A Comprehensive Guide to Designing Multi-Agent Systems

Author: Simon Lacasse (Microsoft)

Table of Contents

Introduction 3

Modular Design and Abstraction 5

Modular Design: The Foundation of Flexibility 5

Abstraction: Hiding Complexity, Enhancing Usability 6

Wildfire Drone Example 9

Navigating Abstraction in Multi-Agent Design 16

Conceptual, Functional, Behavioral and Technical Abstraction 17

Top-Down, Bottom-up approach and Balancing the two 19-21

A Step-by-Step Guide to Designing a Multi-Agent Web Controller System 25

Step 1: Conceptual Level – Defining the Big Picture 27

Designing Agent Goal, Roles, Objectives, Abstraction 27-30

Step 2: Functional Level – Assigning Concrete Responsibilities 34

Designing Agent Function 34

Step 3: Behavioral Level – Defining Interactions and Protocols 40

Designing Agent Worflow, Evaluation, Orchestration, Memory, Communication 40-47

Step 4: Technical Level – Implementing the Details 54

Models, Frameworks, Implementation Details 40-47

Conclusion 63

Endnotes 65

Introduction

"Making the simple complicated is commonplace, making the complicated simple, awesomely simple, that's creativity." - Charles Mingus[1]

Great design doesn't erase complexity—it reshapes it, transforming intricate challenges into elegant, intuitive solutions. Complexity, if left unchecked, breeds rigidity, inefficiency, and fragility, limiting the potential of systems to scale and adapt. But when harnessed thoughtfully, complexity becomes an asset, empowering systems to evolve and tackle challenges far beyond their individual components.

This principle is crucial in **multi-agent systems**, where independent entities—software programs, robots, or intelligent assistants—must collaborate, negotiate, and make autonomous decisions. Like members of a **society**, each agent contributes to a shared goal without needing full awareness of every detail within the entire system. Without thoughtful design, these interactions risk descending into chaos, confusion, and inefficiency. However, by adopting a strategic approach centered on **modularity** and **abstraction**, multi-agent systems can achieve powerful coordination, resilience, and adaptive intelligence.

Modular design promotes **agency** by clearly separating concerns, enabling individual agents to operate autonomously, adapt flexibly, and innovate without interfering with the broader system. **Abstraction** complements this by facilitating collaboration, defining clear interfaces and roles that simplify interactions and coordination among agents. Together, modularity and abstraction create a robust multi-agent environment that balances independent decision-making with cohesive, structured cooperation.

Consider a **jazz ensemble**, where each musician contributes uniquely without needing to predict every note their peers will play. The magic happens not because of chaos, but because the underlying composition (abstraction) provides structure and harmony, while allowing modularity—the freedom for performers to innovate creatively within their own roles. Similarly, a well-designed multi-agent system empowers each agent to excel individually, orchestrating their collective intelligence into harmonious collaboration.

In this white paper, we'll explore core design principles for effective multi-agent systems, focusing specifically on:

Modular Design – Structuring systems into specialized, self-contained agents to enhance scalability and flexibility.
Abstraction – Layering complexity through clear interfaces to facilitate intuitive collaboration.
Design Strategies – A systematic approach for conceptualizing and implementing multi-agent solutions, from initial planning to final deployment.

Through practical examples—including autonomous wildfire drone networks and intelligent multi-agent web assistants—we'll demonstrate how these principles convert decentralized interactions into elegantly designed solutions.

By the end, you'll have gained a clear, structured approach to designing multi-agent systems that are scalable, resilient, efficient, and profoundly intelligent.

Modular Design and Abstraction

"You don't understand anything until you learn it more than one way."
— Marvin Minsky[2]

The most effective multi-agent systems don't just automate tasks— they **adapt**, **reason**, and **collaborate**. Their strength lies in how they are structured**: modular** in design and **abstracted** in execution. This combination allows systems to scale, evolve, and remain intuitive, even as they tackle increasingly complex problems.

Modular Design: The Foundation of Flexibility

Modular design is a methodology that breaks down complex systems into distinct, self-contained units (modules), each with a specific function. In multi-agent systems, these modules take the form of **agents**—independent, interchangeable components that can be developed, tested, enhanced and maintained separately while collaborating through well-defined interfaces.

This approach prevents the system from becoming a rigid, monolithic structure. Instead, it consists of **specialized agents**, each handling a particular task, such as data retrieval or decision-making. By structuring the system this way, modular design ensures **flexibility**, **scalability**, and **adaptability**.

Why This Matters

Modular design ensures **flexibility** because individual agents can be easily modified or replaced without disrupting the entire system.

Imagine a chatbot with different agents handling various tasks, such as answering FAQs and processing payments. If a new payment method needs to be added, only the payment-processing agent requires an update—no need to rebuild the whole chatbot.

Scalability is another key benefit. If a system needs to handle more users or process larger amounts of data, additional agents can be introduced to share the workload.

Consider an online shopping assistant with separate agents for product recommendations and customer support. As traffic grows, more recommendation agents can be added to maintain performance without overloading the system.

Finally, modular design supports **adaptability** by allowing the system to evolve over time.

If a weather prediction system originally designed for local forecasts needs to expand globally, new agents can be integrated to collect and process international weather data—without reworking the existing ones.

By structuring multi-agent systems modularly, developers ensure that they remain efficient, easy to update, and capable of growing alongside new requirements and technologies.

Abstraction: Hiding Complexity, Enhancing Usability

Modular design's separation of concerns lays the groundwork for effective abstraction.

Abstraction is a design principle that simplifies complex systems by focusing on high-level concepts while hiding unnecessary details. In multi-agent systems, abstraction allows agents to interact efficiently without needing to understand each other's internal workings. Instead, agents operate based on well-defined **roles**, **interfaces**, and **shared protocols**.

This approach prevents the system from being cluttered with excessive complexity. Instead, it consists of **hierarchical layers** of functionality, where each agent performs a specific role without being burdened by low-level implementation details. By structuring the system this way, abstraction ensures **clarity**, **efficiency**, and **interoperability**.

Why This Matters

Abstraction enhances **clarity** by enabling developers to work with high-level representations rather than dealing with unnecessary specifics.

Consider a smart home system where agents manage lighting, security, and temperature control. Each agent only needs to know what actions to take (e.g., "adjust temperature") rather than how the thermostat operates internally. This separation makes the system easier to design and expand.

Efficiency is another key benefit. By focusing on essential details, agents avoid processing irrelevant information, improving overall system performance.

Imagine an autonomous vehicle with agents handling navigation, obstacle detection, and speed control. The navigation agent doesn't need to process raw sensor data—it only requires abstracted insights, like "obstacle detected," allowing it to focus on route planning.

Finally, abstraction supports **interoperability** by allowing different agents and systems to communicate using standardized protocols, regardless of their internal complexity.

For example, in a financial transaction system, an AI-powered fraud detection agent can interact with payment processing agents using predefined security alerts, without needing to understand their exact algorithms.

By integrating abstraction into multi-agent systems, developers ensure that these systems remain scalable, maintainable, and adaptable while minimizing complexity and maximizing efficiency.

Summary

By integrating **modularity** and **abstraction**, multi-agent systems achieve both structural flexibility and operational efficiency. Modularity ensures that individual components can evolve independently, while abstraction simplifies interactions**,** making systems easier to use and scale. By leveraging **abstraction**, these systems enable both users and developers to focus on high-level objectives without being overwhelmed by low-level implementation details. Together, these principles create **intuitive**, **flexible**, and **resilient** multi-agent solutions capable of navigating complex and dynamic environments. Just as a smartphone abstracts the intricate workings of its hardware behind a simple touchscreen interface, a well-designed multi-agent system organizes specialized agents that collaborate efficiently while encapsulating complexity.

Now, let's apply the principles to a hypothetical scenario: **a wildfire drone example.**

Wildfire Drone Example

"Simple things should be simple, complex things should be possible."
— Alan Kay[3]

A wildfire is spreading unpredictably across a vast landscape, and we need an **intelligent drone system** to monitor and manage it in real time. The environment is constantly changing, fire behavior is erratic, and quick, data-driven decisions are critical to containing the spread.

Designing such a system is no simple task. It involves breaking down complexity into manageable components, each responsible for critical functions like fire detection, path planning, and multi-drone coordination. Moreover, the system must be **scalable** to accommodate wildfires of varying sizes and intensities, ensuring that additional drones can be deployed effectively as the fire spreads. It also needs to be **adaptable** to handle unpredictable fire behavior and changing environmental conditions, allowing it to adjust flight paths and response strategies in real time. Finally, the system must be **resilient**, capable of functioning even if some drones fail or communication is disrupted, ensuring continuous monitoring and response. Achieving these qualities requires a **modular design approach** and **strategic abstraction**, allowing us to build an intelligent, efficient, and robust drone system.

The Low-Altitude Drones

At a minimum, we need drones equipped with infrared cameras to detect heat intensity and thermal sensors for close-up hotspot identification. These drones will focus on gathering precise, high-resolution data on fire hotspots and temperature variations. This defines our first **modular drone**—a specialized, **data-gathering unit** that is highly capable but designed for a narrow, specific role.

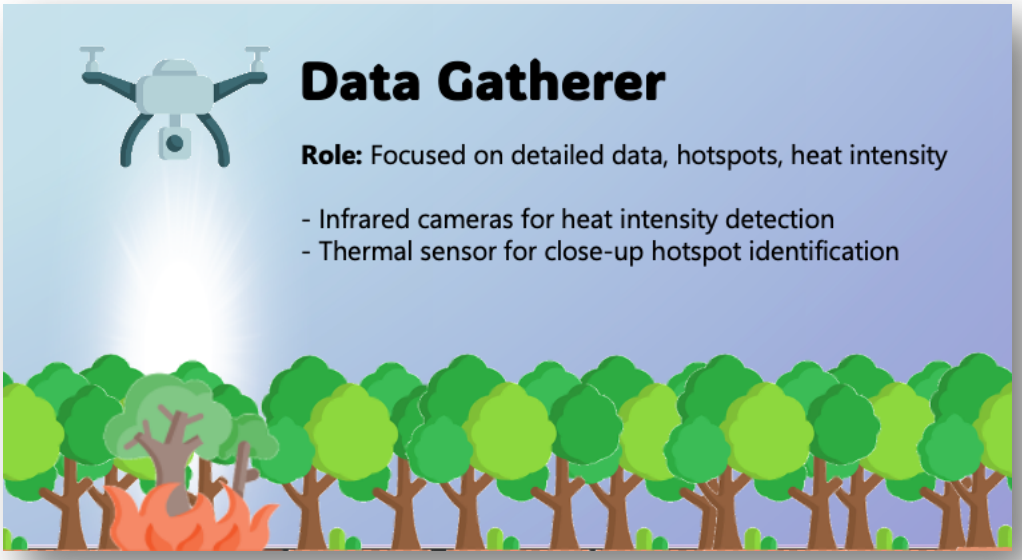

Figure 1. Data Gatherer Drone (Modularity)

To effectively monitor the wildfire, we must strategically position several of these drones across different sections of the fire, ensuring comprehensive coverage. This introduces our first level of abstraction: **the low-altitude layer**, where drones operate close to the fire, capturing critical real-time data from the ground level.

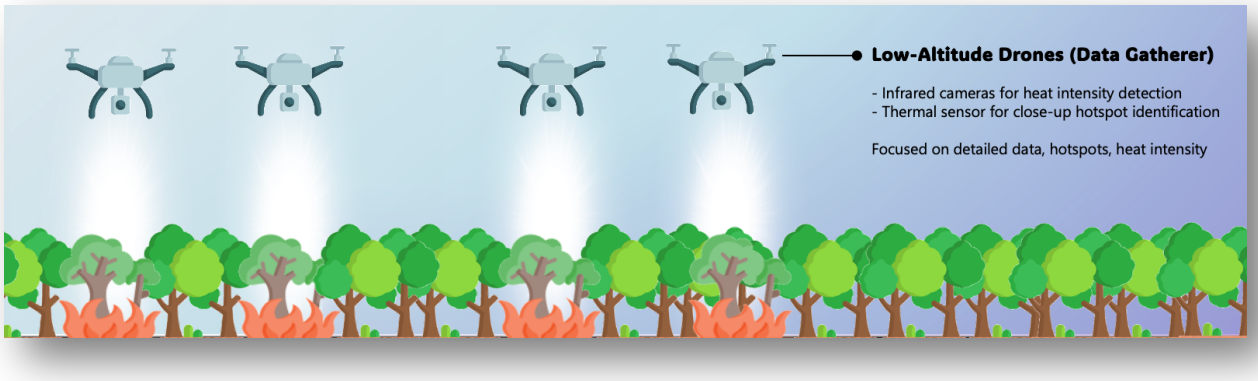

Figure 2. Low Altitude layer (Abstraction)

While this setup ensures localized data collection, it introduces several **limitations**.

The drone's function in isolation, with no communication or coordination between them. They are entirely unaware of other drones operating within the low-altitude layer, making collision

avoidance difficult and preventing dynamic collaboration. Their situational awareness is also limited to their immediate surroundings, restricting their ability to predict fire spread beyond their field of view. Additionally, system updates are inefficient—any improvements, such as refining collision avoidance algorithms, must be manually implemented on each drone, resulting in high maintenance costs and operational inefficiencies.

The Mid-Altitude Drones

To overcome the limitations of low-altitude drones, we introduce a new modular drone type that serves as a **tracking** coordinator. These tracking drones oversee multiple low-altitude units, optimizing their flight paths, preventing collisions, and dynamically redirecting them toward emerging hotspots. Equipped with image recognition technology, they can map fire progression and track wind patterns, allowing them to analyze fire spread and guide low-altitude drones more effectively.

