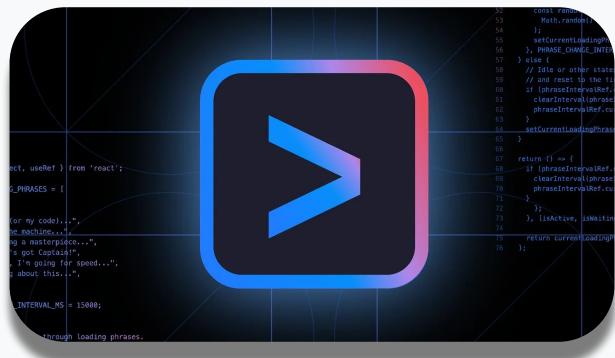


Under the Hood of Gemini CLI: A Technical Architecture Deep Dive



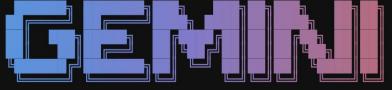
Contents

- 01 Overview**
- 02 High-Level Architecture**
- 03 Core Mechanics**
- 04 Extensibility**

Overview

Gemini CLI is an Open Source, terminal-based AI agent that is grounded in your local development environment.

- Uses a ReAct (Reason and Act) loop to perform complex coding tasks, debug, and interact with local files.
- It has built-in tools that allow it to read/modify your local files, execute shell commands, search the web, and more.
- It sees your code, understands your project structure, and helps you execute complex tasks without breaking your flow.



