ChatGPT

MCP Server MVP Implementation Guide

Overview: Purpose and Role of the MCP Server

The **Model Context Protocol (MCP)** server acts as the coordination hub between users, external data sources, and AI agents (LLM-based assistants like ChatGPT or Claude). Its core purpose is to provide *context-aware interactions* – ensuring that AI agents have access to relevant user context, tools, and memory when generating responses. MCP was introduced as an open standard to connect AI systems with the data they need, replacing fragmented one-off integrations with a unified protocol [1]. In practice, an MCP server bridges AI assistants to real-world information securely and seamlessly. For example, an MCP server can expose enterprise data, personal files, IoT devices, or web APIs to an AI agent in a standardized way, turning previously isolated data into a discoverable and callable ecosystem for the agent [2]. This means an agent like ChatGPT or Claude can *reason and act within trusted boundaries*, using the MCP server to fetch context (like your calendar events or smart home status) and to execute permitted actions on your behalf.

In an **MCP-based architecture**, the AI application (e.g. a chat client or IDE plugin) includes an MCP *client* component that connects to one or more MCP *servers*. The MCP server provides the AI agent with three types of capabilities [3]:

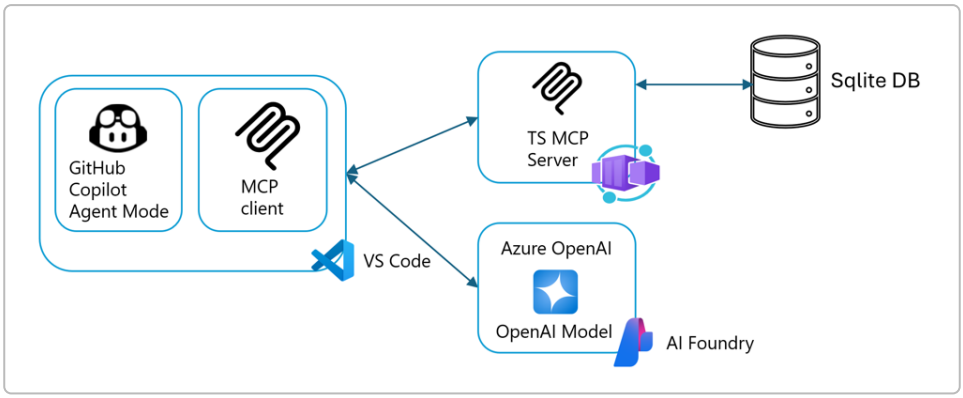
- **Tools (AI Actions):** Secure functions the agent can call to perform actions (e.g. send an email, create a calendar event). Tools have defined inputs/outputs and often require user approval before execution [4] [5].
- **Resources (Context Data):** Structured data sources that the agent can query or read as context (e.g. documents, calendar entries, sensor readings). Resources are exposed via URIs and can be fetched by the agent to incorporate into its prompt context [6] [7].
- **Prompts (Templates):** Pre-defined prompt or conversation templates that the agent or user can invoke for common tasks (e.g. “Plan my day” or “Summarize my inbox”), to guide interactions in a consistent format [8] [9].

By following the MCP protocol, the server and agent engage in a **two-way exchange**: the agent can request context or tool usage from the server, and the server can send back data or ask the agent to perform certain subtasks (like asking the user for input or logging a message) [10]. This design enables *context to persist and travel with the agent* across different tools and sessions. In essence, the MCP server is the memory and operations center for the agent: it “knows” about the user’s environment and state, and supplies that knowledge to the AI on demand [11].

Why is this useful? Without MCP, each integration (calendar, email, smart home, etc.) would require custom prompts or APIs the AI might not handle gracefully. MCP provides a *standard interface* for all such integrations [1]. For the developer, this means you can build one server that connects to many data sources and expose them uniformly. For the user, it means AI assistants become far more useful – they can refer to your actual calendar when scheduling meetings, control your home devices by talking to them, or personalize responses using your profile data. This context-awareness leads to smarter, more relevant AI behavior. As Anthropic notes, *MCP turns siloed environments into a unified, callable ecosystem for AI* [2].

In summary, the MCP server's role is to maintain **traceable shared context** between the human user and the AI agent. It ingests information about the user's world (state, history, external events), maintains that context securely (locally, in this MVP), and serves it to the AI agent or uses it to trigger helpful actions. The server orchestrates the interaction so that the AI is no longer operating in a vacuum, but rather is "plugged in" to the user's reality – much like a USB-C port for AI applications, providing a standard way to connect to various peripherals of data and functionality [12].

System Architecture and Components



High-level architecture of an MCP integration. In this example, an AI application (left, e.g. VS Code's Copilot with an MCP client) connects to an MCP server (right) which uses a local SQLite database for storage. The MCP server provides tools and context data to the AI agent, while the agent uses an LLM (e.g. Azure OpenAI) to fulfill user requests [13]. The MVP we build will use a Python/FastAPI server in place of the TS server shown here, but the core principles remain the same.

The MCP server MVP will be implemented in **Python** using **FastAPI** (for a web API interface) and **SQLite** (for local data storage). The architecture is designed in a *modular* way, separating concerns into different services or components. This modular design follows best practices for MCP servers: *"Modularize: Build MCP servers with modularity in mind"* [14]. Each module handles a specific aspect of context management or agent interaction, making the system easier to extend and maintain. The major components of the architecture are:

- **Trace Memory Service:** Responsible for logging and storing interaction traces (conversation history, events, and actions). This acts as the long-term memory for context.
- **Agent Routing Service:** Handles communication with AI agents (LLMs) and routes requests or queries to the appropriate agent if multiple are in use. It manages multiple agent endpoints (e.g. directing some queries to ChatGPT vs Claude) and coordinates multi-agent workflows if needed.
- **User State Modeling Service:** Maintains a structured representation of the user's state, including a user profile and dynamic state snapshots (current context like time, location, recent activity). This service updates the state based on incoming context and makes it available for context frames.
- **Rule-Based Nudging Service:** Implements a simple rule engine that monitors the user's context and interaction trace to trigger *nudges* – proactive suggestions or alerts – based on predefined rules. It can push these nudges to the user or agent when certain conditions are met.
- **Integration Connectors:** A collection of adapters that connect to external systems (such as Google Calendar, Home Assistant, Telegram, etc.) to ingest context or perform actions. Each connector module handles authentication and data exchange with the external service, translating external events into the MCP server's internal format (and vice versa).

All these components run within a single FastAPI application (for the MVP), but they are logically separated. **FastAPI** will expose a set of HTTP endpoints (a RESTful API) that allow clients – whether that's a front-end UI, an automation script, or an AI agent's MCP client – to interact with the server. The use of HTTP+JSON as the transport (instead of MCP's stdio transport) means our server can run as a standalone web service accessible to any authorized client (optionally including a web frontend). We will ensure that the API design aligns with MCP's data models (tools, resources, etc.) so that the MVP can be compatible with MCP clients. The local SQLite database will persist all data (context logs, user profile, etc.), emphasizing a *local-first* design: user data stays on the user's machine by default for privacy [15].

Below, we detail each component of the architecture and how they work together:

Trace Memory Service (Context Trace Logging)

The Trace Memory Service keeps an **episodic memory** of interactions and context changes, which is essential for context-aware AI behavior. Every significant event – user queries, agent responses, tool invocations, external context updates – is recorded in a **trace log** (stored in SQLite). This chronological log (or *timeline*) of events can be used to provide historical context to the AI or for debugging and audit purposes. Traceability is a key design goal: *"No eval = no trust. Design for LLM traceability/ explainability"* [16], meaning we should log what the AI was asked, what it did, and why. In this MVP:

- Each log entry might include a timestamp, event type (e.g. "user_message", "agent_response", "context_update", "tool_execution"), the content or data involved, and references to related context (like which user or which agent session).
- The service provides methods to **append** new trace entries and **query** recent history. For example, when the user sends a message via the UI, we log it; when the agent replies or calls a tool, we log those as well.
- We can also implement summarization or pruning strategies if the log grows too large (MVP focus is on basic logging, but in the future, a background job might summarize old conversations into a condensed memory).

By maintaining this trace, the server can furnish the agent with **relevant conversation history** on demand (e.g. "the last 5 messages") to maintain continuity beyond the agent's short-term memory. It also allows after-the-fact analysis of what the agent was doing – crucial for debugging and for responsible AI use. In essence, the trace memory is the **long-term memory** module of our agent system.

Agent Routing Service (Multi-Agent Orchestration)

The Agent Routing Service manages the interfaces to one or more AI agents. In a simple scenario, you might have just one agent (say OpenAI's ChatGPT via API) that handles all user queries. However, an advanced personal assistant might use **multiple specialized agents** and delegate tasks between them – for example, one agent specialized in scheduling, another in Q&A, or even a pair of agents that debate each other's answers for better accuracy. Our architecture plans for multiple agents by abstracting an "Agent" as a configurable entity:

- We will define **Agent definitions** (in a config or database) that include details like the model/API to call (e.g. OpenAI GPT-4, Anthropic Claude v2), credentials or API keys, and any role parameters (for instance, an agent might have a description like "Finance Advisor Agent" vs "Home Automation Agent"). This allows the server to know how to contact each agent.
- The Agent Routing Service exposes methods or endpoints for **invoking an agent** with a given prompt or request. It will take care of injecting the appropriate context (context frame) into the

prompt and calling the external API. For the MVP, this could be as straightforward as a function that takes a prompt string and forwards it to the agent's API, then returns the agent's response.
- If multiple agents are available, the service can implement simple **routing logic**: e.g., route the request to a specific agent based on user selection or based on the content of the request. (For instance, if the user query is about code, use a "CodingAgent"; if it's a general question, use a "GeneralAgent".) In MVP, the routing can be manual or rule-based (configurable).

By centralizing agent communication here, we make it easier to later introduce *multi-agent orchestration*. As an example, one could implement a **planner agent** that breaks a complex task into sub-tasks and delegates to other worker agents – a pattern of *"Multi-Agent Orchestration"* where specialized agents collaborate [17]. The routing service would facilitate this by passing messages between agents or choosing the right agent for each sub-task. Another future scenario is *agent debate* or adversarial collaboration, where two agents (perhaps one acting as a "devil's advocate") receive the same context and we aggregate their responses. Our MVP lays the groundwork by supporting multiple agents and providing a clear interface to engage each. (Advanced coordination logic can be added on top of this in the future roadmap.)

