ChatGPT

FtSystem – Multi-Agent AI CLI Design Prompt

Overview

FtSystem is an open-source CLI application (Python) that orchestrates multiple AI agents to collaborate in real time. The user interacts with FtSystem via a terminal UI, issuing commands in Polish (the development context is Polish) while the system's code and internal prompts remain in English for precision. A central **AGENT Master** coordinates five specialized helper agents (Agents 1–5) in a shared "forum" style conversation. The Master agent decomposes user requests and delegates tasks to the helpers, who then communicate with each other by exchanging messages on a common channel. They refine each other's outputs and correct errors collaboratively, emulating a team of experts brainstorming. The final result – which could be in JSON, CSV, TXT or another format as requested – is compiled by the Master and returned to the user. This design aims to leverage collective intelligence (multiple agents with distinct roles) to solve complex tasks more effectively [1] [2]. Crucially, the system maintains state between sessions: the Master agent persistently stores condensed, professional summaries of past sessions to preserve context for future queries. The overall goal of this prompt is to guide an LLM in helping design FtSystem's architecture and code – focusing on modularity, efficiency, and integration of cutting-edge frameworks – so that a Polish developer can practically implement this multi-agent CLI tool.

Agent Architecture

FtSystem follows a mediator-based multi-agent architecture. The **AGENT Master** acts as an orchestrator (supervisor), while **Agent 1–5** are specialized sub-agents with defined roles or expertise. Each agent is an LLM-powered entity with a specific domain (for example, one might be a research expert, another a coding assistant, another a data analyst, etc., as needed). The Master receives the user's input and decides which agents to invoke for sub-tasks, effectively routing the query to the appropriate experts [3]. All agents share a common communication context – essentially a multi-party chat forum visible to all of them. The Master can post the user's query and a plan or specific assignments to this forum. Each helper agent then contributes by posting its findings, thoughts, or solutions to the forum. They **converse in natural language** (Polish for user-facing content, but potentially English internally for technical accuracy), so one agent's output becomes part of the prompt for the others [4]. This allows back-and-forth dialogue: agents can ask each other for clarifications, build on each other's answers, or point out mistakes, leading to iterative refinement. Such a design draws on recent multi-LLM paradigms like CAMEL and MetaGPT, where an initial prompt defines distinct agent roles and goals, and then the agents autonomously continue the discussion towards a solution [5] [1].

To coordinate the conversation, the Master may enforce a turn-taking policy or a structured workflow. For example, the Master could first prompt all agents to provide an initial response from their perspective, then allow a round of critique and improvement where agents react to each other's inputs. The Master monitors the discussion and intervenes if it goes off-track or needs convergence. Finally, the Master synthesizes the agents' insights into a coherent answer. The table below outlines an example interaction flow:

Step	Interaction	Involved Agents
1	**Master**: Receives the user's request (in Polish) and formulates a plan. It posts the problem and sub-task assignments on the forum (e.g. "Agent1: gather relevant data; Agent2: analyze X; ...").	Master (initiates)
2	**Agents 1–5**: Each agent works on its assigned task (possibly using tools or external APIs) and posts its results or intermediate findings to the forum. For instance, Agent1 might post research facts, Agent2 might draft code or logic, etc.	Agents 1–5 (parallel contributions)
3	**Agents (discussion)**: The agents read each other's messages on the forum. They engage in a discussion – e.g. Agent3 notices an inconsistency in Agent1's data and corrects it; Agent4 improves the code Agent2 provided; Agent5 suggests a better format for the output. They iterate until they converge on a solution.	Agents 1–5 (collaborative loop)
4	**Master**: The Master observes the dialogue and, once satisfied, aggregates the outcomes. It may ask a final agent (or itself) to format the result according to user's requested format (JSON, CSV, etc.), or do this formatting directly.	Master (synthesis)
5	**Master -> User**: The Master returns the final answer to the user through the CLI. It then generates a concise session summary (preserving key points and decisions) and saves it to the session history for long-term memory.	Master (output)

This architecture ensures that complex queries are handled by a "team" of AI specialists working in tandem, rather than a single monolithic model. By defining clear roles and a coordination mechanism, the system can break down problems and solve them via agent teamwork [6]. The shared forum (common context) is key to this collaboration, allowing the agents to stay on the same page. In implementation, this could be represented by a shared message buffer or an event loop broadcasting messages to all agents. Each agent object might have a method like `receive_message(message)` that the Master calls to update it with new forum content, and an `act()` method where the agent decides if it should contribute next. Designing careful stopping criteria (e.g., number of iterations or a signal from the Master) will prevent infinite loops in their discussion. The Master remains the only agent that communicates with the user, insulating the user from the chaotic multi-agent chatter and providing a single, authoritative response.

