Multi-Agent Systems for Autonomous Laboratory Instrument Operation

Nathan S. Johnson
Carl Zeiss Research Microscopy Solutions
Dublin, CA 94568
nathan.johnson@zeiss.com

August 14, 2025

Abstract

Recent advancements in large language models (LLMs) have opened new possibilities for laboratory automation through natural language interfaces. This paper reviews modular, multi-agent frameworks for controlling complex scientific instruments- such as microscopes, spectrometers, synchrotrons, or self-driving labs - using LLM-powered agents. These agents operate in defined roles (e.g., planner, instrument controller, data analyst) and communicate through structured, machine-readable outputs, enabling robust, scalable, and interpretable automation workflows. Multi-agent systems can also incorporate retrieval-augmented generation (RAG) for contextualizing prompts with relevant documentation and memory recall, enhancing reliability and grounding agent behavior in validated prior knowledge.

This paper overviews major concepts in multi-agent systems for instrument operation, followed a review of currently published working examples. Finally, we contribute our own prototype system for the Zeiss Versa 630 X-ray microscope, capable of executing 2D and 3D tomography workflows entirely from natural language prompts. Code-writing agents generate Python scripts to interface with the Versa instrument API, while analysis agents segment images and extract quantitative features. Memory embedding and retrieval from databases of prior workflows significantly improved performance, reducing failure rates and increasing reproducibility. Benchmarks of the prototype show reliable execution on complex tasks, with detailed logs and structured metadata facilitating transparency and traceability. Common errors in the system included prompt brittleness and inter-agent communication. We also review safety risks, reproducibility challenges, and engineering best practices for real-world deployment.

1 Introduction

A new opportunity has emerged for laboratory automation: enabling scientific instruments to be controlled through natural language interfaces, powered by large language models (LLMs). This development holds the potential to make complex instruments such as electron microscopes, X-ray microscopes, synchrotron beamlines, and a wide range of spectroscopic methods more accessible to a broader range of researchers, regardless of their familiarity with low-level programming interfaces or proprietary control software.

In the past five years, LLMs like GPT-4 [1], PaLM [2], LLaMA [3] and DeepSeek-R1 [4] have demonstrated remarkable capabilities in reasoning, planning, and generating code from natural language prompts. These capabilities have enabled a new class of systems known as *multi-agent*

architectures, in which several specialized LLM-powered agents cooperate to execute complex workflows. Each agent is typically given a defined role (e.g., planner, instrument operator, data analyst), and communicates with other agents via structured, machine-readable messages (often JSON).

The scientific community has begun to explore these architectures for laboratory automation. Notable examples include systems that autonomously operate atomic force microscopes [5], plan and run spectroscopy-based chemical experiments [6], and even collaborate to design and execute new synthesis pathways [7]. These prototypes show that multi-agent LLM systems can translate high-level goals - such as "scan this sample and measure density" - into executable code, retrieve relevant documentation, analyze collected data, and revise plans dynamically based on instrument output.

This paper presents a general framework for building such multi-agent systems to control laboratory instruments through APIs. We provide a high level overview of multi-agent architectures and review previous applications in the laboratory space. We also present a real-world case study: a prototype autonomous control system for a Zeiss Versa X-ray microscope. Built using GPT-based agents, this system demonstrates how a multi-agent architecture can perform full scan workflows - stage movement, image acquisition, segmentation, and analysis - entirely from a natural language prompt.

Our goal is to offer researchers across disciplines a clear, extensible pattern for building their own LLM-based instrument control systems.

2 Multi-Agent Systems for Laboratory Control

Modern scientific instruments - particularly high-complexity tools such as electron microscopes, X-ray microscopes, spectrometers and synchrotron beamlines - are increasingly operated through application programming interfaces (APIs). This programmability creates an opportunity for large language models (LLMs) to serve as intelligent agents that interpret natural language goals and generate structured commands for controlling instruments and analyzing data. A single LLM can often handle simple tasks. For robust, scalable operation a **multi-agent system** offers distinct advantages in modularity, error recovery, task specialization, and explainability.

This section outlines the architecture of such multi-agent systems, with a focus on structured outputs, agent orchestration, execution pipelines, and the use of retrieval-augmented generation (RAG) for memory and knowledge integration. The outline presented is based on a survey done by the authors of *current* multi-agent architectures; this is a rapidly evolving field and these definitions may become outdated quickly.

2.1 General Architecture of Orchestrated Multi-Agent Systems

The architecture of a multi-agent system typically includes:

- One or more **LLM-powered agents**, each with a clearly defined role.
- A central **orchestrator** agent that plans and coordinates actions.
- Optional **hard-coded modules**, such as calculators or digital twins.
- An overarching computer program for sending/receiving messages, parsing messages, and executing the workflow.

This modular design supports scalability and reliability, as individual agents can be updated, re-prompted, or restarted without affecting the whole system. New capabilities can be incorporated

by adding new agents, instead of modifying previous ones. While the orchestrator-based model is common, peer-to-peer or decentralized architectures also exist and are an area of active research in the AI community [8, 9], [10].

Core Elements of Each Agent

Each individual agent is defined by:

- A **large language model** for text generation
- A **system prompt**, specifying its role, behavior, expected inputs/outputs, and error-handling logic.
- A **structured output format**, typically in JSON, ensuring machine-readability.

The choice of LLM can be different for different agents. Often, agents are assigned different LLMs based on the needs of their task. A general purpose agent might use a large powerful model like OpenAI `GPT 4`, whereas a chat-only agent might use a smaller, faster mode like `o3-mini`. Agents for text retrieval and document searching commonly use models like `text-embedding-3-large`.

A **system prompt** is a special initial instruction that sets the behavior, tone, rules, and constraints of the model. It is not visible to the end user but strongly influences how the model behaves throughout the interaction. In a multi-agent system, each agent - Orchestrator, Instrument, Analysis, etc. - has a distinct role, and therefore each one requires a specialized system prompt tailored to its responsibilities. Examples of system prompts for various types of agents can be found in Appendix 5.3.

2.2 Structured Outputs: The Glue of Multi-Agent Systems

While LLMs are often celebrated for their natural language generation, their utility in laboratory automation stems from their ability to produce **machine-interpretable structured outputs**. In this context, agents can emit:

1. **Executable Code:** Python scripts or command-line instructions for instrument control or data analysis.
2. **Structured Data:** JSON-formatted metadata, execution status, file paths, or instructions for downstream agents.

This dual-mode output enables LLM agents to function both as planners and as task executors in automated scientific workflows.

Example Structured Output

A representative structured text response from an LLM agent might look like this:

```
{
"status": "success",
"code": "move_stage(0,1,-1) ",
"user_message": "Successfully moved the instrument stage."
}
```

The entire text above, starting from the first opening bracket and including the final closing bracket, is an LLM output. The placement of alphanumeric characters and punctuation determine it's structure. Opening and closing curly brackets **{ }**, along with key-value pairs indicated by a **:** colon make it a JSON format. There is structure within the key-value pairs as well. The value associated with the key "code" could be valid executable Python code. The value associated with the key "user_message" is natural language English text.

This structure conveys semantic and operational meaning. It enables downstream agents or the orchestrator to validate, log, and react to outputs, making the entire workflow interpretable and debuggable. API commands inside the `code` field can be parsed and executed for instrument control.

Most importantly, the above output is *machine-readable* meaning it can be parsed in a computer program. The above JSON-formatted text is a valid Python dictionary that can be fed into a hard coded Python program. This over-arching Python program can extract information from individual fields, such as the `'status'` field, to inform hard-coded logic. Furthermore, the instrument API code inside the `'code'` field can be extracted, saved, and run to initiate instrument actions. In this way the LLM can control the instrument.

Many common LLM interfaces, such as Ollama [11] or the OpenAI Python API [12], allow users to enforce structured outputs from LLM calls. Python libraries like Pydantic [13] can also be incorporated to enforce and validate structured LLM outputs.

There are many public, community-supported interfaces for building agentic systems and structuring their outputs. LangGraph is a Python library for building multi-agent workflows with many of the wide-ranging capabilities described in this paper [14]. Anthropic's Model Context Protocol (MCP) provides a standardized interface between LLM powered agents and external tools, especially computer-based interfaces like web tool APIs or messaging system [15].

Every LLM in the system requires a `schema` for their structured output. The schema defines the overall format of the output (e.g. JSON), the fields that the LLM is allowed to output, and what type should be in that field.

An example of a Pydantic schema is:

Listing 1: Generic Pydantic schema for an LLM output. The code formatting is Python.

```python
class LLMResponse(BaseModel):
    status: int = Field(..., description="Boolean status indicator; 1 for
        success, 0 for error.")
    code: str = Field(..., description="Executable Python code written by
        the agent")
    user_message: str = Field(..., description="Human-readable message to
        show to the user")
    embeddings: List[float] = Field(..., description="A list of float
        values, e.g., vector embeddings or output scores")
    debug_info: Optional[str] = Field(None, description="Optional field
        for additional debugging context")
```

When enforced, the LLM should return an output that matches this schema every time. Agents can be restricted to return specific types of values; for example, the "status" key in 1 is restricted to a Boolean integer. The "embeddings" key is restricted to be a list of floats. THe other fields are all strings. While the overall structure of the output is rigid (JSON formatting), the actual contents inside each field like `status` or `code` can be varied. In this way, the agent can call instrument API commands flexibly but in a structured way that is parsable by a hard coded program.

