Setting Up MCP Servers in Windsurf for a Full-Stack FastAPI & React Application

In this tutorial, we will walk through configuring Model Context Protocol (MCP) servers in the Codeium Windsurf IDE to supercharge Cascade, the built-in AI agent. By the end, Cascade will be able to scaffold and improve a full-stack FastAPI & React module (with SQLite/PostgreSQL databases) using specialized tools – all without over-relying on premium AI models. We’ll cover the MCP concept, Windsurf setup, installing key MCP servers (database, API, Git, UI, and web search), a real example (user authentication), and best practices for smooth, secure operation.

1. Overview of MCP Servers in Windsurf

What is MCP? Model Context Protocol (MCP) is an open standard that lets AI assistants (like Cascade) connect to external tools and data sources. In simple terms, an MCP client (Cascade) can make requests to MCP servers which provide specialized functionalities . Think of MCP servers as plugins or “skills” that Cascade can use – for example, querying a database, calling a web API, searching the web, or manipulating a Git repo. This extends Cascade’s capabilities beyond code generation, enabling it to perform actions and retrieve information that a standalone model normally couldn’t.

Why use MCP servers with Cascade? By leveraging MCP tools, Cascade can work more efficiently and accurately. Instead of guessing your database schema or API behavior, it can ask the database MCP for the real schema or use an API MCP to fetch actual responses. This means Cascade can scaffold modules with concrete data and context, reducing hallucinations and the need for an expensive model to fill in gaps. In essence, MCP gives Cascade a “Swiss army knife” to interact with your tech stack in a standardized way . For example, “Imagine adjusting your database schema simply by describing the change in plain English. This is possible by combining Codeium Windsurf with the Neon MCP Server.” . In practice, Cascade (the AI agent in Windsurf) can interface with multiple MCP servers to handle backend, database, and frontend tasks, acting like an AI DevOps assistant orchestrating the whole stack.

How do MCP servers work with Cascade? Cascade’s agentic chatbot is MCP-aware – it can detect when a certain tool is needed and invoke the corresponding MCP server. Under the hood, each server runs as a separate process that communicates with Cascade via JSON-RPC (through STDIO) following the MCP protocol . When Cascade “calls” a tool, Windsurf launches the MCP server (if not already running) and exchanges messages. For instance, if Cascade needs to search the web, it will call a brave\_web\_search tool on the Web Search MCP, which in turn executes the search and returns results for Cascade to use . All of this happens behind the scenes once configured, allowing you to interact in natural language while Cascade handles the tool usage. In summary, MCP servers extend Cascade’s context and abilities by giving it controlled access to your databases, APIs, codebase, and external knowledge.

2. Setting Up Windsurf IDE for MCP Servers

Before adding any MCP servers, make sure you have Windsurf Editor installed and updated (Windsurf is Codeium’s next-gen AI IDE available for Mac, Windows, Linux ). You’ll also need the necessary runtimes for MCP servers: typically Node.js (for JavaScript/TypeScript servers) and Python (for Python-based servers). Ensure you have Node 18+ and npm, and/or Python 3.8+ installed, since we’ll be running servers written in these languages . It’s also a good idea to have a code project set up with FastAPI (backend), React (frontend), and a database ready, as our MCP servers will interface with these.

Accessing MCP settings: In Windsurf, go to Settings > Advanced Settings and find the Cascade section (or open the Command Palette and choose “Open Windsurf Settings Page”) . Here you’ll see options to manage MCP servers. Click “Add Server” to add a new MCP server entry. Windsurf provides a few popular servers in a dropdown, but you can also add custom servers. If you choose a preset (like a web search or database tool), Windsurf will pre-fill some fields. For custom servers, it will open the JSON configuration file for editing.

The MCP config file: Windsurf stores MCP server settings in a JSON file located at ~/.codeium/windsurf/mcp\_config.json . This file contains a top-level "mcpServers" object where each server is listed by a key (your chosen name). Each server entry specifies how to start the server. For example, an entry might look like this:

{

"mcpServers": {

"google-maps": {

"command": "npx",

"args": ["-y", "@modelcontextprotocol/server-google-maps"],

"env": {

"GOOGLE\_MAPS\_API\_KEY": "<YOUR\_API\_KEY>"

}

}

}

}

In general, each server definition includes: a name (e.g., "google-maps"), a command to run (which can be an executable or command like npx, python, node, etc.), args (an array of command-line arguments to launch the server), and optionally an env dictionary for environment variables (like API keys, database URLs, etc.) . Windsurf will spawn the process using these settings.

Adding servers via the UI: If you use the Add Server button, you can select one of the known MCP servers or “Add custom server +”. This will insert a template into the JSON (or prompt you for details). Enter the required info (we’ll detail specifics for each server type in the next section). After editing, save the mcp\_config.json file. Then, in the Cascade chat panel, click the “Refresh” icon for MCP servers to load the new configuration . If everything is correct, Windsurf’s toolbar will indicate the servers are connected – for example, “1 available MCP server” (or more, depending on how many you added) should appear in Cascade’s MCP toolbar . Now Cascade is ready to use those tools.

Tip: If Windsurf doesn’t seem to connect to a server, double-check the config JSON for typos. On Windows, you may need to specify full paths to executables (Node, Python, etc.) if the PATH isn’t recognized by the GUI process . We’ll cover troubleshooting later, but a quick test is to run the same command and args in your terminal to ensure the MCP server starts without errors.

With the environment ready and the config in place, let’s proceed to install and configure the specific MCP servers for our full-stack app.

3. Installing and Configuring MCP Servers

Now we’ll set up the key MCP servers needed for a FastAPI-React full-stack project:

• Database MCPs (SQLite and PostgreSQL) – so Cascade can inspect the DB schema, run queries, and even manage migrations.

• API-aware MCP (FastAPI) – to help Cascade understand and interact with our FastAPI backend (e.g., by fetching API docs or calling endpoints).

