Architectural Convergence: Orchestrating Multi-Agent AI Workflows on Windows 11 via WSL2

1. Introduction: The Paradigm Shift to Poly-Agentic Development

The software development lifecycle is currently undergoing its most significant transformation since the advent of distributed version control. We are transitioning from a tooling era defined by static Integrated Development Environments (IDEs) and manual command-line execution to an era of "Agentic AI." In this new paradigm, the developer functions less as a manual operator of syntax and more as an architect of intent, orchestrating a suite of specialized Artificial Intelligence (AI) agents. These agents—powered by Large Language Models (LLMs) such as Google's Gemini, Anthropic's Claude, and OpenAI's GPT-series—are no longer confined to browser-based chat windows. They have migrated to the terminal, the native habitat of the engineer, where they possess the agency to execute shell commands, manipulate files systems, and interact with version control directly.

For the professional developer operating within the Microsoft ecosystem, Windows 11 serves as the host OS, but the execution environment of choice is undeniably the Windows Subsystem for Linux (WSL2). This architecture provides a unique hybrid advantage: the hardware support and user experience of Windows combined with the kernel-level compatibility and tooling ecosystem of Linux, which is required for the majority of modern AI agent runtimes.[1]

However, the proliferation of these tools introduces a new challenge: fragmentation. A developer may wish to utilize Gemini CLI for its massive context window to understand a legacy codebase, Claude Code for its superior reasoning capabilities during complex refactoring, and a Codex-based CLI for its rigorous "diff-review" application methodology. Constructing a workflow that allows these distinct agents to operate simultaneously, share context, and utilize a unified set of external tools requires a sophisticated architectural approach. This report provides an exhaustive technical analysis of building such a multi-CLI workflow. We will explore the optimization of the WSL2 virtualization layer, the detailed configuration of the "Agentic Triad" (Gemini, Claude, Codex), and the implementation of the Model Context Protocol (MCP)—the universal standard that binds these disparate intelligences into a cohesive engineering unit.

2. Infrastructure Layer: Optimizing Windows 11 and

WSL2

The foundation of a high-velocity multi-CLI workflow is the virtualization layer. While modern AI command-line tools often support Windows natively via PowerShell or Command Prompt, the ecosystem of Model Context Protocol (MCP) servers, Node.js-based utilities, and Python-driven analysis tools thrives in a Unix-like environment. WSL2 offers full system call compatibility and near-native performance, which is critical for agents performing heavy file I/O operations (such as indexing a repository for semantic search) or running Dockerized sandboxes for code execution.[1]

2.1 The Virtualization Architecture

WSL2 fundamentally differs from its predecessor by utilizing a lightweight utility virtual machine (VM) running a genuine Linux kernel. This architecture allows for full system call compatibility, which is essential for AI agents that may need to spawn subprocesses, manage memory allocation for large context loading, or interact with low-level networking interfaces.

Distribution Selection Strategy:
For this specific workflow, Ubuntu 24.04 LTS or 22.04 LTS is the recommended distribution. The rationale lies in the package repository support: the AI CLIs in question (Gemini, Claude Code) rely heavily on recent versions of Node.js (v18+) and Python (v3.10+). Ubuntu's repositories and community support for these runtimes are unmatched, minimizing the friction often encountered with library dependencies on other distributions like Alpine or Debian Stable.1

2.2 Resource Allocation and Kernel Tuning

A common failure mode in multi-agent workflows is resource exhaustion. AI agents are memory-intensive; they often load significant portions of the project context into memory, spawn multiple subprocesses (e.g., ripgrep for code search, headless browsers for web testing), and maintain persistent connections to MCP servers. If left unmanaged, a concurrent session involving Gemini CLI analyzing a codebase and Claude Code refactoring a module can trigger the Windows host's Out-Of-Memory (OOM) killer, potentially destabilizing the entire system.

To mitigate this, the .wslconfig file (located in the Windows user profile directory %UserProfile%) must be tuned to define strict resource boundaries.

