ChatGPT

Technical Specification: Modular AI Chat Backend

Overview and Goals

The goal is to design a **modular backend system** for an AI chat interface that supports a **single-user, multi-agent architecture**. In this design, a **Router Agent** will intelligently delegate user messages to specialized **Agent** modules based on message content. Each agent can have its own tools (e.g. web search, calculators) and domain expertise. The system will be built in Python using **FastAPI** for a WebSocket-first API (enabling real-time streaming responses), **LangChain** for LLM integration and tooling, and **OpenRouter** as the LLM provider. Configuration of agents and their tools will be driven by **YAML files**, and the project will be managed with **Poetry** for dependency and packaging, with a focus on SOLID design principles and extensibility. The backend is decoupled from any frontend (the UI will be built later), exposing a clean API that a frontend can consume. Key requirements include:

- **Multiple Specialized Agents:** Modular agents each with a single responsibility or domain (e.g. math, science, etc.), which can be extended or added without modifying existing code (Open/Closed principle).
- **Router Agent:** A central agent that receives all user input and, using a strategy (LLM-based content classification), routes the query to the appropriate specialized agent.
- **Registry Pattern:** A dynamic plugin-like system where new agent classes (and new tool classes) are automatically discovered and registered, avoiding tight coupling.
- **Streaming Responses via WebSocket:** Responses should stream token-by-token to the client for interactivity. The API will primarily use WebSocket endpoints to push LLM output incrementally.
- **YAML-Based Agent Configuration:** Agents, their initial prompts, tools, and model settings will be defined in YAML files. The system will load this config at startup. The design will allow swapping out the config source (file, database, etc.) in the future without significant changes (Dependency Inversion principle).
- **CLI Support:** A simple CLI interface (or script) to interact with the agents (via the router) for testing and demos, without needing a frontend.
- **Docker-ready & Local Development:** The project will be developed as a Poetry package for easy local development, and a Dockerfile will be provided to containerize the app for deployment. The design will consider environment configuration (e.g. API keys) and reproducibility.

By adhering to **SOLID OOP principles**, the system will remain **well-organized, flexible, maintainable, and scalable** [1]. The following sections describe the architecture and components in detail, followed by an implementation plan.

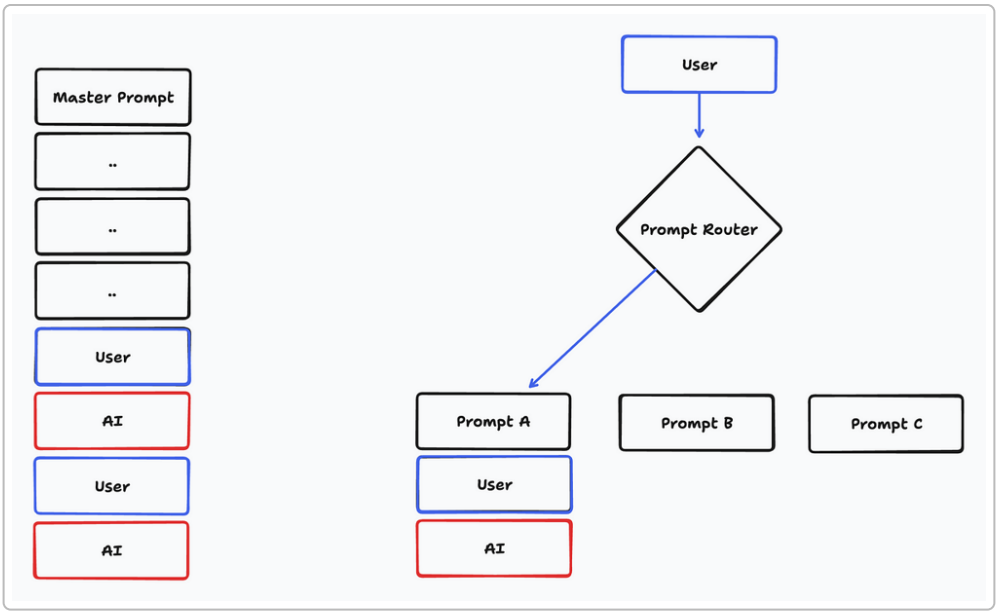

Figure: Comparison between a single monolithic prompt (left) vs. a prompt-router architecture (right). In a routed multi-agent system, a Prompt Router (or Router Agent) acts as an intermediary to direct the user's query to the appropriate specialized prompt/agent, rather than using one large prompt for all queries [2]. This modular approach yields more maintainable, debuggable prompts and responses.

Architecture Overview

System Architecture: The backend is organized into distinct modules: **Agents**, **Router**, **Tools**, **Config**, and **API Interface**. The single user interacts with the system through a WebSocket connection (or CLI), sending messages. The **Router Agent** receives each message and uses a routing strategy (by default, an LLM-driven system prompt classification) to determine which specialized agent should handle it. That agent then processes the query (possibly using its tools and an LLM via OpenRouter) to generate a response. The response is streamed back to the user over the WebSocket. All agents share a common interface and are registered in a central registry so the router can discover available agents dynamically.

Key steps in a typical request-response cycle: 1. **User Message Intake:** The user sends a message via the WebSocket connection (or via CLI/stdin). The system wraps this in a standardized **message object** (including metadata like conversation ID, timestamp, etc.). 2. **Routing Decision:** The Router Agent, which contains a prompt-based or algorithmic routing strategy, examines the message content. It consults the registry of agents (and possibly their declared domains from config) to decide which agent is best suited. This decision can be LLM-driven: e.g., the router uses a prompt asking *"Given the user query, which agent (by name) should handle it?"* and the LLM returns a choice [3] [4]. Alternatively, simpler strategies (keyword matching or ML classification) could be used or configured, but the default is an LLM with a system prompt for routing. 3. **Delegate to Agent:** Once an agent is chosen, the Router forwards the message to that agent. The agent has a **system prompt/persona** (from YAML config) defining its role and tools. Using LangChain (or a direct OpenRouter API call), the agent formulates a response. If the agent uses tools, it may invoke those (via LangChain's tool integration) before finalizing an answer. 4. **Streaming Response:** As the agent's LLM generates output, tokens are streamed back through the Router to the WebSocket. The FastAPI WebSocket endpoint sends these tokens to the client incrementally. This interactive streaming is enabled by

LangChain's callback handlers or generator functions, ensuring the user sees the answer in real-time. 5. **Completion and Memory:** Once the answer is complete, the router may log the interaction. Conversation state or memory (if maintained) is updated. Each agent could maintain its own short-term memory (e.g. using LangChain's conversation buffer for that agent), and the Router may record which agent was last active for context continuity. (This allows follow-up questions to stay within the context of the last agent when appropriate, unless the new message clearly belongs elsewhere.)

OpenRouter Integration: OpenRouter will serve as the backend for LLM calls. **OpenRouter** provides a unified API to many LLMs (open-source and closed) and handles provider failover automatically [5] [6]. This means our system can specify an OpenRouter model (such as GPT-4, Claude, etc.) and call it through a single API key, benefiting from fallback endpoints and a consistent interface. OpenRouter's API is **OpenAI-compatible**, so we can use LangChain's `ChatOpenAI` class (or a subclass thereof) by pointing it to OpenRouter's endpoint [7] [8]. For example, the agent's LLM could be initialized as: `ChatOpenAI(model_name="openrouter/gpt-4", openai_api_base="https://openrouter.ai/api/v1", openai_api_key=OPENROUTER_API_KEY, streaming=True)`. This will allow streaming responses and tool usage just as if using OpenAI's API. The model names and API keys will be provided via config or environment variables.

