

A Comprehensive Guide to Designing Multi-Agent Systems

Author: Simon Lacasse (Microsoft)



Table of Contents

<i>Introduction</i>	3
<i>Modular Design and Abstraction</i>	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
<i>A Step-by-Step Guide to Designing a Multi-Agent Web Controller System</i>	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	33
Designing Agent Function	34
Step 3: Behavioral Level – Defining Interactions and Protocols	39
Designing Agent Workflow, Evaluation, Orchestration, Memory, Communication	40-47
Step 4: Technical Level – Implementing the Details	51
Models, Frameworks, Implementation Details	40-47
<i>Conclusion</i>	63
<i>Endnotes</i>	62

Introduction

“Making the simple complicated is commonplace, making the complicated simple, awesomely simple, that’s creativity.” - Charles Mingus¹

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 designed 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 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 Minsky²

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 and modular design**, 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 Kay³

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.

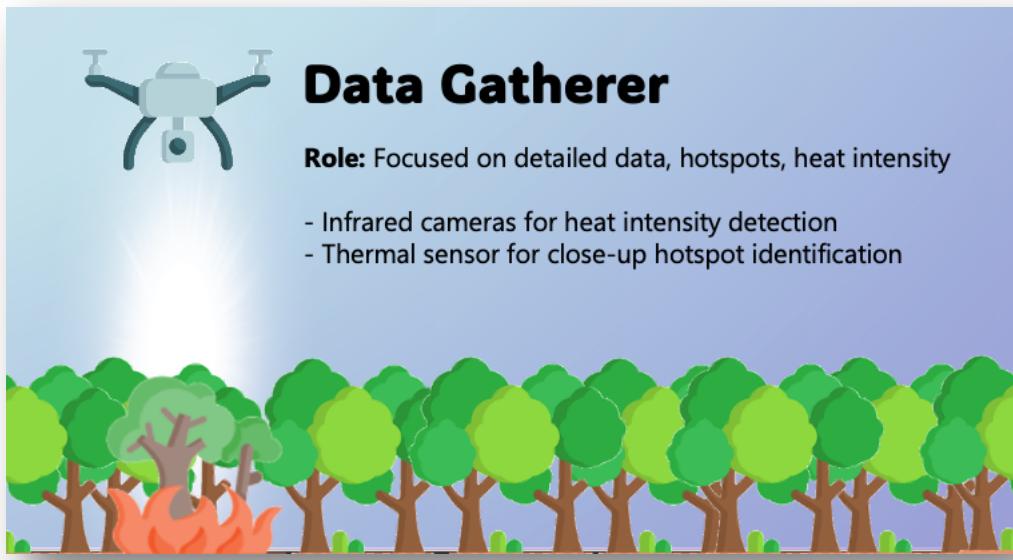


Figure 1. Data Gatherer Drone (Modularity)

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.

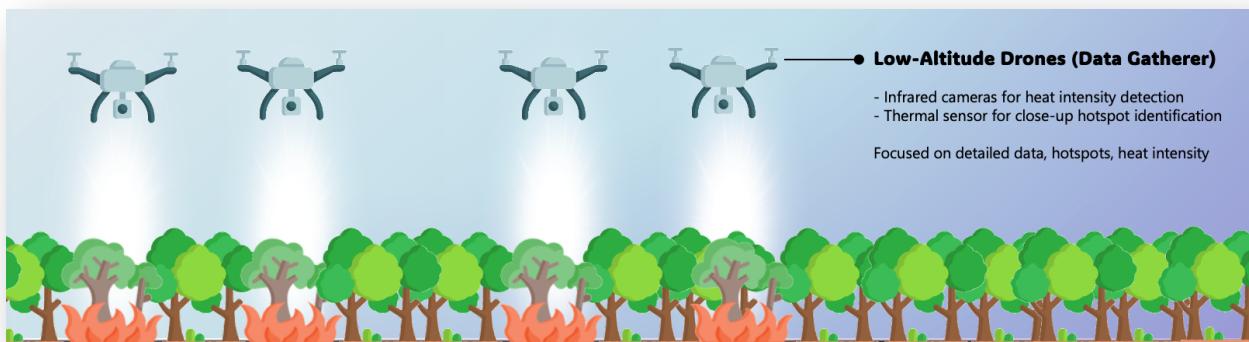


Figure 2. Low Altitude layer (Abstraction)

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—if we were to implement enhancements such as collision avoidance algorithms, they would need to be manually installed on each individual drone, leading to increased maintenance costs and reduced operational efficiency.

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 coordinator 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.

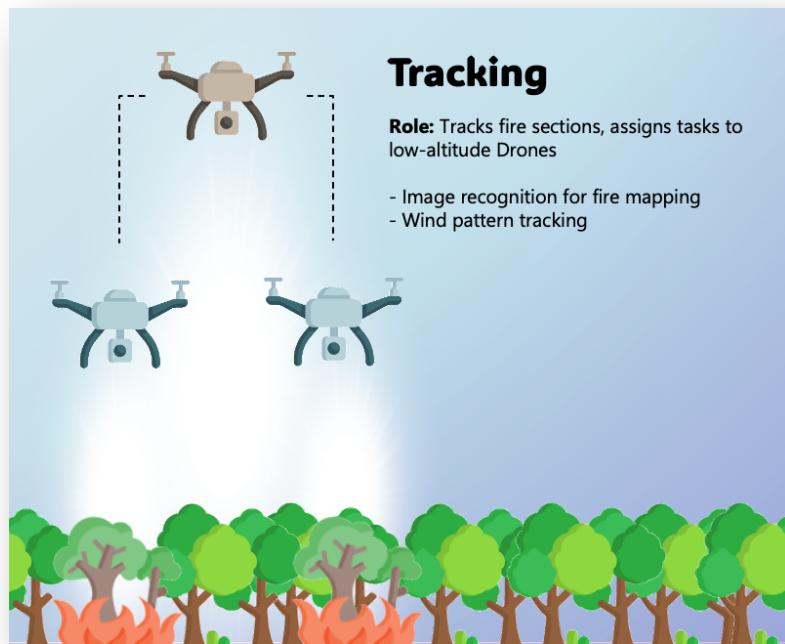


Figure 3. Tracking Drone (Modularity)

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.