Memory Reservation:
A high-performance workflow typically requires reserving at least 50% of the host system's RAM for the Linux kernel. For a developer workstation with 32GB of RAM, allocating 16GB to WSL2 is a prudent baseline. This ensures that the Linux kernel has sufficient page cache to handle the rapid file reads performed by agents during context gathering.2
Swap Configuration:
The default swap configuration in WSL2 can be conservative. Node.js applications—which

underpin both Gemini CLI and Claude Code—can crash with "Heap Limit" errors when processing massive JSON objects or large text files. Configuring a substantial swap file acts as a safety net.
Processor Scheduling:
Compiling code (often triggered by agents running tests) and indexing repositories are CPU-intensive tasks. Assigning specific processors or ensuring all cores are available allows the Linux scheduler to balance the load of multiple active agents effectively.
Proposed .wslconfig Configuration:

Ini, TOML

```
[wsl2]
memory=16GB        # Critical for concurrent agent execution
processors=8       # Dedicated cores for compilation/indexing tasks
swap=12GB          # Essential for Node.js heap overflows during large context loading
localhostForwarding=true # Enables seamless port access from Windows browsers
kernelCommandLine=sysctl.vm.max_map_count=262144 # Required for Elasticsearch/database containers
```

Table 1: Critical .wslconfig Parameters for AI Workflows

Parameter	Function	Implication for AI Agents
memory	Limits VM RAM usage	Prevents agents from crashing the Windows UI during deep repository analysis.
processors	Limits CPU Core usage	Ensures background indexing (ripgrep) doesn't freeze the host machine.
swap	Virtual Memory file size	Prevents Node.js crashes when context windows exceed physical RAM allocation.
localhostForwarding	Network bridging	Allows Windows browsers

		to access web apps launched by agents inside WSL.

2.3 The Runtime Environment: Managing Node.js and Python

The majority of modern AI CLIs are distributed as NPM packages. A critical failure point in multi-CLI setups is version mismatching and permission conflicts. Gemini CLI and Claude Code generally require Node.js 18.0 or higher.[1]

The Pitfalls of System Package Managers:
Direct installation via apt (e.g., sudo apt install nodejs) is strongly discouraged for this workflow. System package managers often lag behind the rapid release cycles of Node.js. Furthermore, installing global packages with sudo creates permission ownership issues in the /usr/lib/node_modules directory. When an AI agent attempts to self-update or install a required MCP server, it will fail due to lack of write permissions, breaking the automated workflow.1
Node Version Manager (NVM) Strategy:
The use of Node Version Manager (nvm) allows for user-space installation. This decouples the Node.js runtime from the system root, allowing global packages to be installed without sudo. This ensures that agents have full control over their own update cycles and configuration files.
Python Environment Management:
Python is essential for specific MCP servers (e.g., local analysis tools, specialized data fetchers). The traditional pip workflow is increasingly being replaced by uv, a high-performance Python package installer and resolver written in Rust. uv is recommended for this workflow due to its speed, particularly when agents need to dynamically install dependencies in a sandboxed environment to execute a Python script generated during a session.6

2.4 Terminal Emulation and Shell Integration

The interface through which the developer interacts with these agents is the terminal emulator. On Windows 11, **Windows Terminal** is the gold standard, supporting GPU acceleration for text rendering—a subtle but important feature when AI agents output large streams of text or complex TUI (Text User Interface) elements like diff tables.

Shell Configuration (Zsh/Fish):
While Bash is the default, Zsh (with Oh My Zsh) or Fish are preferred for their superior autocompletion capabilities. AI CLIs are complex, with numerous flags and subcommands.

- **Gemini CLI Autocompletion:** Support for shell autocompletion is actively being developed to improve command discoverability.[7]
- **Claude Code Autocompletion:** Community scripts exist to enable tab-completion for Claude's flags, essential for managing long context-window commands.[8]

- **Fish Shell Integration:** For users of the Fish shell, specific functions can be defined to handle the "dangerously-skip-permissions" flags often used in autonomous modes, effectively creating short aliases for high-velocity coding sessions.[9]

3. The Agentic Triad: Installation, Configuration, and Architecture

A robust, multi-CLI workflow does not rely on a single model. Instead, it leverages the unique strengths of different architectures: **Google Gemini** for its massive context window and speed; **Anthropic Claude** for its high-reasoning capabilities in code generation and refactoring; and **OpenAI Codex** (via wrappers) for logic tasks and architectural planning.