High-Level Component Diagram:
- **User (Client)** – Connects via WebSocket (or CLI).
- **FastAPI WebSocket Endpoint** – Receives user messages, passes to Router; streams back responses.
- **Router Agent** – Central decision-maker. Contains routing logic (system prompt or rules) to choose an **Agent**.
- **Agent (Base Class)** – Abstract class defining common interface (e.g. `process(message) -> response`). Concrete **Specialized Agents** implement this. Each agent uses: - A dedicated LLM prompt (system prompt defining its expertise/persona). - Possibly specific **Tools** (via LangChain tools or custom functions) for its domain. - Internal state or memory for context (if needed). - **Agents Registry** – A registry that holds references to all agent classes (and instances if instantiated at startup), identified by a name or key. The Router consults this to dispatch messages. - **Tools** – Discrete utility classes or functions (e.g. SearchTool, CalculatorTool) that agents can invoke. They follow a common interface (method to execute, a name, description for the LLM, etc.). They can be integrated via LangChain's tool mechanism, allowing agents to invoke them as actions. - **Configuration Manager** – Loads the YAML configuration that defines agent settings, tool availability, model parameters, etc. It uses file-based YAML initially, but is abstracted so that a different source (database, API) could be used later (Strategy pattern for config source). - **LLM Provider (OpenRouter)** – External service called by agents (through LangChain). Hidden behind LangChain's abstractions, it provides the actual AI completions. Could be swapped for another provider or local model by changing configuration or minimal code (Dependency Inversion: code depends on LangChain's `BaseLLM` interface rather than a specific provider).

In summary, the architecture separates concerns into modular components: **routing, agent logic, tool execution, configuration, and delivery (API)**. Each piece can be maintained or extended independently (e.g., adding a new agent doesn't require changing the router's code, just updating the config and ensuring the agent registers itself).

Core Components and Design

Agents Module

Agent Base Class: We define an abstract base class `Agent` (using Python's ABC or a base class in our module). This class enforces the interface all agents share. For example, an `Agent` could have methods like `initialize()` (to load any resources), `choose_action(user_message)` (if agent internally decides how to respond or which tool to use), and `generate_response(user_message, stream_callback)` which handles querying the LLM or invoking tools to form a reply. It may also include common utilities like accessing a shared memory or logging. The base class might also include a class-level registry mechanism (discussed below) so that all subclasses are tracked.

- **Single Responsibility:** The base Agent class's responsibility is defining the contract and possibly basic common behavior (e.g., a default way to call an LLM). Each concrete agent subclass will have a single focus/domain, complying with the **Single Responsibility Principle (SRP)** – e.g., a `MathAgent` only handles math questions, a `ScienceAgent` handles science queries, etc., each encapsulating logic only for that domain.
- **Agent State:** Each agent could hold a `memory` object (for example, a LangChain ConversationBufferMemory) so it can maintain context of the conversation *when it was active*. This means if the router sends multiple sequential queries to the same agent, that agent can remember prior user questions and its answers. (The router might decide to keep the same agent for follow-ups unless a topic shift is detected – this can be a future enhancement using the routing strategy).
- **LLM Integration:** Agents will use LangChain's LLM wrappers to interact with OpenRouter. Each agent can have its own model or share a global model. For flexibility, model parameters or even model choice can be specified per agent in the config (for instance, a simple agent might use a faster model like GPT-3.5, while a complex agent might use GPT-4). The Agent base or a utility can wrap around LangChain's `ChatOpenAI` (with OpenRouter endpoint) to create an LLM instance. The agent then provides a prompt (system + user message) to get a completion. By using LangChain, we can easily enable tools: LangChain's AgentExecutor can allow an agent to decide to use a tool if the LLM output indicates an action.

Specialized Agent Classes: We will create subclasses for each agent specified in the configuration. For example, `MathAgent`, `ScienceAgent`, `DefaultAgent` (for queries that don't fall into others, perhaps). Each subclass will: - Register itself with the registry (via a decorator or metaclass). - Define a unique name or key (to be referenced by the router or config). - Provide its own **system prompt or behavior**. For instance, `MathAgent` might have a system prompt: *"You are an AI math expert who can solve math problems and explain solutions step-by-step."* The agent might also have a calculator tool. - Override or implement the `generate_response` method. In many cases, this can be largely handled by a common routine (call LLM with the system prompt + user message, stream output). If the agent has tools, we might implement it as a LangChain Tool-using agent: e.g., a LangChain `initialize_agent` with that agent's tools and an LLM. The agent class can encapsulate this setup. - Ensure that the output is appropriate and maybe do post-processing if needed (e.g., ensure code blocks are formatted, etc., as per domain).

Extensibility: Adding a new agent in the future should be straightforward: create a new subclass of `Agent`, implement its specifics, and add an entry in the YAML config. The registry will auto-detect it, and the router will route to it when appropriate. This fulfills the **Open-Closed Principle (OCP)**: the system can be extended with new agent classes without modifying the existing routing logic or other agents' code [9].

Router Agent

The **Router Agent** is a special agent whose sole job is to decide which other agent should handle a user query. It implements the same base interface as other agents (it's essentially an Agent that produces a decision instead of a direct answer). However, its internal logic is unique:

- **Routing Strategy (System-Prompt-Based):** The default strategy will use an LLM prompt to classify the user's message. We will craft a system prompt that lists the available agents and their specialties, and asks the LLM to respond with the name of the best-suited agent for the query. For example, the router's system prompt (constructed from YAML config at startup) might be:

"You are a router agent. You have the following specialized assistants available: (1) MathAgent – expert in mathematics, (2) ScienceAgent – expert in scientific knowledge, (3) ChatAgent – a general conversational agent. When the user asks a question, analyze it and decide which agent should answer. Respond with the agent's name only."

The router then provides the user's query to this prompt and the LLM (using a quick, possibly cheaper model) returns a name or token corresponding to one of the agents. This approach leverages a "General LLM model" to do intent classification – *"the easiest and most common way... create a prompt that takes in the user message and responds with the prompt template to route to"* [10] . Using a powerful model like GPT-4 for routing is straightforward and effective [4] , though one could also fine-tune a smaller model or use deterministic rules for routing if needed in the future [11] .

Alternate Strategies: The design will allow swapping out the routing strategy (Strategy pattern). For instance, one could implement a `KeywordRouter` that checks keywords (e.g., "math" or presence of equations triggers MathAgent). Another could embed the user query and compare to example vectors per agent (Vector Distance strategy) [12] . These strategies can be encapsulated in classes or functions, and the router agent can choose one based on config. Initially, we implement the LLM-based approach because it's flexible and requires minimal hard-coding.

- **Registry Lookup:** Once the Router gets a decision (e.g. "MathAgent"), it looks up the corresponding Agent class or instance via the registry. The router then invokes that agent's `generate_response` method with the user's query. In effect, the router "hands off" the conversation to the specialist. The router might pass along conversation context if needed (for instance, if a conversation was ongoing with that same agent, the agent's memory will already contain prior turns).
- **System Prompt Routing Data:** The router agent needs to know about all agents and their descriptions. This will be populated from the YAML config (each agent entry can include a description of what it's best at). The router uses this to inform the LLM. This makes the system highly configurable – simply editing the YAML descriptions can change routing behavior without code changes.
- **Output Handling:** The Router itself doesn't craft the final answer; it simply returns whatever the chosen agent produces. But the router is the one connected to the WebSocket. So, the Router will act as a middleman in streaming: as the specialized agent yields tokens, the Router's code forwards those to the WebSocket. In terms of code, the Router agent might wrap the chosen agent call in a streaming loop or use LangChain's callbacks to relay messages.

- **Example:** Suppose the user asks *"Integrate x^2 from 0 to 5."* The Router's LLM prompt likely classifies this as math -> returns "MathAgent". The Router then calls `MathAgent.generate_response("Integrate x^2 from 0 to 5")`. The MathAgent might use a math tool or just solve via LLM and returns the solution text. The Router takes that and streams it out. If the next user question is *"What about the science behind rocket launches?"*, the Router's prompt now suggests "ScienceAgent", and context from MathAgent is not needed. This shows modularity in action.

