OPERATING ROBOTIC LABORATORIES WITH LARGE LANGUAGE MODELS AND TEACHABLE AGENTS

Aikaterini Vriza*
Argonne National Laboratory
avriza@anl.gov

Michael H Prince
Argonne National Laboratory
mprince@anl.gov

Henry Chan*
Argonne National Laboratory
hchan@anl.gov

Tao Zhou
Argonne National Laboratory
tzhou@anl.gov

Matthew J Cherukara*
Argonne National Laboratory
mcherukara@anl.gov

ABSTRACT

Advanced scientific user facilities, including self-driving laboratories, are revolutionizing scientific discovery by automating repetitive tasks and enabling rapid experimentation. However, these facilities must continuously evolve to support new experimental workflows, adapt to diverse user projects, and meet growing demands for evermore sophisticated instrumentation. This continuous development introduces significant operational complexity, necessitating a focus on usability, reproducibility, and intuitive human-instrument interaction. In this work, we explore the integration of agentic AI, powered by Large Language Models (LLMs), as a transformative tool to achieve this goal. We present our approach to developing a pipeline for operating a robotic station dedicated to the design of novel materials. Specifically, we evaluate the potential of various LLMs as trainable scientific assistants for orchestrating complex, multi-task workflows, optimizing their performance through human input and iterative learning. We demonstrate the ability of AI agents to bridge the gap between advanced automation and user-friendly operation, paving the way for more adaptable and intelligent scientific facilities.

1 INTRODUCTION

The design, synthesis, characterization, and testing of materials often involve a multi-step sequence of complex actions. Although self-driving laboratories have made significant progress in automating experimental procedures, achieving true autonomy in scientific discovery involving tasks such as experimental workflow design, data analysis, execution planning, and decision making remains an active challenge that will likely continue to require human feedback and expert knowledge (Hung et al. (2024)). The emergence of Large Language Models (LLMs) (OpenAI (2024), Editorial (2023),Luo et al. (2024)), has created new opportunities for human-AI collaboration in scientific facilities, enabling safer and more efficient laboratory operations. While significant progress has been made in adapting LLMs for various scientific applications (Prince et al. (2024), Van Herck et al. (2025)), substantial challenges remain in deploying them as effective agents for real-world laboratory experimentation in operating within science laboratories and user facilities (Mathur et al. (2024)).

Tasks that human scientists perform intuitively, such as manipulating laboratory equipment, present significant challenges to autonomous systems. For example, picking up and moving a vial requires precise coordination of multiple components like grippers and vial holders, along with accurate spatial awareness. Traditional robotic systems rely on predefined functions for these operations, limiting their flexibility and ability to adapt to different experimental environments. Moreover, while existing LLM agents can utilize predefined functions as tools, they often lack the capability to incorporate user feedback and autonomously design experimental processes.

In this work, we develop and evaluate an agentic system for managing sequential laboratory operations, where task order is crucial for experimental success. The system handles increasingly complex tasks initiated by human researchers. An important aspect of this study is the ability of the system to learn from laboratory users through in-context learning during conversations. This teachability component stores human-AI interactions in a vector database as long-term memory, retrieving relevant information as needed rather than loading all memories into the context window. This approach enables the agent to retain and apply user-taught skills across multiple sessions. We evaluated the agent's ability to learn from human scientists and their potential for self-improvement.

2 Related work

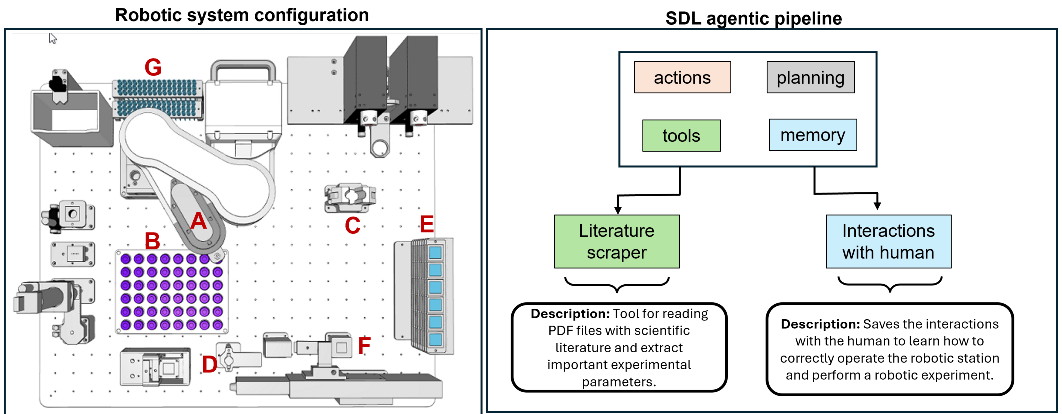
Significant progress has been made in the deployment of LLMs in scientific research, from decision-making and literature screening to the execution of experimental protocols (Ruan et al. (2024), Birhane et al. (2023), White et al. (2023)). Research teams have successfully expanded LLM capabilities by integrating them with diverse sets of tools (Boiko et al. (2023), Bran et al. (2024)).

