1. Reasoning

1.1 Context

# Context

* My project uses AG2 multiagent framework to implement an AI multi-agent platform.
* My platform UI is a Chat.
* In this chat the final user can post questions about business matters.
* My platform can access multiple API from multiple business software that my multinational company already commercializes.
* Using these APIs and responding to user prompts the users can interact with all this Apps.
* Perform operations, get data, Analize data, etc. directly in the chat.
* Behind the chat there are multiple agents, LLMS, intelligence, skills, actions, code that support the chat execution.
* This way the users can do much more, in an easy manner, and add intelligence and natural language to their operations, analysis, etc.
* They can do much more that they were able to do, with the normal applications, and in an easier and intelligent way.

1.2 Prompt

# Prompt

* My platform uses a NG2 GROUP CHAT.
* This pattern allows for multiple agents to interact with each other, and try to solve the problems that the user prompts.
* This way of doing things is not good.
* I need to understand how to introduce the NG2 Reasoning Agent and the new capabilities of the reasoning LLMs like OpenAI O1 and O3 and other technologies/tools, to plan the execution of a procedure.
* This planned procedure will allow for the best solution to be found.
* With the best solution I need my platform to select the correct agents from all the available ones.
* Then I need my platform to execute this procedure, coordinating the execution of the previously selected multiple agents in the most optimized way, to achieve the best result.
* Create a report that analyses all my problems and researches technologies/tools/solutions and document them - State of the art (SOTA).
* Then after the SOTA I need the report to display the best approach and explain the rationale used to choose the best approach.  
  In the end I need the report to contain a step-by-step, detailed guide on how to implement the best solution.

1.3 Optimization

**Context:** I am developing an AI-powered multi-agent platform leveraging the AG2 (<https://docs.ag2.ai/docs/home/home>) multi-agent framework. The platform provides a chat-based interface where end-users interact with multiple APIs from various business software solutions commercialized by our multinational company. Users can perform operations, request data, analyze information, and execute complex business processes directly within the chat interface, significantly enhancing the usability and intelligence of their interactions compared to traditional applications. This interaction is powered by multiple AI agents, LLMs, intelligent skills, actions, and backend code.

Currently, the system utilizes an AG2 Group Chat pattern, where multiple agents interact dynamically to address user prompts. However, this approach is proving suboptimal for solving complex user requests efficiently and effectively.

**Research Objectives:**

1. **Identify and analyze the limitations** of the current AG2 Group Chat pattern in multi-agent interaction.
2. **Framing my problem** in the context of the evolution of LLMs and the technology of major players in the area such as OpenAI, Antropic, etc., as well as in the more general problem of the evolution of AI to AGI.
3. **Explore advanced reasoning technologies**, particularly focusing on reasoning agents (AG2 Reasoning Agent) and state-of-the-art reasoning LLMs such as OpenAI O1, O3, and similar cutting-edge models.
4. **Evaluate tools, methodologies, and technologies** suitable for planning, orchestrating, and optimizing the execution of procedures involving multiple intelligent agents.
5. **Recommend and justify the optimal solution** by clearly articulating the rationale behind selecting specific technologies, reasoning agents, and orchestration methodologies.

**Deliverables:**

Your response must include a comprehensive report structured as follows:

1. **Problem Analysis:**Clearly detail the challenges faced by the current AG2 Group Chat pattern, explaining why it fails to deliver optimal results for complex, business-oriented user interactions.
2. **Framing my problem**Describe how my problem relates to the context of the evolution of LLMs and the technology of major players in the area such as OpenAI, Antropic, etc., as well as in the more general problem of the evolution of AI to AGI.
3. **State-of-the-Art Research (SOTA):**Conduct an extensive review of current best practices, innovative technologies, and recent advancements in:
   * Multi-agent reasoning systems
   * Advanced reasoning models (e.g., OpenAI O1, O3)
   * Procedure planning and execution frameworks
   * Agent orchestration tools and methodologies
4. **Recommended Approach:**Clearly define the best-suited architecture, agents, and reasoning models to address the identified challenges. Explain your decision-making criteria, including technical robustness, scalability, ease of integration, performance optimization, and user experience enhancement.
5. **Detailed Implementation Guide:**Provide a clear, step-by-step implementation roadmap covering:
   * Integration of the AG2 Reasoning Agent
   * Deployment and configuration of chosen reasoning LLMs (e.g., OpenAI O1, O3)
   * Procedure planning methodologies
   * Selection and orchestration of appropriate agents
   * Optimization strategies for coordinated multi-agent execution

Your response should facilitate immediate understanding, decision-making, and technical implementation to significantly enhance my platform's capabilities.

1.4 Questions

1. What industry or domain does your multinational company operate in (e.g., finance, logistics, healthcare, etc.)?

Cegid is a French multinational company that specializes in providing cloud-based business management solutions across various industries. Its offerings cater to sectors such as finance, human resources, retail, and fashion. Cegid's product portfolio includes finance and tax software, human resources management tools, and solutions designed to optimize retail operations and enhance customer experiences. citeturn0search4

In the retail domain, Cegid delivers omnichannel solutions aimed at improving in-store customer interactions and streamlining supply chain processes. For human resources, the company offers platforms that facilitate talent acquisition, management, and learning. Additionally, Cegid provides tools for financial management, including accounting, taxation, and cash flow optimization. citeturn0search0

Overall, Cegid operates primarily in the software industry, with a strong focus on delivering specialized solutions for finance, human resources, and retail sectors.

1. Are there specific business processes or API use cases you'd like us to prioritize (e.g., ERP data queries, CRM updates, supply chain workflows)?  
   All if possible: ERP data queries, CRM updates, supply chain workflows
2. Do you already use or plan to use a specific orchestration tool (e.g., LangGraph, CrewAI, Autogen, or a custom orchestrator)?  
   No
3. What is your current deployment environment (e.g., cloud provider, on-premises, hybrid)?  
   AZURE
4. Are you targeting real-time interaction, batch processing, or both?  
   both

1.4.1 Second

To tailor this research effectively to your needs, could you please clarify a few points:

1. What types of complex user requests are currently proving difficult with the AG2 Group Chat pattern? (e.g., multi-step workflows, cross-API data aggregation, error handling)  
   all: multi-step workflows, cross-API data aggregation, error handling
2. Are you already using specific APIs or LLM providers (e.g., OpenAI, Anthropic) in your backend, or are you open to evaluating all major players?  
   I’mn using OpenAI but I’m open to evaluating all major players
3. Is your deployment environment primarily cloud-based (e.g., AWS, Azure) or hybrid/on-premise?  
   AZURE
4. Do you have any constraints or preferences regarding licensing, latency, or model hosting (e.g., using hosted APIs vs. running models locally)?  
   hosted apis
5. Are there existing orchestration layers, planning components, or agent memory systems you’re using that this research should take into account?  
   no

2. Report

## **Problem Analysis**

The current NG2 Group Chat pattern, although useful for simple, dynamic agent interactions, faces significant limitations for complex business-oriented tasks involving multiple APIs, reasoning steps, and procedural execution. Its limitations include:

* **Lack of Coordinated Planning**: Agents respond reactively without structured strategic planning, leading to inefficient or redundant processes.
* **Inefficient Reasoning**: Simple agent conversations struggle with deep reasoning tasks, causing incomplete or suboptimal responses.
* **Scalability Issues**: Increasing complexity significantly reduces system responsiveness and reliability due to the exponential growth of interactions.
* **Limited Orchestration Capabilities**: Current methods lack sophisticated orchestration, making it difficult to manage dependencies and workflows efficiently.

## **Framing the Problem**

Your challenge sits at the forefront of AI evolution from basic LLM-driven interactions towards more advanced Artificial General Intelligence (AGI). Industry leaders like OpenAI and Anthropic are evolving their models towards greater reasoning capabilities and structured, orchestrated interactions. Advanced models (e.g., GPT-4 Turbo/O1, Claude 3/O3) increasingly integrate structured reasoning, context retention, and enhanced orchestration, reflecting broader AGI trends.

This evolution emphasizes the necessity for structured reasoning and orchestration technologies in enterprise settings, supporting robust, scalable, and intelligent business interactions.

## **State-of-the-Art Research (SOTA)**

### **Multi-agent Reasoning Systems**

* **LangGraph** (LangChain Ecosystem): Provides structured planning, complex reasoning capabilities, and optimized agent communication.
* **AutoGen**: Facilitates advanced orchestration, efficient state management, and enhanced multi-agent planning through a dynamic execution environment.

### **Advanced Reasoning Models**

* **OpenAI GPT-4 Turbo (O1)**: Strong contextual reasoning, improved procedural execution, and specialized fine-tuning capabilities.
* **Anthropic Claude 3 (O3)**: Exceptional reasoning depth, context management, and reliable performance for complex problem-solving scenarios.

### **Procedure Planning and Execution Frameworks**

* **Semantic Kernel (Microsoft)**: Integrates procedural planning directly within AI agents, supporting structured execution and sophisticated multi-step interactions.
* **AutoGen Framework**: Particularly effective for orchestrating complex workflows involving multiple reasoning agents and execution paths.

### **Agent Orchestration Tools**

* **LangGraph**: Provides scalable, structured workflows with clear dependency management.
* **AutoGen**: Optimized for multi-agent scenarios, providing efficient workflow management, agent selection, and action sequencing.

## **Recommended Approach**

Given your scenario and objectives, the optimal solution comprises:

* **Reasoning Framework**: AutoGen enhanced with LangGraph for advanced reasoning and orchestration.
* **Reasoning LLMs**: Primarily OpenAI GPT-4 Turbo (O1) combined selectively with Anthropic Claude 3 (O3) for high-complexity reasoning.
* **Integration Approach**: Using AutoGen for backend orchestration, enabling agents to effectively execute ERP queries, CRM operations, and supply chain management tasks seamlessly.

