implementing Shift+Enter for newlines in terminals like iTerm2 and VS Code.

Implementation

The command is implemented in `commands/terminalSetup.ts` as a type: 'local' command that configures terminal-specific settings:

```
import { Command } from "../commands";
import { EOL, platform, homedir } from "os";
import { execFileNoThrow } from "../utils/execFileNoThrow";
import chalk from "chalk";
import { getTheme } from "../utils/theme";
import { env } from "../utils/env";
import { getGlobalConfig, saveGlobalConfig } from "../utils/config";
import { markProjectOnboardingComplete } from "../ProjectOnboarding";
import { readFileSync, writeFileSync } from "fs";
import { join } from "path";
import { safeParseJSON } from "../utils/json";
import { logError } from "../utils/log";

const terminalSetup: Command = {
  type: "local",
  name: "terminal-setup",
  userFacingName() {
    return "terminal-setup";
  },
  description:
    "Install Shift+Enter key binding for newlines (iTerm2 and VSCode
only)",
  isEnabled:
    (platform() === "darwin" && env.terminal === "iTerm.app") ||
    env.terminal === "vscode",
  isHidden: false,
  async call() {
    let result = "";

    switch (env.terminal) {
      case "iTerm.app":
        result = await installBindingsForITerm2();
        break;
      case "vscode":
        result = installBindingsForVSCodeTerminal();
        break;
```

```
    }

    // Update global config to indicate Shift+Enter key binding is
installed
    const config = getGlobalConfig();
    config.shiftEnterKeyBindingInstalled = true;
    saveGlobalConfig(config);

    // Mark onboarding as complete
    markProjectOnboardingComplete();

    return result;
  },
};

export function isShiftEnterKeyBindingInstalled(): boolean {
  return getGlobalConfig().shiftEnterKeyBindingInstalled === true;
}

export default terminalSetup;
```

The command includes specialized implementation functions for different terminal types, with unique approaches for iTerm2 and VS Code.

Functionality

The `terminalSetup` command provides a targeted terminal enhancement:

1. **Terminal Detection**:
    - Determines the current terminal environment automatically
    - Supports iTerm2 on macOS and VS Code's integrated terminal
    - Only enables the command when in compatible terminals
2. **Keyboard Shortcut Configuration**:
    - Configures Shift+Enter to insert a newline without submitting input
    - Uses terminal-specific mechanisms for each supported terminal
    - For iTerm2, uses the defaults command to modify keybindings
    - For VS Code, modifies the keybindings.json configuration file
3. **Configuration Status Tracking**:
    - Updates global configuration to track installation state
    - Provides a utility function to check if bindings are installed

- Avoids redundant installations by checking state

4. **Onboarding Integration**:
   - Marks project onboarding as complete after setup
   - Integrates with the broader onboarding workflow
   - Provides a smooth setup experience for new users

Technical Implementation Notes

The `terminalSetup` command demonstrates several sophisticated technical approaches:

1. **Platform-Specific Implementations**: Implements different strategies based on the terminal type:

```
switch (env.terminal) {
  case "iTerm.app":
    result = await installBindingsForITerm2();
    break;
  case "vscode":
    result = installBindingsForVSCodeTerminal();
    break;
}
```

2. **macOS Defaults System**: For iTerm2, uses the macOS defaults system to modify key bindings:

```
async function installBindingsForITerm2(): Promise<string> {
  const { code } = await execFileNoThrow("defaults", [
    "write",
    "com.googlecode.iterm2",
    "GlobalKeyMap",
    "-dict-add",
    "0xd-0x20000-0x24",
    `<dict>
      <key>Text</key>
      <string>\\n</string>
      <key>Action</key>
      <integer>12</integer>
      <key>Version</key>
      <integer>1</integer>
      <key>Keycode</key>
      <integer>13</integer>
      <key>Modifiers</key>
      <integer>131072</integer>
    </dict>`,
  ]);
  // Error handling and success message...
}
```

3. **VS Code Configuration File Handling**: For VS Code, directly modifies the keybindings.json file:

```
function installBindingsForVSCodeTerminal(): string {
  const vscodeKeybindingsPath = join(
    homedir(),
    platform() === "win32"
      ? join("AppData", "Roaming", "Code", "User")
      : platform() === "darwin"
      ? join("Library", "Application Support", "Code", "User")
      : join(".config", "Code", "User"),
    "keybindings.json"
  );

  try {
    const content = readFileSync(vscodeKeybindingsPath, "utf-8");
    const keybindings: VSCodeKeybinding[] =
      (safeParseJSON(content) as VSCodeKeybinding[]) ?? [];

    // Check for existing bindings...
    // Add new binding...
    // Write updated file...
  } catch (e) {
    // Error handling...
  }
}
```

4. **Cross-Platform Path Handling**: Uses the path.join utility along with platform detection to handle OS-specific paths:

```
const vscodeKeybindingsPath = join(
  homedir(),
  platform() === "win32"
    ? join("AppData", "Roaming", "Code", "User")
    : platform() === "darwin"
    ? join("Library", "Application Support", "Code", "User")
    : join(".config", "Code", "User"),
  "keybindings.json"
);
```

5. **Safe JSON Parsing**: Uses a utility function for safe JSON parsing to handle potential errors:

```
const keybindings: VSCodeKeybinding[] =
  (safeParseJSON(content) as VSCodeKeybinding[]) ?? [];
```

User Experience Benefits

The `terminalSetup` command addresses a specific pain point in terminal interaction:

1. **Improved Input Experience**: Allows multi-line input with Shift+Enter without submitting prematurely.
2. **Workflow Enhancement**: Makes it easier to compose complex prompts or code snippets.
3. **Consistency Across Environments**: Provides similar behavior in different terminal environments.
4. **Seamless Integration**: Configures the terminal without requiring users to understand terminal-specific configuration files.
5. **Visual Feedback**: Provides clear success messages when binding installation completes.

The `terminalSetup` command exemplifies Claude Code's attention to detail in the user experience, addressing a specific friction point in terminal interaction to create a more seamless interaction pattern for complex inputs.