Tips for getting started:
1. `/help` for more information.
2. Ask coding questions, edit code or run commands.
3. Be specific for the best results.

```
> write a short python snippet with a brief explainer (15 lines max!) that shows how cool Gemini is
+ Okay, here's a Python snippet and a concise explainer for your slide:

Python Snippet:

# Python: Quickly summarize text
def summarize(text, max_words=10):
    return " ".join(text.split()[:max_words]) + "..."

my_article = "Gemini is a family of multimodal AI models..." # (Imagine full article)
print(summarize(my_article))
# Output: Gemini is a family of multimodal AI models developed by...

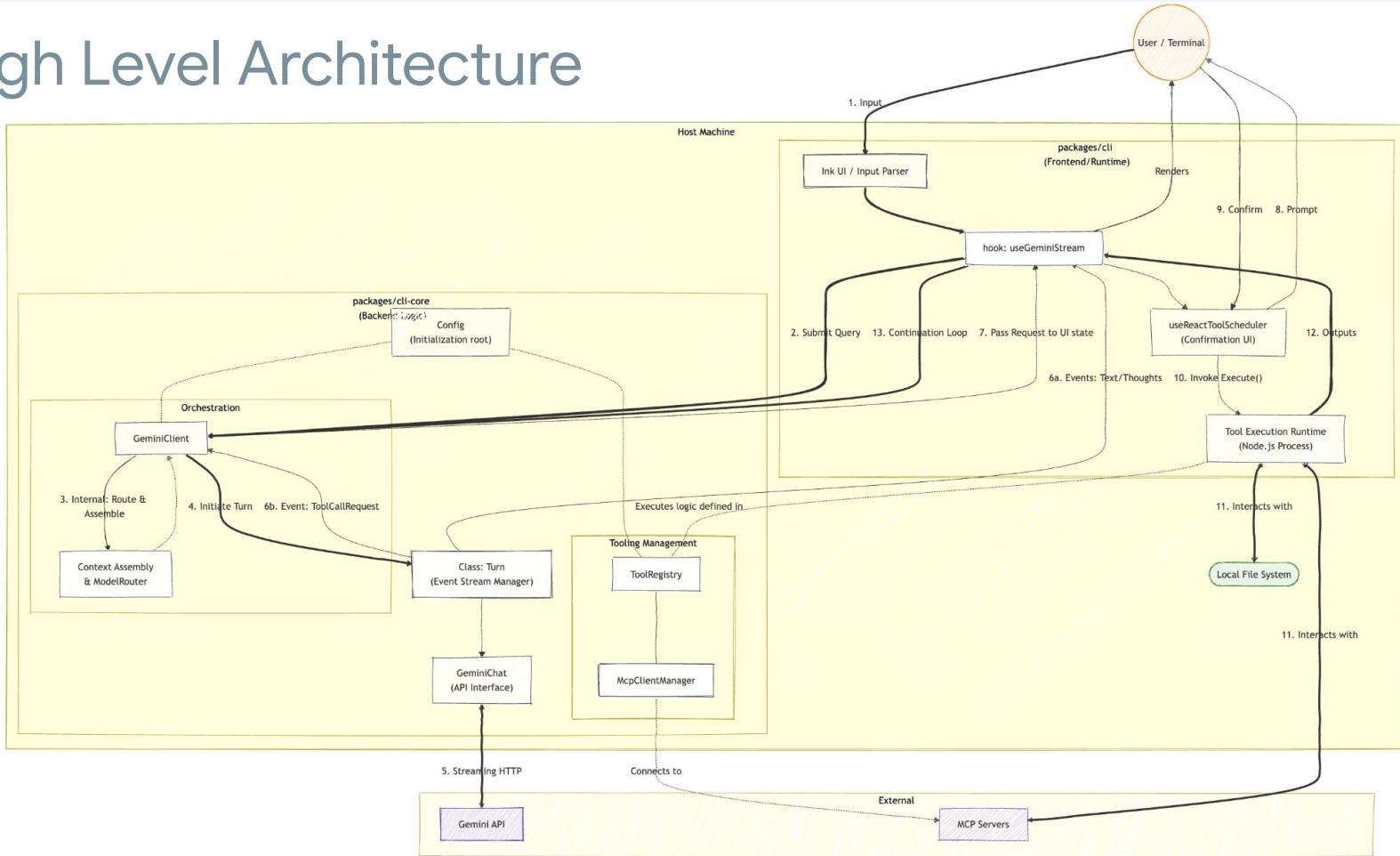
Explainer: Gemini - Your AI Coding Partner! (10 lines)

This Python snippet gives a basic text summary.
With Gemini, you could:
1. Ask: "Write Python to summarize text." -> Get this code.
2. Ask: "Explain this to me." -> Understand it easily.
3. Ask: "Make it summarize by sentences instead." -> Refine it.
4. Ask: "Convert this to JavaScript." -> Translate it.
Gemini helps you write, understand, and improve code, fast!
It's like having a brilliant pair programmer, 24/7.

cwd: /usr/local/google/home/cperry/gemini-code
> Enter your message or use tools (e.g., @src/file.txt)...
```

? for shortcuts no sandbox (see README) gemini-2.5-pro-preview-05-06 | CLI 0.1.0-1245fe4.0

High Level Architecture



Monorepo Structure

packages/cli

Frontend

- Ink/React UI, User Input/Parsing, Markdown Rendering
- Tool Confirmation (Human-in-the-Loop)
- Processes and allows interaction via slash commands

packages/core

Backend

- Hosts the majority of the business logic pertaining to the agent loop
- Context Assembly, Model Routing, Compression
- Interacts with the Gemini models

Key Classes for Agent Loop

The main agent's loop is an event-driven, ReAct-style flow that spans across the CLI and Core packages.

`useGeminiStream` (CLI)

-  **Location:** `src/ui/hooks/useGeminiStream.ts`
-  **Role: UI & State Driver**
- Drives the React UI.
- Orchestrates the end-to-end flow.
- Consumes events from `Turn` to render output.

`GeminiClient` (Core)

-  **Location:** `src/core/client.ts`
-  **Role: Core Orchestrator**
- Main entry point into the `core` package.
- Assembles context and routes requests.
- Initiates a new `Turn` for each interaction.

`Turn` (Core)

-  **Location:** `packages/core/src/core/turn.ts`
-  **Role: Interaction Manager**
- Manages a single logical "turn".
- Handles the streaming response from the model.
- Yields structured events (content, thoughts, tool calls).