CLI Structure

The FtSystem CLI will be built using **Typer**, a modern CLI framework that makes it easy to define commands and options with minimal boilerplate. The CLI serves two modes: (1) a **configuration mode** for managing agents and sessions, and (2) an **interaction mode** for handling user queries via the Master agent. The user can invoke the tool in the terminal with various subcommands to configure and use the agent system. For example, possible CLI commands might include:

- `ftsystem agent create "<name>" --role "<description>" --model "<LLM_model>"` – create a new agent with a given role description and assign it a default LLM model/backend.
- `ftsystem agent list` – list all current agents (e.g., Agents 1–5) with their roles and settings.

- `ftsystem agent select <name>` – (if applicable) select or activate a specific agent profile or set of agents (though in this design, all five helper agents are usually active; this could be used to enable/disable certain agents).
- `ftsystem run "<user_query>" [--format json]` – send a one-time query to the Master agent and output the result in the specified format. This would trigger the multi-agent process internally.
- `ftsystem interactive` – enter an interactive chat session, where the user can converse with the Master agent in multiple turns (the Master will coordinate with helpers behind the scenes each turn).
- `ftsystem history show <session_id>` – display the saved summary of a past session (to review what the agents did).
- `ftsystem config set <option> <value>` – adjust configuration settings (like API keys, model defaults, etc.).

Using Typer, these can be implemented as Python functions decorated with `@app.command()` for each subcommand, giving a clean structure to the CLI. Typer will handle parsing arguments (e.g., `--format json`) and providing help messages automatically. For instance, a simplified snippet might look like:

```
import typer
app = typer.Typer()

@app.command()
def run(query: str, format: str = "text"):
    """Send a query to FtSystem and get the answer."""
    result = master_agent.handle_query(query, desired_format=format)
    print(result)

# ... (other commands like agent_create, agent_list, etc.)
```

When the user runs `ftsystem run "Zaplanuj mi tygodniowy plan treningowy" --format json`, the CLI will call the Master agent's logic to process the Polish prompt "Plan a weekly training schedule for me" and return the output in JSON. Under the hood, `master_agent.handle_query` would orchestrate the multi-agent workflow described earlier. The CLI structure should ensure that long-running tasks (e.g., multiple agents doing API calls) remain responsive; using Python's `asyncio` and launching the multi-agent deliberation as an asynchronous task can allow the CLI to show a spinner or log intermediate steps. For example, Typer can integrate with `asyncio` by running the event loop (since Python doesn't natively support `async def` commands, one can use `asyncio.run()` inside the command to execute async logic). In an interactive mode, the CLI might continuously prompt the user for input and call `master_agent.handle_query` in a loop, maintaining the session context until the user exits. This structure separates concerns nicely: the CLI (UI/UX layer) collects user input and displays output, while the core logic lies in the agent system classes.

Tools and Frameworks

To implement FtSystem efficiently, we will leverage several powerful Python libraries and frameworks:

- **Typer** – for building the CLI interface. Typer (built on Click) allows declarative definition of commands and arguments through function decorators and type hints, resulting in a clean and

user-friendly command-line tool. It automatically generates `--help` text and input validation. Using Typer will make our CLI code concise and readable while providing a professional UX (consistent with tools like `git` or `pip`).

- **Asyncio** – for concurrent execution and real-time agent communication. The multi-agent forum involves potentially parallel activities (e.g., Agent1 fetching data from the web, Agent2 computing something, etc.). Python's asyncio event loop enables us to run tasks concurrently without blocking the entire program. We can spawn each agent's task as an async coroutine and use `asyncio.gather` to have them work in parallel, which can greatly improve performance for I/O-bound operations (like API calls to LLM services). This aligns with the requirement for real-time communication – agents can effectively "talk" over the forum without strict sequential waits. We must ensure thread-safety or async-safety for shared resources (like the forum messages list), possibly by using asyncio locks or by designing the message exchange in a synchronous round-based fashion.