Structured outputs act as the common language of the system, allowing agents to communicate effectively regardless of their roles. An example of how structured outputs can be incorporated into

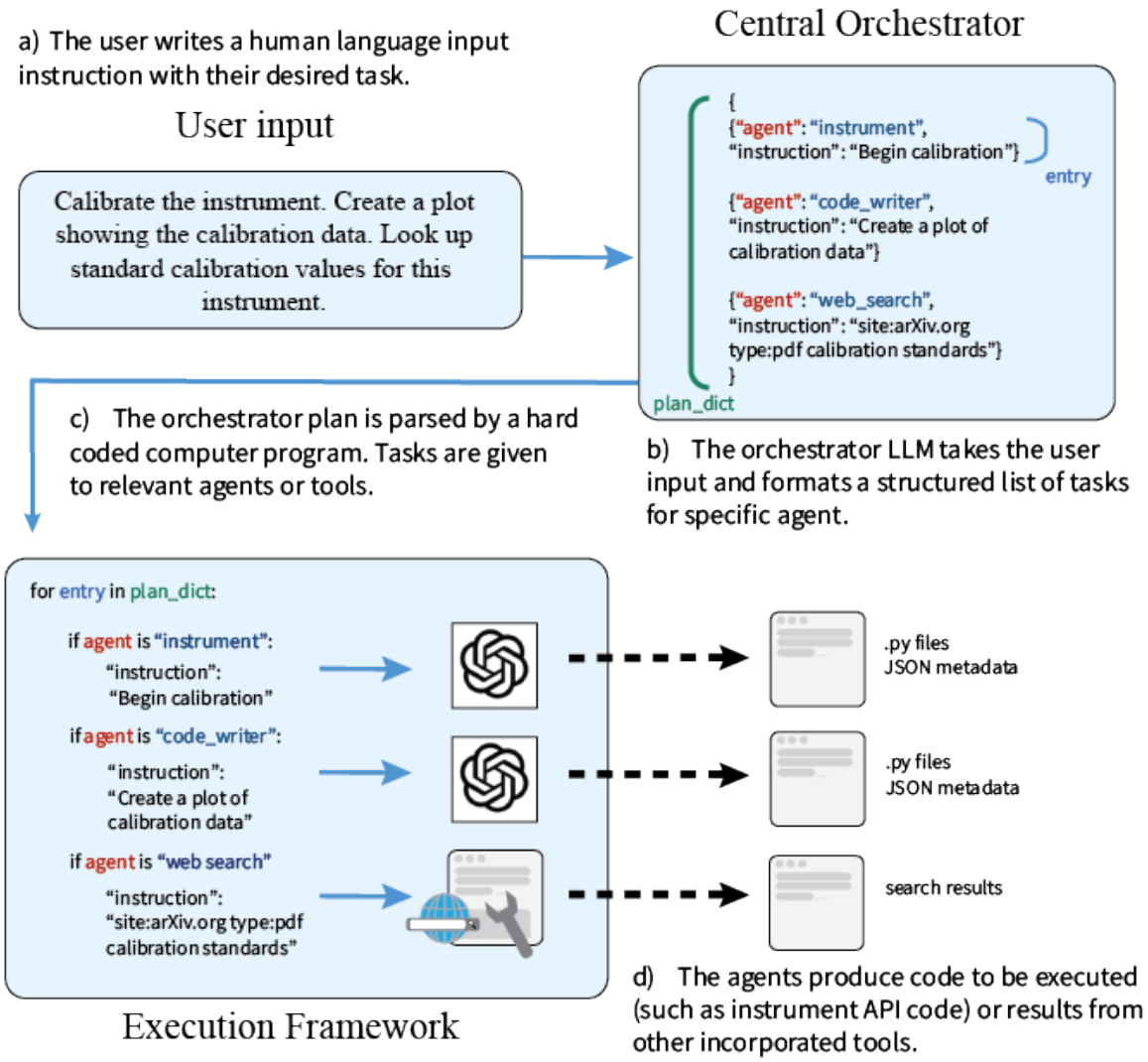


Figure 1: A generic example of an orchestrator-based multiagent workflow.

a Python framework are outlined in Appendix 5.3

The Orchestrator Agent

Many multi-agent systems rely on a central LLM agent to create workflow plans and delegate tasks. These agents have variously been called Orchestrators, Supervisors, Planners, and Managers. For this manuscript we have chosen to use Orchestrator. Inspired by systems like HuggingGPT [16], the orchestrator acts as the high-level planner. It interprets the user's natural language goal and produces a structured plan:

```
{
"plan": [
{"agenttocall": "instrument", "instruction": "Acquire an image at 4X magnification
    ."},
{"agenttocall": "analysis", "instruction": "Segment the image and compute porosity
    ."}
```

```
]
}
```

The orchestrator can revise plans mid-execution in response to:

- Execution errors (e.g., failed stage move).
- Unexpected data (e.g., image misalignment).
- Human clarification or interruption.

It may retry steps, rephrase instructions, or abort workflows as needed. This flexibility is central to autonomous operation in real-world laboratories. The orchestrator updates plans by being sent a new input message, such as the most recent error from an agent, and returns a new structured plan. This plan is then parsed by the overarching framework in a new workflow.

A generic overview of an orchestrator-based multi-agent system can be seen in Figure 1. The initial user message is parsed by the orchestrator into a structured list of concrete actions; the structured list is then parsed by a hard coded execution framework. Individual tasks are delegated to appropriate agents, such as code writers or other tools. The outputs of these LLMs - written code, search results, documents, etc - can be further processed to execute actions on an instrument, or passed back to the orchestrator for further decision making.

A similar approach uses a **Parser** agent in between the user and the orchestrator. The Parser agent takes natural language instructions and itemizes them into lists of actions. Parsers can be useful in many different cases. One scenario is when the user provides vague, unclear, or impossible directions; the Parser can make precise statements out of vague ones or ignore impossible instructions. Another use case for Parsers is when the complexity of user inputs is very high. Parser agents can be used to break context-heavy, complicated text into individual actions that the Orchestrator can better understand.

Code Writing and API Calling Agents

Code writing agents can be used to flexibly call API commands or write-free form code for data processing. These code writing agents require an additional framework to parse code or commands correctly.

Instrument Agents: Controlling an instrument with an LLM is based on two assumptions:

- The instrument has a programmable interface
- An LLM can reliably convert human language input into valid instrument API commands

Laboratory instruments increasingly have programmable interfaces. Enforcing an LLM to produce correct text to operate an instrument requires providing the LLM with a) the available commands it can call and b) context around how those commands translate to instrument actions. The manner of providing context can vary, but often the LLM is shown examples of instrument API code either in the system prompt or at the time of user prompting. Examples are often excerpts from the API documentation, which contains both the syntax of commands and human language explanations of what the command does. Well-supported documentation is a key for instrument control with LLMs.

Instrument agents will need to be customized for the specific instrument that they operate. This includes providing relevant context in the Agent's system prompt and building a framework to actually execute the API commands called by the agent.

In the prototype shown herein, the instrument is a Zeiss Versa X-ray Microscope. It has a Python 2.7 API that allows instrument operation through Python scripts. As part of the system prompt, the agent has been provided with the full documentation of the Versa Python API, including all possible commands that it can call, natural-language descriptions of each command, and relevant data types for command input/output. Showing the entire documentation is not a requirement; you may only want to enable the agent to call *certain* commands to restrict behavior. In this case, retrieval-augmented-generation (RAG) pipelines, discussed later, may be of use.

The method of executing instrument API commands written by the agent is different depending on the instrument API. In the case of a Python API, like the Versa's, an example code snippet is provided in Appendix 5.3. In summary: the Python code is extracted from the LLM's structured response. This Python code is then saved locally in a `.py` file. That `.py` file is then executed using Python's `subprocess()` command.

Architects of multi-agent systems must be very wary of using LLMs to make instrument API calls. Additional safety checks on both the multi-agent and the instrument will be required to prevent harm.

Analysis Agents: There are many benefits to real-time data analysis and feedback in instrument operation. Many instrument tasks, such as tuning parameters, require iterative data collection and data analysis to achieve optimal instrument conditions. An example workflow for a microscope might be a) acquire an image, b) measure contrast in the image, c) change the instrument exposure settings to adjust contrast, d) re-acquire a new image. An instrument agent can be used to collect the images and change exposure times; another agent may be required to process the image and measure contrast.

Types of analysis in scientific laboratories is a wide-ranging topic and a truly general framework cannot be presented here. Instead, the authors will focus on 2D image data analysis through Python scripting. The same methodology can be used for tabular, scalar, and series data.

In a similar way that an Instrument Agent can write code to operate an instrument, LLMs can also write code that open files and process data. The user can give a vague instruction such as "segment the image and measure the surface area." An LLM can take this instruction and return a Python script that:

- imports necessary packages, such as `skimage` or `opencv`
- load a tiff file
- applies an Otsu threshold to the image
- Calculates surface area of the segmented region from the known pixel size and number of segmented pixels
- Saves the surface area in a file

Once the surface area is saved in the file, it can be accessed and opened by other agents for further processing. Executing free-form Python code from an Analysis agent can be done with the same approach outlined in Appendix 5.3.

As with the Instrument Agent, extra care must be taken when using free-form code writing LLMs. Improper or malicious code can cause damage to the local workstation. Containerization of freeform Python scripts may be required in some applications. A human-in-the-loop architecture is also a common safety tool, whereby code is reviewed by a human before execution.

Sometimes, analysis does not need to be 'free form' but instead researchers already have data processing modules set up. These could be through user-defined Python scripts, or an API call to a third party software. In this case, it can be useful to use a Middle Agent to initiate calls to a hard coded module.

Middle Agents, Digital Twins, and Hard-Coded Modules

Some workflows require bridging between LLMs and rigid, non-LLM components. For example, an instrument may have a digital twin used for simulating data acquisition conditions. This digital twin likely requires specifically formatted inputs. The user may have written their own data analysis python file and they want to execute it on a specific dataset. A **middle agent** transforms human or LLM-generated context into a well-formed call to the digital twin or hard coded module.

These agents operate on hybrid logic: loosely structured LLM prompts for input, and precisely structured outputs for deterministic modules. This bridge design has been effective in systems like CLAIRify [17] and ChemAgents [7].

An example of how a middle agent can be placed between the Orchestrator and a hard-coded digital twin model is shown in Appendix 5.3.

The Instrument or Analysis Agent approach defined above, where the agent writes free form Python code to be saved and executed, may be too brittle or unsafe. In these cases, you can adopt this middle agent approach to limit the types of commands that the agents can call.

2.3 Retrieval-Augmented Generation in Multi-Agent Systems

Retrieval-Augmented Generation (RAG) combines LLM flexibility with access to external information sources, enhancing accuracy, reproducibility, and user alignment [18]. In a lab setting, two applications are particularly valuable: technical knowledge retrieval and memory storage of previous tasks.

Understanding Embedding and Retrieval

RAG relies on the concept of *semantic embedding*, where a sentence, paragraph, or user query is converted into a high-dimensional vector (a list of numbers that captures its meaning). These vectors are stored in a **vector database**. When done correctly, text embeddings that are *semantically similar* (have a similar meaning) should be embedded near each other. When a new query is issued, it too is embedded and compared against the stored vectors using similarity metrics (such as cosine distance). Closely matching past documents or workflows are retrieved and injected into the agent's prompt.

Many relevant text documents, such as the user manual of an instrument, are too long to be directly injected into an agent system prompt. Furthermore, users often only need particular sections of a document, such as a specific procedure, instead of the entire document. RAG can be used to extract only relevant information from documents and show them to the LLM agents to provide relevant context.

This same methodology can also allow the LLM to recall relevant prior knowledge or tasks, even if the language used by the user is different from the stored version.

