Data/Code

LARA-HPC: Validation-Driven Agentic Supercomputer Workflows for Atomistic Modeling

William Dawson [1], Louis Beal [2], Yoann Curé [3], Giuseppe Fisicaro [4,*], Dorian Rolland [3] and Luigi Genovese [3]

[1] RIKEN Center for Computational Science, Kobe, Hyogo 650-0047, Japan

[2] Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

[3] University Grenoble Alpes, MEM, L_Sim, F-38000 Grenoble, France

[4] Istituto di Microelettronica e Microsistemi, Consiglio Nazionale delle Ricerche, Z.I. VIII Strada 5, I-95121 Catania, Italy

(Received 29 April 2026; published 10 November 2026)

Large language models and agentic systems have recently demonstrated potential for automating scientific workflows, including atomistic simulations. However, their deployment in high-performance computing (HPC) environments remains limited by the lack of mechanisms ensuring correctness, reproducibility, and safe interaction with computational resources. Generated workflows suffer from inconsistencies, incorrect application programming interface usage, or invalid physical configurations—leading to failed or unreliable simulations. In this work, we introduce LARA-HPC (Lara Agentic Research Assistant for HPC), a validation-driven agentic framework to enable reliable workflow generation for atomistic modeling on HPC systems. Our approach is based on three key components: (1) a controlled execution layer that mediates all interactions with HPC resources; (2) simulation-native validation through dry-run capabilities, enabling execution-level verification without incurring resource cost; and (3) a multiphase agentic pipeline combining retrieval-augmented generation and iterative refinement. We demonstrate the effectiveness of this approach by performing an end-to-end atomistic simulation workflow on HPC by applying LARA-HPC to density functional theory simulations. The results show that validation-driven generation significantly improves robustness and enables iterative correction of both syntactic and physical inconsistencies. More broadly, this work advocates for a shift from generation-first to validation-first paradigms in artificial-intelligence-assisted scientific computing. We argue that the future task of the computational physics community is to develop domain-specific agentic systems based on structured tooling to realize an HPC enabled copiloted research ecosystem.

DOI: 10.1103/cp6v-t93q

I. INTRODUCTION

Recent advances in large language models (LLMs) have significantly expanded the capabilities of automated code generation, enabling the construction of increasingly complex workflows from natural-language descriptions. These models have demonstrated strong capabilities in reasoning over structured tasks, leading to the development of artificial intelligence (AI) copilots for software engineering [1,2]. In scientific domains such as computational chemistry and materials science, these developments open new opportunities for automating simulation pipelines and accelerating research workflows [3–5].

In particular, LLM-based copilots and agentic systems (Fig. 1) have demonstrated promising results in assisting scientific tasks, including experimental automation and atomistic simulations [6–12]. Several frameworks have been proposed to automate molecular dynamics workflows or computational chemistry pipelines [4,13,14]. Agentic systems like DREAMS (DFT-based Research Engine for Agentic Materials Simulations) aim at automating high-fidelity density functional theory (DFT) simulations through coordinated LLM agents, enabling autonomous, expert-level materials discovery and systematic error handling [15]. Despite these advances, the deployment of such systems in high-performance computing (HPC) environments remains challenging. Scientific software stacks are often developed within specialized communities, evolve rapidly, and lack stable or fully documented interfaces.

Scientific simulations in HPC environments involve multiple stages, including input preparation, job submission, execution, and postprocessing. While workflow management systems have been developed to automate these processes and ensure reproducibility [16,17], integrating AI-driven systems into these environments introduces additional challenges, such as the need for controlled access to resources and compatibility with scheduling systems. LLM-generated workflows frequently suffer from inconsistencies, incorrect application programming interface (API) usage, or incomplete configurations. In HPC contexts, these issues are critical as erroneous inputs lead to wasted computational resources, failed jobs, or scientifically invalid results. It is not sufficient to use a system with merely a high success rate (e.g., 95%), as erroneous runs can have substantial consequences. HPC systems

*Contact author: giuseppe.fisicaro@cnr.it

Published by the American Physical Society under the terms of the Creative Commons Attribution 4.0 International license. Further distribution of this work must maintain attribution to the author(s) and the published article's title, journal citation, and DOI.