3.1 Google Gemini CLI: The Context Specialist

The Gemini CLI brings the power of Gemini 1.5 Pro and 2.5 models directly to the terminal. Its defining characteristic is the 1-million-token context window (extensible to 2M in some previews), allowing it to ingest entire repositories, documentation sets, and log files in a single prompt.[10]

Architecture and The ReAct Loop:
Gemini CLI utilizes a "Reason and Act" (ReAct) loop. Upon receiving a prompt, it does not merely generate text; it evaluates which tools (local or remote) are necessary to fulfill the request. It might decide to search the web using Google Search integration, read a file using cat, or query an MCP server. This loop continues until the task is complete or user input is required.12
Installation Protocol:
The Gemini CLI is distributed via the Google registry on NPM.

```
Bash
```

```
npm install -g @google/gemini-cli
```

Verification involves running gemini --version to ensure the binary is correctly linked in the user's path.[1]

Authentication Models:

- **Google OAuth:** The simplest method for individual developers. It uses the developer's personal Google account quotas.
- **Vertex AI / API Keys:** For enterprise workflows or heavy automation, authenticating via a Google Cloud Project (Vertex AI) is recommended to access higher rate limits and

enterprise-grade data privacy guarantees.[11]

Configuration Hierarchy (settings.json & GEMINI.md):
Gemini utilizes a hierarchical configuration system that allows for granular control over agent behavior.

- **GEMINI.md:** This file acts as a persistent "system prompt" or context file. The CLI looks for this file in the current directory, and recursively up the directory tree.
  - *Global Context:* ~/.gemini/GEMINI.md can contain universal rules (e.g., "Always output JSON").
  - *Project Context:* A GEMINI.md at the root of a repository can contain architectural diagrams or coding style guides.
- **settings.json:** Located in ~/.gemini/, this file controls the technical configuration, including the registry of MCP servers and tool permission policies.[12]

Operational Modes:
Gemini introduces the concept of "Yolo Mode" (enabled via geminicodeassist.agentYoloMode). In standard operation, the agent asks for permission before executing sensitive tools (like writing files or accessing the network). Yolo Mode bypasses these checks, enabling fully autonomous refactoring loops. While powerful, this presents significant security risks and should only be used in sandboxed environments or strictly version-controlled repositories where git reset is an option.13

3.2 Anthropic Claude Code: The Reasoning Engine

Claude Code is positioned not just as a chatbot, but as an "AI Pair Programmer." Powered by the Claude 3.5 Sonnet and Opus models, it excels in complex logical reasoning, refactoring, and maintaining deep situational awareness of the project state.[4]

The TUI Experience:
Unlike standard CLIs that output streams of text, Claude Code offers a rich Text User Interface (TUI). When it proposes a code change, it presents a unified diff view directly in the terminal, allowing the developer to review, accept, or reject changes interactively. This "human-in-the-loop" design is central to its philosophy.15
Installation and Setup:
Claude Code is also an NPM-based tool but has stricter system dependencies. It requires ripgrep for efficient file searching and context gathering.

Bash

```
sudo apt install ripgrep
npm install -g @anthropic-ai/claude-code
claude login
```

The login process authenticates against an Anthropic Console account or a Pro subscription.[5]

Context Management (CLAUDE.md):
Similar to Gemini, Claude Code utilizes a CLAUDE.md file. This file acts as the project's memory, storing architectural notes, common testing commands, and style preferences. The /init command can analyze a project and auto-generate a baseline CLAUDE.md, bootstrapping the agent's understanding of the codebase.15
Slash Commands and Workflow:
Claude Code creates a command-line REPL (Read-Eval-Print Loop).

- /compact: Summarizes the conversation history to free up context tokens, a critical feature for long debugging sessions.
- /bug: Initiates a diagnostic loop to identify errors in the current context.
- !: Allows the user to execute shell commands directly within the Claude session (e.g., !npm test), the output of which is immediately fed back into Claude's context.[15]

3.3 Codex CLI: The Customizable Wrapper

While "Codex" serves as the underlying model family for GitHub Copilot, in the context of open-source CLI tools, "Codex CLI" refers to community-driven wrappers (such as those by Vladimir Siedykh) that bring OpenAI's models (GPT-4, o1) to the terminal. These tools are often highly configurable and cater to power users who want granular control over the model parameters.[18]

Configuration via TOML:
Distinct from the JSON-centric configuration of Gemini and Claude, Codex CLI implementations often utilize TOML (Tom's Obvious Minimal Language).