The field is now advancing toward collaborative multi-agent systems that enhance the capabilities of LLMs through iterative feedback, role specialization, and coordinated teamwork (Ansari et al. (2024), Ma (2025)). These systems draw inspiration from human scientific discovery processes and are designed to complement human creativity and expertise with AI's ability to analyze vast datasets, navigate hypothesis spaces and execute repetitive tasks. The adoption of pre-built frameworks such as LangGraph (langchain ai), Autogen (Wu et al. (2024))and Swarm (OpenAI)has significantly streamlined the integration of LLMs with scientific workflows, enabling efficient coordination between LLM agents and external tools (Yin et al. (2024)). AI agents can impact areas across various scientific disciplines. In biology, for instance, biomedical AI agents are being applied to areas such as virtual cell simulation, programmable control of phenotypes, cellular circuit design, and the development of novel therapies Gao et al. (2024). In materials science, platforms like AtomAgents, are being developed as agentic systems for knowledge retrieval, multi-modal data integration, physics-based simulations, and comprehensive results analysis (Ghafarollahi & Buehler (2024)).

3 Methods

The core of our agentic pipeline is the Autogen (AG2) framework (Wu et al. (2024)), which serves as the backbone for orchestrating an autonomous experiment to create a polymer thin film (See Appendix A.1). The robotic environment consists of an N9 robot from North Robotics equipped with finger and vacuum grippers to transfer material samples across the modules. The pipeline leverages autogen's built-in capabilities, such as linter tool for code checking and the executor platform for running the generated code. The specialized agents are presented in detail in Table 1. AI agents have access to the N9 robot's operation commands, station layout, and task descriptions (See Appendix A.2). Given a task, the agents can write, review, and execute the code as Python scripts and request human input and approval before execution to the robotic station. To further enhance functionality and reliability, we integrated a teachability component (See Appendix A.3). In this approach, agent capabilities are updated by appending new instruction to the system message, integrating storage and retrieval functionalities, and interacting with a background agent that decides whether the human provided information is important and should be stored. In that way, any new interaction with the human researcher that provides significant guidelines for the execution of the experimental protocol is stored as an input-output pair in a local vector database (See Figure 5). This process enables the agents to recall and apply previously learnt information when facing similar tasks, thereby supporting long-term learning and adaptability.

We evaluated the ability of our agentic system to plan polymer thin film design experiments by using robotic operation functions and writing the code for the N9 robot (Figure 1). They are then provided with tasks of increasing complexity, ranging from simple operations such as moving a vial to more advanced tasks such as activating a vacuum system or extracting optimal experimental parameters from the literature to guide robotic operations. Given these tasks, the pipeline decided on which agents should participate, generates, and validates operational code (See Appendix A.4). Before execution, it requires human approval to ensure safety.



Equipment

A. Gripper
B. Vial with polymer A
C. Clamp holder
D. Vacuum gripper
E. Substrate rack
F. Blade coating station
G. Pipette tips rack

System files

- N9 robot python operation commands
- Layout of vials and substrates

	Tasks	Complexity	No of sequential steps
Task 1	Pick up the vial with PEDOT:PSS and move it to the clamp.	Easy	8
Task 2	Pick up a substrate and move it to the coating station.	Medium	10
Task 3	Create a polymer film on a substrate using PEDOT:PSS.	Hard	25

Figure 1: Configuration of the robotic environment of a self-driving laboratory, including the main equipment components used and the system files that are provided as context to the agentic pipeline. The agentic pipeline is integrated external tools and memory.The table summarizes the evaluation tasks including their complexity and number of required steps.

Table 1: Available agents and their operating capabilities

Agent	**Agentic Tasks**
Code writer	Write the code provided the relevant system files
Code critic	Check the code written from the code writer and provides feedback.
Administrator	Interacts with the other agents. Executes the code in the robotic working environment and interacts with the human for feedback.
Paper scraper	Scrapes research articles for relevant information extraction.
Manager	Distributed the tasks among the agents and organises the feedback .
Teachability	Memory retrieval of previously saved interactions between human and agents.

4 Results

We evaluated the performance of representative LLM assistants, including GPT-3.5, GPT-4o and Claude 3.5 Sonnet, on three autonomous sequential tasks of increased complexity (See Figure 1 and Appendix A.5). The evaluation metrics are categorized as follows: i) **Code quality check:** verifying that the agent performs code quality analysis for potential errors, stylistic inconsistencies, and best-practice violation. ii) **Code correctness:** evaluating the functionality of the generated code and checking whether the sequential steps are performed in the correct order. iii) **Code execution:** evaluating the functional correctness of the code, basically measuring in how many steps it will stop working in the real robotic environment, iv) **Code repeatability:** evaluating whether the same code is generating after running the same prompt for three times. v) **Code reproducibility:** evaluating whether the generated code is the same after modifying the task prompts whilst keeping the main task the same. The results are demonstrated in Table 2 for the cases where no human input is provided to the models. A detailed example of how the evaluation is performed is provided in Appendix A.5 and Table 4. Whereas all agents are struggling with the higher complexity task, such as Task 3, when human feedback was provided and stored as memories, a significant improvement was observed in all tasks as shown in Table 3. The schematic of the agentic pipeline operating the highest complexity task, defined as Task 3 is shown in Figure 2.

