Task Alignment Outweighs Framework Choice in Scientific LLM Agents

Nawaf Alampara [1,*] **Martiño Ríos-García** [1,*] **Chandan Gupta** [2,*] **Sajid Mannan** [2]

Santiago Miret [3] **N. M. Anoop Krishnan** [2,4,5]

Kevin Maik Jablonka[1,6,7]
[1]FSU Jena* [2] IIT Delhi †
[3] Intel Labs
[4] IIT Delhi ‡
[5] IIT Delhi §
[6]CEEC Jena¶ [7] HIPOLE Jena ‖
* These authors contributed equally to this work.
krishnan@iitd.ac.in mail@kjablonka.com

Abstract

Large language models (LLMs) equipped with external tools through agentic frameworks promise to overcome domain-specific limitations by providing specialized capabilities for scientific applications. However, the extent to which these systems genuinely enhance performance in complex scientific domains remains poorly understood. Here we present *Corral*, a modular benchmarking framework that systematically evaluates LLM-based agents across four expert-designed environments spanning molecular dynamics, machine learning, catalysis, and spectroscopy in chemistry and materials science. Through comprehensive evaluation of state-of-the-art models using different agent scaffolds, we demonstrate that the choice of agentic framework—whether ReAct or tool-calling—plays a surprisingly minor role in determining success. Instead, performance depends critically on the semantic alignment between available tools and task requirements, measured through embedding similarity. When this alignment is poor, even sophisticated reasoning frameworks cannot compensate for inadequate tool provisioning and a lack of domain knowledge. Conversely, when base models possess sufficient domain knowledge, agentic frameworks can introduce unnecessary overhead without meaningful benefits. Our findings challenge the assumption that agentic systems provide a universal solution to model limitations, revealing instead that, currently, successful scientific agents

*Laboratory of Organic and Macromolecular Chemistry (IOMC), Friedrich Schiller University Jena, Humboldstr. 10, 07743 Jena, Germany

†School of Interdisciplinary Research, Indian Institute of Technology Delhi, Hauz Khas, New Delhi 110016, India

‡Department of Civil Engineering, Indian Institute of Technology Delhi, Hauz Khas, New Delhi 110016, India

§Yardi School of Artificial Intelligence, Indian Institute of Technology Delhi, Hauz Khas, New Delhi 110016, India

¶Center for Energy and Environmental Chemistry Jena, Friedrich Schiller University Jena, Philosophenweg 7, 07743 Jena, Germany

‖Helmholtz Institute for Polymers in Energy Applications Jena (HIPOLE Jena), Lessingstraße 12-14, 07743, Jena, Germany

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

might require the same level of domain expertise in tool design that was promised to be circumvented.

1 Introduction

Large language models (LLMs) excel at generating text but struggle when tasks require interacting with external systems, accessing real-time information, or executing multi-step procedures [1, 2]. Agent frameworks have been proposed to address these limitations by enabling models to plan sequences of actions, use specialized tools, and operate within specific environments [3, 4, 5, 6, 7, 8, 9, 10]. This approach might be particularly critical in scientific applications, where precise domain knowledge is essential for accurate decision-making [11, 12], in which base models show limitations [13, 14]. However, the extent to which these tools genuinely enhance model performance remains unclear. While initiatives like the Model Context Protocol (MCP) [15] have improved tool accessibility, the mere provision of additional tools can be counterproductive; tools may act as distractors rather than valuable assets, leading to inefficient or erroneous outputs [16, 17].

Current evaluation methodologies predominantly focus on high-level metrics such as overall performance, tool usage count, and execution time, with an emphasis on coding and machine learning tasks [18]. Although essential for assessing production-ready systems, these metrics often overlook some of the critical components of the systems and do not provide practical guidance on how to improve them [19]. By focusing solely on the end result, without considering the process of tool utilization, these evaluations limit our understanding of the models' true capabilities and can even result in agents failing to fully comprehend the functions available to them [20]. This lack of granularity is particularly problematic in scientific fields like chemistry or biology, where agents utilize tools not only for retrieval, but also for experiment execution, and knowledge synthesis [21].

Moreover, the importance of task-tool alignment is not properly understood. It is often unclear whether the provided tools are truly relevant or if the agent understands when and how to use them. Here, we demonstrate that the alignment between tasks and tools is indicative of agentic performance. To do so, we built a novel modular benchmarking framework, systematically implemented multiple scientific environments and agents, and conducted ablation studies.

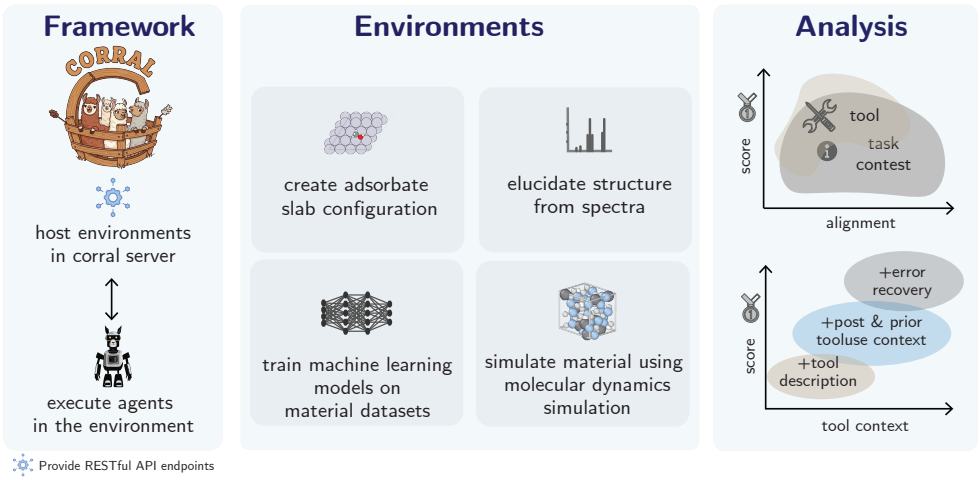

Figure 1: **Representation of the core contribution in this work**. We present the *Corral* framework, which hosts environments on a server separating them from the agents, enabling an evaluation of how the different components affect the results. This analysis of the different components allows us to make actionable recommendations on how to improve LLM-based agents.

As illustrated in Figure 1, our main contributions are:

- We present *Corral*, a modular framework to implement, run, and test LLM-based agents. Within *Corral*, four expert-designed environments are used to assess the performance of current models.
- We benchmark two SOTA LLMs and two of the most widely implemented agent scaffolds, providing a comprehensive evaluation of current agent capabilities.
- We perform an analysis of the factors that determine the success of an LLM-based agent. In particular, we propose the use of embedding similarity between tools and tasks to estimate agents' performance. The alignment between tools and tasks is especially important when the agent lacks the foundational knowledge required to carry out the task.

2 Related Work

LLM-based Agents LLM-based agents demonstrated proficiency in coding tasks [22]. This has extended to specialized domains including ML development [23], algorithm optimization [24], and AI research automation [25]. *CodeAct Agent* is a multi-agent system that enables agents to either converse with humans in natural language for clarification and confirmation or perform tasks by executing code, such as running bash commands, Python scripts, or browser-specific programming languages [26]. Similarly, *AIDE* is an ML-focused agent that autonomously drafts, debugs, and optimizes code through iterative tree-search strategies based on user-defined goals and metrics [7].

In science, agents have targeted distinct phases of the research pipeline, including data retrieval [27, 28], simulation execution [29, 30, 31, 32], and data analysis [33]. Some notable examples are those that can run certain lab experiments [34, 35, 36].

Scientific Agent Benchmarks The advancements in agents are facilitated by high-quality benchmarking frameworks that rigorously quantify agent capabilities in coding [37, 38, 39, 40], chemistry [41], and materials science [42, 43]. Tian et al. [44] demonstrated limitations of agents replicating real-world scientific scripts. Even top-performing models achieved below 10 % accuracy in generating specialized scientific code. This performance gap extends to other scientific contexts, as shown by Cui et al. [45]'s multi-domain benchmark featuring complex tasks like DFT calculations and materials property extraction. Complementary approaches like *Aviary* [46] or *AutoGen* [47] address system design by isolating environments from agents. These "agent gym" frameworks enable tool decoupling and targeted optimization, yielding performance gains.

Existing benchmarks target specific scientific tasks given diverse evaluation needs [48, 19]. There have also been attempts to measure whether the provided tools are truly relevant or if the agent understands when and how to use them. One proposed way to assess this is by checking if expected tools are used at each step [49], but this approach is inadequate for environments requiring open-ended planning and strategic reasoning [48, 39]. In such cases, the agent's ability to decompose a task and strategically select tools becomes paramount, and currently, there is no way to measure this ability.

Overall, there is no systematic understanding that can be used as practical guidance for improving scientific agents. By systematically ablating the components within the agentic framework for a set of scientific environments, our work identifies critical performance limitations and provides actionable pathways for agent improvement.

3 *Corral* : Agent Evaluation Framework

The *Corral* framework, represented in Figure 2, is designed to define, execute, and evaluate agent capabilities within different isolated environments (code snippets demonstrating how to implement the different components in the framework are shown in Appendix D). Inspired by *Aviary* [46] and prior works [50, 51, 52], we adopt a model similar to a Language Decision Process (LDP), which is a specific formulation of a Partially Observable Markov Decision Process (POMDP) where observations and actions are represented in text.

3.1 The Environment

The Environment ($\mathcal{E}$) is responsible for serving the task guide to the agent (which includes the task description and input data from previous tasks, if any), providing the necessary tools, and evaluating the agent's performance. Formally, the environment implements the core components of the LDP.

State Space ($\mathcal{S}$). The state s in $\mathcal{S}$ represents a complete snapshot of the environment at a given time step. In *Corral*, this is captured by the `TaskState` object. A state s_t at time t is a tuple:

$$s_t = (P, M_t, C_t, d, \sigma) \in \mathcal{S} \tag{1}$$

where:

- P is the static task prompt or problem description.
- $M_t = \{m_1, m_2, ..., m_t\}$ is the set of messages exchanged between the agent and the environment.
- $C_t = \{c_1, c_2, ..., c_k\}$ is the history of k tool calls made up to time t. Each c_i is a `ToolCall` object containing the tool name, arguments, result, and status.
- d is a boolean flag indicating if the task is completed (`is_completed`).
- σ is the final score, which is defined only when d is true.

During benchmarking, d and σ are not observable to the agent. However, this could be used to optimize the system interacting with the environment during training.

Tools. A key responsibility of the environment is provisioning a set of tools $\mathcal{T}_{env}$. Each tool $\tau \in \mathcal{T}_{env}$ is a function that maps a set of arguments to a result returned as a string.

Task ($\mathcal{K}$). A task $\mathcal{K}$ defines a specific, self-contained problem. It is formally a tuple:

$$\mathcal{K} = (P, \mathcal{T}_{req}, f_{\text{score}}, I_0) \tag{2}$$

where:

- P is the task prompt given to the agent.
- $\mathcal{T}_{req} \subseteq \mathcal{T}_{env}$ is the specific subset of tools required for this task.
- f_{score} is the custom scoring function used to implement the reward function R.
- I_0 is the initial input data for the task.

Task Group ($\mathcal{G}$). A task group allows for the creation of complex workflows by defining dependencies between tasks. Formally, a task group is a directed graph:

$$\mathcal{G} = (V, E) \tag{3}$$

where $V = \{\mathcal{K}_i\}_{i=1}^{N}$ is a set of N tasks serving as the vertices of the graph. The set $E \subseteq V \times V$ contains the directed edges, where an edge $(\mathcal{K}_i, \mathcal{K}_j) \in E$ implies that the output of task $\mathcal{K}_i$ is a required input for task $\mathcal{K}_j$. The directed nature of the graph guarantees a well-defined execution order.

3.2 The Agent

The Agent is the decision-making entity that interacts with the environment to solve a task. It is implemented as a policy π that maps the history of observations to a probability distribution over the available actions.

Action Space ($\mathcal{A}$). The action space $\mathcal{A}$ consists of all possible actions the agent can take. This space is the union of all possible tool calls and a special submission action, `submit`.

$$\mathcal{A} = \left(\bigcup_{\tau_i \in \mathcal{T}_{env}} \texttt{call}(\tau_i, \text{args}_i) \right) \cup \texttt{submit}(\text{answer}) \tag{4}$$

where args_i represents a set of arguments for tool τ_i. An action $a_t \in \mathcal{A}$ is thus either a tool invocation or the submission of a final answer.

Policy (π). The policy π selects an action a_t at time t based on the history of observations $h_t = (o_1, o_2, \ldots, o_t)$.

$$a_t \sim \pi(\cdot|h_t) \tag{5}$$

The $\cdot$ stands for any possible action a from the entire action space $\mathcal{A}$. The *Corral* framework is agnostic to the internal reasoning of the agent, allowing for different implementations of π, such as ReAct [53], planning algorithms [54], or simple tool-calling models.

3.3 Interaction Dynamics

The *Corral* runner orchestrates the agent-environment interaction over a series of tasks (Figure 2 illustrates the *Corral* framework). Each task is part of a separate and isolated environment on the *Corral* server. Within its corresponding isolated environment, the agent's interaction unfolds over a sequence of discrete time steps, concluding only when it executes the `submit` action. Upon this terminal action, the *Corral* runner finalizes the interaction with the current environment and proceeds to the next task.

Transition Function (T). The state transition function $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ describes how the state evolves.

$$s_{t+1} = T(s_t, a_t) \tag{6}$$

If $a_t = \texttt{call}(\tau, \text{args})$, the environment executes the tool and updates the state by appending a new `ToolCall` record to the history C_t, resulting in C_{t+1}. If $a_t = \texttt{submit}(\text{answer})$, the environment updates the state by setting d to true and computing a score.

Observation Space ($\mathcal{O}$) and Function (Z). After the environment transitions to a new state s_{t+1}, the agent receives an observation $o_{t+1} \in \mathcal{O}$. The observation function $Z : \mathcal{S} \to \mathcal{O}$ could be stochastic depending up on the tool.

$$o_{t+1} = Z(s_{t+1}) \tag{7}$$

The observation is typically the string-based result or error message from the most recent tool call, which corresponds to the `result` or `error_message` field in the last element of the tool call history C_{t+1}.

Reward Function (R). The reward function $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ provides the feedback signal to the agent. In *Corral*, during benchmarking, rewards are sparse and terminal. A reward is only issued when the agent takes the `submit` action.

$$R(s_t, a_t) = \begin{cases} f_{\text{score}}(\text{answer}) & \text{if } a_t = \texttt{submit}(\text{answer}) \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

The reward is the agent's score in that particular task. In our experiments, we stick to binary rewards (either 0 or 1).

4 Results

The environments for our evaluations, detailed in Appendix E, are deliberately designed to represent a spectrum of complexity and require specialized knowledge. The `ML` and `OpenCatalyst` tasks are more aligned with standard computational workflows, while molecular dynamics (`MD`) requires nuanced scientific knowledge, and especially `Spectra` requires reasoning over and validating the hypotheses it generates based on chemical observations.

To disentangle the effect of the framework in the policy, we compare two agent strategies, ReAct [53] and tool-calling. ReAct promotes an iterative *thought-action-observation* loop. The tool-calling agent is a more direct, function-execution approach, which might be more efficient for well-defined workflows. We probe the capability of the core reasoning engine of the policy (π) by testing two different foundation models.

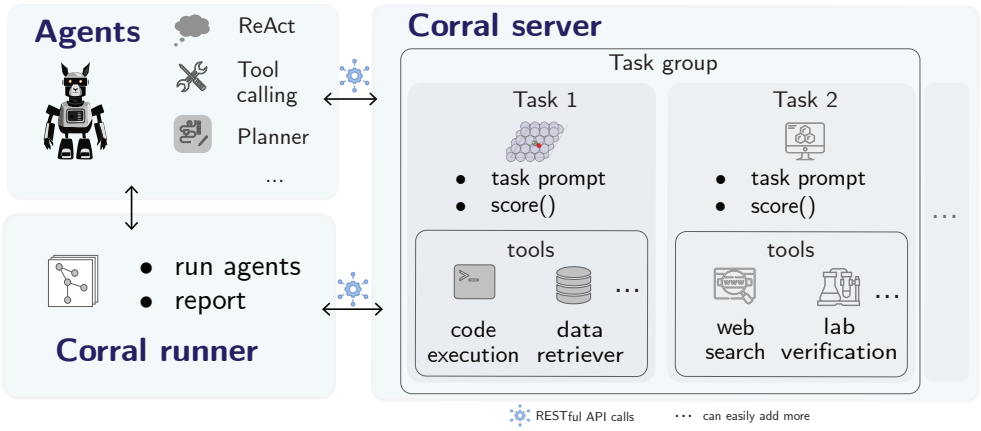

Figure 2: **Illustration of the *Corral* framework.** The architecture is composed of the *Corral* server (implementing the environment dynamics), agents (the policies π), and the *Corral* runner (orchestrating the agent-environment interaction over a series of tasks).

4.1 Overall Performance

Figure 3 shows the agents' ability to successfully complete the task within five attempts (pass@5, metrics used in the work are detailed in Appendix A). The illustration provides a high-level view of their capabilities across the four scientific environments, which are ranked by the domain expertise required to solve the tasks. The SPECTRA environment requires most domain-specific skills, the ML environment the least. The results underscore the significant impact of domain complexity on the policy's (π) ability to achieve a high reward, R. Interestingly, these results follow an inverse trend compared to the number of tool calls, as reported in Appendix G, suggesting that agents tend to sample more tools when unsure of how to solve the tasks.