Tools Module

Tool Interface: Tools are external utilities that agents can use to fulfill user requests (for example, searching the web, doing calculations, accessing databases, etc.). We define a base interface or abstract class for Tools, perhaps with attributes: `name`, `description`, and a method like `run(tool_input: str) -> str` (which executes the tool's function). This interface aligns with LangChain's tool API, so that tools can be easily added to a LangChain agent. Each tool's description will be used in the agent's prompt (LangChain does this automatically when an agent has tools: it tells the LLM which tools exist and what they do).

- **Example Tools:** We might implement a `CalculatorTool` (for basic arithmetic or using something like python's eval securely or sympy for integration), a `WebSearchTool` (that calls an API or uses LangChain's SerpAPI integration), etc. These tools are classes in a `tools` module. They register themselves in a Tools registry similarly, or are referenced explicitly in agent config.
- **Tool Configuration:** In the YAML config for an agent, we can list which tools it has access to (by name or class reference). At agent initialization, the system will attach those tools to the agent's tool list. For instance:

```
agents:
  - name: MathAgent
    class: myapp.agents.MathAgent
    description: "Expert in mathematics"
    model: gpt-4
    tools: [CalculatorTool, WolframAlphaTool]
```

  The system will load `CalculatorTool` and `WolframAlphaTool` classes (from the tools registry or by importing known tool modules), instantiate them (some tools might require API keys configured separately), and give them to MathAgent's LangChain agent executor. This design again supports OCP — new tools can be created and added without altering core logic; an agent's abilities can be tweaked just by config.
- **Custom Tool Execution:** If not using LangChain's built-in agent loops, an agent class could manually decide when to call a tool. For example, ScienceAgent might parse the query and if it detects a "search:" prefix, use the search tool. However, leveraging LangChain's agent capabilities is preferable since it can automatically decide using the LLM. In either approach, tools remain modular.
- **Extensibility:** New tools can be integrated by following the Tool interface. The agent logic or LangChain will remain unchanged, satisfying Open/Closed. If tool implementations depend on external services, those can be configured via environment (for API keys, etc.). Each tool class can have single responsibility (e.g., `CalculatorTool` only handles calculations). This also exemplifies

Interface Segregation Principle (ISP) – tools implement a narrow interface `run()` that agents use, and agents that don't need tools aren't forced to depend on tool specifics.

Registry and Plugin System

To allow dynamic discovery of agents (and possibly tools), we will use a **Registry Pattern**. The idea is to avoid hard-coding agent classes in a central list. Instead, each agent class will self-register or be automatically discovered:

- **Agent Registry:** We can implement this via a metaclass or a class decorator on the Agent base class. For instance, using a metaclass, whenever a new subclass of Agent is defined, it gets added to a global registry dictionary (keyed by class name or a given agent name). Alternatively, a simpler approach is a module-level registry: agent classes call a function or decorator at definition time like:

```
class MathAgent(Agent):
    name = "MathAgent"
    def __init__(...): ...

AgentRegistry.register(MathAgent)
```

  But doing it manually risks forgetting; a metaclass can automate it. We'll choose one and document it.
- **Discovery Phase:** At application startup, the system will **discover all Agent classes**. If they are all imported (by importing the `agents` package), the registry should be populated. The YAML config will list agent entries by name, which should match a registry key or class. The system will cross-check that each configured agent has a corresponding class available. This discovery means the core router doesn't need to know about each agent explicitly – it just queries the registry by name when routing [13]. This decoupling is critical: the router can work with any new agent that appears, as long as it adheres to the Agent interface.
- **Registration Mechanism:** Borrowing from plugin architecture concepts, **Discovery** is "the mechanism by which a running application can find out which plugins (agents) it has at its disposal" [13], and **Registration** is how a plugin (agent class) signals "I'm here, ready to do work" [13]. In our design, discovery could simply be importing all modules in an `agents/` directory, triggering class registration. Registration is handled by the base class/metaclass capturing subclass definitions.
- **Tool Registry:** Similarly, we can maintain a registry of Tool classes. Tools can register themselves by a name. The YAML `tools` list for an agent could either reference tool class names (which we look up and instantiate) or a pre-initialized tool object. A simple registry (even just a dictionary mapping name->ToolClass) would suffice for tools.
- **Registry Usage:** The **Router Agent** or a factory will use the registry when instantiating agents from config. For example, the YAML might specify `class: myapp.agents.MathAgent`. The system can import that path and instantiate it. But if class paths are not given, alternatively just `name: MathAgent` and we use `AgentRegistry.get("MathAgent")` to get the class. We then call `agent = MathAgent(config_for_that_agent)`.
- **Extensibility & Dynamism:** This pattern allows **dynamic loading**. In the future, one could even have a mechanism to load additional agent modules at runtime (e.g. via entry points or plugins installed via Poetry). The current scope is just to auto-discover in the codebase. It also aligns with **Dependency Inversion Principle (DIP)** – high-level code (the router or main app) doesn't hardcode

dependencies on concrete agent classes; it depends on an abstraction (registry or base class) to get them. New agents can be added without modifying the router.

YAML Configuration and Config Manager

YAML Config Format: We will use a YAML file (e.g. `agents.yaml`) to configure the agents and system parameters. This file may look like:

```
agents:
  - name: MathAgent
    class: myproject.agents.math_agent.MathAgent
    description: "Expert in mathematics and calculations."
    model: openrouter/gpt-4-32k
    tools:
      - CalculatorTool
      - WolframAlphaTool
    system_prompt: |
      You are MathAgent, a math expert AI. You can solve complex math
problems...
  - name: ScienceAgent
    class: myproject.agents.science_agent.ScienceAgent
    description: "Expert in general science and physics."
    model: openrouter/gpt-3.5-turbo
    tools: []
    system_prompt: |
      You are ScienceAgent, a scientific expert AI...
router:
  model: openrouter/gpt-3.5-turbo  # model used for routing decisions
  strategy: system_prompt         # could be "system_prompt" or "keywords", etc.
  system_prompt: |
    You are a router agent. Based on user input, choose the best agent:
    MathAgent (math questions), ScienceAgent (science questions), or
DefaultAgent (general)...
```

In the above: - The `agents` list defines each agent. We include the class path or name (for dynamic loading), a human-readable description (used by the router's prompt), the model to use, the tools it has, and its system prompt template. - The `router` section defines the router's own model (maybe we use a cheaper model for routing for efficiency), the strategy (here `system_prompt` meaning use the given system prompt template for LLM-based routing), and the prompt template itself. Alternatively, we could generate the router prompt from the agent descriptions automatically. - We could have other global settings (like OpenRouter API key could be picked up from env rather than YAML for security).

Config Manager: A `ConfigManager` or loader class will handle reading this YAML and validating it. We can use Pydantic models to define the schema of this config (ensuring required fields are present). For example, a `AgentConfig` Pydantic model with fields `name, class_path, description, model, tools, system_prompt`. Pydantic can even directly instantiate classes if we provide custom logic, but

likely we will parse into plain data then manually instantiate classes. - The Config loader will produce a list or dict of AgentConfig objects, and a RouterConfig. These will then be used to initialize the agents and router.

Decoupling Config Source: In alignment with DIP, the rest of the application should not hardcode file access. We can define an abstract interface like `AgentConfigStore` with a method `load_config() -> ConfigData`. The default implementation reads the YAML file. But later we could implement, say, `DatabaseConfigStore` that reads from a DB. The application startup code will choose which to use based on environment or parameters. By programming against an interface, we can swap storage without changing agent or router logic. For now, a simple YAML loader is fine, but we ensure the design allows substitution. - For example, `ConfigManager.load()` might internally use `YamlConfigStore` by default. If tomorrow we add a flag to use a cloud config, we can implement that and plug it in.