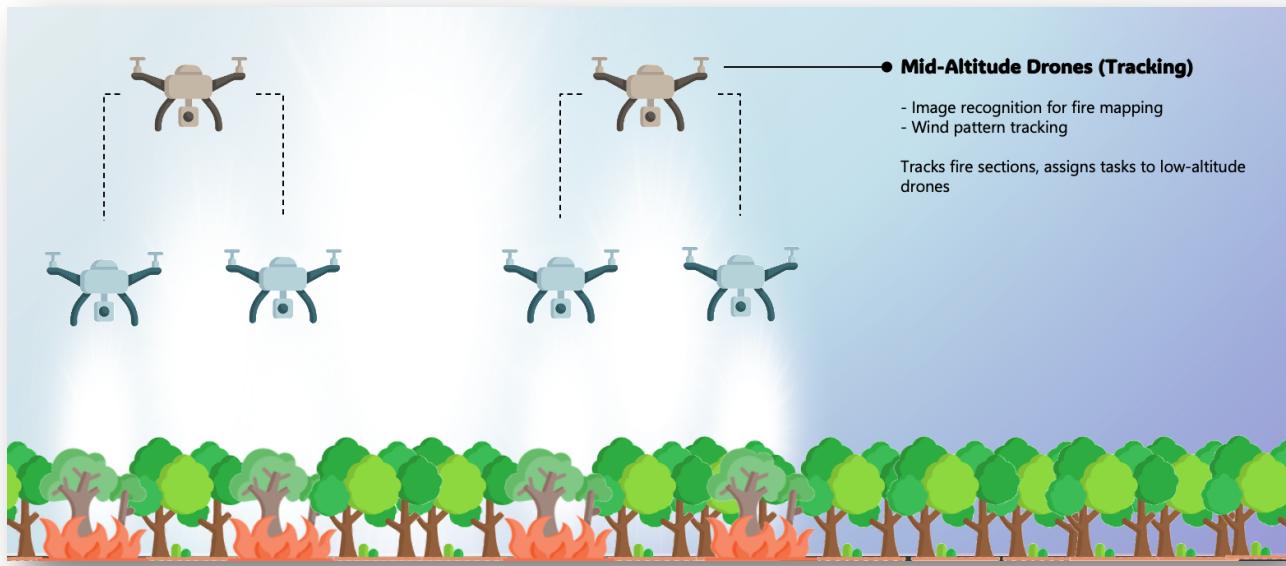


Figure 4. Mid-Altitude layer (Abstraction)

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.

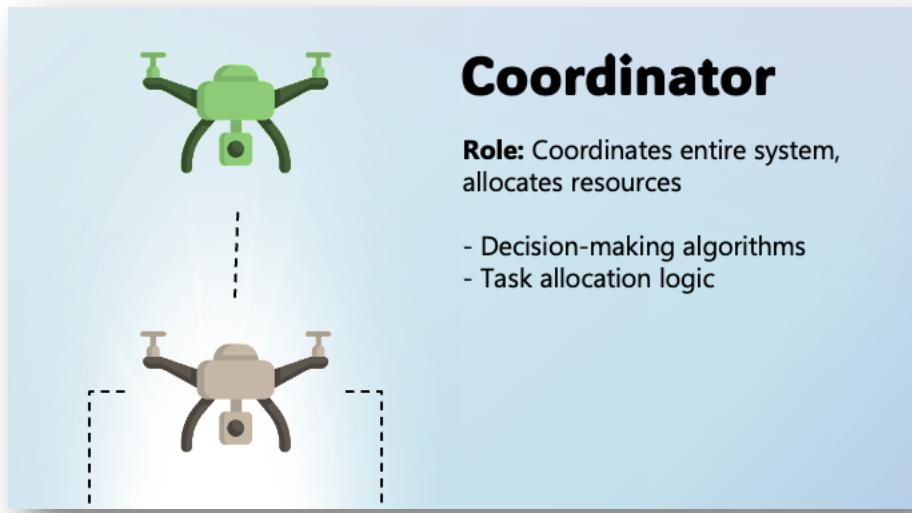


Figure 5. Coordinator Drone (Modularity)

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, while also remaining responsive to input from human operators who can adjust their course of action in real time. 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.