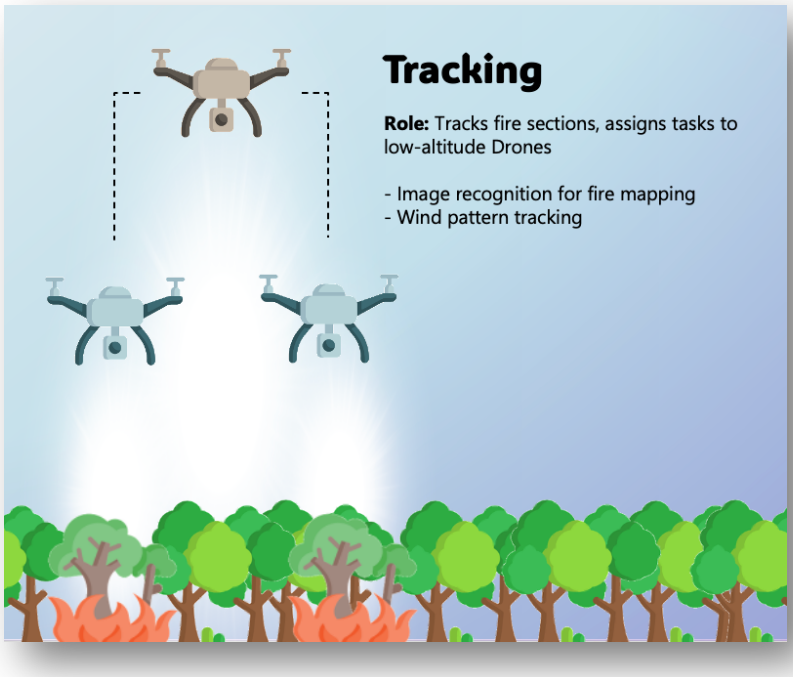

Figure 3. Tracking Drone (Modularity)

By leveraging a **hierarchical abstraction**, we encapsulate and streamline these tracking capabilities, shielding low-altitude drones from unnecessary complexity. This introduces our second level of abstraction: **the mid-altitude layer**. With this structured separation of roles,

low-altitude drones remain focused on real-time data collection, while mid-altitude drones handle coordination, tracking, and strategic direction—creating a more efficient and scalable wildfire monitoring system.

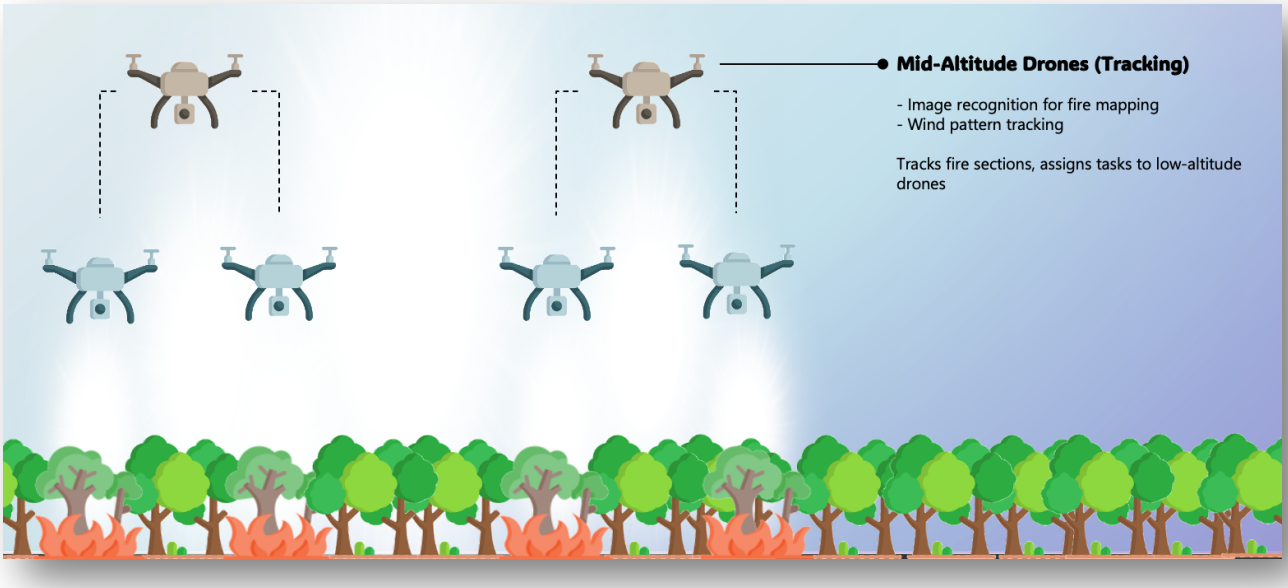

Figure 4. Mid-Altitude layer (Abstraction)

While this approach improves coordination, it still presents **challenges**:

Mid-altitude drones provide a broader perspective of the fire but still lack complete visibility, as they can only make decisions based on data received from low-altitude drones within their range. This reliance on lower-tier drones creates a dependency that limits their ability to assess the full extent of the fire independently. Additionally, coordinating between mid-altitude drones and multiple low-altitude drones introduces significant communication overhead, requiring efficient data-sharing protocols to ensure timely information exchange.

The High-Altitude Drones

To overcome these challenges, we introduce another modular drone type—a specialized **coordinator** drone with a holistic view of the wildfire's progression. These coordinator drones integrate data from both mid- and low-altitude drones, enabling strategic decision-making for resource allocation, emergency response, and system-wide coordination. Equipped with advanced

decision-making algorithms and task allocation logic, they direct mid-altitude drones in reorganizing low-altitude units as new fire outbreaks emerge. By dynamically distributing resources across different fire-affected zones, these drones ensure an optimal and adaptive response to rapidly changing fire conditions. Additionally, they serve as a crucial link between autonomous firefighting systems and human operators, providing real-time intelligence to enhance situational awareness, decision-making and human in the loop scenarios.

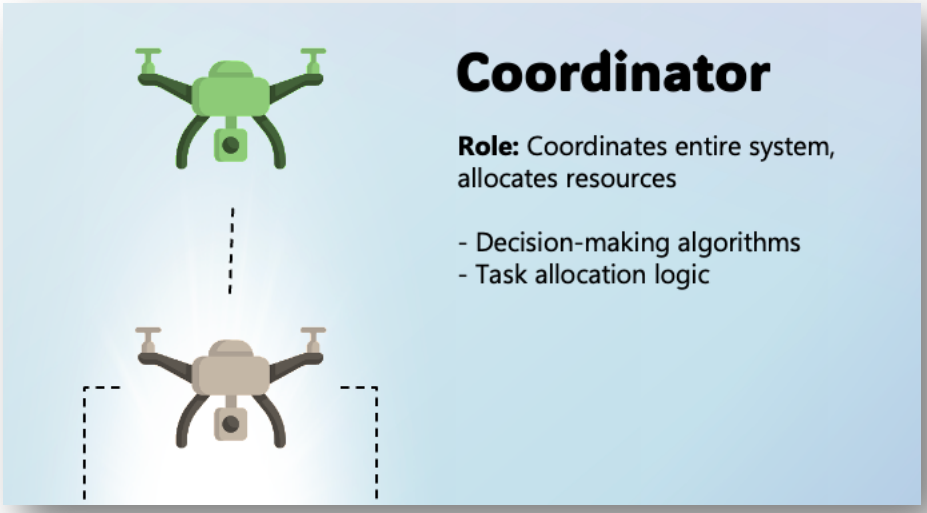

Figure 5. Coordinator Drone (Modularity)

The final layer of abstraction introduces the **high-altitude drone layer**, which acts as the overarching command and control system. Unlike mid- and low-altitude drones, which focus on localized data collection and tactical adjustments, high-altitude drones operate with a broader strategic perspective. They continuously analyze large-scale fire behavior, predict potential spread patterns, and optimize drone deployment to maximize efficiency. This layered, hierarchical approach ensures that each level of the system has a well-defined role, allowing for coordination, improved response times, and a more effective wildfire management strategy.

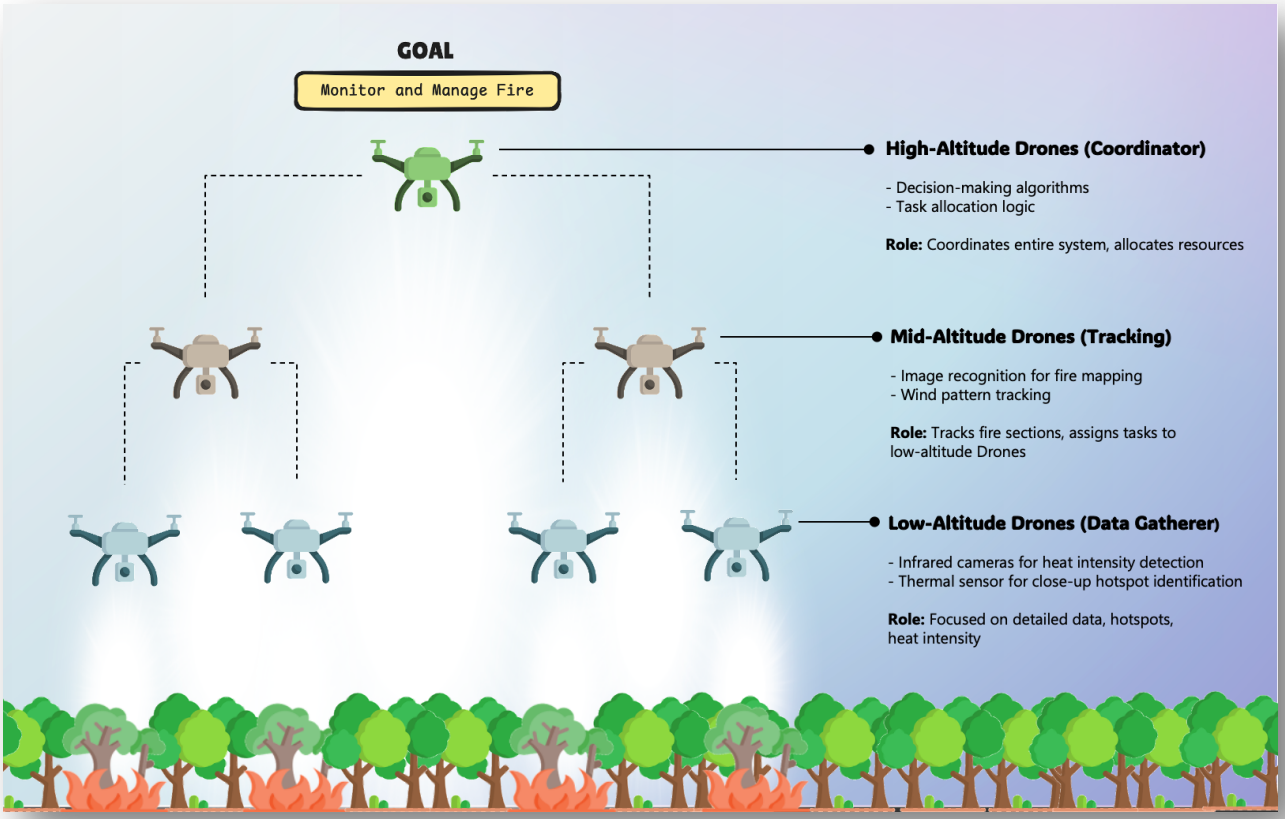

Figure 6. High Altitude layer (Abstraction)

Why it Matters

While all drones share the **common goal** of monitoring and managing wildfires, each operates with a distinct purpose. By leveraging hierarchical abstraction, the system ensures that drones focus on their primary tasks while higher-level coordination optimizes efficiency. This structured approach simplifies complexity, as each drone functions within a specific abstraction layer, promoting collaboration and more effective wildfire management.

This methodology aligns with fundamental principles of system design and architecture, where complexity is managed through abstraction and modularity. Just as low-level components in a system handle specialized tasks without requiring a global view, low-altitude drones focus on localized fire monitoring. Mid-altitude drones act as intermediaries, akin to middleware in distributed systems, facilitating communication and optimizing performance. Meanwhile, high-altitude drones function as orchestrators, overseeing execution, dynamically allocating resources,

and ensuring fault tolerance by adjusting to failures or evolving conditions.

Summary

The effectiveness of this layered approach extends beyond wildfire response, demonstrating the power of modular design and hierarchical abstraction in autonomous systems and industrial automation. By structuring the system into well-defined roles, it enhances scalability, fault tolerance, and efficiency, ensuring that failures in one component do not compromise the entire network. This abstraction **simplifies development**—allowing for easier debugging, upgrades, and expansion—while also improving user experience by hiding complexity behind an intuitive interface.

But what happens when systems grow in complexity? How do we prevent modular components from becoming disjointed fragments instead of a cohesive whole? Abstraction isn't just about structuring systems—it's also a powerful tool in the design process itself.

In the next chapter, we'll explore how abstraction guides decision-making at every stage, from defining high-level objectives to implementing technical details. By applying conceptual, functional, behavioral, and technical abstraction, we can break down complexity, ensure integration, and design systems that are both adaptable and scalable.

Navigating Abstraction in Multi-Agent Design

> "The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."
> — Edsger Dijkstra[4]

Designing multi-agent systems requires a carefully **structured approach** to avoid overwhelming complexity, fragmented solutions, and poorly integrated components. Without a clear design **methodology**, we risk getting lost in technical details or building an inefficient system due to a lack of cohesive conceptual planning. To address this, we leverage abstraction again, but this time, we apply it horizontally at every level of our hierarchal abstraction—across high, mid, and low layers—ensuring a consistent, standardized methodology that guides us throughout the multi-agent design process.

At every level, we apply abstraction across four key dimensions:

- **Conceptual Abstraction** – Defines the overarching mission and strategic goals of the system.
- **Functional Abstraction** – Structures roles and functions, ensuring each agent has a well-defined purpose.
- **Behavioral Abstraction** – Governs interactions, coordination, orchestration and decision-making among agents.
- **Technical Abstraction** – Focuses on the implementation details, optimizing performance and integration.

This iterative, structured abstraction approach allows for a balanced **top-down** and **bottom-up** design strategy. From a top-down perspective, we maintain focus on high-level objectives, ensuring that technical implementations align with the broader system goals. Conversely, from a bottom-up perspective, we ensure that low-level technical components inform and refine the system's conceptual design, creating a continuous feedback loop between implementation and strategy. By structuring abstraction at every stage, we achieve a **harmonized system**

architecture that is both adaptable and scalable, enabling developers to navigate complexity while ensuring all system components integrate seamlessly.

