AI agents and Optimization problems

Introduction

Everyone loves camping! However, knowing what to pack and balancing necessity with weight limitations can be challenging. Imagine you will be heading to the mountains next weekend, and you can only carry **10kg** of gear and have a **150€ budget** to purchase everything you need.

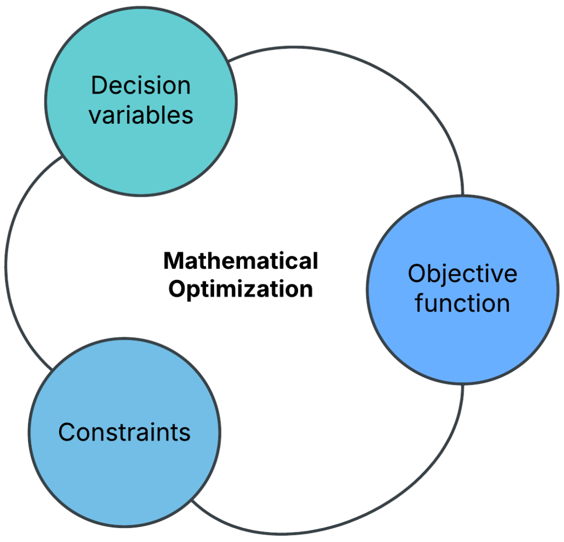
You decide on a **tent (60€)** and a high-quality **sleeping bag (40€)**, leaving you with **50€** and a **6.5kg** carrying capacity. Now comes the tricky part: what additional items should you choose to **maximize value** for your trip while staying within your weight and budget limits?

This is a classic **optimization problem** – finding the best solution given a set of constraints.

What is Optimization?

Optimization problems involve finding the best possible solution by maximizing or minimizing an objective while adhering to certain constraints. It consists of three main components:

- **Objective function**: What we are trying to maximize or minimize (e.g., maximizing value in our camping gear selection, minimizing cost, etc.)
- **Decision variables**: The choices we can make (e.g., which items to choose from)
- **Constraints**: The rules we must follow (e.g., weight and budget limits)



Optimization problems appear across various industries, from supply chain management to finance, resource allocation and scheduling. However, a major challenge in optimization is translating real-world problems into mathematical formulations, a task typically handled by expert mathematicians and operations researchers.

Given this complexity, could AI agents help automate this process? Our research project explored how LLMs (Large Language Models) and AI agents could assist in extracting and formulating the correct mathematical model for optimization problems.

AI Agents: What Are They?

According to an article published by IBM in July 2024, AI agents are defined as:

> *A system or program that is capable of autonomously performing tasks on behalf of a user or another system by designing its workflow and utilizing available tools.*

In simpler terms, AI agents act as smart assistants that can process information, make decisions and execute tasks. Unlike a traditional LLM, an AI agent doesn’t just generate text, it can also interact with tools, retrieve data from the web, run code, and more.

Why AI Agents for Optimization?

One of the key challenges in AI-assisted optimization is accurately extracting constraints, decision variables, and objectives from natural language descriptions. AI agents offer several advantages:

- **Structured Workflow**: Unlike a single LLM, each agent handles specific tasks that contribute to solving a larger problem.
- **Prompt Engineering**: Each agent can be prompted differently based on its specific task, allowing for greater flexibility, control and improved accuracy.
- **Tool Integration & Function Calling**: Agents can use external tools like solvers, databases and APIs, while intelligently choosing when and which Python functions to call, optimizing the problem-solving process.

Given these, AI agents seemed the perfect candidates for our research into automated formulation of optimization problems.