- **LangChain** – for LLM orchestration, prompt management, and possibly integration of tools. LangChain provides abstractions to call different LLMs (OpenAI, Anthropic, etc.) with a unified interface, and it includes constructs for agent behavior, memory, and tool use. For instance, LangChain's agent frameworks (like ReAct or Tool-using Agents) could be adapted for our multi-agent setup, or we could use LangChain simply to manage conversation history and call models with retries, etc. It also offers **Memory** components that could help with storing and retrieving conversations. Given our multi-agent scenario, we might use LangChain's ConversationChain or the newer LangGraph to manage interactions, though some custom logic will be needed for truly simultaneous agents. Using LangChain can accelerate development by handling a lot of boilerplate (LLM API calls, formatting prompts, etc.), and it is recommended for LLM integration [7]. We should explore features like **ConversationSummaryMemory** (which automatically summarizes a dialogue to keep context short) and **AgentExecutor** classes for tool usage, customizing them to multiple agents.

- **Pydantic** – for data modeling and validation. We can define Pydantic `BaseModel` classes for various internal data structures: agent configurations, messages in the forum, and the session summary format. By using Pydantic, we get automatic validation (e.g., ensure a message has a sender, content, timestamp) and easy serialization to JSON/YAML. For example, a `Message` model might have fields: `agent_name: str`, `role: Literal["user","assistant","system"]`, `content: str`, `timestamp: datetime`. Pydantic can help parse and enforce types when reading/writing the history files or exchanging info between agents. It's also useful for validating final outputs – e.g., if the user requests JSON output, we could define a Pydantic model for that output schema and have the Master agent's result validated or corrected against it. This approach improves robustness and clarity of the codebase.

In addition to these, standard libraries like `logging` (to trace agent actions) and perhaps **Rich** (to color-code console output per agent, improving readability) would be useful. The chosen frameworks emphasize **modularity** (Typer commands, Pydantic models) and **concurrency** (asyncio, LangChain's async support), aligning with our goal of a scalable, real-time system.

Memory Management

Persistent memory is crucial for FtSystem, as each agent should "remember" important context from previous sessions. The strategy is to maintain both short-term and long-term memory:

- **Short-term memory (within a session):** This is essentially the conversation context that's actively kept in the prompt for the agents during a live session. It includes the user's query and a summary of the relevant recent dialogue on the forum. Because we have multiple agents chatting, the context can grow quickly, so the Master should summarize or prune it as the conversation progresses (for example, keeping a running summary of the discussion so far, rather than every single message, to avoid exceeding token limits). LangChain's in-memory `ConversationBuffer` or `ConversationSummaryMemory` can be adapted so that the Master feeds each agent a concise context of what has transpired.

- **Long-term memory (across sessions):** After each session, the Master agent will generate a **session history file** containing a condensed but information-rich summary of the interaction. This summary is meant to preserve essential context (facts uncovered, decisions made, the user's preferences, etc.) in a permanent form that can be reloaded in future sessions. The format of this memory file should balance human readability with machine parseability. Possible formats include JSON Lines (JSONL), YAML, Markdown, or a hybrid. For efficiency and structured access, **JSONL** is a strong option: each line could be a JSON object representing a turn or a summary chunk, easily appendable and loadable. Alternatively, a Markdown file with section headings (for each session and agent) could be more readable for developers inspecting it, though parsing it might be trickier. We could even combine approaches: e.g., a Markdown file that contains embedded JSON snippets. The key is that retrieving past knowledge should be faster than recomputing it; the system might load the latest summary and provide it to the Master agent as context at the start of a new session (or on demand when relevant).

For example, after a session, the Master might save a JSONL file like `history_2025-07-27.jsonl` where each line has `{ "session": 42, "summary": "In this session, user asked about X, agents concluded Y after debating Z.", "key_facts": [...], "outcome": "final answer summary" }`. On the next run, the Master can quickly search these summaries for relevant keywords or just load the last summary to remind itself of the recent context. This approach is inspired by techniques in conversational AI where short-term interactions are summarized for long-term storage [8]. Storing conversations in a database or file with both short-term and long-term memory ensures the system can handle nuanced follow-up questions (e.g., if in a previous session the user defined "Project Alpha", and next time they ask "Did we meet the goals for Project Alpha?", the system can recall what Project Alpha is). By summarizing, we reduce noise and only carry forward important information, mitigating context window limits.

In implementation, we could have a `MemoryManager` component. It might use Pydantic models like:

```
class SessionSummary(BaseModel):
    session_id: str
    date: str
    user_query: str
    agent_findings: List[str]  # major points from agents
    conclusion: str  # final answer given
```

The **MemoryManager** would have methods like `save_summary(SessionSummary)` and `load_recent_summaries(n)`. We could also integrate vector databases (e.g., FAISS or Pinecone) to store embeddings of past knowledge for semantic recall, but to start, simple structured text files may suffice. The session summaries should be written in a professional, technical style (since the user is a developer, they'll appreciate succinct, factual recap rather than verbose prose). If multiple sessions exist, the Master might load a few recent ones or those tagged as relevant. Ensuring the memory files are easily diff-able (for version control, given this is open-source) is also beneficial – e.g., JSONL or Markdown in a git repo can show how the knowledge evolves. In summary, **the Master agent will act as the historian**, creating and consulting these summaries to give the multi-agent system continuity over time.