Finally, the Agent Routing module is also responsible for **formatting context frames** for the agents. Before sending a prompt to an agent API, it will retrieve the latest context frame (from the User State and Trace Memory services) and prepend or embed it into the prompt according to the agent's needs. This ensures the agent always has up-to-date context, regardless of which underlying model is being used.

User State Modeling Service (Profile & Context State)

The User State Modeling Service manages data about the user and the environment that persists beyond a single conversation turn. We divide this into two parts: the **User Profile** (static or slowly changing information about the user) and the **User State Snapshots** (dynamic context that changes frequently).

- **User Profile:** This includes relatively static data like the user's name, preferences, key dates (birthday, work hours), frequent locations, contacts, etc. Essentially, anything you'd put in a personal assistant's "about the user" file. This could also include permissions or settings (e.g. whether the user allows the agent to auto-send emails or requires confirmation). The profile is stored in SQLite in a `user_profile` table and can be edited via an API or UI. Having a structured profile means the agent can be given factual information about the user when needed (for example, "<User>'s preferred language is French" or "<User> usually prefers morning meetings").
- **State Snapshots:** These capture the *current context* at a point in time – often reflecting external context from integrations. For example, a snapshot might include current time, location (if available from Home Assistant or device GPS), the next event on the calendar, current weather, unread email count, etc. The server will update this state whenever new context comes in (e.g., when the Google Calendar connector pulls in new events, or when Home Assistant sends an update that the user arrived home). Each snapshot is time-stamped and stored (e.g. in a `user_state` table), and the latest snapshot represents "now". We might also keep a short history of snapshots to see trends (like how the state changed throughout the day).

When the agent requests context, the User State service can supply a **context summary** of the user's situation. For instance: *"It's 5:45 PM, you are at home, and you have an upcoming event 'Dinner with Alice at 6 PM' on your calendar. You have 2 unread important emails."* This kind of information can dramatically improve the relevance of the AI's responses. It essentially gives the AI situational awareness. The MCP

concept of transferring *memory and state across domains* is crucial here – by modeling user state explicitly, we ensure the AI can access this state when needed [18].

The User State Modeling also ties into privacy: all profile and state data is stored locally in SQLite. None of this sensitive data is sent to the cloud or to the LLM provider unless absolutely necessary for a query, and even then it can be filtered. (For example, if your profile contains your address but the agent doesn't need it, we won't include it in context.) Future plans like **Federated Learning** would rely on this component – if the system learns user preferences over time, those preferences live in the profile/state, and only anonymized or model updates might be shared out (more on that in the roadmap).

Rule-Based Nudging Service (Proactive Assistance)

One of the powerful features of a context-aware assistant is the ability to **proactively assist or notify** the user, rather than just reacting. The Nudging Service implements this by using a set of **if-then rules** that run whenever new context or events arrive. Each rule defines a condition to watch for and an action (nudge) to perform if the condition is met. This component is essentially a lightweight rules engine tailored to personal assistant behaviors.

Some examples of nudging rules we might include in an MVP:

- **Calendar reminder nudge:** *If* a meeting event is starting within 10 minutes and the user hasn't opened the meeting link *then* send a reminder (e.g. "You have a meeting in 10 minutes. Do you want me to open the video call link or draft a quick agenda?").
- **Location-based nudge:** *If* the user's Home Assistant reports they just arrived home and it's evening *then* push a greeting or context update (e.g. "Welcome home! I've adjusted the thermostat and here's a summary of what you missed today.").
- **Inactivity nudge:** *If* no user interaction has occurred for some time but there are pending to-do items *then* suggest one (e.g. "It's been a quiet afternoon. You mentioned needing to finish a report – shall I help you outline it?").
- **Email summary nudge:** *If* a new important email arrives (context from Gmail integration) *then* alert the user or summarize it (this could tie into an agent tool that summarizes emails).

Each rule's condition can be based on the **user state, recent trace events, or external triggers**. The Nudging Service listens to updates (it could be implemented by subscribing to the Trace and State services' events). When a condition is satisfied, the service executes the nudge. The nudge itself could be delivered in various ways depending on context: it might send a message via Telegram to the user, or create a notification in the front-end UI, or even speak via a text-to-speech if such output exists. In our MVP, a simple approach is to log the nudge in the trace (so it's visible in UI) and optionally send it through a channel like Telegram for real-time user notification.

Importantly, we design nudges to be *helpful and not intrusive*. They are essentially scripted suggestions arising from user context. The concept of nudging is already emerging in AI agents – for example, a "Customer Journey Agent" might monitor user behavior and *"nudge conversion through personalized flows"* in a retail context [19]. In our case, the nudges are personal and based on explicit rules the user (or developer) sets, so the user remains in control of what proactive behaviors the assistant has.

Under the hood, rules can be stored in the database (a `rules` table) with fields like `condition` (which could be a simple expression or a reference to a Python function in our implementation) and `action`. For MVP simplicity, one might hard-code a few example rules in Python, but we will outline a schema so developers can add new rules easily. Over time, this could evolve into a more sophisticated

system (even using ML to generate or prioritize nudges), but rule-based logic is transparent and easier to validate for now.

External Integration Connectors

To gather context and perform actions, the MCP server connects with external systems through dedicated **connector modules**. For the MVP, we focus on a few key integrations:

- **Google Calendar Integration:** This connector uses Google's Calendar API to fetch events and updates from the user's calendars. It can periodically pull upcoming events (or subscribe to push notifications via Google's webhook system if configured). When new events or updates are fetched, the connector sends them to the Context Ingest API (described later), which updates the User State (e.g., next meetings, free/busy status) and Trace log ("New event on calendar: …"). The connector can also provide a **Tool** for the agent to create or modify events. For example, it might expose a tool `create_calendar_event` which the agent can call (with user approval) to schedule something [20] [21]. OAuth2 is used for authentication to the user's Google account, storing the token securely (likely in the database or config). This integration ensures the assistant is aware of the user's schedule and can help manage it.
- **Home Assistant Integration:** Home Assistant is a popular local smart home hub. The MCP server can integrate with Home Assistant in two ways: (1) Home Assistant itself now offers an MCP server integration [22], but in our architecture we assume we pull data from Home Assistant's API. Using a Long-Lived Access Token (or OAuth with IndieAuth, as HA supports MCP OAuth flows [23]), our server can query Home Assistant for entity states (e.g., lights, thermostat, presence, sensors) and subscribe to events. For instance, we might get an event when the front door is unlocked or when a motion sensor triggers. These become context updates (e.g., "motion detected in living room at 7:00 PM"). We can also expose Home Assistant's services as Tools – e.g., a `turn_on_light` tool for the agent. This allows the user to say "turn off the kitchen lights" and the agent, via MCP, will call our server's tool which then calls Home Assistant to perform it. With this integration, a user can control their smart home by chatting with the agent, as reported by early adopters (e.g., controlling lights from Claude AI [11]). We will carefully limit which devices are exposed, to align with user's privacy and safety settings.
- **Telegram Integration:** Telegram is used in our design as a two-way communication channel with the user. We can set up a Telegram Bot (using the Telegram Bot API) that the user can interact with. The connector would receive messages from the user (via Telegram webhook or polling) and forward them to the MCP server as user input (which could then be answered by an agent). Conversely, the server can send messages to the user through Telegram, which is useful for **nudges or alerts** when the user is not at their computer. For MVP simplicity, Telegram can serve as a convenient front-end: you could chat with your personal AI on your phone through Telegram, and the bot's backend is the MCP server routing queries to the agent. This integration requires storing the Bot API token and verifying the user's chat ID to ensure we respond only to the authorized user. It demonstrates how an MCP server can integrate with messaging platforms to extend its reach. (Other messaging or voice platforms could be added similarly.)
- **Other Integrations (future):** The modular design means we can add connectors for things like Email (Gmail API), Task Managers (e.g., Todoist, as many have done via MCP [24] [25]), Web Search (using an API or DuckDuckGo as referenced [26]), or any REST API. Each connector will follow the pattern of either polling or receiving webhooks, then calling the server's context ingest endpoint with new data. Likewise, if an external action is needed, the server will invoke the service's API (e.g., send a POST to a webhook, call a REST endpoint, etc.).

All integration modules must handle **authentication securely** (e.g., API keys, OAuth tokens are stored in a config file or database securely, not exposed). The FastAPI server could use dependency injection or

environment variables for these credentials. Connectors might run as background tasks (for polling) or as separate threads that subscribe to events. Since our focus is MVP, we can simplify by requiring the user to provide API keys or tokens in a config, and doing periodic polling (e.g., check Google Calendar every 5 minutes).

Integration data flows into the server via defined APIs, which leads us to the next section: the API specifications.

API Specifications

We will design a set of RESTful HTTP API endpoints for the MCP server. These endpoints allow both **external systems** and **client applications (including a potential front-end or AI agent)** to interact with the server's functionalities. All endpoints will be implemented in FastAPI, leveraging Pydantic models for request/response schemas. Here are the key endpoints and their purpose in the MVP:

Context Ingestion Endpoints

These endpoints handle incoming context data (external events or updates):

- `POST /context/ingest` – Ingest new context information. External integrations (like a Google Calendar webhook, or a Home Assistant event) will call this endpoint with a JSON payload describing the context update. For example, Google Calendar might POST a payload like `{"source": "google_calendar", "event": {"id": "...", "title": "Meeting with Bob", "start": "2025-08-02T15:00:00"}}`. The server will authenticate the request (e.g., via an API key or token from the integration) and then process the update: store it in the SQLite DB (e.g., as an event in a Calendar table), update the User State (next meeting time etc.), and log the update in the Trace Memory ("Calendar event added/changed"). This endpoint returns a simple acknowledgement (200 OK or a status message).
    - *Authentication:* For local hooks, we can assume trust (if running on localhost). If exposed remotely, use a secret token or Basic Auth that the integration includes for verification.
    - *Processing:* The internal logic will likely route the payload to the appropriate connector handler. For MVP, it can be a single endpoint with conditional logic on `"source"` field.
- `POST /actions` – (Optional) This endpoint can be used to ingest *user or agent actions* that aren't just natural language messages. For instance, if the user performs an action in a GUI (like clicking a button "snooze reminder") or the agent completes a tool action, we might POST that to `/actions`. The payload could be `{"type": "user_snooze", "details": "User snoozed meeting reminder for 10 minutes."}` or `{"type": "agent_tool", "tool": "create_event", "status": "success", "result": {...}}`. Logging these as actions keeps the trace complete. This is not strictly required if all actions ultimately stem from either context updates or agent messages, but it provides a structured way to record things that happen. It also allows an external UI to inform the server of user-driven events (like a user confirming/denying a tool execution via a button click in the frontend).