Table 2: Evaluation results across different models and tasks without receiving human feedback.

	GPT-4o-mini			GPT-4o			Claude 3.5 Sonnet		
	Task1	Task2	Task3	Task1	Task2	Task3	Task1	Task2	Task3
Code quality check	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Code correctness (%)	87.5	70	44	100	90	48	75	60	64
Code execution (%)	75	50	24	100	70	39	50	60	32
Code repeatability	100	100	90	100	100	100	100	90	25
Reproducibility	100	100	60	100	90	75	90	80	25

Table 3: Evaluation results across different models and tasks after receiving human feedback as memory

	GPT-4o-mini			GPT-4o			Claude 3.5 Sonnet		
	Task1	Task2	Task3	Task1	Task2	Task3	Task1	Task2	Task3
Code quality check	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Code correctness (%)	100	100	84	100	100	100	100	100	100
Code execution (%)	100	100	84	100	100	100	100	100	100
Code repeatability	100	90	90	100	100	100	100	100	100
Reproducibility	100	90	80	100	100	100	100	100	100

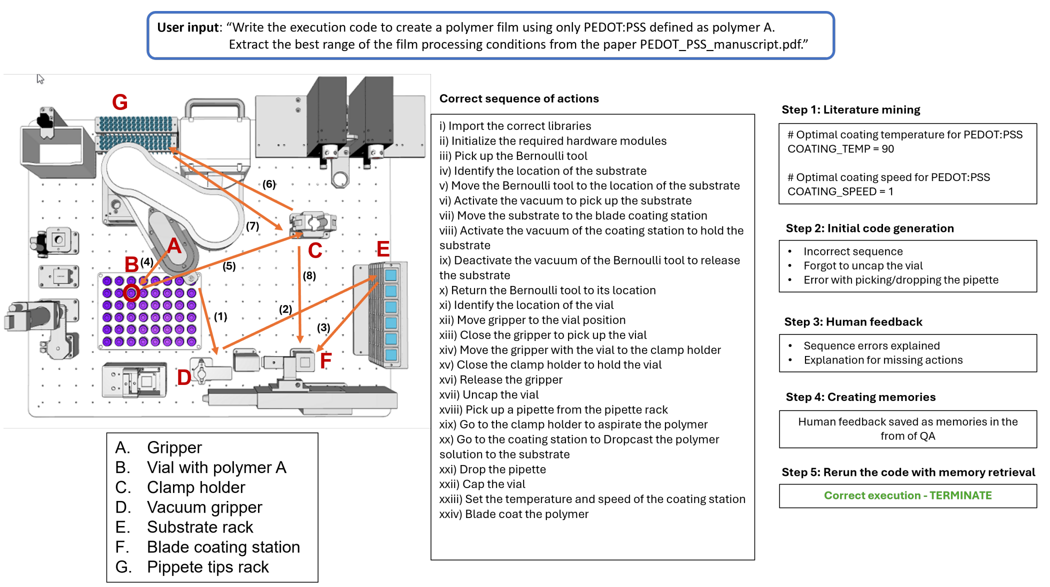


Figure 2: Schematic of the agentic pipeline to operate a highly complex set of robotic action given a user prompt. The system had to first read a scientific paper and identify the conditions that lead to a good quality film. The initial code was corrected with human feedback so that the agents will understand the correct sequence of steps to create the polymer film, which include: 1) Use the vacuum gripper to pick up an available substrate from the substrate rack, 2) Place the substrate to the blade coating station and release the vacuum gripper, 3) Pick up the polymer from the vials rack and move it to the clamp, 4) Uncapping the clamp and aspirate the polymer with the pipette, 5) dropcast the polymer to the substrate and blade coat it.

5 DISCUSSION

In this work, we explored the capabilities and reliability of LLM-based scientific agents in operating a robotic platform for materials design. By integrating multiple LLMs into an autonomous multi-agent system, we evaluated their performance in a real-world robotic environment developed for creating polymer thin films. In addition, we further investigated the impact of incorporating user guidelines into the system to enhance its ability to handle complex design and operational tasks.

A key focus of this study was the advantage provided by trainable agents in the operation of complex scientific workflows. Our results indicate that the agentic system is capable of learning from user interactions with the platform, refining its understanding of processes over time, and generating reliable code for robotic operations. This adaptability highlights the potential of trainable agents to improve the autonomy of self-driving laboratories, where human-in-the-loop interactions play a critical role in guiding and refining AI-driven workflows. While our teachable agent framework reduces the need for continual human intervention by storing and reusing past instructions, complete automation in real-world laboratory and user facility settings requires further safeguards. In particular, instrument drift and other hardware variances require regular calibration to maintain reliability over long experimental cycles. Likewise, reconfigurable workflows may introduce new instruments or procedures that must first be taught before running autonomously. Despite these challenges, once the core pipeline has been aligned with the lab environment and a sufficient body of memorized instructions has been built up, the agent can execute the tasks with minimal oversight.