• Git Repository MCP – to let Cascade read and modify the codebase (e.g., open files, write changes, run git commands).

• UI Scaffolding MCP (React) – to streamline creating React components and possibly manage the dev server or project structure.

• Web Search MCP – to give Cascade access to external documentation, guides, and best practices on the internet.

We will install each server (via npm or pip as needed) and add them to Windsurf’s mcp\_config.json. Each sub-section below provides step-by-step instructions for the specific MCP server.

3.1 Database MCP (SQLite & PostgreSQL)

SQLite MCP (local database): For development and quick prototypes, SQLite is a common choice. The MCP server for SQLite allows Cascade to run SQL queries against a SQLite database file, inspect tables, and even modify data or schema. In fact, the official SQLite MCP server offers tools for reading data (read\_query for SELECTs), writing data (write\_query for INSERT/UPDATE/DELETE), creating tables, and inspecting schema (listing tables and describing table structure) . This means Cascade can ask the MCP server “What tables exist?” or “Create a new table X with these columns”, and the server will execute the SQL accordingly.

Installation: The SQLite MCP server is available as a Python package. You can install it in your Python environment (the same one running FastAPI, for convenience) with:

pip install mcp-server-sqlite

Alternatively, it’s also available via Node (@modelcontextprotocol/server-sqlite on npm) if you prefer using npx. Here we’ll assume the Python route. After installing, no separate “service” needs to be running; Windsurf will spawn the server when Cascade needs it.

Configuration: Add an entry for SQLite in mcp\_config.json. For example:

{

"mcpServers": {

"sqlite-db": {

"command": "python",

"args": ["-m", "mcp\_server\_sqlite", "--db-path", "/path/to/your\_app.db"]

}

}

}

Replace "/path/to/your\_app.db" with the path to your SQLite database file (you can use an absolute path or relative to your project). The above config tells Cascade to launch the SQLite MCP by running python -m mcp\_server\_sqlite --db-path your\_app.db. Once added and refreshed, Cascade can now use SQLite-specific tools. For instance, if you ask “What tables do we have?”, Cascade might call the list\_tables tool and get back an array of table names . If you instruct “Create a new table for users with id, username, password”, Cascade can formulate a CREATE TABLE query and invoke the create\_table tool . The MCP server will execute it on the SQLite file and respond with a confirmation. This is far more reliable than the AI guessing SQL syntax – it actually runs it, so Cascade knows what happened.

PostgreSQL MCP: For a production-ready setup, PostgreSQL is often used. We have a couple of options here: a generic Postgres MCP server or the Neon MCP server (if using [Neon’s serverless Postgres](https://neon.tech)). The official MCP reference for PostgreSQL (available on GitHub/npm) supports read-only operations and schema inspection . It can list tables and run SELECT queries, which is useful for Cascade to understand the database. However, for creating/dropping tables or running migrations, a specialized tool is needed. This is where Neon’s MCP server shines – it extends capabilities to manage schema changes using Neon’s branching and API.

If you are using a plain local Postgres database, you can still use the generic Postgres MCP for introspection. (There’s also a community “SQL” MCP that supports connecting to any MySQL/Postgres with credentials , which could be configured similarly by providing connection details.) But to illustrate full capabilities including migrations, we’ll describe the Neon MCP setup:

Installation (Neon Postgres MCP): The Neon MCP server is published on npm. Ensure you have Node.js installed. No separate install command is needed thanks to npx. You will need a Neon account and Neon API Key (find this in your Neon project console) . Once you have the API key, you can configure Windsurf to use the Neon MCP.

Configuration (Neon): Add an entry in mcp\_config.json like so:

{

"mcpServers": {

"postgres-db": {

"command": "npx",

"args": ["-y", "@neondatabase/mcp-server-neon", "start", "<YOUR\_NEON\_API\_KEY>"]

}

}

}

This example (adapted from Neon’s guide) uses the @neondatabase/mcp-server-neon package . We pass the command start and the API key as an argument. (Make sure to replace <YOUR\_NEON\_API\_KEY> with your actual key; keep it secret!) The name "postgres-db" is an arbitrary label; you could call it "neon" as well. After saving and refreshing, Windsurf will spawn the Neon MCP server.

The Neon MCP acts as a bridge to your actual database on Neon’s cloud. It provides many tools that Cascade can call – listing projects and branches, running SQL queries, and orchestrating migrations on a temporary branch for safety . For example, Neon MCP defines actions like get\_database\_tables (to list all tables in a DB), describe\_table\_schema (to get column info of a table), run\_sql (to execute arbitrary SQL), and importantly prepare\_database\_migration and complete\_database\_migration to safely apply schema changes . This means Cascade can not only read from the database, it can also alter it under controlled conditions. In practice, if you say “add a new column to my users table,” Cascade might call prepare\_database\_migration – the MCP server will create a new Neon branch, apply the ALTER TABLE, verify it, then Cascade can call complete\_database\_migration to merge it in . All with your confirmation in the loop.

If you’re using a local Postgres (not Neon), you might not have the migration convenience. In that case, Cascade could still retrieve the schema (e.g., via a generic Postgres MCP that offers list\_tables and maybe a run\_query for SELECTs). For applying schema changes, you might need to run migrations via your normal tools, or use a community MCP that accepts a connection string. (One such tool is the “MCP SQL” server on mcp.so, which supports MySQL/Postgres by providing DB credentials in env vars .) The setup would be analogous, but you’d include something like "command": "npx", "args": ["-y", "@modelcontextprotocol/server-postgresql"] and provide env vars for host/user/password. Choose the path that fits your environment. For simplicity, we continue assuming Neon for the advanced use-case.