Context Retrieval Endpoints

These endpoints allow clients (UI or agents) to retrieve context data or frames from the server:

- `GET /context/frame` – Retrieve a **Context Frame** for the current moment (or for a given conversation session). A *context frame* is a structured compilation of all relevant context for the user-agent interaction at a given time. This typically includes: a summary of user state (from the User State service), recent conversation turns or events (from Trace Memory), and possibly any active nudges or alerts. The response will be a JSON object encapsulating this info. For example:

```
{
  "timestamp": "2025-08-02T17:45:00Z",
  "user_profile": {
    "name": "Alice",
    "time_zone": "Australia/Hobart",
    "preferences": {"meeting_length": "30min", "language": "en"}
  },
  "current_state": {
    "location": "home",
    "next_event": "Dinner with Alice at 18:00",
    "unread_emails": 2,
    "weather": "10°C, Clear"
  },
  "recent_interactions": [
    {"time": "17:30", "sender": "user", "message": "Any updates for this
evening?"},
    {"time": "17:30", "sender": "agent", "message": "It looks like you have
dinner at 6 PM. No other updates."}
  ],
  "nudges": [
    {"id": 101, "text": "Reminder: Dinner with Alice at 6 PM - leave by 5:50
to be on time.", "priority": "info"}
  ]
}
```

This frame gives an agent everything it might need to know at that point. The content and format of the context frame can evolve, but the idea is to provide *structured context* rather than just dumping all logs (akin to the "structured context frames" used in some AI systems [27]). Clients can call this endpoint before invoking the LLM to get context, or an agent could call it during a conversation when it needs fresh context (e.g., "What's the latest state?").

We could allow query parameters like `?since=<timestamp>` to get changes since a time, or `?session=<id>` if multiple sessions. MVP can start simple (always give current full context snapshot).

- `GET /resources` – (Optional, MCP-specific) If we align with MCP spec for resources, we might implement endpoints like `GET /resources/list` and `GET /resources/{id}` to list available resource URIs and retrieve their content [28]. For example, the server might define resources like `calendar://events/today` or `home://sensors/temperature` which, when fetched, return data in JSON or text that the agent can consume. This adds formality to context retrieval. However, since `GET /context/frame` already composes multiple pieces, we might

not need a full resources API for MVP – instead, we embed needed data directly in the frame. We mention this because an MCP client might expect to call `resources/list` per the standard. Adapting to that standard could be a next step (e.g., mapping our context frame pieces to resource URIs).

Interaction (Conversation) Endpoints

If the MCP server also intermediates the conversation between user and agent (in case we have our own frontend or Telegram), we need endpoints to handle those messages:

- `POST /chat` (or `/message`) – Accept a user's message and get a response from the agent (with context). This would effectively combine context retrieval and agent invocation in one step. The payload might be `{"user_message": "Hey, what do I have scheduled tomorrow?"}` and the server will handle it by:
- Logging the user message (Trace Memory),
- Fetching context frame (User State says what tomorrow's schedule is, etc.),
- Sending the user query + context to the LLM agent via the Agent Routing service,
- Getting the agent's response,
- Logging the response and any tool usage,
- Returning the agent's answer to the caller.

This endpoint essentially turns our MCP server into a unified backend for a chatbot interface. If we are using a Next.js front-end, this is the endpoint the front-end would call to get replies from the AI. It's optional in the sense that if an AI application is itself the MCP client (like Claude Desktop), that app might connect and pull context directly rather than using this endpoint. But for a web-based UI or Telegram bot, having the server manage the full round-trip is convenient.

The response would include the assistant's message, and possibly structured data about any actions: e.g. `{"answer": "<text>", "actions": [ {...} ]}`.

- `GET /history` – Fetch past conversation or events, e.g. for display in a UI. This would query the Trace logs and return recent messages and actions. This is essentially a filtered view of the trace for the chat history. You might allow parameters like `?limit=50` or `?since_id=...`. The front-end can poll this to update the conversation view or show a timeline of nudges and events.

Administrative Endpoints

These endpoints allow managing or inspecting the server's data and configurations:

- `GET /agents` – List the configured agents (their names/IDs and statuses). For example, returns `[{ "id": "openai_gpt4", "model": "gpt-4", "type": "openai", "status": "online"}, ...]`. This helps a UI show which AI backends are available.
- `POST /agents` – Register a new agent or update config (e.g., provide a new API key or switch model). In MVP we might not implement a full agent management API, but it's worth designing for extensibility.
- `GET /rules` – List all active nudging rules and their conditions (so developers or advanced users can see what automatic behaviors exist).
- `POST /rules` – Add or modify a rule. For example, post a JSON like `{"condition": "state.location == 'home' and 18 <= hour < 19", "action":`

`"send_dinner_reminder"}` to add a rule. In MVP, editing rules might also be as simple as editing a config file, but an API makes it possible to build a UI for it later.
- `GET /profile` – Retrieve the user profile data. (In practice, might be part of `/context/frame` already, but separate endpoint could allow editing.)
- `POST /profile` – Update user profile fields (e.g., change preference or add info).
- `GET /metrics` (optional) – Could return basic stats like number of context updates processed, last agent response time, etc., for debugging and performance monitoring. Not essential for MVP but useful for developers.

All these endpoints should be documented with their request/response schemas. Using **OpenAPI** (which FastAPI can generate automatically) is useful so that both developers and AI agents (in the future) can understand the interface. In fact, an interesting possibility is generating an OpenAPI spec for our MCP server and using an automated tool to create MCP clients or even stub out server code (there are projects like *AutoMCP* that generate MCP servers from OpenAPI specs [29], showing the synergy between clear API specs and MCP tooling). We will include example requests and responses in this guide for clarity.

Note on JSON-RPC vs REST: The official MCP specification uses JSON-RPC 2.0 for communication between clients and servers [30]. In our implementation, we are exposing a RESTful API for simplicity (each endpoint corresponds to an MCP method or a function). One could implement a JSON-RPC over HTTP endpoint to adhere strictly to MCP spec (for example, a single `/rpc` endpoint that accepts JSON-RPC payloads for `tools/list`, `resources/read`, etc.). However, given FastAPI's strength in REST and our need for an MVP, we opt for straightforward REST endpoints. This doesn't prevent us from being MCP-compatible – conceptually each REST call corresponds to an MCP operation. For instance, `GET /context/frame` is analogous to the agent performing multiple MCP calls like `resources/list` and `resources/read` for various resources. As we iterate, we can add a JSON-RPC layer or adjust the interface to fully align with the MCP spec, but the MVP prioritizes developer accessibility and testability.

Example API Usage Flow

To make this concrete, here's an example **workflow using the APIs**:

1. **External Context Update:** The user adds a new event on Google Calendar (say via their phone). Google's API sends a webhook (through our integration) to `POST /context/ingest` with the event details. Our server authenticates the request, logs the new event, updates the state (next_event = "Meeting at 3 PM tomorrow"), and triggers the Nudging Service. One rule sees a meeting was added for tomorrow with a guest, so it might prepare a nudge: "You have a meeting with Bob tomorrow at 3 PM. Want me to draft an agenda?" The nudge is stored (and perhaps sent via Telegram immediately or scheduled for the morning).
2. **User Interaction (Query):** That evening, the user opens the Next.js frontend (or sends a message "What's on my agenda tomorrow?" via Telegram). The front-end calls `GET /context/frame` to show the user a snapshot of their context on the dashboard (it displays upcoming events, etc.). It then calls `POST /chat` with the user's query. The server receives the query, logs it, fetches a context frame (which now includes the new event and any relevant nudges or notes), and calls the Agent Routing to forward "What's on my agenda tomorrow?" along with context to, say, the default agent (Claude). The agent responds with: "You have a meeting with Bob at 3 PM. No other meetings – maybe use the morning for focused work. I have drafted a tentative agenda for your meeting if you'd like to review." The server logs this response and returns it to the front-end (which displays it to the user).

3. **Agent-Initiated Action:** The user says "Yes, show me the agenda." The agent then calls a Tool (via the MCP interface) – for instance, a `generate_document` tool that our server provides to create an agenda outline. The agent's request comes through (perhaps through `POST /chat` or a specific `/tools/execute` endpoint if we had one). Our server receives the tool call, logs it, perhaps asks for user confirmation (this could be another API call or a prompt to the UI – tools require user approval [31] [5]). Suppose the user approves via UI (which calls `POST /actions` to notify approval). The server executes the tool logic (maybe just uses the agent's output to create a text, or calls an external API if needed), then returns the result to the agent (through the same channel). The agent then includes the agenda content in its final reply. All these steps are recorded in Trace Memory (with entries like "Agent called tool X", "User approved tool X", "Tool X result delivered").
4. **Proactive Nudge Delivery:** The next morning at 9 AM, our Nudging Service sees the rule for meeting reminders triggers (meeting at 3 PM upcoming in 6 hours). It creates a nudge: "Reminder: Meeting with Bob at 3 PM today. I can send him a confirmation or prepare notes if needed." The server sends this to the user via Telegram (as a message from the bot). The user gets notified even without explicitly asking – a proactive assist. If the user replies to the bot, that goes back into `POST /chat` and the cycle continues.

This illustrates how the API endpoints come together to enable a rich interaction. We have **ingestion (from external triggers), context assembly, conversation handling, tool execution, and proactive notifications** all playing roles. Next, we'll dive into the data models that make this possible.

Data Models and Schema Design

To implement the above functionality, we need to design the database schema and corresponding data models (using Pydantic for FastAPI and possibly SQLAlchemy or similar for SQLite persistence). Below are the key data entities in the system and their schemas:

User Profile

The `UserProfile` contains static user information and preferences. In the database, this could be a table with one row (for a single-user system) or multiple if supporting multi-user. Key fields:

- **user_id** (integer primary key)
- **name** (string)
- **email** (string, optional)
- **preferences** (JSON or separate columns for specific prefs; e.g., preferred_language, notification_settings, etc.)
- **created_at**, **updated_at** (timestamps for record keeping)

Using Pydantic, we could define it as:

```
class UserProfile(BaseModel):
    user_id: int
    name: str
    email: Optional[str] = None
    preferences: Dict[str, Any] = {}
}
```

For MVP, this might be as simple as a dictionary stored in a JSON column. It provides context like **who the user is and how the assistant should behave** (tone, languages, etc.). This data will often be embedded in context frames to give the agent grounding in who it's assisting.