In our experiments, agents such as GPT-4o and Claude3.5-Sonnet were relatively easy to teach, whereas GTP-4o-mini required more iterative feedback to achieve an equivalent level of understanding. Automation of protocols in an autonomous laboratory requires advanced models that can manage the sequential nature of tasks, perform quality checks, and provide real-time feedback to users. As multi-agent paradigms continue to evolve, they are expected to drive significant advancements in autonomous research, adaptive experimentation, and AI-guided discovery. Our results indicate that agentic systems can serve as a valuable resource for building self-driving laboratories with enhanced autonomy, capable of learning from human expertise and improving over time.

6 ACKNOWLEDGMENTS

Work was performed at the Center for Nanoscale Materials and Advanced Photon Source, both U.S. Department of Energy Office of Science User Facilities, supported by the U.S. DOE, Office of Basic Energy Sciences, under Contract No. DE-AC02-06CH11357.

7 DATA AND CODE AVAILABILITY

All data, code and agentic pipeline used to produce results in this study are publicly available in the following GitHub repository: `https://github.com/katerinavr/SDL-Agents`.

REFERENCES

Mehrad Ansari, Jeffrey Watchorn, Carla E. Brown, and Joseph S. Brown. dziner: Rational inverse design of materials with ai agents, 2024. URL https://arxiv.org/abs/2410.03963. arXiv preprint.

Abeba Birhane, Atoosa Kasirzadeh, David Leslie, and Sandra Wachter. Science in the age of large language models. *Nature Reviews Physics*, 5(5):277–280, May 2023. ISSN 2522-5820. doi: 10.1038/s42254-023-00581-4. URL https://doi.org/10.1038/s42254-023-00581-4.

Daniil A. Boiko, Robert MacKnight, Ben Kline, and Gabe Gomes. Autonomous chemical research with large language models. *Nature*, 624(7992):570–578, December 2023.

M. Bran, A. Cox, S. Schilter, et al. Augmenting large language models with chemistry tools. *Nature Machine Intelligence*, 6:525–535, 2024. doi: 10.1038/s42256-024-00832-8. URL https://doi.org/10.1038/s42256-024-00832-8.

Editorial. Prepare for truly useful large language models. *Nature Biomedical Engineering*, 7: 85–86, 2023. doi: 10.1038/s41551-023-01012-6. URL `https://doi.org/10.1038/s41551-023-01012-6`.

Shanghua Gao, Ada Fang, Yepeng Huang, Valentina Giunchiglia, Ayush Noori, Jonathan Richard Schwarz, Yasha Ektefaie, Jovana Kondic, and Marinka Zitnik. Empowering biomedical discovery with ai agents. *Cell*, 187(22):6125–6151, 2024. doi: 10.1016/j.cell.2024.09.022. URL `https://doi.org/10.1016/j.cell.2024.09.022`.

Alireza Ghafarollahi and Markus J. Buehler. Atomagents: Alloy design and discovery through physics-aware multi-modal multi-agent artificial intelligence. *arXiv*, 2024. URL `https://arxiv.org/abs/2407.10022`.

Linda Hung, Joyce A. Yager, Danielle Monteverde, Dave Baiocchi, Ha-Kyung Kwon, Shijing Sun, and Santosh Suram. Autonomous laboratories for accelerated materials discovery: a community survey and practical insights. *Digital Discovery*, 3(7):1273–1279, 2024. doi: 10.1039/D4DD00059E. URL `http://dx.doi.org/10.1039/D4DD00059E`.

langchain ai. Langgraph. `https://github.com/langchain-ai/langgraph`. Accessed: 2025-02-03.

X. Luo, A. Rechardt, G. Sun, and et al. Large language models surpass human experts in predicting neuroscience results. *Nature Human Behaviour*, 2024. doi: 10.1038/s41562-024-02046-9. URL `https://doi.org/10.1038/s41562-024-02046-9`.

Kangyong Ma. Ai agents in chemical research: Gvim – an intelligent research assistant system. *Digital Discovery*, pp. –, 2025. doi: 10.1039/D4DD00398E. URL `http://dx.doi.org/10.1039/D4DD00398E`.

Shray Mathur, Noah van der Vleuten, Kevin Yager, and Esther Tsai. Vision: A modular ai assistant for natural human-instrument interaction at scientific user facilities. *arXiv preprint arXiv:2412.18161*, 2024. URL `https://doi.org/10.48550/arXiv.2412.18161`.

OpenAI. Swarm agents. `https://github.com/openai/swarm`.

OpenAI. Openai, 2024. URL `https://openai.com`. Accessed: 2025-01-16.

M. H. Prince, H. Chan, A. Vriza, et al. Opportunities for retrieval and tool augmented large language models in scientific facilities. *npj Computational Materials*, 10:251, 2024. doi: 10.1038/s41524-024-01423-2.