The Architecture

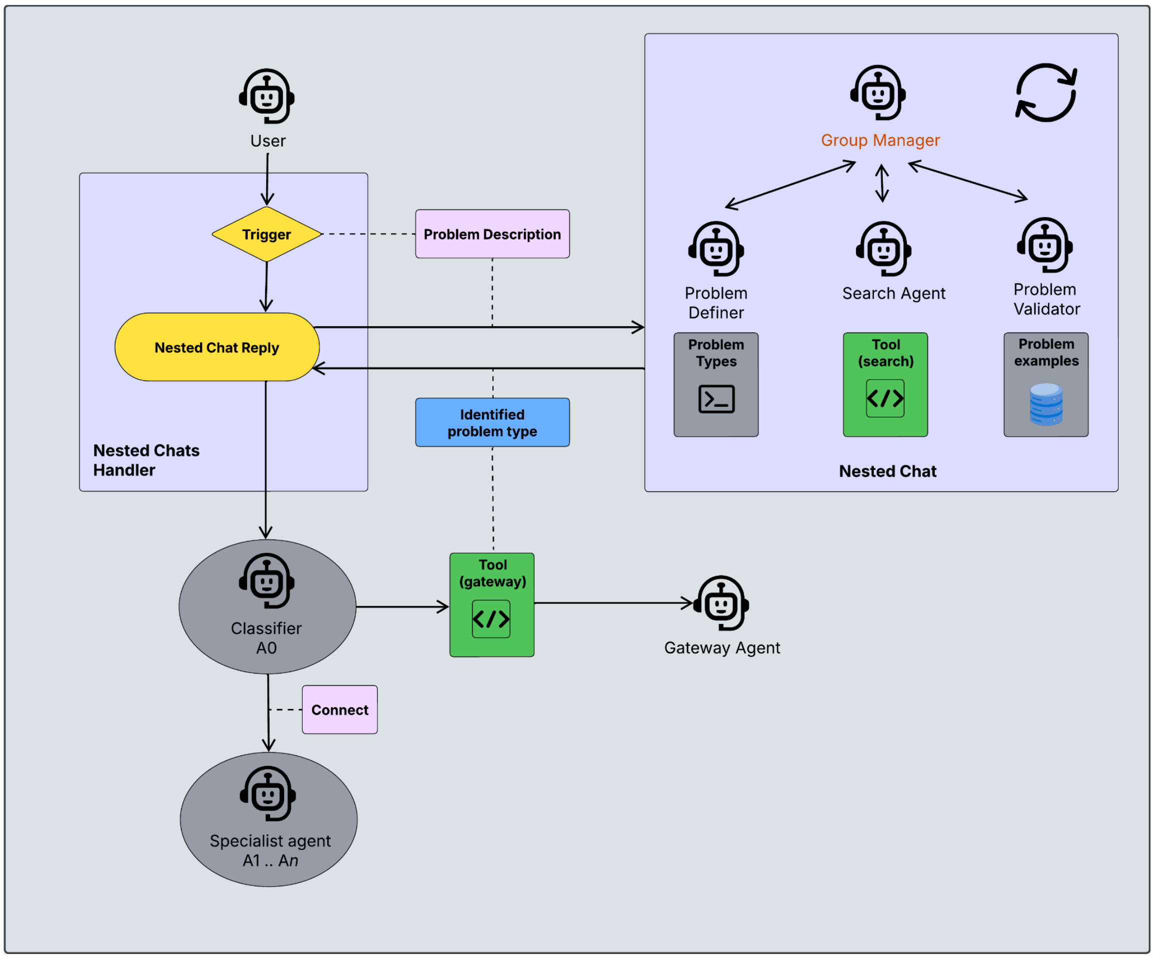
To process a problem description and generate a structured output, we designed a modular pipeline consisting of three key components:

- **Classifier**: Identifies the type of optimization problem.
- **Formulation**: Extracts the objective function, decision variables and constraints from the input.
- **Output**: Based on the user's preference, returns either the mathematical formulation of the problem or a Python script that solves it, with an option to utilize the HiGHS solver library.

Each module operates as its own pipeline, where multiple AI agents, each with a well-defined role, collaborate to complete the overall task.

Classifier: Identifying the Problem Type

The Classifier module is responsible for analyzing the problem description and categorizing it into a predefined problem type. Once the type is identified, the classifier connects with the appropriate formulation agents to proceed with the next steps.