Usage: Once configured, Cascade can use the database MCP. For instance, you might see Cascade issuing a tool call like > MCP Tool: postgres-db / get\_database\_tables and then it will present the list of tables it got . Or, after asking it to add a column, Cascade might respond with something like “I’ll create a migration for that” and internally do postgres-db / prepare\_database\_migration. The benefit is Cascade knows the exact outcome (success or errors) of the SQL operations. You, as the developer, get to just converse in natural language, while Cascade ensures the database is updated accordingly. (Always verify changes in your DB or through logs – Cascade will usually describe what it did, and you can double-check via your DB client or Neon’s dashboard.)

3.2 API-Aware MCP (FastAPI Backend)

Our backend is FastAPI, and we want Cascade to be “aware” of it. This means Cascade should be able to retrieve information about the API (available endpoints, schemas) and even test the endpoints. While Cascade will have access to your FastAPI code via the Git repo MCP (so it can read the source), having a live API perspective is useful. For example, Cascade could fetch the OpenAPI documentation or try making a sample request to see what the response looks like. This helps it understand how to integrate or modify the API.

Solution: We can use a Fetch MCP server to let Cascade perform HTTP requests. The Fetch MCP is a generic tool that can fetch web content (like a simplified curl for the AI) . By configuring this, Cascade can call any URL and get the result (HTML, JSON, etc.), which it can then analyze or use to adjust its code generation.

Installation: The Fetch server is part of the MCP reference set (written in TypeScript). No separate install needed other than Node.js. We will use npx to run it on demand.

Configuration: Add an entry for fetch in the MCP config:

{

"mcpServers": {

"fetch": {

"command": "npx",

"args": ["-y", "@modelcontextprotocol/server-fetch"]

}

}

}

This tells Windsurf to use the server-fetch package. (No API keys or special args required for basic web fetch.) Once this is added, Cascade can fetch URLs.

Using Fetch for FastAPI: First, make sure your FastAPI development server is running (e.g., uvicorn main:app --reload on http://127.0.0.1:8000). Cascade could then do something like: “Fetch the OpenAPI spec”. In practice, you might ask Cascade, “What does the API documentation say?” If Cascade decides to use the tool, it will call fetch on http://127.0.0.1:8000/openapi.json (the JSON spec) or perhaps the docs URL. The fetch MCP will return the JSON text, which Cascade can parse to learn your endpoints and schemas. This can greatly inform Cascade’s next steps (for example, it won’t create a duplicate endpoint if it sees one already exists, or it will use the correct field names from your Pydantic models).

Cascade can also use fetch to test endpoints. Suppose Cascade implemented a new /login endpoint for you – it could call fetch on http://127.0.0.1:8000/login with a sample payload to verify it gets the expected status (perhaps as part of its validation steps). This is similar to integration testing driven by the AI. Keep in mind that you might need to instruct Cascade to perform such tests (it may not do it automatically unless prompted or if it encounters an error and wants to debug).

Alternative approaches: If you wanted a more specialized API-aware MCP, you could write a custom MCP server using FastAPI itself that exposes your application’s internals (for example, a tool that lists all router paths directly from FastAPI’s app.routes). However, this is advanced and typically unnecessary if using fetch plus the Git repository context. Another existing MCP server is OpenRPC which targets JSON-RPC APIs – not directly applicable to REST, but shows the concept of API integration. For our needs, the fetch server is usually sufficient to give Cascade “eyes” on the running API.

3.3 Git Repository MCP (Code Navigation & Editing)

Cascade’s strength is improved by its awareness of your codebase. Windsurf already indexes your open files and some project context, but the Git MCP server adds powerful tools for repository-wide navigation and version control operations. With the Git MCP, Cascade can search your entire code for references, open or modify files it hasn’t seen yet, diff changes, and even commit to your Git history.

What can the Git MCP do? According to the MCP server directory, mcp-server-git provides “tools to read, search, and manipulate Git repositories” . Key features include checking the status and diffs of your working directory, staging changes, committing, resetting, creating/switching branches, and reading commit logs . Essentially, Cascade can act like a collaborator using Git: find where a function is defined, make a new branch for a feature, apply edits, and commit them with a message.

Installation: The Git MCP server is available both as a Python package and a Node package. We’ll use the Python version (to keep it similar to SQLite’s setup). Install via pip:

pip install mcp-server-git

Ensure this is installed in an environment accessible to Windsurf (global or the project venv).

Configuration: Add a config entry for the Git server. We need to tell it which repository to operate on. For example:

{

"mcpServers": {

"git-repo": {

"command": "python",

"args": ["-m", "mcp\_server\_git", "--repository", "/path/to/your/project"]

}

}

}

Replace the repository path with the root path of your project’s Git repo (the folder where your .git directory lives). This will launch the Git MCP pointing at that repository. (Important: If your project is not already a git repo, initialize it with git init so that the MCP server has something to work with. Even for just file operations, it expects a repo context.)

After adding and refreshing, Cascade gains a lot of knowledge about your code. For instance, if you ask Cascade to implement a new FastAPI route, it could use the Git MCP’s search tool to find the file where your API routes are defined instead of relying solely on file name hints. It might do something like: > MCP Tool: git-repo / search\_code with a query, and then open the file containing the search term. From there, Cascade can insert code. Without MCP, the AI might only consider open files or a small context; with Git MCP, it can actively fetch any file from the repository it needs to modify.

Example usage: Suppose you have a main.py or routes.py where API endpoints reside. Cascade can call a read\_file tool (provided by the Git server) to pull in that file’s content if it’s not already in context, then propose edits. When you approve changes, Cascade might eventually stage and commit via git-add and git-commit tools. If configured, it could even auto-commit with a message, though typically you’d want to review changes first. Another scenario: Cascade could use the Git log tool to see when something was last modified (for understanding context or debugging).

The Git MCP essentially makes Cascade a participant in your version control system, which is powerful but also something to use carefully. Always review what Cascade is about to commit. You can instruct Cascade to show a diff (it can use the MCP to get a diff of working changes) if you want to double-check. This integration ensures improvements Cascade makes are tracked and can be reverted if needed.