Application 1: Memory Retrieval

One of the most powerful features of a multi-agent system is its ability to learn from past experience. This is achieved through **memory retrieval**, a process in which past inputs and outputs - especially

successful ones - are embedded in a vector database and recalled during future tasks. Python packages for multi agent workflows, such as LangGraph [19], implement memory embedding and retrieval as part of their module.

A high level overview of memory retrieval implementation is shown in Figure 2. Every agent in the system communicates via text: user instructions, structured prompts, code outputs, and status updates are all ultimately strings. Because of this, both the inputs to and outputs from agents can be converted into *semantic embeddings* - dense vector representations that capture the meaning of the text - and stored in a database.

When a new user instruction is received, it is embedded and compared against this memory bank to identify semantically similar tasks. If a close match is found, the system retrieves the prior agent responses and injects them into the prompt of the relevant agent. This encourages the agent to repeat or adapt a previously successful workflow instead of generating a novel response from scratch.

This reuse of prior outputs has two main benefits:

1. It improves reliability by reducing the chance of generating malformed or inconsistent code.
2. It allows the system to respond more effectively to ambiguous or colloquial user instructions.

Ambiguous or colloquial user instructions can be matched and embedded with more specific, context-heavy worfklows. For example, a user might provide a vague instruction such as:

> "Center my sample on the rotation axis."

While understandable to a human, this instruction is under-specified for automation. If the system has embedded a previous, more technical workflow that achieved this same goal, it can match the vague instruction to that workflow and reproduce it. The matched workflow for "center my sample" might consist of:

- Acquiring an image.
- Segmenting the image and calculating the centroid.
- Determining the offset of the centroid from the rotation center.
- Moving the stage accordingly.
- Reacquiring the image for confirmation.

This memory-based semantic matching lowers the barrier for novice users while preserving expert behavior.

However, implementing memory retrieval requires care. Embedding low-quality or incorrect outputs can degrade performance. If "bad" memories are stored and later retrieved, they can be injected into new prompts and cause repeated failure. It is important to validate and curate the memory database, track task outcomes, and optionally tag or filter embedded responses based on success or reliability.

Properly implemented, memory retrieval gives the system a kind of experiential learning: a way to remember what worked, avoid what failed, and tailor its behavior to past context.

Application 2: Knowledge Retrieval

In addition to recalling past workflows, RAG enables agents to retrieve relevant technical knowledge from static sources such as:

- Instrument manuals (e.g., optimal scan parameters for various materials)
- Scientific literature (e.g., prior acquisition workflows, image analysis techniques)
- Standard operating procedures and lab documentation

For example, an agent may be asked:

> "What exposure time should I use for titanium?"

Instead of guessing or relying on outdated heuristics, the agent can query a knowledge base containing embedded documentation. It retrieves the most semantically relevant excerpts - such as a manual entry or experimental note from a titanium scan - and injects this context into the prompt before generating a response. This retrieval step ensures that the answer is not only plausible but grounded in real technical precedent.

Knowledge retrieval via RAG is already being adopted in automated laboratories, LLM-driven experimental planning systems, and autonomous literature search tools [20]. It plays a vital role in extending the capabilities of LLMs beyond their training data, allowing them to reason with live, domain-specific information.

RAG System Design

To implement knowledge retrieval effectively, documents must first be preprocessed and stored in a way that supports efficient similarity-based lookup. This typically involves:

- **Chunking**: Documents are divided into smaller units (typically 50–500 tokens) to match semantic granularity and stay within LLM token limits.
- **Embedding**: Each chunk is converted into a high-dimensional vector using a model such as `text-embedding-3-small/large` or `SentenceTransformers`.
- **Indexing**: The vectors are stored in a vector database such as FAISS or Weaviate.
- **Querying**: At runtime, the user's input or agent prompt is embedded and compared to the stored vectors to retrieve the top-k most similar chunks.
- **Prompt Injection**: Retrieved text chunks are added to the LLM's input text, enriching its response with grounded, task-relevant knowledge.

Proper chunking is essential for high-quality retrieval. It ensures that unrelated or distracting sections of a document (e.g., "Troubleshooting Errors") are not conflated with more relevant ones (e.g., "Common Procedures"). Each chunk should reflect a distinct semantic idea so that retrieval matches user intent more accurately.

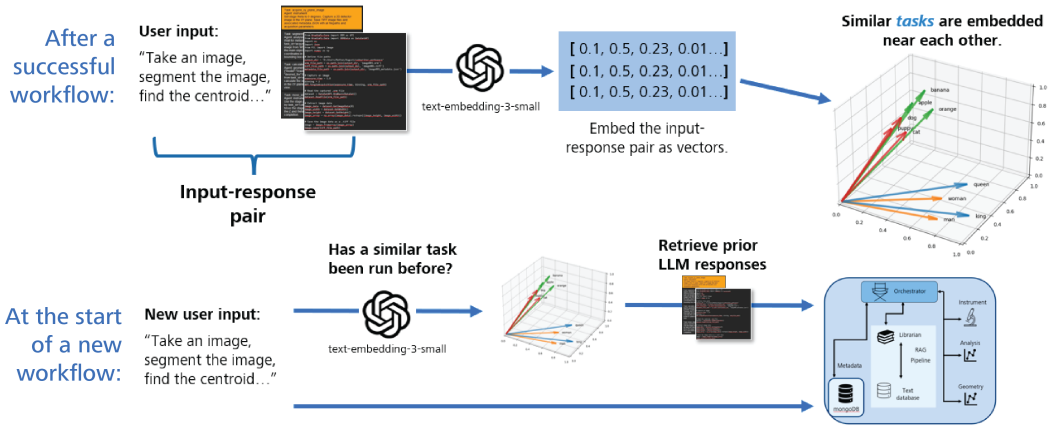


Figure 2: Retrieval-augmented generation architecture for embedding prior successful tasks and retrieving them for new user inputs.

Benefits and Caveats

RAG enhances multi-agent systems in several key ways:

- **Workflow repeatability**: Access to validated procedures promotes consistent execution.
- **User accessibility**: Allows vague or incomplete instructions to be supplemented with expert documentation.
- **Decision grounding**: Ensures that agent responses are anchored in factual, verifiable sources.

However, RAG is only as effective as its underlying corpus and retrieval accuracy. Poorly curated or low-quality documents can introduce noise. Bad matches may inject irrelevant or incorrect context into prompts. Therefore, systems should implement safeguards such as similarity thresholds, validation rules, usage tracking, and - where necessary - human-in-the-loop review.

When designed carefully, RAG transforms an LLM from a general-purpose model into a domain-adapted assistant, capable of drawing on local, validated knowledge to assist with scientific reasoning and instrument control. At the same time, a multi agent system with bad RAG can perform worse than a multi agent system with no RAG at all.

2.4 Summary of Multi Agent Systems

Multi-agent systems provide a flexible, interpretable framework for laboratory automation using LLMs. Structured outputs and a centralized orchestrator enable robust coordination. Retrieval-augmented generation further strengthens these systems by anchoring decisions in memory and documentation. The resulting architecture is modular, extensible, and capable of executing scientific workflows end-to-end from natural language instructions.

3 Literature Review of Existing Applications

Over the past five years, the idea of using large language models (LLMs) to operate scientific instruments has evolved from isolated experiments to fully autonomous, multi-agent systems. Researchers across disciplines have begun to harness LLMs not only as interface layers but also as planners,

tool users, and data analysts. This section surveys the most notable academic and industrial efforts to apply either single LLMs or multi-agent architectures to real-world laboratory instrumentation, including atomic force microscopy (AFM), electron microscopy (EM), spectroscopy, and automated chemical synthesis.

3.1 Single-Agent LLM Controllers: From Interface to Autonomy

The earliest efforts explored using a single LLM to interpret natural language and translate it into code or control sequences. These systems often acted as intelligent translators between human input and existing instrument APIs. For example, Yoshikawa et al.'s CLAIRify [17] translated plain-language chemistry protocols into executable robot instructions using a single LLM followed by verification modules. This approach highlighted the potential of LLMs for procedural reasoning and command generation, especially when paired with structured output formats like JSON or XML.

Similarly, the ThermoFisher **EM CoPilot**, presented by Geurts et al. [21], demonstrated the feasibility of integrating a multimodal LLM interface with a commercial electron microscope. The system combined a large vision-language model (e.g., GPT-4 with a vision encoder) with real-time microscope control via API. It allowed users to issue spoken or typed commands such as "zoom in on the region with granular texture" or "acquire a high-resolution scan of the bright phase," which the LLM interpreted to control the scanning electron microscope (SEM) or transmission electron microscope (TEM). By incorporating visual feedback and structured reasoning, EM CoPilot served as an accessibility layer for non-expert users, enabling them to perform sophisticated operations through natural language [22]. The LLM controls the instrument through a Python-based API for ThermoFisher microscopes called AutoScript. User instructions are converted into Python-based commands that execute instrument actions. ThermoFisher has also extended this capability to multi-modal microscopes. Guerts has demonstrated a similar agentic approach for controlling a combined focused ion beam (FIB) and SEM microscope [23].

These systems showed that even without full multi-agent decomposition, a well-prompted LLM can generate code, infer intent, and control instruments.

3.2 Multi-Agent Architectures for Autonomous Microscope Control

Several research groups have turned to **multi-agent systems**, where distinct LLM-powered agents are assigned specialized roles such as planning, instrumentation, analysis, or data interpretation. These systems are modular, interpretable, and better equipped to manage complex workflows involving multiple steps, conditions, or feedback loops.

One of the most comprehensive demonstrations is the **AILA system** (Artificially Intelligent Lab Assistant) developed by Mandal et al. [5]. AILA operates an atomic force microscope (AFM) through a three-agent architecture:

1. **Planner Agent**: Parses user goals and decomposes them into subtasks.
2. **AFM Handler Agent**: Generates Python control code for the instrument using the manufacturer's API.
3. **Data Handler Agent**: Performs post-acquisition image processing such as height-map segmentation and surface roughness calculation.

AILA was evaluated on AFMBench, a suite of AFM benchmark tasks including calibration, imaging of graphene samples, and multilayer thickness estimation. It achieved high success rates

(over 80%) on multi-step workflows and demonstrated the ability to adapt plans based on error states or missing data. Crucially, the study benchmarked LLM outputs against expert-written protocols and found the multi-agent approach significantly outperformed a monolithic single-agent baseline. However, it also highlighted common issues:

- Prompt brittleness: Small changes in wording could derail execution.
- Tool misselection: The planner sometimes invoked the wrong agent or misunderstood context.
- Out-of-scope actions: the multi-agent system sometimes choose actions that were outside the specified requirements

The common errors in AILA are also present in our prototype, as will be discussed in later sections.