Prompt Engineering Rules

To harness the full power of the LLMs and avoid confusion in a multi-agent setting, we must carefully craft the prompts and conversation rules for each agent. The prompt given to the LLM (this very document) should include instructions that ensure structured, role-based interaction and high-quality outputs. Key prompt engineering considerations include:

- **Role Definitions:** At the start of a session (or when initializing the system), provide each agent with a clear "persona" and objective in a system prompt. For example: *"You are Agent 1, an AI specialized in web research. Your job is to find relevant data and share it with the team."* Similarly: *"You are Agent 2, a Python coding expert who writes and evaluates code."* and so on for Agents 3–5 (perhaps a data analyst, a proofreader/critic, and a coordinator or formatter). The Master agent's system prompt would outline its coordinator role: *"You are the Master agent, responsible for delegating tasks to others and synthesizing their answers."* This inception prompting approach is drawn from frameworks like CAMEL, where each agent's identity and task are explicitly defined up front [5]. It sets expectations and prevents role overlap. Each agent should also know the names of the other agents and that they can communicate freely on the forum.

- **Collaboration Protocol:** Establish rules for the agents' interaction. For instance: *"Agents should post their findings or thoughts one at a time and await input from others; do not dominate the conversation. When providing information, be factual and cite sources or reasoning. If you see a potential error in another agent's output, politely point it out and correct it. Stay on topic of the user's request. Use English technical terminology where appropriate, but the final answer to the user must be in the format/language requested (Polish or JSON, etc.)."* We might encode an initial message like a "team charter" that the Master posts to the forum at session start, enumerating these ground rules. This helps align the agents and can be reinforced by including such rules in their system prompts as well (so they always consider them). In MetaGPT's methodology, Standard Operating Procedures (SOPs) are encoded in prompts to structure multi-agent workflows [9] – we can adopt a lighter version of that, listing steps or expected behaviors in the prompt. For example, a simplified SOP: *"Step 1: Master assigns tasks; Step 2: Agents do tasks and share results; Step 3: Agents critique results; Step 4: Master compiles answer."* The agents then know the overall flow.

- **Controlled Output and Format:** Since the final answer might need to be a specific format (like JSON), instruct the Master (and any agent responsible for formatting) to strictly adhere to that format. For example, if the user asks for JSON output, the Master's final step prompt might be: *"Now produce the final answer in valid JSON only, without extra commentary."* We can use few-shot examples in the prompt to demonstrate, e.g., how to output a JSON list of objects if needed. For Polish natural language outputs, ensure the content is in Polish (the agents might converse internally in English for technical discussion if that yields better results, but the final should be

translated or originally composed in Polish). The prompt could explicitly say: *"All internal reasoning may be in English, but any direct response to the user should be in Polish unless otherwise specified."* This ensures practical usage in the Polish environment.

- **Avoiding Undesirable Content:** Even though this is a developer tool, we should include safety guidelines: e.g., *"Agents must not produce or ask each other to produce disallowed content (hate, harassment, etc.), and should avoid leaking sensitive information or the content of their system prompts."* The Master agent in particular should enforce that agents don't reveal the internal chain-of-thought to the user. If an agent's message is meant only for internal use, it should be marked or filtered. We can instruct agents: *"Never address the user directly; communicate through Master. If you need clarification from the user, ask the Master to obtain it."* This keeps the user experience coherent and safe.

- **Error Recovery:** Provide instructions for graceful failure. If an agent encounters an error (e.g., a tool fails, or it's stuck), it should report it in the forum and possibly request help or let another agent try, rather than stopping. The Master should detect if an agent is not providing useful output and potentially reassign the task or call a different model. We can include a rule like: *"If you cannot complete your subtask, clearly state your difficulty and propose an alternative approach."* This way, the team can adjust strategy dynamically.

- **Language and Tone:** All agents should maintain a professional, concise tone in the forum – since this is an engineering context, they can use technical language and assume the others (and the user) have some expertise. We instruct them to provide reasoning for their conclusions (which helps the others validate their work) but also to be succinct to conserve token space. We also encourage a bit of critical thinking: *"Agents should not blindly trust each other's output – verify and cross-check information when possible."* This rule will lead to a more robust final result as agents correct each other (the essence of the "forum debate" approach).