This recommendation prioritizes:

* **Technical Robustness**: AutoGen + LangGraph are proven frameworks supporting enterprise-scale complexity.
* **Scalability**: Easily scales horizontally across Azure infrastructure.
* **Ease of Integration**: Compatible with Azure's ecosystem and REST APIs, simplifying integration efforts.
* **Performance Optimization**: Optimal load balancing and dynamic agent orchestration capabilities.
* **Enhanced User Experience**: Rich, coherent interactions driven by advanced reasoning models.

## **Detailed Implementation Guide**

### **Step 1: Integration of NG2 Reasoning Agent**

* Migrate current NG2 agents into AutoGen.
* Introduce LangGraph layer to manage complex reasoning paths and agent communications.

### **Step 2: Deployment and Configuration of Reasoning LLMs**

* Deploy GPT-4 Turbo (O1) and Claude 3 (O3) via Azure OpenAI and Anthropic APIs.
* Configure specialized fine-tuning using enterprise-specific data for enhanced performance.

### **Step 3: Procedure Planning Methodologies**

* Implement Semantic Kernel to define complex workflows and procedural planning.
* Embed planning logic within AutoGen orchestrators to manage execution dynamically.

### **Step 4: Agent Selection and Orchestration**

* Define clear agent responsibilities (ERP agent, CRM agent, supply chain agent).
* Configure orchestration in AutoGen to dynamically select and invoke agents based on task complexity and context.

### **Step 5: Optimization Strategies**

* Continuously monitor agent interactions, optimizing orchestration via AutoGen's adaptive state management.
* Implement telemetry and analytics within Azure Monitor for real-time insights and proactive optimization.

This approach ensures efficient, intelligent, and scalable interaction, transforming your AI-driven business processes into highly effective and insightful experiences.

2.1 Second

# **Optimizing AG2-Based Multi‑Agent Platform for Cegid**

## **1. Problem Analysis**

Cegid’s multi-agent chat platform (built on AG2’s Group Chat pattern) currently faces significant challenges with **complex user requests**.

The **AG2 Group Chat pattern** allows multiple LLM-driven agents to converse freely, but it lacks structure for coordinating **multi-step workflows**, aggregating data from multiple APIs, and handling errors reliably.

The shortcomings can be summarized as follows:

* **Multi-step Workflows:** Group Chat orchestrates agents via free-form conversation, without an explicit plan for multi-step tasks. This means there is no guaranteed stepwise execution order – agents speak opportunistically, guided only by a GroupChatManager’s next-speaker selection. As a result, complex workflows can derail or loop because the system doesn’t enforce a structured sequence of actions. In a business scenario (e.g. an ERP query that requires subsequent approval then an update), the free-form approach may skip or repeat steps, leading to incomplete transactions or requiring repeated user prompts.
* **Cross-API Data Aggregation:** In the current pattern, each agent may call a different business API (ERP, CRM, supply chain, etc.) and report its result in the chat. However, there is no dedicated mechanism to **aggregate these outputs** into a coherent answer. The conversation pattern relies on agents themselves to integrate information, which is unreliable. For example, one agent might retrieve inventory data from an ERP, and another fetch customer data from CRM, but unless an agent is explicitly prompted to combine them, the user sees disjointed answers. Structured orchestration frameworks (like LangGraph) explicitly model an “aggregator” step for this purpose, whereas a pure group chat does not. The result is a poor user experience: users must manually piece together information from multiple responses instead of receiving a unified insight.
* **Error Handling:** The Group Chat pattern lacks robust error-catching or recovery flows. If an agent encounters an exception (e.g. an API call fails or returns invalid data), the default behavior is often to output the error as a message or fall silent. There is no built-in retry logic or alternate path on failure. In practice, a failed API call can halt the workflow without resolution, forcing the user to intervene. By contrast, more structured agent frameworks incorporate error-handling strategies – for instance, a LangGraph workflow can branch to a recovery node on error, and OpenAI’s Swarm will catch a function error and allow the conversation to continue rather than hard-stop. The absence of such mechanisms in the current system means **failed operations lead to dead-ends or confusing outputs**, degrading user trust. For example, if a “Create order” agent crashes due to a missing field, the system may not gracefully ask for clarification or try a fallback, leaving the user with an unhelpful error message.

**Impact on Business Operations:** These limitations directly affect performance and user experience across Cegid’s use cases:

* In **ERP queries** (e.g. financial reports or inventory checks), multi-step requests (such as “fetch Q4 revenue and then forecast next quarter”) may not execute correctly – the agents might only do the first part or produce separate answers. The user experiences frustration when a single query requires multiple clarifications. High-value workflows like financial close processes involve many dependent steps, which the current system struggles to maintain in order.
* In **CRM updates** (e.g. finding a client record then updating their status), a free-form agent conversation might retrieve the data but fail to execute the update if not explicitly prompted in sequence. Errors (like insufficient permissions or validation failures) are not handled gracefully, so the user might get a generic failure and need to retry manually. This slows down CRM tasks and reduces confidence in the assistant.
* In **supply chain workflows**, such as checking stock across warehouses and placing a replenishment order, the need to combine data from multiple APIs is common. Without proper aggregation, the assistant might return inventory levels from one system and separately confirm order placement, but not correlate them (e.g. ensure that the lowest-stock item was ordered). Any API hiccup (network timeout or data format issue) could terminate the entire chain. This leads to inefficiencies and potential business risks (like missing a restock alert) because the AI agent cannot reliably carry out the full workflow end-to-end.

The table below summarizes these limitations and their impact:

| **Challenge** | **Issue with Group Chat Pattern** | **Impact on Business Use Cases** |
| --- | --- | --- |
| **Multi-step workflows** | Lacks an explicit step-by-step execution plan, relying on unguided agent dialog. Complex, branched tasks are hard to manage in a free-form chat. | Multi-step processes (ERP approvals, multi-stage supply orders) may execute incorrectly or not at all, requiring repeated user intervention and reducing reliability. |
| **Data aggregation across APIs** | No built-in mechanism to combine outputs from multiple agents/APIs; each agent’s result remains separate by default. The conversation doesn’t enforce a final aggregation step. | Users receive fragmented information (e.g. separate ERP and CRM outputs) instead of a unified answer. This increases cognitive load and chance of error when making decisions based on partial data. |
| **Error handling** | Errors/exceptions are not caught systematically. A failed API call or tool use can halt the conversation without recovery. No automatic retries or fallback logic in free chat. | Failed operations lead to abrupt stops or cryptic error messages. Users must manually diagnose or retry, hurting productivity and trust. In critical workflows (finance, procurement), lack of error recovery can cause process delays or data inconsistency. |

Overall, the current AG2 Group Chat approach, while flexible for simple Q&A, is **ill-suited for complex, multi-step business operations**. It negatively impacts performance (through inefficiencies and repeated attempts) and user experience (through incomplete or confusing interactions), highlighting the need for a more structured and intelligent orchestration solution.

## **2. Framing the Problem in the Broader Context**

The challenges Cegid faces are symptomatic of the broader evolution in **LLMs and multi-agent systems** as we push toward more **agentic AI** and, ultimately, AGI. Early LLM applications were often single-turn or single-agent, handling one query at a time. However, enterprise tasks (like those in finance, HR, and retail) demand *composed actions* – chains of reasoning, tool usage, and decisions. The industry recognizes that enabling AI to handle such complexity autonomously is a key stepping stone toward more general intelligence ([OpenAI’s Level-3: Agents | by Nicholas Domnisch | EE Solutions](https://medium.com/ee-solutions/openai-level-3-agents-agi-275e1615d5dc#:~:text=their%20intended%20use%20case%20and,%E2%80%9D%20%28Analytics%20India%20Magazine)).

**Multi-Agent Systems as a Path to AGI:** Leading AI researchers view multi-agent collaboration and orchestration as crucial for improving reasoning and achieving more generalized problem-solving. OpenAI, for example, has explicitly stated that *“multi-agent is a path to even better AI reasoning.”* ([OpenAI’s Level-3: Agents | by Nicholas Domnisch | EE Solutions](https://medium.com/ee-solutions/openai-level-3-agents-agi-275e1615d5dc#:~:text=their%20intended%20use%20case%20and,%E2%80%9D%20%28Analytics%20India%20Magazine)) In OpenAI’s framework for progress toward AGI, a critical milestone (“Level 3”) involves agents that can autonomously plan and execute tasks with minimal human guidance ([OpenAI’s Level-3: Agents | by Nicholas Domnisch | EE Solutions](https://medium.com/ee-solutions/openai-level-3-agents-agi-275e1615d5dc#:~:text=The%20rapid%20advancement%20of%20artificial,WallStreetPit)) ([OpenAI’s Level-3: Agents | by Nicholas Domnisch | EE Solutions](https://medium.com/ee-solutions/openai-level-3-agents-agi-275e1615d5dc#:~:text=OpenAI%E2%80%99s%20AI%20development%20framework%20consists,%E2%80%9D%20%28%2017%20Analytics%20India)). This implies moving beyond single-step Q&A to systems of agents that **coordinate on complex objectives**. The difficulties Cegid’s platform encounters (maintaining context over multiple steps, integrating diverse information sources, handling unexpected outcomes) are exactly the challenges being addressed to reach that next level of AI capability.