3.3 Multi-Agent Architectures for Synchrotron Beamlines

Another notable example of multi-agent system controlling complex instrumentation is Virtual Scientific Companion (VISION) by Mathur et al [24]. VISION is a multi-agent LLM-based assistant developed to enable natural language control of synchrotron X-ray beamlines. It employs a modular cognitive architecture composed of specialized LLM "cogs," each tailored to a specific function (e.g., transcribing speech, interpreting commands, code generation for instrumentation, or data analysis). These cogs operate in a deterministic workflow that parses user inputs and sequentially executes corresponding tasks, effectively translating a researcher's voice or text instructions into beamline operations and analytical procedures. The system's implementation integrates a user-friendly interface (voice/text via a PyQt5 GUI) with a high-performance backend server for LLM inference, interfacing with existing beamline control and analysis software (such as Bluesky and SciAnalysis) to run generated commands on actual instruments in real time.

VISION does not operate strictly on the orchestrator/planner architecture of other examples. Instead, a *Classifier* agent is used to parse the users intention and routes the request to appropriate agents. VISION uses `Qwen2` and `Qwen2.5-coder` through the Ollama interface to write Python code that operates beamline instruments and processes data. These are called the Operator and Analyst agents, respecitvely. Mathur's multi-agent system relies on a human-in-the-loop safety check, where Python codes are shown to the user and approved before execution.

VISION also implements a novel ability for users to create and store pre-defined functions and workflows. Users can define a beamline operation through a natural language description; this operation is then parsed by a ChatGPT-4o instance into a structured JSON that encodes the workflow into agent commands. These commands can be accessed later by the Operator and Analyst agents to re-run specific actions.

Mathur et al. demonstrated VISION in operation by conducting a fully voice-driven X-ray scattering experiment, achieving the first voice-controlled beamline run with low latency on standard beamline hardware. This work highlights the novelty and significance of LLM-based assistants in laboratory automation: VISION's extensible multi-cog design and natural language interface streamline human–instrument interaction, marking a step toward more intuitive autonomous laboratories.

3.4 Applications Beyond Microscopy: Chemistry and Spectroscopy

Multi-agent systems have also been explored in the context of automated chemistry. Notably, Boiko et al.'s **Coscientist** system [6] used GPT-4 to autonomously plan, execute, and analyze chemical

experiments using real-world lab hardware, including an Opentrons OT-2 liquid handler and UV-Vis spectrometer. While the system used a single LLM agent as a planner, it was integrated with multiple tools: literature search modules, database lookups, and code execution environments.

In a separate line of work, the **ChemAgents** system by Song et al. [7] demonstrated a hierarchical five-agent system designed for autonomous chemical discovery. Each agent had a distinct role:

- Literature Reader
- Experiment Designer
- Robot Operator
- Computation Modeler
- Task Manager (Orchestrator)

ChemAgents coordinated robotic platforms, simulation tools, and analysis pipelines to discover new photocatalysts and optimize reaction conditions, with minimal human supervision. The modular agent design made it adaptable to new lab environments and instrumentation.

3.5 Cross-Cutting Themes and Common Patterns

Across all systems surveyed - AFM, EM, spectroscopy, chemical robotics - several shared principles and challenges emerged:

Structured Communication: All multi-agent systems relied on structured outputs (usually JSON) to enable agents to exchange interpretable and machine-verifiable messages. This included task plans, filepaths, code blocks, and error reports. Structured messages acted as the backbone for orchestration and traceability.

Semantic Translation: A major benefit of LLM-based control systems is their ability to semantically translate vague or high-level goals (e.g., "zoom on defect area" or "prepare 5 mM solution") into precise execution steps. Systems like AILA and Coscientist used embedded memory or retrieval to match ambiguous commands to previous successful plans.

Tool Use and Verification: LLMs often served as tool users, calling internal APIs, external code libraries, or deterministic solvers (e.g., digital twins). Some systems implemented verification modules to check generated code for safety and correctness before execution [17].

Reactivity and Planning: Orchestrators played a central role in decomposing tasks, sequencing agents, and recovering from failure. Agents capable of mid-plan updates were better suited to instrument environments, which are often noisy, real-time, and error-prone.

Limitations and Open Problems: Despite their promise, most prototypes shared common weaknesses. Multi-agent systems can show soor error recovery when hardware state diverged from expectations. Multi-agent systems can often try to proceed with workflows as planned, despite an error occuring on one step. This initial error propagates through the workflow, causing downstream cascading failures. Robust error detection is required so that orchestrators can re-plan workflows after a failed instrument or analysis step.

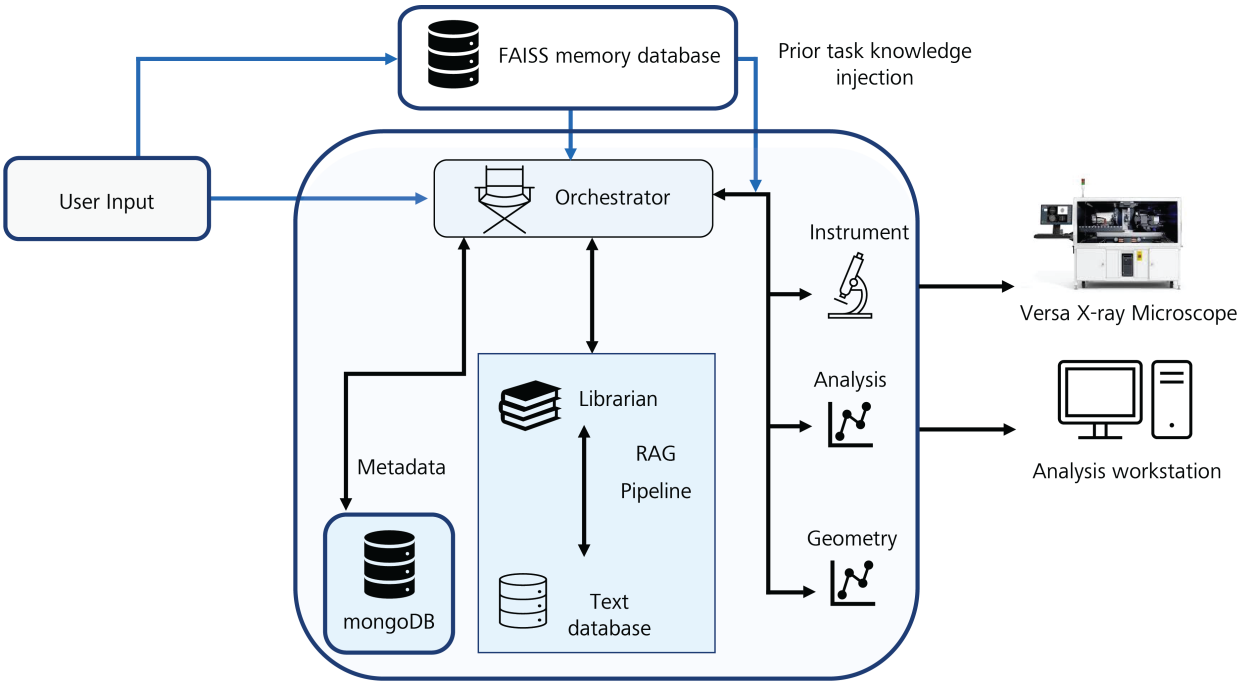


Figure 3: High level architecture of the multi-agent Versa XRM controller prototype.

Brittleness in error handling is related to brittle system prompts and user inputs. Often, multi-agent systems only operate when prompted with exactly the right syntax, context, and level of detail. When users provide vague or unclear instructions, the orchestrator can easily plan a workflow that diverges from user intent. Multi-agent systems need a greater robustness against syntactical changes in user inputs.

3.6 Literature Review Summary

Recent years have witnessed the emergence of LLM-powered systems capable of controlling laboratory instruments, planning experiments, and analyzing results with increasing autonomy. Researchers have shown that LLMs - especially when organized as modular agents - can act as powerful collaborators in scientific workflows. These systems share common structures (planner–executor–analyzer), rely on structured messaging, and aim to lower the barrier for interacting with sophisticated hardware. However, they also reveal key challenges in prompt reliability and system-level validation. The next section will demonstrate a concrete implementation of these principles through our prototype: a multi-agent controller for an X-ray microscope.

4 Case Study Prototype: X-ray Microscope

We have developed a full prototype multi-agent controller for a Zeiss Versa 630 X-ray microscope. This system interprets natural language commands and executes full experimental workflows - from image acquisition to image segmentation and result summarization - using a modular multi-agent architecture.

4.1 System Architecture and Agent Overview

An overview of the system architecture is shown in Figure 3. The workflow is initiated from a natural language user input. This input is given to the orchestrator, who develops a task list and deploys tasks to the various other agents or modules. The architecture includes seven distinct components, each with a defined function and communication protocol: the overview of the agents can be seen in Table 1

Agent	Function
Orchestrator	Plans workflows, delegates tasks, monitors agent success/-failure, and revises plans dynamically.
Instrument Agent	Writes Python 2.7 scripts for controlling the Zeiss API: moving stages, configuring scans, executing 2D or 3D acquisitions. Saves metadata and file paths.
Analysis Agent	Writes Python 3.11 scripts for image/volume segmentation, feature analysis (e.g., centroid, FOV, porosity), and metadata saving.
Geometry Module	Deterministic, non-LLM module written in Python. Computes stage/source/detector positions given sample size, bounding box, and desired user magnification.
Geometry Parser (Middle Agent)	Converts loosely specified tasks into precise JSON inputs for the Geometry Module.
Librarian Agents	Uses RAG (Retrieval-Augmented Generation) to answer technical questions based on embedded user manuals and literature.
Reporter Agent	Summarizes agent outputs and reports task progress and final outcomes to the user.

Table 1: Agent roles in the Zeiss Versa multi-agent control system.

4.2 Code Writing Agents - Instrument Control and Analysis

The Instrument Agent uses code templates and memory recall to construct valid Python 2.7 scripts for microscope control. It supports:

- Direct 2D scan acquisition via function calls
- Full 3D tomography through recipe files
- Metadata generation in JSON files (exposure time, pixel size, position)

The Analysis Agent uses `scikit-image`, `OpenCV`, and `numpy` to segment images and compute centroids, bounding boxes, and volume features. It also stores all outputs and metrics as JSON for downstream use.