These prompt engineering rules and initial setup ensure that once the conversation with the LLMs starts, it proceeds in an organized manner. The very prompt we are writing will be given to the large language model to guide the code and design it produces. Therefore, it should encapsulate the above points clearly. In summary, by **defining roles, rules, and an interaction protocol in the prompt**, we align the LLM's output with the desired multi-agent system behavior [10]. The result will be that the LLM (like GPT-4, etc.) will generate code and plans that naturally incorporate these rules (such as creating classes for agents with role descriptions, implementing the turn-taking logic, etc.).

API Integration

FtSystem's design allows each agent to use a different LLM or tool backend via API – this flexibility is crucial for leveraging the strengths of various models (OpenAI GPT-4, Anthropic Claude, Google Gemini, etc.). To implement dynamic model selection, the system will incorporate an **API integration layer** abstracting the differences between providers. Concretely, we might have a configuration (in a JSON/YAML config file or environment variables) that maps each agent to a specific model or service. For example: `{agent1: "openai:gpt-4", agent2: "anthropic:claude-v2", agent3: "google:gemini", agent4: "openai:gpt-4-32k", agent5: "local:LLama2"}`. The Master or a factory function can read this config and initialize API clients accordingly. LangChain can simplify this, as it offers wrappers for many LLM APIs – we could use `OpenAI()` for GPT-4, `Anthropic()` for Claude, etc., all conforming to a common `.predict()` or `.generate()` interface. If not using LangChain, we'll utilize the official SDKs: e.g., OpenAI's `openai` Python library, Anthropics' `anthropic` client, etc. We should design an **AgentModel** class or similar to encapsulate the API calls,

so that an agent can do `response = self.model_client.query(prompt)` without worrying about which API it is. This also allows easy switching of models or providers – for instance, if one API fails or hits a rate limit, the Master could fall back to a different model dynamically.

Concurrency is again important here: calling multiple external APIs in parallel can speed up responses (making use of asyncio and Python HTTP libraries like httpx which support async). However, we must consider API rate limits and cost – a real implementation might not always query all five agents simultaneously for every user question, especially if using expensive models. The Master's logic could be adaptive: it might only engage the agents it deems necessary. For instance, a straightforward query that doesn't need coding might not invoke the coding agent at all. This decision could be rule-based or even learned. Initially, though, a simple mapping from user query type to agent might be enough (for example, if query contains "code" or "implement", call the coding agent).

We will also integrate **tool use** via APIs. Some agents might act less as pure LLMs and more as tool callers. For example, an agent could use an API to fetch information (like a web search API, a database query, etc.) instead of relying only on its model's knowledge. This could be achieved by implementing certain agents as wrappers that call tools: e.g., Agent1 might use a search API (SerpAPI or others) and then feed results to GPT-4 for analysis. LangChain's tool system or OpenAI's function-calling feature could be utilized here: the agent's prompt could be engineered to output a function call (like `search("query")`) which our system catches and executes, then the agent continues with the result. This is an advanced pattern; a simpler route is to explicitly code the agent to use Python `requests` or specialized SDKs when needed (like an agent with access to a weather API if the task demands).

To manage API credentials securely, we won't hard-code keys. Instead, we use environment variables or a config file (Typer can help prompt the user to set these up via `ftsystem config`). For example, we might require `OPENAI_API_KEY`, `ANTHROPIC_API_KEY`, etc., in the environment. The system can check at startup that the necessary keys for the chosen agents are present, and give a clear error if not. This mirrors how many tools handle secrets.

Testing the integration with multiple models is important – each LLM has its quirks (prompts may need slight tweaking per model). The prompt should encourage building a **modular API layer** where adding a new model is as easy as writing a small adapter class. E.g., a `BaseLLMClient` class with subclasses `OpenAIClient`, `ClaudeClient`, etc., each implementing a `generate(text)->str` method. This way, Master and agents don't care what the backend is. We also consider local models (perhaps via Hugging Face Transformers interface) to allow offline or self-hosted usage.

Finally, since FtSystem might eventually integrate with the user's OS (like installing packages or running code as per the user's request), those actions themselves can be seen as "API" integrations – e.g., using Python's `subprocess` module for shell, or a package manager API. The design should keep these as pluggable behaviors that an agent (or Master) can call when authorized. By planning the API integration carefully, we future-proof FtSystem to use new LLMs or tools as they emerge, simply by updating configurations or adding new agent classes.