User State Snapshot

A `UserState` (snapshot) represents dynamic context at a certain time. We might keep a table of these snapshots over time or just update a single current-state record. Fields might include:

- **state_id** (primary key, or use timestamp as key)
- **timestamp** (when this snapshot was taken)
- **user_id** (foreign key to UserProfile)
- **location** (string or JSON e.g., `{"latitude": ..., "longitude": ...}` or a place name; could be derived from Home Assistant or device)
- **activity** (string describing what the user is doing, if known – e.g. "In a meeting" or "At home, idle")
- **status** (maybe a high-level status like "available" vs "busy")
- **next_event** (text or JSON with details of the next calendar event)
- **unread_count** (unread email count, or similar notifications)
- **custom_states** (JSON field for any other integration-specific states, e.g., `{"weather": "Clear 10°C", "lights_on": 2}`)

Pydantic example:

```
class UserState(BaseModel):
    timestamp: datetime
    location: Optional[str]
    activity: Optional[str]
    next_event: Optional[str]
    unread_emails: Optional[int]
    extra: Dict[str, Any] = {}  # for other contextual info
}
```

In the database, some of these could be columns, others could reside in a JSON blob (`extra`). The latest UserState is what we use for context frames. We might link it to trace entries as well (e.g., when logging "user arrived home", we create a state snapshot and also a trace log entry of that event).

Context Frame

While not necessarily stored as a separate table, the **ContextFrame** is a structured view composed from Profile, State, and recent Trace. We define its schema logically for documentation:

- **timestamp**: when the frame is generated.
- **user_profile**: an embedded UserProfile (or subset of it relevant to context, like name and key preferences).
- **current_state**: embedded UserState (or selected fields of it).
- **recent_interactions**: a list of recent trace events (e.g., last N user and agent messages or last N significant events). Each interaction might include `{time, sender, type, content}`.
- **nudges**: list of active nudges waiting to be delivered or acknowledged, with fields like `{id, text, priority}`.

This is returned via API but we may not store it permanently (it can be regenerated anytime). However, for consistent replies one might cache the last frame used for a query (so the agent's response can reference it explicitly if needed).

Agent Definition

The `Agent` model defines an AI agent that the server can utilize. Fields:

- **agent_id** (string or short name, e.g., `"openai_gpt4"`, `"anthropic_claude2"`)
- **agent_type** (string enum: e.g., `"openai"`, `"anthropic"`, `"azure_openai"`, `"local"` etc. – defines which client library or API to use)
- **model** (string, the model name or ID, e.g., `"gpt-4"` or `"claude-v2"`)
- **api_key** (string, if needed; stored securely or as env var reference)
- **endpoint_url** (for custom endpoints or local models)
- **description** (optional human-readable description like "OpenAI GPT-4 via API")
- **enabled** (boolean, if this agent is active)

We can store agents in a table so that new ones can be added without code changes. In MVP, configuring via a config file is fine. The Agent Routing service will reference these definitions to know how to call each agent. For example, for `agent_type="openai"`, we'd use OpenAI's SDK with the provided `api_key` and model.

We could also include a **role prompt or persona** in the definition (e.g., a default system prompt that the agent should always get), for specialized agent behaviors. That would be a text field.

Tool Definition (if applicable)

Tools are actions that the agent can invoke on the server. In our context, many tools correspond to integration capabilities (e.g., schedule a meeting, send a telegram message, toggle a device). A `Tool` model might include:

- **tool_name** (string, unique)
- **description** (string, what it does)
- **input_schema** (JSON schema for the tool's parameters, as MCP uses JSON Schema [4])
- **output_schema** (JSON schema for the result, if any)
- **allowed_agents** (which agents can use it, if restrictions apply)
- **implementation** (this could be a reference to a Python function or script that is executed when the tool is called)

For MVP, we might not formalize this fully in the DB – it could be a Python dict or simple registry in code. But it's useful to outline. For example, a tool definition for sending a Telegram message might be:

```
{
  "name": "sendMessage",
  "description": "Send a Telegram message to the user",
  "inputSchema": {
    "type": "object",
    "properties": {
      "text": {"type": "string", "description": "The message to send"}
    },
```

```
    "required": ["text"]
  },
  "outputSchema": { "type": "object", "properties": { "success": {"type":
"boolean"} } }
}
```

Multiple tools like this would be listed by an API (like `GET /tools`). The agent discovers them and can call via MCP. Each tool call results in an entry in the trace (and typically requires user permission to actually execute, ensuring safety [5]).

Rule (Nudge Rule) Definition

A `Rule` entry defines a nudging rule. Fields:

- **rule_id** (primary key)
- **name** (short identifier, e.g., `"meeting_reminder_rule"`)
- **condition** (text or code – could be a simple expression evaluated against the state and events. MVP might use a Python lambda or a small DSL like `"if next_event_time - now < 30min then ..."` stored as text.)
- **action** (text describing the action or referencing an action type, e.g., `"send_nudge('Meeting starting soon')"`)
- **enabled** (bool)
- **last_triggered** (timestamp of last time it fired, to avoid spamming).

In MVP, implementing a full rules engine is complex, so we might simplify by hardcoding a few rule checks in the Nudging Service loop. Still, having a data representation means we can allow enabling/disabling specific rules without code changes. For instance, a developer could toggle off the "meeting reminder" rule by updating a DB flag.

Trace Log Entry

The Trace logs in SQLite will capture events as they happen. A `TraceLog` table might include:

- **log_id** (primary key)
- **timestamp**
- **event_type** (string: "user_message", "agent_message", "tool_usage", "context_update", "nudge", etc.)
- **source** (who/what caused it: "user", "agent:<id>", "system")
- **content** (text of message if applicable, or description of action)
- **metadata** (JSON for additional info, e.g., tool name and parameters, context keys updated, etc.)
- **session_id** (if we want to group logs by conversation session; could be null or a foreign key if multi-session is tracked)

In FastAPI we might not expose this entire log directly (except via history endpoints), but internally it's crucial for building context and debugging. It essentially forms the **knowledge base of "what has happened so far."** If we foresee many entries, indexing by event_type and timestamp will help querying recent chats vs all events.

Because all data is in SQLite, it's accessible on the local machine and can easily be inspected. SQLite's simplicity fits our local-first requirement (no external DB service needed). As usage grows, migrating to

PostgreSQL or another DB would be straightforward if we use an ORM with migrations. But for an MVP, SQLite with direct SQL or SQLAlchemy will do fine.

ER Diagram: Summarizing relationships, if we were to draw an entity-relationship diagram, it would look like:

- **UserProfile** (1) — (N) **UserState**: A user has many state snapshots.
- **UserProfile** (1) — (N) **TraceLog**: A user has many trace entries (though trace might include agent/system entries too).
- **Agent** (N): separate, not directly tied to user (or could be tied if personal per user, but here global config).
- **Rule** (N): can reference user or be global; can also possibly reference agent or context types in conditions.
- **Tool** (N): global definitions; trace logs reference tool usage events.

We will ensure **migration scripts or DDL** is provided for these tables so developers can set up the database quickly. (FastAPI can integrate with Alembic migrations if using SQLAlchemy.)

Below is a simplified table schema for reference (in SQL-like pseudo-code):

```
CREATE TABLE user_profile (
  user_id INTEGER PRIMARY KEY,
  name TEXT,
  email TEXT,
  preferences TEXT -- JSON string
);

CREATE TABLE user_state (
  state_id INTEGER PRIMARY KEY,
  user_id INTEGER REFERENCES user_profile(user_id),
  timestamp DATETIME,
  location TEXT,
  activity TEXT,
  next_event TEXT,
  unread_emails INTEGER,
  extra TEXT -- JSON
);

CREATE TABLE agent (
  agent_id TEXT PRIMARY KEY,
  agent_type TEXT,
  model TEXT,
  api_key TEXT,
  endpoint_url TEXT,
  description TEXT,
  enabled BOOLEAN
);

CREATE TABLE rule (
  rule_id INTEGER PRIMARY KEY,
```

```
  name TEXT,
  condition TEXT,
  action TEXT,
  enabled BOOLEAN,
  last_triggered DATETIME
);

CREATE TABLE trace_log (
  log_id INTEGER PRIMARY KEY,
  timestamp DATETIME,
  event_type TEXT,
  source TEXT,
  content TEXT,
  metadata TEXT, -- JSON
  session_id TEXT
);
```

This schema can be adjusted as needed, but serves as a baseline for implementing the MVP.

Integration Points with External Systems

As discussed, the MCP server will integrate with external systems to both **ingest context** and **perform actions**. Here we outline how the MVP will interface with the examples given (Google Calendar, Home Assistant, Telegram), along with notes on other potential integrations.

Google Calendar Integration

Purpose: Enable the AI assistant to be aware of the user's schedule and to manage events (create/update/delete events via agent commands).

Ingestion: We can use Google's Calendar REST API. A service account or OAuth 2.0 will be needed for access. In a personal assistant scenario, the user likely goes through an OAuth flow to authorize our MCP server to read their calendar (since it's local, an OAuth device flow or installed app flow might be used, resulting in tokens stored in our server). For MVP simplicity, we might allow the user to place a Google API refresh token in a config file after a one-time manual authorization.

Once authorized, our server will use Google Calendar API endpoints (via the Google Python client or direct HTTP calls) to fetch events. Two modes are possible:

- **Polling:** The server periodically calls *Events.list* for primary calendar (and any other calendars of interest) for events in the near future (say next 24 hours) and compares with what's already known. New or changed events trigger `POST /context/ingest`.
- **Push Notifications:** Google Calendar supports webhooks (channel subscription). However, setting that up might be overkill for MVP (requires a public endpoint with HTTPS, not trivial for a local server). Polling might suffice initially.

Data Handling: The Calendar events fetched will be transformed into our internal representation (e.g., next_event in UserState, or a resource list of today's events). The context frame might include a summary like "Tomorrow: [Event A at 10am], [Event B at 3pm]" based on this data.