Conceptual, Functional, Behavioral and Technical Abstraction

In this multi-agent design approach, abstraction operates at different levels—**conceptual, functional, behavioral, and technical**—each offering distinct perspectives to manage complexity effectively.

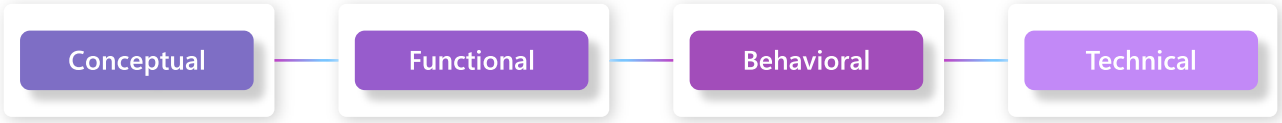

Figure 7. Levels of Abstraction – Conceptual, Functional, Behavioral and Technical

By consciously leveraging these levels, we can design systems that balance elegant **orchestration** with robust **scalability**. Like switching between wide-angle and zoom lenses on a camera, abstraction enables us to shift focus dynamically: capturing the big picture, examining fine details, or balancing somewhere in between.

The abstraction hierarchy—**conceptual**, **functional**, **behavioral**, and **technical**—not only helps manage complexity but also shapes how we design agents. Take our drone example: the **conceptual level (**monitoring the fire) defines the overarching mission and informs the modular design of our drones. **Functional abstraction** (roles like perimeter mapping or hotspot detection) clarifies responsibilities and helps define the necessary functions for each drone. **Behavioral abstraction** (interaction protocols) dictates how drones collaborate and coordinate. **Technical abstraction** (data structures and implementation details) ensures each drone can execute its role.

By structuring design in this way, we isolate complexity to what is most relevant at each stage, keeping unnecessary implementation details in the background and streamlining the development process.

Conceptual Level

At the conceptual level, everything is about the **"what"**. *What is the purpose of the system? What are the goals, and how will agents interact at the highest level?*
At this point, agents are seen as black boxes, working together to solve a problem.

When to Use It: *Conceptual abstraction is perfect for the early stages of design when you're framing the problem and defining roles. It gives you a clear vision of the system without getting bogged down in details.*

Functional Level

Functional abstraction dives a level deeper. Here, the focus shifts the **"roles and responsibilities**" which translates into functions and workflows. You're defining what each agent does, breaking tasks into logical components, and ensuring clear boundaries between them and how the interconnect.

When to Use It: *Use functional abstraction when designing functional workflows, assigning tasks, and building the overall architecture. It's where modularity begins to shine, making systems easier to scale and adapt. Using functional abstraction in an enterprise is a great way to map out existing business processes.*

Behavioral Level

Behavioral abstraction is all about the **'how'**. *How do agents interact? What rules govern their behavior?* This level focuses on the communication protocols, orchestration and task execution rules that allow agents to work together effectively.

When to Use It: *Behavioral abstraction is useful when implementing the system, defining precise interactions, and ensuring that agents behave predictably under different conditions. In an enterprise setting this is a great way to start mapping out your cognitive architecture.*

Technical Level

At the technical level, the system opens, and the inner workings of how each agent work. Here,

you're dealing with the **nuts and bolts**—functions, models, frameworks, and data structures.

When to Use It: *Use technical abstraction when you're ready to build, debug, and optimize the system. This is where you make engineering decisions like choosing tools, models, frameworks, and ensuring performance efficiency.*

Top-Down Approach

When designing a multi-agent system, a **top-down approach** starts at the conceptual level and progressively refines the system through functional, behavioral, and technical layers. This method is particularly effective for greenfield projects, where a system is being designed from scratch. By beginning with high-level goals and breaking them down into specific roles, interactions, and technical implementations, this ensures that the system is **goal-driven** and structured from the outset, making it easier to align with predefined objectives and constraints.

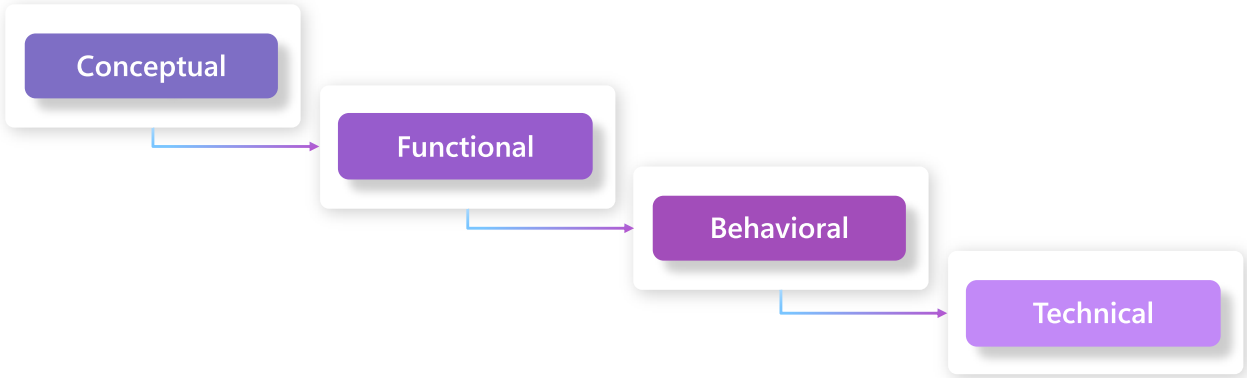

Figure 8. Top-Down Approach

For example: In a wildfire drone system designed with a top-down approach, the process begins with the overarching objective: "monitor and manage fire". From this goal, the system is structured into key components, including high-altitude, mid-altitude, and low-altitude drones. Each category of drone is assigned specific functions and roles, followed by the development of tracking and coordination mechanisms. Finally, the implementation phase focuses on integrating sensors and cameras to optimize detection and response capabilities.

While this method is effective for ensuring that the entire system is driven by a cohesive vision and set of goals, it can also introduce challenges. Early, broad-stroke decisions made at the conceptual level may overlook critical technical constraints or emergent behaviors, leading to designs that are overly **rigid** or that must be significantly revised later. As a result, top-down approaches can sometimes stifle flexibility and adaptability if not continuously informed by feedback from lower-level details.

Bottom-Up Approach

A **bottom-up approach** starts at the technical level and moves upward. That means you first define the **nuts and bolts** (technical details like data structures and models), then build interaction rules (behavioral abstraction), assign roles (functional abstraction), and finally ensure alignment with high-level objectives (conceptual abstraction). This approach allows for **flexibility** in implementation because lower-level components shape the system's overall design. This works well when optimizing existing systems or solving isolated technical challenges before scaling up.

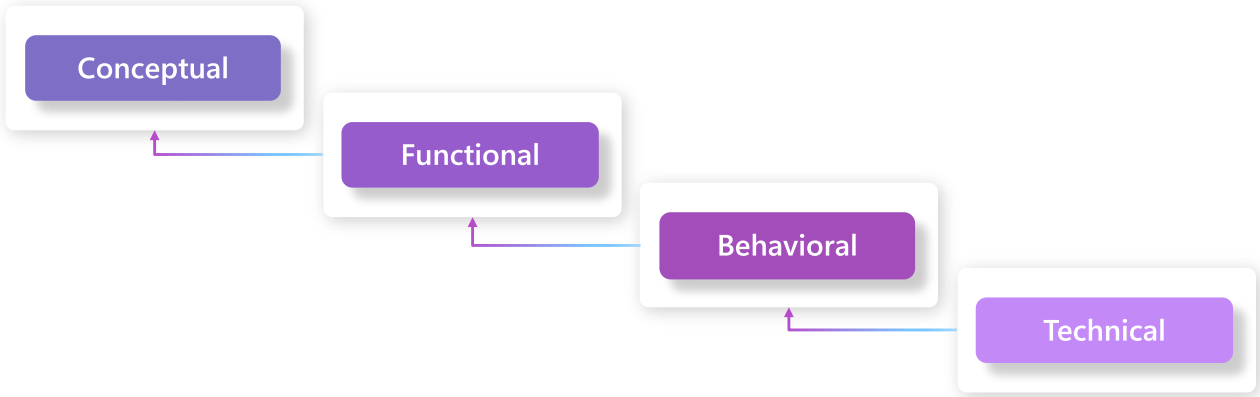

Figure 9. Bottom-Up Approach

Example: *A bottom-up wildfire drone system starts by building and testing low-level components (like sensors and basic communication). Developers then define and refine interaction rules between drones, scale up their capabilities, assign them specific roles and functions, and finally ensure the entire system aligns with the high-level goal of rapid wildfire detection and response.*

Often, abstraction is approached from the bottom up, starting with technical components like frameworks, APIs, or algorithms, and then working upward to define behaviors, roles, and goals.

While this approach can be effective in certain cases, such as optimizing an existing system or addressing specific technical challenges, it can also lead to fragmented designs that lack cohesion. Key decisions made at the technical level may not always align with overarching objectives, resulting in inefficiencies or the need for significant rework.

Balancing the Two Approaches

The most effective strategy often lies in striking a **balance** between the top-down and bottom-up methods. By starting with high-level objectives (top-down), you ensure that every component of the system is guided by a cohesive vision. At the same time, building from the technical level (bottom-up) uncovers practical constraints and opportunities for optimization early on. **Iterating** between these perspectives enables designers to refine conceptual goals using **feedback** from lower-level prototypes, and vice versa. This approach not only helps maintain alignment with overarching objectives but also encourages flexibility, adaptability, and continuous improvement throughout the development process.

For example: *A top-down strategy might define the conceptual behavior of agents in a simulation, but insights from early technical prototypes could inform adjustments to the high-level design. Conversely, bottom-up development might uncover emergent behaviors that reshape the initial conceptual vision.*

When you get this balance right, abstraction becomes a **superpower**. It allows you to tackle complexity **piece by piece**, designing systems that work at every level.

Levels	When to Use	Outcome
Conceptual	Early planning; defining goals.	A clear high-level design.
Functional	Designing workflows and tasks.	Well-defined functions and workflow.
Behavioral	Implementing behaviors and logic.	Structured and predictable interactions.
Technical	Building and optimizing systems.	Efficient and performant implementation.

Figure 10. Levels of Abstraction Guide

Wildfire Drone Revisited

Let's revisit our wildfire drone example and see how **conceptual**, **functional**, **behavioral**, and **technical** abstraction play a role at every level of the system. By leveraging this approach, we can break down complexity, streamline coordination, and enhance the efficiency of drone operations in wildfire management.

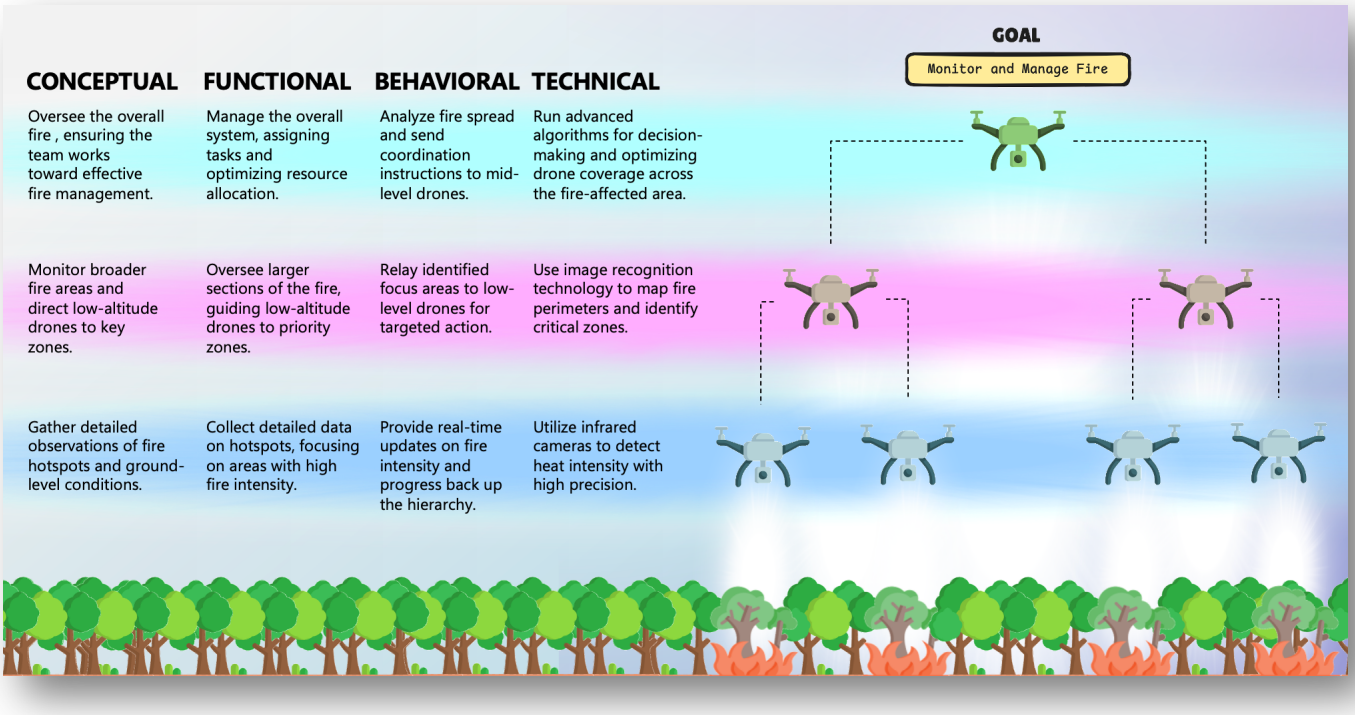

Figure 11. Conceptual, Functional, Behavioral and Technical Design (Drone Example)

Conceptual Abstraction

The **goal** is clear—use drones to monitor and manage a forest fire. Drones work together to observe the fire, gather data, and coordinate a response. The system consists of three layers of modular drones—**High-Altitude**, **Mid-Altitude**, and **Low-Altitude Drones**—each responsible for a specific role: strategic oversight, tracking, and data collection.