**Major AI Players and Complexity Challenges:** Top AI companies are rapidly innovating to handle multi-step reasoning and action:

* **OpenAI:** Recently introduced the *O1* model series (and subsequent *O3*), which incorporate *agentic reasoning loops*. Unlike traditional LLMs that produce an answer in one go, O1 can iteratively break down a problem, step through a solution, and even call tools/functions in the process. This “step-by-step” internal workflow significantly improves its ability to tackle complex prompts. OpenAI’s O1 Pro mode uses these loops to work through prompts systematically, and the upcoming O3 extends these capabilities further with impressive results in complex task benchmarks. In parallel, OpenAI has experimented with a multi-agent orchestration framework called **Swarm**, which introduces structured concepts like *routines* and *handoffs* to coordinate agents (ensuring one agent’s output feeds correctly to the next) ([Introducing OpenAI's Swarm: A New Open-Source Multi-Agent ...](https://www.kommunicate.io/blog/openai-swarm/#:~:text=Introducing%20OpenAI%27s%20Swarm%3A%20A%20New,for%20orchestrating%20multiple%20AI%20agents)) ([Building Intelligent Multi-Agent Systems with OpenAI's Swarm - DZone](https://dzone.com/articles/building-intelligent-multi-agent-systems-with-swarm#:~:text=DZone%20dzone,agentic%20AI%20in%20this%20guide)). These developments indicate OpenAI’s multi-pronged approach: more *intelligent models (O1/O3)* and better *orchestration frameworks (Swarm)* to manage complexity.
* **Anthropic:** Anthropic’s Claude models have been designed with a focus on safety and long-form reasoning. The latest Claude versions (e.g. Claude 3.x “Sonnet”) boast very large context windows and strong performance on reasoning tasks ([Claude's agentic future and the current state of the frontier models](https://www.interconnects.ai/p/claudes-agency#:~:text=Claude%27s%20agentic%20future%20and%20the,use%20chat%20models)). Moreover, Anthropic has endowed Claude with the ability to perform actions – for example, Claude can invoke a “computer use” mode to execute commands or use tools as part of its reasoning. This is analogous to an agent calling an API or running a script when needed. By enabling tool use and extended “thinking,” Anthropic is addressing the need for multi-step task handling within a single agent. Claude’s development shows the trend of **integrating reasoning and action** directly into the model’s capability, which is another approach to solving complex workflows.
* **Others (Progression Toward AGI):** Google’s upcoming Gemini model (e.g. *Gemini 2.0* with a “Flash Thinking” mode) and startups like Adept and DeepMind are also pursuing the vision of models that can plan and execute multi-step strategies. Techniques such as **Chain-of-Thought prompting**, **ReAct (Reason+Act)**, and **Self-Refinement** emerged in research and have been quickly adopted into these production models ([LLMs Evolve with Agentic Workflows, Enabling Autonomous Reasoning and Collaboration](https://www.deeplearning.ai/the-batch/llms-evolve-with-agentic-workflows-enabling-autonomous-reasoning-and-collaboration/#:~:text=,model%20to%20act%2C%20evaluate%2C%20reflect)). The idea is to get LLMs not just to answer, but to **think aloud, make decisions, and correct themselves**, much like an agent in a workflow. This broader context underscores that Cegid’s platform is part of a general movement: turning powerful LLMs into **reliable cognitive agents** that can manage elaborate tasks.

In summary, the difficulties with the current system (like multi-step orchestration and error handling) are *not unique*. They reflect the frontier problems AI companies are actively solving on the road to AGI. Both improved model capabilities (as seen in O1, O3, Claude, etc.) and improved multi-agent frameworks (AutoGen/AG2, LangChain’s LangGraph, OpenAI’s Swarm, etc.) are converging to handle such complexity. Cegid’s platform stands to benefit directly from these advances by adopting the latest techniques in multi-agent coordination and reasoning.

## **3. State-of-the-Art Research and Technologies**

To design a solution, it’s important to leverage the **state-of-the-art in multi-agent reasoning, LLM capabilities, and planning/orchestration tools**. Below we review key developments:

### **Advanced Multi-Agent Reasoning and Coordination**

**AG2 Reasoning Agent (Advanced Reasoner):** The AG2 framework itself has evolved beyond the basic Group Chat. One of the flagship advancements is the **ReasoningAgent** introduced in AG2 v0.5+, which serves as an “agentic brain” for complex tasks. The ReasoningAgent implements a *Tree-of-Thoughts* approach: it **explores multiple reasoning paths in parallel and uses a grader (evaluation) agent to select the best outcome** ([ReasoningAgent - Tree of Thoughts with Beam Search in AG2 - DEV Community](https://dev.to/ag2ai/reasoningagent-tree-of-thoughts-with-beam-search-in-ag2-3ki1#:~:text=,for%20DPO%20and%20PPO%20training)). In practice, this means instead of answering directly, the agent can consider several possible solution steps, simulate outcomes, prune bad paths, and refine its approach – much like a human brainstorming and evaluating different strategies. The underlying algorithms include **Beam Search** (keeping a fixed number of best hypotheses at each step) ([ReasoningAgent - Tree of Thoughts with Beam Search in AG2 - DEV Community](https://dev.to/ag2ai/reasoningagent-tree-of-thoughts-with-beam-search-in-ag2-3ki1#:~:text=,for%20DPO%20and%20PPO%20training)), and options for **Monte Carlo Tree Search (MCTS)** and **LATS (Language Agent Tree Search)** for even more powerful search-based reasoning ([Qingyun Wu's Post - LinkedIn](https://www.linkedin.com/posts/qingyun-wu-183019a6_here-comes-ag2-reasoning-agent-httpslnkdin-activity-7290048934922645505-dk7q#:~:text=Qingyun%20Wu%27s%20Post%20,agent%20and%20Deepseek%20reasoning)) ([Language Agent Tree Search - AG2](https://ag2.airt.ai/0.8.4/docs/use-cases/notebooks/notebooks/lats_search#:~:text=Language%20Agent%20Tree%20Search%20,time%20compute)). For example, with MCTS or LATS, the agent can incorporate reflection and backtracking: it generates possible next actions, reflects on their results (using a self-critique or external feedback), and backpropagates scores to decide the next exploration step ([Language Agent Tree Search - AG2](https://ag2.airt.ai/0.8.4/docs/use-cases/notebooks/notebooks/lats_search#:~:text=Language%20Agent%20Tree%20Search%20,time%20compute)) ([Language Agent Tree Search - AG2](https://ag2.airt.ai/0.8.4/docs/use-cases/notebooks/notebooks/lats_search#:~:text=2,trajectories%20based%20on%20the%20outcomes)). This capability is particularly relevant for multi-step business queries – the ReasoningAgent can essentially **plan out a sequence of actions** by internally simulating different multi-step solutions (e.g. “First do X, then Y, if failure do Z”) before committing to a final answer. It’s a significant improvement in reasoning depth and reliability, and is seen as an *“agentic alternative”* to relying purely on a single large model like OpenAI’s O1.

**Central Orchestrator Agents:** Beyond individual reasoning, the trend is toward specialized coordinator agents that manage other agents. AG2 has introduced concepts like the **CaptainAgent** (sometimes called a “Manager” or “Planner” agent) which is essentially a top-level orchestrator ([Reasoner Agents | Driving controlled autonomy... | by RAJIB DEB | Medium](https://medium.com/@rajib76.gcp/from-rule-aware-to-context-aware-systems-88c747986d88#:~:text=AG2%E2%80%99s%20Captain%20Agent%3A%20A%20Practical,Example)). This agent can interpret a high-level user goal and **automatically break it into subtasks, then delegate those to expert agents**, overseeing the whole process ([Reasoner Agents | Driving controlled autonomy... | by RAJIB DEB | Medium](https://medium.com/@rajib76.gcp/from-rule-aware-to-context-aware-systems-88c747986d88#:~:text=An%20instantiation%20of%20such%20agent,complex%20systems%2C%20the%20Captain%20Agent)). It acts as a project manager for the AI team, ensuring each step is completed and integrating the results. This aligns with Marvin Minsky’s “Society of Mind” theory – individual agents handle small tasks, while a **central reasoning agent (the “builder” or captain) organizes them into a coherent strategy** ([Reasoner Agents | Driving controlled autonomy... | by RAJIB DEB | Medium](https://medium.com/@rajib76.gcp/from-rule-aware-to-context-aware-systems-88c747986d88#:~:text=Now%20that%20we%E2%80%99ve%20established%20the,concept%20of%20the%20Builder%20Agent)) ([Reasoner Agents | Driving controlled autonomy... | by RAJIB DEB | Medium](https://medium.com/@rajib76.gcp/from-rule-aware-to-context-aware-systems-88c747986d88#:~:text=Key%20Advantages%20of%20Reasoner%20Agents%3A)). The advantage is twofold: (1) **Structured Planning** – the system does not rely on emergent conversation alone, but on an explicit plan devised by the reasoning agent; (2) **Controlled Autonomy** – constraints and safety checks can be applied at the planner level (e.g. the CaptainAgent can decide not to execute a risky action, or to involve a human if needed) ([Reasoner Agents | Driving controlled autonomy... | by RAJIB DEB | Medium](https://medium.com/@rajib76.gcp/from-rule-aware-to-context-aware-systems-88c747986d88#:~:text=Instead%20of%20dispersing%20constraints%20across,enhances%20system%20transparency%20and%20safety)) ([Reasoner Agents | Driving controlled autonomy... | by RAJIB DEB | Medium](https://medium.com/@rajib76.gcp/from-rule-aware-to-context-aware-systems-88c747986d88#:~:text=%2A%20can%20interpret%20high,ensure%20safety%20and%20goal%20alignment)). State-of-art frameworks often implement this pattern under various names (e.g. a “Supervisor” agent in LangGraph, or a “Crew” lead in CrewAI). The key point is that **multi-agent systems benefit from a hierarchical structure**: a reasoner/planner agent at the top and specialist agents for each tool or domain beneath.

**Coordination Strategies:** Research has also produced techniques for agent coordination such as agent voting or debate (having multiple agents propose solutions and reconcile) and self-consistency checks. For example, an agent might produce multiple answers and then another agent (or the same agent in a different mode) evaluates which answer is best – this was inspired by *self-consistency decoding* in LLMs ([LLMs Evolve with Agentic Workflows, Enabling Autonomous Reasoning and Collaboration](https://www.deeplearning.ai/the-batch/llms-evolve-with-agentic-workflows-enabling-autonomous-reasoning-and-collaboration/#:~:text=,model%20to%20act%2C%20evaluate%2C%20reflect)). AG2’s ReasoningAgent actually uses a form of this: the “thinker” and “grader” roles within it can be seen as two agents (generator and evaluator) working together ([ReasoningAgent - Tree of Thoughts with Beam Search in AG2 - DEV Community](https://dev.to/ag2ai/reasoningagent-tree-of-thoughts-with-beam-search-in-ag2-3ki1#:~:text=,for%20DPO%20and%20PPO%20training)). Such patterns echo human collaborative problem-solving and are becoming standard in advanced agent systems to improve result quality and reliability.