Y. Ruan, C. Lu, N. Xu, et al. An automatic end-to-end chemical synthesis development platform powered by large language models. *Nature Communications*, 15:10160, 2024. doi: 10.1038/s41467-024-54457-x. URL `https://doi.org/10.1038/s41467-024-54457-x`.

Joren Van Herck, María Victoria Gil, Kevin Maik Jablonka, Alex Abrudan, Andy S. Anker, Mehrdad Asgari, Ben Blaiszik, Antonio Buffo, Leander Choudhury, Clemence Corminboeuf, Hilal Daglar, Amir Mohammad Elahi, Ian T. Foster, Susana Garcia, Matthew Garvin, Guillaume Godin, Lydia L. Good, Jianan Gu, Noémie Xiao Hu, Xin Jin, Tanja Junkers, Seda Keskin, Tuomas P. J. Knowles, Ruben Laplaza, Michele Lessona, Sauradeep Majumdar, Hossein Mashhadimoslem, Ruaraidh D. McIntosh, Seyed Mohamad Moosavi, Beatriz Mouriño, Francesca Nerli, Covadonga Pevida, Neda Poudineh, Mahyar Rajabi-Kochi, Kadi L. Saar, Fahimeh Hooriabad Saboor, Morteza Sagharichiha, K. J. Schmidt, Jiale Shi, Elena Simone, Dennis Svatunek, Marco Taddei, Igor Tetko, Domonkos Tolnai, Sahar Vahdatifar, Jonathan Whitmer, D. C. Florian Wieland, Regine Willumeit-Römer, Andreas Züttel, and Berend Smit. Assessment of fine-tuned large language models for real-world chemistry and material science applications. *Chem. Sci.*, 16:670–684, 2025. doi: 10.1039/D4SC04401K. URL `http://dx.doi.org/10.1039/D4SC04401K`.

Andrew D. White, Glen M. Hocky, Heta A. Gandhi, Mehrad Ansari, Sam Cox, Geemi P. Wellawatte, Subarna Sasmal, Ziyue Yang, Kangxin Liu, Yuvraj Singh, and Willmor J. Peña Ccoa. Assessment of chemistry knowledge in large language models that generate code. *Digital Discovery*, 2:368–376, 2023. doi: 10.1039/D2DD00087C. URL `http://dx.doi.org/10.1039/D2DD00087C`.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang (Eric) Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Ahmed Awadallah, Ryen W. White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation. In *COLM 2024*, August 2024. URL `https://www.microsoft.com/en-us/research/publication/autogen-enabling-next-gen-llm-applications-via-multi-agent-conversation-framework/`.

Xiangyu Yin, Chuqiao Shi, Yimo Han, and Yi Jiang. Pear: A robust and flexible automation framework for ptychography enabled by multiple large language model agents. *ArXiv*, abs/2410.09034, 2024. URL `https://api.semanticscholar.org/CorpusID:273323364`.

A Appendix

A.1 Workflow description

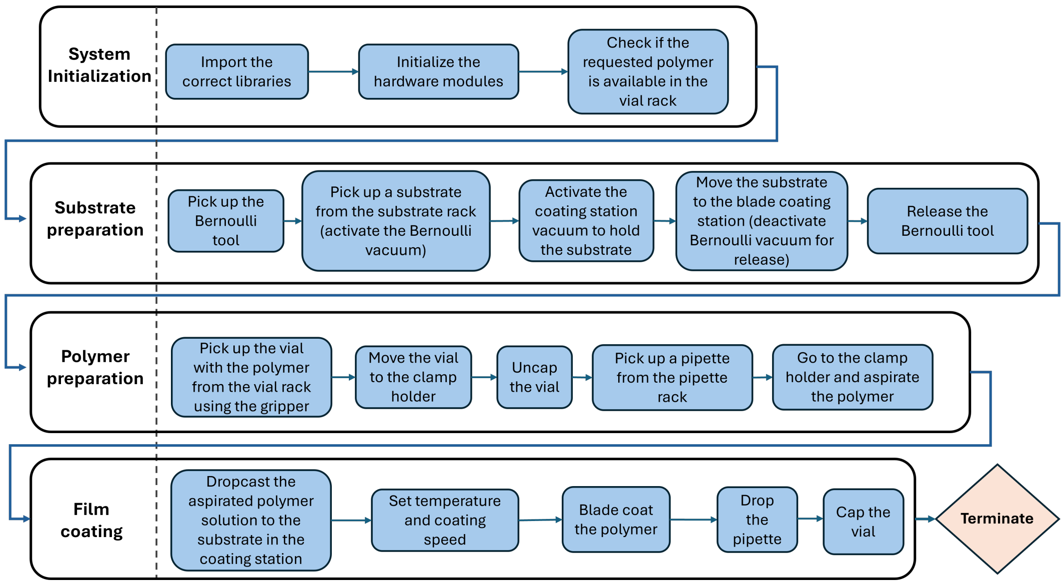


Figure 3: Detailed pipeline with the required sequential steps to create a polymer thin film using the N9 robotic station.

```
# These are the functions that should always be imported before running the N9 robotic system
import loca  # location information
import pandas as pd
import robotics as ro
from robotics import procedure as proc
import rack_status

# Vaccum functions
c9.set_output('substrate_tool', True) # when True activate the vaccum on the bernoulli gripper
to pick up substrates
c9.set_output('substrate_tool', False)  #when False deactivate the vaccum bernoulli gripper to
releaze substrates

# aspirate the solution in the clamp
c9.aspirate_ml(0, 0.5)  # aspirate 0.5mL
c9.dispense_ml(0, 0.2)  # dispense 0.2mL

# Pick up a pipette
proc.new_pipette(c9)  # get a new pipette
proc.remove_pipette(c9)  # remove the pipette
```