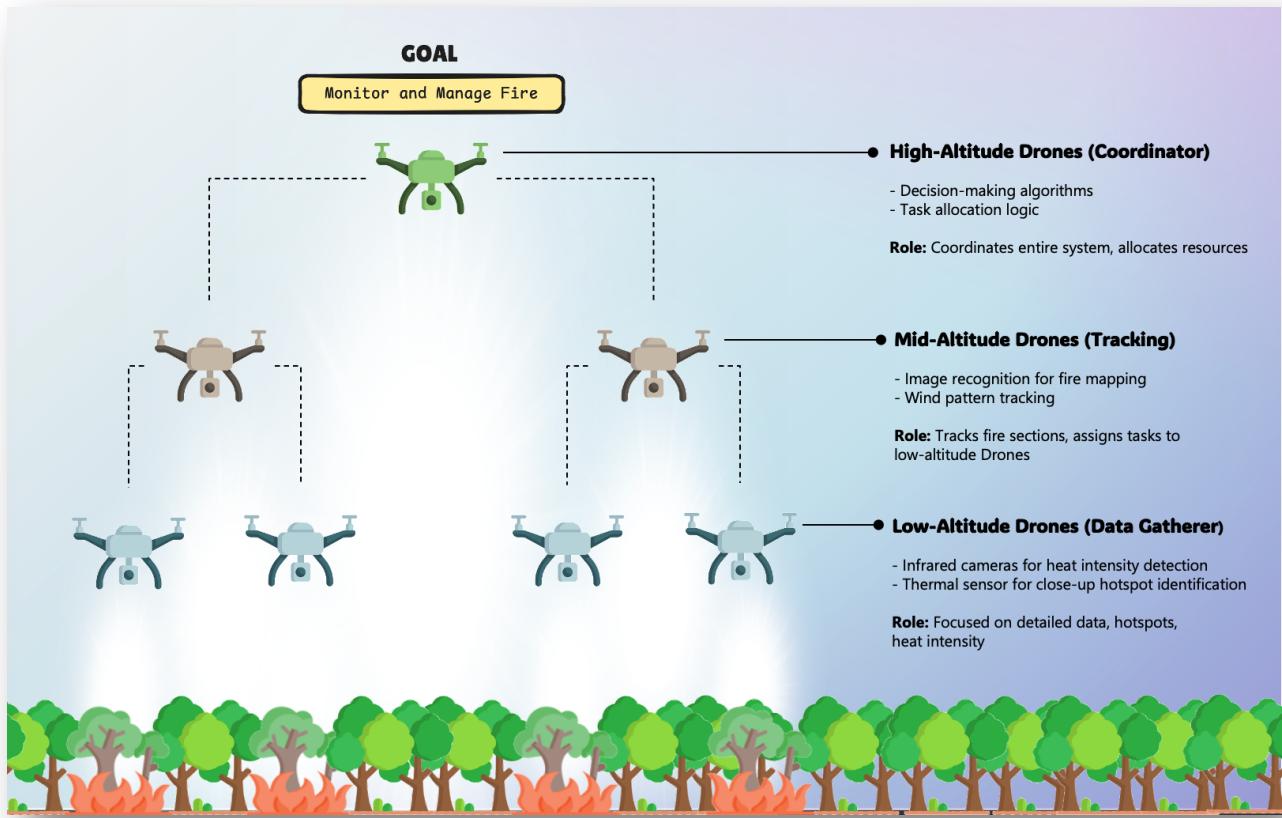


Figure 6. High Altitude layer (Abstraction)

Why it Matters

While all drones share the **common goal** of monitoring and managing wildfires, they each operate 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 a single 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. We propose a methodology for design complex multi-agent systems by applying **conceptual**, **functional**, **behavioral**, and **technical** abstraction. This methodology helps 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 Dijkstra⁴

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. To address this, we leverage abstraction again, but this time, we apply it horizontally during the design phase at every level of our hierachal 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, adaptability and decision-making among agents.
- **Technical Abstraction** – Focuses on the implementation details, optimizing performance and integration.

We can use this iterative, structured abstraction approach in a top-down manner while also incorporating bottom-up elements, allowing us to effectively balance both design strategies. 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 methodology, abstraction operates at different levels—**conceptual, functional, behavioral**, and **technical**—which can really help us design multi-agent systems in a structured way and manage complexity effectively.

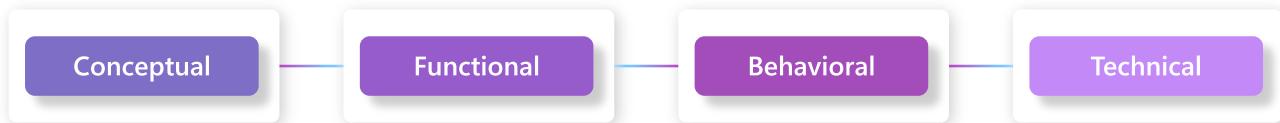


Figure 7. Levels of Abstraction – Conceptual, Functional, Behavioral and Technical

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.

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 they 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.

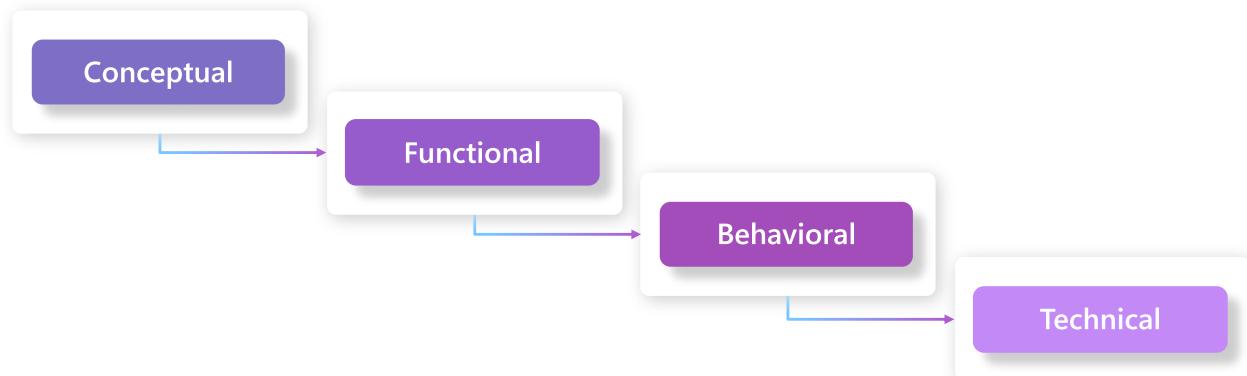


Figure 8. Top-Down Approach

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.

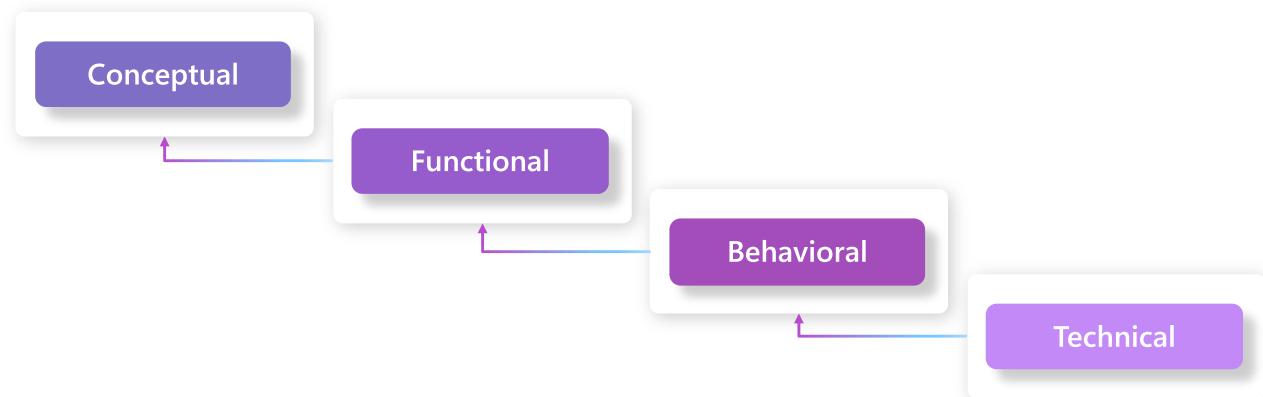


Figure 9. Bottom-Up Approach

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 design efficient drone operations in wildfire management.