Validation and Error Handling: The ConfigManager should verify that each `agents[i].name` is unique, each `class` path is importable and is a subclass of Agent, and each tool name is recognized. It can also inject default values (like if an agent has no `system_prompt` provided, maybe use a default generic one). - Also, we will ensure sensitive data (like API keys) are not stored in YAML. Those should come from environment variables for security. The YAML can refer to them indirectly or leave model API keys blank. For instance, OpenRouter API key will be provided via an env var `OPENROUTER_API_KEY` which our code will read (the LangChain OpenAI wrapper expects `OPENAI_API_KEY` env var; we will set that or use parameters).

WebSocket API (FastAPI Integration)

WebSocket-First Design: The primary API for client interaction will be a WebSocket endpoint (e.g. `/ws/chat`). This allows the server to push streaming responses to the client easily. FastAPI supports WebSockets out of the box.

- **Endpoint Definition:** We will define something like:

```
@app.websocket("/ws/chat")
async def chat_websocket(websocket: WebSocket):
    await websocket.accept()
    # (optionally authenticate single user if needed)
    session_id = uuid4()
    while True:
        data = await websocket.receive_text()
        # Pass data to handler...
```

  On receiving a user message (`data`), the endpoint will call into our chat system (likely a function like `router_agent.handle_message(data, session_id)`).
- **Handling Streaming:** When the specialized agent generates a response, we want to stream it. Using LangChain, one approach is to use an **async generator** or callbacks:
- If using LangChain's `AsyncCallbackHandler`, we can implement `on_llm_new_token(token: str, **kwargs)` in a custom handler that sends the token to the WebSocket as it arrives [14] [15]. We would attach this handler to the LLM or chain's callback manager before invoking the chain [16].

- Alternatively, if we write a custom loop around `openai.ChatCompletion.create` with `stream=True`, we get a generator of tokens which we can iterate and send. But since we are using LangChain for tools, the callback approach is cleaner.
- We might incorporate a small utility: e.g., `WebSocketTokenCallback(websocket)` that implements LangChain's `BaseCallbackHandler` and streams tokens. The agent's LangChain LLM or agent executor will use an `AsyncCallbackManager` with this handler [17]. So, when we do `agent_chain.ainvoke(user_prompt)`, tokens will be sent out in real-time.

- The WebSocket endpoint could also gather the full message if needed (for logging or to send a final confirmation). But the important part is the immediate streaming.

- **Session and State:** Since it's single-user, we might not need an authentication or multi-session management in the MVP. However, to prepare for possible extension, we can treat each WebSocket connection or conversation as a session with an ID. We can maintain a dictionary of session states if needed (like which agent was last used, conversation memory objects, etc.). For now, the router can handle context continuity as described, and memory is mostly on a per-agent basis.

- **Error Handling:** If an agent or LLM call errors out (API error, etc.), the system should catch exceptions and inform the client (perhaps sending a special message or an end-of-stream indicator). We will incorporate try/except around the agent call, and if an error occurs, log it and send an error message over WebSocket before closing or waiting for the next user input.

HTTP Endpoints: While WebSocket is primary, we might also implement a basic REST endpoint for non-streaming calls (for testing or simple integrations). For example, a POST `/chat` that takes a user message and returns the full response after processing (no streaming). This could reuse the same logic but collect the output fully. This is optional, but easy to add given our core logic is abstracted away from the specific interface.

Cross-Origin and Security: As the frontend is separate, we should enable CORS as needed in FastAPI for the WebSocket and any HTTP routes so that a web frontend can connect. We also consider authentication (maybe a simple token or none since single-user scenario). For now, likely no auth or a dev token check could be included.

CLI Interface

In addition to the WebSocket API, a **command-line interface** will be provided for convenience and early testing: - We can create a Poetry script or entry point (e.g. `poetry run chat-cli`) that launches an interactive session in the terminal. This CLI will load the same YAML config and instantiate the router and agents. - It will then read user input from stdin in a loop and use the Router Agent to process it, then print out the agent's response to stdout. We can simulate streaming by printing tokens as they arrive (though in a terminal it may just print whole lines). - This CLI uses the same backend logic (we avoid code duplication by calling into a shared function, say `ChatSession.handle_text(user_input)` that returns the response or yields tokens). - Purpose: The CLI is extremely useful for quick testing of the multi-agent logic without needing to spin up a server or UI. It also serves as a demonstration in environments where running the FastAPI server might not be feasible. - Implementation details: possibly use the `cmd` Python library or just a simple loop. We might allow commands like `\exit` to quit, etc.

Adherence to SOLID Principles & Design Patterns

Throughout the design, SOLID principles are emphasized to ensure a robust architecture: - **Single Responsibility Principle (SRP):** Each class and module has one well-defined responsibility: - The Agent subclasses focus only on how to respond in a given domain. - The Router focuses only on selecting agents (not how to answer questions itself). - The Config manager only handles configuration loading. - The WebSocket endpoint only manages the connection and delegates logic to the core. - Tools do one thing each (calculator calculates, search searches, etc.). This makes the code easier to maintain and reason about. If a bug arises in routing, we look at RouterAgent; if a math error, we check MathAgent or its tools, etc. - **Open-Closed Principle (OCP):** The system is open for extension, closed for modification. Adding new agents or tools doesn't require altering the router or existing agents – just create new classes and update config. The router and registry will accommodate them. For example, if we need a new agent for SQL queries, we write `SQLAgent` and register it; the router (if LLM-based) might start routing to it if the prompt is updated with its description. - **Liskov Substitution Principle (LSP):** All Agent subclasses can be used interchangeably via the base `Agent` interface. The Router doesn't care if it's handing off to a MathAgent or a ScienceAgent – it just calls `generate_response` on it and expects a coherent result. This is ensured by designing Agent base class methods that each subclass honors. Similarly, all Tools adhere to the same interface so an agent can call any tool's `run()` without caring about internal details. - **Interface Segregation Principle (ISP):** We avoid bloated interfaces. The Agent interface is minimal (just what's needed to generate a response, maybe an optional hook for tool usage). We don't, for example, force all agents to implement a `use_tool` method unless they actually use tools – those can be handled internally or via LangChain. We could define smaller protocols if necessary (e.g., a `ToolUsingAgent` mixin if some agents use tools, but others don't need to know about it). This keeps classes lean. - **Dependency Inversion Principle (DIP):** High-level components (like the Router or main server logic) depend on abstractions (Agent interface, Config interface), not concrete classes. For instance, the router just knows it needs an agent that can handle a query, it doesn't hardcode "if science then ScienceAgent" etc. It asks an LLM or uses a map from config. The creation of agents is through a factory/registry, decoupling the router from agent class definitions. Also, by using LangChain (which abstracts LLM providers) and by designing a pluggable ConfigStore, we ensure that low-level modules (file I/O, OpenAI API specifics) don't leak into the high-level logic. This makes the system **flexible and maintainable** as requirements change [1].