Functional Abstraction

Core functions and **functional workflows** are assigned. High-altitude coordinate drones analyze the entire forest and send directions to mid-altitude drones, which track fire progression and assign tasks to low-altitude drones, responsible for collecting real-time data.

Behavioral Abstraction

The drones **communicate** and **collaborate** to ensure smooth operations. High-altitude drones oversee and adjust strategies, mid-altitude drones track movements and relay information, and low-altitude drones work autonomously within their assigned areas.

Technical Abstraction

The system is implemented through **software** and **hardware** solutions. High-altitude drones run decision-making algorithms to optimize drone positions and coverage., mid-altitude drones employ image recognition to map fire perimeters and manage tracking, and low-altitude drones are equipped with sensors to gather fire data and use infrared cameras to detect heat.

Summary

Building multi-agent systems is as much about managing complexity as it is about enabling intelligence. Without a structured approach, we risk creating systems that are brittle, inefficient, and difficult to scale. By applying abstraction at every level—**conceptual**, **functional**, **behavioral**, and **technical**—we create a design framework that brings order to chaos, ensuring that agents work together while maintaining adaptability.

This layered abstraction model does more than just simplify design; it provides a roadmap for balancing high-level strategic vision with low-level technical execution. It allows us to design with **clarity**, preventing technical noise from drowning out overarching objectives, while also ensuring real-world constraints shape high-level decision-making. When applied effectively, this approach transforms multi-agent systems into powerful, coordinated entities capable of tackling complex problems—like wildfire detection—**efficiently** and **autonomously**.

So how do we actually implement this methodology in practice? What does it look like to build a system that integrates these principles from the ground up? In the next section, we'll walk through a **Step-by-Step Guide to Designing a Multi-Agent Web Controller System**, demonstrating how to bring these abstract concepts to life.

A Step-by-Step Guide to Designing a Multi-Agent Web Controller System

> "Good design is actually a lot harder to notice than poor design, in part because good designs fit our needs so well that the design is invisible."
> — Donald Norman[5]

Imagine having a digital assistant that doesn't just answer your questions but actively browses the web on your behalf—researching information, comparing products, booking reservations, and even securing that last-minute dinner reservation at your favorite restaurant. Instead of spending hours sifting through websites, this intelligent system does the heavy lifting for you, browsing the web, gathering relevant data and making decisions based on your preferences.

This guide will walk you through the process of designing an **autonomous multi-agent web controller**—a system where multiple AI agents work together to navigate the internet through a web browser efficiently. By leveraging the four key levels of abstraction, we can break down the complexity of multi-agent design into structured, manageable steps. From defining system architecture to optimizing intelligent agent coordination, this approach enables the creation of a **scalable** and **adaptable** web browsing multi-agent system.

Building such a system requires more than just automation—it demands intelligent coordination among multiple agents, each specializing in different tasks.

Designing an **autonomous multi-agent web controller system** may seem daunting at first—and to some extent, it is. However, by employing a structured approach based on **abstraction** and **modular design**, we can break down this complexity into manageable layers. By applying these four levels of abstraction in our design process, we can first establish a clear system architecture, define modular workflows, ensure agent coordination, and refine the technical implementation for optimal performance. Ultimately, this proposed abstraction approach provides a structured pathway to simplify complexity and create a more intuitive and efficient design.

Note: There are numerous approaches to developing multi-agent systems, and this step-by-step guide is intended as an illustration rather than a one-size-fits-all solution. While we will walk through the design principals for developing a multi-agent web controller system and provide working code, the primary goal of this guide is to demonstrate how abstraction and modular design can be effectively applied in a practical scenario. Rather than prescribing a rigid framework, we aim to showcase key principles and methodologies that can be adapted to suit various multi-agent system architectures.

The aim of this multi-agent web controller system is to create an autonomous agent capable of executing tasks by browsing the internet and taking actions on your behalf until the objective is achieved. It interacts with your web browser just like a human—clicking, moving the mouse, entering text in search bars, and navigating web pages.

Each step will be accompanied by a detailed explanation, code examples, and a complete working implementation to illustrate the principles in action. The illustration below provides a high-level architecture diagram of what we are going to design.

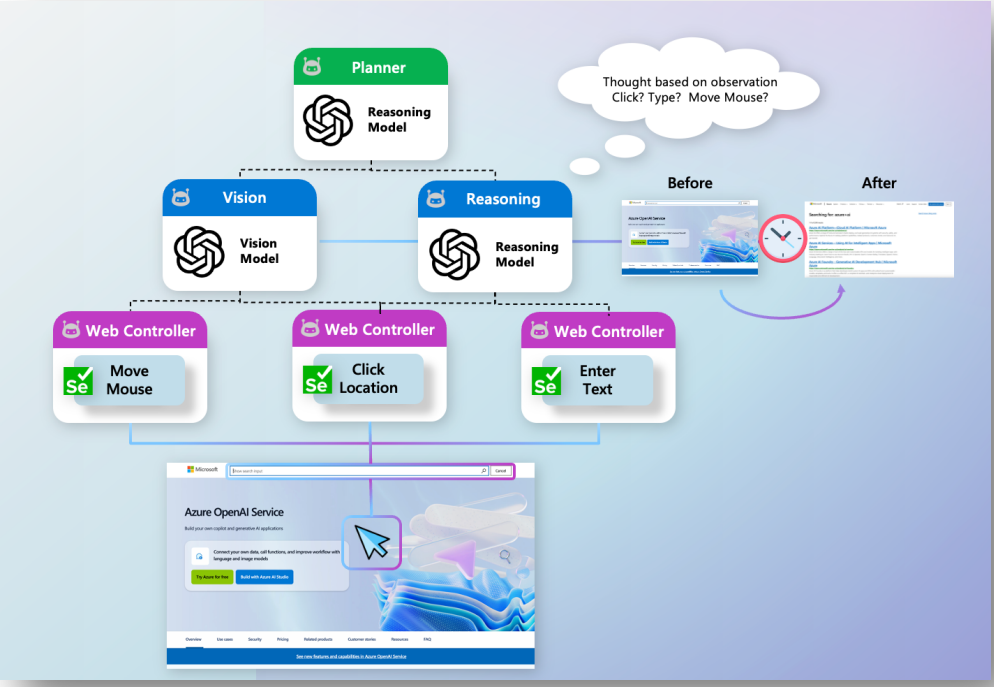

Figure 12. Multi-Agent Web Controller High Level Architecture

Step 1: Conceptual Level – Defining the Big Picture

Remember, at the conceptual level, it's all about defining the **"what."** What is the system designed to accomplish? What are its core objectives, and how will the agents interact at a high level to achieve them? At this stage, we don't focus on the internal mechanics of each agent but rather see them as independent entities working together as part of a larger problem-solving framework.

The conceptual level serves as the **foundation**, offering a bird's-eye view to ensure that every subsequent design decision aligns with the overarching goal. Without clear guidance at this stage, the system risks becoming fragmented or unnecessarily complex.

Defining the Overall Goal

The first step at the conceptual level is to determine the **overall goal**. This part might sound easy, but it requires careful thought and precision. Defining the overall goal sets the foundation for every subsequent decision, from the design of individual agents to the orchestration of the entire system. It requires identifying not only what the system should achieve but also, the constraints, priorities, and metrics for success.

Let's beginning by defining our overall goal, here we can use a concrete example to help us ground our abstract ideas, making the goal more relatable and testable.

- **Goal:** Autonomously browse the internet to complete the use's tasks.
- **Example:** Create a travel plan for a trip to Cancun, Mexico.
- **Requirements:** The system must navigate various websites (e.g., travel aggregators, hotel booking portals, tourism information pages) autonomously to gather data and compile a comprehensive itinerary.

Once the overall mission and goals are defined, the next step is to structure the system into **modular components**, each handling different levels of complexity.

Defining the Agent Roles

The overall goal is to have an autonomous system that can browse the internet and perform user tasks. This means the system must be able to:

- Plan a sequence of web operations (e.g., search for information, navigate pages, click buttons).
- Understand and parse web content (e.g., extract relevant text, images, links, buttons).
- Execute the necessary actions (e.g., clicking on links, entering text).

A key principle in this design is the clear definition of **roles**. By structuring the system with distinct, purpose-specific agents, we foster modularity and scalability. This prevents role overlap and ambiguity while ensuring that each agent operates within a well-organized hierarchy.

Clearly defining roles helps translate abstract responsibilities into concrete, modular agents. By identifying specific responsibilities—planning tasks, parsing and analyzing web content, and executing actions—we clarify the boundaries and interactions among agents. This approach allows us to recognize precisely which distinct agents we require, such as a **Web Coordinator** for orchestrating tasks, a **Web Browsing Agent** for understanding web content, and a **Web Controller Agent** for performing actions directly on webpages. The clear delineation of roles ensures each agent contributes uniquely and effectively, thus optimizing the overall efficiency and maintainability of the system.

Below is an overview of what our agents and their roles look like:

Web Coordinator (Planning & Orchestration)

Role: This component is responsible for formulating a plan to achieve the user's goal. It determines which tasks need to be performed, in what order, and assigns those tasks to the appropriate agents.

Web Browsing Agent (Parsing & Analysis)

Role: This agent interacts with the content of the web page to extract actionable elements. It understands the structure of the page (HTML, DOM) and identifies items such as buttons, input

fields, links, or other relevant elements.

Web Controller Agent (Action Execution)

Role: This agent takes direct actions on the web page, such as moving the mouse, clicking buttons, or entering text. It interacts with the browser's interface and the underlying webpage elements.

Defining Agent Abstraction layers

Just as in our top-down drone example—where a top-level planner agent handled strategic decision-making, a mid-level controller managed task execution, and a low-level executor carried out precise actions—we can apply a similar approach to our web browsing system. This is where **hierarchical abstraction** becomes essential.

By designing the system to operate across multiple layers of abstraction, we establish a clear separation of responsibilities. The system operates on a top-down hierarchy where a **high-level Web Coordinator** sets overall goals and breaks them down into actionable steps. These steps are then passed to **mid-level Browser Managers**, which interpret web elements and decide the next best actions, such as clicking or typing. Finally, **low-level Web Controllers** execute these specific tasks like moving the mouse or entering text, ensuring that each action aligns with the broader strategy.

This hierarchical structure improves coordination and efficiency and enhances scalability an adaptability of our web controller agent system by enabling agents to work together while maintaining modularity and flexibility.

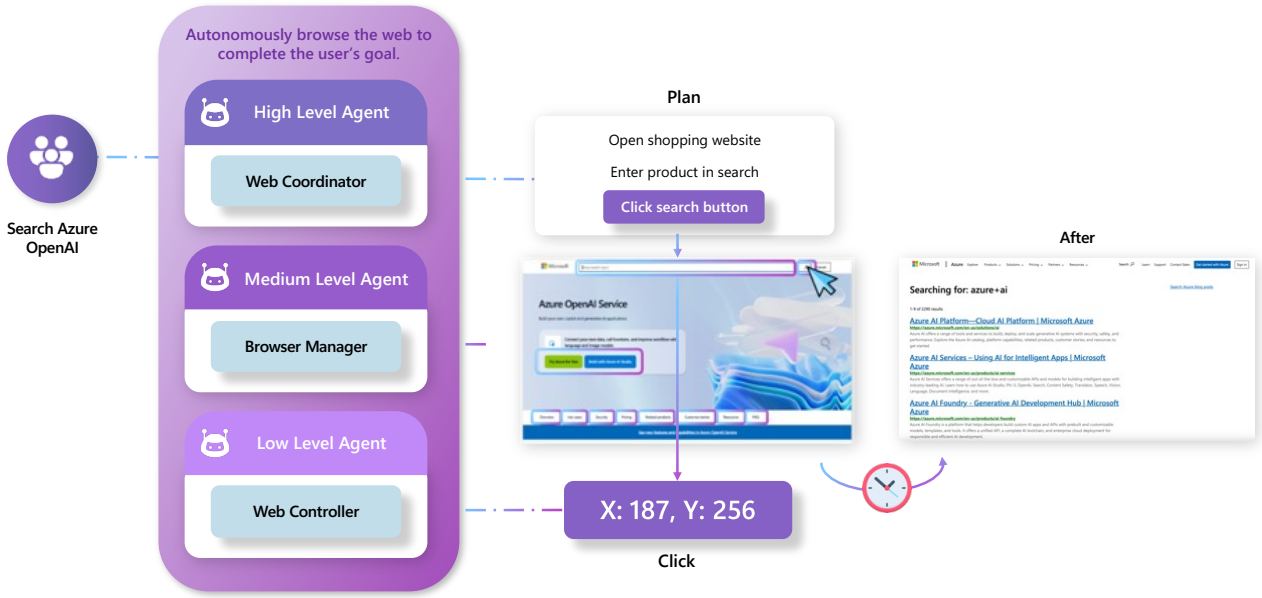

Figure 13. Conceptual Design – Modular and Abstraction

Designing Agents with Objectives

Once roles are established, we can assign **objective functions** that serve as the guiding principles that drive system performance. An objective function is a measurable criterion that evaluates an agent's effectiveness, informs decision-making, and ensures alignment with the system's overarching goals. By defining **roles** first, we create a foundation upon which **objective functions** can be tailored to specific agent responsibilities, ensuring that each agent contributes meaningfully to the system's mission.

Let's apply this approach to our conceptual design:

High-Level Agent - Web Coordinator

> **Objective Function:** Generates well-structured, goal-aligned plans that effectively guide the system toward the user's objective.

Medium-Level Agent - Browser Manager

Objective Function: Ensures accurate task execution by correctly interpreting and translating steps into precise actional web elements while maintaining efficiency and consistency.

Low-Level Agent - Web Controller

Objective Function: Maximizes execution accuracy by ensuring precise clicks, error-free text inputs, and smooth navigation, minimizing failures or unintended interactions.

Agents	Roles	Objective Function
High Level	Creates a structured plan by breaking the mission into subtasks and assigning them to subordinate agents.	Generates well-structured, goal-aligned plans that effectively guide the system toward the user's objective.
Medium Level	Translates high-level plans into browsing tasks and retrieves relevant data.	Ensures precise and efficient execution of high-level plans in web interactions.
Low Level	Executes web interactions like clicking, typing, and scrolling as directed.	Ensures accurate clicks, error-free inputs, and smooth navigation while minimizing errors.