Performance and Domain Familiarity Comparison In the OPENCATALYST and ML environments, both `Claude-3.5 Sonnet` and `GPT-4o` achieve near-perfect scores. This suggests that for these tasks, the policies (π) can effectively map the initial task prompt P to the correct sequence of tool calls. Conversely, environments demanding deep, specialized knowledge as MD and SPECTRA, result in lower scores. The particularly low scores in the SPECTRA environment highlight the difficulty the policy faces in interpreting chemical data to select the optimal sequence of actions from $\mathcal{A}$.

Agent and Model Comparison In the more demanding MD and SPECTRA environments, the differences between agents are very small. This is surprising since the tool-calling agent is a much simpler implementation that does not enforce agents' reasoning compared with ReAct. However, in other environments where we suspect that the policies are trained on the reasoning chain in ReAct, it turns out to be helpful. `Claude-3.5 Sonnet` consistently shows a slight performance advantage over `GPT-4o`, especially in these challenging domains. This might suggest that `Claude-3.5 Sonnet` is more familiar with chemistry and material science compared to `GPT-4o`, consistent with prior research [14, 13].

4.2 Decomposing Tasks

To validate our hypothesis that the policy, π, is less likely to converge on a successful action trajectory for unfamiliar tasks, we evaluate the tasks in two different formats. As *single* tasks ($\mathcal{K}$), and *chained* tasks, where the *single* tasks are broken down to smaller single-step problems, forming a task group $\mathcal{G}$. In a chained configuration, dependencies are defined and the output of task $\mathcal{K}_i$ serves as the initial input, I_0, for task $\mathcal{K}_j$ (see example in Appendix E.4.3). By breaking down the tasks into smaller tasks, the action space (or the action trajectory) over which the policies have to reason is constrained.

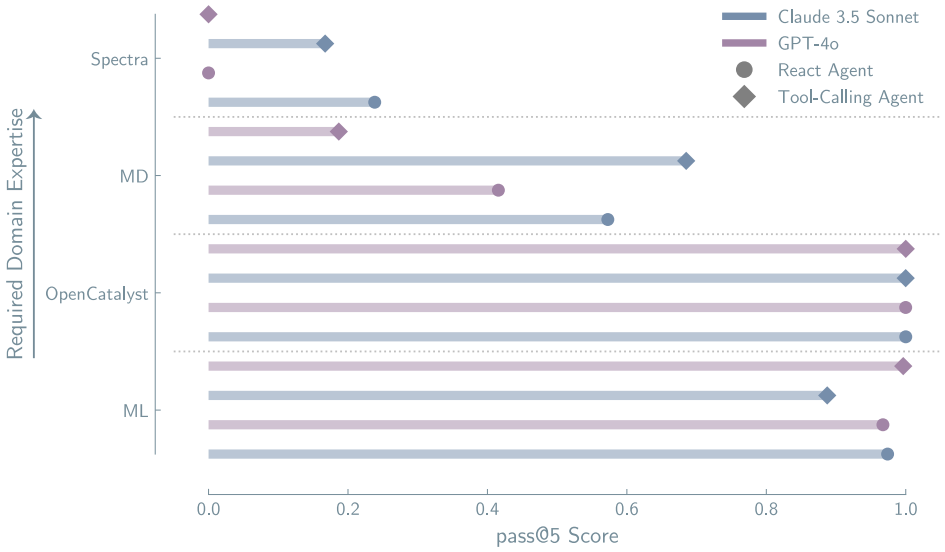

Figure 3: **Impact of domain expertise on agent performance across environments**. This plot shows the pass@5 scores for each agent across the four environments, ordered by the level of required domain expertise (from low at the bottom to high at the top). There is a clear inverse correlation: as the need for specialized knowledge increases, the agents' ability to successfully complete the task within five attempts decreases. This visualizes the significant challenge that deep domain expertise poses to the agents' policies.

Improved Performance through Decomposition in Complex Domains As shown in Figure 4, for the `MD` and `SPECTRA` environments, all agent and model configurations show a significant improvement in performance when moving from a *single* task to a *chained* one. For example, in the `SPECTRA` environment, the `Claude-3.5 Sonnet` tool-calling agent's score jumps from below 0.25 on the single task to nearly 0.50 on the chained task. This strongly suggests that decomposing a complex problem into a structured sequence of smaller tasks provides a crucial scaffold that helps agents succeed if many domain skills are needed.

Performance Breakdown by Task Categories To further ground our understanding that lower performance could be attributed to the inability of the model to reason and validate scientific hypotheses and tasks, we classified all the decomposed tasks into a set of categories ("retrieval", "code execution", "experiment execution", "reasoning", and "validation"). For example, fetching a structure from *Materials Project* (MP)[55] in the `OPENCATALYST` environment is classified as "retrieval", while running a Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) simulation is classified as "experiment execution". This classification was done manually by the experts who designed the environments.

Figure 5 plots the average pass@5 score for each of the categories. This breakdown reveals that the primary challenge for current agents lies in higher-order reasoning, validation of the generated history, and execution rather than simple tool use. Again, as for the single task results, we observe that both agents, ReAct and tool-calling, perform very similarly despite the difference in complexity.

4.3 Tool Verbosity Ablation

It is well known that the amount of context provided to an LLM is important for the performance [56, 57]. In an attempt to systematically understand the importance of this for the agent performance, we encoded the tool descriptions in a structured format (described in Appendix C) and ablated the effect of only showing parts of this to the model. We evaluated agent performance across three progressively detailed verbosity levels: `BRIEF`, `WORKFLOW`, and `COMPREHENSIVE`. This tiered approach allows us to measure how performance scales as the agent is supplied with increasingly rich contextual and operational knowledge about its tools.

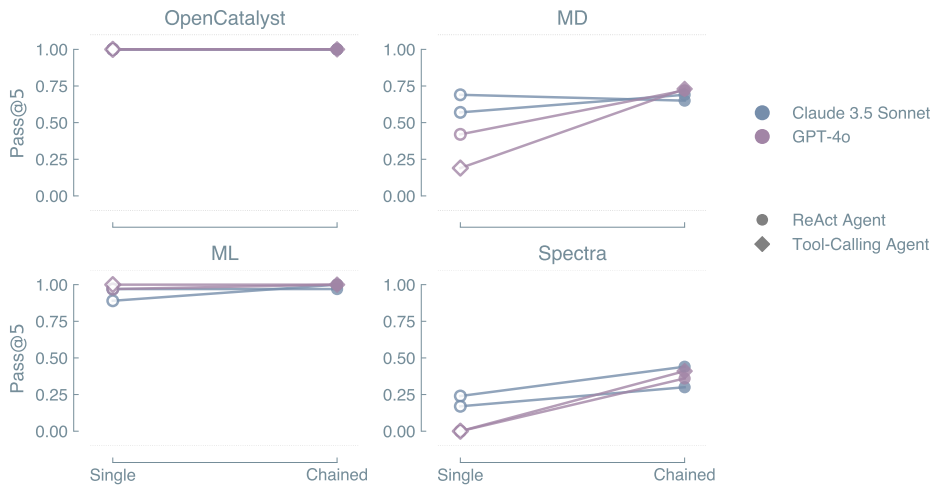

Figure 4: **Performance comparison between *single* and *chained* tasks**. This plot shows the pass@5 scores when the task is decomposed into simple one-step tasks, and when it is not. All tools are in COMPREHENSIVE verbosity (refer to Appendix C for a detailed explanation about the different levels of tool descriptions considered). In the more challenging MD and SPECTRA environments, all agent configurations show a significant increase in overall score when the task is presented as a chained sequence of sub-tasks. Table comparing results for all different tool verbosity in Table I.11

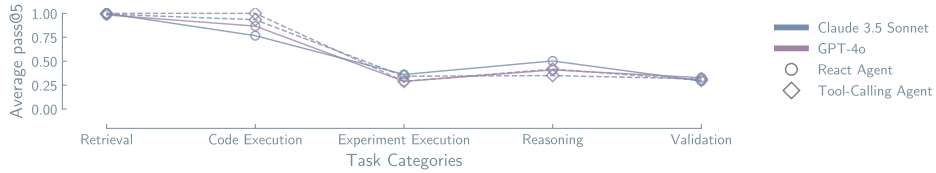

Figure 5: **Performance breakdown by task categories**. This figure illustrates the average pass@5 scores of different agent configurations across five task categories. All tasks were manually classified into different categories by the experts who designed the environments. While agents excel at concrete tasks like "retrieval" and "code execution", performance declines for more complex categories such as "experiment execution", "reasoning", and "validation". The average score here is the mean of pass@5 scores for each of the chained tasks.

Level of Description Is Not All You Need As detailed in Table I.11, for OPENCATALYST and ML environments, performance is consistently high for the three levels of tool description that were considered. For the most complex environments, SPECTRA and MD, the scores, despite being worse than for the other environments, do not show a clear trend in which verbosity is better. For example, in the SPECTRA environment, it is possible to see that the best score overall is achieved in the BRIEF description scenario by the tool-calling agent in the chained fashion. However, that same agent has a very similar performance when the tool descriptions follow the WORKFLOW level. Similarly, when evaluating the full or single task, the best score is obtained with the COMPREHENSIVE level of description.

4.4 Dependence on Task and Tool Similarity

In our analysis, we observe that the SPECTRA environment performs much worse than the rest of the environments, independent of the level of detail in the tool description. We hypothesize that this is because the tools provided in this environment are less aligned with the most difficult step in the tasks compared to other environments. To probe this hypothesis, we embedded the tools and task descriptions and measured their distances. A detailed description of how the embeddings are obtained and the distances calculated is provided in Appendix J. Table 1 shows that SPECTRA shows the largest mean minimum distances between tasks and tools, followed by the MD, ML, and

OpenCatalyst environments. This ordering correlates with the overall performance of the agents in these environments. Additionally, Spectra presents the second-highest mean distance between tasks, only after ML. We hypothesize that the larger distances between consecutive tasks in ML explain the scores observed for Pass$^\wedge k$, detailed in Appendix F.

Table 1: **Embedding distance analysis**. Mean cosine distances between consecutive tasks in a chain, between all tasks and tools, and between the minimum or closest tool to each task. The results show that Spectra is among the highest values for the measured distances, which could explain the bad results in this environment.

Task	Tools	Mean of consecutive tasks	Verbosity	Distance Tools-Tasks	
				Mean	Mean of Minimum
Spectra	15	0.52	brief	0.70	0.52
			workflow	0.75	0.59
			comprehensive	0.84	0.73
MD	4	0.26	brief	0.64	0.58
			workflow	0.65	0.60
			comprehensive	0.72	0.66
OpenCatalyst	6	0.38	brief	0.61	0.38
			workflow	0.60	0.42
			comprehensive	0.71	0.57
ML	12	0.65	brief	0.68	0.39
			workflow	0.65	0.40
			comprehensive	0.72	0.52

This correlation can be used to strategically improve agentic systems. To test this, we augmented the Spectra environment with a new tool that works as a hypothesis generator by proposing molecular fragments for the candidate molecule. While this value might seem insignificant, as Figure 6 shows, we find that with this tool `Claude-3.5 Sonnet` now performs comparably in both single and chained settings. `GPT-4o`, which is less proficient in spectroscopic tasks in other benchmarks [13], fails to leverage this additional tool.

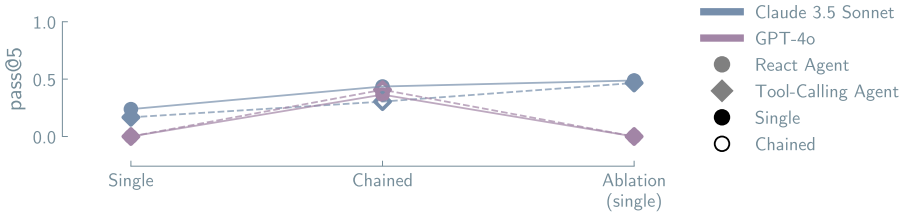

Figure 6: **Results comparing single, chained, and the ablation for the `Spectra` environment**. The figure shows that by incorporating a new tool, the distance between tools and tasks is closed, resulting in a massive improvement in the scores of the single task being comparable to the ones in the chained task.

5 Conclusion

Large language models have demonstrated remarkable capabilities across diverse tasks, yet significant performance gaps persist in specialized domains like chemistry and materials science. The field has responded with increasing enthusiasm for agentic systems—frameworks that equip LLMs with external tools and reasoning scaffolds. This approach emerged from a compelling belief: that agents could serve as a universal bridge, compensating for any base model's limitations simply by providing domain-specific tools. The promise was elegant in its simplicity—rather than training

specialized models for each domain, we could build flexible agents that adapt to any task through careful framework engineering and tool provisioning.

Our systematic evaluation using our *Corral* framework reveals this foundational assumption to be deeply flawed. Through comprehensive benchmarking across four expert-designed environments spanning molecular dynamics, machine learning, catalysis, and spectroscopy, we demonstrate that the choice of agentic framework—whether ReAct, or tool-calling—plays a surprisingly minor role in determining success. Instead, performance depends critically on the alignment between available tools and task requirements, coupled with the base model's existing domain knowledge. When models already possess adequate domain expertise, agentic frameworks introduce unnecessary overhead without meaningful benefits. Conversely, when domain knowledge is absent, only meticulously crafted tools that precisely match task demands can bridge the gap—a requirement that eliminates the promised flexibility and demands exactly the kind of deep domain expertise that agents were meant to circumvent.

These findings fundamentally challenge the notion that agentic systems provide a universal solution to model limitations. Rather than offering a flexible path around the need for specialized knowledge, agents reveal themselves to be highly dependent on the very expertise they promised to democratize. Our work highlights an enduring truth: the absence of domain-specific knowledge in foundation models cannot be overcome through framework sophistication alone, but only through targeted tool design that requires the same level of domain expertise as training specialized models—suggesting that domain-specific models are far less obsolete than one might think.

6 Acknowledgments

This work was supported by the Carl Zeiss Foundation, OpenPhilanthropy, and a "Talent Fund" of the "Life" profile line of the Friedrich Schiller University Jena.

Part of the work of M.R.G. was supported by Intel and Merck via the AWASES Center.

K.M.J. is part of the NFDI consortium FAIRmat funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project 460197019.

K.M.J. and N.M.A.K. acknowledge the Google Research Scholar Award.

N.M.A.K. acknowledges the Alexander von Humboldt Foundation for funding support, and the HPC IIT Delhi for computational and storage resources.

We thank Carmelo Gonzales, Kelvin Lee, and Rafael Rosales for discussions. Moreover, we thank Gordan Prastalo and Mara Schilling-Wilhelmi for feedback on an early version of the manuscript.

References

[1] Andrew D. White. "The future of chemistry is language". In: *Nature Reviews Chemistry* 7.7 (May 2023), pp. 457–458. ISSN: 2397-3358. DOI: 10.1038/s41570-023-00502-0. URL: http://dx.doi.org/10.1038/s41570-023-00502-0.

[2] Nawaf Alampara et al. "General purpose models for the chemical sciences". In: *arXiv preprint arXiv: 2507.07456* (2025).

[3] David Silver and Richard S Sutton. "Welcome to the era of experience". In: *Google AI* 1 (2025).

[4] John Yang et al. "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering". In: *arXiv preprint arXiv: 2405.15793* (2024).

[5] Bang Liu et al. "Advances and Challenges in Foundation Agents: From Brain-Inspired Intelligence to Evolutionary, Collaborative, and Safe Systems". In: *arXiv preprint arXiv: 2504.01990* (2025).

[6] Samuel Schmidgall et al. "Agent Laboratory: Using LLM Agents as Research Assistants". In: *arXiv preprint arXiv: 2501.04227* (2025).

[7] Zhengyao Jiang et al. "AIDE: AI-Driven Exploration in the Space of Code". In: *arXiv preprint arXiv: 2502.13138* (2025).

[8] Xingyao Wang et al. "OpenHands: An Open Platform for AI Software Developers as Generalist Agents". In: *arXiv preprint arXiv: 2407.16741* (2024).

[9] Theodore R. Sumers et al. "Cognitive Architectures for Language Agents". In: *arXiv preprint arXiv: 2309.02427* (2023).

[10] Noah Shinn et al. "Reflexion: Language Agents with Verbal Reinforcement Learning". In: *arXiv preprint arXiv: 2303.11366* (2023).

[11] Alán Aspuru-Guzik and Varinia Bernales. "The rise of agents: Computational chemistry is ready for (R)evolution". In: *Polyhedron* (July 2025), p. 117707. ISSN: 0277-5387. DOI: 10.1016/j.poly.2025.117707. URL: http://dx.doi.org/10.1016/j.poly.2025.117707.

[12] Santiago Miret and NM Anoop Krishnan. "Enabling large language models for real-world materials discovery". In: *Nature Machine Intelligence* (2025), pp. 1–8.

[13] Adrian Mirza et al. "A framework for evaluating the chemical knowledge and reasoning abilities of large language models against the expertise of chemists". In: *Nature Chemistry* (2025), pp. 1–8.

[14] Nawaf Alampara et al. "Probing the limitations of multimodal language models for chemistry and materials research". In: *Nature Computational Science* (2025), pp. 1–10.

[15] Anthropic. *Model Context Protocol*. Protocol Specification. Open protocol for connecting AI applications to external data sources and tools. Anthropic, PBC, Nov. 2024. URL: https://modelcontextprotocol.io.