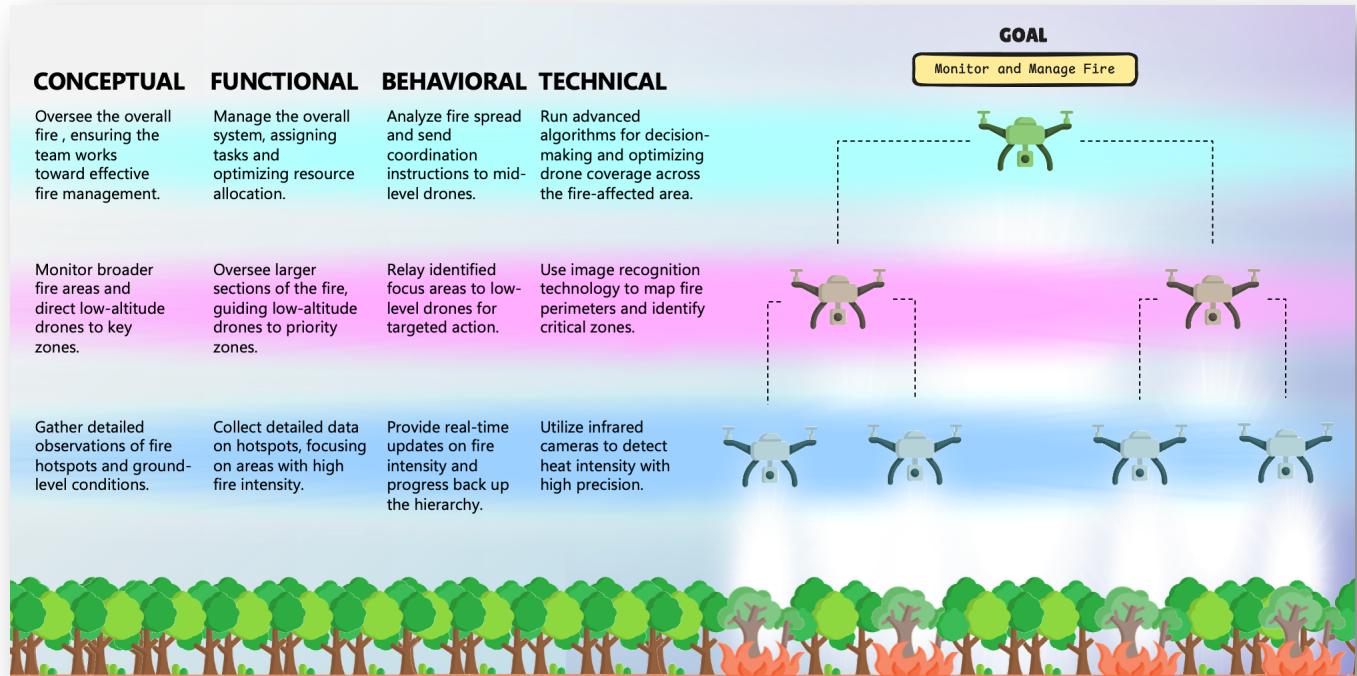


Figure 11. Conceptual, Functional, Behavioral and Technical Design (Drone Example)

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 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 design approach, we risk creating systems that are brittle, inefficient, and difficult to scale. By applying—**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 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 Norman⁵

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.

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 methodology 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.

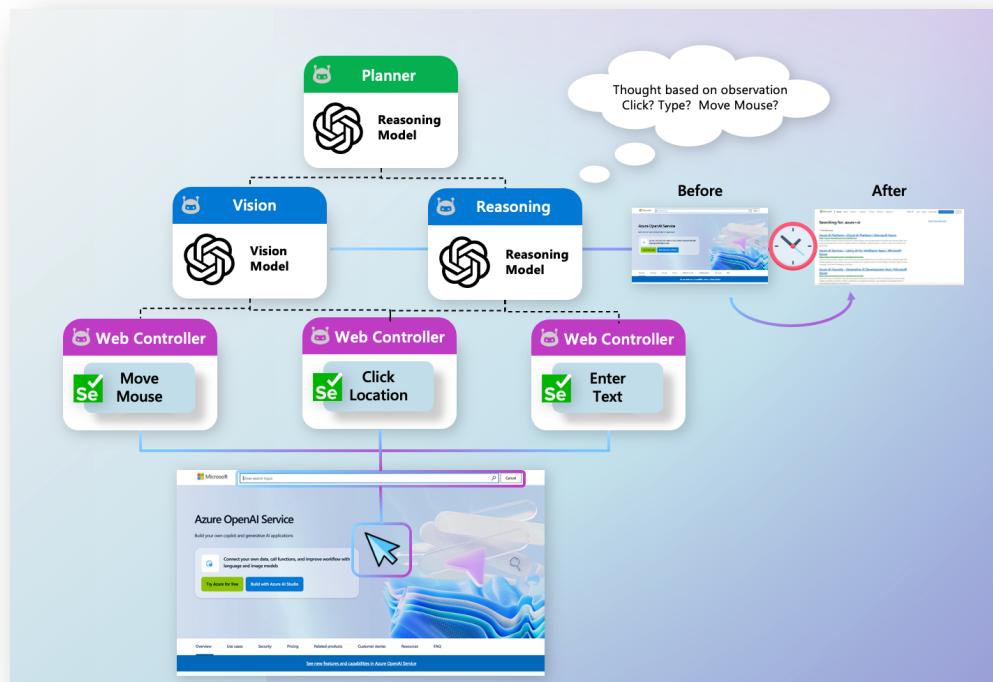


Figure 12. Multi-Agent Web Controller High Level Architecture

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. Let's start with the conceptual design of our multi-agent web controller system by outlining its overall goals, key agent roles, and the high-level interactions required to coordinate their behavior.

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 begin 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 user'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.

Now that we have defined the overall goal of the system, the next step is to decompose it into essential components—this essentially boils down to identifying which agents are needed to accomplish our goal and what roles they will play within the system.