`GeminiChat` (Core)

-  **Location:** `src/core/geminiChat.ts`
-  **Role: API Layer**
- Manages low-level Gemini API communication.
- Processes the raw data stream.
- Maintains conversation history.

The Flow

1. **Input:** User provides a prompt in the CLI. `useGeminiStream` processes it.



2. **Routing & Context:** `GeminiClient` selects a model and gathers context.



3. **API Call:** `Turn` calls `GeminiChat` to send the request to Gemini.



4. **Stream Processing:** `Turn` yields events based on the model's response.

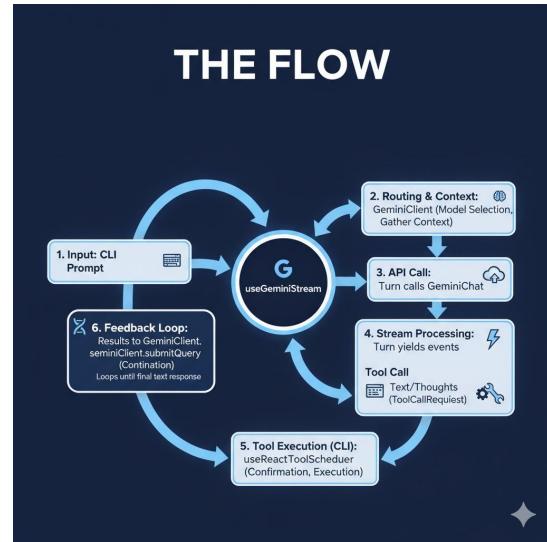
- **Text/Thoughts:** Displayed to the user via the CLI.
- **Tool Call:** `Turn` yields a `ToolCallRequest` event.



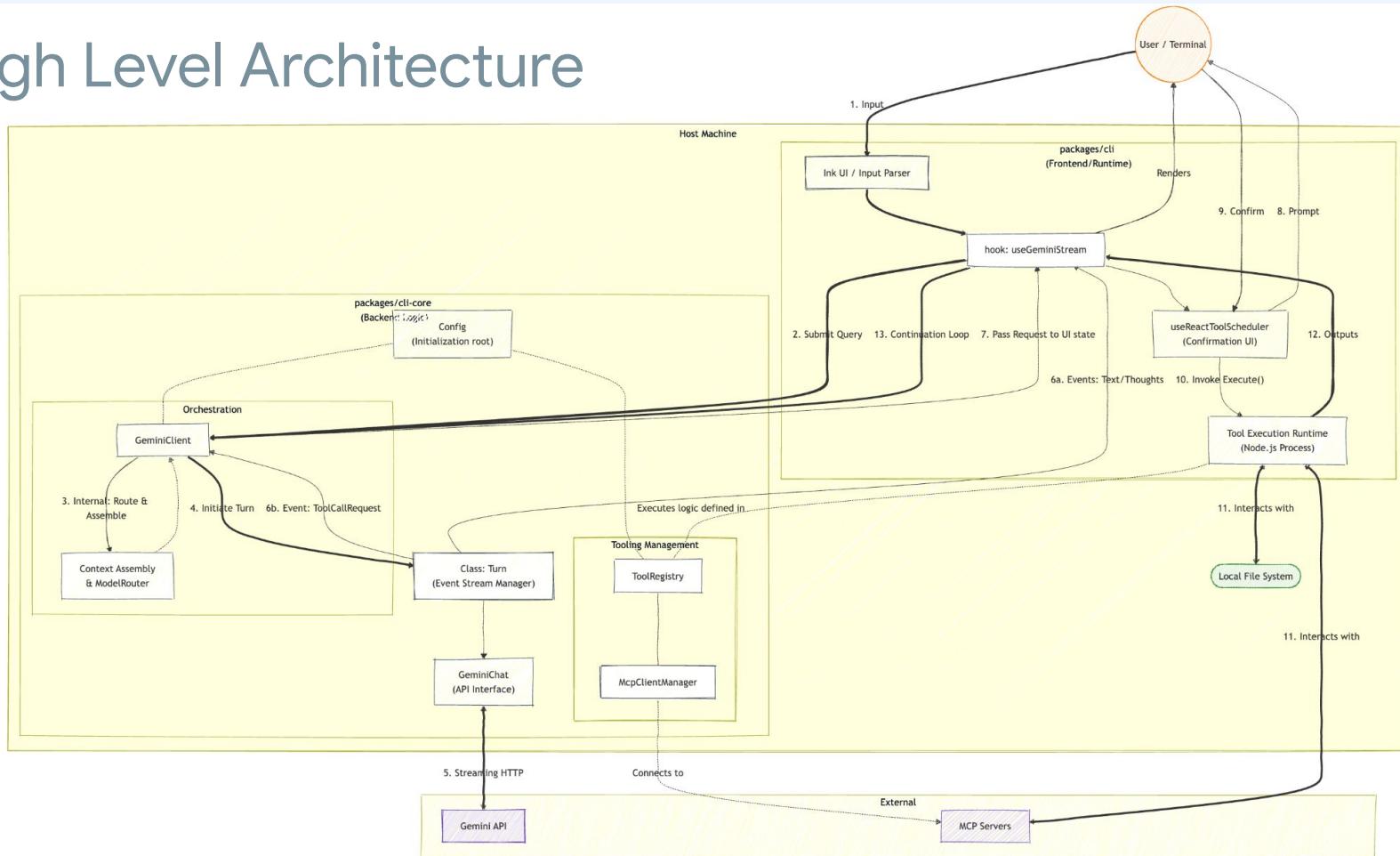
5. **Tool Execution (CLI):** `useGeminiStream` receives the request. It uses `useReactToolScheduler` to handle user confirmation (if needed) and execute the tool.