Figure 4: Example of the operating functions for the N9 robot. The main instructions and commands are provided to the LLM agents as context to create the operating environment, accompanied by a brief description of each function as text.

A.2 Agents

A detailed description of the agents that comprise our current framework is described below. Each of the agents can be configured with an LLM.

Administrator agent: The administrator agent is designed to function as an administrative interface between users and the other agents. Its primary responsibilities include task delegation to other agents, receiving human input before a further action is taken and code execution in the specified working directory.

Literature Scraping Agent: The literature scraping agent is responsible for extracting key information from scientific literature and transferring this knowledge to other agents. It has access to an external tool that converts PDFs into textual format, enabling efficient data extraction.

System Message:

- ”You are a PDF scraper and can extract information from any provided PDF using the available tools.”
- ”After reading the text, you can provide specific answers based on the PDF’s context.”
- ”Return ’TERMINATE’ when the scraping process is complete.”

Code writer agent: This agent is responsible for providing the execution code as a python script. We provide the operating functions with a short description about the utility of each function in the system message. An example is shown in Figure 4.

Code reviewer agent: This agent is designed to evaluate the code created by the code writer and provide feedback.

System Message:

- ”Your task is to review the code provided by the code writer agent and provide feedback on necessary corrections.”
- ”Ensure that all required libraries are imported and that only the existing approved operation functions are used.”

Manager: This agent is creating a group with all the existing agents and is responsible for selecting the next relevant agent in the workflow.

A.3 TEACHABILITY

Autogen provides a teachability feature that can add memory capabilities to any existing agent. Once the teachability class is added to an agent, the agent’s system message is appended with a note about the this new ability: ”*You’ve been given the special ability to remember user teachings from prior conversations.*” and the agent is provided with extra functionalities and interaction with a text analysis agent. The background `TextAnalyzerAgent` (configured with the same LLM) is used to examine user comments and decide whether any important information should be stored. Internally, it uses the following prompting:

- Step 1: Ask, “Does any part of the text ask the agent to perform a task or solve a problem? Answer with just one word, yes or no.”
- If ‘yes’:

  “Briefly copy any advice from the text that may be useful for a similar but different task in the future. If no advice is present, respond with ‘none’.”*
- If the advice is not ‘none’:

  “Briefly copy just the task from the text, then stop. Don’t solve it, and don’t include any advice.”

  “Summarize very briefly, in general terms, the type of task described in the text. Leave out details that might not appear in a similar problem.”

By analyzing incoming user messages, this mechanism detects whether the text contains general information that should be remembered or a task/problem accompanied by useful advice. If such content is found, it is extracted and stored as input–output pairs in a ChromaDB vector database (Figures 6 and 5). These pairs are stored as text embeddings using the default Chroma Sentence

Transformers, all-MiniLM-L6-v2, to enable semantic similarity searches. When a new task is provided, the teachable agent performs a similarity search to find relevant past teachings, allowing it to apply previous learnings automatically. To achieve that, the system converts the new request into an embedding and searches the local ChromaDB for stored input-output pairs based on cosine similarity

By default, a teachable agent stores all new entries, even if they are very similar to existing ones. To avoid duplicates, we added a similarity check before storage: the system compares new memories against existing ones and only stores a new entry if it exceeds a certain distance threshold in the embedding space. Finally, because the teachability component does not require re-training the underlying model, it can be used in concert with larger or more expensive LLMs for creating memories, yet still function effectively with smaller or open-source models for day-to-day operations.

Human input for task 2:

You should first activate the vacuum of the coating stage to hold the substrate and then deactivate the vacuum of the gripper to release the substrate.

Related input-output memory that was created:

Memory ID: 8
Input:
The task involves using a vacuum mechanism to secure and release an object.
Output:
First activate the vacuum to hold the object, then deactivate the vacuum to release the object.

Memory ID: 9
Input:
What should I do first to hold the substrate in place during the coating stage?
Output:
Activate the vacuum of the coating stage to hold the substrate and then deactivate the vacuum of the gripper to release the substrate.

Figure 5: Example of memories created from GPT4o after providing human input for task2.