Figure 14. Agent Roles and Responsibility

Why This Matters

This layered approach demonstrates why abstraction is key in multi-agent design:

- The **high-level agent** doesn't care about pixels or HTML elements—it simply orchestrates everything at a strategic level.
- The **mid-level agents** don't execute every detail; they focus on processing the screen and deciding actions.
- The **low-level agents** don't need to worry about decision-making; they just follow commands.

By structuring the system this way, this multi-agent web controller can autonomously navigate the web without manually programming every step. Instead of hardcoding every webpage interaction, this modular architecture makes the system **adaptive**, **scalable**, and **generalizable**—a perfect example of how multi-agent abstraction powers real-world AI applications.

By thinking in terms of these **hierarchal abstraction layers**, high, medium, low, we can better understand how the system should function and design agents with greater granularity. This approach allows us to clearly define each agent, ensuring their tasks align with the overarching mission.

Summary

In essence, our conceptual design is a testament to the power of modular thinking and layered abstraction. By separating our system into distinct components—each responsible for a well-defined task—we not only simplify the design but also enhance scalability and adaptability. This approach allows the high-level coordinator to remain focused on the overarching mission without getting bogged down in the details of individual actions. Meanwhile, the mid- and low-level agents execute their roles with precision, each contributing to a cohesive, goal-driven whole. As we look ahead, our next phase—Functional Level: Assigning Concrete Responsibilities—will transform these abstract modules into actionable, real-world tasks, bridging the gap between strategy and execution.

Technical Implementation

At this stage, we have established our overall goal, defined each agent as a modular component using **hierarchical abstraction**, and assigned them specific **roles** and **objective functions.** With this structured foundation in place, we can now begin implementing our agent classes. To ensure clarity and consistency, we will use a system message to define the roles and objectives for each agent, guiding their behavior within the system. Considering these agents will be grounded by a general-purpose model, the **system prompts** acts as our high-level agent abstraction. Here we can give our agents their **specific roles** and **objective functions**.

Python

```
class HighLevelAgent:
 system_prompt = " You are a planner agent that creates a structured plan to
 achieve the user's goal by determining tasks and their order. It assigns tasks
 to the right agents to ensure effective goal completion."
```

```
class MediumLevelAgent:
 system_prompt = " You are a Web Browser agent. Given a planning task and a list
 of actionable UI elements, determine the best UI action to perform.”

class LowLevelAgent:
 system_prompt = " You are the Low-Level Web Controller. Your responsibility is to
 ensure that the UI action is executed precisely;”
```

Step 2: Functional Level – Assigning Concrete Responsibilities

At the **functional level**, we define the core capabilities of each agent as **isolated functions**. These functions represent the fundamental actions an agent can perform, such as extracting elements, clicking buttons, or generating a task breakdown. This stage does not define how these functions interact but ensures that each agent has a well-defined set of operations that support its role in the system. This step ensures that each agent has well-defined functional responsibilities, avoiding inefficiencies or breakdowns in execution.

Defining Agents Functions

To achieve this, we start by analyzing the conceptual roles of each agent and breaking them down into specific, **actionable functions**. The conceptual design provides a high-level understanding of what each agent is responsible for, but to transition into functional design, we need to explicitly define the set of functions that will enable the agent to fulfill its role within the system. These functions establish the **core capabilities** of each agent and provide the foundation for each of our agents.

At the functional level, we focus on structuring the functions that each agent will perform, ensuring they align with the overall system objectives. For example, if an agent is responsible for executing actions on a user interface—we need to define its functions for moving the mouse, clicking buttons, entering text, and handling errors. Each of these functions serves as a building block that allows the agent to interact with its environment in a controlled and structured manner.

Furthermore, defining functions at this stage enables a smoother transition to the **behavioral level,** where we refine how agents interact and execute tasks under different conditions. With a solid functional design, we can begin outlining the inputs, outputs, and expected operations of each agent, which will later guide their behavioral strategies and decision-making mechanisms.

Below are the core functions needed for each of our agents:

- **High-Level Agent (Web Coordinator) Functions**
  - **Generate Subtask:** This function takes the users goal and decomposes the goal into the first subtasks and generates subsequent subtasks to accomplish the goal.
- **Medium-Level Agent (Browser Manager) Functions**
  - **Extract Elements:** This function extracts the actionable elements from the web page DOM
  - **Assigns Elements:** This function assigns the actionable element based on the subtask.
- **Low-Level Agent (Web Controller) Functions**
  - **Move Mouse:** This function executes the mouse move action.
  - **Single click:** This function executes the single click action.
  - **Enter Text:** This function executes the enter text action.

The functions for each agent are illustrate below in our diagram.

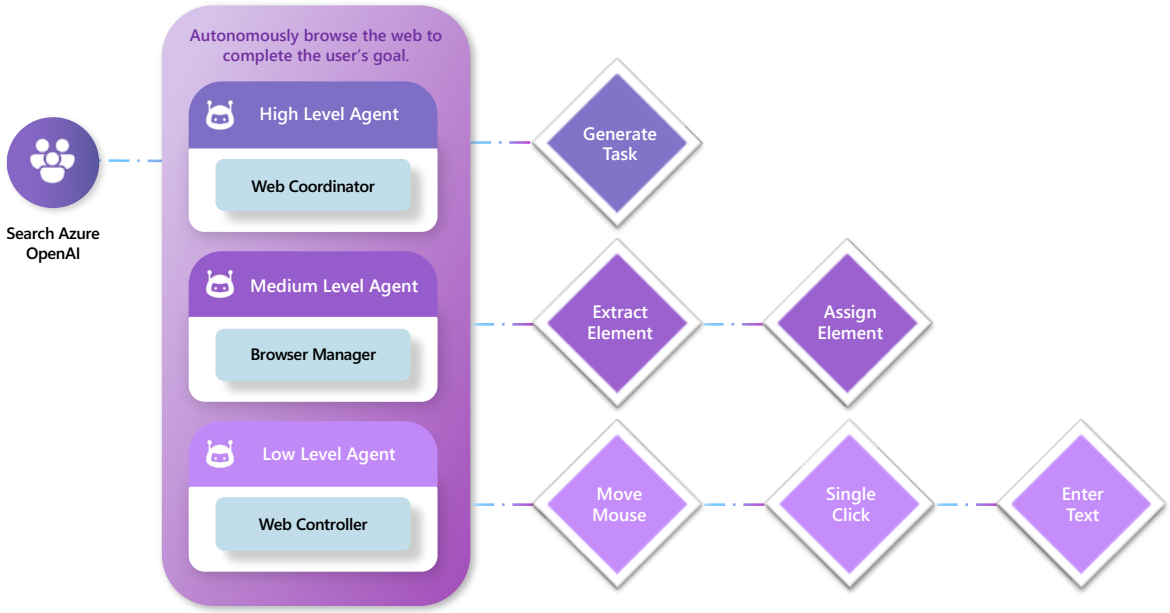

Figure 15. Agent Functions

Why This Matters

Defining concrete functions at the functional level is more than just an exercise in structuring code—it is about ensuring that agents are actionable. Multi-agent systems thrive on well-defined capabilities, and without this step, even the most sophisticated conceptual design collapses into inefficiency and ambiguity.

This phase is crucial because it turns abstraction into **capability**—agents now have explicit tools to act on their roles, making the system executable rather than just theoretical.

Additionally, by isolating functions at this level:

- We **prevent dependency chaos** – Agents can operate independently within their scope without unnecessary cross-communication, reducing complexity and potential failure points.
- We establish the **system's constraints** – By defining what an agent can and cannot do, we limit unintended behaviors and create a predictable operational framework.
- We optimize **adaptability** – A modular function-based structure allows agents to be reused, swapped, or expanded, ensuring that the system can evolve with minimal overhaul.

In essence, the functional level is the bridge between what the system is supposed to do and how it actually does it. It is here that we create the true backbone of autonomy—without these functional definitions, there is no execution, only intention.

Summary

At this stage, we have moved from conceptual abstraction to **concrete execution**. Each agent now has a well-defined set of functions, ensuring that the system is structured, scalable, and efficient.

However, defining functions is not enough—real-world environments are unpredictable, requiring agents to adapt and make intelligent decisions based on context. This leads us to the **Behavioral Level**, where we will shift our focus from isolated actions to dynamic interactions, decision-making, and error handling.

This transition is where autonomy becomes not just structured, but **intelligent**—transforming predefined functions into adaptive, goal-driven behavior that can navigate the complexities of real-world web automation.

Technical Implementation

At the **functional stage**, we have established our core modular structure for each of our agents.

Below, our classes from the conceptual design have been updated with the necessary functions and **high-level pseudocode,** outlining their intended behavior. This includes defining the logic required for each function. However, it is important to note that only the pseudocode and logical structure are required at this stage—actual implementation will be addressed at the **technical level.** This approach allows us to refine our design and ensure alignment before moving forward with development.

Python

```
class HighLevelAgent:
  system_prompt = …


 def generate_task(self) -> Subtask:
     """
     Step 1: Capture Screenshot
         - Take a screenshot of the current webpage and encode it.

     Step 2: Prepare AI Input
         - Create a prompt using the user's goal and memory.
         - Include the encoded screenshot in the message.

     Step 3: Query AI Model
         - Send the request to an AI model to generate an initial subtask.

     Step 4: Return Subtask
         - Parse and return the generated subtask.
     """
```

Python

```
class MediumLevelAgent:
  system_prompt = …

 def extract_ui_elements(self) -> List[dict]:
     """
     Step 1: Find Interactive Elements
         - Identify buttons, inputs, links, and other interactive elements on the
 webpage.

     Step 2: Filter Visible Elements
         - Exclude elements that are not currently visible.

     Step 3: Extract Relevant Data
         - Gather attributes and position information for each element.

     Step 4: Return List of Actionable Elements
         - Return structured data for use in further processing.
```

Python

```
 def translate_task_to_action(self, task: PlanningTask) -> Optional[UIAction]:
   """
     Step 1: Format Input Data
         - Convert actionable elements into a structured prompt.

     Step 2: Query AI Model
        - Send the subtask and webpage elements to an AI model to determine the best UI
        action.

     Step 3: Validate AI Response
         - Check if the model returns a valid action.
         - Handle errors or incomplete responses appropriately.

     Step 4: Return Action
         - If valid, return the action object.
         - Otherwise, return None.
     """
```

Python

```
class LowLevelAgent:
    system_prompt = …

   def execute(self, action):
       Step 1: Input
           - Takes an action object containing the type of action and any necessary
   parameters.
       Step 2: Perform Action
           - Identifies the action type and executes the corresponding UI interaction

   def move_mouse(self, x, y):
       Step 1: Input
           - Takes x and y coordinates.
       Step 2: Perform Action
           - Moves the mouse to the specified coordinates and clicks.

   def single_click_at_location(self, x, y):
       """
       Step 1: Input
           - Takes x and y coordinates.
       Step 2: Perform Action
           - Moves the mouse to the specified coordinates and performs a single click.
       """
```

Step 3: Behavioral Level – Defining Interactions and Protocols

At the **behavioral level**, we define how agents interact, coordinate, and execute workflows based on their defined functions. This level introduces decision-making, error handling, and task delegation mechanisms that allow agents to work together as a cohesive system. More than just a structural layer, the behavioral level is what gives the system its **intelligence**—enabling agents to perceive their environment, interpret signals, and make informed decisions that drive effective action. It serves as the glue that binds individual agent capabilities into a unified, adaptive, and goal-driven system.

By governing how agents **collaborate**, **make decisions**, and **communicate,** the behavioral level orchestrates task execution while ensuring adaptation to changing conditions. It empowers agents with **perception**—allowing them to process external inputs, monitor task progress, and respond dynamically to obstacles. Coordination mechanisms facilitate fluid interactions between agents, ensuring efficient task delegation and workload distribution. Intelligent error handling, such as automated retries, alternative strategy selection, or escalation protocols, enhances system resilience and reliability. Decision-making frameworks at this level guide agents in determining the best course of action based on **real-time feedback, learned experience**, and **predefined rules**.

For example, in a multi-agent system, a high-level agent may assess a failed subtask, analyze environmental factors, and determine whether to retry, reassign, or escalate the task. This ability to process information, adapt strategies, and optimize workflows in response to real-world conditions is what transforms a collection of agents into an intelligent, **self-organizing system**. The behavioral level is the critical foundation that enables agents to work not just as individual executors of tasks, but as an interconnected, perceptive, and autonomous network.

Designing the Agents Workflow

To build an effective multi-agent system, we start by designing the high-level agent workflow. This workflow provides a logical sequence of actions that agents will follow, ensuring they operate in a structured and coordinated manner. At this stage, we focus on defining the fundamental flow of tasks, establishing how agents interact with their environment, and determining how they execute

actions. The decision-making mechanisms and detailed orchestration logic will be incorporated in later stages.

The agent workflow consists of sequential steps that break down a user goal into actionable subtasks. Each step ensures that agents work systematically, progressing from goal interpretation to task execution. The key steps in this workflow include:

- **1. Generating a Subtask** – The system decomposes the user's goal into smaller, manageable subtasks.
- **2. Extracting Elements** – Relevant elements from the environment (e.g., web page elements, UI components) are identified and retrieved.
- **3. Assigning Elements** – The system assigns extracted elements to the appropriate agent for execution.
- **4. Executing Actions** – Actions such as mouse movements, clicks, and interactions with the web interface are performed.
- **5. Evaluation Loop** – After each action, the system evaluates its success and determines the next step.