4.3 Memory Retrieval and Document Searching Pipeline

Each agent stores prior input-output pairs as vector embeddings using FAISS [25]. These records are (optionally) retrieved during runtime to repeat previously successful workflows and translate vague instructions into validated plans.

The memory retrieval pipeline searches the vector space for similar user requests and, if matches are found, appends retrieved context into the agent prompt, as shown in Figure 3. This boosts reliability in complex tasks by biasing agents toward known-good behavior.

A separate RAG pipeline exists for searching through embedded technical documents. A FAISS text database was populated using a wide range of ZEISS user manuals and technical procedures. The orchestrator has been instructed to search the manuals database for proper technical procedures when the user provides a vague input. An under-specified command, such as "perform a

4.4 System Infrastructure and Interface

- **Context window:** Each agent operates with a prompt window containing current instruction, metadata, prior steps, and optionally retrieved memories.
- **Metadata storage:** Each code-writing agent saves relevant metatdata in a JSON file within their code. For the instrument agent this includes parameters like stage position, image exposure, image binning, detector position, etc. For the analysis agent this can be calculated values (centroid, bounding box) or parameters used in analysis (threshold values, image filters, etc).
- **Execution environment:** Instrument & Analysis Agent code is sandboxed and executed using Python `subprocess`; output logs and filepaths are parsed automatically.
- **GUI:** The user interacts through a lightweight Python GUI where they submit prompts, review outputs, and monitor status. In this prototype the GUI is for development purposes only.

4.5 Multi Agent System Hyperparameters

Table 2 lists all parameters and hyperparameters used in the prototype. There are two separate RAG pipelines: one that searches a database of Zeiss user manuals and technical documentation; another searches a database of published literature related to X-ray microscopy. Embedding these documents separately improved system performance at knowledge retrieval.

4.6 Benchmarking and Evaluation

We benchmarked the system on two key workflows, each evaluated over multiple independent runs. All tasks are logged with colored bar charts. Blue represents LLM responses from code writing agents where the code executed correctly; red is for responses where the code did not execute. Dark gray represents responses where the Orchestrator decided to update the workflow mid-plan. This is not necessarily good or bad, but if the Orchestrator is changing the plan too many times it can be a sign of system failure. Light gray represents overhead LLM calls; these are responses that are required by the system (such as the initial Orchestrator plan, the Reporter communicating back with the user, or the Geometry parser LLM). Overhead is required, but too much overhead can also be a sign of system issues.

Task 1: 2D Axis of Rotation Centering A very common initial step in any X-ray microscopy workflow is to center the sample on the axis rotation. This workflow involves finding the sample centroid and moving it to the center of the detector image, then repeating the same steps in an orthogonal rotation plane. The workflow can be broken down into:

Component	Parameter	Value
Manuals RAG	Chunk size / overlap	500 tokens / 50 tokens
	Top-k retrieved	5
	Similarity threshold d	1.5
Literature RAG	Chunk size / overlap	1000 tokens / 200 tokens
	Top-k retrieved	5
	Similarity threshold d	1.5
Memory Retrieval	Top-k retrieved	2
LLM Models	Orchestrator, Instrument, Analysis	OpenAI GPT-4o
	Geometry Parser, Reporter	OpenAI `o3-mini`
	RAG Embedding Model	OpenAI `text-embedding-3-small`

Table 2: System hyperparameters for RAG, memory retrieval, and LLM configuration.

1. Move sample to a random position; acquire initial image
2. Segment sample in image and compute sample centroid
3. Calculate offset and new stage position using Geometry Module
4. Move stage and reacquire image
5. Rotate sample and repeat steps in an orthogonal plane

This task was benchmarked using a standard input prompt. There was no syntactical change in the prompt between runs; in this sense, the system is very brittle against prompt variation. The results below show random samples selected from benchmark tests.

Results: Figure 5 shows the multi agent performance on this task both with and without prior memory retrieval. The bar charts in Figure 5 shows the total number of LLM calls made during the workflow (y-axis). The number above each bar represents the time required to execute the workflow.

The system performed with 100% human-verified success in this workflow on the static prompt. An amalgamation of 10 randomly selected benchmark tasks can be seen in Figure 4, including the initial sample position (left) and final sample position (right).

Task 2: 3D Scout and Zoom Acquisition Another common user workflow is often called a 'Scout and Zoom' workflow. This workflow involves collecting an initial overview 3D tomography of the entire sample. The user then wants to zoom in on a particular feature and collect a second, higher resolution/magnification 3D tomography. The agent steps for this workflow involves:

1. Instrument agent collects an initial 3D tomography
2. Analysis agent segments the 3D volume, then computes the centroid and sample size
3. The geometry module uses the sample centroid and size to compute source/detector positions to achieve a desired magnification

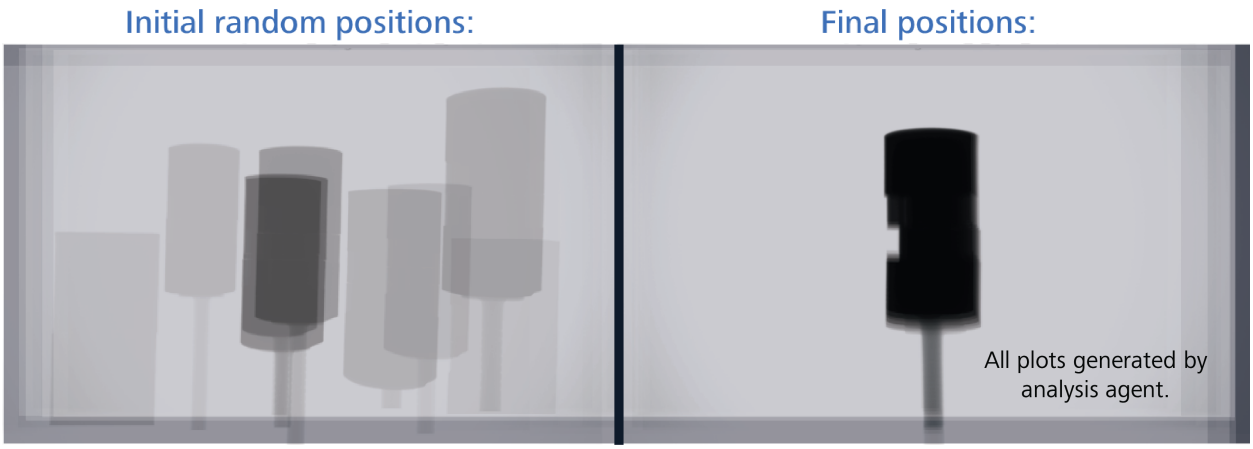


Figure 4: Superimposed initial and final positions from 10 random samples of the 2D centering task.

4. The instrument agent modifies a recipe file with the new source/detector positions and collects a second 3D tomography

For each task, the geometry module was given a different random magnification value. For a given magnification, the sample should occupy a specific field of view (FOV), or percentage of the total volume. As with the 2D centering workflow, this task was benchmarked by looking at how many LLM calls were required, how many code-writing responses executed correctly, and how long the workflow took to complete. All final results were verified by a human; the final FOV was measured by a human in third party image analysis software.

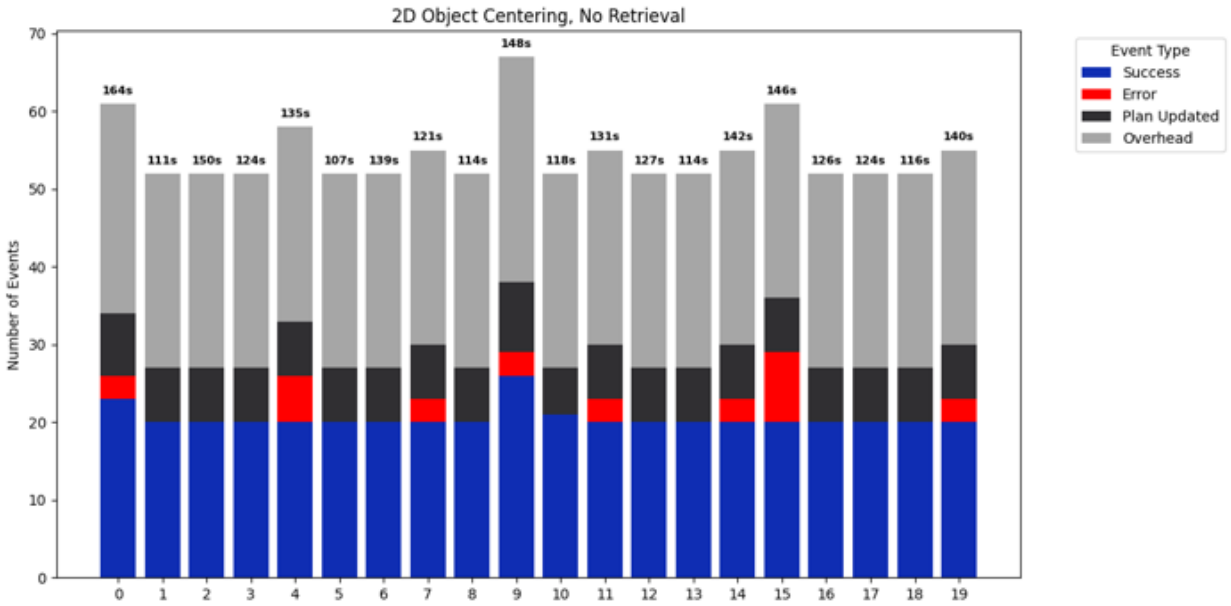
This workflow was run using a 3D printed PLA block with dimensions 26mm $\times$ 26 mm $\times$ 26mm. Examples of 2D slices through a tomography of the block can be seen in Figure 7.

Results: The Python scripts for this task were considerably longer and more complex than for the 2D centering workflow. Whereas the scripts for the 2D workflow were often less than 20 lines of code, for the 3D Scout and Zoom workflow the scripts could sometimes be 100+ lines of code. This introduces more room for failure.