### 

### **State-of-the-Art LLMs for Reasoning and Planning**

**OpenAI O1 and O3 Models:** OpenAI’s *O1* (introduced in late 2024) is a cutting-edge large language model specifically geared towards agentic behavior. It features an internal mechanism to **work through tasks step by step**, essentially performing chain-of-thought internally. O1 can plan intermediate steps, call functions (via OpenAI’s function calling interface), and adjust its approach based on tool outputs – all within one prompt/response cycle. This greatly improves its ability to handle multi-step queries without external orchestration. O1 was followed by *O3*, an even more advanced model with extended context and refined reasoning loops. Early reports indicate O3 can solve more complex problems and make fewer reasoning errors, effectively extending what O1 started. These models are relevant to Cegid’s platform because they reduce the “prompt engineering” burden – an O1-powered agent can on its own break down a command like *“Find the top 5 customers by revenue and update their status to VIP in CRM”* into a sequence of actions, whereas earlier models (GPT-3/4) might require explicit step-by-step prompting or an external planner. However, using O1/O3 alone in a multi-agent system still benefits from oversight (to handle inter-agent coordination), but they bring much stronger built-in reasoning to each agent.

**Anthropic Claude:** Anthropic’s latest *Claude* models (e.g. Claude 3.x series) are also at the forefront of reasoning. Claude is known for a very large context window (allowing it to consider long dialogues or many documents at once) and for a training focus on harmlessness and helpfulness. In terms of planning, Claude has demonstrated the ability to produce detailed step-by-step plans when asked, and with the newest updates, it can even execute those plans by controlling tools. As noted earlier, Claude 3.5 “Sonnet” introduced a feature where the model can output special commands to, say, browse the web or use a terminal. In a multi-agent context, this means a single Claude agent can act somewhat autonomously, deciding when to call an API tool. Claude’s strength is often in **robust natural language understanding** and maintaining coherent narratives (useful for summarizing aggregated data or explaining results), which could complement OpenAI models.

**Other Notable Models:** The landscape also includes **DeepSeek’s R1** and **Google’s Gemini**, which have been mentioned as incorporating similar agentic reasoning capabilities. These models are built to handle *decision-making and tool use* as part of their core functionality. For instance, Google’s Gemini (successor to PaLM/LaMDA family) is expected to integrate strong planning abilities possibly influenced by DeepMind’s work in reinforcement learning. While Cegid’s focus will likely be on OpenAI or Anthropic due to Azure integration, it’s good to note that the capabilities (iterative reasoning, large context, tool integration) are becoming standard in top-tier LLMs by 2025.

In summary, the state-of-art LLMs today are far more capable for reasoning and planning than previous generations. Models like O1/O3 and Claude can serve as the “brains” of agents, able to interpret complex instructions, break them into steps, and even execute or call tools in mid-thought. Leveraging these in Cegid’s platform (via Azure OpenAI service or Anthropic’s API) will significantly boost the agents’ ability to handle the multi-step workflows and decision-heavy tasks required.

### **Planning Methodologies and Memory Models**

**Advanced Planning Techniques:** Research into LLM prompting and agent behavior has yielded several techniques that are now applied in practice. **Chain-of-Thought (CoT)** prompting gets models to explain or enumerate steps, which can be used to force a multi-step reasoning process. **Self-Consistency** involves generating multiple reasoning chains and choosing the most common result, reducing randomness in complex reasoning ([LLMs Evolve with Agentic Workflows, Enabling Autonomous Reasoning and Collaboration](https://www.deeplearning.ai/the-batch/llms-evolve-with-agentic-workflows-enabling-autonomous-reasoning-and-collaboration/#:~:text=,model%20to%20act%2C%20evaluate%2C%20reflect)). The **ReAct** framework (Reason+Act) interleaves thought (CoT) and actions (tool use) – an approach that is foundational to many agent systems now ([LLMs Evolve with Agentic Workflows, Enabling Autonomous Reasoning and Collaboration](https://www.deeplearning.ai/the-batch/llms-evolve-with-agentic-workflows-enabling-autonomous-reasoning-and-collaboration/#:~:text=,model%20to%20act%2C%20evaluate%2C%20reflect)). Building on ReAct, methods like **Reflexion** and **Self-Refine** let an agent critique its own output and try again, which is invaluable for error recovery. These methodologies have influenced frameworks like AG2’s ReasoningAgent and planning agents. For example, the ReasoningAgent’s loop of generate → evaluate → iterate is essentially an automated CoT with self-reflection at scale. For Cegid, this means our system can incorporate these methods either implicitly (by using models/agents that implement them) or explicitly (by prompt engineering: e.g., asking an agent to provide a plan first, then execute). Advanced planning algorithms (like the aforementioned **MCTS and LATS**) go even further by introducing search algorithms to the reasoning process, which is useful when there is a way to evaluate partial progress toward a goal. If certain business tasks have a well-defined success check (for instance, “order placed successfully” as an end condition), an agent could use a search-based planning to explore different sequences of API calls until it finds one that achieves the goal.

**Agent Memory Models:** As multi-agent interactions grow complex, maintaining state and memory is critical. There are two levels of memory to consider: **short-term conversational memory** and **long-term knowledge or context memory**.

* In AG2 (and Autogen before it), the default short-term memory is the message history between agents. Each agent sees the past few messages as context (subject to token limits). This works for short dialogues, but multi-step workflows can produce a lot of context (especially if agents are verbose) and risk exceeding model context windows. The Group Chat pattern had each agent remember the conversation implicitly, which is fragile for long sequences.
* Newer frameworks and patterns have introduced **explicit state tracking**. For example, LangChain’s LangGraph uses a *global shared state* (a structured object) that all agents read from and write to, rather than relying purely on chat history ([Technical Comparison of AutoGen, CrewAI, LangGraph, and OpenAI Swarm | by Omar Santos | Feb, 2025 | Artificial Intelligence in Plain English](https://ai.plainenglish.io/technical-comparison-of-autogen-crewai-langgraph-and-openai-swarm-1e4e9571d725#:~:text=LangGraph%20maintains%20a%20global%20shared,for%20memory%20and%20context%20passing)). This state can persist beyond a single session and can be stored (in a database or memory store) and reloaded, enabling long-running or paused workflows ([Technical Comparison of AutoGen, CrewAI, LangGraph, and OpenAI Swarm | by Omar Santos | Feb, 2025 | Artificial Intelligence in Plain English](https://ai.plainenglish.io/technical-comparison-of-autogen-crewai-langgraph-and-openai-swarm-1e4e9571d725#:~:text=LangGraph%20maintains%20a%20global%20shared,for%20memory%20and%20context%20passing)). Similarly, some agent systems use a blackboard memory model where intermediate results are posted to a common knowledge store that others can access.
* In a business context, we may also use **vector databases or knowledge bases** as memory. For example, if an agent needs to recall details from earlier in the day or from a prior related task, we can embed that information and store it in a vector DB (like Azure Cognitive Search or Redis Vector store) keyed to the conversation or task. Agents can then query this store when needed (this would be a tool call like search\_memory). This approach extends memory beyond the immediate context window of the LLM.
* Long-term memory is especially important for recurring processes (say, a monthly financial close workflow) – the agent could retrieve what it did last time or any notes left from failures/successes (enabling a form of continuous improvement or learning over time, albeit manually programmed since current agents don’t truly learn on their own between sessions without fine-tuning).

State-of-the-art systems combine these: **ephemeral working memory (for the active task)** and **persistent memory (for cross-task context)**. For Cegid, implementing a shared memory context (so that, for instance, an ERP agent’s result is stored and accessible to a CRM agent in the next step without relying on possibly truncated chat history) will be crucial. This can be done within AG2 by utilizing its message store or by custom integration of a memory tool. The goal is to ensure no loss of information as the agents execute multi-step plans.

### **Multi-Agent Orchestration Frameworks**

Several orchestration frameworks have emerged (besides AG2 itself) that address the exact problems we identified. It’s useful to briefly note their approaches, as they inform our recommended architecture:

* **Microsoft Autogen / AG2:** Autogen (which evolved into AG2) pioneered the idea of conversational agents that could **asynchronously message each other to collaborate**. It is highly flexible (agents are just ChatGPT-like entities conversing), which made it easy to set up systems like a “user proxy + assistant” for code generation or question answering. The introduction of patterns like GroupChat, Sequential, and Swarm in AG2 is an attempt to provide more structure when needed. Notably, **Swarm** in AG2 and OpenAI’s framework is a pattern where agents have a defined handoff sequence or can operate in parallel on sub-tasks, then converge. The latest AG2 also has features for *nested chats* (agents spawning sub-conversations to solve subproblems) and conditional logic in conversations. The strength of AG2 is its **extensibility** – one can incorporate new agent types (like the ReasoningAgent, CaptainAgent) and tools easily, and it’s designed to be integrated in Python applications (including Azure deployments). However, as we saw, using it effectively means choosing the right pattern and perhaps augmenting the default ones for more complex flows.
* **CrewAI:** CrewAI takes a pipeline approach – you define a sequence of agents (a “crew”) each with a task, and the system runs them one after the other under a controlling process. This is more rigid than a free-form chat, but it ensures clarity: each step is executed in order and you know which agent is responsible for what. CrewAI logs the entire sequence, which aids in debugging and auditing. It also supports conditional branches, but these must be scripted. The benefit is **robustness**: if something goes wrong at step 3, you know exactly where it happened and can retry or fix that step. Error handling in CrewAI can be managed by the host program (since you get structured outputs from each agent). The trade-off is less dynamic flexibility – you have to design the workflow upfront. For Cegid, adopting a CrewAI-like approach would mean identifying common multi-step flows (e.g. “Query ERP -> Process Data -> Update CRM -> Summarize to user” as a fixed pipeline). This could greatly improve reliability for known use cases, though it can be less adaptable to on-the-fly tasks.
* **LangChain’s LangGraph:** LangGraph is very relevant because it targets exactly the need for **explicit multi-step orchestration**. It uses a directed graph (DAG) where nodes are tasks or agent invocations and edges define the flow (including branches, loops). Essentially, it’s a workflow engine for LLM agents. Each node can have conditions, so you can handle errors by routing to an alternate path (for example, if API call node returns an error, follow an edge to an error-handling node). LangGraph also emphasizes **visibility** – you can visualize the graph, which is useful in enterprise for explaining how the AI works and auditing its decisions. The downside is complexity in setup – one must design and maintain the graph. For Cegid’s needs, if we foresee a standard set of complex workflows (which might be the case for many business processes), using a graph approach ensures nothing is left to chance. We could incorporate some ideas of LangGraph into AG2 by having a coordinator agent that effectively implements a graph logic (noting that AG2 allows custom orchestration code).
* **OpenAI Swarm and Others:** OpenAI’s experimental Swarm (which AG2 even has an implementation for) marries the conversation style with structured transitions. It introduces the concept of **routines** (predefined sequences of steps that can be invoked as one unit) and **handoffs** (explicitly passing control from one agent to another) ([How OpenAI's SWARM Simplifies Multi-Agent Systems - YouTube](https://www.youtube.com/watch?v=LBih635lzps#:~:text=How%20OpenAI%27s%20SWARM%20Simplifies%20Multi,Agent%20Systems%2006%3A05%20Example)) ([Building Intelligent Multi-Agent Systems with OpenAI's Swarm - DZone](https://dzone.com/articles/building-intelligent-multi-agent-systems-with-swarm#:~:text=DZone%20dzone,agentic%20AI%20in%20this%20guide)). It’s like a middle ground: not as rigid as a fixed pipeline, but more controlled than free chat. Swarm ensures that only one agent speaks at a time and that there’s logic determining who that should be next. It also keeps the conversation going by catching errors (as noted, it injects an error message rather than stopping, allowing subsequent agents to react to it). This concept is quite aligned with what we want: **structured turn-taking with error resilience**.