- **Config File:** ~/.codex/config.toml.
- **Granularity:** This file allows users to define specific model versions (e.g., pinning to a specific snapshot of gpt-4), set "temperature" controls for determinism, and define approval policies (approval_policy = "on-request") for tool execution.[20]

The Diff-First Philosophy:
Codex CLI tools often emphasize a "diff-first" workflow. Rather than generating a whole file, they generate a patch. This aligns closely with professional software engineering practices where code review is mandatory. The CLI acts as the submitter of the patch, and the developer acts as the reviewer.18

3.4 Comparative Feature Analysis

To assist in selecting the right tool for a specific task within the workflow, the following comparison highlights the architectural differences.

Table 2: Architectural Comparison of AI CLIs

Feature	Gemini CLI	Claude Code	Codex CLI
Primary Strength	Massive Context (1M+ Tokens)	Reasoning & Refactoring	TUI & Diff-Review
Config Format	settings.json	settings.json	config.toml
Context File	GEMINI.md	CLAUDE.md	AGENTS.md
Auth Model	Google OAuth / Vertex AI	Pro Sub / Console API	OpenAI API Key
Native Tooling	Google Search, Web Fetch	Bash Execution, File Edit	Shell Execution
WSL2 Support	Excellent (Native Node)	Excellent (Native Node)	Excellent (Native Node)
Cost Model	Free Tier / Pay-as-you-go	Subscription / Token-based	Pay-as-you-go

4. The Connectivity Layer: Model Context Protocol (MCP)

The challenge in a multi-CLI environment is tooling fragmentation. Historically, connecting Gemini to a PostgreSQL database required a proprietary plugin, while connecting Claude to the same database required a completely different integration. The **Model Context Protocol (MCP)**, introduced by Anthropic and rapidly adopted by Google and others, solves this by standardizing the connection between AI models (Hosts) and data sources/tools (Servers). This is the glue that makes a multi-CLI workflow viable.[22]

4.1 MCP Architecture and Mechanics

The MCP operates on a standardized Client-Host-Server architecture, utilizing JSON-RPC 2.0 as the transport protocol.

1. **MCP Host:** The AI application (e.g., Claude Code, Gemini CLI, VS Code).
2. **MCP Client:** The internal component of the Host that implements the MCP protocol standards.

3. **MCP Server:** An external program that exposes capabilities to the client.

Server Capabilities:
MCP servers expose three primary primitives:

- **Resources:** Read-only data sources (e.g., database schemas, log files, API documentation).
- **Tools:** Executable functions that can perform actions (e.g., execute_sql_query, restart_service).
- **Prompts:** Pre-defined interaction templates (e.g., "Analyze this error log").[24]

Transport Mechanisms:

- **Stdio (Standard Input/Output):** The default for local development on WSL2. The Host spawns the Server as a subprocess and communicates via stdin/stdout. This is secure, zero-latency, and requires no networking configuration.
- **SSE (Server-Sent Events) / HTTP:** Used for remote servers. The Host connects to a URL. This decouples the server execution from the host, allowing for Dockerized or cloud-hosted MCP servers.[23]

4.2 Configuring MCP Servers in a Multi-CLI Environment

The true power of this workflow is realized when multiple CLIs share the same set of MCP servers. However, each CLI currently maintains its own configuration file, leading to the "N x M" configuration problem.

Gemini CLI Configuration:
Gemini defines servers in ~/.gemini/settings.json under the mcpServers key.

JSON

```
{
  "mcpServers": {
    "filesystem": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-filesystem", "/path/to/project"]
    },
    "github": {
      "httpUrl": "https://api.githubcopilot.com/mcp/",
      "headers": { "Authorization": "token..." }
    }
  }
}
```

12

Claude Code Configuration:
Claude utilizes ~/.claude/settings.json (global) or .claude/settings.json (project-specific).

JSON

```
{
  "mcpServers": {
    "filesystem": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-filesystem", "/path/to/project"]
    }
  }
}
```

28

Codex CLI Configuration:
Codex typically uses TOML syntax in ~/.codex/config.toml.

Ini, TOML

```
[mcp_servers.filesystem]
command = "npx"
args = ["-y", "@modelcontextprotocol/server-filesystem", "/path/to/project"]
```

20

4.3 Strategic Insight: The "Write Once, Run Everywhere" Capability

The standardization of MCP means a developer can write a custom Python script using the FastMCP library that queries internal company documentation or a legacy database. By exposing this script as an MCP server, Gemini can use it to answer architectural questions, Claude can use it to generate code based on the retrieved docs, and Codex can use it to validate compliance—all interacting with the same underlying code. This unifies the "skills" available to your entire fleet of AI agents.[6]

5. Operational Workflows and Unified Orchestration

Managing three distinct configuration files for the same set of tools is inefficient and error-prone. To achieve a seamless multi-CLI workflow, a unified orchestration strategy is required.