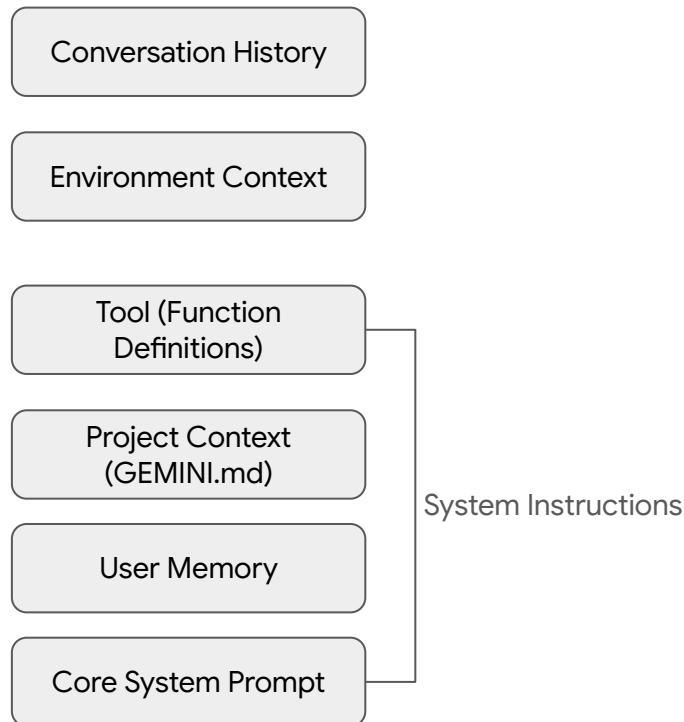
6. **Feedback Loop:** Once tools are executed, `useGeminiStream` feeds the results back to `GeminiClient.submitQuery` as a continuation, triggering **Step 3** again. This loops until the model provides a final text response.



High Level Architecture



Context Management - The Stack



1. **Core System Prompt:** Persona, operational mandates, and git state.
2. **User Memory:** Long-term facts saved via `save_memory` tool.
3. **Project Context:** Hierarchical instructions from GEMINI.md files.
4. **Tool Definitions:** From Built-in tools, MCP tools, Subagents, etc.
5. **Env Context:** OS details, CWD
6. **History:** Previous turns managed by GeminiChat.