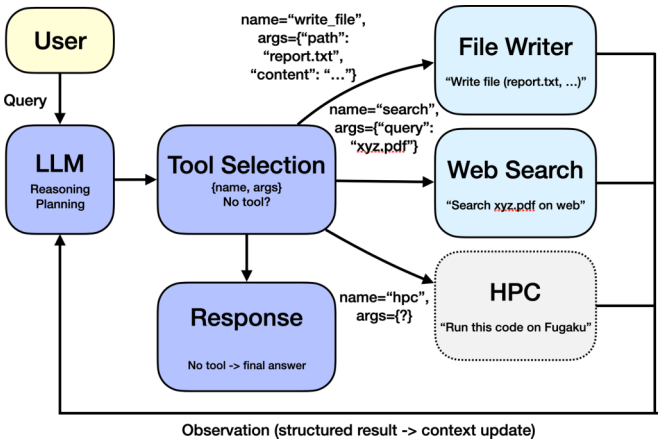


FIG. 1. ReAct (Reasoning and Acting) loop [21] at the foundation of agentic AI. The inner reasoning monologue of the LLM produces text that corresponds with specific tool calls. These tool calls perform actions, which result in structured observations being injected into the LLM's context for further reasoning. The missing piece addressed by this work is the integration of HPC.

also often have high latency for job execution; hence, agentic systems cannot achieve productivity using a trial-and-error-driven strategy, even if that is sufficient on a local workstation. In addition, the substantially number of steps to handle is also a source of further difficulty in understanding the process, not only when something goes wrong, but also in terms of reproducibility of the results.

The difficulty is not limited to software engineering. Consider the context of electronic-structure calculations of materials based on the widely used Kohn-Sham density functional theory [18]. In such studies, a scientifically meaningful result depends on a coherent chain of modeling decisions: How the system is built, the approximations used for electron interactions (choice of exchange-correlation functional), the treatment of atomic cores and spin, the numerical accuracy settings, and the analysis of the results must all be compatible with one another [19]. A workflow that is syntactically correct may still be scientifically wrong. This makes atomistic simulation particularly demanding for agentic AI—correctness must be established at the level of the computational method, not only at the level of code generation.

Existing approaches primarily focus on improving generation quality through prompt engineering, retrieval-augmented generation (RAG), or iterative interaction loops [13,14]. Among other techniques, RAG has emerged for improving reliability by grounding generation in external knowledge sources [20]. In scientific applications, RAG can leverage curated datasets such as training materials, documentation, and previous workflows to reduce hallucinations and improve consistency. While these strategies improve the relevance of the generated code, they do not provide guarantees of correctness prior to execution. In particular, they cannot capture domain-specific dynamic constraints related to simulation setup, resource allocation, or domain-specific validity.

This issue becomes especially visible when one moves beyond a fixed tool-calling scenario. A narrowly defined tool interface may be sufficient for a predetermined task, but realistic scientific exploration quickly requires methodological branching. For instance, during a DFT study, one may need to rethink the choice of model system, change the approximations used for the description of atoms' core electrons and nuclei (pseudopotential families), compare open- and closed-shell references, perform convergence studies, or introduce a new postprocessing step in response to intermediate results. Such adaptations are routine for expert practitioners, but are difficult to encode in advance through static tools.

In this work, we introduce LARA-HPC (Lara Agentic Research Assistant for HPC), a validation-driven agentic framework designed to bridge this gap. The key idea is to shift from a *generation-first* paradigm to a *validation-first* approach ("Look Before You Leap"). This is achieved by combining (1) a controlled HPC execution layer based on our Python `remotemanager` package to address operational and scheduling constraints; (2) simulation-native validation utilizing a "dry-run" primitive; and (3) a multiphase agentic pipeline allowing for workflow generation and iterative refinement.

The dry-run primitive refers to running a computer program in a way that the code only validates the input and provides information about the process to be executed (an example of this feature is in the Linux utility `rsync`). Unlike static analysis, dry-run evaluation leverages the internal logic of the simulation code to detect errors in input configuration, resource requirements, or physical consistency, without the cost of a full calculation. This enables iterative refinement in which workflows are progressively corrected until they satisfy all validation criteria and are ready for safe submission.