Example of memories created from Claude 3.5 Sonnet

Memory ID: 4
Input:
What is the Python code to move a substrate from the rack to the coating stage using a robot arm?
Output:
```python
# Step 1: Initialize the robot controller and set the tool
c9 = ro.system.init('controller')
c9.tool = 'substrate_tool'  # Pick up the Bernoulli substrate gripper tool

# Step 2: Move to the substrate rack and pick up a substrate
# We'll use the first available 'new' substrate
substrate_index = proc.find_rack_index('substrate', 'new')
c9.position = loca.substrate_rack[substrate_index]

# Activate the vacuum to pick up the substrate
c9.set_output('substrate_tool', True)

# Step 3: Move the substrate to the coating stage
c9.move_axis('z', 0)  # Move the arm all the way up for safe travel
c9.position = loca.s_coater  # Move to the coating station position

# Step 4: Place the substrate on the coating stage
c9.set_output('coater_stage_vacuum', True)  # Activate the coating stage vacuum
c9.set_output('substrate_tool', False)  # Release the substrate

# Step 5: Move the arm away from the substrate
c9.move_axis('z', 0)  # Move the arm all the way up

# Step 6: Return the substrate tool
c9.tool = None  # Drop off the Bernoulli substrate gripper tool

# Print a message to confirm the operation is complete
print("Substrate has been successfully moved to the coating stage.")
```
...

Memory ID: 5
Input:
What is the alias used for the imported procedure module?
Output:
from robotics import procedure as proc

Figure 6: Example of code related memories created from Claude 3.5 Sonnet after providing human input for task2.

A.4 Generated prompts

```
Task:
"""Write the execution code to create a polymer film using only PEDOT:PSS defined as polymer A. Extract the best range of the film processing conditions from the paper PEDOT_PSS_manuscript.pdf."""

Most relevant retrieved memory:
INPUT-OUTPUT PAIR RETRIEVED FROM VECTOR DATABASE:
  INPUT1
    What do I do first, move the substrate to the coating station or pick up the vial?
  OUTPUT
    you always fist move the substate to the coating station and then pick up the vial
  DISTANCE
    1.6069608286244856

Communication between the code writer agent an d the manager:
code_writer_agent (to chat_manager):
Based on the extracted information from the paper, the best range of film processing conditions for creating a polymer film using PEDOT:PSS (polymer A) includes:
1. **Additive Concentration**: Use ethylene glycol (EG) as an additive at a concentration of 5 vol%.
2. **Coating Speed**: A relatively low coating speed of 1 mm/s.
3. **Coating Temperature**: A relatively high coating temperature of 90 C .
4. **Post-Processing**: The post-treatment step involves solvent rinsing, repeated twice more, to remove PSS content and improve PEDOT packing structure.
Now, let's write the execution code to create a polymer film using these conditions. The code will follow the steps mentioned in your memories and the extracted information.
```python
# filename: create_polymer_film.py
import loca  # location information
import pandas as pd
import robotics as ro
from robotics import procedure as proc
import rack_status
# Initialize hardware modules
c9 = ro.system.init('controller')  # N9 robot controller
t8 = ro.system.init('temperature')  # temperature controller
coater = ro.system.init('coater')  # coating station
# Set the coating temperature
T = 90  # Coating temperature in Celsius
t8.set_temp(1, T)
# Move substrate to the coating station
c9.tool = 'substrate_tool'
c9.set_output('substrate_tool', True)  # Activate vacuum to pick up substrate
c9.position = loca.substrate_rack_seq[0, 0]  # Pick up substrate from rack
c9.position = loca.s_coater  # Move substrate to the coating station
c9.set_output('coater_stage_vacuum', True)  # Activate vacuum on coater stage
c9.set_output('substrate_tool', False)  # Release substrate
c9.tool = None  # Drop off the tool
# Pick up the vial containing polymer A (PEDOT:PSS)
sol_label = 'polymer_A'
vial_index = proc.find_rack_index('vial', sol_label)
c9.position = loca.vial_rack[vial_index]  # Move robot arm to the solution
c9.set_output('gripper', True)  # Close gripper to pick up the vial
```
```

```
c9.position = loca.clamp  # Move vial to the clamp
c9.set_output('clamp', True)  # Close clamp to hold the vial
c9.set_output('gripper', False)  # Open gripper to release the vial
# Uncap the vial
uncap_position = c9.uncap(pitch=1.75, revs=3.0, vel=5000, accel=5000)  #
    Uncap the vial
c9.position = uncap_position  # Move gripper back to the recorded
    position

# Pick up a pipette
proc.new_pipette(c9)
# Aspirate the solution in the clamp
c9.position = loca.clamp  # move robot arm to the clamp
c9.aspirate_ml(0, 0.5)  # Aspirate 0.5mL

# Move pipette to the coating station
c9.position = loca.p_coater  # Move pipette to the coating station
c9.dispense_ml(0, 0.2)  # Dispense 0.2mL

# Remove pipette
proc.remove_pipette(c9)

# Cap the vial
c9.position = uncap_position
c9.cap(pitch=1.75, revs=3.0, torque_thresh=1000, vel=5000, accel=5000)

# Return vial to rack
c9.set_output('clamp', False)
c9.set_output('gripper', True)
c9.position = loca.vial_rack[1, 2]
c9.set_output('gripper', False)

# Coating process
coater.position = 45  # Move coater blade to the starting position
coater.velocity = 1  # Set the coating velocity to 1 mm/s
coater.position = 75  # Move blade all the way to the right
print("Polymer film creation process completed.")
```
This code will execute the process of creating a polymer film using PEDOT
    :PSS with the specified conditions. Please save the code in a file
    named `create_polymer_film.py` and execute it. TERMINATE
```

Listing 1: Example of the communication between the teachable agents to create a polymer film.

### A.5 PERFORMANCE EVALUATION METRICS

For the evaluation of the agents performance we Focused on five tasks:

1. Code quality check: Checking if the code reviewer was initiated to check the initial code written by the code writer. That is basicallt focusing checking if the correct functions have been imported and that the code is operational.

2. Correctness: Checking if the final code is correct and calls the correct functions sequentially. Evaluating the functionality of the generated code and checking whether the sequential steps are performed in the correct order. The score is counting how many steps of the required ones have been identified from the agents. The best score is 8, 10, 25 for the first, second and third task accordingly.

3. Code execution: How many of the steps can the code run without doing something wrong. This differs from the code correctness, as all the steps might have been identified correct but placed in a wrong sequence and as such the operation of the robot will break. The score is counting how many steps of the required ones have been identified from the agents. The best score is 8, 10, 25 for the first, second and third task accordingly.
```