Context Mechanics

GEMINI.md Discovery

Global (~/.gemini/)



Root → Up to CWD



CWD → Downward

Assembly: Files are concatenated with clear markers.

Features: Supports @ import directives for modular context.

Compression

Tracking: `UiTelemetryService` monitors `usageMetadata` from API.

The Trigger: Context > 70% window size.

The Strategy: Summarization.

Before: [Oldest History] ... [Recent History]

After: [XML Summary of Old] + [Recent History]

Tooling & Model Context Protocol (MCP)

Unified Tooling Architecture

ToolRegistry: The central repository and single source of truth for all tools provided to the LLM.

Code Structure: Separation of schema and logic.

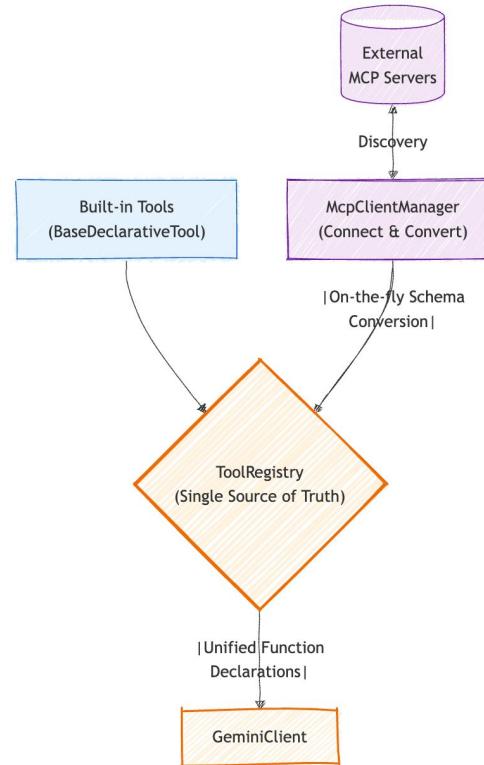
- **BaseDeclarativeTool:** Defines the JSON schema (inputs/outputs) for the model.
- **BaseToolInvocation:** Contains the actual Node.js execution logic.

MCP Integration (External Tools)

Role: Gemini CLI acts as an MCP Client.

McpClientManager: Manages connections to external MCP servers and tool discovery.

Dynamic Conversion: Converts MCP tool definitions into Gemini-compatible function declarations on the fly before registering them in the **ToolRegistry**.



Security: Human-in-the-Loop

To safely execute actions, a callback-driven mechanism bridges the Core logic and the CLI UI.

CoreToolScheduler (Core): Manages tool lifecycle and determines if confirmation is needed via `shouldConfirmExecute()`.

useReactToolScheduler (CLI): React hook that syncs Core state with UI state via callbacks.

ToolConfirmationMessage (CLI): The UI component rendered to prompt the user.

Steps

1. **Determination:** `CoreToolScheduler` checks if a tool requires confirmation via `shouldConfirmExecute()`.
2. **State Update:** If confirmation is needed, the tool's state is set to `awaiting_approval`.
3. **Notification:** An `onToolCallsUpdate` callback notifies `useReactToolScheduler` in the CLI of the state change.
4. **Prompt:** The CLI renders `ToolConfirmationMessage`, presenting the user with options (Approve, Deny, Always Allow).
5. **Response:** The user's selection triggers an `onConfirm` callback, which calls back into `CoreToolScheduler`.
6. **Execution:** `CoreToolScheduler` updates the state to `scheduled` (to run) or `cancelled` based on the user's choice.