Sample Code Skeleton

Below is a simplified **code skeleton** for FtSystem, illustrating the core components and their interactions. The code is annotated with comments to explain each part and to highlight areas for future debugging or scaling. This is not full implementation, but a starting blueprint:

```python
# agents.py
from typing import List, Any
import asyncio
import openai  # or anthropic, etc., depending on the agent

class Agent:
    """Base class for an AI agent in the FtSystem."""
    def __init__(self, name: str, role: str, model_client: Any):
        self.name = name            # e.g., "Agent1"
        self.role_description = role  # e.g., "Web Researcher Agent"
        self.model = model_client   # an LLM API client or wrapper
        self.memory: List[str] = [] # local memory of messages relevant to
this agent

    async def process(self, task: str, forum: List[str]) -> str:
        """
        Handle a task assigned by the Master. `task` is a prompt or
instruction.
        `forum` is the shared conversation so far (if needed for context).
        Returns the agent's message (as a string) to post on the forum.
        """
        # Combine role, task, and context into a prompt for the LLM
        prompt = f"{self.role_description}\nYou are tasked: {task}\nForum
discussion so far:\n"
        prompt += "\n".join(forum[-5:])
# include last 5 messages from forum for context
        prompt += f"\nYour answer:"
        # Send to model (e.g., OpenAI API) - this is a placeholder call:
        response = await self.model.generate(prompt)
        # ^ model.generate would be an async method calling the actual API
(OpenAI, etc.)
        answer = response.strip()
        # Save to agent's memory (could also update a global memory)
        self.memory.append(answer)
        return f"{self.name}: {answer}"

class MasterAgent:
    """Master agent that coordinates the helper agents."""
    def __init__(self, agents: List[Agent]):
        self.agents = agents
        self.forum: List[str] = []  # collects all messages exchanged

    async def handle_query(self, user_query: str, format: str = "text") ->
str:
        # Initial system message or plan (Master decides what to do)
        self.forum.append(f"Master: User asked: {user_query}")
        # Simple strategy: assign the same query to all agents (could be
refined per role)
        tasks = [agent.process(user_query, self.forum) for agent in
self.agents]
```

```
        # Run all agents concurrently
        agent_replies = await asyncio.gather(*tasks)  # each agent returns
its forum message
        # Post all replies to forum
        for reply in agent_replies:
            self.forum.append(reply)
        # Potentially, allow a round of discussion among agents (omitted for
brevity)
        # Finally, Master compiles the answer:
        final_answer = await self._synthesize_answer(format)
        # Save session summary to file for long-term memory
        self._save_session_summary(user_query, final_answer)
        return final_answer

    async def _synthesize_answer(self, format: str) -> str:
        """Compile a final answer from forum messages, possibly using an
LLM."""
        # Here we simply gather all agent outputs. In practice, we could
prompt another LLM
        # or one of the agents (e.g., Agent5 as a 'Synthesizer') to create
the final answer.
        all_outputs = "\n".join([msg for msg in self.forum if
msg.startswith("Agent")])
        if format.lower() == "json":
            # If JSON requested, wrap outputs in JSON format (simplified
example)
            import json
            data = {"responses": {f"agent{i+1}": msg.split(":",1)[1].strip()
                                  for i, msg in
enumerate(all_outputs.splitlines())}}
            return json.dumps(data, ensure_ascii=False, indent=2)
        else:
            # Plain text format: just return aggregated outputs
            return f"Odpowiedź:\n{all_outputs}"

    def _save_session_summary(self, user_query: str, answer: str):
        """Summarize the session and save to a history file (e.g., JSONL or
Markdown)."""
        summary = {
            "user_query": user_query,
            "agents": [agent.name for agent in self.agents],
            "conversation": self.forum,
            "final_answer": answer
        }
        # For brevity, write summary as JSON (could append to a .jsonl file)
        import json
        with open("ftsystem_history.jsonl", "a", encoding="utf-8") as f:
            f.write(json.dumps(summary, ensure_ascii=False) + "\n")
        # In a real scenario, we would store only a condensed summary, not
full conversation.
```

```
# model_clients.py (wrappers for different LLM APIs)
class OpenAIClient:
    def __init__(self, model_name: str, api_key: str):
        openai.api_key = api_key
        self.model_name = model_name
    async def generate(self, prompt: str) -> str:
        # Async call to OpenAI chat completion (note: openai lib may not be
async,
        # so consider using `await asyncio.to_thread(...)` to avoid blocking)
        resp = await asyncio.to_thread(openai.ChatCompletion.create,
                                       model=self.model_name,
                                       messages=[{"role": "user", "content":
prompt}])
        return resp['choices'][0]['message']['content']

# Similar clients could be made for Claude, Gemini, etc., each with
a .generate() method.

# main.py (CLI using Typer)
import typer
app = typer.Typer()
master_agent: MasterAgent = None  # will be initialized in init_config

@app.command()
def init_config(openai_key: str = typer.Option(..., prompt=True,
hide_input=True)):
    """Initial configuration: setup API keys and agents."""
    global master_agent
    # Initialize model clients for each agent (example: 5 GPT-4 agents here)
    model_client = OpenAIClient(model_name="gpt-4", api_key=openai_key)
    agents = []
    roles = ["Researcher", "Coder", "Analyst", "Critic", "Summarizer"]
    for i, role in enumerate(roles, start=1):
        agents.append(Agent(name=f"Agent{i}", role=f"You are an expert
{role}.", model_client=model_client))
    master_agent = MasterAgent(agents=agents)
    typer.echo("Agents initialized and Master is ready.")

@app.command()
def query(q: str, format: str = "text"):
    """Ask a question to the Master agent (with optional output format)."""
    if master_agent is None:

typer.echo("Error: System not initialized. Run 'ftsystem init-config'
first.")
        raise typer.Exit(code=1)
    # Run the master_agent to handle the query
    answer = asyncio.run(master_agent.handle_query(q, format=format))
    typer.echo(answer)
```

```
# Additional CLI commands (agent management, history, etc.) would be defined
similarly.
```