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. This prevents role overlap and ambiguity while ensuring that each agent operates within a well-organized hierarchy.

Based on these roles, we can clearly define three agents that make up our system: a **Web Coordinator Agent** responsible for orchestrating tasks, a **Web Browsing Agent** focused on parsing and analyzing web content, and a **Web Controller Agent** that performs actions directly on webpages. Clearly defining our agent roles helps translate abstract responsibilities into concrete, modular components. By identifying these specific responsibilities, we clarify the boundaries and interactions among agents. This structured approach ensures that each agent contributes uniquely and effectively, optimizing the overall efficiency and maintainability of the system.

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 UI 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

Now that we have identified our modular agents, the next step is to define our abstraction topology—organizing these agents into layers based on their level of decision-making and control. Just as in our top-down drone system—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 top-down hierarchical abstraction to our web browsing system. This structure enables clearer coordination, better scalability, and more efficient management of complexity across the system.

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 and adaptability of our web controller agent system by enabling agents to work together while maintaining modularity and flexibility.

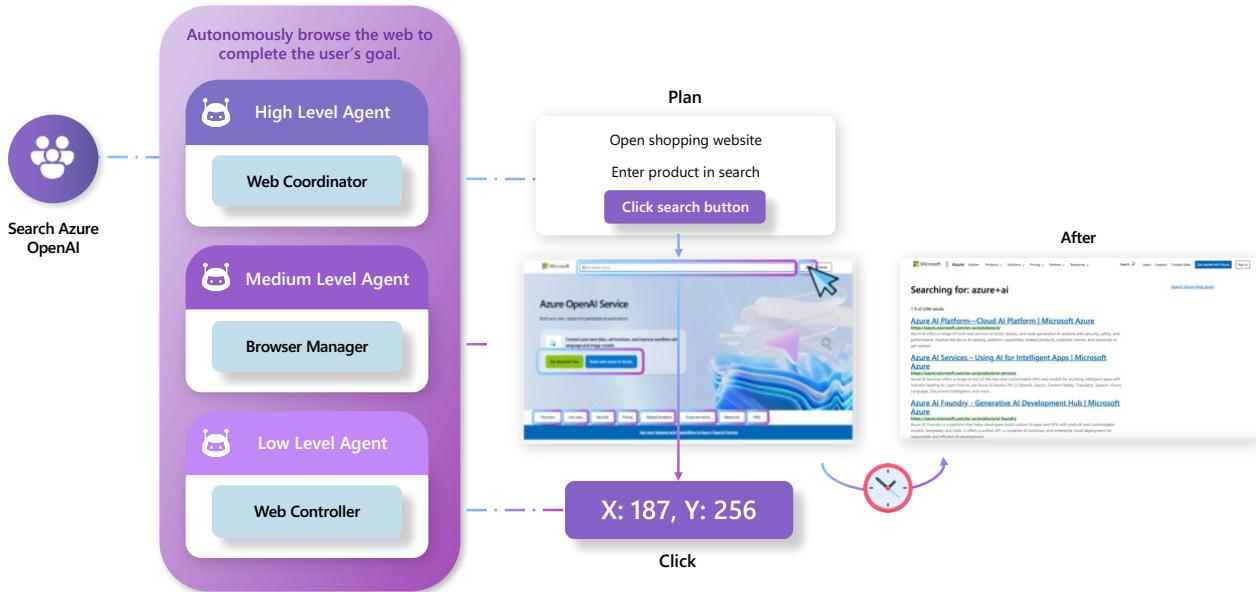


Figure 13. Conceptual Design – Modular and Abstraction

Designing Agents with Objectives

Once the roles, agents, and abstraction topology are established, the next step is to assign **objective functions**—measurable criteria that guide system performance and agent behavior. These functions act as guiding principles, evaluating each agent's effectiveness, informing its decision-making, and ensuring alignment with the system's overall goals. By first defining the roles and agents, we lay a solid foundation that allows objective functions to be precisely tailored to each agent's responsibilities. This ensures that every agent contributes meaningfully to the system's mission while maintaining a sense of **autonomy** and **purpose** within its respective layer.

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.

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 the **Functional Level** 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. The **system prompts** can be used here to 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

Now that we have conceptually designed our overall system, we move on to the **functional level**. Here, 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. At this stage, we are not yet concerned with how these functions interact; rather, the focus is on ensuring that each agent has a clearly defined set of operations that support its role in the system. This clarity helps prevent inefficiencies or breakdowns during execution by establishing well-defined functional responsibilities for each agent.

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 and the actions that they will take. These functions establish the **core capabilities** of each agent and provide the foundation for each of our agents.

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.

- **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 illustrated below in our diagram.