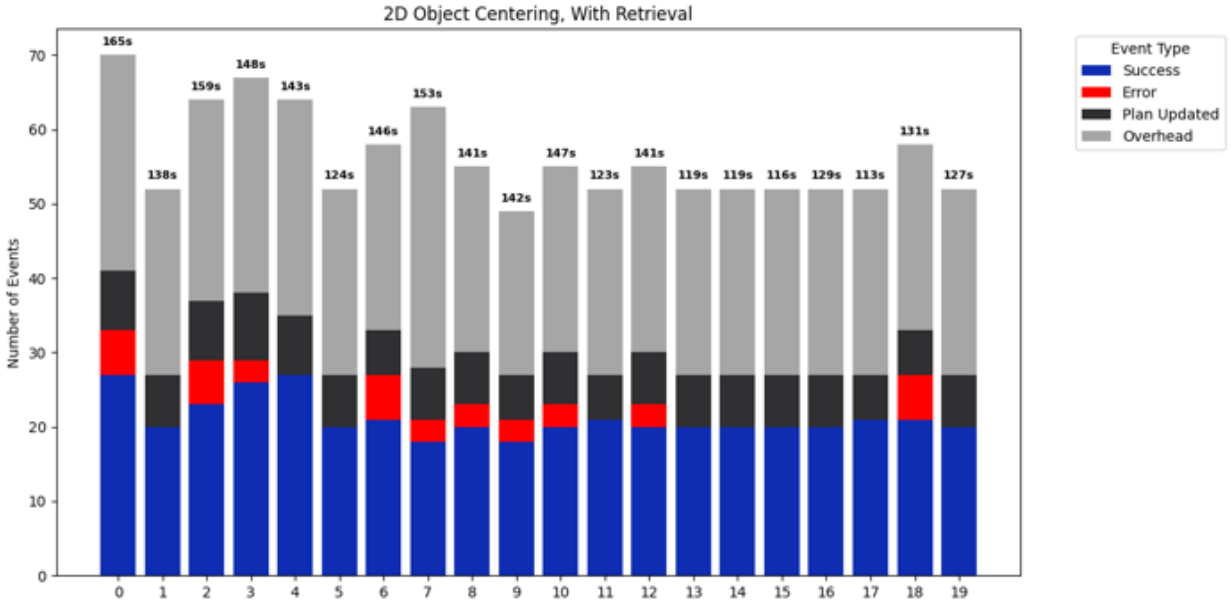
Figure 6 shows benchmark performance both with and without memory retrieval, taken from 10 random samples. Bars with red circles above them represent workflows that had to be stopped by the user, or were aborted by the Orchestrator. Without memory retrieval, the workflow only completed 60% of the time. After memory retrieval was implemented, the completion success rate improved to 100%. Even in cases with cascading failures, such as Task 2 from Figure 6b, the Orchestrator was able to right itself and finish the task *after* the user intervened and provided updated instructions.

The improvement in system performance with memory retrieval is significant. For a complex code-writing task, agents can return a wide variety of responses that all *approximately* complete the task. However, if the agents are writing new code for the same task every time then there can be wide variability in their responses and a higher risk of error.

Implementing memory retrieval means that the code writing agents are shown examples of successful scripts before writing their own code. This grounds the agent response in code that has definitely worked before. The memory retrieval system has not been tested against semantically different inputs yet. Right now, the code writing agents are being shown very similar retrieved memories every time. It is expected that this might break if the user asks a question using very



(a) 2D axis of rotation centering without memory retrieval.



(b) 2D axis of rotation centering with memory retrieval.

Figure 5: Multi agent system performance on 20 random samples of the 2D axis of rotation centering task with and without memory retrieval.

different language/semantics than that stored in the agent memory.

Examples of successful and unsuccessful 3D scout and zoom workflows can be seen in Figure 7. For successful workflows, the magnification was correctly calculated and set to within 1% of the user-requested FOV. Unsuccessful workflows usually fell into one of three categories: a) Because the acquisition settings (X-ray voltage/power, exposure time, binning etc) were not specified in the user request, the Instrument agent occasionally chose poor settings, resulting in unusable data (bottom right corner of Figure 7, b) segmentation by the analysis agent produced incorrect measurement of sample centroid or size, resulting in the sample being un-centered in the 3D volume (top right corner of Figure 7, or c) segmentation by the analysis agent failed altogether, causing all downstream agent tasks to likewise fail.

4.7 Common Errors

As with many of the examples given in the literature review, this system suffered from prompt brittleness. All of the input prompts for the 2D task were semantically identical. Testing is currently underway to expand the syntax variation in input prompts. There was wider variation in the 3D scout and zoom workflow; the user had to specify a different magnification value or pixel size for each different workflow. This may have contributed to the higher error rate when memory retrieval was not implemented.

There is one common error that persists across both workflows: one agent will create a file (such as JSON with image metadata) and another agent is unable to locate the file. The core issue is filename communication among agents. Sometimes, the orchestrator will fail to specify what name to give a file, so the agent will create its own name. If the agent fails to tell the orchestrator what the filename is, then all downstream agents will not know where to find the file. Almost all code writing errors in Figure 5 and 6 are caused by the agent not being provided a correct filename. Recent advancements in protocols for AI agents, such as Anthropic's **Model Context Protocol** have been developed specifically for these issues [15].

4.8 Summary

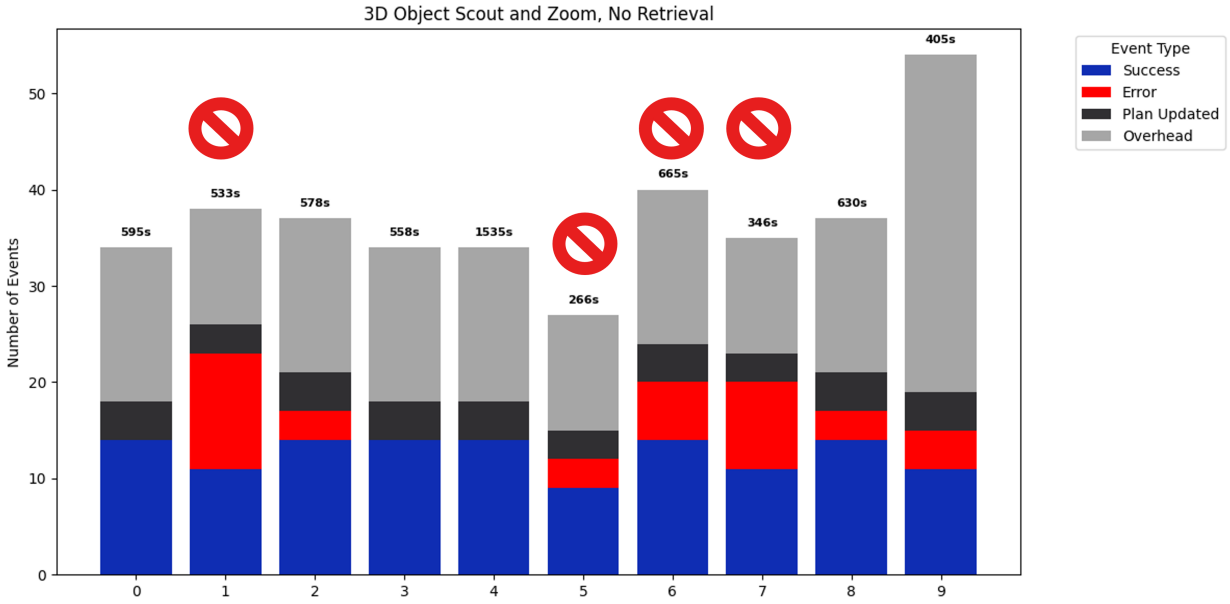
This prototype demonstrates that a multi-agent architecture powered by LLMs can control complex laboratory instruments using only natural language inputs. The system performed reliably on simple workflows and showed substantial improvement on complex workflows when memory retrieval was enabled. These results highlight the practicality of multi-agent systems for real-world scientific instrumentation, while also demonstrating that significant engineering obstacles still exist.

5 Outlook and Discussion

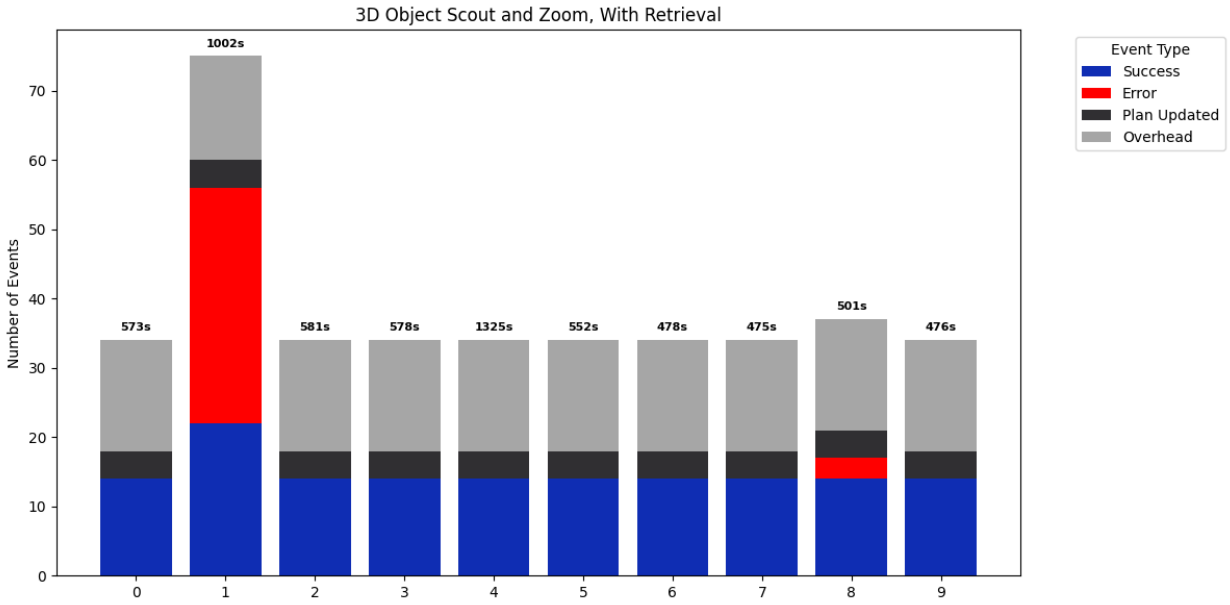
The integration of large language models (LLMs) into laboratory automation systems, particularly through multi-agent architectures, opens a new frontier in scientific research. These systems offer the promise of rapid iteration, flexible natural language interfaces, and adaptive planning across a wide variety of scientific instruments. However, the path to widespread adoption must be guided by careful attention to engineering discipline, safety, and scientific reproducibility.

5.1 Safety Considerations for Physical Instrumentation

Unlike digital assistants or web-based agents, LLM-driven instrument controllers operate in the physical world. This introduces the risk of real-world damage - to equipment, samples, or even



(a) 3D scout and zoom task without memory retrieval.



(b) 3D scout and zoom task with memory retrieval.

Figure 6: Multi agent system performance on 10 random samples of a 3D scout and zoom operation both with and without memory retrieval. Bars with red circles above them indicate tasks that failed entirely (the workflow had to be stopped).