Each of these state-of-art tools confirms that **solving multi-step, multi-agent interactions is a known challenge** and that solutions exist by injecting more structure and planning into the process. We will draw from these ideas – the parallel reasoning of AG2’s ReasoningAgent, the structured sequencing of CrewAI/LangGraph, and the dynamic handoff control of Swarm – to design the optimal solution for Cegid.

## **4. Recommended Approach and Architecture**

Based on the above, the best solution is a **hybrid architecture** that combines AG2’s advanced agent capabilities (ReasoningAgent and others) with a clear orchestration and memory strategy. The goal is to preserve flexibility (the system can handle a variety of requests) while adding the structure needed for reliability. We propose the following high-level architecture:

* **Central Reasoning Orchestrator:** At the core, introduce a *Reasoning Orchestrator Agent* (using AG2’s **ReasoningAgent** or the related CaptainAgent pattern) as the first point of contact for user requests. This agent’s role is to **interpret the user’s query and devise a plan** to fulfill it. Upon receiving a user prompt, the orchestrator will break it down into actionable steps – essentially performing a multi-step reasoning internally or via a planning routine. For complex queries, it can engage its tree-of-thought reasoning (leveraging O1/O3’s capabilities or using beam search/MCTS via the AG2 ReasoningAgent) to decide on the sequence of operations. This agent acts as the “brain” of the system, coordinating everything else ([Reasoner Agents | Driving controlled autonomy... | by RAJIB DEB | Medium](https://medium.com/@rajib76.gcp/from-rule-aware-to-context-aware-systems-88c747986d88#:~:text=Now%20that%20we%E2%80%99ve%20established%20the,concept%20of%20the%20Builder%20Agent)) ([Reasoner Agents | Driving controlled autonomy... | by RAJIB DEB | Medium](https://medium.com/@rajib76.gcp/from-rule-aware-to-context-aware-systems-88c747986d88#:~:text=An%20instantiation%20of%20such%20agent,complex%20systems%2C%20the%20Captain%20Agent)).
* **Specialized Sub-Agents (Expert Agents):** Under the orchestrator, maintain multiple **specialist agents**, each responsible for interacting with a specific business API or performing a specific type of task. For example, an “ERP Agent” knows how to query the ERP system (and is equipped with the necessary API credentials and tool functions), a “CRM Agent” handles CRM queries/updates, a “SupplyChain Agent” for supply chain system, etc. These would be implemented as AG2 ConversableAgents (or AssistantAgents) with appropriate system prompts and tools. They don’t need to be highly intelligent in planning; they just execute a single-step instruction: e.g. “Fetch customer X data from CRM” or “Update inventory for product Y to quantity Z”. Each such agent can use an LLM (possibly a lighter model or the same O1 model with a relevant prompt) to format the API call and handle the response. By modularizing by API, we encapsulate the details of each system’s interaction.
* **Tool/Function Integration:** Many API calls can be done via function calling. We will integrate the business APIs as **tools** that agents can invoke. In AG2, we can use register\_function to allow an agent to call a Python function that wraps the actual API call. For instance, the ERP Agent might have a query\_erp(query\_params) function registered as a tool. This way, when the agent’s LLM decides to get data, it can directly call the function (via function call mechanism or via the executor agent pattern in AG2) instead of outputting text for another agent to read. This reduces the number of back-and-forth messages, improving latency. It also provides more deterministic operation – the agent either gets the data or an error/exception that we can catch.
* **Orchestration Logic (Swarm Pattern):** Instead of relying on the ad-hoc GroupChat manager to choose speakers, implement a more deterministic orchestration flow. There are two possible approaches, and we can even combine them:  
  1. **Plan-based Sequence:** The Orchestrator Agent, upon devising a plan, explicitly calls each relevant sub-agent in turn. For example, it might send a message to the ERP Agent like “Execute Step 1: get X from ERP”. Once it receives the result, it sends the next instruction to, say, the CRM Agent: “Execute Step 2: use X to update CRM record Y”. This is a controlled sequence, similar to CrewAI’s pipeline but directed at runtime by the Orchestrator’s plan. In AG2, this can be done by having the orchestrator agent call initiate\_chat with each agent in the desired order, or by using the **Swarm** feature where we predefine a chain of hand-offs (Agent A -> B -> C).
  2. **AG2 CaptainAgent AutoBuild:** Optionally, we could leverage AG2’s CaptainAgent to automate some of this. CaptainAgent can *auto-build* a team of expert agents and initiate a group chat among them to solve the subtasks ([CaptainAgent - AG2](https://ag2.airt.ai/0.8.1/docs/user-guide/reference-agents/captainagent/#:~:text=CaptainAgent%20is%20an%20agent%20enhanced,down%20and%20solve%20complex%20tasks)) ([CaptainAgent - AG2](https://ag2.airt.ai/0.8.1/docs/user-guide/reference-agents/captainagent/#:~:text=AutoBuild%20can%20initiate%20a%20group,with%20tools%20for%20advanced%20coding)). For instance, if the CaptainAgent knows it needs an ERP expert and a CRM expert, it can spin those up (from a library of agent definitions) and have them collaborate. However, given our need for deterministic control, we would configure the CaptainAgent to still follow a structured approach (it can be guided by specifying the subtasks, or we ensure the auto-generated chat follows the plan).
* The recommended approach is a **guided Swarm**: basically implementing a directed workflow but within the AG2 framework. We might script the core logic: *Orchestrator receives user message -> Orchestrator plans steps -> Orchestrator issues each step to the appropriate agent and waits for result -> Orchestrator aggregates results and responds to user*. This way, we maintain **full clarity on the workflow** while still using agents for the heavy lifting at each step.
* **Data Aggregation and Synthesis:** The Orchestrator (Reasoning Agent) will also serve as the **aggregator** of results. After collecting data from various sub-agents/APIs, it uses the LLM to merge these into the final answer for the user. Because the orchestrator has a memory of all prior steps (we will ensure it retains the outputs from sub-agents), it can compose an answer that, for example, combines ERP financial data and CRM client data in one coherent summary. If needed, a separate "Summarizer Agent" could be used for particularly complex aggregation (for instance, if a lot of textual data needs summarizing, a specialized agent could do that), but likely the orchestrator itself (using a powerful model like O3) can handle summarization and reasoning about the combined data.
* **Error Handling and Recovery:** Incorporate explicit error-handling at each step of the orchestration. If a sub-agent returns an error or fails (e.g., the API tool raises an exception, or the agent returns an “I couldn’t get the data” message), the orchestrator agent will **not simply give up**. We will program the orchestrator to detect this (it can check for error flags or content in responses). Upon error, it can decide: either retry the step (maybe with a modified query or after a short delay), or take an alternate action. For example, if an ERP query fails due to a timeout, the orchestrator can try once more, and if it still fails, maybe ask a different agent (if available) or return a message to the user like, “ERP system is not responding, would you like to proceed with partial data?”. We could also integrate a **Human-in-the-Loop** at error points: AG2 allows an agent to request human input. In an enterprise workflow, that could mean if an automatic method fails, the system can prompt, “I encountered an issue updating CRM. Should I try again or skip this step?” – letting the user decide. The architecture thus includes **branching for errors**: each planned step has a primary path and a fallback path. This draws from LangGraph’s approach of encoding error transitions and from Swarm’s approach of continuing the conversation with an error message rather than stopping. Technically, we can implement this by wrapping each tool call in try/except in the Python function (ensuring it returns a structured error), and have the orchestrator’s logic interpret that. The Orchestrator (ReasoningAgent) itself, being an LLM, can even be prompted in its system prompt to handle messages that indicate an error in a special way (“If any step returns an error, do X…”).
* **Memory and Context Sharing:** To ensure continuity, we will implement a **shared memory context** for the duration of each user query (which might involve multiple back-and-forth turns with the user, if it’s a dialogue). All agents involved in a single workflow will have access to a common context object that includes: the original user request, relevant extracted parameters (like customer ID, etc.), and the results of each completed step so far. In AG2, since all agents in a group chat or swarm share a ChatManager, we can store data in that manager or a blackboard accessible to agents. Alternatively, the Orchestrator can explicitly include necessary context in its messages to sub-agents (e.g., “Using data X from the previous step, do Y”). For longer-term memory across separate conversations or for large reference data, we integrate an external memory store (e.g., an Azure Cognitive Search index for company knowledge, or a database of prior queries). This isn’t strictly required for the immediate improvement, but it sets the stage for scaling the system to handle multi-turn dialogues that reference older information. Essentially, by not relying solely on the prompt history, we avoid context loss issues and can handle more complex workflows that generate a lot of intermediate data.
* **Use of Latest LLMs:** We will utilize **OpenAI’s O1 or O3 models via Azure OpenAI** as the backbone for the reasoning orchestrator agent (and possibly for all agents, though smaller ones could use GPT-4 or 3.5 if needed for cost). O3, being the most advanced, could serve as the Orchestrator’s model since it will perform the most demanding task (planning and synthesis), whereas O1 might suffice for the domain agents. Azure’s hosted version of these models ensures data compliance and leverages Azure’s infrastructure for scaling. The orchestrator agent’s LLM config might be something like {"model": "openai-o3", "temperature": 0} for deterministic planning, while a CRM agent might use {"model": "openai-o1", "temperature": 0.3} if a bit of creativity is needed in forming queries. We will also keep the option open to integrate **Anthropic Claude** (if available in Azure or via API) for tasks requiring extremely large context (e.g., reading lengthy financial reports as part of a query). The architecture is model-agnostic at the high level – meaning we can swap out the LLM in an agent as needed – which provides flexibility to take advantage of future model improvements.
* **Azure Compatibility and Deployment:** All components will be deployed in an Azure-hosted environment. The architecture can be realized as, for example, an Azure Function or set of microservices: one service might handle the Orchestrator Agent logic, calling out to sub-agent functions or containers for specific API interactions. Because AG2 is a Python framework, we can run it within Azure Container Instances or AKS (Azure Kubernetes Service) for scalability. Each user session (or each query) could spawn an orchestrator agent instance that in turn initializes the needed sub-agents. With Azure’s robust support for Python and machine learning, integration of OpenAI’s models (via Azure OpenAI) and any Azure-specific services (like Azure Service Bus if we want to queue tasks, or Azure Monitoring for logging) will be straightforward. The recommended architecture is fully **cloud-native**: stateless where possible (each request self-contained with its plan and memory), which means we can scale out to handle many simultaneous conversations by just adding more containers. Latency will be managed by Azure’s low-latency networking to the OpenAI endpoint and by minimizing the number of LLM calls (the orchestrator will gather info in parallel when possible, as described below).
* **User Experience Improvements:** From the user’s perspective, this architecture will still present as a chat interface, but they will notice that even complex requests are handled more gracefully. The system might respond with a single rich answer that encapsulates multiple steps (“I have updated the CRM and ERP as requested. Customer X is now VIP in CRM, and the ERP notes an updated status as well. Inventory is adjusted.”), rather than a back-and-forth of partial answers. If additional clarification is needed (e.g. the user didn’t specify which region’s ERP to query), the orchestrator can ask a focused follow-up question, because it’s aware of what’s missing (this is effectively a sub-step it plans: “If parameter Y is missing, ask user for it”). This leads to more conversational, intelligent interactions where the AI proactively manages the workflow.