How it works

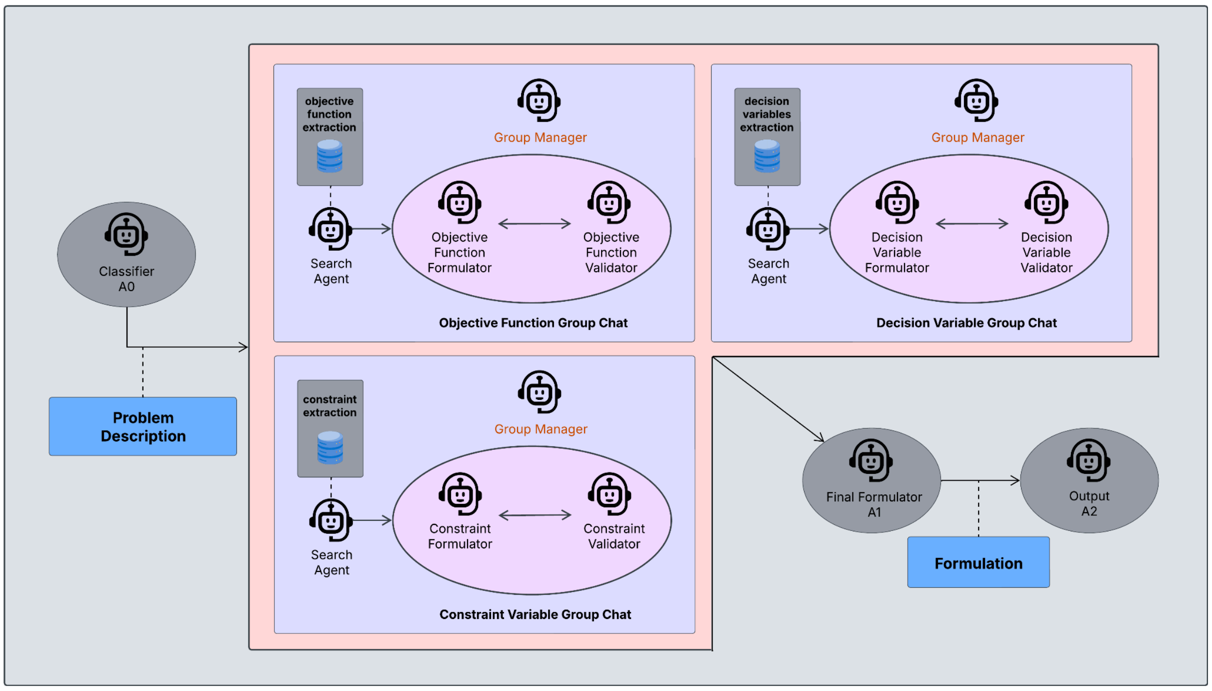
1. User Input – The user submits the problem description through a proxy agent, which acts as an interface between the user and the system.

2. Nested Group Chat – A structured conversation takes place among specialized agents. The **Group Manager** oversees the flow of the conversation, ensuring complete control over the order in which agents speak. The **Problem Definer** identifies the problem type using predefined knowledge embedded in its system prompt, while the **Search Agent** (RAG agent) retrieves relevant examples to support the problem type identification. The **Problem Validator** critiques the problem type, comparing it against the examples to ensure accuracy.

3. Classification & Routing – Once a consensus is reached on the problem type, the **Classifier (A0)** processes the result and forwards it to the **Gateway Agent**, which then connects to the appropriate formulation agents to solve the problem based on its type.

Formulation: Deriving the Mathematical Model

The Formulation module is the core of our architecture as it plays a crucial role in deriving the correct formulation that can be solved. To build this formulation, we focus on separately extracting the three key components that make up an optimization problem: the objective function, the decision variables, and the constraints – concepts that will likely be familiar to the reader by now.