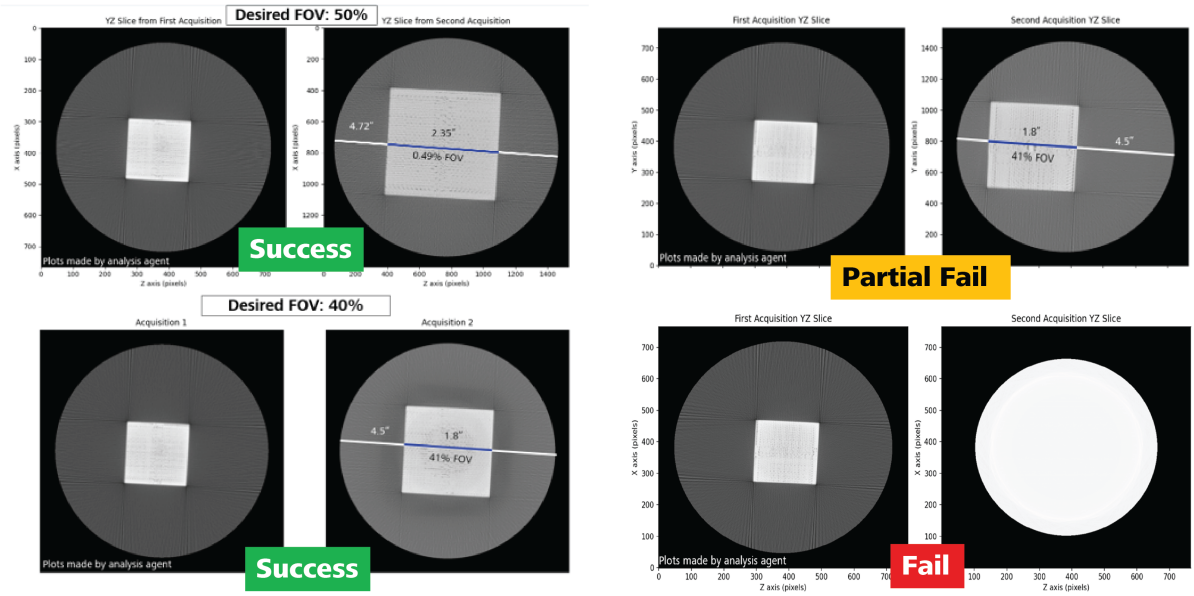


Figure 7: Taxonomy of successful and unsuccessful 3D scout and zoom tasks. Each image shows a single 2D slice (in the YZ plane) through the 3D volume. The field of view (FOV) is calculated by what percent of the volume is occupied by the sample.

personnel - if commands are misinterpreted, poorly validated, or executed at the wrong time.

To mitigate these risks, additional engineering controls must be implemented: Only allow the LLM to emit commands that fall within a predefined whitelist of safe API calls. For example, scanning commands may be allowed, but calibration or configuration changes may be disallowed. Engineers should use hard-coded or rule-based validators to check LLM-generated scripts for dangerous values (e.g., motion beyond travel limits, unsupported voltages).

Additional hardware safety checks may be necessary as well. The agentic system should integrate physical or firmware-based constraints (e.g., motion stops, beamline shutters) to prevent unsafe operations, regardless of software output. To avoid dangerous code from being executed consider running generated code in containerized or virtualized environments where its effects can be simulated or bounded.

An additional safeguard is using a human-in-the-loop architecture for irreversible or safety-critical steps. For instance, the system may pause and prompt the user for confirmation before executing any code that manipulates the instrument's core configuration or exposes a sample to high energy. Human-in-the-loop can remove some autonomy from the system. It is important to identify which safety-critical steps require human validation and leave safer tasks for the agentic system to perform.

5.2 Scientific Reproducibility and Transparency

One of the most compelling features of multi-agent LLM systems is their potential to enhance scientific reproducibility. Unlike ad hoc or manual workflows, LLM agents can: Log all decisions, inputs, code, metadata, and results; Capture structured memory of each task; and provide consistent responses to semantically similar requests. This enables future researchers to replicate not just the final results but the full chain of reasoning and execution that led to them.

However, reproducibility is also at risk with agentic workflows. Risks include an overreliance

on opaque LLM decision-making. Colloquially, this behavior is often called "vibe-coding." Users of complex instruments making decisions without full knowledge of how the instrument operates, or how data is processed, can easily lead to false conclusions or confusing results. Novice users of agentic instruments may have difficulty in understanding autogenerated code or parameters. Safeguards should be in place either a) provide explainable decision making processes so the user understands instruments actions, or b) prevent the user/agent from making ill-informed decisions.

There are several practices that can re-inforce reproducability from agentic instrument workflows. There must be a high level of transparency in agent decision making. All agent outputs should be inspectable and traceable. Users must have access to the exact code and logic used. Each agent, prompt schema, and model version should be logged with each task to help reproduce workflow successes **and** errors. While proprietary models may not disclose their full training sets, any fine-tuning or embedding corpus used by local agents (e.g., instrument manuals) should be cited and versioned. Finally, override systems should always be in place and no agent should have absolute autonomy. Users should be able to inspect and manually adjust any step in the plan.

5.3 Future Outlook

There are many exciting avenues for research and engineering developments in agentic instrument control.

Increasing the usability of complex instruments by lowering the technical barrier to entry is an obvious path forward. Human-AI collaborative hybrid workflows, where the system prompts for user instructions or assists in exploratory tasks, are already being developed across many complex applications and not just laboratory instruments.

Once single-instrument agentic workflows have been proven and fully engineered, the next step is multi-instrument multi-agent laboratory controllers. Multi-agent frameworks that schedule and coordinate actions across multiple instruments (e.g., microscopy + spectroscopy) can further improve data collection and experimental productivity. Multi-instrument multi-agent workflows can also assist in live data processing and analysis. For example, agents can replan experiments based on intermediate data analysis during a workflow.

A current gap in the research is open benchmarks for agentic systems. The field needs development of reproducible task benchmarks and test suites that can be compared across LLM model types, agentic architectures, and instrumentation. This will provide users with indicators for how well agentic workflows perform for various tasks.

Ultimately, multi-agent LLM architectures may become foundational to the next generation of intelligent labs. However, this future must be grounded in safeguards and transparency to ensure that automation enhances, not undermines, the integrity of scientific discovery.

Generative AI Statement

Portions of this manuscript were drafted and refined with the assistance of OpenAI's GPT-4. Specifically, generative AI was used to synthesize literature summaries, outline technical concepts, suggest phrasing, and assist in formatting LaTeX code. All outputs generated by AI were reviewed and edited for accuracy, clarity, and technical correctness by the authors.

No data, results, or analyses were generated solely by AI systems; all empirical content, including case studies and prototype descriptions, reflects original work by the authors. The use of generative AI was limited to augmenting the writing and organization of the manuscript. Responsibility for the content and conclusions of this paper rests entirely with the author.

References

[1] OpenAI. "GPT-4 Technical Report". In: *arXiv preprint* (2023). Accessed: 2025-07-23. DOI: https://doi.org/10.48550/arXiv.2303.08774. URL: https://openai.com/research/gpt-4.

[2] Aakanksha Chowdhery et al. "PaLM: Scaling Language Modeling with Pathways". In: *arXiv preprint* (2022). DOI: 10.48550/arXiv.2204.02311. arXiv: 2204.02311 [cs.CL]. URL: https://arxiv.org/abs/2204.02311.

[3] Hugo Touvron et al. "LLaMA: Open and Efficient Foundation Language Models". In: *arXiv preprint* (2023). DOI: https://doi.org/10.48550/arXiv.2302.13971. arXiv: 2302.13971 [cs.CL]. URL: https://arxiv.org/abs/2302.13971.

[4] DeepSeek-AI. "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning". In: *arXiv preprint* (2025). Submitted on 22 Jan 2025. DOI: https://doi.org/10.48550/arXiv.2501.12948. arXiv: 2501.12948 [cs.CL].

[5] Indrajeet Mandal et al. *Autonomous Microscopy Experiments through Large Language Model Agents*. 2025. arXiv: 2501.10385 [cs.CY]. URL: https://arxiv.org/abs/2501.10385.

[6] Daniil A. Boiko et al. "Autonomous chemical research with large language models". In: *Nature* 624 (2023), pp. 570–578. DOI: https://doi.org/10.1038/s41586-023-06792-0.

[7] Tao Song et al. "A Multiagent-Driven Robotic AI Chemist Enabling Autonomous Chemical Research On Demand". In: *Journal of the American Chemical Society* 147.15 (2025), pp. 12534–12545. DOI: https://doi.org/10.1021/jacs.4c17738.

[8] Hanqing Yang et al. "LLM-Powered Decentralized Generative Agents with Adaptive Hierarchical Knowledge Graph for Cooperative Planning". In: *arXiv preprint* (2025). arXiv: 2502.05453 [cs.AI]. URL: https://arxiv.org/abs/2502.05453.

[9] Yingxuan Yang et al. "AgentNet: Decentralized Evolutionary Coordination for LLM-based Multi-Agent Systems". In: *arXiv preprint* (2025). arXiv: 2504.00587 [cs.MA]. URL: https://arxiv.org/abs/2504.00587.

[10] LangChain AI. *LangGraph Multi-Agent Concepts*. https://langchain-ai.github.io/langgraph/concepts/multi_agent/. Accessed: 2025-08-08. 2024.

[11] Ollama Inc. *Structured Outputs*. https://ollama.com/blog/structured-outputs. Accessed: 2025-07-24. Dec. 2024.

[12] OpenAI. *Structured Outputs Guide (API Mode: Responses)*. https://platform.openai.com/docs/guides/structured-outputs?api-mode=responses. Accessed: 2025-07-24. 2025.

[13] Pydantic AI Team. *Output*. https://ai.pydantic.dev/output/. Accessed: 2025-07-24. 2025.

[14] Jialin Wang and Zhihua Duan. *Agent AI with LangGraph: A Modular Framework for Enhancing Machine Translation Using Large Language Models*. 2024. arXiv: 2412.03801 [cs.CL]. URL: https://arxiv.org/abs/2412.03801.

[15] Model Context Protocol. *Get started with the Model Context Protocol (MCP)*. Accessed: 2025-07-28. 2025. URL: https://modelcontextprotocol.io/docs/getting-started/intro.

[16] Yongliang Shen et al. "HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face". In: (2023). arXiv: 2303.17580 [cs.CL]. URL: https://arxiv.org/abs/2303.17580.