Notes on the skeleton: This code provides a high-level structure. We define an `Agent` class and a `MasterAgent` orchestrator. Each `Agent.process` method builds a prompt including the agent's role and the latest forum context, and then calls its `model_client` to get a response (the actual LLM call, which in this pseudo-code is represented by an async `generate` function). The MasterAgent's `handle_query` coordinates the agents by launching all their tasks concurrently and collecting their replies. After that, `_synthesize_answer` simply concatenates outputs or formats them, but in a more refined implementation this could invoke another LLM (or a designated agent) to do a proper synthesis of all contributions. We also show how to save a session summary to a file (`ftsystem_history.jsonl`) – in practice we'd store a trimmed summary rather than everything. The CLI part demonstrates using Typer to create commands for initializing the system and querying it; in a real application we'd add more commands as needed (like viewing history, adding an agent, etc.). We also included an example of how an `OpenAIClient` might look – encapsulating the API call. In debugging and scaling, this separation of concerns (agents vs. model clients vs. CLI) helps: for instance, if one model API is slow or fails, you can swap that out by editing `model_clients.py` without touching the agent logic. The code has comments to guide further development (e.g., using `asyncio.to_thread` to call sync APIs without blocking, refining the forum context window, etc.). This modular design will make future enhancements (like adding a new agent specialization or changing the memory backend) easier to implement and test.

Suggested Cursor MCPs & Extensions

To extend FtSystem's capabilities in the Cursor IDE environment (or any IDE with AI integration), we should leverage **MCP (Model Context Protocol) servers and other extensions** that grant agents additional tools. Here are some recommended integrations and how they benefit our multi-agent system:

- **Code Execution (Code Interpreter)**: Enabling a code interpreter allows an agent to execute Python code in a sandboxed environment and return results. This is extremely useful for Agents tasked with calculations, data analysis, or code verification. For example, the coding agent could write a snippet and actually run it to test its correctness. In the Cursor ecosystem, the Together AI Code Interpreter (TCI) can be used as an MCP server, making it easy to plug into Cursor or similar IDEs [11]. By configuring this, our agents can seamlessly do things like generate charts, perform file I/O on test data, or execute algorithms – all within safe confines. This turns the abstract reasoning of LLMs into concrete, verifiable actions. In practice, we might implement this by having the agent, upon deciding to run code, call a function (or use LangChain's tool interface) that sends the code to the interpreter service and then captures the output to bring back to the forum.

- **File System Access**: Since FtSystem runs on the user's machine (like VS Code or Cursor environment), it should allow agents to read from or write to the local file system (with the user's permission). This is important for tasks like reading a local dataset, writing out a report or code file, or storing intermediate results. We can integrate a file system tool such that an agent can say (in the forum) something like "I will save these results to `output.csv`" and the system will actually execute that action. Cursor doesn't have a specific MCP for file I/O listed, but we can implement this using Python directly (ensuring we validate paths to avoid destructive actions). Alternatively, treat the shell access (below) as a way to use `cat` / `echo` commands for file reading/writing. We should apply safeguards: e.g., agents should not overwrite important files or

access restricted directories. In the prompt, instruct them to confirm with Master or user before performing major file operations.