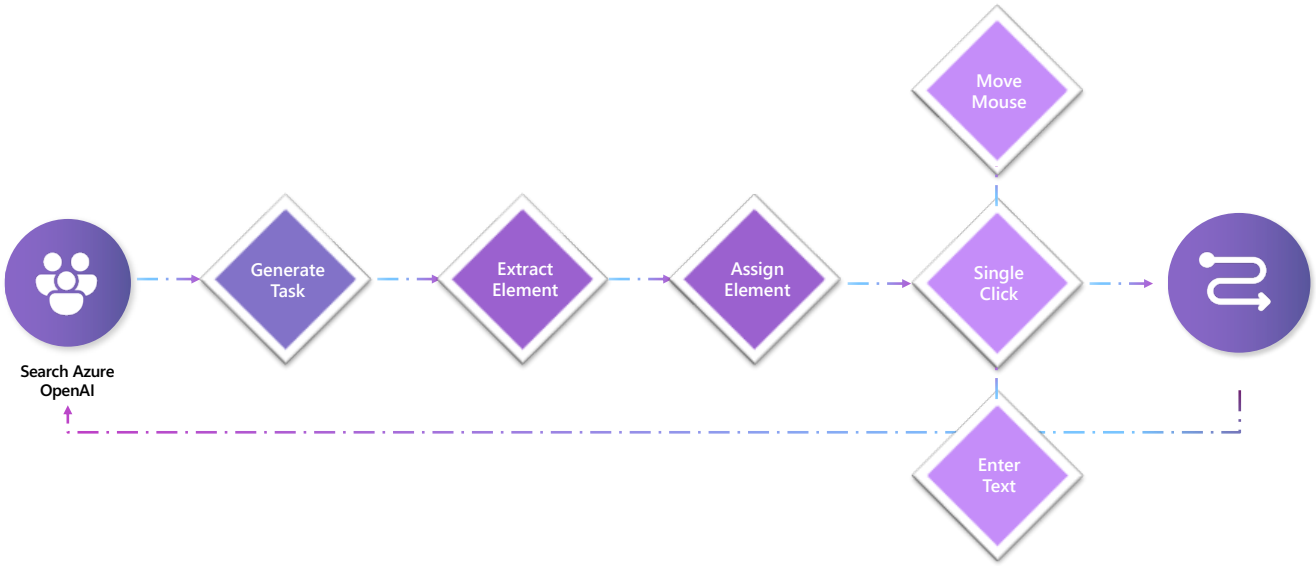

Figure 16. Agent Workflow

Designing the Agent Evaluation

A critical aspect of the workflow is the **evaluation loop**, which acts as the system's perception mechanism. After every action, the system assesses its outcome, verifying whether the intended task was successfully completed or if adjustments are required. This continuous feedback loop ensures that agents can adapt dynamically, detect failures, and refine their execution strategies.

For instance, if a single-click action does not yield the expected result, the system can decide whether to retry, switch to an alternative action, or escalate the issue. This loop is fundamental to the system's intelligence, allowing agents to learn from their environment, optimize workflows, and improve decision-making over time.

By designing this structured agent workflow, we establish a **foundation for orchestration**, ensuring that our multi-agent system operates efficiently, adaptively, and intelligently. The next step is to integrate **decision-making protocols** that will further enhance the system's ability to coordinate and execute complex tasks.

Designing the Agent Orchestration and Decision Logic

Orchestration is the backbone of our multi-agent system, ensuring that agents do not function as isolated executors of tasks but instead operate as a cohesive, intelligent, and adaptive system. It provides a structured framework for how agents coordinate, make decisions, and respond to dynamic environments.

The **Orchestration Flow** introduces decision-making checkpoints that enable agents to:

- Assess progress at each stage of task execution.
- Handle failures through retries, alternative strategies, or escalation mechanisms.
- Dynamically adjust execution based on environmental feedback.

By designing this flow, we establish clear transition states between key stages, including delegation, execution, evaluation, and replanning. Each step in the workflow is accountable, ensuring that the system remains responsive and adaptable to changing conditions.

Below we illustrate what this orchestration flow looks like for our Web Browser system.

Step 1: Task Initialization & Delegation (Web Coordinator)

The Web Coordinator receives the user goal and generates the first subtask to execute based on the first screenshot.

Decision Point: Does the subtask meet the objective?

✅ If the subtask is valid, execution proceeds.
❌ If there are missing components, the system requests adjustments.

Step 2: Medium-Level Execution (Browser Manager)

The Medium-Level Agent inspects the environment (current screenshot) and extracts actionable elements and assigns the proper element to the low-level agent.

Decision Point: Does the environment (screenshot) contain the required elements for the task?

✅ Yes → Assigns element to a subtask and sends them to the Low-Level Agent.
❌ No → Reports an error, triggering replanning.

Step 3: Low-Level Execution (Web Controller)

The Low-Level Agent executes the assigned action (clicking, typing, etc.). The system uses vision capable model to assess if the action met expectations by looking at the before and after screenshot of the web page.

Decision Point: Was the action executed successfully?

✅ Yes → Mark the task as completed.
❌ No → Generate a replanned subtask and retry execution

Step 4: Goal Completion Check (Web Coordinator)

The system evaluates all the steps in order to understand if the user's goal has been met.

Decision Point: Has the user's goal been achieved?

✅ Yes → Stop execution.

❌ No → Continue executing the next step in the plan.

Below is a diagram illustrating how the flow works with each decision point.

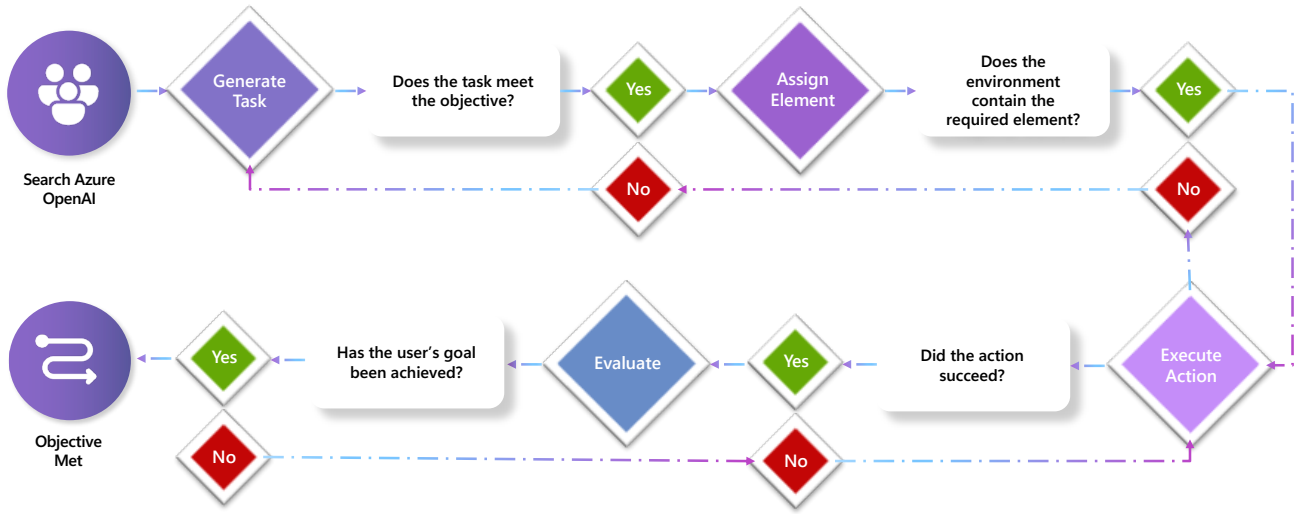

Figure 17. Agent Orchestration

By integrating orchestration and decision logic, we transform our system into an autonomous, perceptive, and self-adaptive framework, capable of handling complex workflows with efficiency and intelligence. The next step is refining the error-handling and learning mechanisms, further enhancing the agents' ability to adapt and improve over time.

Designing Agent Memory

At the behavioral level, **memory** serves as a fundamental framework for understanding how an agent system can learn, adapt, and **improve over time**. While memory operates at multiple levels, its behavioral function is particularly relevant for shaping system interactions and decision-making. Implementing memory in our agent system is incredibly powerful, making careful design crucial to ensure it enhances rather than hinders performance. Although this is not an in-depth exploration of memory, we will focus on **short-term** and **long-term memory**, emphasizing short-term memory in our web browsing solution to demonstrate its practical use.

Short-term memory manages temporary, task-specific information that aids immediate reasoning, while **long-term memory** retains more enduring knowledge to facilitate learning and adaptation.

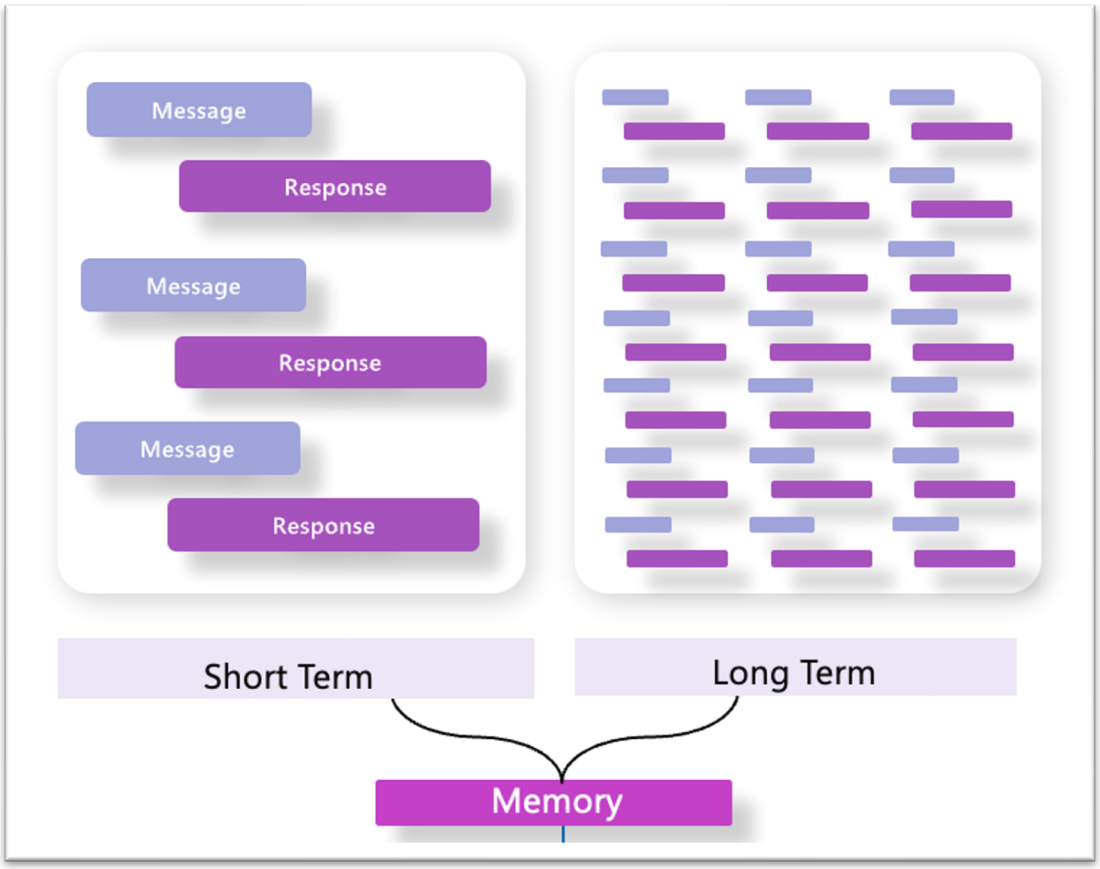

Figure 18. Short-Term and Long-Term Memory

For now, this foundational understanding of memory will enable us to design systems that balance immediate **responsiveness** with **long-term adaptability**. Memory serves as a valuable **historical reference**, enhancing decision-making, guiding us toward our overall objectives, and simplifying debugging and performance optimization.

In the following sections, we explore how short-term memory can be integrated at every level of our agent system. By leveraging short-term memory effectively, our agent can refine its responses, learn from immediate experiences, and maintain coherence over short timescales. This ensures smoother interactions, improved task performance, and a more natural user experience.

Memory in this Web Browser system is implemented through a **shared memory structure**, primarily represented by a timeline that persists events and actions performed by the agents. This memory serves multiple roles, such as tracking progress, assisting decision-making, and handling failures through iterative task refinement. Below is a breakdown of how memory is implemented at used at every agent level.

High-Level Agent (Web Coordinator) Memory

Purpose: The system uses memory to track **past actions and failures,** refining its strategy by learning from mistakes. It avoids redundant errors, adapts plans based on past attempts, and updates memory when generating tasks, detecting errors, or completing goals.

Medium-Level Agent (Browser Manager) Memory

Purpose: The system uses memory to understand task context, **analyze past successes and failures**, and assess previously identified UI elements. It updates memory when translating tasks into UI actions, encountering issues that require subtask adjustments, and logging extracted UI elements.

Low-Level Agent (Web Controller) Memory

Purpose: The system uses memory to determine the appropriate UI action and reference **past screenshots for comparison**. It updates memory when executing actions, capturing before-and-after screenshots, evaluating success, and detecting errors.