In summary, the recommended architecture is **AG2 with a Reasoning/Orchestrator Agent on top, specialized action agents below, and a well-defined flow control**. It takes inspiration from the CaptainAgent (for auto-managing agent teams) and from structured frameworks like LangGraph (for predefined flows), blending them into a solution tailored for Cegid’s APIs and Azure environment. The design emphasizes *scalability* (by modular agents and stateless operation per request), *low latency* (by using direct function calls for tools and parallelizing where possible), *robustness* (with error handling paths), and *maintainability* (each API integration is an isolated agent or tool, which is easy to update as APIs change, and the overall logic is centralized in the orchestrator plan, easy to modify for new requirements).

The key components of the architecture are outlined in the table below:

| **Component** | **Implementation** | **Role in System** |
| --- | --- | --- |
| **Orchestrator Agent** (Planner) | AG2 ReasoningAgent (or CaptainAgent) powered by OpenAI O1/O3 model. | Entry point for user queries. Interprets user intent, devises a multi-step plan (using advanced reasoning if needed), and orchestrates the execution by delegating to specialist agents. Also aggregates results and formulates the final answer. ([Reasoner Agents |
| **Specialist Agents** (Executors) | AG2 ConversableAgents or tool-enabled agents; one per domain (ERP, CRM, HR, etc.), using appropriate LLM (O1 or smaller) and with relevant API tool functions registered. | Carry out specific tasks as instructed by the Orchestrator. For example, fetch data from an ERP endpoint, update a CRM record, perform a calculation, etc. Each agent is confined to its domain and tools for safety (e.g., the ERP agent cannot call CRM directly). Returns results or error messages back to Orchestrator. |
| **Business API Tools** | Python functions wrapping REST API calls, registered via AG2’s tool interface (with an executor agent if needed for function execution). | Provide the actual integration with external systems. Agents invoke these tools to read/write data. This layer abstracts authentication and API details away from the LLM (the LLM just decides to call the tool). Ensures that API calls are executed efficiently (and can be monitored/logged). |
| **Memory Store / Context** | In-memory context object for each session; optional persistence via Azure Cognitive Search or database for long-term memory. | Maintains shared state: user query details, intermediate results, and any contextual data. Accessible by Orchestrator and passed to agents as needed (or agents are prompted with relevant context). If using persistent memory, can store conversation summaries or results for future retrieval (e.g., recall what happened in a previous related task). ([Technical Comparison of AutoGen, CrewAI, LangGraph, and OpenAI Swarm |
| **Orchestration Logic** | AG2 Swarm or custom control flow coded in the Orchestrator (could use Python code within the orchestrator agent, or the CaptainAgent’s auto-orchestration features). | Determines the execution order of tasks. Enforces that, for example, Step 1 (ERP query) completes before Step 2 (CRM update) begins. Handles conditional branches – if a result meets certain criteria, maybe skip a step or perform an extra step. Essentially, this is the “workflow engine” driving the agents, implemented using AG2’s conversation control or by the orchestrator’s plan. |
| **Error Handling Mechanism** | Try/except around tool calls; Orchestrator prompt instructions for errors; fallback agents or human hand-off. | Detects and handles errors at runtime. Ensures the workflow doesn’t silently fail. Could involve retry logic, alternative flows, or asking the user for guidance. All agents are instructed to format error outputs in a consistent way (so that the Orchestrator can recognize them and respond accordingly). |
| **LLM Models** | Azure OpenAI service for O1/O3; possibly Anthropic Claude via API; GPT-4 as fallback for certain tasks if needed. | Powers the intelligence of the agents. The Orchestrator uses the most advanced model available (for reasoning and language quality). Sub-agents can use lighter models to save costs (since their tasks are narrower). All models are hosted in Azure environment ensuring compliance with data security. |
| **User Interface** | Existing chat frontend, with slight modifications to handle multi-step visualization (optional). | From the user’s perspective, still a chat. We might enhance the UI to show that the system is “working on multiple steps” (e.g., a spinner with a message “Gathering data from ERP and CRM...”) to set expectations for longer queries. Once the orchestrator compiles the final answer, it’s displayed normally. |

This architecture directly addresses the earlier limitations: the Orchestrator ensures multi-step flows are planned and executed in order; the specialist agents and shared memory enable data from multiple APIs to be combined; and the explicit error handling paths make the system resilient to failures. Importantly, it leverages **immediately actionable components** – AG2 (which Cegid already uses) can be extended with these patterns, and Azure-hosted models/APIs are utilized, so no exotic infrastructure is needed.

## **5. Detailed Implementation Guide**

Implementing this optimized architecture will involve several steps. Below is a step-by-step roadmap, with each step focusing on a part of the system. This guide assumes an Azure environment, using Azure OpenAI for models and Python for the AG2 framework and API integrations.

**1. Setup AG2 and Configure LLM Access:**

* **Install/Update AG2:** Ensure you have the latest version of AG2 (formerly Autogen) installed in your Azure environment. This can be done via pip: pip install -U ag2[openai,captainagent]. Including the *captainagent* extra will pull in any additional dependencies for advanced agents ([CaptainAgent - AG2](https://ag2.airt.ai/0.8.1/docs/user-guide/reference-agents/captainagent/#:~:text=Install%20AG2%20with%20the%20CaptainAgent,extra)). Verify the version to have features like ReasoningAgent (v0.5+) and CaptainAgent (v0.4+).
* **Azure OpenAI Integration:** Request access to OpenAI’s O1 and O3 models in Azure (if not already enabled). Set up your Azure OpenAI resource and deployment for these models. In your code, configure AG2’s OAI\_CONFIG\_LIST (as shown in AG2 docs) with the Azure OpenAI endpoints and keys for the models. For example, map the model names openai-o1 and openai-o3 to the Azure deployment names. Test a simple prompt completion with O1/O3 to ensure connectivity. If O1/O3 are not available, GPT-4 can be used initially while awaiting access – structure the code to allow switching the model name via configuration.
* **Anthropic (Optional):** If you plan to use Claude for certain agents, set up the Anthropic API key and client. AG2’s LLMConfig can also handle other providers ([CaptainAgent - AG2](https://ag2.airt.ai/0.8.1/docs/user-guide/reference-agents/captainagent/#:~:text=,13)). This is optional and can be deferred; the system can function with OpenAI models alone.

**2. Implement the Orchestrator (Reasoning Agent):**

**Create Orchestrator Agent Class:** Using AG2, instantiate a custom agent for the orchestrator. You can either use the ReasoningAgent directly or subclass a ConversableAgent with a specific prompting strategy. AG2 might provide a CaptainAgent class which auto-handles some planning ([Reasoner Agents | Driving controlled autonomy... | by RAJIB DEB | Medium](https://medium.com/@rajib76.gcp/from-rule-aware-to-context-aware-systems-88c747986d88#:~:text=AG2%E2%80%99s%20Captain%20Agent%3A%20A%20Practical,Example)), but to maintain control, you can start with a manual approach. For instance:  
  
 from autogen import ReasoningAgent, LLMConfig

llm\_config = LLMConfig(model="openai-o3", temperature=0) # deterministic, powerful model

orchestrator = ReasoningAgent("orchestrator", llm\_config=llm\_config,

system\_message="You are a planning agent that will receive user goals and devise step-by-step solutions by coordinating other agents. You never directly call tools, but instruct other agents to do so. Respond with clear plans and integrate results.")