- **Shell Command Execution**: Giving agents the ability to execute shell commands can greatly extend functionality – for instance, installing a needed Python package, running a system utility, or using command-line tools. Cursor supports this via an MCP Shell Server [12], which acts as a bridge for AI agents to safely run shell commands on the host system. By integrating the shell server, an agent could say "I'll run a quick system command to check X" and the command's output would be fed back. This can be helpful for tasks like retrieving system info, managing environment (like starting a local server if the user's request involves that), or using git for version control in a coding task. However, shell access must be used with caution – the system should have rules (as seen in the MCP Shell tool's guidelines [13]) to prevent destructive actions. For example, always explain a command before running it, and never run a command that wasn't explicitly part of solving the user's request. In FtSystem, we'd incorporate these rules in our agent prompt and possibly require a confirmation from the user for high-risk commands.

- **Web Browsing / API Calls**: Sometimes agents need information beyond their training data – e.g., the latest stock prices or an updated library documentation. Integrating a web browsing tool (such as **Browserbase**, an MCP server providing a headless browser [14]) would allow an agent to fetch live webpages. Similarly, an **API tool** can let agents query external APIs directly. For instance, an agent could call a weather API if the task is to plan an outdoor event schedule, or call a translation API for multilingual support. In the Cursor MCP directory, there are connectors like **Zapier** (to interface with thousands of apps) [15] and **Pipedream** (with access to 10k+ API endpoints with built-in authentication) [16]. These could be integrated so that agents have virtually unlimited access to external services in a controlled way. In practice, you would configure API keys for these services and possibly use LangChain's tool wrappers or a custom plugin system where an agent's output like `API_CALL[service, params]` triggers a function in our code.

- **Cursor-specific Extensions**: If using Cursor IDE, take advantage of any built-in features like the *Code Analyzer or Bugbot* for code review. For example, if Agent4 is a "Critic" agent, it might benefit from using a static analysis tool (Cursor's Bugbot) on code produced by Agent2, and then report issues. While not exactly MCP, these are environment features that can be triggered via CLI or API. Additionally, Cursor's "Memories" feature could be explored for storing context in the editor – but since we already manage memory internally, this may overlap.

To incorporate these extensions, one must set up the environment properly. For MCP servers, that means updating Cursor (or VS Code with Cline extension) config files to register these tools (as shown in the Shell server installation instructions) [17] [18]. From the prompt design perspective, we instruct the LLM (when generating code) to include hooks or classes for tool usage. For example, we might have a `ToolManager` class that interfaces with these MCP servers via HTTP or local calls. An agent, upon deciding to use a tool, would call `ToolManager.run("shell", "ls -la")` or `ToolManager.run("code_interpreter", "python_code_string")`, etc., and the ToolManager handles the rest (sending to the MCP server and returning output).

By recommending and integrating these tools, FtSystem moves closer to systems like Auto-GPT or "AI DevOps assistants" that can not only think and communicate but also act on the world (execute code, modify files, query APIs). This dramatically increases practical usefulness. For example, a user could ask, "Agent, create a new Django project and set up a basic app," and with shell and code execution access,

the agents could actually initialize a project folder, run `django-admin startproject`, and so on, rather than just telling the user what to do.

In conclusion, the prompt to the LLM (which generates FtSystem's design) should emphasize modular design allowing these extensions. The code should be written with a mindset that adding a new tool is as simple as adding a new method in ToolManager and a new permission in the agent's rules. By planning for MCP and extensions early, FtSystem will be a forward-looking AI agent CLI, ready to interface with the broader ecosystem of developer tools and APIs, much like providing a universal "USB-C port" for AI capabilities [19] – flexible, powerful, and aligned with the latest industry standards.

---

[1] [4] [5] [10] [19] AI-to-AI Communication: Strategies Among Autonomous AI Agents | by Adnan Masood, PhD. | Medium
https://medium.com/@adnanmasood/ai-to-ai-communication-strategies-among-autonomous-ai-agents-916c01d49c15

[2] [6] [9] What is MetaGPT ? | IBM
https://www.ibm.com/think/topics/metagpt

[3] [7] [8] Help with Building a Multi-Agent Chatbot : r/LangChain
https://www.reddit.com/r/LangChain/comments/1k6wonv/help_with_building_a_multiagent_chatbot/

[11] Together Code Interpreter - Together.ai Docs
https://docs.together.ai/docs/together-code-interpreter

[12] [13] [17] [18] MCP Shell Server | Glama
https://glama.ai/mcp/servers/@mkusaka/mcp-shell-server

[14] [15] [16] Cursor – MCP Servers
https://docs.cursor.com/en/tools/mcp