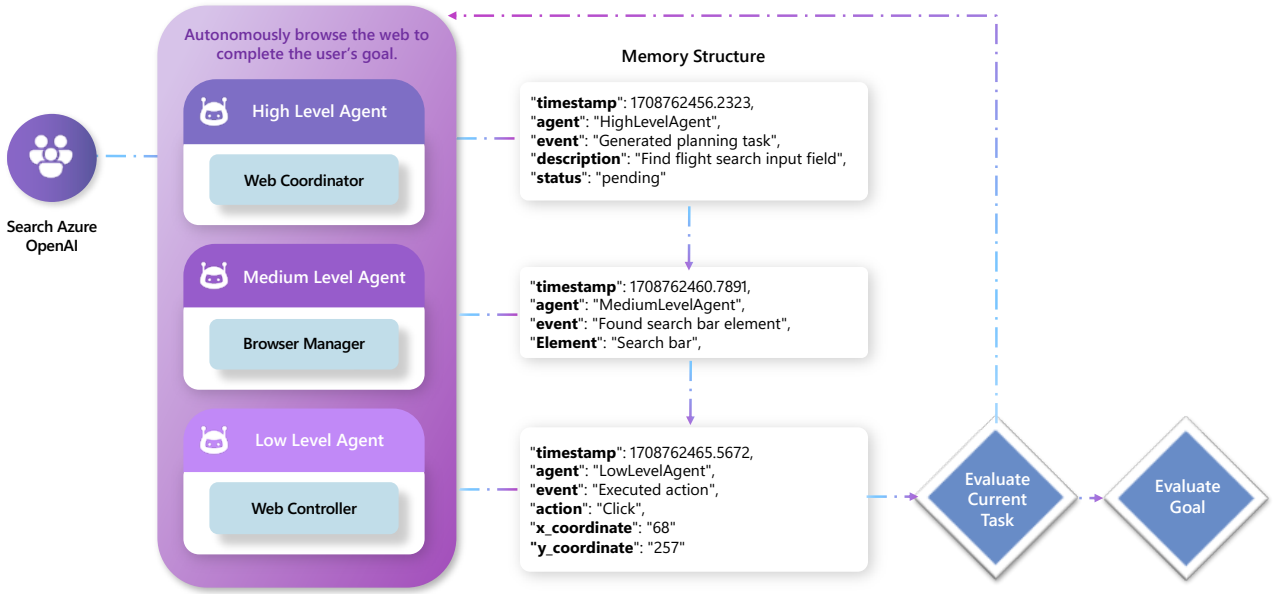

Figure 19. Short Term Memory Structure

By focusing on behavioral design and incorporating memory as a key enhancement, we establish a foundation for intelligent, adaptive multi-agent systems and a robust way for our agents to evaluate their actions. The structured roles of the High-Level, Medium-Level, and Low-Level Agents ensure collaboration, efficient task handling, and robust error recovery.

Designing Memory as a Communication Protocol

While orchestration and agent roles provide structure, true collaboration in a multi-agent system requires a shared context. In our system, memory itself can function as the communication protocol, serving as the central mechanism through which agents coordinate, synchronize, and evaluate progress toward their goals.

Rather than exchanging raw commands or ephemeral messages, agents in our Web Browsing system communicate by reading from and writing to a persistent shared memory structure. This memory acts as the central timeline of events, actions, and results, and serves as both a historical record and a live state reference for all participating agents.

Memory as a Communication Channel

Every agent interacts with the shared memory structure to persist its own state, read updates

from others, and make decisions based on collective progress. This removes the need for complex message-passing logic and ensures that all agent interactions are context-aware and synchronized.

- The **High-Level Agent** writes goals, task breakdowns, and outcome evaluations.
- The **Medium-Level Agent** logs UI contexts, tracks which sub-elements have been handled, and flags incomplete or problematic sections.
- The **Low-Level Agent** records screenshots, UI actions, and success/failure markers, enabling visual and behavioral comparison.

This design turns memory into a structured dialogue layer, where each agent contributes updates and consumes context in a readable, traceable way.

Memory-Driven Goal Completion

Another crucial benefit of using memory as a protocol is goal awareness. Because each agent logs its actions and results into a common memory, the system can infer task and goal completion by analyzing memory state. When all subtasks related to a parent goal are marked as complete, the system can autonomously determine that the overall objective has been achieved.

- This enables distributed progress tracking across agents.
- Failure recovery becomes easier, as memory reveals the last known successful state.
- Debugging and optimization benefit from a full audit trail of actions and decisions.

Consistency and Error Recovery

Using memory as a protocol ensures state consistency, especially in asynchronous or failure-prone environments. Agents can resume or refine tasks based on the most recent memory snapshot, improving reliability. For example, if a UI element was misidentified, that failure and context have persisted, allowing smarter re-attempts that avoid redundant errors.

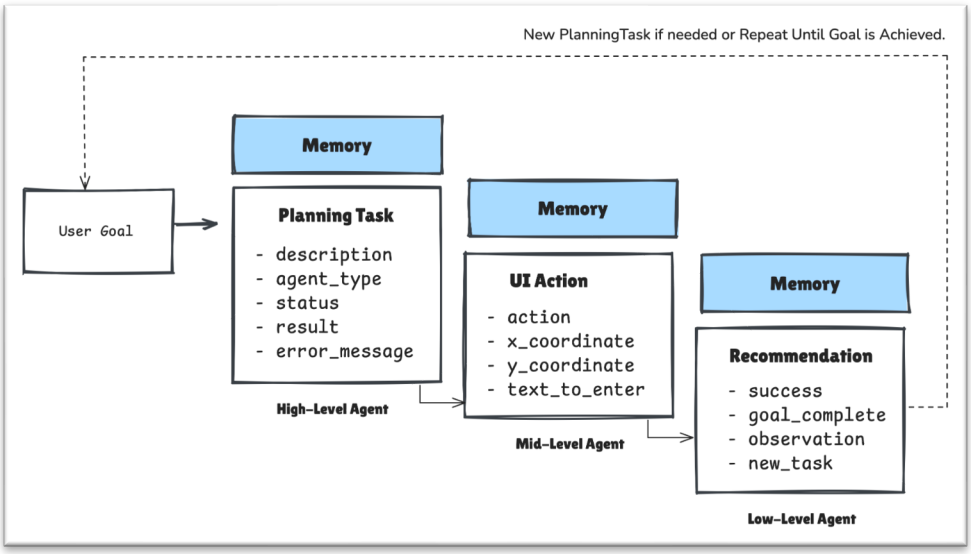

Figure 20. Agent Communication Protocol

Why This Matters

Behavioral design is the crucial link that transforms a set of independent agents into an intelligent, coordinated, and **adaptive system**. Without this layer, agents would operate in silos, unable to effectively collaborate, learn, or respond to dynamic challenges. In multi-agent design, behavioral principles define not just how agents execute tasks, but how they work together, resolve conflicts, handle errors, and adapt to new conditions.

This is particularly critical in autonomous web browsing systems, where agents must navigate complex and unpredictable digital environments. A robust behavioral framework ensures that agents can make real-time decisions, recover from failures, and optimize workflows without human intervention.

By incorporating **structured workflows, evaluation loops, decision checkpoints, and inter-agent communication with memory**, we create a system that is not only functional but also resilient and self-improving. The ability to dynamically adjust strategies, manage memory, and make informed decisions based on past experiences makes the difference between a rigid system and one capable of true autonomy. **Behavioral design** empowers multi-agent systems to function as cohesive, intelligent networks rather than disjointed executors of predefined tasks.

Summary

At the **behavioral level**, we move beyond static functions to define how agents interact, coordinate, and execute tasks in a structured and intelligent manner. This includes the implementation of workflows, decision logic, evaluation loops, memory, and inter-agent communication, ensuring that agents can collaborate effectively and adapt to dynamic environments.

Key takeaways from the behavioral design phase:

- **Structured Workflows:** Ensuring that agents follow a logical sequence of task delegation, execution, and evaluation.
- **Evaluation Loops:** Enabling agents to assess the success of their actions and make adjustments dynamically.
- **Decision Logic:** Incorporating intelligence into agents, allowing them to retry, reassign, or escalate tasks based on real-time feedback.
- **Memory Integration:** Providing agents with short-term memory to enhance contextual awareness and improve long-term adaptability.
- **Inter-Agent Communication:** Defining standardized messaging protocols to enable coordination and data exchange.

With a solid **behavioral framework** in place, the next step is translating these interactions into a concrete **technical design**. In this phase, we will focus on defining the system architecture, specifying data structures, function implementations, models and frameworks that bring everything to life. By bridging the gap between conceptual strategy and implementation, we will ensure that our multi-agent system is not only well-structured but also technically sound and scalable.

Technical Implementation

At the **behavioral level**, we established a structured orchestration mechanism that enables intelligent decision-making, adaptive workflows, and coordinated agent collaboration. The technical implementation must reflect this sophistication by ensuring that each agent operates within a cohesive, perceptive, and self-correcting system. To achieve this, the implementation

must incorporate orchestration, real-time evaluation loops, and robust inter-agent communication protocols.

For simplicity, we will focus on defining the logic and pseudocode for our **orchestrator** and **real time evaluation loop**, leaving the actual implementation to be addressed at the **technical level**. This pseudocode allows us to refine our design, validate the workflow, and ensure alignment before moving forward with actual implementation and development, reducing complexity and enhancing clarity in the early stages.

Orchestrator Function Description

Since the high-level agent is responsible for planning, it is logical for it to handle coordination as well, given its position at the top of our abstracted hierarchical design. The Orchestrator streamlines task execution by managing a Medium-Level Agent, which analyzes tasks, and a Low-Level Agent, which carries out actions. It ensures tasks are completed accurately, addressing failures through evaluation and dynamic replanning when necessary.

Python

```
Function orchestrate():
    """
    1. **Initialize Execution**:
       - Capture an initial screenshot for context.
       - Generate the first task using the initial prompt.

    2. **Main Execution Loop**:
       - While the goal is not achieved:
            a. Mark the current task as in progress.
            b. Capture a "before" screenshot.

            c. **Translate Task to UI Action**:
               - The Medium-Level Agent translates the task into a UI action.
               - Decision: Is the action valid?
                   - NO → Log error, generate a new task, and continue.
                   - YES → Proceed to execution.

            d. **Execute UI Action**:
               - The Low-Level Agent attempts to execute the UI action.
               - Decision: Was the execution successful?
                   - NO → Log error, generate a new task, and continue.
                   - YES → Capture an "after" screenshot.
```

```
            e. **Evaluate Execution**:
               - Compare the "before" and "after" screenshots.
               - Assess if the action produced the expected outcome.
               - Log observations and store memory.

            f. **Check Goal Completion**:
               - Decision: Has the user goal been met?
                   - YES → Log success and terminate execution.
                   - NO → Determine next task.
                       - If a recommended task is available, proceed with it.
                       - Otherwise, generate a new task based on observations.

    3. **Error Handling & Recovery**:
       - If failures persist despite replanning, log errors and adjust execution
strategy.
       - If a critical failure occurs, abort execution.

    4. **Final Output & Completion**:
       - Log final execution status.
       - Return structured results.
```

Evaluate Function Description

Since actions are executed at the Low-Level Agent, it is logical for this agent to handle evaluation as well, given its proximity to the execution process. The Evaluate function ensures task accuracy by assessing pre- and post-execution states, leveraging screenshots, task details, and user goals to determine success. By embedding evaluation at the lowest level, the system can promptly identify discrepancies and suggest corrective actions when needed. This feedback mechanism enhances adaptability, allowing dynamic adjustments and continuous improvement in execution. Additionally, the function logs evaluation results, contributing to iterative refinement and long-term system efficiency.

Python

```
Function evaluate_execution(task, pre_screenshot_path, post_screenshot_path, user_goal):
    """
    1. **Load Screenshots**:
       - Read the 'before' screenshot and convert it to base64.
       - Read the 'after' screenshot and convert it to base64.

    2. **Construct Evaluation Prompt**:
```

```
        - Create a structured evaluation request:
            - Include the task description.
            - Include the user goal.
            - Compare 'before' and 'after' screenshots to determine success.
            - Suggest a corrective action if necessary.
            - Expect output formatted as a Recommendation.

    3. **Send Evaluation Request**:
       - Format the system and user messages.
       - Submit the request to the evaluation model (e.g., GPT).

    4. **Process Evaluation Response**:
       - Decision: Was the response successfully received?
           - YES → Extract the recommendation details.
           - NO → Log error and return a failed recommendation.

    5. **Log Evaluation Outcome**:
       - Store the evaluation result in memory.
       - If successful, return the recommendation.
       - If an error occurs, log the failure and return a default failure response.

End Function
```

Step 4: Technical Level – Implementing the Details

Now, we turn our attention to the **technical implementation** of the agents, focusing on the **frameworks**, **tools**, **models**, **data structures**, and the actual implementation of our functions. This level transforms the abstract designs from earlier stages into a functional technical implementation.

At this stage, the system's design takes on a concrete form, emphasizing how **data flows** through the system, how it is represented and transformed, and the implementation details of our orchestrator that governs transitions and states and memory. Key considerations include selecting the appropriate models, tools, algorithms, and data structures, as these choices directly impact the system's performance and efficiency. Poorly chosen models, tools or frameworks can result in a system that is slow, error-prone, or fails to meet the demands of its tasks.

Choosing the Right Framework

With many **agent frameworks** available, selecting the right one is a critical decision. Key factors to consider include **speed of development**, **ease of integration**, and **long-term maintainability**. While some prefer established frameworks for their built-in support and structure, others—especially those with strong development teams—may opt to build their own for greater control and flexibility. Designing a framework from scratch allows for deep understanding and customization, aligning precisely with specific needs. Ultimately, the choice depends on personality, skill set, and development philosophy. However, many organizations adopt frameworks due to a lack of in-house expertise, the need for a supported, ready-to-use solution, and the urgency to accelerate time-to-market as designing your own framework from scratch can be time consuming.

For this implementation, I have chosen **Python** as the primary language, with **Pydantic** for defining structured data models and **Selenium and Pyautogui** as the controller library. These choices stem from my familiarity with these tools and my preference for building a framework that I am comfortable with and can tailor to my needs.

Figure 21. Frameworks

Choosing the Right Model

Selecting the right **model** is a key factor in shaping your multi-agent system's capabilities. With a wide range of models available, the best choice depends on the specific tasks your agents need to perform.

In our web browser example, the model must support reasoning, structured output, and both language and vision capabilities. Fortunately, a model like GPT-4o meets all these requirements, providing a powerful foundation for agents that need to interpret and interact with web-based environments efficiently.

For more specialized agents, a **task-specific model** or a **fine-tuned model** may be necessary, or a smaller model might be preferable for cost efficiency. Combining multiple models within your system can enhance performance, allowing agents to leverage different strengths and optimize both accuracy and resource usage.

Data Structure Implementation

Structured data is essential for ensuring consistency, clarity, and reliability in our multi-agent system. By defining clear data structures, we enable agents to interpret and exchange information in a standardized way, reducing errors and improving coordination. For this, we will use **Pydantic structures**, which provides an intuitive and efficient way to define and validate structured data. Pydantic's ease of use and compatibility with our model make it a natural choice. Since our structured plan serves as the primary communication protocol between agents, maintaining well-defined data structures is crucial for collaboration and execution.