* This system message sets the stage. The ReasoningAgent in AG2 can be initialized with internal parameters for beam search or MCTS if desired. For example, if using beam search: ReasoningAgent(..., search\_algorithm="beam", beam\_width=3), or if using MCTS: ... search\_algorithm="mcts", simulations=20 (hypothetical API based on AG2 docs). These parameters control how exhaustive the reasoning is ([ReasoningAgent - Tree of Thoughts with Beam Search in AG2 - DEV Community](https://dev.to/ag2ai/reasoningagent-tree-of-thoughts-with-beam-search-in-ag2-3ki1#:~:text=,for%20DPO%20and%20PPO%20training)) ([Language Agent Tree Search - AG2](https://ag2.airt.ai/0.8.4/docs/use-cases/notebooks/notebooks/lats_search#:~:text=Language%20Agent%20Tree%20Search%20,time%20compute)). Start with moderate settings (e.g., beam search with width 3) so as not to incur too much latency.
* **Define Planning Behavior:** We might need to guide the orchestrator to output actionable plans. One approach is to use a two-phase prompting: first get the agent to outline a plan, then execute it. For example, when the user message comes in, you could prompt the orchestrator: “Outline the steps to achieve this:” and get a numbered list. Then feed that plan (either into the same agent or just parse it with code) to actually dispatch tasks. Alternatively, leverage the ReasoningAgent’s internal ability to think – it might not require an external prompt to plan, as it will internally generate thoughts. In AG2’s design, ReasoningAgent can maintain a thought tree and final answer ([ReasoningAgent - Tree of Thoughts with Beam Search in AG2 - DEV Community](https://dev.to/ag2ai/reasoningagent-tree-of-thoughts-with-beam-search-in-ag2-3ki1#:~:text=,for%20DPO%20and%20PPO%20training)). However, here we want the process, not just the final answer. It may be simpler to have the orchestrator agent operate in a loop: manually parse the plan and call sub-agents step by step.
* **Handling Orchestrator Output:** Plan that the orchestrator’s “reply” will not go directly to the user, but rather trigger the execution of steps. For example, if the orchestrator says (in pseudo-code) “Step 1: ask ERP agent for X; Step 2: given X, ask CRM agent to do Y; Step 3: summarize result”, you will capture that output in your driver code (or via a callback in AG2). AG2’s Conversation objects or the ChatManager could be used to intercept the messages. Another method: use CaptainAgent’s AutoBuild feature which might do this automatically – it would read the user goal and spin up sub-agents and manage a group chat ([CaptainAgent - AG2](https://ag2.airt.ai/0.8.1/docs/user-guide/reference-agents/captainagent/#:~:text=CaptainAgent%20is%20an%20agent%20enhanced,down%20and%20solve%20complex%20tasks)). If using CaptainAgent, read AG2 documentation on how to supply an agent library and tool library so that it knows what experts (sub-agents) are available and what they can do ([CaptainAgent - AG2](https://ag2.airt.ai/0.8.1/docs/user-guide/reference-agents/captainagent/#:~:text=break%20down%20and%20solve%20complex,tasks)) ([CaptainAgent - AG2](https://ag2.airt.ai/0.8.1/docs/user-guide/reference-agents/captainagent/#:~:text=We%20begin%20with%20demonstrating%20how,refer%20to%20docs%20on%20nested_mode)). This can save effort, but make sure it doesn’t produce an unpredictable chain – you might need to constrain it with parameters like number of subtasks or provide exemplars.

**3. Develop Specialist Agents and Tools:**

* **Identify Required Agents:** List the external systems the AI needs to interact with: e.g., ERP (finance system), CRM, HR database, supply chain management, etc. For each, create an agent. These can be simple AssistantAgent or ConversableAgent instances with a system prompt that defines their role (e.g., “You are an ERP assistant. You receive requests to get data from or send data to the ERP. You have a function query\_erp available to you to retrieve information.”). The agent’s model can be smaller – GPT-4 or O1 should suffice, and you could even try GPT-3.5 if cost is a concern, since these agents mostly transform a request into an API call. Keep their temperature low for consistency.

**Implement API Wrapper Functions:** For each system, implement Python functions that perform the needed API calls (using the system’s SDK or REST endpoints). For instance:  
  
 def get\_customer\_info(customer\_id: str) -> str:

# call CRM API

response = crm\_client.get\_customer(customer\_id)

return response.data\_json() # or format as needed

def update\_order\_status(order\_id: str, status: str) -> str:

# call ERP API to update order status

result = erp\_client.update\_order(order\_id, status)

return "success" if result.ok else f"error: {result.error}"

* Focus on the key actions. These will be registered as tools. Use strong typing and docstrings to help the LLM understand them (AG2 uses type annotations and descriptions as part of function tool registration). For example, annotate customer\_id: Annotated[str, "Customer ID in CRM database"] – this will help the agent choose the right function.
* **Register Tools with Agents:** Using AG2’s register\_function, register each function with the appropriate agent and an executor (if using the two-agent tool invocation pattern). The executor agent can be a simple ConversableAgent with human\_input\_mode="NEVER" that just runs the function when instructed. Essentially, this means the specialist agent can “decide” to call a function, and the executor will carry it out and return the result. AG2 handles this interaction under the hood after registration. Confirm that each agent’s tool works by testing in isolation (e.g., have the ERP agent take a sample request in a small script and see if it calls the function and returns output). Ensure that errors in the API call are caught and returned as part of the function result (or raise exceptions that propagate to the agent as error messages).
* **Limit Agent Scope:** As a safety/robustness measure, configure each specialist agent to only have access to its relevant tools. Do not give the CRM agent the ERP functions and vice versa. This prevents confusion and also acts as a permission layer. In AG2, since you manually register functions per agent, this is naturally enforced. You can also use the system prompt to warn: “If asked to do anything outside CRM, respond with an error or deferral.” This ensures, for example, that if the orchestrator accidentally asked the CRM agent to do an ERP task, the CRM agent would refuse, prompting the orchestrator to correct the plan.

**4. Implement Orchestration Logic:** With agents and tools ready, implement the logic that ties them together in a workflow. There are a couple of ways:

* **Within the Orchestrator Agent (via prompts):** You could feed the orchestrator agent a prompt like: “User asked: {user\_request}. Plan steps and execute: (1) formulate ERP query, (2) await result, (3) formulate CRM update, (4) await result, (5) provide final answer.” However, having the orchestrator literally execute steps via natural language might be unwieldy. Instead, orchestrator can output something like “Plan: [Step1: ...; Step2: ...]” which your code interprets. It might be easier to not fully automate this part with the LLM, but to combine LLM planning with Python control.
* **Python-Orchestrated Flow (Recommended):** Use a driver function (could be the Azure Function handler or main loop in your app) to manage the conversation. For each user query:  
  1. Send the user query to the Orchestrator agent and get its response (which may be a plan or some analysis of the task). If using ReasoningAgent’s tree search, you might directly get a final reasoning output that includes sub-tasks.
  2. Parse the orchestrator’s output to identify the sub-tasks. You might implement a simple parser for a list like “Step 1: Get customer data; Step 2: Update order; Step 3: ...”. The orchestrator could also output JSON or some structured format if prompted to (AG2 agents can output structured data if asked, which might simplify parsing).

For each identified step, determine which agent/tool is needed and formulate a message to that agent. Essentially, you translate the orchestrator’s high-level plan into a concrete instruction. For example, plan says “get customer data from CRM”, you then call something like:  
  
 crm\_agent\_response = crm\_agent.initiate\_chat(executor\_agent, message="Retrieve customer info for ID 12345", max\_turns=1)

* 1. (If the CRM agent is conversable and has the tool registered, this should trigger the tool and return the result in one turn). AG2’s initiate\_chat will manage the internal communication between the CRM agent and its tool executor, returning the final output.
  2. Check the response. If crm\_agent\_response contains an error (either a structured error string or some indicator), handle it (see step 5 below for error strategy). If it’s successful, store the result in a context dictionary (e.g., context['customer\_info'] = crm\_agent\_response).
  3. Move to the next step. Some steps might need information from previous ones (the orchestrator’s plan or the code controlling it should account for that). For instance, if Step 2 is “Use the customer’s status from Step 1 to update ERP”, ensure you pass the customer\_info or relevant part of it to the ERP agent. This could mean extracting a field from the CRM output and including it in the ERP agent’s instruction.
  4. Continue until all steps are executed.
  5. Finally, take all gathered results and feed them back into the Orchestrator (or a final formatting agent) to produce the user-facing answer. The orchestrator agent can be re-prompted with something like: “You have obtained the following information: ... Now provide a concise answer to the user.” Since the orchestrator has memory of the whole interaction (if done in one session), it might already be able to do this in its initial reasoning output. Alternatively, have a dedicated summarizer: e.g., use the orchestrator agent itself or another agent to collate the results into a final message. The output should be checked for correctness and completeness (possibly the orchestrator can double-check: e.g., ensure all requested actions were performed).
* **Parallelism:** Where possible, execute independent steps in parallel to improve latency. For example, if the plan involves querying *both* an ERP and a HR system and those are independent, you don’t need to do one then the other. You could fire off both agent calls asynchronously (if your environment supports async calls or threads). Python’s asyncio could be used with AG2 if it allows async operation, or simply spawn two threads to call the agents and wait for both to finish. AG2’s asynchronous support (if using v0.4+ with an event loop) might help here. This parallelism should be used carefully (only when steps truly don’t depend on each other) but can greatly speed up responses for data aggregation queries.
* **Swarm / CaptainAgent Option:** As an alternative to manual coding, you can try using AG2’s Swarm orchestration. Define a Swarm with a specific sequence and transitions. For instance, define that the conversation starts with Orchestrator, then goes to ERP agent, then back to Orchestrator, then CRM agent, then Orchestrator, etc. AG2 might allow defining a custom speaker selection or using the speaker\_selection\_method="manual" where you programmatically choose the next speaker. The CaptainAgent’s AutoBuild could also potentially take a user query and automatically do similar, but it might not guarantee the exact sequence you want. If time permits, experiment with CaptainAgent: give it a user goal and see how it breaks down the task with auto-generated expert agents ([CaptainAgent - AG2](https://ag2.airt.ai/0.8.1/docs/user-guide/reference-agents/captainagent/#:~:text=CaptainAgent%20is%20an%20agent%20enhanced,down%20and%20solve%20complex%20tasks)) ([CaptainAgent - AG2](https://ag2.airt.ai/0.8.1/docs/user-guide/reference-agents/captainagent/#:~:text=We%20begin%20with%20demonstrating%20how,refer%20to%20docs%20on%20nested_mode)). It could handle a lot of orchestration for you, but ensure it aligns with business logic (you might need to feed it a library of agents with descriptions of their capabilities, so it knows to use, say, a “CRMAgent” for CRM-related subtasks).