[16] Varatheepan Paramanayakam et al. "Less is More: Optimizing Function Calling for LLM Execution on Edge Devices". In: *Design, Automation & Test in Europe Conference, DATE 2025, Lyon, France, March 31 - April 2, 2025*. IEEE, 2025, pp. 1–7. DOI: 10.23919/DATE64628.2025.10992798. URL: https://doi.org/10.23919/DATE64628.2025.10992798.

[17] Guozhao Mo et al. "LiveMCPBench: Can Agents Navigate an Ocean of MCP Tools?" In: *arXiv preprint arXiv: 2508.01780* (2025).

[18] Mahmoud Mohammadi et al. "Evaluation and Benchmarking of LLM Agents: A Survey". In: *arXiv preprint arXiv: 2507.21504* (2025).

[19] Nawaf Alampara, Mara Schilling-Wilhelmi, and Kevin Maik Jablonka. "Lessons from the trenches on evaluating machine-learning systems in materials science". In: *arXiv preprint arXiv: 2503.10837* (2025).

[20] Xuanqi Gao et al. "MCP-RADAR: A Multi-Dimensional Benchmark for Evaluating Tool Use Capabilities in Large Language Models". In: *arXiv preprint arXiv: 2505.16700* (2025).

[21] Mayk Caldas Ramos, Christopher J. Collison, and Andrew D. White. "A review of large language models and autonomous agents in chemistry". In: *Chemical Science* 16.6 (2025), pp. 2514–2572. ISSN: 2041-6539. DOI: 10.1039/d4sc03921a. URL: http://dx.doi.org/10.1039/D4SC03921A.

[22] Carlos E. Jimenez et al. "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" In: *arXiv preprint arXiv: 2310.06770* (2023).

[23] Jun Shern Chan et al. "MLE-bench: Evaluating Machine Learning Agents on Machine Learning Engineering". In: *International Conference on Learning Representations* (2025).

[24] Alexander Novikov et al. "AlphaEvolve: A coding agent for scientific and algorithmic discovery". In: *arXiv preprint arXiv: 2506.13131* (2025).

[25] Juraj Gottweis et al. "Towards an AI co-scientist". In: *arXiv preprint arXiv: 2502.18864* (2025).

[26] Xingyao Wang et al. "Executable Code Actions Elicit Better LLM Agents". In: *Forty-first International Conference on Machine Learning*. 2024. URL: https://openreview.net/forum?id=jJ9BoXAfFa.

[27] Michael D. Skarlinski et al. "Language agents achieve superhuman synthesis of scientific knowledge". In: *arXiv preprint arXiv: 2409.13740* (2024).

[28] Yuan Chiang et al. "LLaMP: Large Language Model Made Powerful for High-fidelity Materials Knowledge Retrieval and Distillation". In: *arXiv preprint arXiv: 2401.17244* (2024).

[29] Shuxiang Cao et al. "Agents for self-driving laboratories applied to quantum computing". In: *arXiv preprint arXiv: 2412.07978* (2024).

[30] Quintina Campbell et al. "MDCrow: Automating Molecular Dynamics Workflows with Large Language Models". In: *arXiv preprint arXiv: 2502.09565* (2025).

[31] Yunheng Zou et al. "El Agente: An Autonomous Agent for Quantum Chemistry". In: *arXiv preprint arXiv: 2505.02484* (2025).

[32] Mhd Hussein Murtada, Z. Faidon Brotzakis, and Michele Vendruscolo. "MD-LLM-1: A Large Language Model for Molecular Dynamics". In: *arXiv preprint arXiv: 2508.03709* (2025).

[33] Indrajeet Mandal et al. "Autonomous Microscopy Experiments through Large Language Model Agents". In: *arXiv preprint arXiv: 2501.10385* (2024).

[34] Andres M. Bran et al. "Augmenting large language models with chemistry tools". In: *Nature Machine Intelligence* 6.5 (May 2024), pp. 525–535. ISSN: 2522-5839. DOI: 10.1038/s42256-024-00832-8. URL: http://dx.doi.org/10.1038/s42256-024-00832-8.

[35] Kourosh Darvish et al. "ORGANA: A robotic assistant for automated chemistry experimentation and characterization". In: *Matter* 8.2 (Feb. 2025), p. 101897. ISSN: 2590-2385. DOI: 10.1016/j.matt.2024.10.015. URL: http://dx.doi.org/10.1016/j.matt.2024.10.015.

[36] Daniil A. Boiko et al. "Autonomous chemical research with large language models". In: *Nature* 624.7992 (Dec. 2023), pp. 570–578. ISSN: 1476-4687. DOI: 10.1038/s41586-023-06792-0. URL: http://dx.doi.org/10.1038/s41586-023-06792-0.

[37] Hjalmar Wijk et al. "RE-Bench: Evaluating frontier AI R&D capabilities of language model agents against human experts". In: *arXiv preprint arXiv: 2411.15114* (2024).

[38] Megan Kinniment et al. "Evaluating Language-Model Agents on Realistic Autonomous Tasks". In: *arXiv preprint arXiv: 2312.11671* (2023).

[39] Zachary S. Siegel et al. "CORE-Bench: Fostering the Credibility of Published Research Through a Computational Reproducibility Agent Benchmark". In: *arXiv preprint arXiv: 2409.11363* (2024).

[40] Grégoire Mialon et al. "GAIA: a benchmark for General AI Assistants". In: *arXiv preprint arXiv: 2311.12983* (2023).

[41] Ludovico Mitchener et al. "BixBench: a Comprehensive Benchmark for LLM-based Agents in Computational Biology". In: *arXiv preprint arXiv: 2503.00096* (2025).

[42] Ziru Chen et al. "ScienceAgentBench: Toward Rigorous Assessment of Language Agents for Data-Driven Scientific Discovery". In: *arXiv preprint arXiv: 2410.05080* (2024).

[43] Jingyi Chai et al. "SciMaster: Towards General-Purpose Scientific AI Agents, Part I. X-Master as Foundation: Can We Lead on Humanity's Last Exam?" In: *arXiv preprint arXiv: 2507.05241* (2025).

[44] Minyang Tian et al. "SciCode: A Research Coding Benchmark Curated by Scientists". In: *arXiv preprint arXiv: 2407.13168* (2024).

[45] Hao Cui et al. "CURIE: Evaluating LLMs On Multitask Scientific Long Context Understanding and Reasoning". In: *arXiv preprint arXiv: 2503.13517* (2025).

[46] Siddharth Narayanan et al. “Aviary: training language agents on challenging scientific tasks”. In: *arXiv preprint arXiv: 2412.21154* (2024).

[47] Microsoft. *AutoGen: A Programming Framework for Agentic AI*. `https://github.com/microsoft/autogen`. Accessed: 2025-08-20. 2025. URL: `https://github.com/microsoft/autogen`.

[48] Sayash Kapoor et al. “AI Agents That Matter”. In: *Trans. Mach. Learn. Res.* 2025 (2025). URL: `https://openreview.net/forum?id=Zy4uFzMviZ`.

[49] Yue Huang et al. “MetaTool Benchmark for Large Language Models: Deciding Whether to Use Tools and Which to Use”. In: *arXiv preprint arXiv: 2310.03128* (2023).

[50] Thomas Carta et al. “Grounding Large Language Models in Interactive Environments with Online Reinforcement Learning”. In: *arXiv preprint arXiv: 2302.02662* (2023).

[51] Muning Wen et al. “Reinforcing language agents via policy optimization with action decomposition”. In: *arXiv preprint arXiv:2405.15821* (2024).

[52] Yifan Song et al. “Trial and error: Exploration-based trajectory optimization for llm agents”. In: *arXiv preprint arXiv:2403.02502* (2024).

[53] Shunyu Yao et al. “ReAct: Synergizing Reasoning and Acting in Language Models”. In: *International Conference on Learning Representations* (2022).

[54] Chan Hee Song et al. “LLM-Planner: Few-Shot Grounded Planning for Embodied Agents with Large Language Models”. In: *arXiv preprint arXiv: 2212.04088* (2022).

[55] Matthew K Horton et al. “Accelerated data-driven materials science with the Materials Project”. In: *Nature Materials* (2025), pp. 1–11.

[56] Yushi Bai et al. “LongBench v2: Towards Deeper Understanding and Reasoning on Realistic Long-context Multitasks”. In: *arXiv preprint arXiv: 2412.15204* (2024).

[57] Cheng-Ping Hsieh et al. “RULER: What’s the Real Context Size of Your Long-Context Language Models?” In: *arXiv preprint arXiv: 2404.06654* (2024).

Appendix

A Metrics

To evaluate the overall performance of the benchmark, we employ a suite of metrics that capture different aspects of the agent's capabilities, from correctness and efficiency to resource consumption.

A.1 Average Score

The average score provides a measure of the mean performance across all tasks. It is calculated as the average of the mean scores for each individual task.

Let T be the set of all task IDs, and for each task $t \in T$, let S_t be the set of scores for each trial of that task. The average score for a single task t is given by:

$$\mu_t = \frac{1}{|S_t|} \sum_{s \in S_t} s \tag{A1}$$

The overall average score is then the mean of these individual task averages:

$$\text{Average Score} = \frac{1}{|T|} \sum_{t \in T} \mu_t \tag{A2}$$

A.2 Overall Success Rate

The overall success rate is the average of the success rates of all individual tasks. A trial is considered successful if its score is greater than zero and no error message is generated.

Let T be the set of all tasks. The success rate for a single task t, denoted as SR_t, is the proportion of successful trials for that task. The overall success rate is then:

$$\text{Overall Success Rate} = \frac{1}{|T|} \sum_{t \in T} SR_t \tag{A3}$$

A.3 Pass@k

The pass@k [1] metric evaluates the probability that at least one out of k trials for a given task succeeds. It is a valuable metric for assessing the likelihood of achieving a correct solution within a limited number of attempts. The overall pass@k is the average of the task-specific pass@k values.

For a single task, let n be the total number of trials and c be the number of successful trials. The probability of a single trial succeeding is $p = c/n$. The pass@k is then calculated as:

$$\text{pass@}k = 1 - (1-p)^k = 1 - \left(1 - \frac{c}{n}\right)^k \tag{A4}$$

The overall pass@k is the average of this value across all tasks:

$$\text{Overall pass@}k = \frac{1}{|T|} \sum_{t \in T} \text{pass@}k_t \tag{A5}$$

A.4 Pass$^\wedge k$

The pass$^\wedge k$ (pass-hat-k) metric [1] measures the probability that all k trials for a given task succeed. This metric is a stricter evaluation of reliability and consistency. The overall pass$^\wedge k$ is the average of the task-specific pass$^\wedge k$ values.

Using the same notation as for pass@k, the pass$^\wedge k$ for a single task is given by:

$$\text{pass}^\wedge k = p^k = \left(\frac{c}{n}\right)^k \tag{A6}$$

The overall pass$^\wedge k$ is the average of this value across all tasks:

$$\text{Overall pass}^\wedge k = \frac{1}{|T|} \sum_{t \in T} \text{pass}^\wedge k \tag{A7}$$

A.5 Token Usage

To measure the computational cost, we record the token usage for each trial. The total token usage is the sum of tokens consumed across all trials in the benchmark. This can be broken down into prompt tokens, completion tokens, and total tokens.

A.6 Tool Call Statistics

We also track the usage and success of tool calls made by the agent. This includes the total number of successful and failed tool calls across all trials and tasks. For failed tool calls, the failure is also classified to `invalid_tool` (invalid tool name), `invalid_args` (invalid tool arguments) or `execution_error` (error in executing the tool).

A.7 Duration Metrics

We measure various aspects of the benchmark's execution time:

- **Total Tool Execution Duration:** The sum of the execution time for all tool calls across all trials.
- **Overall Average Trial Duration:** The average duration of a single trial across all tasks.
- **Total Duration:** The total time taken to run all trials for all tasks in the benchmark.

B The Benchmark Server API Routes

To facilitate interaction between the agent and the environments, we have developed a benchmark server with a *RESTful* API. This server exposes a set of endpoints that allow the agent to retrieve task information, interact with the provided tools, and submit its final answer. The API is designed to be clear, consistent, and stateless, adhering to the principles of REST. The available endpoints are detailed in Table B.1. Each endpoint is described with its HTTP method, URL path, a brief description of its functionality, and any relevant path or query parameters.

Table B.1: **Benchmark Server API Endpoints**. The table specifies all the endpoints that the server hosting the environments admits.

Method	Path	Description
	Task and Environment Discovery	
GET	`/tasks`	Retrieves a list of all available task IDs in the benchmark.
GET	`/dependency_chain`	Indicates whether any tasks in the benchmark are part of a dependency chain.
GET	`/tasks/{task_id}/prompt`	Fetches the specific prompt for a given task. The `{task_id}` is a path parameter representing the unique identifier for the task.
GET	`/tasks/{task_id}/guide`	Returns the complete environment guide for a task, including the prompt and a detailed description of the available tools. Accepts an optional `verbosity` query parameter (e.g., `MINIMAL` or `WORKFLOW`) to control the level of detail in the tool descriptions.
	Tool Interaction and Introspection	

Table B.1 continued from previous page

Method	Path	Description
GET	`/tasks/{task_id}/tools`	Returns a structured JSON object detailing the available tools for a task, compatible with function-calling APIs. The level of detail in the argument descriptions is controlled by the optional `verbosity` query parameter.
GET	`/tasks/{task_id}/tools/guide`	Provides a guide to the tools available for a specific task. Also accepts the optional `verbosity` query parameter.
POST	`/tasks/{task_id}/tools/execute`	Executes a specified tool with the provided arguments within the task's environment. The tool name and arguments are sent in the request body.
State Management and Task Submission		
GET	`/tasks/{task_id}/state`	Retrieves the current state of the specified task environment.
POST	`/tasks/{task_id}/submit`	Submits a final answer for the task. The submitted answer is in the request body. The response includes the score and the final state of the trial.
GET	`/tasks/{task_id}/status`	Gets the completion status of a task, including whether it is completed, the final score, the submitted answer, and tool usage statistics.
GET	`/tasks/{task_id}/trials`	Retrieves the states of all trials that have been run for a specific task.
GET	`/tasks/{task_id}/trials/{trial_id}`	Fetches the state of a specific trial for a given task, identified by its unique `{trial_id}`.

C Structured Tool Descriptions and Verbosity Levels

To systematically ablate the information provided to the agent, we developed a structured, tag-based format for all tool docstrings. This allows us to programmatically control the content of the tool descriptions presented to the LLM at different verbosity levels. When a new tool is developed, its function docstring is written once in a comprehensive, structured format. During the setup of a benchmark run, our framework dynamically parses this complete docstring. Based on the selected verbosity strategy (e.g., `BRIEF`, `WORKFLOW`), it then filters and formats the content to generate the precise, tailored tool description that will be exposed to the agent.

C.1 Documentation Tags

Each part of the tool's documentation is enclosed in a specific tag, allowing for granular filtering. The primary tags used are:

`[BRIEF]` A single-sentence, high-level summary of the tool's function.

`[DETAILED]` A more thorough explanation of what the tool does and how it works internally.

`[PROCEDURAL]` Guidance on when and why to use the tool in a problem-solving process.

`[CONTEXTUAL]` Information about the tool's operational context, such as its dependencies or data sources.

`[WORKFLOW_INTEGRATION]` An explicit, structured guide on how the tool fits into a multi-step plan, outlining prerequisites, the current action, and follow-up steps.

`[SYNTACTICAL]` Concrete, valid usage examples of the tool's syntax.

[RAISES] A list of potential exceptions or errors the tool might raise, including conditions and recovery suggestions.

[LIMITATIONS] Known limitations or edge cases where the tool might not perform as expected.

[EXAMPLES] Illustrative examples, often for arguments or return values.

C.2 Verbosity Levels

Our ablation studies focus on three distinct verbosity levels, which are constructed by including content from a specific subset of the tags described above.

- `BRIEF`**:** Includes only content from the `[BRIEF]` tags. This provides the agent with the most minimal functional description.
- `WORKFLOW`**:** Includes content from `[BRIEF]`, `[DETAILED]`, `[PROCEDURAL]`, `[CONTEXTUAL]`, and `[WORKFLOW_INTEGRATION]`. This level is designed to give the agent rich strategic and operational context.
- `COMPREHENSIVE`**:** Includes all available tags. This level equips the agent with complete knowledge for robust execution, including syntax, error handling, and limitations.

C.3 Example: The `get_structure_from_mp_text` Tool

The following examples demonstrate the exact text shown to the agent for the same tool at each of the three verbosity levels. Note that the descriptive tags themselves are filtered out, and only the enclosed content is presented.

Example: `BRIEF` **Verbosity Description**

Retrieve a pymatgen structure from Materials Project using its API and return CIF content as text.

Args:

`mp_id:` Materials Project identifier string.

Returns:

`str:` CIF content string containing the crystal structure data.

Example: `WORKFLOW` **Verbosity Description**

Retrieve a pymatgen structure from Materials Project using its API and return CIF content as text.

This tool connects to the Materials Project database to download crystal structure data for a given material ID. It retrieves the structure object and converts it to CIF (Crystallographic Information File) format, which is the standard format for storing crystal structure information. CIF is then returned as string

When to use: Use when you need to retrieve bulk crystal structures from the Materials Project database. Best suited for materials with known MP IDs. Usually first step in simulation workflows. Recommended for obtaining crystal structures for preparing bulk structures, supercells, bulk cells, slabs etc. Avoid when you need multiple structures.