5.1 Centralized Configuration Management

Tools like mcp-client-configuration-server and mcpdog have emerged to bridge the configuration gap. These act as "meta-servers" or configuration managers.[30]

The Proxy Approach (mcpdog):
mcpdog acts as a proxy layer. Instead of configuring the filesystem server in Gemini, Claude, and Codex separately, you configure them all to point to the mcpdog instance. mcpdog then manages the actual connections to the tools.

- *Benefit:* Add a tool once in mcpdog, and it effectively becomes available to all connected CLIs.
- *Implementation:* The CLIs connect to mcpdog via HTTP/SSE transport, simplifying the client-side config to a single URL.[30]

The Synchronization Approach (cc-switch):
Tools like cc-switch programmatically edit the configuration files of the target CLIs. A user can define a "master" list of servers, and the tool syncs this list to settings.json (Claude/Gemini) and config.toml (Codex). This approach maintains native stdio performance (avoiding HTTP overhead) while reducing manual config management. Advanced managers can even detect environment variable conflicts (e.g., different API keys for the same service) and visualize them for the user.32

5.2 Unified Context Strategies

While MCP standardizes tools, it does not inherently standardize "memory" or context files. However, the file system itself acts as the shared state.

The "Source of Truth" Strategy:
To ensure all agents operate with the same context, developers should maintain a master context file (e.g., PROJECT_RULES.md) and reference it in the agent-specific files.

1. **Create:** PROJECT_RULES.md containing style guides, architectural decisions, and prohibited patterns.
2. **Link (Gemini):** In GEMINI.md, add: "Refer to PROJECT_RULES.md for coding standards."
3. Link (Claude): In CLAUDE.md, add: "Read PROJECT_RULES.md before generating code." This ensures that if the architectural rules change, the update is propagated to all agents immediately without modifying their individual system prompts.13

5.3 Practical Workflow Scenarios

The following scenarios demonstrate how to leverage the specific strengths of each CLI in a unified workflow on Windows 11.

Scenario A: The "Architect-Builder" Loop (Refactoring)

1. **Architecture (Gemini CLI):** Use Gemini's massive context window to ingest the entire codebase.
   - *Command:* gemini -p "Analyze the entire codebase. Create a migration plan to Go. Save output to MIGRATION_PLAN.md."
   - *Insight:* Gemini's ability to hold 1M tokens allows it to "see" global dependencies that smaller context models might miss.[11]
2. **Implementation (Claude Code):** Use Claude's reasoning to implement the plan.
   - *Command:* claude -p "Read MIGRATION_PLAN.md. Implement the 'User Service' module in Go."
   - *Insight:* Claude's superior reasoning ensures the code follows best practices and correctly interprets the architectural intent.[4]
3. **Review (Codex CLI):** Use Codex to review the changes.
   - *Command:* codex -p "Review the new user_service.go against the original python implementation."
   - *Insight:* Using a different model family to review the work mimics human peer review, catching model-specific "blind spots".[33]

Scenario B: Deep Debugging with Browser Automation

1. **Tool Setup:** Install browser-tools-mcp to give agents control over a headless Chrome instance running in WSL2.[34]
2. **Diagnosis (Claude Code):**
   - *Command:* claude -p "Navigate to http://localhost:3000. Click 'Login'. Capture console logs."
   - *Action:* Claude drives the browser, reproduces the error, and captures the stack trace.
3. **Search & Fix (Gemini CLI):**
   - *Command:* gemini -p "Search web for this React error. Suggest a fix."
   - *Action:* Gemini uses Google Search to find up-to-date library issues, providing a solution potentially too new for Claude's training data.[12]

6. Advanced Customization, Security, and Troubleshooting

As agents gain the ability to execute shell commands and edit files, security becomes the paramount concern.

6.1 The Security Model: Trust but Verify

The "Human-in-the-Loop" model is the default. Tools like Claude Code and Gemini CLI will prompt for permission before executing "sensitive" actions (writes, deletes, network access).

Sandboxing:
Running agents directly in WSL2 gives them root-like access to the subsystem. For high-risk tasks, it is safer to run the CLI inside a Docker container.