3.4 UI Scaffolding MCP (React Frontend)

Frontend development can be iterative and visual. Cascade can generate React code using its base AI capabilities, but a UI scaffolding MCP for React can accelerate this by giving Cascade direct control over the React app environment. There are a couple of ways to achieve this:

• Use the Git MCP (already set up) to handle file creation/updates for React components, plus perhaps a Node MCP to run build/test commands.

• Or use a dedicated React MCP server that provides higher-level tools to manage a React project (create new apps, start dev server, manage dependencies, etc.).

To maximize Cascade’s efficiency, we’ll look at the dedicated React MCP approach. A community-developed example is React MCP by Streen9 . This MCP server essentially automates React project tasks. It allows an agent (Claude/Cascade) to: “Create new React applications, run React development servers, manage files and directories, install npm packages, execute terminal commands, and track long-running processes” . In other words, Cascade can not only write React component code, but also start the app and see if it runs, add necessary dependencies (like UI libraries), and even create new project scaffolds using tools like Create React App or Vite.

Installation: Currently, such React MCP servers may not be on pip or npm registries; instead, you might need to clone the repository. For example, to use the React MCP by Streen9, you would:

1. Clone the GitHub repo (e.g., git clone https://github.com/Streen9/react-mcp.git).

2. Install its dependencies: go into the cloned directory and run npm install .

3. (Optional) Test it by running node index.js to ensure it starts without error.

This server is intended to run locally (since it directly interfaces with local file system and processes) .

Configuration: After installing, configure Windsurf to run this server. Since it’s a Node project, use the Node executable. For example:

{

"mcpServers": {

"react-app": {

"command": "node",

"args": ["/path/to/react-mcp/index.js"]

}

}

}

Adjust the path to wherever the index.js (or main script) of the React MCP is located. Save and refresh Windsurf. If successful, Cascade can now utilize the React MCP’s tools.

Capabilities: With the React MCP, Cascade could do things like:

• Generate Components: You could ask, “Create a new <LoginForm> component with a username and password field.” Cascade will generate the code, and via the MCP it could directly write a new file LoginForm.jsx in your project (using a file tool, similar to Git MCP’s file write).

• Manage Project: If you said, “Initialize a new React project for this module,” the MCP has a create-react-app tool that could scaffold an app (or use Vite, etc.) . It can effectively run the equivalent of npx create-react-app behind the scenes.

• Run Dev Server: Cascade can instruct the MCP to start the development server (npm start or vite dev). The MCP will handle running the process and can stream back logs. This means Cascade could detect compilation errors or linter warnings in real-time and then fix the code accordingly – all automated.

• Install Dependencies: If your component needs a library (say you want to use Tailwind CSS or a UI kit), Cascade can invoke an install\_package tool with the package name. The MCP will run npm install and report success/failure. Cascade then knows the package is available and can import it in the code it writes.

All these capabilities make frontend scaffolding much faster. Instead of you manually creating files, running commands, or looking up package docs, Cascade can handle many of those steps. One user noted that such iterative support is key: “What I need is something that can iteratively support and update my existing codebase — my current screens and app components.” (from a Codeium user on Reddit discussing Windsurf for UI work). The React MCP is designed for that iterative development loop.

Alternative / Simpler approach: If setting up the React MCP feels heavy, you can still rely on the Git MCP and Cascade’s base coding ability for UI: Cascade can create new .jsx/.tsx files by using the Git MCP to write files, and you can manually run your dev server to test. However, you lose the feedback loop where Cascade sees the runtime results. As a compromise, you could integrate a Browser automation MCP like Puppeteer or Playwright later, which could let Cascade open a headless browser and snapshot the UI (there are MCP servers for Puppeteer) – but this is quite advanced and not necessary for our module scaffolding goal. Start with file generation and perhaps gradually incorporate more if needed.

In summary, adding a UI scaffolding MCP like the React server enables Cascade to more directly create and manage frontend components. This offloads repetitive setup tasks and allows Cascade to ensure the frontend and backend stay in sync (Cascade can modify both ends in one go, e.g., update an API and adjust the React fetch call accordingly).

3.5 Web Search MCP (External Documentation)

Even with all the above servers, Cascade’s knowledge is limited to what it already knows or what your project provides. For general questions or best practices (e.g., “How should passwords be hashed in FastAPI?” or “What’s the typical response format for authentication APIs?”), it’s invaluable to let Cascade search the web. A Web Search MCP gives Cascade an internet connection (read-only) to fetch documentation, code examples, or forum solutions. This helps Cascade make informed suggestions without hallucinating details, and reduces the need for you to manually paste docs.

Choice of Search MCP: A popular option is the Brave Search MCP (which uses the Brave Search API) . It has a free tier (as of writing, ~2k queries/month) – you just need to sign up for an API key . Another choice is Tavily (an AI-tailored search engine) or Search1API – but for this guide we’ll use Brave Search.

Installation: It’s a Node-based server. No manual install beyond having Node; we’ll use npx to run it. Obtain your Brave Search API Key from [Brave’s developer portal](https://api.search.brave.com/) (sign up for an account, get an API key – the free tier should suffice).

Configuration: Add the Brave search server in the config, including the API key in env:

{

"mcpServers": {

"web-search": {

"command": "npx",

"args": ["-y", "@modelcontextprotocol/server-brave-search"],

"env": {

"BRAVE\_API\_KEY": "<YOUR\_BRAVE\_API\_KEY>"

}

}

}

}

Replace <YOUR\_BRAVE\_API\_KEY> with the key string from Brave. After saving and refreshing, Cascade can now perform web searches. The Brave MCP server exposes tools like brave\_web\_search (for general web queries) and possibly brave\_local\_search (for searching a local index, if configured). The brave\_web\_search tool allows specifying a query and number of results, etc. .