We also treat the Calendar as a **Tool** for the agent: The agent can request to create or modify events. For instance, if the user says "Schedule a meeting with Bob tomorrow at 2 PM", the agent (with the prompt understanding) could invoke a `schedule_meeting` tool on our server. Our server's implementation of that tool will call Google Calendar's *Events.insert* API to create the event, then return success or details (and update our context). Because MCP tools must be user-approved [5], the workflow might be: 1. Agent says it wants to use `schedule_meeting` with params. 2. Our server sees this and either automatically allows it (if user pre-approved certain actions) or asks the user via UI ("Agent wants to create an event… Allow?"). 3. If allowed, server calls Google API, creates event, logs it, and possibly notifies the user ("Event created").

Security: The Google OAuth tokens are sensitive; store them in SQLite encrypted if possible, or in a file with restricted permissions. Also, ensure the scope is limited (just Calendar access, not everything). The integration should respect the user's privacy – e.g., maybe by default only reading busy/free or event titles unless more detail is needed, depending on user comfort.

Status in MCP ecosystem: Many have built Google Calendar MCP connectors [32], so we have examples to follow. Ours being local-first and integrated means the user isn't giving a third-party service their data – it stays in their personal server. In the future, the server could even function offline with cached data and only sync when network is available.

Home Assistant Integration

Purpose: Leverage smart home data and control. Home Assistant (HA) can provide a wealth of context (e.g., whether the user is home, room temperatures, device statuses) and also allow the assistant to act (turn things on/off, change settings).

Connection Approaches: - If the user already runs Home Assistant, the easiest path is to use Home Assistant's *MCP integration* directly [22] – essentially HA can act as an MCP server exposing its entities as tools/resources. However, if we want to keep our single MCP server architecture, we'll directly query HA's API. HA has a REST API and also a WebSocket API for real-time updates. We will likely use the REST API for simplicity, combined with long-lived access tokens for auth (which the user can generate in HA and paste into our config) [33]. - A simple method: poll HA's REST API for certain entity states (like person.location, or sensor values) periodically. A better method: open a WebSocket connection to HA's `api/websocket` which can stream state changes. For MVP, polling a few key states (like presence) every minute might be enough.

Data Ingestion: We decide which HA entities are relevant to import as context. For example: - The user's presence (`person.username` entity) – if it changes to "home" or "not_home", we ingest that as a context event ("User arrived home"). - Security sensors (if alarm goes off, maybe notify user via assistant). - Environment sensors (temperature, etc. could be context but not critical unless we want the assistant to comment "It's 10°C in the living room, you might want a jacket" – interesting but optional). - Device states: lights on/off, media playing – likely not needed in context unless user asks something like "Is the kitchen light on?" which the agent could answer via context.

So we configure the HA connector to watch a set of entities (the user can specify which, or by default just presence and a few basics). Each change on those triggers a `POST /context/ingest` with `{"source": "home_assistant", "entity": "light.kitchen", "new_state": "on"}` for example. The server then updates state (maybe `extra` field like `lights_on_count`) and logs it.

Actions: We expose **Home Assistant services** (their term for actions) as MCP tools. E.g., `turn_on_light` or a generic tool `call_service` which takes parameters `domain, service, entity_id`. But to make it user-friendly for the agent, we could define higher-level tools like `turn_on(device)` or `set_temperature(device, temp)` for thermostat. The agent could then say `turn_on("light.kitchen")` via the tool interface. Our server will receive that, map to HA's `light.turn_on` service call, execute it, and return a result. The Home Assistant MCP integration natively does similar things by exposing Assist API, but here we replicate needed bits in our server for control [34].

Security: Home Assistant is local, so the data stays on the local network. The user's HA token should be kept secret. Also, we must be careful which actions to allow; potentially dangerous actions (unlock doors, etc.) should absolutely require explicit user confirmation in real-time. Since it's all local, we rely on the user's existing HA security model (if someone can call our MCP API, presumably they already have system access; still, maybe use an auth token for our API too). We might include a config of allowed entity domains for agent control (maybe allow lights and switches, but not alarm system by default).

Value: Integrating HA truly grounds the AI in the physical context of the user. The user can ask "Did I leave any lights on?" – the agent can answer from context, or "Turn off everything downstairs" – the agent can do so via tools. It makes the AI assistant part of the smart home experience. And since both MCP server and HA are local, this can even work without internet (for local voice control etc., if the model is local or cached). The Home Assistant team's integration shows a commitment to using MCP for contextual smart home control [11], which validates our approach.

Telegram Integration

Purpose: Provide a text-based conversational interface and notification channel via a messaging app. Telegram is a convenient choice because it's easy to create bots and it's cross-platform.

Bot Setup: The developer (or user) would create a Telegram Bot using BotFather (getting a token). That token is configured in our server. We also need the chat ID of the user (which can be obtained by having the user start the bot and sending a message, or via an API call). For MVP, a simple method is: when the bot gets a message from an unknown chat, we assume that's our user and store that chat ID.

Receiving Messages: We can use the Telegram API in two ways: long polling or webhooks. Given our server might not be HTTPS-public, we likely use polling. For example, start a background thread that calls `getUpdates` on Telegram API every couple of seconds. When a new message arrives (text), the connector calls our `POST /chat` endpoint or directly processes it: - It will treat the Telegram message as a user input. So it can either call an internal function equivalent to `handle_user_message(user_message)` or literally do an HTTP call to our own `/chat` endpoint with the message (though that's redundant if we're inside the app; we can just call the logic directly). - It then waits for the agent's response (the `/chat` handling will produce a response via agent). - Once a response is ready, the connector sends it back via `sendMessage` through Telegram API to the user's chat.

This essentially makes the Telegram chat behave like a chat UI for the assistant. The user can be anywhere and still interact with their local MCP server (as long as the server has internet to talk to Telegram's servers).

Sending Messages (Notifications): For nudges or any proactive message, the server can call Telegram's `sendMessage` method to the user's chat ID. For example, when a nudge triggers, Nudging Service calls something like `telegram_bot.send_message(user_chat_id, text="Reminder: ...")`. This is a push mechanism that doesn't require the user to ask anything. It's great for timely alerts. The user will get a message from the bot as if a friend sent it.

Security & Privacy: All messages go through Telegram's servers, so while they are encrypted in transit, they are not E2E encrypted (unless using Secret Chats which bots do not support). Users should be aware of that. Sensitive data in nudges (like "Your front door is unlocked!" or personal info) is being sent over Telegram; if that's a concern, one could opt for a different channel or ensure the bot is private and trusted. For MVP, we use it for convenience. In the future, integration with a self-hosted messaging or Signal could be considered for more privacy.

Alternate/Additional Front-ends: If not Telegram, one could integrate Slack, WhatsApp (with an API), or simply use the Next.js web UI. Next.js could provide a richer interface (with buttons for approvals, rich formatting, etc.), whereas Telegram is plain text (with some button capabilities). There's no one "right" interface – the MCP server can serve multiple simultaneously. For instance, user might get notifications on Telegram but also see them in a web dashboard. The server should handle concurrent channels gracefully (this just means multiple clients hitting the API).

Other Integrations

We mention a few others to illustrate extensibility, though they might be beyond MVP scope:

- **Email (Gmail)**: Similar to calendar, using Gmail API to ingest new email subjects (context: e.g., unread important emails) and providing a tool to send emails or draft replies. This could let the assistant read you your email or send emails by command. (Already, integration lists show Gmail and Calendar frequently combined [24] [35].)
- **To-Do Apps (Todoist, Google Tasks)**: Ingest tasks due or completed. The assistant can remind you of tasks or mark them done when you tell it. A tool to add a task via natural language is useful ("Add 'buy milk' to my todo list").
- **Knowledge Bases (Notion, local files)**: To give the assistant long-term knowledge, one might integrate with a notes app or load local text/PDF files through an MCP fileserver. This turns the assistant partly into a personal knowledge management tool.
- **Web search / APIs**: If the assistant needs open-domain info, an integration with a search API can be a tool (though at that point, it's like any web-connected agent). But you could define a `search(query)` tool that hits DuckDuckGo or Google and returns results for the agent to summarize. This is outside pure "personal context", but useful for completeness.

Integrations are modular – they mostly communicate with the MCP server via the `ingest` endpoint and tool endpoints, meaning each integration can be developed and tested in isolation. This modular approach aligns with *the MCP philosophy of modular connectors* [14]. In our project structure, we might have a folder `integrations/` with modules like `google_calendar.py`, `home_assistant.py`, `telegram_bot.py`, etc., each encapsulating the API calls and providing a couple of hooks to the main app (like `start()` for starting listeners or scheduled jobs).

It's worth noting that each integration potentially brings in external dependencies (Google's client lib, etc.), which should be managed in requirements and made optional if not used.

Finally, we stress that **all integration data remains local** in our design. For example, even though we call Google's API to fetch calendar events, those events are stored in our local DB and not sent to any

other third-party. The only parties that see the data are the ones the user has authorized (e.g., Google already had the data, we just fetched it; Telegram will see messages we send to the user). There is no central server aggregating everyone's data – each user's MCP server is a personal silo. This local-first approach ensures *"raw data stays within local systems"*, preventing leakage of sensitive info [15].

Security and Privacy Considerations

Security and privacy are crucial in a system that centralizes personal context and interfaces with powerful AI agents. Our MCP server MVP is **local-first**, meaning by default it runs on the user's machine (or home server) and stores data in a local SQLite database. This gives a strong baseline of privacy: unlike cloud services, your context data (calendar, home status, etc.) remains with you and isn't uploaded to an AI provider or our servers. However, there are still several security and privacy factors to consider:

1. Authentication & Access Control: We should ensure that the FastAPI server is not openly accessible to unauthorized parties. For instance, if running on a personal laptop, binding the server to `localhost` (127.0.0.1) is a good default, so only the user's device can access it. If the user wants to access it from other devices (say running the server on a Raspberry Pi at home and accessing from phone), they should set up authentication. This could be as simple as an API token that all requests must include (via header). FastAPI can easily be configured with an `Depends(auth_function)` on all routes to enforce a check. For now, an **API key** in an environment variable that the Next.js front-end or Telegram bot also knows could suffice. In the future, we can integrate OAuth or similar standardized auth if needed.