- **Gemini Sandbox:** gemini --sandbox runs the agent in a containerized environment.[35]
- **Codex Sandbox:** config.toml supports sandbox = "workspace-write" modes to restrict write access to specific directories.[19]

API Key Hygiene:
Never hardcode API keys in settings.json. Use environment variable expansion. Store keys in ~/.bashrc (export ANTHROPIC_API_KEY=...) and configure MCP settings to read these variables (e.g., "apiKey": "${ANTHROPIC_API_KEY}").28

6.2 Troubleshooting Common Issues

Path Conflicts in WSL2:
Windows allows accessing Linux files via \\wsl$. However, running Node.js tools from Windows against Linux files is slow and error-prone.

- *Solution:* Always run the CLIs *inside* the WSL2 terminal. Ensure the tools are installed in the Linux environment (via nvm), not the Windows host.[5]

Auth Token Expiry:
OAuth tokens for Gemini and Claude can expire.

- *Solution:* Use gcloud auth application-default login for persistent Google credentials and ensure the Anthropic API key is used for Claude Code if session timeouts occur frequently.[36]

MCP Transport Errors:
"Server not responding" errors often occur when using stdio transport if the command path is incorrect.

- *Solution:* Use absolute paths in settings.json for the server executable (e.g., /home/user/.nvm/versions/node/v18/bin/node instead of just node) to avoid shell environment discrepancies.[27]

7. Conclusion

The convergence of Windows 11's robust virtualization via WSL2, the reasoning power of modern LLMs, and the standardization provided by the Model Context Protocol creates a developer environment of unprecedented capability. By constructing a multi-CLI workflow, developers can move beyond the limitations of a single model, employing Gemini as the architect, Claude as the engineer, and Codex as the reviewer. The future of this workflow lies in increased autonomy, where agents hand off tasks to one another, fully realizing the promise

of agentic AI development. The configurations and strategies outlined in this report provide the blueprint for mastering this new era.

Works cited

1. Gemini's command line tool is a productivity game changer, and it's free - how I use it, accessed on December 16, 2025, https://www.zdnet.com/article/geminis-command-line-tool-is-a-productivity-game-changer-and-its-free-how-i-use-it/
2. Develop and Deploy an Application Locally with Anthropic's Claude in Minutes, accessed on December 16, 2025, https://ozgur-kolukisa.medium.com/develop-and-deploy-an-application-locally-with-anthropics-claude-in-minutes-4edd46247b5d
3. Set up Claude Code - Claude Code Docs, accessed on December 16, 2025, https://code.claude.com/docs/en/setup
4. Claude Code Documentation, accessed on December 16, 2025, https://cc.deeptoai.com/docs/en
5. A complete guide to install Claude Code in 2025 - eesel AI, accessed on December 16, 2025, https://www.eesel.ai/blog/install-claude-code
6. Build an MCP server - Model Context Protocol, accessed on December 16, 2025, https://modelcontextprotocol.io/docs/develop/build-server
7. feat(cli): add shell autocompletion support for bash/zsh/fish via oclif · Issue #1855 - GitHub, accessed on December 16, 2025, https://github.com/google-gemini/gemini-cli/issues/1855
8. Completion file for Claude Code for Fish Shell. Save to `~/.config/fish/completions/claude.fish`. · GitHub - GitHub Gist, accessed on December 16, 2025, https://gist.github.com/r4ai/d3cb3360cd38b1ea0f28228b9473db0c
9. Claude Code as My Default Shell | Everybody wants to be a hacker, accessed on December 16, 2025, https://arnesund.com/2025/07/04/claude-code-as-my-new-shell/
10. Gemini Code Assist | AI coding assistant, accessed on December 16, 2025, https://codeassist.google/
11. google-gemini/gemini-cli: An open-source AI agent that brings the power of Gemini directly into your terminal. - GitHub, accessed on December 16, 2025, https://github.com/google-gemini/gemini-cli
12. Gemini CLI | Gemini Code Assist - Google for Developers, accessed on December 16, 2025, https://developers.google.com/gemini-code-assist/docs/gemini-cli
13. Hands-on with Gemini CLI - Google Codelabs, accessed on December 16, 2025, https://codelabs.developers.google.com/gemini-cli-hands-on
14. Use the Gemini Code Assist agent mode - Google Cloud Documentation, accessed on December 16, 2025, https://docs.cloud.google.com/gemini/docs/codeassist/use-agentic-chat-pair-programmer