Usage: You don’t necessarily have to tell Cascade “use the web search tool.” Instead, you can ask open-ended things and Cascade will decide if it should search. For example, if you prompt: “Find a secure way to hash passwords in FastAPI.”, Cascade might not recall offhand. It can trigger the web-search MCP, e.g., > MCP Tool: web-search / brave\_web\_search with query “FastAPI password hashing secure method”. The MCP will return search results (titles, snippets, and URLs). Cascade can then summarize or follow up. It might even fetch a specific URL’s content if needed, possibly using the Fetch MCP for the actual page (some search MCPs have a direct web content retrieval too).

Another use: Cascade can look up best practices or code examples for React (like how to manage form state, or how to implement JWT authentication on the frontend). This way, Cascade augments its training knowledge with real documentation in real-time, leading to more accurate scaffolding.

Best Practices via Web Search MCP: There is even an MCP server specifically for coding best practices (covering React, Python, etc.) – the Coding Standards MCP . That server provides predefined guidelines (like React style guide or best practices for performance) which could be useful if Cascade is tasked with improving code quality. Setting it up would be similar (it’s a Python MCP that you’d run via uv or pip), but an easier approach is simply to allow Cascade to search for “[Technology] best practices” on the web. For instance, Cascade could search “React accessibility best practices” if you ask it to review your UI for accessibility. The web search MCP will retrieve relevant info that Cascade can incorporate into its suggestions.

With the web search MCP configured, Cascade effectively has the documentation and Stack Overflow at its fingertips (within reasonable usage limits). This dramatically improves its ability to solve problems or implement features correctly without guessing, fulfilling our goal of minimizing reliance on an expensive model for guidance. It’s like having an AI pair programmer who is allowed to Google things when unsure.

At this stage, we have configured all the major MCP servers we need:

• sqlite-db (for SQLite queries/migrations),

• postgres-db (for Postgres operations, e.g., Neon),

• fetch (for calling our FastAPI or other web endpoints),

• git-repo (for codebase navigation and git operations),

• react-app (for managing React frontend scaffolding),

• web-search (for external documentation lookup).

You can verify each by checking Cascade’s MCP toolbar in Windsurf (each server should be listed as available). Now, let’s put this setup into action with a concrete example.

4. Real-World Example: Implementing a User Authentication Module

To demonstrate how Cascade and these MCP servers work together, we’ll walk through building a simple User Authentication feature for our full-stack app. This feature involves creating a user model in the database, API endpoints for registration/login, and a React frontend for users to sign up and log in. We’ll highlight how Cascade uses each MCP tool to scaffold and refine the module.

(This is a hypothetical example for illustration – the actual steps Cascade takes might vary, but it will follow the same general pattern.)

4.1 Planning and Initiation

First, make sure all relevant servers are running (Windsurf will start them as needed). Open Cascade (the chat in Windsurf) and describe what you want to build. For example:

User: “Let’s add a user authentication system. We need a User model in the database (with fields id, username, password\_hash, created\_at), a FastAPI /register route to create users (hashing passwords), a /login route to authenticate and return a token, and a React RegistrationForm and LoginForm component on the frontend. Use SQLite for development and ensure the code follows best practices.”

This initial prompt gives Cascade a clear goal. Cascade will likely break down the task and might ask clarifying questions. Thanks to our MCP setup, it can immediately start gathering information:

• Cascade might verify the current state of the database: “Let me check if a users table exists.” It will call the database MCP.

• Database MCP in action: > MCP Tool: sqlite-db / list\_tables. Suppose no user table exists, the MCP returns []. Cascade knows it needs to create one.

• Cascade might also inspect the backend code to see where to add routes:

• Git MCP in action (code search): It could do > MCP Tool: git-repo / search\_code with query “APIRouter” or “@app.post” to find the file where API routes are defined. The MCP might return matches in app/api/routes.py.

• Cascade then does > MCP Tool: git-repo / read\_file on that file to get the content into context.

With context in hand, Cascade will propose a plan: e.g., “I will create a database table, add two API endpoints, and corresponding React components.”

4.2 Database Schema Creation

Cascade proceeds to implement the database changes:

• Creating the User table: Cascade formulates the SQL for a new table. For SQLite, it might do: CREATE TABLE users (id INTEGER PRIMARY KEY, username TEXT UNIQUE, password\_hash TEXT, created\_at DATETIME);. Instead of just suggesting it, Cascade uses the SQLite MCP to execute it:

• > MCP Tool: sqlite-db / create\_table with an input containing the SQL. The SQLite MCP executes this. If successful, it returns a confirmation (or the new table schema).

• Cascade verifies the result by calling > MCP Tool: sqlite-db / list\_tables again, and possibly describe\_table on users to confirm columns . Now Cascade knows the exact schema in the DB.

If using Postgres/Neon, the process is similar but might involve the prepare\_database\_migration tool:

• > MCP Tool: postgres-db / prepare\_database\_migration with details of the new table. The Neon MCP would create a temp branch, add the table, and report back success .

• Cascade then calls complete\_database\_migration to finalize it in the main branch. The Neon MCP confirms the migration .

• Cascade can list tables via get\_database\_tables to ensure users now exists.

At this point, the database layer is ready with a users table. Cascade will likely mention “Created a users table in the database.”

4.3 Backend API Implementation

Next, Cascade creates the FastAPI endpoints:

• Register endpoint: Cascade writes a /register endpoint that accepts username & password, hashes the password, and stores the user. It might generate code using FastAPI’s Pydantic models and dependency injection for the DB. For example, it might propose:

@app.post("/register", status\_code=201)

def register(user: UserCreate): # UserCreate is a Pydantic model for username & password

hashed\_pw = hash\_password(user.password)

new\_user = models.User(username=user.username, password\_hash=hashed\_pw)

db.add(new\_user)

db.commit()

return {"id": new\_user.id, "username": new\_user.username}

The AI knows standard practices like hashing; if not, it can look them up (more on that soon).

Cascade will insert this into the appropriate place in your code. How? Using the Git MCP file write. After drafting the code, it calls:

• > MCP Tool: git-repo / write\_file with the file path (e.g., app/api/routes.py) and the new content (possibly it fetches the original content first, merges in changes, then writes). Or it might use a more direct apply\_patch tool if available.