**5. Integrate Error Handling and Human-in-the-Loop:**

* **Error Capture:** Ensure every call to a specialist agent/tool is wrapped in error capture. If the tool function throws an exception (say the API endpoint is unreachable), catch it and have the function return a special error value (or let the executor agent handle it). For example, have get\_customer\_info return "ERROR: <message>" string or a dictionary {"error": "..."}. Standardize this format so the orchestrator can check if "ERROR" in result or similar.

**Retry Logic:** Decide on retry policy. For transient errors (timeout, rate limit), the orchestrator can automatically retry the tool call X seconds later up to N times. Use exponential backoff to be polite to APIs. This can be coded easily around the agent call. For example:  
  
 for attempt in range(3):

result = erp\_agent.initiate\_chat(... )

if "ERROR: timeout" in result:

time.sleep(2 \*\* attempt)

continue

else:

break

* If after retries it still fails, then escalate the error to either an alternate path or user notification.
* **Alternate Path:** Think if there’s any alternate way to get data if one path fails. For instance, if the primary ERP API is down, maybe there is a data warehouse or cache to query as a backup. If so, implement a secondary tool and have the orchestrator try that. If an update fails, maybe just log it and inform the user “I couldn’t update X, but I did the rest.” This depends on business priorities – in some cases partial failure is acceptable with notice; in others, a whole transaction should roll back if one part fails (in which case the orchestrator might even issue compensating actions, like if CRM update failed, perhaps undo the ERP change if that makes sense – this is like a transaction rollback, which is advanced but possibly important for things like financial postings). Implement such logic carefully if needed, likely at the orchestrator level where it has context of all steps.
* **Human Confirmation:** For high-stakes actions or unresolved errors, integrate a human-in-loop. AG2 allows an agent to pause and wait for human input. One design: if the orchestrator encounters an unexpected situation (e.g., the user’s request is to delete a lot of records, or an error occurred that it cannot fix), it can be set to flag for human review. In a chat interface, this could be a prompt back to the user like, “The action X failed due to Y. Would you like to retry or cancel?” The user can then respond with instructions. Technically, orchestrator agent could yield control back to the user proxy agent at that point. Since the user is the ultimate authority, this ensures no endless loops or uncontrolled failures. Mark such points clearly in the code (perhaps orchestrator sends a message that is surfaced to user, then stops).
* **Testing Error Scenarios:** Simulate errors during development. For example, force the ERP tool to throw an exception and see how the orchestrator responds. Ensure the system doesn’t crash and that the user gets a meaningful message. Also test what happens if an agent returns something unexpected (like the ERP agent’s LLM mis-formats a response). Ideally, by using function calling, we minimize the latter – the tool call returns data and the agent likely just passes it through. Still, implement sanity checks on outputs where possible.

**6. Optimization for Performance (Real-time vs Batch):**

* **Real-time Interaction:** For interactive chat usage (a user waiting on a response), optimize for minimal latency:  
  + **Limit Reasoning Depth:** While the ReasoningAgent’s thorough search is valuable, in a synchronous setting keep beam search width or MCTS simulations to a moderate number. A beam width of 3 or 5 is probably fine; using 10 or more could slow things significantly for a single query. O3 is powerful enough that it might not need extremely wide search to come up with a good plan. You can also restrict the number of thought iterations. AG2 likely has parameters like max\_tree\_depth or max\_turns for the reasoning cycle.
  + **Parallelize I/O:** As noted, fire off independent API calls in parallel. I/O latency (waiting on external APIs) can dominate time. If the orchestrator can send out two queries at once and then wait for both, it will cut the wall-clock time. This might complicate the orchestration code slightly (need to gather responses), but Python’s async or threading can handle it.
  + **Streaming Responses:** If using OpenAI models that support streaming, you could stream the final answer to the user for a better UX (i.e., the user sees the answer being typed out). This doesn’t make the answer arrive faster in full, but improves perceived latency. AG2 likely supports streaming outputs from agents; if not, you can directly use OpenAI’s SDK for the final answer generation step to stream tokens to the frontend.
  + **Compute Scaling:** Deploy the solution with enough computational resources. Ensure the container/Function has adequate CPU and memory, and possibly GPU if using very large models (though Azure OpenAI is an API call, so our side just needs to handle JSON; no heavy local compute). Scale out by configuring Azure Functions to handle concurrent executions or AKS to autoscale pods based on queue length. This way, multiple users can get responses without queuing behind each other.
* **Batch and Offline Mode:** Some tasks might be suitable to run in a batch mode (e.g., a scheduled report or an end-of-day summary). For those:  
  + **Use Full Reasoning Power:** Since no user is waiting, we can afford to let the agent think more. For instance, allow the ReasoningAgent a larger search (beam width 10, or MCTS with more simulations) to potentially find more optimal solutions or verify results. It could even use *self-consistency*: run the whole plan multiple times and see if results agree, which might catch intermittent issues.
  + **Aggregate Larger Data:** In batch mode, if the task involves large datasets, the agent might need to retrieve many records and process them. Ensure your tools can handle pagination or chunking of data (for example, if pulling thousands of records from ERP, do it in chunks). The orchestrator might loop over multiple sub-queries and then compile a big result. Memory management becomes crucial here: perhaps store intermediate large data in a temporary storage (file or database) rather than keeping everything in RAM or in the prompt. The agent can be given summaries or references to the stored data instead of raw large data (to avoid hitting token limits).
  + **Logging and Monitoring:** For batch runs, log all steps and outcomes to an Azure Log Analytics or Application Insights, so that if something fails in the middle at 3 AM, you can troubleshoot. The orchestrator should perhaps send an email or alert if a batch job fails and cannot ask a user for help (since no user is watching).
  + **Reuse Results:** If appropriate, cache results from expensive operations. For example, if two batch tasks in the same night both need a list of active customers, fetch once and reuse. This can be managed by the orchestrator if it recognizes that it already got some data. A simple cache in memory or a persisted cache (Redis) keyed by query could help. Just be careful with cache staleness in real-time queries (cache mainly for within one run or a short time window).
* **User Experience Tuning:** For longer workflows, consider sending the user intermediate updates via the chat. For instance, if a query will take 30 seconds because it’s aggregating data from 5 systems, the orchestrator might send a message: “Sure, I’m working on that request. Gathering data from various systems, this may take a half minute…” and then the final answer. This way the user isn’t staring at silence. This can be achieved by programming the orchestrator to send an immediate acknowledgment (or using a separate system agent to notify) before doing the heavy work.

**7. Testing and Iteration:**

* **Unit Test Agents:** Individually test each specialist agent with sample prompts to ensure they call the correct tool and handle responses. E.g., test the CRM agent by giving it a prompt to get a known customer and see if it returns the correct info. Test edge cases (customer not found – does it return a “not found” message gracefully?). Similarly, test orchestrator logic by simulating a simple scenario: e.g., ask it “Find customer 123 and update their status” and see if it produces a reasonable plan. You can stub the actual API calls in testing to focus on logic.
* **Integration Test Workflow:** Run end-to-end on a few representative scenarios: a straightforward one (single API query), a multi-step one (ERP->CRM update), a multi-API data aggregation query, and an error scenario (maybe disable one API to see recovery). Check that the final output to user is correct, and that all intermediate steps happened as expected (look at logs or printouts from each step).
* **Performance Test:** Simulate multiple concurrent users or repeated queries to ensure the system scales. Use Azure’s tools or a simple locust/gatling test to send, say, 10 simultaneous chat requests. Monitor that each is handled in a reasonable time. Check Azure monitoring for any bottlenecks (like if waiting on the LLM is the longest pole, or an external API). This will inform if you need to e.g. use asynchronous calls more or scale out.
* **Security and Compliance Check:** Since this deals with business data, ensure that all secrets (API keys, database credentials) are stored securely (Azure Key Vault) and not hard-coded. The system should log activity for audit (who asked for what data and was it retrieved – this might be required for compliance in finance/HR). AG2 likely doesn’t log externally by itself, so add logging statements in your orchestration code around each major action. Also, verify that the LLMs and prompts do not inadvertently leak sensitive data between conversations (AG2 isolates convos by default, but if using any persistent memory, ensure data is scoped per user appropriately).

By following this roadmap, Cegid’s AI platform will evolve from a basic group chat to a **sophisticated multi-agent orchestrator**. The solution remains **immediately actionable**: it uses current, supported technology (AG2 framework on Python, Azure-hosted APIs and models) and requires no fundamental research breakthroughs to implement – it’s about engineering the system with best practices gleaned from state-of-the-art. Once implemented, Cegid can expect the platform to handle complex workflows that span multiple enterprise systems with greater accuracy and robustness, providing end-users a seamless and powerful AI assistant for their business operations. All steps above can be executed and tested incrementally, ensuring a smooth transition to the new architecture with minimal disruption.