2. Secure Integrations: Each external integration must be handled with care: - **OAuth Tokens** (Google, etc.): Store refresh tokens in encrypted form if possible (SQLite doesn't encrypt by default; one might use OS keyring or at least file-system protections). At minimum, limit their exposure by file permissions and not printing them in logs. - **Home Assistant Tokens:** These are essentially like passwords to your home; treat them with the same care. If possible, use HA's OAuth rather than long-lived tokens (which can be revoked easily and scope limited). - **Telegram Bot Token:** If someone obtains this, they could impersonate your bot. Keep it secret (not in public repo, etc.). Also consider restricting who can talk to the bot – e.g., check the user's Telegram ID against an allowed list so strangers can't spam your bot to get it to do things. - **Tool Execution and Permissions:** We will implement a user approval step for any tool/action that has side effects. By default, the agent can *suggest* an action (like sending an email or turning off lights), but the server will wait for user approval (through the UI or a confirmation message) before actually executing. This ensures that even if the AI is compromised or makes a mistake, it cannot do irreparable harm without user intervention. (In advanced usage, users might whitelist certain safe actions to auto-execute, but MVP assume manual approval for safety.)

3. Sandboxing AI Actions: Since the agent could ask the server to do various things, we ensure the server only does what it's explicitly programmed to. For example, if we expose a `run_python_code` tool for the agent (which some might be tempted to for automation), that's very dangerous – essentially remote code execution by the AI. We likely *do not* include such a tool in MVP. All tool implementations are carefully reviewed and limited in scope (like calling specific APIs, not arbitrary code). This is common sense but worth stating: the MCP server should not become a potential *malware server* under the control of a prompt injection. Keeping the rule "user must approve" mitigates this a lot, as does avoiding an overly broad toolset.

4. Encryption & Future E2EE Sync: While data is at rest in SQLite (which by default is plaintext on disk), we can rely on OS security (e.g., disk encryption if the user's device has it). For an MVP, we won't

implement custom encryption of the DB, but developers should be aware that if the DB contains highly sensitive info (like detailed logs of conversations, etc.), they should use it on a secure device or enable OS-level encryption. Looking ahead, if we allow syncing context across devices or backing up to cloud, we will employ **end-to-end encryption (E2EE)**. That means the data would be encrypted on the client (MCP server side) before sending, and only decrypted on the user's other device – no server in between can read it. There are frameworks and libraries to help with E2EE sync (for example, using something like Matrix or peer-to-peer CRDT databases). Those are beyond MVP, but we acknowledge them as part of the roadmap for a truly privacy-preserving system.

5. Privacy of AI Queries and Responses: When we send context to an LLM (like OpenAI or Anthropic), that data is leaving the user's machine and going to a cloud API. Users should be informed and give consent to what data is sent. For example, if the context frame contains the user's exact address or full calendar, and the user asks an OpenAI model a question that includes that frame, OpenAI's systems will see that data. Our design could include a **filter or redaction** step – perhaps marking certain profile fields as "private – don't send to AI" (for instance, you might not want to feed your full contact list or passwords to the prompt). The developer implementing the context assembly should consider an allowlist of what goes into the LLM prompt. Some sensitive info might be better summarized or omitted. Also, using providers that promise not to store data (OpenAI has a policy for their API where they won't use data for training if opted-out, etc.) can alleviate some concerns [36] (the *Governance* aspect of best practices).

6. Logging and User Control: All context and interactions are logged for traceability, but the user should have control over logs. There might be a way to **clear the history** or delete certain logs (especially if something sensitive was accidentally captured). At MVP, we can keep all logs for completeness, but we'll note that adding a "forget" mechanism is important for privacy rights (akin to GDPR "right to be forgotten" if this were a multi-user service). Since it's local, the user can always just delete the database file, but finer control is nicer.

7. Running Untrusted Code or Plugins: If the server in the future allows plugins or is extended by third-party modules, be cautious. For now, all code is first-party, so trust is within our implementation. But if loading external MCP servers or piping to external code, treat that as a potential security boundary. For example, if the agent connects to a remote MCP server (not our own) to fetch context from some service, ensure that the data from that server is sanitized if we integrate it.

8. Updating and Patching: This is more operational, but the developer should keep dependencies updated for security (FastAPI, uvicorn, requests, etc., should be recent versions to pick up any security fixes). If this MVP becomes a running service on someone's machine, they should update it periodically.

Local Deployment: By focusing on local deployment, we inherently reduce many security risks (no broad attack surface on the internet by default). The user can run the server on their device where they have existing security (firewalls, etc.). If one does expose it (for remote access from outside), using HTTPS (via a proxy or using something like Caddy/NGINX) and strong auth is recommended. We could integrate something like **TLS client certificates** for a truly locked-down access (a bit complex for average user, but possible for advanced ones).

Security and Privacy Roadmap: On the roadmap, beyond E2EE sync, we also consider: - **Differential Privacy**: If aggregating any usage data (say usage metrics to improve the agent's behavior or global model), we would add differential privacy noise so that no individual's data can be reconstructed from aggregates [37]. In a personal assistant scenario, DP might be applied if the user wants to contribute learning to a collective model without sharing exact data. - **Federated Learning**: We could allow the local MCP server to train local models (or fine-tune LLMs) on the user's data and only share model

weights updates (not raw data) with a central server or peer network. This would allow collective improvement of AI models while keeping raw data private. It's complex but some frameworks (TensorFlow Federated, etc.) exist. - **Continuous Security Audits**: Because MCP is new, we will watch the community for discovered vulnerabilities. For example, researchers have pointed out that *MCP, if not secured, can be a "privacy nightmare"* [38] or has potential for prompt injection via API inputs [39]. We must therefore implement validations – e.g., if the agent is sending a tool request with weird parameters, maybe flag it; or ensure that an agent's request to the server cannot directly call internal sensitive functions without going through our checks.

To sum up, our MVP will implement a **secure-by-design approach**: - local data storage and processing, - minimal attack surface, - user-in-the-loop for actions, - secure handling of credentials, - transparency (logging) for accountability.

These align with the mantra that *"the 'S' in MCP stands for Security"* (a play on MCP) – we treat security as a first-class concern even in this early implementation.

Developer Setup and Environment Configuration

Setting up the MCP server MVP for development is straightforward. We utilize common tools and aim for minimal friction:

Prerequisites: - Python 3.10+ installed on your system. - (Optional) Node.js 18+ if you plan to run a Next.js frontend. - Access tokens/API keys for any integrations you want to enable (Google API, Telegram bot, etc.). These can be obtained from the respective services (for development, you might skip configuring some integrations if not needed).

Project Structure: Assume our repository is structured like:

```
mcp_server/
├── app.py           # FastAPI application instantiation
├── models.py        # Pydantic models and/or SQLAlchemy ORM classes
├── db.py            # Database setup (SQLite initialization, sessions)
├── routers/
│   ├── context.py   # Routes for context ingest and retrieval
│   ├── chat.py      # Routes for chat messages
│   ├── admin.py     # Routes for profile, agents, rules management
│   └── ...
├── services/
│   ├── trace.py     # TraceMemoryService class
│   ├── state.py     # UserStateService class
│   ├── agent.py     # AgentRoutingService class
│   ├── nudges.py    # NudgingService class
│   └── integrations/
│        ├── google_calendar.py
│        ├── home_assistant.py
│        └── telegram_bot.py
└── config.example.env   # Example environment variable config
```

(This is just one way; you can organize as you see fit, but separation of concerns is key.)

1. Create and activate a virtual environment:

```
python3 -m venv venv
source venv/bin/activate   # on Windows: venv\Scripts\activate
```

2. Install dependencies: We'll use FastAPI and Uvicorn for the server, SQLAlchemy for DB (or you can use Tortoise ORM or Peewee if preferred), and httpx/requests for any HTTP calls to external APIs. For Telegram, the `python-telegram-bot` library is convenient (or just use `requests`). For Google API, `google-api-python-client` or `google-auth` may be needed.

An example `requirements.txt` might include:

```
fastapi
uvicorn[standard]
SQLAlchemy
python-dotenv     # for loading env config
requests          # for general HTTP
python-telegram-bot
google-api-python-client
python-homeassistant-api   # (if a library exists or we use requests)
```

Install with pip:

```
pip install -r requirements.txt
```

(*Note:* If not using a specific integration, you can omit its lib to keep things lightweight. For example, if not using Telegram, don't install it.)

3. Configure environment variables: Create a copy of `config.example.env` as `.env` (FastAPI can auto-load .env, or we use `dotenv`). In this file, you'll set up keys and config:

```
# Example .env content:
OPENAI_API_KEY=sk-... (if using OpenAI)
ANTHROPIC_API_KEY=... (if using Anthropic Claude)
GOOGLE_CLIENT_ID=...
GOOGLE_CLIENT_SECRET=...
GOOGLE_REFRESH_TOKEN=...
HA_URL=http://homeassistant.local:8123
HA_TOKEN=... (Long-lived token)
TELEGRAM_BOT_TOKEN=...
TELEGRAM_USER_ID=... (your chat ID to restrict access)
API_AUTH_TOKEN=... (a token for securing the FastAPI endpoints)
```

And so on for other configs (like database file path if we don't want the default).

Our code will read these via `os.getenv` or use Pydantic's Settings management. For secrets like API keys, environment variables (or a separate secrets file) are preferred over hard-coding. Document in the README how to obtain each credential.

4. Initialize the database: We might include a small script to set up the SQLite DB with tables. If using SQLAlchemy, simply importing the models and calling `Base.metadata.create_all(engine)` on first run can create tables. Alternatively, ship a SQLite file with migrations applied (not likely, easier to just create on first run). If needed, run:

```
python -c "from db import init_db; init_db()"
```

(This `init_db` function would create tables and maybe add a default user profile.)

5. Run the FastAPI server: Use Uvicorn to run the app:

```
uvicorn app:app --reload --port 8000
```

This assumes in `app.py` we have something like:

```
import fastapi
from routers import context, chat, admin
app = fastapi.FastAPI()
app.include_router(context.router)
app.include_router(chat.router)
app.include_router(admin.router)
# possibly startup events to init integrations:
# e.g., app.add_event_handler("startup", start_integrations)
```

The `--reload` flag is useful in development to auto-reload on code changes. In production, we'd run without `--reload` and possibly behind a reverse proxy.

6. (Optional) Run Next.js frontend: If a Next.js app is provided in, say, `frontend/` directory, instructions would be:

```
cd frontend
npm install
npm run dev
```

This would start the Next.js dev server on (by default) port 3000. Ensure the Next.js app knows where the FastAPI is (maybe via an environment var like `NEXT_PUBLIC_API_URL="http://localhost:8000"`). We should enable CORS on FastAPI to allow the Next.js origin to call it. This can be done with:

```
from fastapi.middleware.cors import CORSMiddleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000"],
```

```
    allow_methods=["*"],
    allow_headers=["*"]
)
```

during setup.