15. A developer's Claude Code CLI reference (2025 guide) - eesel AI, accessed on December 16, 2025, https://www.eesel.ai/blog/claude-code-cli-reference
16. Quickstart - Claude Code Docs, accessed on December 16, 2025, https://code.claude.com/docs/en/quickstart
17. Building agents with the Claude Agent SDK - Anthropic, accessed on December 16, 2025, https://www.anthropic.com/engineering/building-agents-with-the-claude-agent-sdk
18. Codex CLI and VSCode: complete developer integration guide, accessed on December 16, 2025, https://vladimirsiedykh.com/blog/codex-cli-vscode-integration-complete-guide-2025
19. codex/docs/advanced.md at main · openai/codex - GitHub, accessed on December 16, 2025, https://github.com/openai/codex/blob/main/docs/advanced.md
20. Codex MCP Configuration: TOML Setup Guide for CLI and VSCode ..., accessed on December 16, 2025, https://vladimirsiedykh.com/blog/codex-mcp-config-toml-shared-configuration-cli-vscode-setup-2025
21. codex/docs/config.md at main · openai/codex - GitHub, accessed on December 16, 2025, https://github.com/openai/codex/blob/main/docs/config.md
22. Creating Your First MCP Server: A Hello World Guide | by Gianpiero Andrenacci | AI Bistrot | Dec, 2025, accessed on December 16, 2025, https://medium.com/data-bistrot/creating-your-first-mcp-server-a-hello-world-guide-96ac93db363e
23. What is Model Context Protocol (MCP)? A guide - Google Cloud, accessed on December 16, 2025, https://cloud.google.com/discover/what-is-model-context-protocol
24. Guide - Model Context Protocol（MCP）, accessed on December 16, 2025, https://modelcontextprotocol.info/docs/quickstart/guide/
25. MCP: Building your first MCP server and understanding core architecture, accessed on December 16, 2025, https://medium.com/@parichay2406/mcp-building-your-first-mcp-server-and-understanding-core-architecture-2836c21945e5
26. Architecture overview - Model Context Protocol, accessed on December 16, 2025, https://modelcontextprotocol.io/docs/learn/architecture
27. Configuring MCP Servers - Cline Docs, accessed on December 16, 2025, https://docs.cline.bot/mcp/configuring-mcp-servers
28. Connect Claude Code to tools via MCP, accessed on December 16, 2025, https://code.claude.com/docs/en/mcp
29. How to Set Up an MCP Server for Gemini: Step-by-Step Guide - Skywork.ai, accessed on December 16, 2025, https://skywork.ai/blog/how-to-set-up-gemini-mcp-server-claude-desktop-cli/
30. kinhunt/mcpdog: Universal MCP Server Manager - Configure once, manage multiple MCP servers through a single interface. Perfect for Claude Desktop,

Claude Code, Cursor, Gemini CLI & AI assistants. Web dashboard, auto-detection, unified proxy layer. - GitHub, accessed on December 16, 2025, https://github.com/kinhunt/mcpdog
31. MCP Client Configuration Server - Glama, accessed on December 16, 2025, https://glama.ai/mcp/servers/@landicefu/mcp-client-configuration-server
32. farion1231/cc-switch: A cross-platform desktop All-in-One assistant tool for Claude Code, Codex & Gemini CLI. - GitHub, accessed on December 16, 2025, https://github.com/farion1231/cc-switch
33. Use Gemini CLI within Claude Code and save weekly credits : r/ClaudeAI - Reddit, accessed on December 16, 2025, https://www.reddit.com/r/ClaudeAI/comments/1nyizsj/use_gemini_cli_within_claude_code_and_save_weekly/
34. Setting Up MCP Servers in Claude Code: A Tech Ritual for the Quietly Desperate - Reddit, accessed on December 16, 2025, https://www.reddit.com/r/ClaudeAI/comments/1jf4hnt/setting_up_mcp_servers_in_claude_code_a_tech/
35. Gemini CLI installation, execution, and deployment, accessed on December 16, 2025, https://geminicli.com/docs/get-started/deployment/
36. How to Build an MCP Server with Gemini CLI and Go | Google Codelabs, accessed on December 16, 2025, https://codelabs.developers.google.com/cloud-gemini-cli-mcp-go