[17] Naomi Yoshikawa, Leila Darvish, and et al. "CLAIRify: Structured Prompting and Verification for Language-Driven Lab Automation". In: (2023). Published online. URL: https://acrad.github.io/clairify/.

[18] Patrick Lewis et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: 2021. arXiv: 2005.11401 [cs.CL]. URL: https://arxiv.org/abs/2005.11401.

[19] LangChain AI. *LangGraph Memory Concepts*. https://langchain-ai.github.io/langgraph/concepts/memory/. Accessed: 2025-08-08. 2024.

[20] Ozan Gokdemir et al. "HiPerRAG: High-Performance Retrieval Augmented Generation for Scientific Insights". In: *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '25)*. PASC '25. New York, NY, USA: ACM, June 2025, pp. 1–13. DOI: 10.1145/3732775.3733586.

[21] Remco Guerts et al. "Revolutionizing Electron Microscopy Through Intuitive Language-Driven Interfaces: The Emergence of the EM CoPilot". In: *BIO Web of Conferences: Electronic Materials Conference* 129.10018 (2024). DOI: https://doi.org/10.1051/bioconf/202412910018.

[22] Remco Geurts and Pavel Potocek. "Enhancing Automation and Accessibility in Electron Microscopy through Generative AI". In: *Proceedings of the 13th Asia-Pacific Microscopy Conference* (Jan. 2025). URL: https://www.scienceopen.com/hosted-document?doi=10.14293/APMC13-2025-0306.

[23] Remco Guerts. "LLM-Driven Automation in FIB-SEM: Enhancing Microscope Control with AutoScript and Vision-Based AI". In: *Microscopy and Microanalysis* 31 (Supplement 1 July 2025). DOI: https://doi.org/10.1093/mam/ozaf048.1088.

[24] Shray Mathur et al. "VISION: a modular AI assistant for natural human-instrument interaction at scientific user facilities". In: *Machine Learning: Science and Technology* 6.2 (June 2025), p. 025051. DOI: 10.1088/2632-2153/add9e4. URL: http://dx.doi.org/10.1088/2632-2153/add9e4.

[25] Matthijs Douze et al. "The Faiss library". In: (2025). arXiv: 2401.08281 [cs.LG]. URL: https://arxiv.org/abs/2401.08281.

[26] Shunyu Yao et al. "ReAct: Synergizing Reasoning and Acting in Language Models". In: (2023). arXiv: 2210.03629 [cs.CL]. URL: https://arxiv.org/abs/2210.03629.

[27] Noah Shinn et al. "Reflexion: Language Agents with Verbal Reinforcement Learning". In: (2023). arXiv: 2303.11366 [cs.AI]. URL: https://arxiv.org/abs/2303.11366.

[28] Qingyun Wu et al. *AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework*. 2023. DOI: 10.48550/arXiv.2308.08155. arXiv: 2308.08155 [cs.AI]. URL: https://arxiv.org/abs/2308.08155.

[29] CrewAI Documentation. *Introduction*. https://docs.crewai.com/en/introduction. Accessed: 2025-07-29. CrewAI, 2025.

Appendix A: Useful Tools and Packages

Building a robust, scalable multi-agent framework for laboratory instrument control requires a combination of modern language model APIs, Python execution tools, retrieval systems, data handling layers, and interface components. This section catalogs the most useful libraries, frameworks, tools, and resources, based on our implementation and survey of the field.

Language Model APIs and Libraries

- `Ollama`: software tool for downloading, customizing, and running large language models locally on a computer
- `HuggingFace Hub`: cloud based service for hosting models, code repositories, and web applications
- `LangChain`: Framework for managing multi-agent chains, prompt templates, memory, and RAG pipelines.
- `Pydantic`: For defining structured JSON schemas for agent input/output validation.

Retrieval and Memory Systems

- `FAISS`, `Chroma`, `Weaviate`: Vector embedding databases and similarity search engines used to build semantic memory banks.
- `SentenceTransformers` and `text-embedding-3-small/large`: Pretrained models for generating dense text embeddings.
- `MongoDB`: Lightweight structured database for tracking agent interactions, metadata, and logs.

Relevant Papers and Frameworks

- **ReAct** [26]: Reasoning + action prompting framework foundational to many agent designs.
- **HuggingGPT** [16]: Demonstrated centralized LLM orchestration of specialized tools.
- **ChemAgents** [7]: A multi-agent chemistry research assistant with persistent memory and simulation planning.
- **AILA** [5]: Benchmarked multi-agent LLM system for AFM with detailed error analysis.
- **Reflexion** [27]: Memory-driven improvement of agent behavior over time.
- **LangGraph** [14], **AutoGen** [28], **CrewAI** [29]: Multi-agent orchestration libraries with strong community support.

Appendix B: Code Examples

Parsing a Structured Output in Python

Listing 2: Parsing a structured output from a language model in a Python program

```
# Example structured output from LLM (as JSON string)
llm_output = '''
{
    "status": "success",
    "code": "move_stage(0,100,0)",
    "user_message": "Hello, your task completed.",
    "embeddings": [0.1, 0.3, -0.2, 0.8],
    "debug_info": "Model:v1"
}
```

```
'''

import json
import instrumentAPI

# Parse the structured output
response = json.loads(llm_output)

# Use fields from the output
if response["status"] == "success":
    print(response["user_message"]) # print the user message
    instrumentAPI(response["code"]) # pass the instrument API code to
        another program to execute
    average = sum(response["embeddings"]) / len(response["embeddings"]) #
        process data values
    print("Average embedding value:", average)
```

Generic System Prompt Examples

Orchestrator

You are the Orchestrator Agent in a scientific laboratory automation system. Your role is to take natural language instructions from the user and convert them into a structured multi-step workflow. Each step should call a specific agent (e.g., instrument, analysis, librarian), with a clear instruction in plain English.

You must respond in valid JSON using the following schema:

```
{
  "plan": [
    {"agent_to_call": "instrument", "instruction": "..."},
    {"agent_to_call": "analysis", "instruction": "..."}
  ]
}
```

Guidelines:

- Interpret vague user requests based on memory and context.
- If the user instruction is incomplete, attempt a reasonable default.
- If an agent fails or returns unexpected output, replan using updated context.
- Keep steps concise and actionable.
- Do not write any executable code yourself.
- Do not call undefined agents.

If you do not understand the instruction or cannot proceed, return a plan with a single step asking the user to clarify.

Instrument Operation Agent

You are the Instrument Agent in a multi-agent control system for a laboratory microscope. Your task is to generate Python 3 code that interacts with the microscope API to carry out specific hardware instructions.

You will be given a user task (via the orchestrator), along with metadata such as previous image paths, current stage position, and geometry parameters.

You must return a structured JSON object with the following schema:

```
{
  "status": "success",
  "code": "...",
  "filepaths": [...],
  "metadata": {...},
  "orchestrator_message": "..."
}
```

Guidelines:

- Only use supported functions from the Instrument API (provided in docs).
- Use absolute stage coordinates.
- Always save output metadata (e.g., pixel size, exposure time) to a '.json' file.
- Never overwrite previous files unless specified.
- Do not include import statements unless necessary.
- Be concise and robust in your code.

If input parameters are ambiguous, return an error message in the 'status' field and request clarification from the orchestrator.

Librarian Agent (as part of a RAG pipeline)

You are the Librarian Agent in a scientific multi-agent system. Your task is to answer technical questions using only the text chunks retrieved from a vector database. These chunks are taken from trusted documents such as instrument manuals, standard operating procedures (SOPs), and peer-reviewed scientific literature.

You will be given:

- A user question or system instruction
- A list of text passages retrieved from the document corpus
- Optionally, previous memory logs or metadata

Your goal is to return a grounded, accurate answer based entirely on the retrieved documents. You must follow these guidelines:

- Base your answer strictly on the retrieved documents. Do not guess or invent information.
- Use clear, concise scientific language.

- If the answer is not present in the retrieved material, say: "The retrieved documents do not contain a definitive answer to this question."
- If multiple sources provide conflicting information, summarize the discrepancy.
- Include citations or file names when possible for transparency.
- Do not generate executable code unless explicitly requested.

Respond in the following structured JSON format:

```
{
  "status": "success",
  "answer": "...",
  "sources_used": ["manual_page_12.txt", "xrm_paper_segmentation2024.pdf"
     ],
  "orchestrator_message": "Knowledge retrieved from RAG for titanium scan
     setup."
}
```

Your responses will be appended to other agent prompts, so be clear and specific. Always assume that your answer may influence a real laboratory workflow.

Parsing and Executing Instrument API Commands from an LLM Response

Listing 3: An example of extracting Python code from a structured LLM response, saving it in a .py file, and executing the code.

```
# Example structured output containing executable code
llm_output = '''
{
    "status": "success",
    "code": "from instrumentAPI import move_stage\n    new_pos =
       move_stage(100,-10,5) ",
    "user_message": "Move the stage to position (100,-10,5)."
}
'''

import json
import subprocess

# Parse the output
response = json.loads(llm_output)

# Check status and extract code
if response["status"] == "success":
    print(response["user_message"])  # Inform user
    python_code = response["code"]

    # Write code to a file
    with open("generated_script.py", "w") as f:
        f.write(python_code)

    # Execute the code using subprocess
```

```
    result = subprocess.run(["python", "generated_script.py"],
        capture_output=True, text=True)

    # Print the result
    print("Output from generated code:")
    print(result.stdout)
```

Using a Middle Agent between the Orchestrator and a Digital Twin or Hard-Coded Module

Listing 4: The LLM acts a middle agent, taking a loosely formatted instruction from the Orchestrator and converting it into a structured output that can be given to a hard-coded program like a digital twin.

```
# Simulated orchestrator command (vague or loosely formatted)
orchestrator_command = "move the print head to 25mm height and start
    heating to 200C"

# Simulated LLM output - parsing the vague command into structured format
# In practice, this would be produced by an actual LLM call.
structured_output = {
    "action": "set_print_head",
    "parameters": {
        "z_position_mm": 25.0,
        "temperature_C": 200
    }
}

# Hard-coded digital twin function with strict input format
def digital_twin_interface(action: str, parameters: dict):
    if action == "set_print_head":
        z = parameters.get("z_position_mm")
        temp = parameters.get("temperature_C")
        print(f"[Digital Twin] Moving print head to {z} mm and heating to
            {temp} deg C.")
    else:
        print("[Digital Twin] Unknown action.")

# Middle agent: passes structured command to digital twin
if structured_output["action"] and structured_output["parameters"]:
    digital_twin_interface(
        action=structured_output["action"],
        parameters=structured_output["parameters"])
```