Common Design Patterns: - **Factory Pattern:** We will likely implement factories for creating agents from config. For example, a function `create_agent(agent_config)` that uses the class name in config to instantiate the agent, set its properties (like system prompt, tools, model). This abstracts object creation, making it easy to change how agents are built (e.g., if we move from direct instantiation to using a dependency injection container, etc.). - **Strategy Pattern:** The routing strategy is pluggable. The RouterAgent can have a `RoutingStrategy` attribute that adheres to an interface (e.g. a `route(message) -> agent_name` method). We provide a `LLMRoutingStrategy` (system-prompt-based) by default. One could add a `KeywordRoutingStrategy` or others. We can select the strategy via config (as shown with `strategy: system_prompt`). - **Observer/Callback (Pub-Sub) Pattern:** The streaming of tokens to WebSocket uses callbacks (LangChain's callback system). This is akin to Observer pattern, where our WebSocket callback subscribes to token events from the LLM. It decouples the LLM generation from the WebSocket pushing mechanism. - **Registry/Plugin Pattern:** As detailed, our registry for agents/tools is a plugin system. It follows the idea of discovery and registration so the application is extensible without modifying the core for each addition [13]. - **Decorator Pattern:** If needed, we could use decorators for cross-cutting concerns (like logging or timing) on agent responses. For instance, a decorator on the `generate_response` method could log the time taken or handle exceptions. This isn't explicitly required,

but the design allows adding such decorators since classes are small and focused. - **Facade Pattern:** We provide a simple interface (`ChatService` or `ChatSession` class) that the API layer uses to interact with all this complexity. For example, the WebSocket endpoint can call something like `response = chat_service.handle_user_message(user_id, message)`. This `ChatService` (a facade) internally handles calling the router, etc. This shields the FastAPI code from needing to know details of agents and routing (improving separation of concerns between the web layer and the business logic layer).

All these patterns and principles contribute to a clean architecture where components are **modular and testable**. We can unit test an individual agent easily by injecting a fake LLM or tool, test the router logic by mocking the strategy, etc., without needing a running server.

Project Structure and Management

We will organize the code as a Poetry project (say project name `ai-chat-backend`). A possible directory structure:

```
ai_chat_backend/
├── pyproject.toml  (Poetry config, listing dependencies like fastapi, uvicorn,
langchain, openrouter SDK)
├── ai_chat_backend/  (Python package)
│   ├── __init__.py
│   ├── agents/
│   │   ├── __init__.py (import all agent classes to trigger registration)
│   │   ├── base.py (Agent base class and possibly metaclass/registry)
│   │   ├── math_agent.py (MathAgent implementation)
│   │   ├── science_agent.py (ScienceAgent implementation)
│   │   └── ... (other agent modules)
│   ├── tools/
│   │   ├── __init__.py (possibly auto-register tools)
│   │   ├── base.py (Tool base class)
│   │   ├── calculator.py (CalculatorTool implementation)
│   │   └── ... (other tools)
│   ├── router/
│   │   ├── __init__.py
│   │   ├── router_agent.py (RouterAgent class)
│   │   └── strategy.py (RoutingStrategy classes)
│   ├── config/
│   │   ├── __init__.py
│   │   ├── schema.py (Pydantic models for config)
│   │   ├── loader.py (ConfigManager and ConfigStore implementations)
│   ├── api/
│   │   ├── __init__.py
│   │   ├── server.py (FastAPI app, WebSocket endpoint definitions)
│   │   └── cli.py (CLI interface script)
│   └── main.py (entry point to launch the API server)
├── agents.yaml (the default configuration file)
```

```
├── Dockerfile
└── README.md (usage instructions)
```

This structure separates agents, tools, etc., improving maintainability. Poetry will manage dependencies – for example: - `fastapi` and `uvicorn[standard]` for the API. - `langchain` and `openai` (for OpenRouter compatibility) and possibly `langchain-openrouter` if needed. - `pyyaml` or just use pydantic which can parse YAML via an extra or use `ruamel.yaml`. - Testing libraries if needed (pytest).

We also ensure to pin versions in `poetry.lock` for reproducibility. The project can be installed as a package, which means others could import `ai_chat_backend` in other contexts if needed.

Docker & Deployment Considerations

Even though primary dev is local (Poetry), we will create a **Dockerfile** to containerize the app: - Use a lightweight Python base image (e.g. `python:3.11-slim`). - Copy pyproject.toml and poetry.lock, install dependencies (possibly using `poetry install --no-dev` in the image, or converting to `requirements.txt` to use pip for smaller image). - Copy in the application code and the `agents.yaml`. - Set an entrypoint to run `uvicorn ai_chat_backend.api.server:app --host 0.0.0.0 --port 8000`. - Ensure environment variables like `OPENROUTER_API_KEY` can be passed in (Dockerfile won't contain them, but documentation will say to supply `-e OPENROUTER_API_KEY=...`). - The Docker image should include all needed dependencies (including possibly system libs if any needed by certain tools).

We will also likely include a `docker-compose.yml` for convenience (as seen in similar projects [18]), but not strictly necessary. The image will be tested locally to ensure parity with dev environment.

Development Workflow: Using Poetry, developers can run `poetry shell` then `uvicorn ai_chat_backend.api.server:app --reload` to iterate. The CLI can be run with `poetry run ai_chat_backend-cli` (if we set that in pyproject `[tool.poetry.scripts]`). Documentation will highlight these.

Logging and Debugging: We should integrate Python's logging library, have the router and agents log key events (e.g. "Router selected MathAgent for query X" or "MathAgent used CalculatorTool"). In development, those logs can help debug routing decisions. In production, logs could be directed to stdout (Docker best practice) and managed by the container environment. The logging can be made configurable (debug vs info level).

Implementation Plan

With the specification established, the following step-by-step plan outlines how to implement the system:

1. Project Setup with Poetry

- Initialize a new Poetry project (e.g. `poetry new ai-chat-backend` or manually create `pyproject.toml`).
- Add dependencies:
    - **FastAPI** (for API) and **uvicorn** (for running the server).

    - **LangChain** and **OpenAI API** (`openai` package) for LLM integration. Include `langchain-openrouter` if it simplifies using OpenRouter (or use the method of overriding API base).
    - **PyYAML** (for reading YAML) and **pydantic** (for config validation models).
    - Optionally **websockets** (FastAPI includes starlette which has websockets support).
    - **Poetry** will manage a lockfile for deterministic builds.
- Set Python version (>=3.10 for dataclasses and type hints, matching LangChain requirements).
- Ensure the package `ai_chat_backend` is created with the structure as above.
- Set up version control (git) if not already, to track changes.

2. Define the Agent Base Class and Registry

- In `agents/base.py`: Define `class Agent(metaclass=AgentMeta)` where `AgentMeta` is a custom metaclass that registers subclasses. The metaclass's `__init__` can add the class to a registry dict (e.g. a global `AGENT_REGISTRY` within that module). Alternatively, use a simpler approach: define `AGENT_REGISTRY = {}` and a decorator `register_agent` that appends classes. For clarity and reliability, the metaclass approach ensures no subclass is missed.
- Agent interface: decide on key methods:
    - `generate_response(self, message: str, context: ConversationContext = None) -> AsyncIterator[str] or str`: Should it be sync or async? Given LangChain's LLMs can be used asynchronously, an async method is beneficial for streaming. We might have it yield strings (tokens or chunks) for streaming use, or have a separate mechanism for streaming callbacks.
    - Perhaps a simpler design: `generate_response()` returns a complete string (for sync usage), and we separately handle streaming via callbacks. We can implement both: one method for full response (which internally calls LangChain synchronously), and an async method to stream (calls LangChain with streaming).
    - We also include a property for `name` (matching config name) and maybe `description`.
    - Constructor could accept a config object (like AgentConfig) to set up its system prompt, model, and tools.
- AgentMeta (or registration decorator) should require each agent class to have a unique name. We can use the class name or a class attribute. For explicitness, we might use a class attribute `agent_name`. The metaclass can do: `registry[cls.agent_name] = cls`.

- Write basic docstrings for clarity.

- **Example** (simplified):

```
AGENT_REGISTRY = {}
class AgentMeta(type):
    def __init__(cls, name, bases, attrs):
        super().__init__(name, bases, attrs)
        if hasattr(cls, "agent_name") and cls.agent_name:
            AGENT_REGISTRY[cls.agent_name] = cls
class Agent(metaclass=AgentMeta):
    agent_name: str = None
    def __init__(self, description:str="", model=None,
tools:List[Tool]=None):
```

```
    ...
def generate_response(self, message:str) -> str:
    raise NotImplementedError
```

This way, any subclass that sets `agent_name` will be auto-registered.