Coming soon → Policy Engine

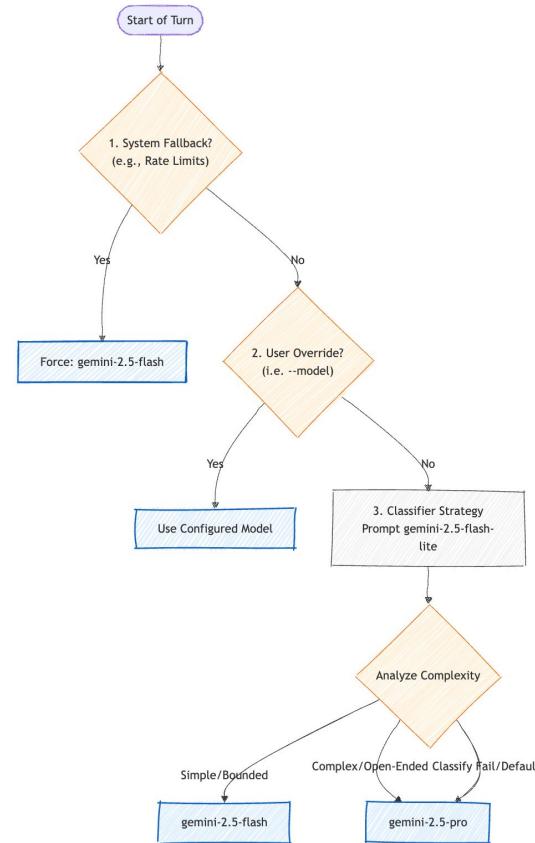
Intelligent Model Routing ('auto' mode)

Goal: Dynamically balance performance and cost per interaction.

Key Behaviors:

- **Timing:** Routing occurs exactly once at the beginning of a user turn inside `GeminiClient`.
- **Stickiness:** The selected model is locked for the entire duration of the turn.

Why? All subsequent autonomous tool calls and reasoning steps use the same model to ensure context consistency.



Subagent Architecture (“Agents as Tools”)

Concept: Hierarchical Delegation

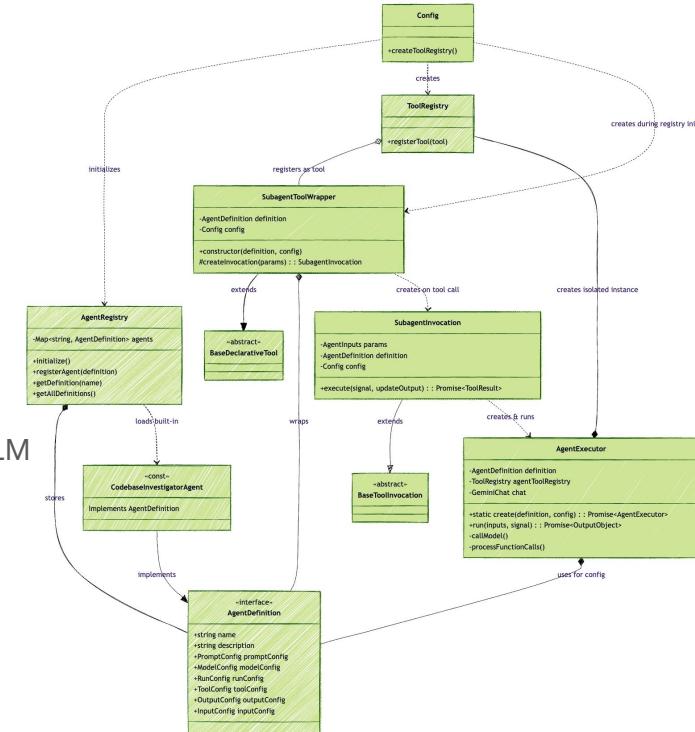
Instead of a monolithic context, we use a fractal architecture. Specialized agents are registered as tools.

- **Benefit:** Main agent delegates complex tasks to an isolated sandbox.
- **Context Hygiene:** Prevents polluting the main context window with intermediate steps of a sub-task.

Technical Implementation

- **AgentDefinition (Config):** Defines persona, restricted toolset (read-only), and Zod Input/Output schemas.
- **SubagentToolWrapper (The Bridge):** Dynamically converts the definition into a standard **DeclarativeTool** format that the main LLM can understand and call.
- **AgentExecutor (The Runtime):**
- Spins up an isolated ReAct loop with its own **GeminiChat** and **ToolRegistry**.
- Streams "thoughts" to UI for observability.