The Next.js app could provide a UI for viewing context (state, upcoming events), chatting with the agent, and managing settings (like toggling rules or editing profile). However, since Next.js is optional, one can also interact purely via API or CLI.

7. (Optional) Running the Telegram bot: If configured, the Telegram integration might start automatically on app startup (for example, in `start_integrations` we call something like `telegram_bot.start_polling()`). Ensure that the bot token is set and that the polling thread is not blocking the main thread (most likely it runs async or in background). The developer should watch the console logs to see that it connects and starts receiving messages. You can test by sending a message to your bot and observing the server log for processing, and the bot's reply.

8. Testing the setup: Try a simple manual test: using `curl` or a tool like HTTPie to call the API. For example:

```
# Check server is up
curl http://localhost:8000/context/frame
# (if auth is needed, include -H "Authorization: Bearer <API_AUTH_TOKEN>")
```

Initially, it might return mostly empty/default values. Then simulate an event:

```
curl -X POST http://localhost:8000/context/ingest -H "Content-Type:
application/json" \
    -d '{ "source": "test", "data": {"message": "Hello World"} }'
```

This is just to see that ingest doesn't error (our handler might just log it). Also try sending a chat message if we have `/chat`:

```
curl -X POST http://localhost:8000/chat -H "Content-Type: application/json" \
    -d '{ "user_message": "Hello" }'
```

The expected result would be an agent response in JSON. If the agent is properly configured (say OpenAI key provided), you should get a response. If using an LLM that requires internet, ensure your machine has connectivity.

9. Developer Workflow: We recommend running the FastAPI server in one terminal (with `uvicorn --reload`) and if using Next.js, running `npm run dev` in another. The reload feature of FastAPI picks up code changes, and Next.js hot-reloads front-end changes. This allows rapid development.

For debugging, you can use logging (FastAPI's logging or just print statements) to see the flow of data. Consider setting `LOG_LEVEL=DEBUG` in env to get verbose logs. We might implement custom logging where important actions produce log lines.

10. Environment-specific Config: For production or different environments, one can use different .env files or environment variables. For instance, in production you might not use `--reload` and might set `ENV=production` which could disable some debug features or use a different database path.

Deployment (beyond dev): If the user wants to run this server continuously, they might deploy it as a service (e.g., a systemd service on Linux, or a Windows Service). In a more advanced setup, containerization is an option: a Dockerfile could be created to run uvicorn with our app. That would make it easy to run on a NAS or cloud instance. For now, simply running the script in a screen or background process is fine for MVP.

Front-end Integration: If not using Next.js, even a simple HTML/JS page could interact with the API (thanks to CORS). Or a desktop app (maybe built with Tauri or Electron) could wrap around it. All of these would use the API, so as long as the API is consistent, multiple interfaces can be attached.

Example Configuration for a Developer: Suppose Alice is a developer setting this up. She would: - Git clone the repository. - Create a virtual environment, install reqs. - Obtain an OpenAI API key (if she wants to use GPT-4 for the agent) and a Telegram bot token (if she wants that interface). - Put those in `.env`. - Run the server. The console might show "*Running on http://127.0.0.1:8000*". - If she configures the Next.js UI, run that and open `http://localhost:3000` in browser to see the interface. - If she sends a message from the UI or Telegram, she should see responses coming (verifying the integration). - She can then tweak rules in code or config and immediately see changes (due to auto-reload).

We will provide some **documentation** in the repo (README) summarizing these steps and perhaps a Postman collection or httpie examples for testing endpoints.

Dependency Management: It's advisable to pin versions of critical libs in requirements to ensure a known working set. Also mention Python version needed; we choose 3.10+ to use modern typing (if 3.11, even better for performance). If someone is on 3.9, they may still run it but we'd adjust code if needed (like `list[str]` vs `List[str]` syntax differences).

Troubleshooting Tips: - If the agent calls are failing (e.g., OpenAI returning 401), check the API key and internet connectivity. - If Google Calendar isn't working, check that credentials are correct and perhaps use Google's OAuth playground to get a refresh token (document this in an integration-specific doc). - If Telegram bot isn't responding, ensure the bot token is correct and that the bot isn't privacy-restricted (by default, bots cannot see messages in group chats unless privacy mode off; for 1-1 chat it's fine). - Database issues: if locking errors or similar (shouldn't be common in SQLite for our load), ensure not accessing from multiple threads incorrectly (using a proper async engine or locks around writes might be needed; SQLAlchemy's default engine with check_same_thread=False can allow multi-thread writes, but we might use mostly single-thread event loop for FastAPI).

Performance Considerations (MVP): Not a huge concern for one user – FastAPI can handle hundreds of requests per second easily on a PC. The main performance bottleneck is calls to external APIs (LLM, Google, etc.), which are network-bound. We should use **async** IO for those where possible (httpx in async mode, etc.) so that the server isn't blocked. We might define our route handlers as `async def` and use `await` for network calls, which FastAPI supports. Also, if using the same event loop for Telegram polling, consider running that in an async background task instead to not block.

Monitoring: During development, using the interactive docs (Swagger UI at `http://localhost:8000/docs`) can be very helpful. FastAPI generates those automatically. We should annotate our endpoints with responses and schemas so that the docs are clear. This is also useful for integration

testing and even for an AI agent using self-describing API (though MCP typically uses JSON schemas through the protocol).

In summary, the developer setup is meant to be as simple as: - clone, install, config, run. We leverage FastAPI's self-docs and Python's ease of use to lower the barrier.

Example Workflows in Operation

Finally, let's walk through a few concrete example scenarios to illustrate how the MVP MCP server behaves. These will tie together the components, APIs, and data models described above, demonstrating the end-to-end flow.

Workflow 1: Processing a Context Update (Calendar Event)

Scenario: The user's Google Calendar has a new event added (or an existing event changed) for later today. We want this information to flow into the MCP server so the assistant can use it.

- **Step 1 – Google Calendar triggers an update:** Suppose at 9:00 AM, a new event "Team Sync at 4:00 PM" is added to the user's calendar (perhaps by a colleague's invite). If we have webhook integration, Google sends an HTTP request to our `POST /context/ingest` endpoint with the event details. If we use polling, our integration module calls the Google Calendar API at 9:05 AM, sees the new event, and then internally invokes the same ingest logic (calling it with the new data).

- **Step 2 – MCP Server ingests the event:** The `ingest` endpoint (or handler) receives the payload: e.g. `{"source": "google_calendar", "event": {"title": "Team Sync", "start": "2025-08-02T16:00:00+10:00", ...}}`. It authenticates the source (ensuring it's our integration, not an attacker) possibly via a token or the fact it's a loopback call. Then it processes:

- The event is parsed and stored in the database (e.g., in a `calendar_events` table or directly updating the `UserState.next_event` field). Let's say we update `UserState` with `next_event = "Team Sync at 4:00 PM"` (if this is now the soonest upcoming event).
- The Trace Memory Service is invoked to log the update: A new trace entry: `[09:05] (system) Context Update: Calendar event added - "Team Sync at 4:00 PM today."`
- The Nudging Service is notified about this update (could be via an event or simply it will pick it up on its next cycle). It checks rules: do we have any rule for when a new meeting is added? Possibly a rule like "if an event is added for today with keyword 'Sync', then at event creation time, do nothing immediate but maybe prepare a reminder." No immediate nudge now, but it sets the stage for later.

- If the event was very soon (say within 30 minutes), maybe a rule would fire a nudge right away, but in this case 4 PM is hours away, so likely not yet.

- **Step 3 – Confirmation (optional):** The server could respond to the webhook with 200 OK immediately. No direct user output is produced yet (the user might not even know our server got this update, unless they check logs or UI).

- **Step 4 – User checks context:** Now, at 10:00 AM, suppose the user asks "What does my day look like?" via the assistant. When the agent (or our front-end) requests the context frame (`GET /context/frame`), the server will include the newly ingested info. The frame's `current_state.next_event` would show "Team Sync at 4:00 PM". The agent's answer might be: "You have a team sync meeting at 4:00 PM. Otherwise, your day is open until then." The user is thus informed of the event through the AI's response, which was only possible because the MCP server captured that context update earlier.

- **Step 5 – Later Nudge:** As 3:45 PM approaches, the Nudging Service has a rule "Meeting reminder 15 minutes before events." At 3:45 PM, it triggers: it creates a nudge "Reminder: Team Sync at 4:00 PM is starting in 15 minutes." This nudge is delivered via Telegram (user's phone buzzes with the message) and also logged in trace. If our UI is open, it might show this as a notification too. The user is now well-informed without having to ask.

This workflow highlights how external data flows in and is used proactively and reactively to assist the user.

Workflow 2: Triggering a Rule-Based Nudge (Location-Based Reminder)

Scenario: The user has a rule: "When I arrive home, if it's after 6 PM, suggest starting my evening routine." This is a hypothetical rule for nudging a routine when home.

- **Step 1 – Home Assistant detects location change:** The user's phone (via Home Assistant app or router detection) marks them as "home" at 6:30 PM. Our HA integration polls and finds `person.alice` state changed to "home" at that time (or HA pushes an event).
- **Step 2 – MCP Ingests location context:** `POST /context/ingest` is called with `{"source": "home_assistant", "entity": "person.alice", "new_state": "home"}`. The server processes it:
- Update UserState: `location = "home"` (with timestamp).
- Log Trace: "[18:30] (system) Context Update: User is now home."
- Nudging Service checks rules. We have a rule defined (maybe in DB or hard-coded): condition: `state.location == "home" and current_time > 18:00`, action: `suggest_routine`. This condition is met (the user arrived home and it's 6:30 PM, which is after 18:00).
- **Step 3 – Nudge execution:** The Nudging Service triggers the action. Suppose `suggest_routine` is defined to send a Telegram message: "Good evening! Would you like to review your day's summary or plan for tomorrow?" It could also be designed to directly engage the agent (maybe auto-open a prompt for the agent to summarize something). For simplicity, it sends a message on Telegram.
- Our server calls Telegram API to send the message to the user.
- Also log this in Trace: "[18:30] (assistant) Nudge: Offered evening routine suggestion."
- **Step 4 – User receives nudge and responds:** The user sees the message on their phone and replies "Sure, show me the summary." That message goes to our Telegram polling, which then sends it into the chat flow (`POST /chat` with user_message "Sure, show me the summary.").
- **Step 5 – Agent responds with context:** Our server knows the user is home and it's evening, etc., and likely our agent (if prompt) will interpret "show me the summary" as a cue to do something. Because the context frame includes maybe a summary of the day's events or we have a specific prompt template for "daily summary" in our Prompts list, the agent can comply. Perhaps we have a prompt template for summarizing the day (one of MCP *Prompts* [9]). The agent (LLM) produces a nice summary of what happened today (e.g., "Today you completed X tasks and had

2 meetings. Everything on your agenda was finished. For tomorrow, you have a dentist appointment at 9 AM.").