Classifier A0
Problem Description
objective function extraction
Group Manager
Search Agent
Objective Function Formulator
Objective Function Validator
Objective Function Group Chat
decision variables extraction
Group Manager
Search Agent
Decision Variable Formulator
Decision Variable Validator
Decision Variable Group Chat
constraint extraction
Group Manager
Search Agent
Constraint Formulator
Constraint Validator
Constraint Variable Group Chat
Final Formulator A1
Output A2
Formulation

How it works

1. Group Chats – Extracting each of the three components (objective function, decision variables and constraints) follows a consistent pattern:

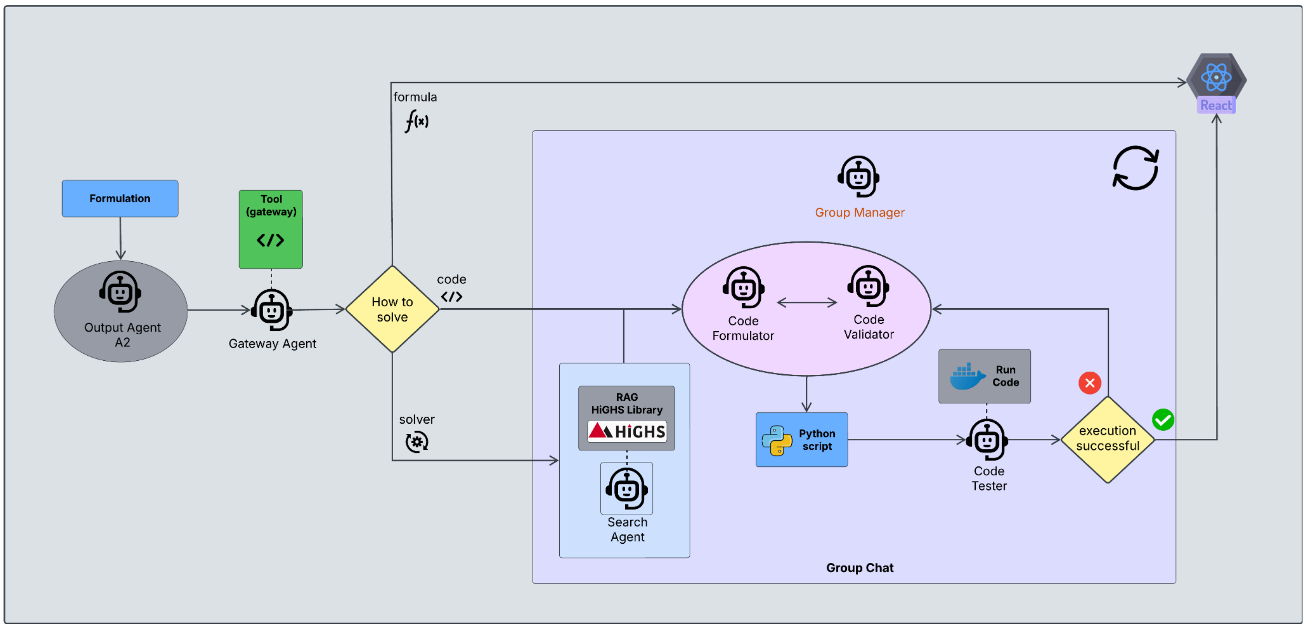
- First, the **Search Agent** (RAG agent) retrieves relevant information from its knowledge base to assist the formulation agents.
- Then, the **Formulator Agent** extracts the respective component from the problem description.
- Finally, the **Validator Agent** reviews the extracted component, ensuring its accuracy and completeness, and provides feedback to the Formulator where necessary.
- This process, overseen by the **Group Manager**, continues iteratively until the Formulator and Validator agree on the final extraction.

2. Final Formulation – Once all three components are finalized, they are sent to the **Final Formulator**, who combines them into a well-structured JSON formulation, ready for solving.