3. Implement a Few Specialized Agents

- Create files for at least a default agent and one or two domain agents to illustrate:
  - `agents/default_agent.py`: e.g. `class DefaultAgent(Agent)`. This could be a fallback for general questions. It might have a general system prompt like "You are a helpful general assistant." It may not use any special tools. Register with name "DefaultAgent".
  - `agents/math_agent.py`: e.g. `class MathAgent(Agent)`. Set `agent_name = "MathAgent"`. In `generate_response`, possibly implement logic to use a calculator tool if needed. But initially, it can just rely on the LLM (with a math-focused prompt) to answer.
  - For now, these methods can actually use a common utility to query the LLM. Rather than duplicating code in each agent, consider factoring out:

    ```
    self.llm_chain = LangChain LLM or chain set up during __init__
    def generate_response(self, message):
        # format prompt with system (from self.description or a stored
    prompt) + message
        # call self.llm_chain (maybe self.llm_chain.run(message)
    or .predict)
        return result
    ```

    Use LangChain's `PromptTemplate` and `LLMChain` for each agent: at init, do:

    ```
    prompt = ChatPromptTemplate.from_messages([
         SystemMessage(prompt=self.system_prompt),
         HumanMessage(prompt="{input}")
    ])
    self.chain = LLMChain(llm=self.llm, prompt=prompt)
    ```

    Then `generate_response(msg)` simply does `return self.chain.run(input=msg)`.
  - **Tool usage:** If agent has tools, we might create a LangChain AgentExecutor. For example, if using OpenAI functions or the standard toolkit agent:
  - Setup tools list (LangChain tools from our Tool classes, see below).
  - Use `initialize_agent(tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)` to get an executor that will use the tools based on the prompt.
  - Then `generate_response` could call this executor's `run(message)`. LangChain will decide if a tool is needed by looking at the LLM's output.

    - This requires the agent's system prompt to be embedded in the prompt (LangChain's Zero-shot agent automatically uses tool descriptions; we might incorporate our domain instruction in the prefix prompt).
    - This part can be complex; as a starting point, one might implement simple agents without tools, and add tool integration once basic flow works.

- **Memory:** Decide if each agent should have its own memory. Possibly use LangChain's `ConversationBufferMemory` in the chain, with an identity separate per agent. This memory would reset when agent changes, but persistent per agent across turns. Implement this later if needed (to avoid complexity initially).

- Write tests or at least small experiments to ensure each agent can produce a response via OpenRouter (using a real API key in dev).

4. Implement Tools and Tool Interface

- In `tools/base.py`: Define `class Tool` (could be abstract) with `name: str`, `description: str`, and `__call__` or `run()` method.
- Implement one example tool:
    - `tools/calculator.py`: `class CalculatorTool(Tool)`. This might evaluate basic arithmetic expressions from a string using `numexpr` or Python's `eval` safely. Its `description` might be "Calculator: solves arithmetic expressions". This can be used by MathAgent.
    - If we have time or need, implement a dummy `SearchTool` that maybe returns a fixed dummy result or requires an API. (Better to keep it simple unless a real API like SerpAPI is configured).
- Similar to agents, maintain a `TOOL_REGISTRY` mapping names to classes or instances. Tools can be stateless, so a single instance could be reused. Or instantiate per use. Simpler: we can just store classes and instantiate when assigning to an agent.
- If using LangChain's `Tool` class (langchain.tools), we can adapt our tools to that interface. For instance, LangChain expects a function and description. We might not need to reinvent this – possibly just create LangChain `Tool` objects from our functions. But writing our interface keeps it consistent with our architecture.
- Example integration: In an agent's `__init__`, do:

```
tool_objs = []
for tool_name in config.tools:
    tool_cls = TOOL_REGISTRY[tool_name]
    tool_objs.append(tool_cls())
self.tools = tool_objs
```

If using LangChain's agent executor, we'd convert these to LangChain `Tool` objects by something like:

```
lc_tools = []
for t in self.tools:
```

```
    lc_tools.append(
       Tool(name=t.name, func=t.run, description=t.description)
    )
self.agent_executor = initialize_agent(lc_tools, self.llm, ...)
```

- Ensure to handle any required config for tools (e.g., if a tool needs an API key, use env variables or add optional config sections for tools in YAML).
- **Test** a tool in isolation if possible, e.g., `CalculatorTool().run("2+2")` returns "4".

5. Router Agent Implementation

- Create `router/router_agent.py`:
  - `class RouterAgent(Agent)` or possibly not a subclass of Agent if that complicates the design. It could be a separate class, but making it an Agent means it fits in the registry if we wanted (though we might not need to route to a router).
  - It will have a reference to the `RoutingStrategy` interface. Define in `router/strategy.py` something like:

    ```
    class RoutingStrategy:
        def choose_agent(self, user_message: str, agents: Dict[str,
    AgentMetaData]) -> str:
            ...
    ```

    where `AgentMetaData` could be a simple class holding agent name and description (to feed the prompt).
  - Implement `LLMRoutingStrategy(RoutingStrategy)` which on init loads the router LLM (as per config, e.g., ChatOpenAI model for routing) and perhaps prepares a prompt template. Possibly use few-shot examples or just a single prompt with agent list as shown above.
  - `choose_agent(msg)` will format the prompt (inserting the user message and the list of agent descriptions) and call the LLM to get a completion. Parse the output (likely the name of agent).
  - We have to ensure the output exactly matches one of the known agent names. Techniques: we can instruct the LLM to answer with the exact agent name. If there's ambiguity or an unknown name, have a fallback (e.g., if LLM output not recognized, default to DefaultAgent).
  - Alternatively, use a simpler method: e.g., have the LLM output a JSON with a field “agent”: name. But parsing might be overkill for just a single word.
  - The RouterAgent’s `generate_response` would basically:
  - Use strategy to get chosen_agent_name = strategy.choose_agent(message, agents_info).
  - Find the agent instance for that name. We may instantiate all agents at startup and keep them in a dict, or instantiate on the fly (but likely keep them ready, with their models loaded, for performance).
  - Delegate: If streaming, we might call `agent.generate_response_stream(message)` which yields tokens and we yield from RouterAgent. Or if not streaming, just call `agent.generate_response(message)` and return the result.
  - Possibly prepend an identifier or meta (but probably not – the user doesn’t need to know which agent answered, though could be useful for debugging).

    - Create the router's LLM using OpenRouter GPT-3.5 or similar (to keep it fast). Ensure to set `streaming=False` for the routing LLM since we want a quick one-shot answer.
- If not making RouterAgent a subclass of Agent, it could just be a separate component. But treating it as an Agent (with name "RouterAgent") could allow recursion (not needed) or uniform handling. In practice, the router isn't invoked by another agent, so it could be standalone in main flow.