Let's define a few key Pydantic structures to provide better clarity. For further details, please refer to the GitHub repository linked below.

```python
class PlanningTask(BaseModel):
    id: str
    description: str
    agent_type: str  # e.g., "planner" or "user"
    status: TaskStatus = TaskStatus.pending
    result: Optional[List[dict]] = None
    error_message: Optional[str] = None
```

```python
class UIAction(BaseModel):
    action: str
    x_coordinate: int
    y_coordinate: int
    text_to_enter: Optional[str] = None
    key: Optional[str] = None
    end_x: Optional[int] = None
    end_y: Optional[int] = None
```

```python
class Recommendation(BaseModel):
    success: bool
    goal_complete: bool
    observation: str
    new_task: Optional[PlanningTask] = None
```

```python
class MemoryEvent(BaseModel):
    timestamp: float, agent: str, event: str
    task_id: Optional[str] = None
    ui_action: Optional[Dict[str, Any]] = None
    observation: Optional[str] = None
    error: Optional[str] = None
    path: Optional[str] = None
```

Orchestration Implementation

This function runs a loop that directs an automated agent system to complete a UI-based task step by step. It starts by taking an initial screenshot to capture the current UI state and then generates a task based on a predefined prompt. In each loop cycle, the agent marks the task as "in progress," takes a screenshot before making changes, and uses a Medium-Level Agent to convert the task into a specific UI action. If no suitable action is found, an error is logged, and a new task is generated. When a valid action is identified, a Low-Level Agent executes it and checks if it was successful. If execution fails, another error is logged, and a new task is created. After the action, a second screenshot is taken, and the Low-Level Agent evaluates the change by comparing the before-and-after images. If the goal is met, execution stops; otherwise, a new task is planned based on the **evaluation**. The loop continues until the agent successfully **completes the overall goal**, ensuring an iterative and adaptable approach to UI automation.

Python

```python
def run(self) -> None:
        """
        Main orchestration loop.
        """
```

```
        # Capture an initial screenshot for context.
        init_screenshot = "screenshots/initial.png"
        self.low_agent.capture_screenshot(init_screenshot)

        # Generate the initial planning task.
        current_task = self.generate_task(self.INITIAL_PROMPT_TEMPLATE)

        while True:
            print(f"HighLevelAgent: Executing task: {current_task.description}")
            current_task.status = TaskStatus.in_progress

            # Capture a "before" screenshot.
            pre_screenshot_path = f"screenshots/{current_task.id}_before.png"
            self.low_agent.capture_screenshot(pre_screenshot_path)

            # Translate the planning task to a concrete UI action.
            ui_action = self.medium_agent.run(current_task)
            if ui_action is None or ui_action.action == "change_subtask":
                error_info = "No appropriate UI element found; changing subtask."
                self.memory.append(MemoryEvent(
                    timestamp=time.time(),
                    agent="HighLevelAgent",
                    event="Translation failure/change_subtask",
                    task_id=current_task.id,
                    error=error_info
                ))
                current_task = self.generate_task(self.NEXT_PROMPT_TEMPLATE, error_info)
                continue

            # Execute the UI action.
            execution_result = self.low_agent.run(ui_action)
            if not execution_result.get("success"):
                error_info = execution_result.get("error_message", "Unknown execution error.")
                self.memory.append(MemoryEvent(
                    timestamp=time.time(),
                    agent="HighLevelAgent",
                    event="Execution error",
                    task_id=current_task.id,
                    error=error_info
                ))
                current_task = self.generate_task(self.NEXT_PROMPT_TEMPLATE, error_info)
                continue

            # Capture an "after" screenshot.
            post_screenshot_path = f"screenshots/{current_task.id}_after.png"
            self.low_agent.capture_screenshot(post_screenshot_path)

            # Evaluate the execution using both screenshots.
```

```
            recommendation = self.low_agent.evaluate_execution(current_task,
pre_screenshot_path, post_screenshot_path, self.user_goal)
            print("HighLevelAgent Observation:", recommendation.observation)
            self.memory.append(MemoryEvent(
                timestamp=time.time(),
                agent="HighLevelAgent",
                event="Evaluation completed",
                task_id=current_task.id,
                observation=recommendation.observation
            ))

            # Check if the goal is complete.
            if recommendation.goal_complete:
                self.memory.append(MemoryEvent(
                    timestamp=time.time(),
                    agent="HighLevelAgent",
                    event="Goal achieved",
                    task_id=current_task.id
                ))
                print("HighLevelAgent: Goal achieved!")
                break

            if recommendation.new_task:
                current_task = recommendation.new_task
            else:
                current_task = self.generate_task(self.NEXT_PROMPT_TEMPLATE,
recommendation.observation)

        print("HighLevelAgent: Execution complete.")
```

Evaluation Implementation

This function evaluates the outcome of a UI action by comparing **"before" and "after" screenshots** to determine if the intended result was achieved. It first reads both screenshots, encodes them in base64, and constructs a prompt that includes the task description and user goal. This data is then sent to an AI model, which analyzes the differences between the images and returns a structured recommendation in JSON format. If the evaluation is successful, the recommendation (which may include feedback on whether the goal is complete or suggest next steps) is logged into memory. If an error occurs, it logs the failure and returns a default response indicating that the evaluation failed. The run function separately handles the actual execution of UI actions by passing them to a lower-level execution method, ensuring that task evaluation and

execution remain distinct but interconnected processes.

```
def evaluate_execution(self, task: PlanningTask, pre_screenshot_path: str,
post_screenshot_path: str, user_goal: str) -> Recommendation:
        """
        Evaluate the outcome of an executed action using both 'before' and 'after'
screenshots.
        """
        with open(pre_screenshot_path, "rb") as pre_file:
            pre_base64 = base64.b64encode(pre_file.read()).decode("utf-8")
        with open(post_screenshot_path, "rb") as post_file:
            post_base64 = base64.b64encode(post_file.read()).decode("utf-8")
        prompt = (
            f"Evaluate the execution of the task '{task.description}'.\n"
            f"User goal: '{user_goal}'.\n"
            "Compare the before and after screenshots to determine whether the
intended outcome has been achieved. "
            "If not, suggest a generic corrective next step. "
            "Return a JSON object matching the Recommendation model."
        )
        messages = [
            {"role": "system", "content": self.SYSTEM_PROMPT_CONTROLLER},
            {"role": "user", "content": [
                {"type": "text", "text": prompt},
                {"type": "image_url", "image_url": {"url":
f"{pre_base64}"}},
                {"type": "image_url", "image_url": {"url":
f"{post_base64}"}}
            ]}
        ]
        try:
            completion = openai.beta.chat.completions.parse(
                model="GPT4o",
                messages=messages,
                response_format=Recommendation
            )
            recommendation: Recommendation = completion.choices[0].message.parsed
            print("LowLevelAgent: Evaluation result:", recommendation)
            self.memory.append(MemoryEvent(
                timestamp=time.time(),
                agent="LowLevelAgent",
                event="Evaluated execution",
                task_id=task.id,
                observation=recommendation.observation
            ))
            return recommendation
        except Exception as e:
```

```
            print("LowLevelAgent: Evaluation error:", e)
            self.memory.append(MemoryEvent(
                timestamp=time.time(),
                agent="LowLevelAgent",
                event="Evaluation error",
                task_id=task.id,
                error=str(e)
            ))
            return Recommendation(
                success=False,
                goal_complete=False,
                observation="Evaluation failed.",
                new_task=None
            )

    def run(self, action: UIAction) -> dict:
        """
        Execute the given UI action.
        """
        return self.execute(action)
```

Generate Task Implementation

This function generates a new planning task by leveraging an AI model to determine the next step toward achieving the user's goal. It first compiles relevant context, including past memory logs and any additional observations, into a structured prompt based on a given template. This prompt is then sent to an AI model, which interprets the information and returns a **structured PlanningTask**. If the task is successfully generated, it is logged into memory and returned for execution. If an error occurs during task generation, the function logs the failure and returns a default fallback task to ensure progress continues. This approach allows the system to dynamically plan tasks, adjust based on past execution results, and ensure the overall goal is met efficiently.

Python

```
def generate_task(self, prompt_template: str, extra_info: Optional[str] = None) ->
PlanningTask:
        """
        Generate a planning task via an LLM call.
        """
        memory_str = json.dumps([e.dict() for e in self.memory])
        prompt = prompt_template.format(
```

```
            user_goal=self.user_goal,
            observation=extra_info or "",
            memory=memory_str
        )
        messages = [
            {"role": "system", "content": "You are a planner agent that creates a
structured plan to achieve the user's goal by determining tasks and their order. It
assigns tasks to the right agents to ensure effective goal completion"},
            {"role": "user", "content": prompt}
        ]
        try:
            completion = openai.beta.chat.completions.parse(
                model="GPT4o",
                messages=messages,
                response_format=PlanningTask
            )
            task: PlanningTask = completion.choices[0].message.parsed
            self.memory.append(MemoryEvent(
                timestamp=time.time(),
                agent="HighLevelAgent",
                event="Generated planning task",
                task_id=task.id,
                ui_action=task.dict()
            ))
            print("HighLevelAgent: Generated task:", task)
            return task
        except Exception as e:
            print("HighLevelAgent: Error generating task:", e)
            self.memory.append(MemoryEvent(
                timestamp=time.time(),
                agent="HighLevelAgent",
                event="Task generation error",
                error=str(e)
            ))
            return PlanningTask(
                id="fallback",
                description="Perform a default UI action to progress toward the
goal.",
                agent_type="planner"
            )
```

Web Controller (Click, Move Mouse)

These functions handle basic mouse interactions using **PyAutoGUI**, enabling the agent to move the cursor and perform clicks at specific locations. **The move_mouse(x, y)** function moves the cursor to the given coordinates, clicks, and introduces a 2-second delay to ensure smooth execution. Similarly, **single_click_at_location(x, y)** moves the cursor and clicks but also prints a debug message to indicate execution. These functions allow Low-Level Agents to interact with UI elements accurately, ensuring reliable execution of actions like clicking buttons, selecting links, or engaging with input fields. By simulating human-like interactions, they facilitate automation in web and GUI-based environments.

Python

```
def move_mouse(self, x, y, **kwargs):
        pyautogui.moveTo(x, y)
        pyautogui.click()
```

Python

```
def single_click_at_location(self, x, y, **kwargs):
        pyautogui.moveTo(x, y)
        pyautogui.click()
```

Summary of the Step-by-Step Guide

The step-by-step guide presented a **structured approach** to designing a multi-agent web controller system capable of autonomously navigating the web to complete user tasks. By applying **conceptual design**, the system was broken down into three agent levels—Planner (high-level), Controller (mid-level), and Executor (low-level)—each with defined roles and responsibilities. The **functional design** assigned concrete tasks to each agent, ensuring efficiency and specialization, while the behavioral level focuses on orchestration, communication protocols, and decision-making. Finally, the **technical design** provides implementation details using Python, Pydantic, Selenium, Pyautogui to bring the system to life. The use of structured plans, error-handling

mechanisms, and memory integration made this multi-agent system adaptable, scalable, and capable of tackling complex web automation tasks.

Designing a multi-agent web controller system requires a careful balance of **abstraction**, **modularity**, and intelligent **orchestration**. By structuring agents into a hierarchical framework, defining clear roles, and implementing a robust orchestration flow, this approach ensures coordination, error recovery, and adaptability. The guide highlights the importance of structured communication, AI-driven decision-making, and modular design in building autonomous systems that can efficiently interact with the web. With the right design principles and implementation strategies, we have designed a multi-agent system that can navigate and automate web-based tasks, unlocking new possibilities for automation and intelligent browsing.

For a concrete example of a web browsing multi-agent system, please refer to the accompanying **GitHub repository**. This implementation, while functional, is intended primarily as a starting point—a template that reflects the principles we've discussed, including abstraction, modularity, and key design patterns. It offers a foundation on which you can build, adapt, and extend. We encourage readers to dive in, experiment, and improve upon it. There is ample opportunity to optimize and innovate, and we challenge you to push the design further.

https://github.com/slacassegbb/WebBrowser_Example

Conclusion

By understanding how we design complexity, can we uncover the hidden patterns that shape existence—and, in doing so, reveal the true nature of intelligence itself.

The success of multi-agent systems lies not in reducing complexity, but in shaping it into something **structured**, **scalable**, and **intuitive**. Throughout this white paper, we explored how **abstraction** and **modularity** serve as foundational principles in designing multi-agent solutions, enabling systems that are both adaptive and efficient.

By leveraging **modular design**, we ensure that individual agents remain specialized and replaceable, promoting flexibility and fault tolerance. Through hierarchical abstraction, we enable agents to function at different levels—**conceptual**, **functional**, **behavioral**, and **technical**—each contributing to a cohesive whole without unnecessary complexity. The step-by-step approach demonstrated how these principles apply to real-world applications, from drone-based wildfire management to autonomous web browsing, illustrating their scalability across domains.

However, effective design is only the beginning. The future of multi-agent systems will demand advancements in real-time decision-making, memory integration, and adaptive learning. Systems that can dynamically adjust their abstraction levels—learning when to generalize and when to focus on details—will push the boundaries of what **autonomous agents** can achieve. Moreover, the challenge of orchestrating large-scale agent collaboration in unpredictable environments presents a frontier for research and innovation.

As we move forward, the focus should shift from simply building intelligent agents to creating intelligent interactions between agents. The true power of multi-agent systems is not just in their individual capabilities, but in their ability to coordinate, negotiate, and self-correct, mirroring the way teams, organizations, and even biological ecosystems operate.

By mastering these principles, we can develop multi-agent architectures that are not only powerful but also trustworthy, scalable, and seamlessly integrated into real-world applications. The journey toward building intelligent, autonomous systems is far from over—but by refining abstraction, modularity, and orchestration, we can design solutions that are not just functional, but transformational.

Endnotes

1. Charles Mingus, attributed, widely quoted in discussions on creativity and jazz composition.
2. Marvin Minsky, The Society of Mind (1986).
3. Alan Kay. (1993). Quoted in Computers and Cognition: Why Minds are Not Like Computers by James H. Fetzer.
4. Edsger Dijkstra, On the Role of Scientific Thought (1982).
5. Donald Norman, The Design of Everyday Things (1988).