- **Step 6 – Delivery of summary:** The server sends this back via Telegram to the user as the chat response, or if the user is now at their computer, maybe the UI shows it. The nudge successfully engaged the user in a helpful interaction they didn't explicitly request initially.

The key point in this workflow is showing how a rule can *proactively initiate* a helpful action, and how it seamlessly transitions into a normal conversation (with the user's response and agent's follow-up). It also showcases integration between two systems: Home Assistant providing a trigger, and Telegram delivering the message.

Workflow 3: LLM Requests Context Information (Agent-Initiated Context Pull)

Scenario: The user asks the agent a question that requires context, e.g., "Do I have any emails from John Doe today?". The agent doesn't have that info in its prompt, so it needs to fetch from the server (which might have a Gmail integration).

This scenario assumes the agent is somewhat autonomous and can decide to call MCP server's resource APIs or tools when it needs data. This is how MCP is intended – the agent should realize what it doesn't know and ask the server.

- **Step 1 – User query:** The user says or types, "Do I have any emails from John Doe today?" This comes in via `POST /chat` to our server.
- **Step 2 – Agent processing:** The agent receives the user query (with whatever base context we provided, e.g., profile saying maybe where emails are). The agent's thought process (not directly visible) determines: this is about emails, likely I should search the user's emails for John Doe. If the agent is chain-of-thought capable, it might think: *"I have a tool or resource for emails. Let me use it."*
- **Step 3 – Agent calls a tool or resource:** The agent issues an MCP request, for example `tools.call` for a tool `find_emails` with parameter `sender="John Doe", date="2025-08-02"`. In our setup, since we don't have direct JSON-RPC, how does it do this? If the agent is integrated via an MCP client, it could directly call our endpoint (if using some JSON interface). But simpler: We could intercept this intention because the conversation agent we implement could be a loop that checks for patterns like "searching emails..." in the agent's response. However, since this is a technical guide, we assume an ideal agent that directly invokes our server's API. Let's say it calls (through whatever mechanism) `GET /resources?filter=email&sender=John Doe&date=2025-08-02` or a dedicated endpoint `GET /emails/search?sender=John Doe&date=today`.
- **Step 4 – Server handles the resource request:** We have a Gmail integration that can fulfill this. Perhaps it has cached email metadata for today or it might live-query Gmail's API for emails from John Doe. It finds, for instance, 2 emails today from John. It returns a resource data structure to the agent: maybe a JSON list of email subjects or a snippet of each. For example:

```
{
  "resource": "email_search_results",
  "query": {"sender": "John Doe", "date": "2025-08-02"},
  "results": [
     {"time": "10:15", "subject": "Re: Project Update", "snippet": "Hi,
here is the update..."},
     {"time": "14:40", "subject": "Lunch Meeting", "snippet": "Shall we
have lunch tomorrow..."}
```

```
  ]
}
```

This goes back to the agent (the MCP client receives it and provides it to the LLM).

- **Step 5 – Agent answers the question:** Now armed with this data, the LLM can answer the user's question. It might respond: "Yes, you have 2 emails from John Doe today. One at 10:15 AM about a project update, and another at 2:40 PM about scheduling a lunch meeting for tomorrow."
- **Step 6 – Server relays answer:** The final answer is sent back to the user via the chat response. The trace logs will show the whole sequence: user question, agent tool call (and likely the tool result), then agent answer. Because we treat tool usage as events, we'd have something like:
- `[Time] (user) Do I have any emails from John Doe today?`
- `[Time] (agent) [Tool Request] find_emails{"sender": "John Doe", ...}`
- `[Time] (system) Tool result: found 2 emails from John Doe.`
- `[Time] (agent) You have 2 emails from John Doe today... (full answer)`

This trace not only serves debugging, but also could be fed back into agent memory if needed to justify its answer.

This workflow emphasizes the **context on demand** aspect of MCP. The agent doesn't need to have all information upfront; it knows it can query the server as needed. This leads to more efficient use of the LLM's context window and allows real-time querying of data. It's essentially how MCP enables *"two-way connections between data sources and AI"* [40] – the agent can ask the server for specifics, not just rely on a static prompt.

Workflow 4: Multi-Agent Collaboration (Future Roadmap Preview)

Scenario (future, not necessarily in MVP): One advanced use-case is having two AI agents collaborate or cross-check each other's outputs, especially for important tasks (this ties into the idea of adversarial or collaborative agents). While the MVP might not fully implement this, let's outline how the MCP server would support it to set the stage for future work:

- Suppose we have Agent A (e.g., ChatGPT) and Agent B (Claude). The user asks a complex question or a decision-making task. We want both agents to contribute: maybe Agent A proposes a plan and Agent B critiques it (adversarial collaboration), then the system synthesizes the result.
- The Agent Routing Service sees that for this type of query, both agents should be engaged. It sends the user query and context to Agent A and Agent B separately.
- Both respond (Agent A gives an answer/plan, Agent B gives an answer or perhaps is prompted to critique A's answer). The server collects both.
- The server might then either:
- Present both answers to the user (with some UI indication of differences), or
- Ask a third process (could be one of the agents or a simple heuristic) to reconcile. For example, we could prompt Agent A again with "Agent B had these points, please consider them and give a final answer" (or vice versa).
- Throughout, the MCP server is mediating: it holds the shared context that both agents use, and it logs all messages from each agent. If using an Agent-to-Agent (A2A) protocol in the future, the server could facilitate the direct communication by essentially passing messages (like a messaging hub).

While the above is speculative in MVP, it shows the architecture is ready: since we already can route to multiple agents, adding logic for them to interact is doable. In terms of data, we might tag trace logs by

agent and session. We could also incorporate something like an “AgentCard” metadata for each agent (A2A concept where agents advertise their capabilities [41]). For instance, our Agent definitions could include a profile that one agent could query about another (if we wanted them to reason about who to ask for help).

Federated Learning Example (Future): If multiple users each run an MCP server and we wanted them to collaboratively improve a model, the workflow would be: - Each server logs interactions and maybe fine-tunes a small local model (or collects gradients). - On a scheduled interval, model updates (not raw data) are sent to a central aggregator or peer-to-peer ring. - A combined improved model is then distributed back to each. - Throughout, differential privacy can add noise to gradients to mask individual data [37]. - The MCP server would have a module for this training process. Perhaps it would train a smaller model (maybe a language model that learns the user’s writing style or preferences) and not the giant LLM itself, as that’s impractical on-device in most cases. But federated fine-tuning of a smaller student model or adapter could be imagined.

Of course, implementing this is beyond MVP, but by mentioning it we outline a roadmap where the MCP server isn’t just a passive data store, but an active learner that **learns from the user privately** and contributes to collective improvements without sacrificing privacy.

In conclusion, the MVP we’ve specified provides a solid foundation: **Python + FastAPI backend with SQLite** to coordinate between user context and LLM agents, and optional **Next.js frontend** to make interactions user-friendly. We covered how it organizes memory, state, tools, and rules to enable an intelligent, context-aware assistant. We also ensured that the design is **secure, local-first, and extensible** for future innovations like multi-agent systems and privacy-preserving learning. By following this guide, developers should be able to implement the MCP server step by step and gradually evolve it into a powerful personal AI orchestration platform. The combination of clear API contracts, modular services, and a focus on user control will help maintain trust and reliability in these context-rich AI interactions, aligning with the emerging best practices in agentic AI systems [14] [42].

Sources: The concepts and architecture presented are informed by the MCP specification and community discussions, such as Anthropic’s introduction of MCP [1], LinkedIn’s overview of agentic AI with MCP/A2A [43] [44], and real-world integration examples (Home Assistant MCP integration [11], multi-platform personal assistant connectors [26]). These references reinforce the design choices made for a secure, modular MCP server that bridges AI with the user’s world.

---

[1] [40] Introducing the Model Context Protocol \ Anthropic
https://www.anthropic.com/news/model-context-protocol

[2] [14] [16] [17] [18] [19] [36] [41] [42] [43] [44] The Age of AI Disruption: With AI Agents and Agentic AI
https://www.linkedin.com/pulse/age-ai-disruption-agents-agentic-ashish-bajpai-lh3lc

[3] [4] [5] [6] [7] [8] [9] [20] [21] [28] [31] Server Concepts - Model Context Protocol
https://modelcontextprotocol.io/docs/learn/server-concepts

[10] [30] Architecture Overview - Model Context Protocol
https://modelcontextprotocol.io/docs/learn/architecture

[11] [22] [23] [33] [34] Model Context Protocol Server - Home Assistant
https://www.home-assistant.io/integrations/mcp_server/

12 Introduction - Model Context Protocol
https://modelcontextprotocol.io/docs/getting-started/intro

13 Build Agents using Model Context Protocol on Azure | Microsoft Learn
https://learn.microsoft.com/en-us/azure/developer/ai/intro-agents-mcp

15 Top 10 Claude MCP Servers for Marketing | Data-Mania, LLC
https://www.data-mania.com/blog/top-10-claude-mcp-servers-for-marketing/

24 25 26 35 Google Calendar MCP Server by Nick | PulseMCP
https://www.pulsemcp.com/servers/thisnick-google-calendar

27 SUPEROPTIX AI
https://superoptix.ai/docs/

29 GitHub - tmgthb/Autonomous-Agents: Autonomous Agents (LLMs) research papers. Updated Daily.
https://github.com/tmgthb/Autonomous-Agents

32 MCP integration for Google Calendar to manage events. - GitHub
https://github.com/nspady/google-calendar-mcp

37 [PDF] Evaluating the Collaboration and Competition of LLM agents
https://aclanthology.org/2025.acl-long.421.pdf

38 "P" in MCP stands for privacy - Crosshatch Blog
https://www.crosshatch.io/post/p-in-mcp-stands-for-privacy

39 Library - PLOT4AI
https://plot4.ai/library