How this tool works: Connects to Materials Project API using authentication key. Searches for the specified material ID (MP ID) in the database (MP ID is given as input parameter or if other tools are available to search for MP ID based on available information, then use those tools). Retrieves the pymatgen Structure object containing atomic positions and lattice parameters. Converts the structure to CIF format string for compatibility with other tools. Returns standardized crystallographic data suitable for further processing.

Workflow: 1. [PREREQUISITE] Ensure that the other more specific tools are not suitable and you dont have to retrieve multiple structures 2. [CURRENT] Apply this tool with a valid MP ID to retrieve bulk structure 3. [FOLLOW_UP] Use the CIF output with slab generation tools like enumerate_slabs_text to create slab structures

Args:

`mp_id`: Materials Project identifier string. The unique identifier used by Materials Project to catalog materials. Should be in the format "mp-XXXXX" where XXXXX is a numerical ID. This ID corresponds to a specific material entry in the Materials Project database.

Returns:

`str`: CIF content string containing the crystal structure data. A properly formatted CIF (Crystallographic Information File) string containing all necessary information about the crystal structure, including lattice parameters, atomic positions, space group, and symmetry operations. This format is widely compatible with crystallographic software and other structure analysis tools.

Example: `COMPREHENSIVE` **Verbosity Description**

Retrieve a pymatgen structure from Materials Project using its API and return CIF content as text.

This tool connects to the Materials Project database to download crystal structure data for a given material ID. It retrieves the structure object and converts it to CIF (Crystallographic Information File) format, which is the standard format for storing crystal structure information. CIF is then returned as string.

When to use: Use when you need to retrieve bulk crystal structures from the Materials Project database. Best suited for materials with known MP IDs. Usually first step in simulation workflows. Recommended for obtaining crystal structures for preparing bulk structures, supercells, bulk cells, slabs etc. Avoid when you need multiple structures.

How this tool works: Connects to Materials Project API using authentication key. Searches for the specified material ID (MP ID) in the database...

Workflow: 1. [PREREQUISITE] Ensure that the other more specific tools are not suitable... 2. [CURRENT] Apply this tool with a valid MP ID to retrieve bulk structure 3. [FOLLOW_UP] Use the CIF output with slab generation tools...

Usage examples:

```
get_structure_from_mp_text("mp-149")  # Silicon structure
get_structure_from_mp_text("mp-20066") # CO2 structure
```

Exceptions:

- `ConnectionError`: When unable to connect to Materials Project API. Recovery: Check internet connection and MP_API_KEY.
- `KeyError`: When the specified MP ID is not found. Recovery: Verify MP ID syntax and existence on the Materials Project website.

Limitations:

- Limited to materials available in the Materials Project database.
- May not include the most recent experimental structures.

Args:

`mp_id`: Materials Project identifier string. The unique identifier used by Materials Project... Should be in the format "mp-XXXXX"... Syntax: "mp-" followed by digits (e.g., "mp-149", "mp-20066"). Examples: "mp-149" (Silicon), "mp-20066" (CO2).

Returns:

`str`: CIF content string containing the crystal structure data. A properly formatted CIF string containing all necessary information... Example:

```
# generated using pymatgen
data_Si
_symmetry_space_group_name_H-M    'P 1'
_cell_length_a    3.83996459
...
```

D *Corral* Framework Usage Examples

The *Corral* framework is built using such a modular design that is simple to implement agents, add new environments, and run the benchmark. For example, the code block below shows how to benchmark an agent using the *Corral* framework. It needs to set up the environment, create a logger for tracking experiments, and then run a benchmark with an agent, outputting the results such as the overall score.

```python
from corral.evaluate import BenchmarkInterface, MatAgentBenchmark
from corral.agents.react import ReActAgent
from corral.report import CorralWandbLogger

# Setup environments server running
interface = BenchmarkInterface("http://localhost:8000")


# Setup the WandB logger
wandblogger = CorralWandbLogger(
    project="corral",
    group="experiment_group",
    name="run_name",
)

# Setup the agent
agent = ReActAgent(model="gpt-4o", max_iterations=10, temperature=0.1)

# Run benchmark
runner = MatAgentBenchmark(interface, agent, logger=wandblogger)
result = runner.bench()

print(f"Overall score: {result.total_score:.2f}")
```

The next code block focuses on setting up and running a *Corral* environment server. In it, a custom environment is defined by implementing the `Environment` class, specifying the task, problem, and expected answer. The server is then created to host the benchmark tasks and make them accessible to agents for evaluation.

```python
from corral.base import Environment
from corral.server import create_benchmark_server


class MyEnvironment(Environment):
```

```
    def __init__(self, task_id: str, problem: str, answer: str):
        self.problem = problem
        self.correct_answer = answer
        super().__init__(task_id)

        # Add your tools
        self.add_tool(my_custom_tool)

    def get_task_prompt(self) -> str:
        return f"Solve this problem: {self.problem}"

    def score(self) -> float:
        if self.state.submitted_answer is None:
            return 0.0
        return 1.0 if self.state.submitted_answer == self.
            correct_answer else 0.0


# Define your tasks
environments = {
    "task_1": MyEnvironment("task_1", "Problem 1", "Answer 1"),
    "task_2": MyEnvironment("task_2", "Problem 2", "Answer 2"),
}

# Create server
if __name__ == "__main__":
    app = create_benchmark_server(environments)
    import uvicorn

    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Similarly, the code block below demonstrates how to create a custom agent within the *Corral* framework. To that, one can define a custom agent class that extends `BaseAgent` and implement the logic for the agent's task processing. This agent interacts with the benchmark interface to retrieve task information and use tools to solve problems.

```
from corral.agents.base_agent import BaseAgent
from corral.evaluate import BenchmarkInterface


class MyAgent(BaseAgent):
    def __init__(self, model: str, **kwargs):
        super().__init__(model, **kwargs)
        # Add your agent-specific initialization

    def run(self, interface: BenchmarkInterface, task_id: str) -> str:
        # Get task information
        guide = interface.get_task_guide(task_id)

        # Your agent logic here
        # Use interface.execute_tool() to call tools

        return "Your final answer"
```

Lastly, the next code block defines a custom tool for the environment, `calculate_molecular_weight`, which can be used by agents to perform specific tasks, in this case, calculating the molecular weight of a given chemical formula.

```
from corral.utils import tool
```

```python
@tool
def calculate_molecular_weight(formula: str) -> float:
    """[BRIEF]Calculate molecular weight from chemical formula. [/
       BRIEF]

    Args:
        formula: Chemical formula (e.g., 'H2O', 'CH4')

    Returns:
        Molecular weight in g/mol
    """
    # Implementation here
    pass
```

E Detailed Environment Descriptions

We compiled four expert-curated environments, with over 35 domain-specific tools, and over 20 specialized scoring functions.

E.1 Overview of the Environments

We implement multiple environments, covering diverse problems in chemistry and materials research.

`Spectra` Spectroscopic analysis is a fundamental technique in chemistry for determining the structure of unknown compounds, using data from methods such as nuclear magnetic resonance (NMR), infrared (IR) spectroscopy, or mass spectrometry (MS). This environment evaluates LLM agents on the task of molecular structure elucidation from spectroscopic data, requiring them to plan the experiments and output a valid SMILES representation of the compounds. This environment mimics a virtual lab where the agent has to first plan what experiments to run, then propose different molecular hypotheses based on the different experiments that it performs, validate those hypotheses, and finally return the valid elucidated molecular structure.

`MD` MD simulations are widely used to study the atomic-scale behavior of materials, providing information on structural, thermal, and mechanical properties at timescales that are difficult to access experimentally [2]. The MD environment explores the ability of agents to use this technique with the software LAMMPS [3]. Agents in this environment can retrieve atomic structures from MP [4], write LAMMPS input scripts, and perform simulations to compute target properties. While the implementation is shared, we define three separate environments based on the task being performed: *Surface Energy*, *Melting*, and *Quenching* simulations.

`OpenCatalyst` This environment is inspired by the *Open Catalyst Project* computational workflow [5]. Catalysis plays a pivotal role in addressing critical global challenges, including renewable energy generation (e.g., fuel cells, solar fuels) and sustainable chemical production (e.g., ammonia synthesis). The discovery of novel catalysts is challenging due to the vast chemical space and diversity of possible surfaces, terminations, adsorption sites, and adsorbate orientations. This environment evaluates agents on performing the key and early steps of a typical catalysis simulation pipeline, which is to create a catalyst slab-adsorbate configuration.

`ML` ML models are increasingly used in materials science to accelerate property predictions and reduce reliance on costly first-principles simulations or experiments. This environment evaluates agents on constructing and training an XGBoost model [6] to predict the formation energies of oxide, nitride, and sulphide polymorphs using data from MP. Agents must retrieve relevant crystal structures, curate polymorph datasets, perform feature engineering, train, and evaluate the model with both test set metrics and cross-validation, and save the results in a structured format.

Below, we provide detailed descriptions of each environment, including associated tasks and scoring functions used to evaluate agent performance.

E.2 Spectra

This environment simulates a virtual lab, providing tools to run various experimental analyses, such as carbon or proton NMR. The scoring function of the single task is simply comparing the smiles to check if they correspond to the same molecule. The different scoring functions of the chained tasks correspond to counting the number of certain atoms, some functional groups, etc.

E.2.1 Tools

Note that for the experimental tools in the SPECTRA environment, no arguments are required. This is because agents are prompted in an environment in which the sample at hand is already loaded into the experimental equipment. For the actual "experimental" tools, we used simulation tools available through API services [7, 8, 9, 10]

Table E.2: Spectra **Tools**. The table describes each tool available in the spectra environment, including its arguments, return values, and potential errors.

Tool
`get_formula_from_smiles` **Arguments:** `smiles` (string): SMILES representation of a molecule. **Return:** Generates the chemical formula of the molecule in Hill notation (C, H, then alphabetical order). Uses RDKit to parse the SMILES string and compute the formula. Returns a string formula if successful, or an error message if the SMILES is invalid. **Raise:** Raises `Exception` for unparseable SMILES strings.
`search_by_smiles` **Arguments:** `smiles` (string): SMILES representation of the compound to search for in the NMRShift database. `top_k` (int, optional, default 10): Maximum number of results to return. **Return:** Searches the NMRShift database for entries matching or chemically similar to the provided SMILES. Returns a list of dictionaries containing relevant information about each compound, such as SMILES, chemical shifts, and other properties. Results are sorted by similarity score in descending order. **Raise:** Raises `Exception` if the search fails or if `top_k` is invalid.
`retrieve_aromatic_protons_shifts` **Arguments:** None **Return:** Retrieves the proton chemical shift ranges for aromatic hydrocarbons. Returns a string representation of a list of dictionaries, each containing the type of proton (e.g., aldehyde, aromatic, alkene) and its corresponding chemical shift range in ppm relative to TMS. **Raise:** No exceptions are raised.
`retrieve_carbon_shifts`

Tool

Arguments: None

Return: Retrieves the carbon chemical shift ranges for various functional groups in organic compounds. Returns a string representation of a list of dictionaries, each containing the functional group (e.g., CH_3-, R_3C-, $=CH$, Ketones, Aldehydes) and its corresponding chemical shift range in ppm relative to TMS.

Raise: No exceptions are raised.

`carbon_nmr_spectra`

Arguments: None

Return: Measures the ^{13}C NMR spectra for the given sample using the `get_c13_nmr_prediction` function. Returns the ^{13}C NMR spectra as a string in ACS-inspired publication format.

Raise: Raises an exception if an error occurs during measurement.

`proton_nmr_spectra`

Arguments: None

Return: Returns the ^{1}H NMR spectra for the given compound using the `get_h_nmr_prediction` function. Provides the ^{1}H NMR spectra as a string following ACS-inspired publication format.

Raise: Raises an exception if an error occurs during measurement.

`ir_spectra`

Arguments: None

Return: Returns the IR spectra for the given compound using the `get_ir_prediction` function. Provides the IR spectra as a string following standard IR notation.

Raise: Raises an exception if an error occurs during measurement.

`hsqc_nmr_spectra`

Arguments: None

Return: Returns the HSQC (Heteronuclear Single Quantum Coherence) NMR spectra for the given compound by making an external API call and formatting the result in standard NMR notation.

Raise: Raises an exception if an error occurs during measurement or if the HSQC spectrum is unavailable.

`mass_spectrometry_spectra`

Arguments: None

Return: Returns the mass spectrometry spectra for the given compound using Electrospray Ionization (ESI) by making an external API call. Outputs the spectrum in the format `m/z value (intensity), ...`.

Raise: Raises an exception if an error occurs during measurement or if the SMILES string is invalid.

Tool
`retrieve_isotope_distribution` **Arguments:** None **Return:** Returns a predefined dictionary containing the isotopic distribution of common elements in organic chemistry (C, H, S, Cl, Br, I, F, N, O). **Raise:** No calculations or API calls are performed; values are typical for organic compounds.
`retrieve_dbe_formula` **Arguments:** None **Return:** Returns the formula for calculating the Double Bond Equivalent (DBE), also known as Degree of Unsaturation (DU) or Index of Hydrogen Deficiency (IHD). **Raise:** No exceptions are raised.
`obtain_isomers` **Arguments:** `smiles` (string): SMILES representation of the compound. **Return:** Retrieves structural isomers for a given compound using its SMILES representation. Uses a remote function to query the PubChem database and return a list of SMILES strings for compounds with the same molecular formula. **Raise:** Returns an empty list if no isomers are found.
`obtain_isomers_from_molecular_formula` **Arguments:** `molecular_formula` (string): Molecular formula of the compound. **Return:** Retrieves structural isomers for a given molecular formula. Uses a remote function to query the PubChem database and return a list of SMILES strings for compounds with the same molecular formula. **Raise:** Returns an empty list if no isomers are found.
`validate_smiles` **Arguments:** `smiles` (string): SMILES representation of the compound. **Return:** Validates a SMILES string to check if it represents a valid chemical structure. Returns `True` if the conversion is successful (valid SMILES) and `False` otherwise. **Raise:** No exceptions are raised.

E.2.2 Scoring Functions

The scoring functions for the SPECTRA environment are described in Table E.3. Note that the `score_molecule` function is the one used to score the single task.

Table E.3: `Spectra` **scoring functions**. The table describes each scoring function, including its arguments, return values, and behavior.

Function
`score_molecule`

Function

Arguments: `prediction` (str, SMILES): Predicted SMILES string.
`ground_truth` (str, SMILES): Ground truth SMILES string.

Return: Returns 1.0 if the canonical SMILES match, ignoring stereochemistry, otherwise 0.0.

`score_molecule_fragments`

Arguments: `prediction` (list[str] or str): Predicted fragments (list or string).
`ground_truth` (str, SMILES): Ground truth SMILES string.

Return: Returns 1.0 if all predicted fragments are substructures of the ground truth molecule, otherwise 0.0.

`score_formula_match`

Arguments: `prediction` (str, formula): Predicted molecular formula.
`ground_truth` (str, SMILES): Ground truth SMILES string.

Return: Returns 1.0 if predicted molecular formula matches the ground truth formula, otherwise 0.0.

`validate_dbe_consistency`

Arguments: `prediction` (int): Predicted degree of unsaturation (DBE).
`ground_truth` (str, SMILES): Ground truth SMILES string.

Return: Returns 1.0 if predicted DBE matches calculated DBE from SMILES, otherwise 0.0.

`score_isotopic_distribution`

Arguments: `prediction` (list[str], element symbols): Predicted isotopic distribution of elements.
`ground_truth` (str, SMILES): Ground truth SMILES string.

Return: Returns 1.0 if predicted elements with isotopic distributions match those in the molecule, otherwise 0.0.

`score_num_hydrogen_symmetry_classes`

Arguments: `prediction` (int): Predicted number of hydrogen symmetry classes.
`ground_truth` (str, SMILES): Ground truth SMILES string.

Return: Counts unique hydrogen symmetry classes using canonical ranking. Returns 1.0 if predicted count matches, otherwise 0.0.

`score_num_carbon_symmetry_classes`

Arguments: `prediction` (int): Predicted number of carbon symmetry classes.
`ground_truth` (str, SMILES): Ground truth SMILES string.

Return: Counts unique carbon symmetry classes using canonical ranking. Returns 1.0 if predicted count matches, otherwise 0.0.

Function
`score_num_aromatic_carbons` **Arguments:** `prediction` (int): Predicted number of aromatic carbons. `ground_truth` (str, SMILES): Ground truth SMILES string. **Return:** Counts aromatic carbons in the molecule. Returns 1.0 if predicted count matches, otherwise 0.0.
`score_num_ch3_groups` **Arguments:** `prediction` (int): Predicted number of CH3 (methyl) groups. `ground_truth` (str, SMILES): Ground truth SMILES string. **Return:** Counts CH3 groups (methyl) in the molecule. Returns 1.0 if predicted count matches, otherwise 0.0.
`score_num_carbonyl_groups` **Arguments:** `prediction` (int): Predicted number of carbonyl groups. `ground_truth` (str, SMILES): Ground truth SMILES string. **Return:** Counts carbonyl groups (C=O) in the canonical tautomer of the molecule. Returns 1.0 if predicted count matches, otherwise 0.0.

E.2.3 Tasks and Subtasks

Here, we present the prompts for a single task along with its associated subtasks. The structure and format are consistent across other tasks within the SPECTRA environment. Note that for these tasks, there are no input parameters since the different machines are already configured.

Organic Compound Analysis

Analyze the provided organic compound sample in a lab environment and output the SMILES string, while minimizing resource consumption due to the costly nature of the process.

Subtask 1: Molecular Formula

State the molecular formula for the compound analyzed.

Subtask 2: Double Bond Equivalents (DBE)

Calculate the Double Bond Equivalents (DBE) for the compound analyzed. Provide your answer as an integer.

Subtask 3: Identify Elements from Mass Spectrum

Based on the isotope distribution patterns visible in the mass spectrum, list all elements that can be definitively identified. Format your answer as a list of element symbols: [ëlement1, ëlement2, ëlement3̈].

Subtask 4: Chemically Equivalent Carbon Atoms

How many different types of chemically equivalent carbon atoms are present in this compound? (Count the number of unique carbon environments, not the total number of carbons)

Subtask 5: Chemically Equivalent Hydrogen Atoms

How many different types of chemically equivalent hydrogen atoms are present in this compound? (Count the number of unique hydrogen environments, not the total number of hydrogens)

Subtask 6: Aromatic Carbons Count

State the count of aromatic carbons in the molecule.

Subtask 7: Methyl Groups Count

How many methyl (CH3) groups are present in the sample's molecule?

Subtask 8: Carbonyl Groups Count

Indicate the number of C=O groups present in the molecule.

Subtask 9: SMILES for Connected Fragments

Return the SMILES strings for the connected fragments of the compound.

Subtask 10: Complete SMILES

Provide the SMILES string representing the compound in the sample.

E.3 `MD`

We deploy the LAMMPS package [3] using the Modal Inference serverless infrastructure [11]. All input/output operations and tool executions are integrated with the Modal platform. This gives the flexibility to execute LAMMPS simulations without requiring local or cloud-based installation, and it automatically provides computational resources at the time of running the simulation. Structure files and potential files required for MD simulations are stored in separate volumes, which the agent can access directly during execution. Table E.4 lists the types of potentials available to the agent and the corresponding materials.

Table E.4: Potentials available to the agent.

Potential Type	**Applicable Materials**
Embedded Atom Method (EAM)	Aluminum, Magnesium, Copper
ReaxFF	Silicon, Alpha-Quartz, Wollastonite

This environment provides the underlying implementation for three task environments: Surface Energy, Melting Dynamics, and Quenching Dynamics.

E.3.1 Tools

Table E.5 provides a detailed description of the tools used in the `MD` environment, including what their arguments, return objects, and the possible exceptions raised.

Table E.5: **Tools for the `MD` environment**. The table describes each tool available in this environment, including its arguments, return values, and potential errors.

Tool
`get_potential_metadata` **Arguments:** `file_path` (string): Path to the LAMMPS potential file. **Return:** Returns structured metadata: potential type, supported elements, pair style. **Raise:** Raises `ValueError` if file is unrecognized or empty.
`get_structure_from_mp_text` **Arguments:** `mp_id` (string): Materials Project ID. `file_path` (string): Destination path for the CIF file. **Return/Behavior:** Retrieves crystal structure from Materials Project, saves CIF file, and returns a confirmation message. **Raise:** Raises `Exception` if retrieval or saving fails.
`convert_structure_to_lammps_data` **Arguments:** `structure_path` (string): Path to the CIF file. `output_file` (string): Output LAMMPS data path. `atom_style` (string, optional, default "charge"): Atom style for LAMMPS conversion. **Return/Behavior:** Converts CIF to a LAMMPS `.data` file using the specified atom style. Returns a confirmation message. **Raise:** Raises `Exception` if the files are invalid or the atom style is unsupported.
`run_lammps` **Arguments:** `input_file` (string): Path to the LAMMPS input script. **Return/Behavior:** Runs LAMMPS simulation via a remote backend, generates a `.log` file, and returns a success message with the log file path. **Raise:** Raises `ValueError` for simulation errors or a generic `Exception` for unexpected issues.

E.3.2 Scoring Functions

For each task environment, the ground truth is generated using predefined simulation parameters. Agents are then evaluated by comparing their outputs against this ground truth, given the same simulation parameters. For numerical outputs, a tolerance of 2% is applied to account for stochastic variations in molecular dynamics (MD) simulations.

Table E.6: **Scoring Functions in the `MD` environment**. The table describes each scoring function available in this environment, including its arguments, return values, and behavior.

Function
`check_potential_file` **Arguments:** `target` (string): Path to the reference potential file. **Return:** Returns a scoring function. Checks if the result filename matches `target` and verifies file existence via Modal. Returns 1.0 if both pass, otherwise 0.0.
`check_numerical` **Arguments:** `target` (float): Expected numerical value. `tolerance` (float): Allowed relative tolerance. **Return:** Returns a scoring function. Parses the numeric result (string, JSON, or dict), compares against target ± tolerance, and checks associated file existence via Modal. Returns 1.0 if all checks pass, otherwise 0.0.
`check_structure` **Arguments:** `target` (string): Path to the reference LAMMPS `.data` file. `atom_style` (string): LAMMPS atom style. **Return:** Returns a scoring function. Reads remote LAMMPS `.data` result via Modal, loads reference and result structures with pymatgen's `LammpsData`, compares using `StructureMatcher`. Returns 1.0 if structures match, otherwise 0.0. Temporary files are automatically cleaned.

E.3.3 Tasks and Subtasks

Here we provide an example of a task and its associated subtask prompts for Aluminum in each task environment. The prompt structure is similar for other materials.

Surface Energy of Aluminum

Calculate the surface energy of Aluminum for the given surface by removing periodic boundary conditions in the given direction and computing the energy difference per unit area between the relaxed bulk and the relaxed slab configuration. Use the Embedded Atom Method (EAM) potential and the following simulation parameters. The simulation should replicate the unit cell 5 times in each of the x, y, and z directions. Return the surface energy in electronvolts/Å^2 units.

Input Parameters:

```
{
  "units": "metal",
  "minimiser": "conjugate gradient",
  "energy_tolerance": 1e-10,
  "force_tolerance": 1e-10,
  "maximum_iterations": 100000.0,
  "maximum_evaluations": 100000.0,
  "materials_project_id": "mp-134",
  "direction": [0, 0, 1],
  "atom_style": "atomic"
}
```

Subtask 1 : Retrieve Aluminum Structure

Description: Retrieve structure of Aluminum and convert it to LAMMPS data format. Return the absolute path of the final structure.

Input Parameters:

```
{
  "materials_project_id": "mp-134",
  "atom_style": "atomic"
}
```

Subtask 2 : Procure EAM Potential File

Description: Procure the appropriate Embedded Atom Method (EAM) potential file for Aluminum for MD simulations. Return the absolute path to the potential file.

Input Parameters: None

Subtask 3 : Bulk Relaxation for Surface Energy

Description: Given paths to LAMMPS-compatible structure file and EAM potential file, perform structure relaxation of Aluminum (5×5×5 replication). Report total energy (eV) and saved path.

Input Parameters:

```
{
  "units": "metal",
  "pbc": "true",
  "minimiser": "conjugate gradient",
  "energy_tolerance": 1e-10,
  "force_tolerance": 1e-10,
  "maximum_iterations": 1e+5,
  "maximum_evaluations": 1e+5,
  "atom_style": "atomic"
}
```

Subtask 4 : Slab Relaxation for Surface Energy

Description: Create a slab by removing periodic boundary conditions along the given direction, perform structure relaxation (5×5×5 replication). Report total energy (eV) and saved path.

Input Parameters:

```
{
  "units": "metal",
  "minimiser": "conjugate gradient",
  "energy_tolerance": 1e-10,
  "force_tolerance": 1e-10,
  "maximum_iterations": 1e+5,
  "maximum_evaluations": 1e+5,
  "atom_style": "atomic",
  "direction": [0, 0, 1]
}
```

Subtask 5 : Surface Energy Calculation

Description: Compute surface energy from relaxed bulk and slab structures. Return energy in eV/Å^2.

Input Parameters: None

Aluminum Melting Simulation

Simulate the melting of Aluminum by first equilibrating the system under NVT conditions, then relaxing it under NPT conditions, followed by a heating stage under NPT where the temperature is gradually increased to induce melting. Use the Embedded Atom Method (EAM) potential and the given simulation parameters. The simulation should replicate the unit cell 5 times in each direction. Report the final density of the system in g/cm^3.

Input Parameters:

```
{
  "units": "metal",
  "pbc": "true",
  "timestep": "1 femtosecond",
  "nvt steps": 10000,
  "nvt temperature": 300,
  "nvt thermostat": "Nose-Hoover",
  "npt relaxation steps": 10000,
  "npt relaxation temperature": 300,
  "npt relaxation pressure": 0,
  "npt relaxation thermostat": "Nose-Hoover",
  "npt relaxation barostat": "Nose-Hoover",
  "npt heating steps": 20000,
  "npt heating initial temperature": 300,
  "npt heating final temperature": 2000,
  "npt heating pressure": 0,
  "npt_heating_thermostat": "Nose-Hoover",
  "npt_heating_barostat": "Nose-Hoover",
  "materials_project_id": "mp-134",
  "atom_style": "atomic"
}
```

Subtask 1: Retrieve Aluminum Structure

Retrieve structure of Aluminum and convert it to LAMMPS data format. Return the absolute path of the final structure.

Input Parameters:

```
{
  "materials_project_id": "mp-134",
  "atom_style": "atomic"
}
```

Subtask 2: Procure EAM Potential File

Procure the appropriate Embedded Atom Method (EAM) potential file for Aluminum for MD simulations. Return the absolute path to the potential file.

Input Parameters: None

Subtask 3: NVT Equilibration

Given paths to LAMMPS-compatible Aluminum structure file and EAM potential file, equilibrate the system under NVT conditions. Replicate the unit cell 5×5×5. Return the absolute path of the equilibrated structure and final density (g/cm^3).

Input Parameters:

```
{
  "units": "metal",
  "pbc": "true",
  "timestep": "1 femtosecond",
  "nvt steps": 10000,
  "nvt temperature": 300,
  "nvt thermostat": "Nose-Hoover",
  "atom_style": "atomic"
}
```

Subtask 4: NPT Relaxation

Given paths to NVT equilibrated Aluminum structure and EAM potential file, relax the system under NPT conditions. Return the absolute path of the relaxed structure and density (g/cm^3).

Input Parameters:

```
{
  "units": "metal",
  "pbc": "true",
  "timestep": "1 femtosecond",
  "npt relaxation steps": 10000,
  "npt relaxation temperature": 300,
  "npt relaxation pressure": 0,
  "npt relaxation thermostat": "Nose-Hoover",
  "npt relaxation barostat": "Nose-Hoover",
  "atom_style": "atomic"
}
```

Subtask 5: NPT Heating / Melting Simulation

Given paths to NPT equilibrated Aluminum structure and EAM potential file, simulate melting by heating under NPT to gradually increase temperature. Report the final density (g/cm^3).

Input Parameters:

```
{
  "units": "metal",
  "pbc": "true",
  "timestep": "1 femtosecond",
  "npt heating steps": 20000,
  "npt heating initial temperature": 300,
  "npt heating final temperature": 2000,
  "npt heating pressure": 0,
  "npt_heating_thermostat": "Nose-Hoover",
  "npt_heating_barostat": "Nose-Hoover",
  "atom_style": "atomic"
}
```

Aluminum Quenching Simulation

Simulate the quenching process of Aluminum by cooling its molten structure under NPT conditions, followed by relaxation under NPT and NVT conditions. Use the Embedded Atom Method (EAM) potential and the given simulation parameters. Report the final density of the system in g/cm^3.

Input Parameters:

```
{
  "path to molten structure": "/structures/task_30/
     heatedsystem_3000_reax.dat",
  "units": "metal",
  "pbc": "true",
  "timestep": "1 femtosecond",
  "npt quench steps": 20000,
  "npt quench initial temperature": 2000,
  "npt quench final temperature": 300,
  "npt quench pressure": 0,
  "npt quench thermostat": "Nose-Hoover",
  "npt quench barostat": "Nose-Hoover",
  "npt relaxation steps": 10000,
  "npt relaxation temperature": 300,
  "npt relaxation pressure": 0,
  "npt relaxation thermostat": "Nose-Hoover",
  "npt relaxation barostat": "Nose-Hoover",
  "nvt steps": 10000,
  "nvt temperature": 300,
  "nvt thermostat": "Nose-Hoover",
  "materials_project_id": "mp-134",
  "atom_style": "atomic"
}
```

Subtask 1: Procure EAM Potential File

Procure the appropriate Embedded Atom Method (EAM) potential file for Aluminum for MD simulations. Return the absolute path to the potential file.

Input Parameters: None

Subtask 2: NPT Quenching

Given paths to LAMMPS-compatible molten Aluminum structure and EAM potential file, simulate the quenching process by cooling the molten structure under NPT conditions. Return the absolute path of the cooled structure and the density in g/cm^3.

Input Parameters:

```
{
  "path to molten structure": "/structures/task_30/
     heatedsystem_3000_reax.dat",
  "units": "metal",
  "pbc": "true",
  "timestep": "1 femtosecond",
  "npt quench steps": 20000,
  "npt quench initial temperature": 2000,
  "npt quench final temperature": 300,
  "npt quench pressure": 0,
  "npt quench thermostat": "Nose-Hoover",
  "npt quench barostat": "Nose-Hoover",
  "atom_style": "atomic"
}
```

Subtask 3: NPT Relaxation

Given paths to LAMMPS-compatible Aluminum structure and EAM potential file, relax the structure under NPT conditions. Return the absolute path of the relaxed structure and density (g/cm^3).

Input Parameters:

```
{
  "units": "metal",
  "pbc": "true",
  "timestep": "1 femtosecond",
  "npt relaxation steps": 10000,
  "npt relaxation temperature": 300,
  "npt relaxation pressure": 0,
  "npt relaxation thermostat": "Nose-Hoover",
  "npt relaxation barostat": "Nose-Hoover",
  "atom_style": "atomic"
}
```

Subtask 4: NVT Relaxation

Given paths to LAMMPS-compatible Aluminum structure and EAM potential file, relax the structure under NVT conditions. Report the final density (g/cm^3).

Input Parameters:

```
{
  "units": "metal",
  "pbc": "true",
  "timestep": "1 femtosecond",
  "nvt steps": 10000,
  "nvt temperature": 300,
  "nvt thermostat": "Nose-Hoover",
  "atom_style": "atomic"
}
```

E.4 OpenCatalyst

E.4.1 Tools

The tools of the OPENCATALYST environment are described in Table E.7. These tools are highly based on functions from the Pymatgen package [12].

Table E.7: OpenCatalyst **tools**. This table describes each tool available in the environment, including its arguments, return values, and potential errors.

Tool
`get_structure_from_mp_text`
Arguments: `mp_id` (string): Materials Project identifier string. The unique identifier used by Materials Project to catalog materials, in the format "mp-XXXXX".
Return: Returns a CIF content string containing the crystal structure data, including lattice parameters, atomic positions, space group, and symmetry operations.
Raise: `ConnectionError`: If unable to connect to the Materials Project API due to network issues or server downtime. `KeyError`: If the specified MP ID is not found in the database, indicating an invalid or non-existent material ID. `AuthenticationError`: If the API key is invalid or missing, preventing access to the Materials Project database.
`enumerate_slabs_text`

Tool

Arguments: `bulk_cif` (string): Bulk crystal structure in CIF string format.
`miller_index` (tuple): Miller indices for surface orientation. Defaults to (1,1,1).
`min_slab_size` (float): Minimum slab thickness in Angstroms. Defaults to 12.
`min_vacuum_size` (float): Minimum vacuum layer thickness in Angstroms. Defaults to 5.

Return: Returns a JSON string mapping slab indices to their CIF representations.

Raise: `ValueError`: If CIF string is malformed or parameters are invalid.
`StructureError`: If slab generation fails for the given structure.

`choose_slab_text`

Arguments: `slabs_json` (string): JSON string mapping slab keys to CIF strings. A JSON-formatted string containing a dictionary where keys are slab identifiers (e.g., "slab_0", "slab_1") and values are the corresponding CIF strings.
`index` (int, optional): Index of the slab to select. Defaults to 0. The numerical index of the slab to select from the JSON dictionary.

Return: Returns a CIF string for the selected slab, containing atomic positions, lattice parameters, and other crystallographic information.

Raise: `ValueError`: Raised if the specified slab index is not found in the JSON dictionary.
`JSONDecodeError`: Raised if the `slabs_json` string is not valid JSON.

`get_adsorption_sites_text`

Arguments: `slab_cif` (string): CIF string of the surface slab structure, including atomic positions, lattice parameters, and surface geometry.

Return: Returns a JSON string containing classified adsorption sites with fractional coordinates.

Raise: `ValueError`: If the CIF string is malformed or doesn't represent a valid slab.
`StructureError`: If the slab structure cannot be analyzed for adsorption sites due to geometric issues.

`choose_adsorption_site_text`

Arguments: `adsorption_sites_json` (string): A valid JSON string containing classified adsorption sites, where keys are site types (e.g., "ontop", "bridge", "hollow") and values are lists of fractional coordinates.
`site_type` (string): The type of adsorption site to select (e.g., "ontop", "bridge", "hollow").
`index` (integer, optional): The index of the site to select within the specified site type. Defaults to 0.

Return: Returns a list of three floating-point numbers representing the fractional coordinates [x, y, z] of the selected adsorption site.

Raise: `ValueError`: If the specified `site_type` is not found in the JSON string.
`IndexError`: If the specified `index` is out of range for the available sites of the specified `site_type`.
`JSONDecodeError`: If the `adsorption_sites_json` string is not valid JSON.

Tool
`add_adsorbate_to_slab_text` **Arguments:** `slab_cif` (string): CIF string of the surface slab structure. `adsorbate_cif` (string): CIF or XYZ string of the adsorbate molecule structure. `height` (float, optional): Height in Angstroms above the surface for adsorbate placement (default is 2.0). `site` (list of floats or None, optional): Fractional coordinates for adsorbate placement [x, y, z], or None to auto-select site (default is None). **Return:** Returns a CIF string of the combined surface-adsorbate structure. **Raise:** `ValueError`: If CIF strings are malformed or the adsorbate cannot be parsed. `StructureError`: If adsorbate placement fails due to geometric constraints. `RuntimeError`: If no adsorption sites are found on the surface.

E.4.2 Scoring Functions

Table E.8 describes the different scoring function used for scoring the OpenCatalyst environment within the *Corral* environment.

Table E.8: OpenCatalyst scoring functions, their arguments, and return values.

Function
`resolve_path` **Arguments:** `path_or_str` (str): Path or string to resolve. **Return:** Returns the resolved path as a string.
`check_valid_json_file` **Arguments:** `json_path` (str): File path to the JSON file. **Return:** Returns 1.0 if a valid, non-empty JSON file exists at the path; otherwise, 0.0.
`check_slabs_json` **Arguments:** `slabs_json` (str): JSON string or file path containing slab data. **Return:** Returns 1.0 if the JSON contains at least one valid slab (parsable CIF); otherwise 0.0.
`check_mp_structure` **Arguments:** `path_or_cif` (str): CIF string or file path. **Return:** Returns 1.0 if a valid CIF structure from Materials Project; otherwise, 0.0.
`check_slab_structure` **Arguments:** `path_or_cif` (str): CIF string or file path. **Return:** Returns 1.0 if a valid slab structure; otherwise, 0.0.

Function
`check_co2_molecule_structure` **Arguments:** `path_or_cif` (str): CIF string or file path. **Return:** Returns 1.0 if the structure contains exactly one CO_2 molecule (1 C, 2 O atoms); otherwise, 0.0.
`check_adsorption_structure` **Arguments:** `slab_elements` (list[str]): List of slab elements. `adsorbate_elements` (list[str]): List of adsorbate elements. **Return:** Returns a scoring function that scores 1.0 if the structure contains both slab and adsorbate elements; otherwise, it scores 0.0.
`check_adsorption_sites` **Arguments:** `sites_json_or_path` (str): JSON string or file path containing adsorption sites. **Return:** Returns 1.0 if the adsorption sites JSON contains at least one recognized site type with valid 3D coordinates; otherwise, 0.0.

E.4.3 Tasks and Subtasks

Here we provide an example of a task and its associated subtask prompts for CO_2 adsorption on Cu_2O. The prompt structure is similar for other compounds.

CO_2 Adsorption on Cu_2O Slab

Create a CO_2 adsorbed structure on a Silicon/Cu_2O slab. Submit the path to the final combined structure CIF file.

Input Parameters:

```
{
  "mp_id": "mp-149",
  "co2_mp_id": "mp-644607",
  "miller_index": [1, 1, 1],
  "min_slab_size": 12,
  "min_vacuum_size": 5,
  "slab_index": 0,
  "site_type": "ontop",
  "site_index": 0,
  "height": 2.0
}
```

Subtask 1: Retrieve Bulk Structure

Retrieve structure of Cu_2O from Materials Project and save it as a CIF file. Submit the path to the CIF file.

Input Parameters:

```
{
  "mp_id": "mp-361"
}
```

Subtask 2: Enumerate Slabs

Enumerate possible slabs from the bulk Cu_2O structure and save the result as a JSON file. Submit the path to the JSON file.

Input Parameters:

```
{
  "miller_index": [1, 1, 1],
  "min_slab_size": 12,
  "min_vacuum_size": 5
}
```

Subtask 3: Select Slab

Choose one slab from the enumerated slabs (by index) and save it as a CIF file. Submit the path to the CIF file.

Input Parameters:

```
{
  "index": 0
}
```

Subtask 4: Retrieve CO_2 Molecule

Retrieve CO_2 molecule structure from Materials Project and save it as a CIF file. Submit the path to the CIF file.

Input Parameters:

```
{
  "mp_id": "mp-644607"
}
```

Subtask 5: Determine Adsorption Sites

Determine possible adsorption sites on the chosen slab and save the results as a JSON file. Submit the path to the JSON file.

Input Parameters: None

Subtask 6: Select Adsorption Site

Choose one adsorption site (preferably ontop site) from the identified sites and save the coordinates to a file. Submit the path to the file.

Input Parameters:

```
{
  "site_type": "ontop",
  "index": 0
}
```

Subtask 7: Place CO_2 on Slab

Place the CO_2 molecule on the chosen slab at the specified adsorption site with a height of approximately 2.0 Å and save the combined structure as a CIF file. Submit the path to the CIF file.

Scoring Parameters:

```
{
  "slab_elements": ["Cu", "O"],
  "adsorbate_elements": ["C", "O"]
}
```

E.5 ML

E.5.1 Tools

Table E.9 shows the different tools available for the agents when working in the ML environment.

Table E.9: ML tools, their arguments, and return values.

Tool
`get_bulk_polymorphs_data` **Arguments:** `composition` (string): Chemical formula specifying the composition for which polymorphs should be retrieved. Should follow standard chemical notation (e.g., TiO_2, Al_2O_3). **Return:** Returns a JSON string containing comprehensive polymorph data, sorted by energy above hull. The data includes properties like Materials Project ID, CIF structure, energy above hull, formation energy per atom, band gap, density, volume, number of sites, space group, and stability information. **Raise:** `KeyError`: Raised when the specified composition is not found in the database. `ValueError`: Raised when the Materials Project API key is not available. `ConnectionError`: Raised when there is a failure to connect to the Materials Project API.
`get_bulk_polymorphs_data_to_file`

Tool

Arguments: `composition` (string): Chemical formula specifying the composition for which polymorphs should be retrieved. Example: TiO_2 (titanium dioxide), "Al2O3" (aluminum oxide).

`save_path` (string or None): Path where JSON data will be saved. Must include the filename with `.json` extension. If None, an error is raised.

Return: Returns the file path where polymorph data was saved in JSON format.

Raise: `ValueError`: If `save_path` is None or API key is missing.

`IOError`: If unable to write to the specified file path (e.g., directory does not exist or is not writable).

`batch_retrieve_polymorphs`

Arguments: `compositions` (list of strings): List of chemical compositions to retrieve polymorphs for.

`max_energy_above_hull` (float, optional): Maximum energy above hull threshold in eV/atom. Defaults to 0.5.

`max_per_composition` (int, optional): Maximum number of polymorphs per composition. Defaults to 10.

`save_directory` (string, optional): Directory path for saving individual composition files. Defaults to "polymorph_data".

`work_dir` (string, optional): Working directory for the tool (hidden argument).

Return: Returns a JSON string containing batch retrieval results and statistics, including successful and failed compositions, total number of polymorphs, and file paths for each composition.

Raise: Raises `ValueError` if parameters are invalid (negative energy, zero max_per_composition). Raises `IOError` if unable to create the save directory or write files.

`sort_and_get_first_from_json`

Arguments: `polymorph_data_json` (string): JSON string containing the data to be sorted, structured as a list of dictionaries with numerical properties for sorting.

`sort_key` (string): Property name to sort the data by (e.g., "energy_above_hull", "band_gap", "density").

`return_key` (string): Property name to return from the first element after sorting (e.g., "material_id", "cif").

Return: Returns the value of the specified `return_key` from the first element after sorting.

Raise: `JSONDecodeError`: If the `polymorph_data_json` is not valid JSON.

`KeyError`: If `sort_key` or `return_key` are not found in the data.

`IndexError`: If the JSON data is empty or contains no elements.

`select_polymorphs_with_strategy`

Tool

Arguments: `polymorphs_data` (string): JSON string or file path containing polymorph data.

`selection_strategy` (string, optional): "diverse_energy", "most_stable", or "diverse_structure" to determine selection method. Defaults to "diverse_energy"".

`max_polymorphs` (int, optional): Maximum number of polymorphs to select. Defaults to 5.

`energy_threshold` (float, optional): Maximum energy above hull in eV/atom. Defaults to 0.5.

`is_path` (bool, optional): Whether `polymorphs_data` is a file path. Defaults to `False`.

Return: Returns a JSON string containing selected polymorphs based on the specified strategy.

Raises: `ValueError`: If invalid selection strategy is specified.

`FileNotFoundError`: If `is_path=True` but file doesn't exist.

`JSONDecodeError`: If `polymorphs_data` contains invalid JSON.

`consolidate_polymorph_datasets`

Arguments: `composition_files` (dict): Dictionary mapping compositions to their JSON file paths.

`output_path` (string, optional): Path for the consolidated dataset (default: "consolidated_polymorphs.json").

`work_dir` (string, optional): Working directory for intermediate steps (hidden argument).

Return: Returns a JSON-formatted string with consolidation results and detailed statistics (e.g., total polymorphs, number of compositions included, etc.).

Raise: `FileNotFoundError`: If one or more input files cannot be found.

`JSONDecodeError`: If input files contain invalid JSON.

`IOError`: If unable to write to the output path.

`select_polymorphs_with_strategy_to_file`

Tool

Arguments: `polymorphs_data` (string): JSON string or file path containing polymorph data.

`save_path` (string): File path where selected polymorphs will be saved in JSON format.

`selection_strategy` (string): selection strategy ("diverse_energy", "most_stable", "diverse_structure"). Default is `diverse_energy`.

`max_polymorphs` (int): Maximum number of polymorphs to select. Default is 5.

`energy_threshold` (float): Maximum energy above hull in eV/atom. Default is 0.5.

`is_path` (bool): Whether `polymorphs_data` is a file path. Default is `False`.

`work_dir` (string | None): Directory to work in (optional).

Return: Returns the file path where the selected polymorphs were saved.

Raise: `ValueError`: If an invalid selection strategy is specified.

`FileNotFoundError`: If `is_path=True` but the input file doesn't exist.

`IOError`: If unable to write to the `save_path` location.

`filter_json_with_strategy`

Arguments: `input_json_path` (string): Path to the input JSON file to be filtered.

`output_json_path` (string): Path where filtered JSON data will be saved.

`custom_code` (string | None): Python code string defining the filtering logic.

`work_dir` (string | None): Directory to work in, optional.

Return: Returns a JSON string with filtering results and comprehensive statistics, including original and filtered data counts, output file path, and reduction percentage.

Raise: `FileNotFoundError`: Raised when the input JSON file doesn't exist.

`JSONDecodeError`: Raised when the input file contains invalid JSON.

`SyntaxError`: Raised when the custom filtering code contains syntax errors.

`RuntimeError`: Raised when the custom filtering code fails during execution.

`prepare_tabular_dataset`

Tool

Arguments: `polymorphs_json_path` (string): Path to consolidated polymorphs JSON file containing material properties and crystal structures.

`output_path` (string): Base directory and filename prefix for saving the dataset files.

`target_property` (string, optional): Property to predict (defaults to "formation_energy_per_atom").

`feature_engineering` (string): feature engineering strategy ("basic", "advanced", "custom").

`test_split` (float, optional): Fraction of data for test set (defaults to 0.2).

`normalize` (bool, optional): Whether to normalize features (defaults to True).

`work_dir` (string, optional): Working directory for intermediate files.

Return: Returns a JSON string with dataset preparation results, including file paths for training and test data, and other metadata.

Raise: `FileNotFoundError` if the polymorphs JSON file is not found.

`KeyError` if the target property is not found in the data.

`ValueError` if feature engineering fails or the data format is invalid.

`evaluate_xgboost_model`

Arguments: `model_path` (string): Path to the saved XGBoost model file.

`test_data_path` (string): Path to the test data CSV file with the same structure as the training data.

`target_column` (string, default = ""formation_energy_per_atom"): The column name in the test data representing the target variable for evaluation.

`detailed_analysis` (bool, default = True): Whether to include detailed analysis and feature importance in the results.

Return: Returns a JSON string containing evaluation success status and metrics, including MAE, RMSE, R2, MAPE, prediction ranges, error analysis, and feature importance (if detailed analysis is enabled).

Raise: `FileNotFoundError`: Raised when the model file or test data file doesn't exist.

`KeyError`: Raised when the target column is not found in the test data.

`ValueError`: Raised when the model and data are incompatible or contain invalid values.

`train_xgboost_model`

Tool
Arguments: `train_data_path` (string): Path to the training data CSV file containing features and target column. `test_data_path` (string): Path to the test data CSV file, with the same structure as the training data. `model_save_path` (string): Path to save the trained model file in `.pkl` format. `target_column` (string, default="formation_energy_per_atom"): Name of the target column for prediction in the dataset. `hyperparameters` (dict | None, optional): Dictionary of XGBoost hyperparameters to override default values. `work_dir` (string | None, optional): Working directory path, if provided. **Return:** Returns a JSON-formatted string with comprehensive training results, including model performance metrics (MAE, RMSE, R2), feature importance, and the model path. **Raises:** `FileNotFoundError`: Raised when the specified CSV files do not exist. `KeyError`: Raised when the target column does not exist in the dataset. `ValueError`: Raised when the data contains invalid values or formatting issues.
`perform_cross_validation` **Arguments:** `train_data_path` (string): Path to the training data CSV file. `target_column` (string): Name of the target column for prediction (default: "formation_energy_per_atom"). `cv_folds` (int): Number of cross-validation folds (default: 5). `hyperparameters` (dict or None): XGBoost hyperparameters for cross-validation (default: None). **Return:** Returns a JSON string containing comprehensive cross-validation results and statistical analysis, including R2, MAE, and other performance metrics. **Raise:** `FileNotFoundError`: Raised if the training data file does not exist or is not found. `KeyError`: Raised if the target column is not found in the dataset. `ValueError`: Raised if the number of cross-validation folds is invalid or the data contains invalid values.

E.5.2 Scoring Functions

Table E.10 show and describe the scoring functions used in the `ML` environment.

Table E.10: **Scoring Functions for the `ML` Environment**. The table describes each scoring function, its arguments, return values, and potential behaviors.

Function
`check_mp_structure` **Arguments:** `path_or_cif` (string): Path to CIF file or CIF string. **Return:** Returns 1.0 if the input is a valid CIF string or file representing a non-empty structure (via `pymatgen.Structure`), otherwise 0.0.

Function

`compare_with_ground_truth`

Arguments: `generated_path` (string): Path to generated JSON file.
`ground_truth_path` (string): Path to reference JSON file.
`comparison_mode` (string, optional): "strict", "keys", "numerical", or "subset".
`tolerance` (float, optional): Numerical tolerance.

Return: Returns 1.0 if the generated file matches the ground truth according to the chosen mode, otherwise 0.0. Modes control the exactness of comparison (structure, keys, numerical values, subset).

`ml_pipeline_score`

Arguments: `model_path` (string): Path to trained ML model file.

Return: Returns a float score (0.0-1.0) evaluating the completeness and performance of an ML pipeline, including model loading, evaluation metrics, and cross-validation evidence.

`polymorph_retrieval_success`

Arguments: `retrieval_results_path` (string): Path to JSON retrieval results.

Return: Returns a float score (0.0-1.0) based on polymorph retrieval success rate, total number retrieved, and distribution per composition.

`score_polymorph_dataset`

Arguments: `consolidated_json_path` (string): Path to consolidated polymorph dataset JSON.

Return: Returns 1.0 if at least one composition has multiple polymorphs, otherwise 0.0.

`ml_dataset_preparation_quality_binary`

Arguments: `ml_metadata_path` (string): Path to ML dataset metadata JSON.

Return: Returns 1 if dataset preparation meets all criteria: existing train/test files, sufficient sample sizes, and adequate feature count; otherwise returns 0.

`model_training_success_binary`

Arguments: `model_path` (string): Path to trained model file.

Return: Returns 1 if model can be loaded, has a `predict` method, and meets performance thresholds ($R^2 \geq 0.5$, MAE ≤ 0.5), otherwise 0.

`model_evaluation_completeness_binary`

Arguments: `evaluation_results_path` (string): Path to model evaluation results JSON.

Return: Returns 1 if evaluation results are complete: includes required metrics (MAE, RMSE, R^2), $R^2 \geq 0.7$, cross-validation stats, feature importance, and detailed analysis; otherwise returns 0.

E.5.3 Tasks and Subtasks

Here we provide an example of a task and its associated subtask prompts for oxide. The prompt structure is similar for sulphide and nitride compounds.

Oxide Polymorph Dataset and XGBoost Model

Generate a comprehensive dataset of oxide polymorphs from Materials Project and train an XGBoost model to predict formation energies. Evaluate the trained XGBoost model using test set and cross-validation metrics. Save the results as a JSON file with keys `test_set_evaluation` and `cross_validation_results`. The `test_set_evaluation` dictionary must contain `mae`, `rmse`, `r2`, and `feature_importance`. The `cross_validation_results` dictionary must contain `r2_mean` and `r2_std`.

Input Parameters:

```
{
  "max_energy_above_hull": 0.3,
  "max_per_composition": 3,
  "max_compositions": 40,
  "target_property": "formation_energy_per_atom",
  "feature_engineering": "advanced",
  "test_split": 0.2,
  "normalize": true,
  "n_estimators": 200,
  "max_depth": 8,
  "learning_rate": 0.1,
  "subsample": 0.8,
  "colsample_bytree": 0.8,
  "reg_alpha": 0.1,
  "reg_lambda": 0.1,
  "detailed_analysis": true,
  "cv_folds": 5
}
```

Subtask 1: Generate Oxide Polymorph Dataset

Generate a diverse list of oxide compositions for dataset creation. Retrieve polymorphs for all oxide compositions from Materials Project database. For each composition, collect multiple polymorphs including both stable and metastable structures. Consolidate all individual polymorph files into a single comprehensive dataset. Each entry should have `source_composition`.

Input Parameters:

```
{
  "max_energy_above_hull": 0.3,
  "max_per_composition": 3,
  "max_compositions": 40
}
```

Subtask 2: Prepare ML Dataset

Transform the consolidated dataset into ML-ready format with engineered features and proper train/test splits. Create a metadata JSON file with `features`, `train_path`, and `test_path`.

Input Parameters:

```
{
  "target_property": "formation_energy_per_atom",
  "feature_engineering": "advanced",
  "test_split": 0.2,
  "normalize": true,
  "output_path": "oxide_ml_dataset"
}
```

Subtask 3: Train XGBoost Model

Train an XGBoost regression model to predict formation energies of oxide polymorphs. Use the prepared dataset with optimized hyperparameters for oxide materials. Focus on achieving good generalization performance across different oxide families and structural types.

Input Parameters:

```
{
  "target_column": "formation_energy_per_atom",
  "hyperparameters": {
    "n_estimators": 200,
    "max_depth": 8,
    "learning_rate": 0.1,
    "subsample": 0.8,
    "colsample_bytree": 0.8,
    "reg_alpha": 0.1,
    "reg_lambda": 0.1
  }
}
```

Subtask 4: Evaluate Model

Evaluate the trained XGBoost model using test set and cross-validation metrics. Save the results as a JSON file with keys `test_set_evaluation` and `cross_validation_results`. The `test_set_evaluation` dictionary must contain `mae`, `rmse`, `r2`, and `feature_importance`. The `cross_validation_results` dictionary must contain `r2_mean` and `r2_std`.

Input Parameters:

```
{
  "target_column": "formation_energy_per_atom",
  "detailed_analysis": true,
  "cv_folds": 5
}
```

F Pass^k

Beyond the standard pass@5 metric, we also computed the pass^5 score (more details about the metrics are presented in Appendix A). The results, presented in Figure F.1 for the four environments, two agents, and two LLMs, reveal trends highly consistent with those of pass@5 (see Figure 3). Specifically, SPECTRA and MD perform notably poorly, while OPENCATALYST achieves near-perfect scores. A key observation, however, is the pronounced decline in performance for the ML environment under the pass^5 metric. Its scores decreased substantially from the pass@5 assessment, falling to a level comparable with those of the MD environment. This, we hypothesize, may be because of the distance between tasks being comparable to those of the SPECTRA environment. As for pass@5, no trends are observed among the different agents and LLMs that were evaluated.

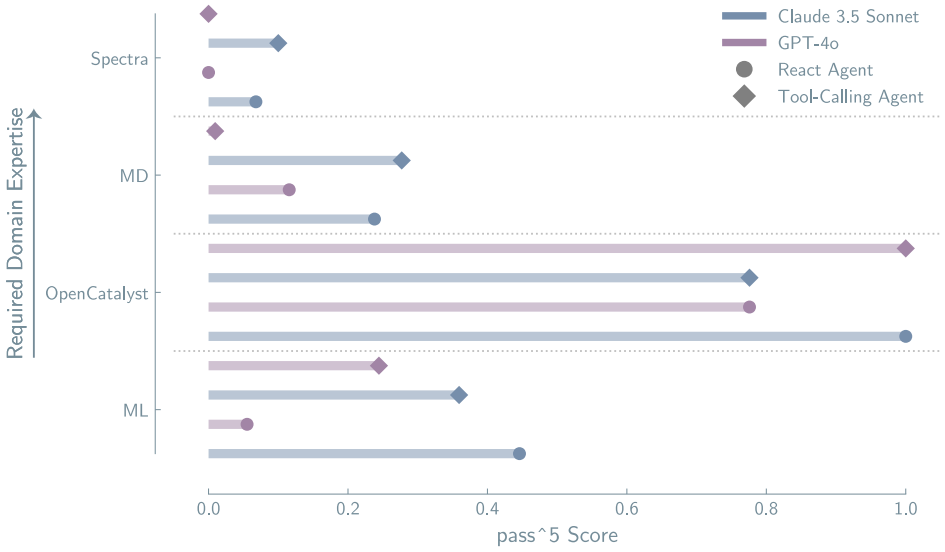

Figure F.1: **pass^5 scores across evaluated environments, agents, and LLMs.** Performance trends largely align with those of pass@5, with OPENCATALYST achieving near-perfect scores and SPECTRA and MD performing poorly. A pronounced deviation is observed for the ML environment, where a substantial performance decline under the pass^5 metric suggests agents struggle to solve its tasks consistently. This decline may be attributed to the increased inter-task distance, which appears comparable to that of the SPECTRA environment. No significant trends were identified across different agents or LLMs.

G Number of Tool Calls

The number of tool calls executed by the agents constitutes another metric captured in *Corral*. While straightforward to quantify, this metric offers valuable insights into an agent's performance and behavioral patterns. As illustrated in Figure G.2, the number of tool calls varies significantly across environments. An inverse relationship is observed between the frequency of tool use and both overall performance and the level of required domain expertise. Specifically, the SPECTRA and MD environments exhibit the highest volume of tool calls. In contrast, the number of tool calls is considerably lower and comparable between the OPENCATALYST and ML environments. Furthermore, the tool-calling agent consistently employs a greater number of tool calls than its ReAct counterpart. This discrepancy is anticipated, as the tool-calling agent's strategy depends more heavily on tool sampling and less on explicit reasoning chains.

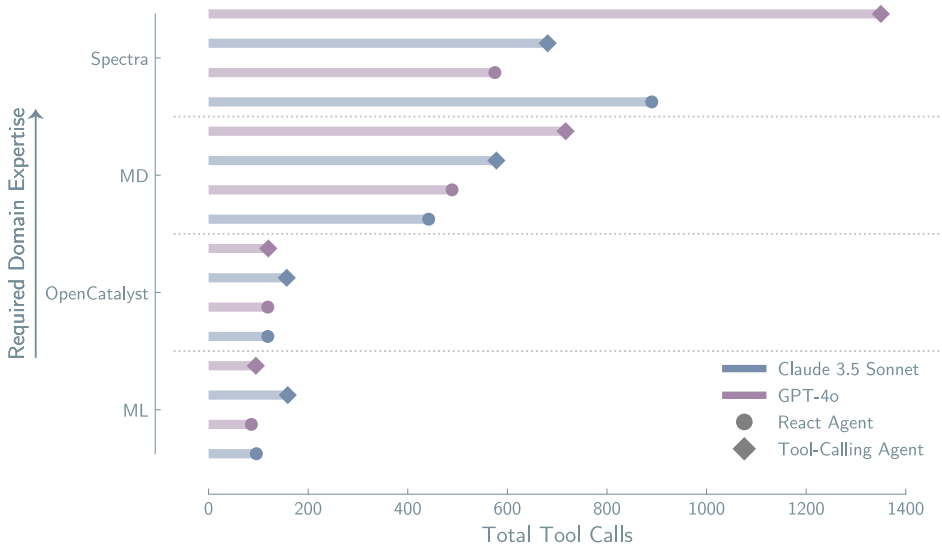

Figure G.2: **Total tool calls per environment, sorted by required domain expertise**. Environments demanding less expertise (OPENCATALYST, ML) exhibit fewer tool calls, while those requiring more (SPECTRA, MD) prompt significantly higher usage. The tool-calling agent consistently employs more calls than the ReAct agent across all environments.

G.1 Performance and Number of Tools

The observed trend suggests a potential inverse relationship between the number of tool calls and pass@5 scores, a comparison visually supported by Figure G.3. This figure indicates that high performance on the OPENCATALYST and ML environments is associated with a low number of tool calls. Conversely, lower scores on the MD and, more markedly, on the SPECTRA environments coincide with a substantially higher frequency of tool calls. One might hypothesize that this trend is driven by an inherent requirement for more steps or a greater number of available tools in certain environments. However, this explanation is not supported by the data. The number of steps required for SPECTRA, MD, and OPENCATALYST is comparable, particularly as both MD and OPENCATALYST involve numerous file management operations absent in SPECTRA. Similarly, the possibility that a larger toolkit acts as a distractor is countered by the fact that the number of tools available in SPECTRA is comparable to that in ML, as detailed in Table 1.

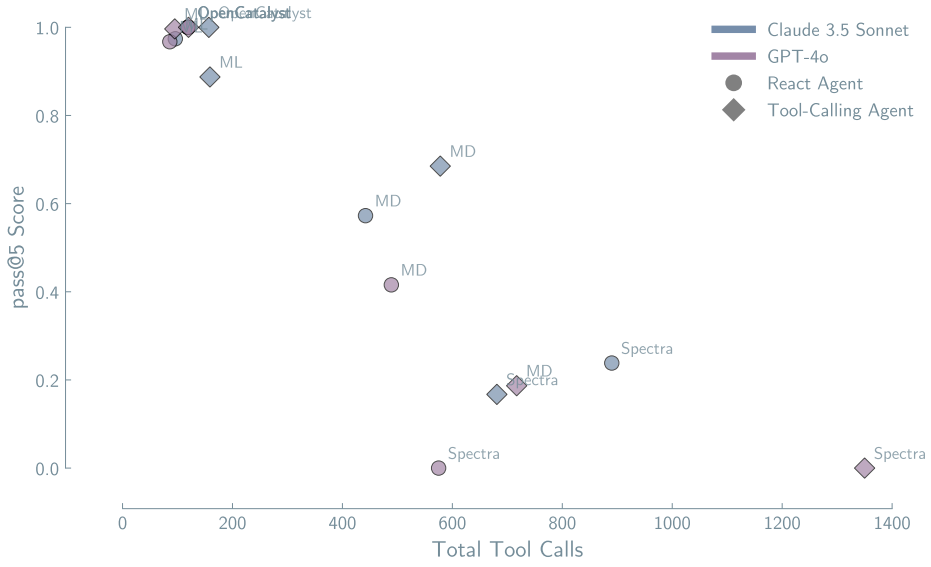

Figure G.3: **Illustration of the inverse trend between pass@5 scores and tool call frequency**. The high scores in OpenCatalyst and ML correspond to low tool usage, while the low scores in MD and Spectra correspond to high, inefficient tool usage. This divergence occurs despite comparable procedural steps and toolset sizes, countering hypotheses that attribute performance solely to environmental complexity.

H Number of Completion Tokens

Although the number of tool calls demonstrates a strong correlation with model performance, the consumption of completion tokens does not exhibit a similar trend, as illustrated in Figure H.4. The count of completion tokens remains largely consistent across the majority of environments and agents. A prominent exception is the `MD` environment, where the ReAct agent implemented with `Claude-3.5 Sonnet` generates a substantially higher volume of tokens. Apart from this outlier, the `ML` environment is the only one that consistently results in a lower token count relative to the others. Regarding agent performance, it is noteworthy that the ReAct agent does not systematically produce more tokens despite its requirement to articulate its reasoning through explicit "thoughts".

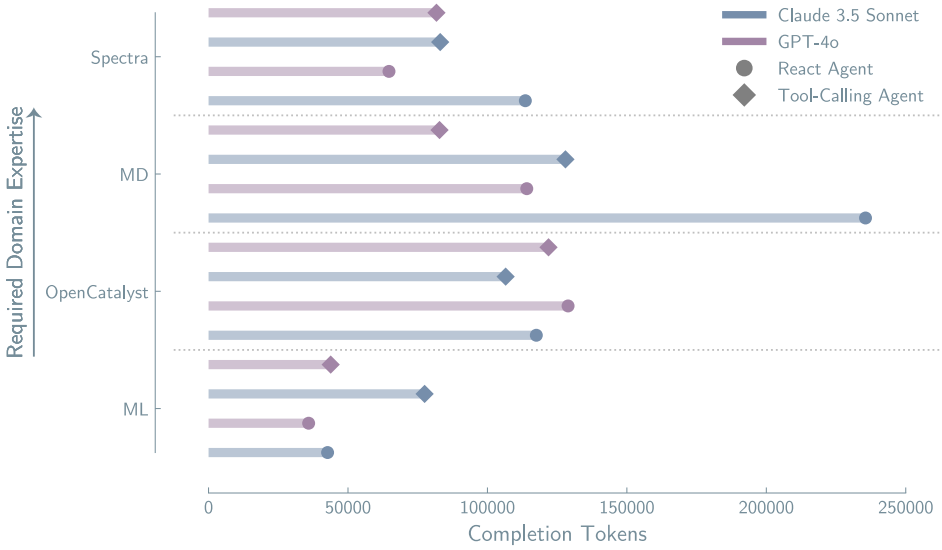

Figure H.4: **Number of completion tokens across environments sorted by the domain expertise required to solve them**. The figure illustrates that the number of tokens is similar across environments and agents. Only the `ML` environment seems to yield a lower number of completion tokens.

I Tool verbosity ablation results

Based on the initial results obtained across the environments, we investigated whether the verbosity level employed in the tool descriptions provided to the agent could significantly influence agent performance. As shown in Table I.11, no clear trend was observed across the different verbosity levels, with the performance trends across environments remaining largely consistent. The most notable performance difference occurred in the MD environment at the COMPREHENSIVE verbosity level when compared to the other two description levels. This difference, however, was less pronounced for the chained tasks.

Table I.11: **Ablation Study on Tool Description Verbosity**. This table presents the results of varying the level of detail in tool descriptions provided to the agents (brief, workflow, and comprehensive). The text in bold shows the best scores across environments. In the simpler OPENCATALYST and ML environments, performance is consistently high regardless of verbosity. However, in the more complex MD and SPECTRA domains, providing more context through workflow or comprehensive descriptions generally improves scores, indicating that richer information about the action space is crucial for the agent's policy to succeed in challenging tasks.

Agent	Verbosity	MD		ML		OPENCATALYST		SPECTRA	
		single	chained	single	chained	single	chained	single	chained
Claude 3.5 Sonnet									
ReAct	brief	0.22	0.5	0.89	0.96	**1.0**	**1.0**	0.19	0.47
	workflow	0.21	0.51	0.56	0.91	**1.0**	**1.0**	0.18	0.43
	comprehensive	0.57	0.69	0.97	0.97	**1.0**	**1.0**	**0.24**	0.44
Tool-Calling	brief	0.52	**0.78**	0.99	0.91	**1.0**	0.66	0.05	**0.49**
	workflow	**0.81**	0.74	**1.0**	0.99	**1.0**	**1.0**	0.1	0.46
	comprehensive	0.69	0.65	0.89	**1.0**	**1.0**	**1.0**	0.17	0.3
GPT-4o									
ReAct	brief	0.2	0.72	0.33	0.99	0.89	**1.0**	0.05	0.44
	workflow	0.51	0.71	0.89	**1.0**	**1.0**	**1.0**	0.0	0.38
	comprehensive	0.42	0.72	0.97	**1.0**	**1.0**	**1.0**	0.0	0.36
Tool-Calling	brief	0.2	0.69	0.33	0.77	**1.0**	0.75	0.05	0.41
	workflow	0.07	0.77	**1.0**	0.97	**1.0**	**1.0**	0.0	0.39
	comprehensive	0.19	0.73	**1.0**	**1.0**	**1.0**	**1.0**	0.0	0.41

J Embeddings distance ablation

For studying the consistently poor performance of agents in the SPECTRA environment, we decided to analyze the cosine distance of tasks and tools from that environment. We used the embeddings obtained through the API of the closed-source model `text-embedding-3-large`.

For obtaining the distances reported in Table 1, each task and tool description is passed through the model to generate its respective embedding. These embeddings are then used to calculate pairwise distances between tools, tasks, and the combinations of tools and tasks. The distance metric used for comparison is the cosine distance, which is calculated as follows:

$$D_{\text{cosine}}(A, B) = 1 - \frac{A \cdot B}{\|A\|_2 \|B\|_2}$$

where:

- A and B are the vectors (embeddings) representing the two entities (tasks or tools),
- $\|A\|_2$ and $\|B\|_2$ are the L2 norms (magnitudes) of the vectors,
- $A \cdot B$ is the dot product between the vectors.

K Overall Results

Table K.12: Performance comparison of agents across SPECTRA, ML and OPENCATALYST tasks with varying tool verbosity.

Task/ Env	Task Level	Tool Verbosity	Agent	Model	Scores			Tool Exec time (s)	Total tokens	Tool calls	
					Overall	pass@5	pass^5			Total	Failed
SPECTRA	Task	brief	Tool-calling	Claude 3.5	0.050	0.050	0.050	2517.794	1999844	316	0
			Tool-calling	GPT-4o	0.020	0.046	0.001	4954.448	1066725	644	0
			ReAct	Claude 3.5	0.140	0.192	0.101	2761.069	1075978	487	0
			ReAct	GPT-4o	0.020	0.046	0.001	4669.816	2982094	855	0
		comprehensive	Tool-calling	Claude 3.5	0.120	0.167	0.100	3321.458	9600785	681	0
			Tool-calling	GPT-4o	0	0.000	0.000	3986.564	7067824	1350	0
			ReAct	Claude 3.5	0.150	0.238	0.068	4672.958	11309758	890	0
			ReAct	GPT-4o	0	0.000	0.000	3372.496	8579030	575	0
		workflow	Tool-calling	Claude 3.5	0.100	0.100	0.100	2909.819	6487221	680	0
			Tool-calling	GPT-4o	0	0.000	0.000	5425.582	4845761	1284	0
			ReAct	Claude 3.5	0.140	0.183	0.104	3491.533	7418732	841	0
			ReAct	GPT-4o	0	0.000	0.000	3555.800	5418950	632	1
	Subtask	brief	Tool-calling	Claude 3.5	0.447	0.492	0.407	14603.170	3052471	1591	0
			Tool-calling	GPT-4o	0.343	0.415	0.286	25426.708	3711126	1797	0
			ReAct	Claude 3.5	0.416	0.467	0.372	12952.609	1845878	1221	0
			ReAct	GPT-4o	0.366	0.436	0.303	13407.503	2414056	1510	0
		comprehensive	Tool-calling	Claude 3.5	0.281	0.304	0.257	2512.472	7929146	1753	20
			Tool-calling	GPT-4o	0.344	0.407	0.294	13817.494	7819026	1857	0
			ReAct	Claude 3.5	0.410	0.436	0.377	11667.673	7265056	1480	48
			ReAct	GPT-4o	0.332	0.365	0.290	6221.911	6186909	1329	10
		workflow	Tool-calling	Claude 3.5	0.417	0.460	0.370	2625.681	7404750	1740	16
			Tool-calling	GPT-4o	0.333	0.388	0.299	14822.657	5269491	1658	0
			ReAct	Claude 3.5	0.388	0.429	0.351	11366.324	5174901	1406	90
			ReAct	GPT-4o	0.331	0.384	0.276	12005.337	7266929	1282	38
ML	Task	brief	Tool-calling	Claude 3.5	0.667	0.993	0.161	1220.296	1845649	147	9
			Tool-calling	GPT-4o	0.267	0.333	0.109	1266.829	1340868	137	6
			ReAct	Claude 3.5	0.533	0.887	0.135	1034.641	933964	116	14
			ReAct	GPT-4o	0.267	0.333	0.109	3337.430	4463267	128	2
		comprehensive	Tool-calling	Claude 3.5	0.600	0.887	0.359	1859.059	5304033	159	7
			Tool-calling	GPT-4o	0.733	0.996	0.244	1176.306	1968853	95	0
			ReAct	Claude 3.5	0.733	0.974	0.446	1093.865	2211623	96	13
			ReAct	GPT-4o	0.533	0.967	0.055	1097.562	1848462	86	0
		workflow	Tool-calling	Claude 3.5	1	1.000	1.000	1425.824	2997344	132	0
			Tool-calling	GPT-4o	0.933	1.000	0.776	1189.020	1600247	103	0
			ReAct	Claude 3.5	0.400	0.557	0.333	1048.410	1731643	99	14
			ReAct	GPT-4o	0.733	0.891	0.667	823.038	1375619	94	4
	Subtask	brief	Tool-calling	Claude 3.5	0.783	0.909	0.589	1024.336	1444605.667	94.333	8.333
			Tool-calling	GPT-4o	0.417	0.772	0.072	2919.916	1283053.667	126.667	12.000
			ReAct	Claude 3.5	0.767	0.959	0.563	628.312	352240.333	76.667	6.333
			ReAct	GPT-4o	0.950	0.994	0.918	572.154	218174.667	49.667	0.000
		comprehensive	Tool-calling	Claude 3.5	0.950	1.000	0.832	602.452	697221.667	65.333	0.000
			Tool-calling	GPT-4o	0.950	0.999	0.867	455.008	511905.000	56.000	0.000
			ReAct	Claude 3.5	0.833	0.966	0.666	604.539	525621.000	62.333	7.000
			ReAct	GPT-4o	0.917	0.999	0.755	687.699	408186.333	49.333	1.667
		workflow	Tool-calling	Claude 3.5	0.933	0.993	0.861	741.300	730223.000	65.000	0.000
			Tool-calling	GPT-4o	0.883	0.966	0.834	414.070	364710.333	54.667	0.000
			ReAct	Claude 3.5	0.833	0.915	0.707	670.169	473178.000	68.333	8.000
			ReAct	GPT-4o	0.967	0.999	0.923	611.969	321803.333	48.333	2.333
OPENCATALYST	Task	brief	Tool-calling	Claude 3.5	0.933	1.000	0.776	2037.189	2711823	188	32
			Tool-calling	GPT-4o	1	1.000	1.000	2130.001	1953905	144	24
			ReAct	Claude 3.5	0.733	0.891	0.667	2290.416	1538987	114	2
			ReAct	GPT-4o	1	1.000	1.000	1702.736	1786819	125	0
		comprehensive	Tool-calling	Claude 3.5	0.933	1.000	0.776	1763.970	3269313	157	37
			Tool-calling	GPT-4o	1	1.000	1.000	2054.796	2314210	120	0
			ReAct	Claude 3.5	1	1.000	1.000	1705.277	2586206	119	0
			ReAct	GPT-4o	0.933	1.000	0.776	2642.857	2563808	119	0
		workflow	Tool-calling	Claude 3.5	1	1.000	1.000	1744.650	2646049	154	26
			Tool-calling	GPT-4o	1	1.000	1.000	2102.629	2357448	153	30
			ReAct	Claude 3.5	1	1.000	1.000	1762.177	2332113	125	0
			ReAct	GPT-4o	0.933	1.000	0.776	2347.130	2064826	160	0
	Subtask	brief	Tool-calling	Claude 3.5	0.552	0.655	0.449	1292.663	793577	111	23
			Tool-calling	GPT-4o	0.714	0.746	0.682	1206.510	851828	140	39
			ReAct	Claude 3.5	0.971	0.999	0.924	810.621	455240	113	10
			ReAct	GPT-4o	0.943	0.996	0.877	1254.770	574773	103	8
		comprehensive	Tool-calling	Claude 3.5	1.000	1.000	1.000	1350.958	981333	126	10
			Tool-calling	GPT-4o	1.000	1.000	1.000	1146.016	692674	109	4
			ReAct	Claude 3.5	1.000	1.000	1.000	1109.015	806821	116	1
			ReAct	GPT-4o	1.000	1.000	1.000	1203.005	627433	107	0
		workflow	Tool-calling	Claude 3.5	0.895	0.999	0.668	1371.961	888996	121	11
			Tool-calling	GPT-4o	1.000	1.000	1.000	1135.358	617183	109	4
			ReAct	Claude 3.5	1.000	1.000	1.000	1086.880	699299	115	0
			ReAct	GPT-4o	1.000	1.000	1.000	1066.127	545052	98	1

Table K.13: Performance comparison of agents across MD tasks with varying tool verbosity

Task/ Env	Task Level	Tool Verbosity	Agent	Model	Scores			Tool Exec time (s)	Total tokens	Tool calls	
					Overall	pass@5	pass^5			Total	Failed
Surface Energy	Task	brief	Tool-calling	Claude 3.5	0.20	0.57	0.00272	3994.60	2372658	243	46
			Tool-calling	GPT-4o	0.00	0.00	0.00	2668.31	2761722	334	53
			ReAct	Claude 3.5	0.40	0.65	0.25	1150.81	740500	132	10
			ReAct	GPT-4o	0.00	0.00	0.00	1937.86	1711175	348	92
		comprehensive	Tool-calling	Claude 3.5	0.30	0.42	0.25	3436.00	4010181	284	52
			Tool-calling	GPT-4o	0.10	0.23	0.0025	2576.81	2756626	376	55
			ReAct	Claude 3.5	0.60	0.75	0.35	2633.79	2173989	204	0
			ReAct	GPT-4o	0.15	0.25	0.02	1872.54	2686373	254	46
		workflow	Tool-calling	Claude 3.5	0.65	0.75	0.41	3143.25	1669994	234	3
			Tool-calling	GPT-4o	0.00	0.00	0.00	2579.06	2608477	346	67
			ReAct	Claude 3.5	0.00	0.00	0.00	2796.24	1455136	91	4
			ReAct	GPT-4o	0.10	0.34	0.0001	1732.40	2116582	309	69
	Subtask	brief	Tool-calling	Claude 3.5	0.86	0.95	0.70	3675.40	2392176	412	21
			Tool-calling	GPT-4o	0.53	0.66	0.42	16050.29	8819312	817	114
			ReAct	Claude 3.5	0.33	0.48	0.21	2707.52	2015689	320	33
			ReAct	GPT-4o	0.55	0.66	0.47	4862.63	3533316	697	131
		comprehensive	Tool-calling	Claude 3.5	0.86	0.95	0.69	3174.83	2310994	327	7
			Tool-calling	GPT-4o	0.56	0.71	0.42	16846.90	7371726	871	155
			ReAct	Claude 3.5	0.61	0.72	0.50	3100.14	2213461	353	26
			ReAct	GPT-4o	0.54	0.70	0.40	5057.13	5066130	702	143
		workflow	Tool-calling	Claude 3.5	0.78	0.94	0.54	3439.93	2623439	394	22
			Tool-calling	GPT-4o	0.55	0.73	0.45	7761.7	7628950	883	154
			ReAct	Claude 3.5	0.60	0.68	0.52	3228.89	2045466	367	27
			ReAct	GPT-4o	0.56	0.68	0.46	4750.14	4227465	682	121
Melting	Task	brief	Tool-calling	Claude 3.5	0.13	0.45	0.0002	2012.93	1843125	138	44
			Tool-calling	GPT-4o	0.00	0.00	0.00	2015.09	1525141	241	57
			ReAct	Claude 3.5	0.0	0.0	0.0	265.25	35472	0	0
			ReAct	GPT-4o	0.00	0.00	0.00	4501.84	2364025	208	35
		comprehensive	Tool-calling	Claude 3.5	0.33	0.64	0.03	1975.85	2215060	170	34
			Tool-calling	GPT-4o	0.20	0.33	0.026	1171.90	1532063	184	7
			ReAct	Claude 3.5	0.67	0.97	0.36	2479.20	1956353	191	24
			ReAct	GPT-4o	0.00	0.00	0.00	1522.47	1554510	131	8
		workflow	Tool-calling	Claude 3.5	0.33	0.78	0.03	1962.91	1398117	139	13
			Tool-calling	GPT-4o	0.00	0.00	0.00	1132.12	902589	169	8
			ReAct	Claude 3.5	0.47	0.64	0.34	1387.85	622430	80	2
			ReAct	GPT-4o	0.07	0.22	0.0001	971.58	765381	132	2
	Subtask	brief	Tool-calling	Claude 3.5	0.55	0.67	0.44	3243.99	3243724	364	69
			Tool-calling	GPT-4o	0.63	0.78	0.45	21719.57	19734289	583	21
			ReAct	Claude 3.5	0.41	0.64	0.22	3795.09	2994944	380	62
			ReAct	GPT-4o	0.53	0.68	0.41	3598.93	2343615	337	3
		comprehensive	Tool-calling	Claude 3.5	0.51	0.57	0.47	3160.90	4366885	348	84
			Tool-calling	GPT-4o	0.57	0.80	0.41	27788.65	7650030	561	44
			ReAct	Claude 3.5	0.80	0.95	0.69	5811.09	5376726	495	76
			ReAct	GPT-4o	0.53	0.64	0.42	3352.13	3057493	313	3
		workflow	Tool-calling	Claude 3.5	0.52	0.74	0.35	3077.21	3278175	348	58
			Tool-calling	GPT-4o	0.64	0.83	0.44	29058.55	4814488	467	18
			ReAct	Claude 3.5	0.48	0.61	0.40	4047.19	2874521	405	64
			ReAct	GPT-4o	0.60	0.69	0.51	3486.44	1861511	315	1
Quenching	Task	brief	Tool-calling	Claude 3.5	0.40	0.56	0.33	2715.42	4212990	204	58
			Tool-calling	GPT-4o	0.27	0.61	0.007	1633.41	2357544	195	27
			ReAct	Claude 3.5	0.0	0.0	0.0	3276.82	650701	30.0	0.0
			ReAct	GPT-4o	0.27	0.61	0.007	3045.26	5253679	198	50
		comprehensive	Tool-calling	Claude 3.5	0.87	0.99	0.55	1575.07	1252706	124	5
			Tool-calling	GPT-4o	0.00	0.00	0.00	1237.00	2489890	157	6
			ReAct	Claude 3.5	0.00	0.00	0.00	3123.35	1933814	47	0
			ReAct	GPT-4o	0.80	0.99	0.33	1188.14	1236142	104	1
		workflow	Tool-calling	Claude 3.5	0.67	0.89	0.44	1606.94	900033	120	3
			Tool-calling	GPT-4o	0.07	0.22	0.0001	1099.34	1549820	150	7
			ReAct	Claude 3.5	0.00	0.00	0.00	3081.22	1279999	47	0
			ReAct	GPT-4o	0.53	0.97	0.06	1817.45	1329878	95	2
	Subtask	brief	Tool-calling	Claude 3.5	0.50	0.71	0.28	4098.50	4427260	362	79
			Tool-calling	GPT-4o	0.38	0.63	0.25	12957.03	15868903	629	40
			ReAct	Claude 3.5	0.28	0.36	0.25	3893.25	6921097	349	62
			ReAct	GPT-4o	0.52	0.82	0.27	3220.80	1996441	334	6
		comprehensive	Tool-calling	Claude 3.5	0.30	0.42	0.25	4638.77	6404970	342	81
			Tool-calling	GPT-4o	0.48	0.69	0.29	10539.12	9541567	566	27
			ReAct	Claude 3.5	0.30	0.42	0.25	6541.34	6996892	511	111
			ReAct	GPT-4o	0.53	0.84	0.29	3328.19	2892921	334	9
		workflow	Tool-calling	Claude 3.5	0.35	0.54	0.20	682.25	4714626	361	85
			Tool-calling	GPT-4o	0.48	0.74	0.28	13034.44	10529810	580	12
			ReAct	Claude 3.5	0.25	0.25	0.25	670.01	8800	97	7
			ReAct	GPT-4o	0.45	0.0.77	0.26	4450.11	5727793	455	102

References

[1] Shunyu Yao et al. *τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains*. 2024. arXiv: 2406.12045 [cs.AI]. URL: https://arxiv.org/abs/2406.12045.

[2] Daan Frenkel and Berend Smit. *Understanding molecular simulation: from algorithms to applications*. Elsevier, 2023.

[3] A. P. Thompson et al. "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales". In: *Comp. Phys. Comm.* 271 (2022), p. 108171. DOI: 10.1016/j.cpc.2021.108171.

[4] Matthew K. Horton et al. "Accelerated data-driven materials science with the Materials Project". In: *Nature Materials* (July 2025). ISSN: 1476-4660. DOI: 10.1038/s41563-025-02272-0. URL: http://dx.doi.org/10.1038/s41563-025-02272-0.

[5] Lowik Chanussot et al. "Open catalyst 2020 (OC20) dataset and community challenges". In: *Acs Catalysis* 11.10 (2021), pp. 6059–6072.

[6] Tianqi Chen and Carlos Guestrin. "Xgboost: A scalable tree boosting system". In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 2016, pp. 785–794.

[7] Kevin Maik Jablonka, Luc Patiny, and Berend Smit. "Making Molecules Vibrate: Interactive Web Environment for the Teaching of Infrared Spectroscopy". In: *Journal of Chemical Education* 99.2 (Jan. 2022), pp. 561–569. ISSN: 1938-1328. DOI: 10.1021/acs.jchemed.1c01101. URL: http://dx.doi.org/10.1021/acs.jchemed.1c01101.

[8] Damiano Banfi and Luc Patiny. "www.nmrdb.org: Resurrecting and Processing NMR Spectra On-line". In: *CHIMIA* 62.4 (Apr. 2008), p. 280. ISSN: 0009-4293. DOI: 10.2533/chimia.2008.280. URL: http://dx.doi.org/10.2533/chimia.2008.280.

[9] Andrés M. Castillo, Luc Patiny, and Julien Wist. "Fast and accurate algorithm for the simulation of NMR spectra of large spin systems". In: *Journal of Magnetic Resonance* 209.2 (Apr. 2011), pp. 123–130. ISSN: 1090-7807. DOI: 10.1016/j.jmr.2010.12.008. URL: http://dx.doi.org/10.1016/j.jmr.2010.12.008.

[10] João Aires-de-Sousa, Markus C. Hemmer, and Johann Gasteiger. "Prediction of 1H NMR Chemical Shifts Using Neural Networks". In: *Analytical Chemistry* 74.1 (Dec. 2001), pp. 80–90. ISSN: 1520-6882. DOI: 10.1021/ac010737m. URL: http://dx.doi.org/10.1021/ac010737m.

[11] Inc. Modal Labs. *Modal Python Package*. Version 1.0. Accessed: 2025-08-21. 2025. URL: https://modal.com/docs/guide.

[12] Shyue Ping Ong et al. "Python Materials Genomics (pymatgen): A robust, open-source python library for materials analysis". In: *Computational Materials Science* 68 (Feb. 2013), pp. 314–319. ISSN: 0927-0256. DOI: 10.1016/j.commatsci.2012.10.028. URL: http://dx.doi.org/10.1016/j.commatsci.2012.10.028.