Crucially: Discards internal history and returns only the final, synthesized output to the main agent.



Subagent Use Case: The codebase_investigator

Status: Currently the sole built-in sub-agent. Designed to help the main agent build a mental model of the codebase.

Configuration profile:

- Role: Deep-dive file exploration and summarization.
- Tool Constraints: Restricted to non-interactive, read-only tools:
 - `ls`, `grep`, `glob`, `read_file`
 - Cannot edit files or run commands.
- Lifecycle: Runs until `complete_task` is called or constraints (time/turns) are met.

Structured Output (JSON):

Returns a structured report to the main agent containing:

1. High-level summary of findings.
2. Trace of exploration steps.
3. List of relevant file paths discovered.



Extensibility Architecture

Modifying CLI behavior without altering core code.

1. Anatomy of an Extension

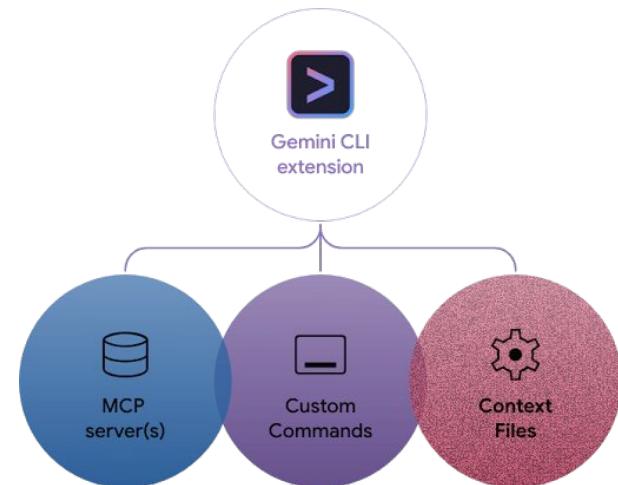
An extension is a directory defined by a `gemini-extension.json` manifest.

It bundles:

- **Custom Commands:** TOML files in `commands/` define slash commands (pre-packaged prompts).
- **Context:** `GEMINI.md` files appended to the system prompt to steer behavior.
- **Tools (MCP):** Configurations to connect to external MCP servers, instantly adding new capabilities.

2. Lifecycle Management

- **Develop:** Scaffold via `gemini extensions new`. Use link for local development.
- **Install:** Supports local paths or Git URLs. Installed to `~/.gemini/extensions`.
- **Runtime:** At startup, enabled extensions are discovered, and their configurations are merged into the main CLI config.
- **Manage:** Full CLI suite to list, enable, disable, and update.



Architecture Summary: Key Takeaways

Foundation & Flow

- **Decoupled Architecture:** Clean separation of concerns between the UI (packages/cli) and Core logic (packages/cli-core).
- **Event-Driven ReAct Loop:** A streaming, stateful loop handles complex, multi-step agentic tasks with human-in-the-loop security.

Context & Intelligence

- **Hierarchical Context:** Structured stacking of system prompts, user memory, project files ([GEMINI.md](#)), etc.
- **Optimization:** Automatic context compression and intelligent "auto" model routing balance performance and cost.

Tooling & Scaling

- **Unified Interface:** A single registry for built-in tools and external MCP servers via on-the-fly schema conversion.
- **Fractal Agents:** "Agents as Tools" architecture allows delegating complex tasks to isolated sub-agents (e.g., codebase_investigator) to maintain context hygiene.

Extensibility

- **Standardized Packaging:** Extensions bundle custom commands, system prompts, and MCP configurations to modify CLI behavior without altering core code.

Questions?



Thank you

Google



Experimental Features

You can take a look at any pending experimental features that may not be defaulted to true.

<https://github.com/google-gemini/gemini-cli/blob/main/packages/cli/src/config/settingsSchema.ts>