We showcase this approach by executing full atomistic simulation workflows on HPC, applying LARA-HPC to orchestrate DFT-based simulations with the BIGDFT program [22]. BigDFT is an *ab initio* electronic-structure code. It is built for extreme-scale HPC systems, using linear-scaling algorithms and efficient parallelization to simulate very large systems, making it relevant for exascale computing architectures. Two representative atomistic simulation tasks will be presented: (1) the calculation of the atomization enthalpy of a molecular system and (2) surface adsorption energies of a large system. Our experiments demonstrate how an agentic system incorporating dry-run validation and structured feedback can identify incorrect API usage, missing inputs, and inconsistent physical settings before production execution. We argue that such validation-driven paradigms are essential for ensuring correctness, reproducibility, and efficient use of computational resources in future copiloted research.

II. HPC ORCHESTRATION WITH `remotemanager` AND MODEL CONTEXT PROTOCOL

Before introducing LARA-HPC itself, it is important to describe the execution substrate on which the framework relies. In our view, the first challenge in connecting agentic systems to atomistic simulation campaigns is not code generation, but the controlled exposure of HPC resources. An AI system should not directly manipulate login shells,

schedulers, filesystem paths, or *ad hoc* environment configuration. Instead, these interactions should be mediated through a reproducible and auditable orchestration layer. For this purpose, we build on `remotemanager` [23], a lightweight Python middleware designed to promote local Python functions into remotely executable tasks on HPC resources. The key design principle of `remotemanager` is to preserve the standard user workflow of scientific computing—batch scripts, file staging, remote execution, and result retrieval—while expressing it through a programmatic interface. In this way, the library does not replace nor circumvent the logic of HPC usage, but formalizes it.

The core abstraction of `remotemanager` is that an ordinary Python function can be *promoted* into a remote procedure call enveloped in a user-defined execution environment (executables, libraries, data, etc.). The user first specifies a jobscript template adapted to the target machine. This template contains the scheduler directives and execution environment, but also exposes a controlled set of tunable parameters such as the number of nodes, MPI ranks per node, and OpenMP threads. Listing 1 illustrates this first step.

Listing 1. A jobscript template of supercomputer Fugaku (a petascale supercomputer at the RIKEN Center for Computational Science in Kobe, Japan) for use with `remotemanager`. Optional parameters such as the number of nodes, MPI ranks per node, and OpenMP threads are explicitly exposed. The jobscript defines the execution environment by loading modules, running containers, setting environment variables, etc. (in this case, through a conda virtual environment).

```
template = """#!/bin/bash -l
#PJM --mpi 'max-proc-per-node=#MPI#'
#PJM -L 'elapse=2:00:00'
#PJM -L 'node=#NODES#'
#PJM -L 'rscgrp=small'
#PJM -g ra000000

export OMP_NUM_THREADS=#OMP#

# Example of environment initialization on
the target resource
eval "$($DATADIR/bin/micromamba shell
hook --shell bash)" micromamba activate
remote"""
```

This template-based design is important for two reasons. First, it reflects how production HPC jobs are actually launched: The scheduler remains the entry point, rather than an opaque service layer. Second, it makes resource selection explicit and inspectable. The agent or higher-level workflow may choose values for exposed parameters, but it cannot arbitrarily redefine the structure of the batch script.

Once the template is defined, the user instantiates a `remotemanager` computer object, completes the machine-specific configuration, and promotes the Python function (Listing 2). The promoted function can then be invoked from the local Python session as if it were a normal function, while `remotemanager` transparently handles file transfer, job submission, remote execution, and result collection.