4. Code repeatability: Evaluating whether the same code is generating after running the same prompt task for two times. The temperature of the models is set to 0 in all cases.

5. Reproducibility: Assess the consistency of the generated code when provided with altered prompts for the same task. This is particularly crucial in user facilities, where different users may describe identical tasks using various expressions. Below are the original prompts tested along with their alternative versions:

- prompt1 = "Write the execution code to move the vial with PEDOT:PSS defined as polymer A to the clamp holder."

- prompt1a = "Write the code to move the vial with polymer A to the clamp."

- prompt2 "Write the execution code to pick up a substrate from the substrate rack and move it to the coating station."

- prompt2a = "Write the code to pick up a substrate and move it to the coating stage."

- prompt3 = "Write the execution code to create a polymer film using only PEDOT:PSS defined as polymer A. Extract the best range of the film processing conditions from the paper `PEDOT_PSS_manuscript.pdf`"

- prompt3a = " Write the code to create a polymer film with only PEDOT:PSS defined as polymer A. Identify the best processing conditions from the paper `PEDOT_PSS_manuscript.pdf`"

An example of how the evaluation was performed is shown below:

The eight (8) sequential tasks are organized as follows:

1. Import the correct libraries
2. Initialize the required system components
3. Identify the location of the vial
4. Move gripper to the vial position
5. Close the gripper to pick up the vial
6. Move the gripper with the vial to the clamp holder
7. Close the clamp holder to hold the vial
8. Release the gripper
9. We give one point for each correct step

One point for each correct step.

```
# filename: move_vial_to_clamp.py
import loca
import pandas as pd
import robotics as ro
from robotics import procedure as proc
import time  # Import time for delays

def move_vial_to_clamp(polymer_label='polymer_A'):
    # Initialize the robot controller
    c9 = ro.system.init('controller')

    # Step 1: Find the index of the vial containing polymer A
    vial_index = proc.find_rack_index('vial', polymer_label)

    # Check if the vial index is valid
    if vial_index is None:
        print(f"Error: Vial with label '{polymer_label}' not found.")
        return

```

```
    # Step 2: Move to the vial position
    c9.position = loca.vial_rack[vial_index]

    # Step 3: Close the gripper to pick up the vial
    c9.set_output('gripper', True)
    time.sleep(1)  # Wait for the gripper to close

    # Step 4: Move to the clamp holder position
    c9.position = loca.clamp

    # Step 5: Open the gripper to release the vial into the clamp holder
    c9.set_output('gripper', False)
    time.sleep(1)  # Wait for the gripper to open


# Execute the function
move_vial_to_clamp()
```

Listing 2: Example of the code produced from the GPT-4o-mini pipeline for executing Task 1.

Evaluation metric	Score	Task explanation	Evaluators comments
Code quality check	V	Check if the code reviewer agents was called to check the code for code quality analysis for potential errors, stylistic inconsistencies and best-practice violation.	Code reviewer was called and provided suggestions for error handling
Code correctness	87.5%	Evaluating the functionality of the generated code and checking whether the sequential steps are performed in the correct order.	7/8 of the steps were identified. It didn't close the clamp to hold the gripper
Code execution (correct order of functions)	75%	How many of the steps can the code run without doing something wrong	Will stop in step 6; the rest is not correct and will break the process (6/8)
Code repeatability	100%	Evaluating whether the same code is generating after running the same prompt task for 3 times.	The same prompt was run three times and produced the same output
Reproducibility (varying prompts)	100%	Evaluating whether the generated code is the same after modifying the task prompt whilst keeping the main task the same	Used a similar prompt to the initial one and performed the same.

Table 4: Example of human evaluation metrics for task1 using GPT-4o-mini