• The Git MCP writes the changes to disk.

• Login endpoint: Cascade similarly writes a /login route. It will likely:

• Query the DB for the user by username (using an ORM or raw SQL via the DB MCP – but more likely it will write code using the ORM or session).

• Verify the password by hashing the input and comparing to stored hash.

• Return a token or success message. (We might not have token infrastructure in this simple example, but Cascade could integrate something like JWT. If you requested it, Cascade may decide to pip install pyjwt – which it can do by calling the React MCP or a Shell MCP to run pip or npm. In absence of a shell MCP, it could instruct you to install the library. There are also MCP servers for pip installs or CLI commands, but we didn’t explicitly add one. However, the React MCP can execute arbitrary commands, and we could use that to run pip too in a pinch.)

Cascade writes the login route code and uses Git MCP to save it.

• Models and utility functions: Perhaps Cascade needs a Pydantic model UserCreate or a helper hash\_password. It will either create those in the same file or a models.py. If models.py doesn’t exist, Cascade can create it:

• > MCP Tool: git-repo / write\_file for app/models.py with a SQLAlchemy model class or similar.

• It might also update app/main.py to include the new router if needed (again using the Git MCP).

• Throughout this, Cascade might consult documentation:

• For example, how to properly hash a password in FastAPI. If Cascade is unsure, it can utilize Web Search MCP:

• > MCP Tool: web-search / brave\_web\_search with query “FastAPI password hashing bcrypt example”. It gets results, finds that using PassLib or bcrypt is common. It might fetch a snippet from FastAPI docs on security . Armed with that, it uses the recommended approach (maybe FastAPI’s PwdContext). If an external library is needed (like passlib), Cascade will either prompt you to install it or use a tool to do so (if you had a pip MCP or similar).

• Or, if Cascade recalls enough, it may implement a simple hash with hashlib for demonstration.

• Testing the API (optional): Thanks to the Fetch MCP, Cascade can test what it just built. It could simulate a registration:

• > MCP Tool: fetch / fetch\_url with the local /register endpoint and a sample JSON payload. If the server is running (we’d have to ensure the FastAPI app is up-to-date and running reload – Cascade might even trigger a restart via the React MCP’s command execution, e.g., running your uvicorn command).

• Suppose the response is 201 Created with some JSON. The fetch MCP returns that to Cascade. If Cascade sees an error (e.g., it forgot to add a unique constraint and gets an integrity error on duplicate username), it can adjust the code and try again.

• This tight loop is extremely helpful: rather than you running the app and testing manually, Cascade can do quick sanity checks via fetch and fix issues on the fly.

At the end of backend implementation, we have working /register and /login endpoints, plus any supporting code (models, hashing). Cascade might commit these changes:

• > MCP Tool: git-repo / git\_commit with a message “Add user register and login endpoints”. The Git MCP creates a commit in your repo, so you have a checkpoint.

4.4 Frontend Component Generation

Now onto the React front end:

• RegistrationForm component: Cascade will create a new React component file, say src/components/RegistrationForm.jsx. It will use either the React MCP or the Git MCP to create the file. Likely:

• > MCP Tool: react-app / write\_file (if React MCP reuses Git-like file operations) or simply the Git MCP if the repo path covers the frontend directory. In our config, we pointed Git MCP to the whole project, so it might handle both backend and frontend files if in one repo. If your frontend is a separate repository or not under the same path, you could add another Git MCP instance for the frontend path.

• The component code might include a form with fields for username, password, and a submit handler that calls the FastAPI /register endpoint (using fetch or axios).

• Cascade might realize we need a way to make HTTP calls from React (like fetch or a library). If using plain fetch, no new dependency is needed. If you wanted to use an SDK or something, Cascade could use the web search MCP to see if there’s an existing NPM package or recommended approach. But a simple fetch in a handleSubmit is straightforward.

• LoginForm component: Similarly, Cascade creates LoginForm.jsx with fields for credentials and on submit, calls the /login API. If the API returns a token, Cascade would store it (maybe in localStorage or context). It might mention this, or implement a minimal version.

• Integration into App: Cascade might update a main App.jsx or routes to include these forms (perhaps adding routes using React Router if your app uses it). It will again use file edit operations to do so.

• Styling and UX: If you specified using a library or styling approach (e.g., Tailwind CSS), Cascade can incorporate that. It might need to install Tailwind via React MCP’s npm install:

• > MCP Tool: react-app / run\_command to execute npm install tailwindcss (for example). After installing, it might configure it if it knows how (like adding a CSS file import). Cascade can also search for “Tailwind React setup” if needed.

• Running and verifying UI: With the React MCP, Cascade could actually start the dev server:

• > MCP Tool: react-app / run\_dev\_server (the exact tool name might differ, but essentially it runs the start script). The MCP will capture the output. Cascade sees in the logs if compilation succeeded or if there were errors.

• If an error: Cascade reads the error (the MCP would return the stdout/stderr of the dev server). Suppose there’s a typo or missing import, Cascade can pinpoint it and fix the code, writing the file again. This iterative fix can continue until the dev server runs clean.

• Cascade might not actually render the UI for itself (unless we had a browser screenshot MCP), but a clean compile is a good sign. In a more advanced setup, you could have it use Puppeteer MCP to open http://localhost:3000 and snapshot the page, then maybe use an OCR or image recognition to verify content. That’s beyond our scope – likely unnecessary here.

At this stage, the frontend components are done. Cascade could commit the changes via Git MCP again: “Add Registration and Login components”.