6. Configuration Loader Implementation

- Use Pydantic to define models in `config/schema.py`:

```
class AgentConfig(BaseModel):
    name: str
    class_path: str
    description: str
    model: str
    tools: List[str] = []
    system_prompt: str
class RouterConfig(BaseModel):
    model: str
    strategy: str
    system_prompt: str = None
class AppConfig(BaseModel):
    agents: List[AgentConfig]
    router: RouterConfig
```

- In `config/loader.py`, implement a `ConfigLoader` that reads YAML:
    - Use `yaml.safe_load(open(file))` to get dict, then do `AppConfig.parse_obj(dict)` for validation.
    - Alternatively, Pydantic can read JSON/YAML via `.parse_file`.
    - Return the AppConfig instance.
- Possibly implement interface as discussed:

```
class ConfigStore(ABC):
    def load(self) -> AppConfig: ...
class YamlConfigStore(ConfigStore):
    def __init__(self, path): self.path = path
    def load(self) -> AppConfig:
        # read file and parse
```

  This is maybe overkill for now, but sets stage for future.
- The loader may also instantiate agent classes. But likely better to separate parsing from instantiation. Do parsing into models first.
- In the main startup (or in a factory function), use the config to initialize:
    - For each AgentConfig, import or get the class:

- If `class_path` is given, do `module_path, class_name = class_path.rsplit(".", 1)` then `cls = importlib.import_module(module_path).__getattribute__(class_name)`.
- Alternatively, if we rely on registry and the `name` corresponds to `agent_name`, we could do `cls = AGENT_REGISTRY[name]`. This avoids needing class_path in config at all, but then the system must import all agent modules beforehand. We can ensure `agents/__init__.py` imports all known agent modules. For flexibility, having class_path in config is explicit, but it's a bit redundant if we auto-register anyway. We might allow either approach for convenience.
- Instantiate each agent: e.g. `agent = cls(description=config.description, model=config.model, tools=...)`. The model field could be a string like "openrouter/gpt-4", we need to create a LangChain LLM from it:
- For OpenRouter, as discussed, we use ChatOpenAI with `openai_api_base` override. If model string contains "openrouter/", we know to target that. Or we might put a mapping in config like:

  ```
  model: gpt-4
  model_provider: openrouter
  ```

  But since we assume everything is via OpenRouter, we might just use the model name directly.
- Implementation: Possibly create a utility in config or agents to create LLM: given model name and maybe temperature, etc., return a ChatOpenAI instance. If using OpenRouter, set env var `OPENAI_API_BASE` once, or pass as param for each instance.
- Also, handle API key: ensure `OPENAI_API_KEY` (OpenRouter key) is in environment. Document that it must be set; optionally allow reading from config (but not recommended to store secrets there).
- Instantiate tools for agent as earlier step.
- Keep track of agent instances in a dictionary (e.g. `running_agents[name] = agent_instance`).
- Instantiate the RouterAgent similarly: create its LLM (from RouterConfig.model), and if `strategy` is system_prompt, instantiate `LLMRoutingStrategy` with the router's system prompt (if provided; else, construct one from agents' descriptions dynamically).
- Pass the strategy into RouterAgent (or if RouterAgent inherits Agent, maybe treat strategy as an internal detail).
- The RouterAgent might not need to be in the registry or config's agent list. It's configured separately under `router:` section, not as part of AGENT_REGISTRY because it's not routed to by itself. So we won't add RouterAgent to AGENT_REGISTRY (unless we wanted to allow e.g. an agent to ask the router something, but that's not a use case).

- Summarizing: Config loader yields config data -> then Initialization phase builds all agent objects + router object. We might implement this in `main.py` or in a separate manager class (like `ChatBackend` class) that holds all components.

7. FastAPI Application Setup

- In `api/server.py`, create FastAPI app. Include CORS if needed:

```
from fastapi import FastAPI, WebSocket
app = FastAPI()
```

- On startup event (`@app.on_event("startup")`), load the config (perhaps using a path from env or default `agents.yaml`). Initialize the agents and router as above. Possibly store them in `app.state` or global variables. For better structure, encapsulate in a `ChatService` object that has `router` and `agents` inside. Then `app.state.chat_service = ChatService(config)`. The ChatService can expose a method `handle_message(session_id, message)` that uses the router to get response (handles both streaming and non-streaming modes).
- WebSocket route: `python`

```
@app.websocket("/ws/chat")
async def chat(websocket: WebSocket):
    await websocket.accept()
    session_id = str(uuid4())  # if we want to track sessions
    try:
        while True:
            raw_text = await websocket.receive_text()
            # Optionally, handle some control messages or ping.
            # Process the message:
            # Use chat_service to get an async generator of response
            async for token in
app.state.chat_service.stream_response(session_id, raw_text):
                await websocket.send_text(token)
    except WebSocketDisconnect:
        # handle disconnect if needed
```

- The `stream_response` method of chat_service (or router agent) would:
  - Call RouterAgent to decide agent.
  - Then either call an async generator from the chosen agent or use callback method.
  - If we implement agent's `generate_response` synchronously returning full text, we need a different approach for streaming. Instead, better to implement an async generator in agents. For instance, for streaming with LangChain:

```
async def stream_response(session, message):
    chosen_agent = router.choose_agent(message)
    agent = agents[chosen_agent]

# Setup a LangChain CallbackManager with a queue or use WebSocket
directly
    callback = WebSocketCallback(websocket)  # not available here
directly, so better to yield instead of callback
    # If agent supports streaming directly:
    async for tok in agent.stream_generate(message):
        yield tok
```

Another approach: use an asyncio Queue. The callback puts tokens into a queue, and this function yields from the queue. But that's more complex. Simpler: rely on LangChain's `ainvoke` as in Medium example [19] : We saw in Shubham's example:

```
callback_manager =
AsyncCallbackManager([LLMCallbackHandler(websocket)])
llm = ChatOllama(... streaming=True,
callback_manager=callback_manager)
chain = ... # rag_chain
query = await websocket.receive_text()
await chain.ainvoke(query)
```

The chain itself sends tokens via the callback handler. In our case, our `stream_response` could set up such a callback to send tokens immediately and then simply await the chain call (and maybe yield nothing or just indicate completion). However, that tightly couples the WebSocket inside the callback; an alternative is to push tokens to the websocket from within the callback directly (which is fine, as shown).

- For clarity, perhaps implement `WebSocketCallbackHandler(AsyncCallbackHandler)` inside the websocket route where we have access to `websocket` object. This handler's `on_llm_new_token` does `await websocket.send_text(token)` (ensuring it's async handler).
- Then to generate:

```
agent.chain.callback_manager = AsyncCallbackManager([ws_callback])
await agent.chain.acall(user_input)  # acall or ainvoke for async
```

Once this returns (meaning LLM finished), we optionally send an end-of-stream marker (or the client just knows the message ended when no new tokens for a bit).

- We have to be careful with `await websocket.send_text` inside the callback – ensure it's done properly in AsyncCallback (LangChain supports async callbacks).
- If the above proves tricky, a fallback is to do blocking generation in a thread and yield results; but since FastAPI is async, better to do async all the way.

- **HTTP endpoint (optional):** Implement a simple POST `/chat` for testing:

```
@app.post("/chat")
async def chat_http(request: ChatRequest):
    response = ""
    async for token in app.state.chat_service.stream_response("default",
request.message):
        response += token
    return {"response": response}
```

Or use a normal (non-async) generate for simplicity: `return {"response": app.state.chat_service.get_response("default", request.message)}` which calls

router and returns full answer. This can be used to quickly test correctness of answers without dealing with streaming.

8. Finalize OpenRouter LLM Integration

- Configure environment for OpenRouter API key: In development, set `OPENAI_API_KEY` to the OpenRouter API Key (OpenRouter uses the same header name by design). Also set `OPENAI_API_BASE=https://openrouter.ai/api/v1` so that OpenAI SDK (used by LangChain's ChatOpenAI) hits OpenRouter endpoint. We might do this in code as well:

```
os.environ["OPENAI_API_BASE"] = "https://openrouter.ai/api/v1"
os.environ["OPENAI_API_KEY"] = config.api_key  # if we had it, but we
likely just rely on user to set env.
```

- Alternatively, use the `langchain_openrouter` integration (if it's stable) – it basically wraps ChatOpenAI as we saw [8]. But manually setting base URL works too.
- Ensure our LangChain LLM instantiation (ChatOpenAI) includes `streaming=True` for those agents where we want streaming (which is likely all, by default).
- Verify that streaming works with OpenRouter models (OpenRouter documentation suggests it supports streaming similar to OpenAI).
- Also plan fallback: if OpenRouter or network is down, perhaps catch exceptions and send a friendly error to user.
- The nice feature: since OpenRouter can route to many models, we could allow model selection in YAML per agent, giving a lot of flexibility to optimize cost/performance per agent.