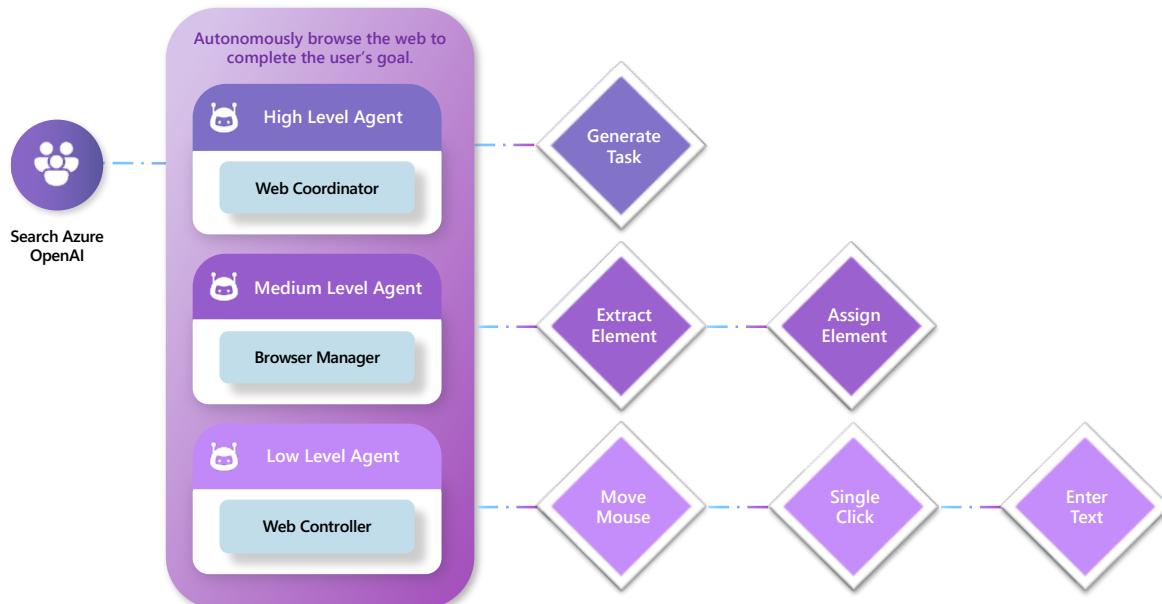


Figure 15. Agent Functions

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 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 functional execution**. Each agent now has a well-defined set of functions, ensuring that the system is structured, and actionable at each abstraction layer.

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 functions for each of our agents.

We can extend our classes from the conceptual design with the necessary functions needed for each agent. 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

Now that we have conceptually designed our hierarchical agents and created the necessary modular functions for each one, we move on to the **behavioral design** of our agents. This level introduces the logic that governs how agents **interact**, **coordinate**, **orchestrate** and **adapt** their environment. It encompasses **decision-making**, **error handling**, and **task delegation**—enabling agents to operate not just independently, but **collaboratively** as part of an intelligent system.

The **behavioral level** is more than just an integration layer; it is what brings the system to life. Here, agents gain the ability to **perceive** their environment, interpret contextual signals, and make informed choices that drive purposeful and adaptive action. It serves as the glue that transforms isolated functionalities into a unified, goal-driven network of agents. 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.

To enable this kind of adaptive intelligence, we now focus on three core aspects of behavioral design: **functional agent workflows**, **orchestration and decision logic**, and the integration of **memory**. **Workflows** define how agents structure and execute tasks; **orchestration** governs coordination and real-time decision-making across abstraction levels; and **memory** allows agents to learn, adapt, and collaborate effectively. Together, these elements form the behavioral foundation that transforms isolated functions into an intelligent, **goal-driven system**.

Designing the Agents Functional Workflow

In our web controller multi-agent system, agents are organized in a **top-down hierarchical structure**—comprising high-level, mid-level, and low-level agents. The overall system flow is **linear** and **deterministic**: high-level agents initiate tasks, mid-level agents decompose and coordinate them, and low-level agents execute specific actions. Designing this flow from top to bottom is critical to ensuring the system behaves predictably and efficiently.

To build an effective multi-agent workflow, we begin by defining the responsibilities and **handoff mechanisms between layers**. The high-level agent interprets the user's intent and generates a task plan. This plan is then passed to mid-level agents, which break it down into actionable

subtasks and allocate them to the appropriate low-level agents. Finally, low-level agents perform concrete actions on the environment, such as interacting with web elements.

The functional workflow follows these key steps:

1. **Generate Subtasks** – The high-level agent decomposes the user's goal into a set of smaller, manageable subtasks.
2. **Extract Elements** – Relevant elements from the web environment (e.g., UI components) are identified for interaction.
3. **Assign Elements** – These elements are mapped to specific agents for execution, based on their roles and capabilities.
4. **Execute Actions** – Low-level agents perform the necessary interactions (e.g., clicks, typing, mouse movements).
5. **Evaluate Results** – After each action, the system checks for success and decides whether to proceed, retry, or escalate.

This hierarchical and stepwise structure ensures that each agent layer functions with clear intent and coordination, laying the foundation for robust decision-making and scalable automation.

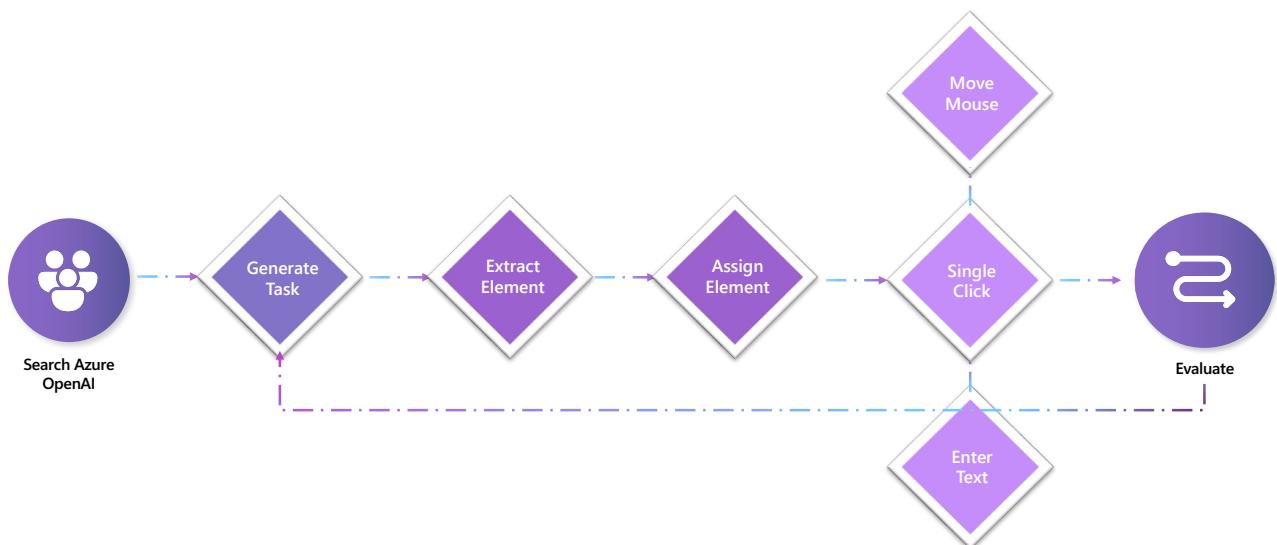


Figure 16. Agent Workflow

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

By designing this structured agent workflow, we've established a clear linear flow—a foundation that ensures our multi-agent system operates efficiently, adaptively, and intelligently. With this groundwork in place, we can now turn our attention to the next layer: **dynamic flow and orchestration**. This involves integrating **decision-making protocols** that enable agents to coordinate in real time, respond to evolving conditions, and tackle complex, interdependent tasks with agility and precision.