4.5 Using MCP Tools to Iterate and Improve

With the first pass implemented, we have a functional but basic auth system. Cascade can now iterate to improve it, leveraging the MCP tools for insights:

• Security improvements: Maybe remind Cascade about security (if it hasn’t already). You might ask, “Is this secure?”. Cascade could use web search to cross-check best practices. For example, it might find that storing plain tokens in localStorage is susceptible to XSS. It could suggest HttpOnly cookies or other mitigations. It might not code the entire JWT flow by itself unless prompted, but with the MCP, it can fetch an article about “FastAPI JWT tutorial” to guide it. If you agree, Cascade can implement those changes (install python-jose for JWT encoding, adjust the login to set a secure cookie, etc.). This involves both backend changes (FastAPI) and front-end (perhaps adjusting to not directly handle token).

• Validation and error handling: Cascade could test edge cases using the Fetch MCP. E.g., “What happens if a username is taken?”. It might call /register twice with the same username. If the second time the DB raises an IntegrityError, Cascade sees maybe a 500 error. It can then modify the backend to catch that and return a 400 with “username taken”. It writes that error handling code and tests again via fetch until it gets a proper 400 response. This showcases how having the DB and API at its fingertips lets Cascade refine robustness.

• User experience tweaks: Perhaps using web search, Cascade finds a React best practice guide for form UX (maybe via our coding standards server or just a general search). It might incorporate, say, disabling the submit button while loading, or adding some basic form validation. These are small improvements but add up. Cascade could implement a simple check that password length > 6, for example, on the frontend, after searching “common password validation rules”.

• Documentation: Cascade could even help write documentation/comments. It could gather some info (with web search) on how to document FastAPI endpoints using docstrings or comment on the React components about usage. While not a tool per se, the web access aids Cascade in following conventions (for instance, it might search “FastAPI add description to APIRouter”).

• Commit final changes: Once everything is polished, Cascade uses the Git MCP to commit. If you have CI tests, Cascade could run them via a command (if we had a CLI MCP or using the React MCP’s command execution – it could run pytest or frontend tests and fix any failures). This way, the module is in a pretty good shape with minimal manual intervention.

Throughout this process, Cascade functioned as an autonomous agent orchestrating multiple tools:

• It queried and modified the database through the DB MCP (ensuring the model layer was correct).

• It read and wrote code through the Git MCP (so it knew exactly where and how to apply changes).

• It fetched running API responses via the Fetch MCP (validating backend behavior).

• It possibly ran and monitored the frontend app via the React MCP (catching errors and verifying integration).

• It researched best practices via the Web Search MCP (so the solutions were aligned with current standards and libraries).

Because of this, Cascade didn’t need to “imagine” what the database or API looked like – it worked with the real thing. The result is a working auth system scaffolded much faster than writing everything from scratch, and with the AI doing the heavy lifting under your guidance.

To give a concrete glimpse, here’s a snippet from a Cascade conversation (using a Neon Postgres MCP) performing a schema change:

User: Add a created\_at timestamp column to the users table.

Cascade: “I’ll help you add a created\_at column to the users table. I’ll create a migration for this change.”

Cascade calls: > MCP Tool: postgres-db / prepare\_database\_migration (the MCP creates a temp branch and adds the column) .

Cascade: “The column was added on a temp branch. Should I apply it to main?”

User: Yes.

Cascade calls: > MCP Tool: postgres-db / complete\_database\_migration.

Cascade: “The migration has been completed successfully! The created\_at column is now in the users table.” (Cascade confirms by describing the updated table structure).

This illustrates how natural the interaction can be: you describe what you want, Cascade uses MCP tools to do it, and then confirms the outcome in plain English . We saw similar flows for creating tables, testing endpoints, etc., in our auth module scenario.

5. Best Practices & Troubleshooting

Setting up and using MCP servers unlocks a lot of power, but it also adds complexity. Here are some best practices and common issue resolutions to ensure a smooth experience:

✅ Clearly scope Cascade’s tasks: Because Cascade will execute commands from your prompts without always asking for confirmation , make sure your instructions are unambiguous and intentional. Always double-check what you ask Cascade to do, especially if it involves destructive actions (like database drops or large code refactors). It’s wise to use version control (Git) and commit changes in stages – this way, if Cascade’s changes aren’t as expected, you can revert easily.

✅ Use MCP tools to verify before major changes: As we demonstrated, Cascade can list tables or fetch data before altering things. Encourage it to do so by asking questions like “What’s the current state?” or instructing it: “Verify X before proceeding.” This helps avoid mistakes. For example, check if a user exists before creating to avoid duplicates, or get a diff of code before committing.

⚡ Performance considerations: Running many MCP servers can be resource-intensive. Each server is a separate process (some may run servers or database connections in the background). If you notice your system slowing, you might not need all servers active at all times. Load only the ones you need for the task at hand. For instance, if you’re focusing solely on backend logic, you might disable the React MCP temporarily. Also, be mindful that web search queries take network time – Cascade will wait for those results, so complex searches might slow down the response. Keep queries specific to get faster answers, or limit the number of search results (the Brave search tool allows a count parameter) .

🔒 Security considerations: Treat MCP servers with the same caution as any code running on your machine:

• Use trusted MCP servers: We mostly used official or well-known community servers. If you add others, ensure they come from reputable sources, as they will have access to your system (e.g., a malicious Git MCP could potentially run arbitrary git commands). The ones we used (database, git, fetch, etc.) are either official reference implementations or popular community tools.

• Limit file system access: Some MCP servers (like Filesystem or Git) might have options to restrict to certain directories. For Git, we pointed it to the project path explicitly (--repository /path/to/project), so it won’t wander outside. Similarly, for SQLite we specified the DB file. Whenever possible, configure these scopes to prevent accidental access to files outside your project.

• Keep API keys secret: As mentioned, use the env field in config for keys (Brave API, Neon API) rather than putting them in prompts or code . The env ensures the key is injected only into the MCP process and not exposed to the AI’s text context. And never share the config file with keys publicly (the Neon guide emphasizes keeping the API key secure ).