9. Implement CLI Tool

- In `api/cli.py` (or create a separate module `cli_tool.py`), implement an interactive loop:

```
def main():
    config = ConfigLoader(...).load()
    service = ChatService(config)
    print("CLI Chat started. Type 'exit' to quit.")
    while True:
        query = input(">> ")
        if query.strip().lower() in ("exit", "quit"):
            break
        # Synchronous handling:
        response = ""
        for token in service.stream_response("cli-session", query):  #
possibly an iterator
            print(token, end="", flush=True)
            response += token
        print()  # new line after response
    ```
```
```

- The above might use synchronous iteration if we adapt stream_response to return a generator of tokens synchronously (this could be done by internally using the async generator but running it via `asyncio.run()` for simplicity if needed, or using the sync LangChain call which returns full response).
- Alternatively, just do `resp = service.get_response("cli-session", query)` and print that. This loses the streaming effect in CLI (which might be fine).
- The CLI will utilize the same underlying router and agents loaded in `service`. So ChatService can maintain state if needed (like a dictionary of last agent per session, if implementing that).
- Ensure this is wired in pyproject as a script entry point, e.g.:

```
[tool.poetry.scripts]
chat-cli = "ai_chat_backend.api.cli:main"
```

- Document usage in README.

10. Testing and Validation

- Before Dockerizing, test the system locally:
  - Start `uvicorn ai_chat_backend.api.server:app` and use `wscat` or a simple HTML page to open the WebSocket and send messages. Observe if streaming tokens arrive correctly and if the routing chooses the correct agent.
  - Test a math question vs a general question to see if router picks different agents.
  - If possible, simulate some edge cases: unknown query (should maybe go to DefaultAgent), or tool usage (ask a math that requires calculation to see if CalculatorTool is invoked).
  - Test CLI in various scenarios as well.
- Write unit tests if time permits for core functions:
  - Test that `RoutingStrategy.choose_agent` returns expected agent given certain inputs (maybe by mocking the LLM to return predetermined outputs).
  - Test that `AgentRegistry` properly registered all agent classes.
  - Test the `Tool.run` methods (like Calculator).
  - Possibly test config parsing (given a sample YAML string, ensure the AppConfig is correct).
- Integrate these tests into a CI if applicable (Poetry can run pytest etc.).

11. Dockerization

- Create Dockerfile with multi-stage build:
  1. Use an official Poetry image or install poetry in builder stage:

```
FROM python:3.11-slim AS builder
RUN pip install poetry
WORKDIR /app
COPY pyproject.toml poetry.lock ./
RUN poetry config virtualenvs.create false && poetry install --no-dev
COPY . .
```

  2. Use a runtime image:

```
FROM python:3.11-slim AS final
WORKDIR /app
# Copy installed deps from builder (if using same base, or just copy
entire /app if slim)
COPY --from=builder /usr/local/lib/python3.11/site-packages /usr/
local/lib/python3.11/site-packages
COPY --from=builder /app .
EXPOSE 8000
CMD ["uvicorn", "ai_chat_backend.api.server:app", "--host",
"0.0.0.0", "--port", "8000"]
```

3. Alternatively, skip multi-stage and just pip install in one layer for simplicity (but larger image).

- Add the `agents.yaml` in COPY so that it's included in image (or better, bake default config into the package as data, but copying is fine).
- Build the image and test it (e.g. `docker build -t ai-chat-backend .` then `docker run -p 8000:8000 -e OPENAI_API_KEY=sk-... -e OPENAI_API_BASE=https://openrouter.ai/api/v1 ai-chat-backend`).
- The container should come up and serve the WebSocket. Test via a client inside the host to confirm.

12. Documentation and Diagrams

- Write a comprehensive README.md explaining how to configure (YAML and env vars), run locally, run in Docker, and how to extend with new agents/tools. Emphasize the design choices in simpler terms for a developer who might work on it.
- Include the architecture diagram (the one we embedded in this answer or an updated one) in the README if possible.
- Possibly include sequence diagrams or flowcharts for the routing logic for clarity.
- Document each agent and tool in code with docstrings so future developers understand their purpose.

13. Future Enhancements (Not immediate, but design supports them)

- **Multi-user support:** If needed, the architecture can be extended to handle multiple users by associating each WebSocket connection with a separate conversation state. The ChatService could maintain a mapping from session/user to a RouterAgent (or at least to conversation memory). Each user could have their own instances of agents if isolation is needed (or share agents but with separate memory contexts).
- **Persistent Config/DB**: Because we isolated config loading, switching to a database or API-based config service is straightforward – implement a new ConfigStore. Could allow dynamic updates to agents without restarting (beyond scope for now).
- **Agent Communication**: Right now, only the router and the chosen agent interact. We could envision agents calling other agents (if one needs help). The design could accommodate a manager agent that orchestrates multiple sub-agents for a single query (like decomposition of tasks). Our supervisor (router) could be extended to chain agents (hierarchical) if needed, but that complicates the flow. For now, one-hop delegation is sufficient.
- **OpenRouter Features**: OpenRouter supports many models and even structured outputs [20] [21]. We could allow YAML to specify parameters like temperature, max tokens, or use OpenRouter's

features like providing a user identifier for rate-limit tracking. Also, consider using OpenRouter's **fallback** ability implicitly provided – our calls automatically get resilience to provider downtime [5].

By following this implementation plan step by step, we will build a robust, extensible backend that meets all the requirements. The end result is a maintainable codebase with clear separation of concerns, where adding a new agent or tool is straightforward, and the system can evolve (thanks to SOLID principles) without breaking existing functionality. The combination of YAML configurability, a plugin registry, and a content-based router ensures the system can scale in complexity (more agents, more tools) while keeping the code clean and organized.

Overall, this modular backend will serve as a flexible foundation for the forthcoming frontend, and any AI chat applications that require orchestration of multiple specialized LLM-powered agents.

Sources:

- Real Python – SOLID Principles in Python [1] (justification of maintainable, scalable design).
- Plugin Architecture Concepts – Discovery and Registration [13] (inspiration for agent registry).
- LangChain Router Chain Example [3] (multi-agent selection scenario).
- PromptLayer Blog – Prompt Router via GPT-4 [10] (LLM-based routing strategy).
- OpenRouter Medium Article – OpenRouter unified LLM API [5] [7] (explanation of OpenRouter benefits and integration).

---

[1] [9] SOLID Principles: Improve Object-Oriented Design in Python – Real Python
https://realpython.com/solid-principles-python/

[2] [4] [10] [11] [12] Prompt Routers and Modular Prompt Architecture
https://blog.promptlayer.com/prompt-routers-and-modular-prompt-architecture-8691d7a57aee/

[3] LangChain's Router Chains and Callbacks | by Reflections on AI | Medium
https://medium.com/@gil.fernandes/langchains-router-chains-and-callbacks-722524c4aa42

[5] [6] [7] [8] [20] [21] OpenRouter + LangChain: Leverage OpenSource models without the Ops hassle | by Gal Peretz | Medium
https://medium.com/@gal.peretz/openrouter-langchain-leverage-opensource-models-without-the-ops-hassle-9ffbf0016da7

[13] Building a plugin architecture with Python | by Maxwell Mapako | Medium
https://mwax911.medium.com/building-a-plugin-architecture-with-python-7b4ab39ad4fc

[14] [15] [16] [17] [19] Websocket based Streaming with Fast API and Local LLAMA 3 | by Shubham | Medium
https://medium.com/@shubham.mdsk/websocket-based-streaming-with-fast-api-and-local-llama-3-46f88eda71a2

[18] GitHub - pors/langchain-chat-websockets: LangChain LLM chat with streaming response over websockets
https://github.com/pors/langchain-chat-websockets