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: Generate Subtasks

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: Extract and Assign Elements

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: Execute Actions

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: Evaluate Results

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.

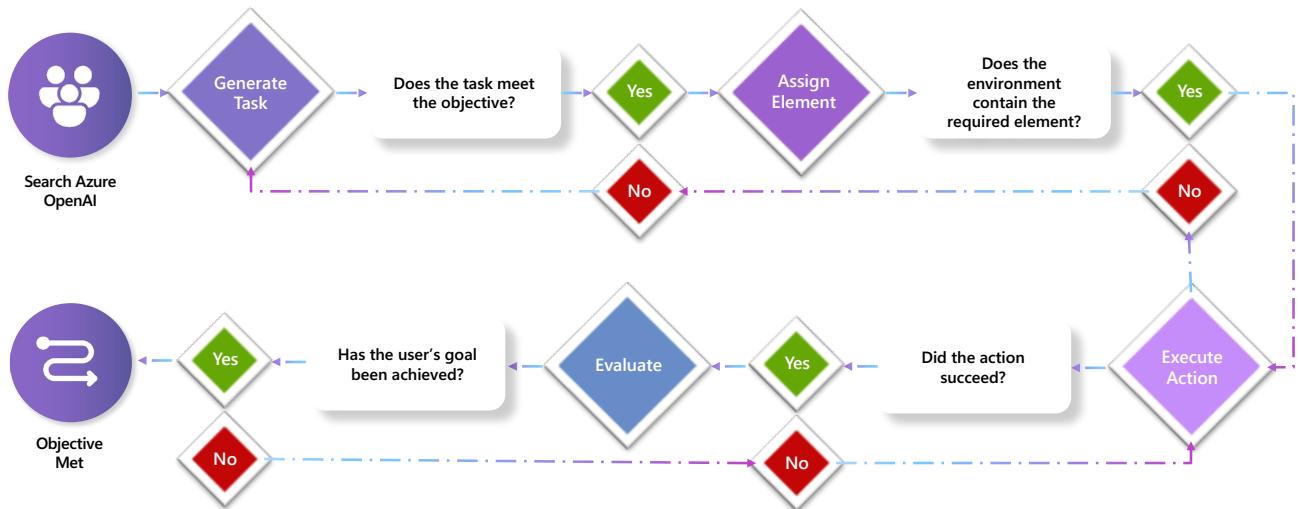


Figure 17. Agent Orchestration

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.

Designing Agent Memory

While workflows and orchestration provide the structural and procedural backbone of our agent system, true adaptability and intelligence emerge through **memory**. In our architecture, memory is not just a support mechanism—it serves as the **primary communication protocol** between agents. It enables **state persistence**, **tracks observations**, **records task transitions**, and **facilitates orchestration**. Memory allows agents to **perceive** what they're doing, understand the context of their actions, and coordinate accordingly. It is through memory that agents determine when individual tasks are completed and assess the overall progress toward the system's goals. Although memory operates across multiple dimensions, our focus here is on **short-term** and **long-term memory**—emphasizing short-term memory in the context of our web browsing solution to highlight its practical value in enabling dynamic, intelligent behavior.

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.

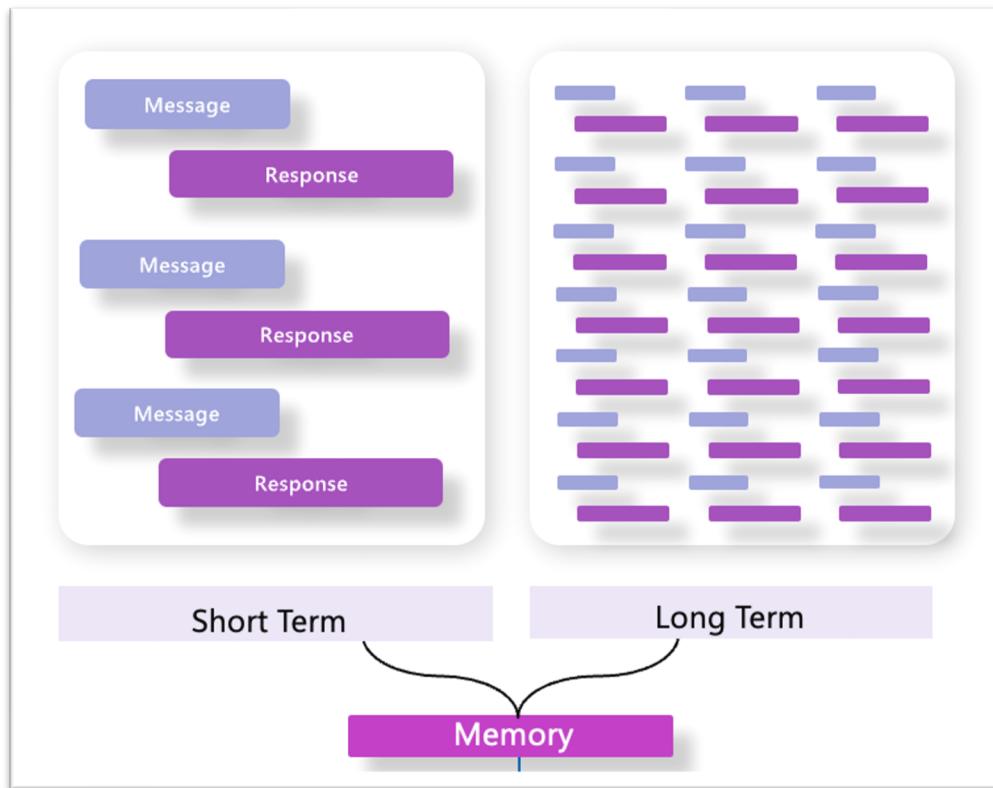


Figure 18. Short-Term and Long-Term Memory

Memory in this Web Browser multi-agent 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 and used at every agent level.

High-Level Agent (Web Coordinator) Memory

Purpose: The system uses memory to track **past tasks and failures**, refining its strategy by learning from mistakes. It avoids redundant errors, adapts tasks 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 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 execution of UI actions by referencing **past screenshots for comparison**. It updates memory when executing actions, capturing before-and-after screenshots, evaluating success, and detecting errors.