Output: Transforming the Formulation into the Desired Format

The Output module is responsible for converting the extracted formulation into the user's preferred format. Users can select from the following output options before submitting the problem description:

- **Raw Formulation**: The formulation is returned as is, without any modifications.
- **Code**: A Python script is generated containing the necessary code to solve the formulation.
- **Solver**: A Python script is generated that integrates the HiGHS solver library for solving the formulation.



How it works

1. Receiving User Input – The **Output Agent** receives the final formulation from the Formulation module and the user's selected output format.

2. Gateway Processing – The received formulation is processed by the **Gateway Agent**, which connects to the appropriate agents based on the user's choice:

- **Raw Formulation**: The formulation is directly returned to the user without modifications.
- **Code and Solver**: A **Group Chat** is initiated for further processing.

3. Group Chats – The **Formulator Agent** creates the output script and the **Validator Agent** reviews it for code quality and completeness. The **Executor Agent** then tests it in a secure Docker environment and produces a PASS/FAIL report. This process continues iteratively until the script passes the validation.

- **Solver** – Includes a **Search Agent** (RAG agent) that retrieves relevant information from the HiGHS library documentation to integrate the solver functionality into the script.

Technology Stack

To build our agentic optimization system we used the following technologies:

- **AI Foundry** – Microsoft's model management platform
- **Autogen 2.0** – Microsoft's agent orchestration framework
- **Azure AI Search** – Vector database powering the RAG agents
- **Models** – GPT-3.5-turbo and Claude 3.7 Sonnet

Final Considerations

Before concluding, we want to briefly highlight two crucial topics that emerged during our work and which raised questions we couldn't overlook.

LLM Capabilities

We tested **GPT-3.5-turbo** and **Claude 3.7 Sonnet**, both of which struggled to correctly extract the different components of an optimization problem. While Claude 3.7 Sonnet performed slightly better, the improvements were minimal, highlighting the limitations of current LLMs in mathematical reasoning.

We also observed significant differences in how the models responded to the same prompt. Claude 3.7 Sonnet had the tendency to overextend its answers, which we mitigated by implementing a structured code-of-conduct guide within the Claude agents' system prompt, to keep it focused.

Finally, while agents are effective at executing tasks and streamlining workflows, we believe incorporating more algorithms to assist these agents could lead to a more efficient and reliable process.

Evaluation

A major challenge in developing our proof-of-concept architecture was testing and assessing the quality of our results. We mainly relied on qualitative metrics, but discussions with industry professionals emphasized the need for a more structured approach. We identified the following key insights:

- **No Immediate Metric**: Early evaluation lacks clear metrics. Trial and error, iterative prompt engineering and domain expertise are crucial for improvement.
- **Continuous Evaluation**: Testing should begin early with simple metrics and evolve throughout development.
- **Domain Knowledge**: A deeper understanding of the domain (mathematical optimization in this case) enables more precise test metrics and reduces reliance on the models' reasoning ability.
- **Isolated Testing**: Each agent and component should be tested separately to identify and resolve issues before integrating them into the overall architecture.
- **Custom Evaluation**: Standard evaluation tools for agent-based workflows are lacking, making tailored evaluation code essential.

Conclusion

Working on this project has been an eye-opening experience, allowing us to explore the intersection of mathematical optimization and AI agents. We gained firsthand insight into both the potential and limitations of current LLMs in handling complex mathematical reasoning.

As a team of three final-year Applied Computer Science students, we developed this project over six weeks as part of our degree program. Throughout the process, we not only deepened our technical knowledge but also recognized the challenges of designing and evaluating AI-driven workflows. This experience has reinforced our appreciation for the growing field of AI agents and their evolving role in automation, making us eager to see how they will evolve and shape the future.

Connect with us

Ayodeji Alli-Smith

Dean Terneu

Inês Bastos

References

Linear optimization explained: From fundamentals to Real-World applications - Gurobi Optimization. (2025, March 12). Gurobi Optimization. https://www.gurobi.com/resources/linear-optimization-explained/

Ibm. (2024, July 3). AI agents. *IBM*. https://www.ibm.com/think/topics/ai-agents