• Sensitive data caution: If your database has real user data, be careful letting Cascade query it. Cascade’s output (which might include query results) could be sent to Codeium’s servers for analysis (since Cascade the agent is powered by remote AI models, unless you’re using an entirely local model). Redact or avoid querying highly sensitive info via the AI. Use a dummy database or sanitized data when possible during AI-driven development.

🚧 Troubleshooting MCP server issues:

• Server not starting / not detected: If Windsurf says a server is unavailable or you see errors like “spawn ENOENT”, it usually means the command in config failed. On Windows especially, the PATH seen by GUI apps can be different . The solution is to specify full paths. For example, instead of "command": "npx", use "command": "C:\\Program Files\\nodejs\\node.exe" and in args include the path to npx (as shown in a Stack Overflow solution) . Similarly, for Python on Windows, you might put the path to python.exe. On Mac/Linux, ensure the command exists: e.g. you may need to install uvicorn or uvx if using those in commands.

• Check logs for clues: Windsurf doesn’t always show the MCP server logs by default. You might run Windsurf from a terminal to see output, or check any log files it produces. Another approach: run the MCP server manually in a separate terminal to see if it throws an error. For example, run npx -y @modelcontextprotocol/server-brave-search with the API key env set – does it start or print an error? If manual start fails, fix that first (perhaps you missed an API key or a package didn’t install).

• Version mismatch or updates: MCP is evolving, so ensure your MCP server packages are up to date. If a tool isn’t working as expected, there might have been updates. For instance, if Brave Search API changed, you might need a newer version of the server. Update via npm or pip accordingly.

• Cascade doesn’t use the tool when expected: Cascade decides when to use an MCP tool. If it’s not using one and you think it should, you can explicitly prompt it. For example: “Use the database tool to list the users table” or “Search the web for FastAPI JWT documentation”. Cascade might have simply not thought it needed the tool. By hinting or instructing directly, you can nudge it. Over time, as you work, Cascade will “realize” it has these capabilities and use them more proactively. Also, check that the server name you reference matches what Cascade expects – e.g., in some responses it might mention “I will use the neon tool” because we named it postgres-db. It’s generally smart about mapping, but clarity helps.

• Conflicts between tools: With many tools, sometimes Cascade might be unsure which to use (for example, if both a generic SQL and Neon SQL tool are present). In our config, we gave them distinct names (sqlite-db, postgres-db). Cascade usually picks the relevant one based on context (it knows if you mention Neon or local). If it ever uses the wrong one, correct it in the prompt (e.g., “Use the SQLite tool, not Neon, for this.”).

⚙️ Optimizing workflow: To get the most out of this setup, consider these tips:

• Iterate in small steps: It’s tempting to ask Cascade to build the entire module in one go. It can try, but guiding it step-by-step yields better results. For example, first ask it to set up the DB model, then the API, then the UI. This way, you can catch issues early (with MCP tools) and course-correct. It also reduces the chance of overwhelming the AI with too many contexts at once.

• Review and refine: Cascade can produce working code, but always review the code quality. Use the Git diff (Cascade can show you what changed) and run your tests. You can even ask Cascade to run tests if you integrate a testing MCP or just use the React MCP to execute pytest. If a test fails, Cascade (with context of the failure message) can fix the code.

• Leverage best practice guides: Through the web search MCP, you can quickly pull in style guides. For instance, ask “What are best practices for React form handling?” Cascade might find a blog or the official React docs. You can then say, “Apply these best practices to the LoginForm component.” Cascade will attempt to refactor accordingly (maybe using controlled components, proper state management, etc.). This is a great way to improve the scaffolded code beyond a naive implementation.

• Keep an eye on environment: Sometimes MCP servers (like database ones) might maintain state. For example, the SQLite MCP is connected to your DB file – if you manually change the DB schema outside of Cascade, the AI might not immediately know. If things get out of sync, you can always call the appropriate describe/list tool again so Cascade refreshes its knowledge. Similarly, if you restarted Windsurf, the servers might reset; just ensure to refresh and re-run any needed context (like listing tables) at the start of a new session.

By following these practices, you’ll harness Cascade’s full potential while maintaining control and security. Cascade, enhanced with MCP servers, can significantly speed up development on a full-stack app. It’s like having an junior developer who can read docs at lightning speed, write code, run tests, and commit – but still benefits from your guidance and oversight to steer the project in the right direction.

In conclusion, setting up MCP servers in Windsurf involves a bit of configuration upfront, but the payoff is an AI assistant that truly understands and works with your entire stack. We covered:

• The concept and benefits of MCP integration in Windsurf .

• How to configure Windsurf and use the mcp\_config.json to add servers .

• Installing and setting up specific servers for database access (SQLite/Postgres with Neon) , API interaction (Fetch for FastAPI), repository management (Git) , UI scaffolding (React) , and web search (Brave) .

• A step-by-step example of building a user auth module where Cascade utilized those tools to create a cohesive implementation.

• Best practices to optimize performance and maintain safety, along with troubleshooting tips for common issues (like PATH problems or tool misuse) .

With this knowledge, you can empower Cascade to do more of the heavy lifting in your development process. As you gain confidence, you might even extend this with additional MCP servers (for example, Docker management if your app needs containers, or cloud deployment tools to let Cascade help with infra). The key is that Cascade becomes not just a coding assistant, but an AI partner that can operate on your code, data, and environment in a controlled, efficient manner. Happy coding with Windsurf and Cascade’s MCP superpowers!

Sources:

1. Codeium Windsurf Documentation – Cascade MCP Integration

2. Neon Tech Guide – Using Windsurf Cascade with Neon Postgres MCP

3. Glama MCP Server Directory – SQLite MCP Server Overview

4. MCP.so Directory – Git MCP Server (features & usage)

5. Glama – React MCP Server (community)

6. Glama Blog – MCP Quickstart with Brave Search

7. Neon Cascade Example – Cascade adding a DB column (conversation log)

8. Codeium/Neon Guide – Security reminder about Cascade commands

9. Stack Overflow – MCP server spawn PATH issue (Windows npx)