Designing Memory as a Communication Protocol

In our system web controller multi-agent system, memory itself can function as our primary 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-Driven Goal Completion

Another crucial benefit of using memory as a protocol is task and 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.

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.

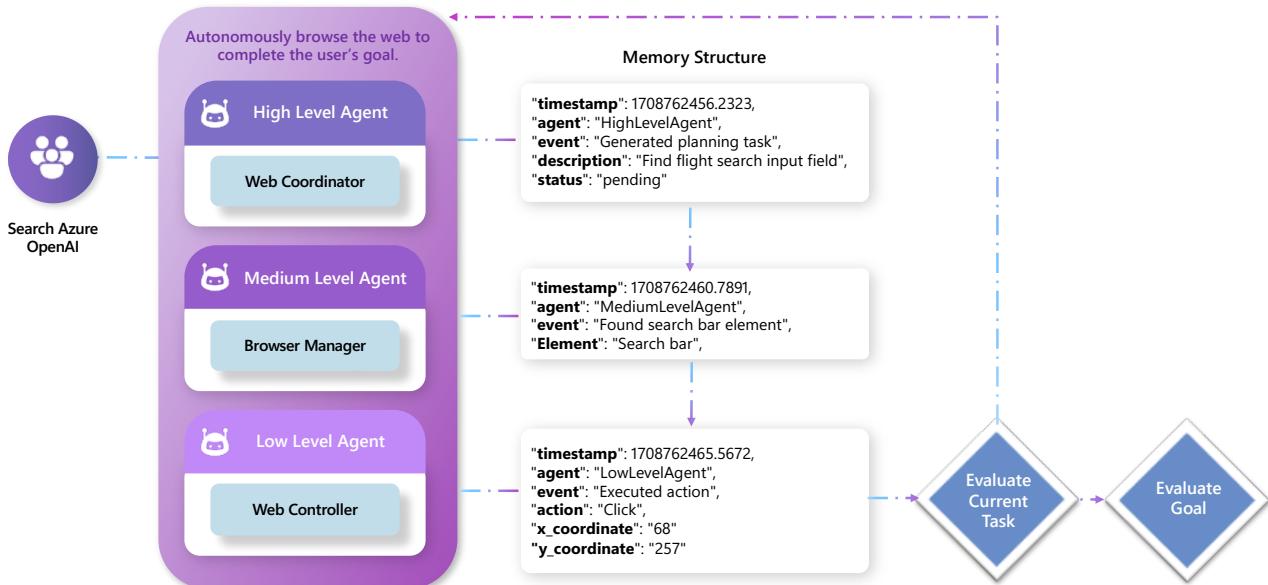


Figure 19. Agent Communication Protocol

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.

- **Orchestration and 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 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

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 orchestration as well, given its position at the top of our abstracted hierarchical design. The orchestration 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**, 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.
    """

    # Implementation logic for the evaluate function
    pass
```

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 technical implementation of our functions. This level transforms the abstract designs and pseudocode from earlier stages into a functional technical implementation.

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**.

Instead of relying on an off-the-shelf framework for our web controller multi-agent system, we demonstrate how to build a custom solution from the ground up. Using **Python** as the core language—along with **Pydantic** for structured data modeling and Selenium and **PyAutoGUI** for control—we prioritize flexibility and fine-grained control, allowing the system to be tailored precisely to our requirement

Python + Pydantic + Selenium+Pyautogui

Figure 20. Frameworks+Libraries

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 controller multi-agent system, the agents must interpret dynamic web content, understand complex instructions, and make intelligent decisions in real-time. This demands a model with strong **reasoning** abilities to break down user goals into actionable subtasks, **structured output** to ensure predictable formatting for downstream functions, **language** understanding to parse and generate natural language interactions, and **vision** capabilities to interpret webpage screenshots and UI elements visually. **GPT-4o** is uniquely suited to this task because it combines all these capabilities into a single multimodal model. It enables agents to analyze HTML content, interpret user goals, understand and interact with UI elements through screenshots, and return structured responses—all within a unified system. This makes GPT-4o an ideal foundation for building an intelligent, autonomous web controller that can reason, perceive, and act with human-like precision.

Data Structure Implementation

Structured output is critical to the success of our multi-agent system because it ensures that communication between agents is consistent, interpretable, and reliable. Rather than relying on free-form text, agents exchange information through clearly defined schemas—enabling precise coordination and minimizing ambiguity during task execution. GPT-4o's ability to generate structured outputs, such as JSON-like objects, directly aligns with this need, allowing it to seamlessly produce data that can be validated and consumed by other agents. To enforce this structure throughout the system, we use Pydantic, which offers a fast and intuitive way to define, validate, and serialize data models in Python. These structured representations act as the backbone of inter-agent communication—ensuring that plans, tasks, and UI actions are passed between agents in a predictable and machine-readable format. By combining GPT-4o's structured output capability with Pydantic's schema enforcement, we create a tightly integrated and error-resilient system that supports robust agent collaboration.

Our multi-agent system relies on four key Pydantic data structures to enable structured

communication and coordination:

PlanningTask – Represents a task created by the planner, including its ID, description, status, results, or errors. Used for task management and execution flow.

UIAction – Defines a low-level interaction (e.g., click, type), including coordinates and optional input data. Used by the Web Controller to perform precise UI actions.

Recommendation – Captures evaluation feedback, indicating success, goal completion, observations, and potential new tasks. Used to inform re-planning and adaptive behavior.

MemoryEvent – Logs key events like actions, errors, observations, and screenshots with timestamps. Used as a shared memory layer for inter-agent communication and recovery.

These structures ensure consistency, reliability, and modularity—enabling agents to coordinate seamlessly and adapt intelligently.

```
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
```

```
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
```

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

```
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 our orchestration loop that directs our agent system to complete a UI-based task step by step going from high, medium to low. 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

```
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"data:image/png;base64,{pre_base64}"}, "type": "image_url", "image_url": {"url": f"data:image/png;base64,{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",
            task_id=None
        ))
```

```

        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 used by the low level web controller agent 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/slacassegb/WebDriver_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).