Listing 2. Configuring a remote machine and promoting a BigDFT single-point calculation of H_2O for remote execution on supercomputer Fugaku.

```
Fugaku = Computer(template=template,
              host="fugaku.r-ccs.riken.jp",
              submitter="pjsub")
Fugaku.nodes = 1
Fugaku.mpi = 1
Fugaku.omp = 48

@SanzuFunction(url=Fugaku)
def run_water_energy(hgrid=0.4,
dry_run=False):
  from BigDFT.Systems import System
  from BigDFT.Fragments import Fragment
  from BigDFT.Atoms import Atom
  from BigDFT.Inputfiles import Inputfile
  from BigDFT.Calculators import
  SystemCalculator

  sys = System()
  sys["WAT:0"] = Fragment([
    Atom({"O": [0.0, 0.0, 0.0], "units":
    "bohr"}),
    Atom({"H": [1.43, 1.11, 0.0], "units":
    "bohr"}),
    Atom({"H": [1.43, -1.11, 0.0], "units":
    "bohr"}),
  ])

  inp = Inputfile()
  inp.set_xc("PBE")
  inp.set_hgrid(hgrid)

  code = SystemCalculator(dry_run=dry_run)
  log = code.run(sys=sys, input=inp,
  name="water")
  return getattr(log, "energy", None)
run_water_energy(hgrid=0.35, dry_run=True)
```

Conceptually, the promoted function defines the *scientific payload*, while `remotemanager` supplies the *execution protocol*. This separation is particularly useful in atomistic simulations, where scientific code such as BIGDFT may evolve rapidly but the operational steps of HPC execution remain comparatively stable.

A. Operational role of `remotemanager`

At execution time, `remotemanager` carries out the sequence of actions that a human user would otherwise perform manually. These typically include

(1) materializing a batch script from the selected template;
(2) staging input files and Python sources to the remote resource;
(3) submitting the job to the scheduler;
(4) monitoring completion;
(5) retrieving output files, return values, and execution logs.

A major strength of this approach is that it does not require any daemon or privileged service on the remote machine. The remote side only needs the standard tools already available to HPC users, such as a shell environment, scheduler commands, and secure file transfer. This makes the approach portable across production systems and compatible with the security constraints of real HPC centers. The design also means that `remotemanager` is not tied to any particular job scheduling software. Equally important is the fact that `remotemanager` captures not only the nominal return value of the remote function, but also standard output, standard error, and execution artifacts. This is essential in scientific settings, where failed runs are often as informative as successful ones, and where debugging requires visibility into the exact job context.

B. Toward an HPC tool call

While `remotemanager` provides the execution substrate, it does not by itself define how an LLM or AI agent should access these capabilities. For that interface layer, we use the Model Context Protocol (MCP), which standardizes the exposure of structured tool calls to large language models. In a typical MCP server based on Python, a function is annotated and published as a tool. The function signature and docstring become part of the semantic interface presented to the model. When this mechanism is combined with `remotemanager`, an MCP tool call can be mapped onto a remote HPC submission. Listing 3 illustrates this composition.

Listing 3. Exposing the same H_2O single-point routine as an MCP tool while keeping the remote resource configuration explicit. Here, we use the FastMCP library (gofastmcp.com).

```
mcp = FastMCP("Example Server")

@mcp.tool()
def water_single_point(hgrid, dry_run, omp,
mpi, nodes):
  """
  Run a BigDFT single-point calculation for
  H2O.
  Args:
    hgrid: real-space grid spacing.
    dry_run: use BigDFT dry-run validation
    mode.
    omp: number of OpenMP threads to use.
    mpi: number of MPI ranks per node.
    nodes: number of nodes to use.
  """
  Fugaku.nodes = nodes; Fugaku.mpi = mpi;
  Fugaku.omp = omp
  return run_water_energy (hgrid=hgrid,
  dry_run=dry_run)
```

This functionality composition is the key bridge between agentic reasoning and production HPC campaigns. The LLM does not receive shell access, scheduler access, or arbitrary filesystem access. Instead, it sees a constrained set of tools with well-defined parameters. Each invocation is then converted into an auditable batch submission by `remotemanager`.

C. Strengths and limitations of the MCP approach

The MCP with `remotemanager` combination already solves a substantial part of the AI-HPC integration problem. It provides a secure and realistic path for exposing supercomputing resources to an external reasoning system without compromising the operational model of the target platform. However, this approach also reveals a structural limitation. Each exposed MCP tool corresponds to a predefined function and therefore to a predefined workflow pattern. This is suitable for repetitive or narrowly scoped tasks, but it becomes restrictive when the scientific workflow must adapt dynamically. One solution would be to provide an agentic framework with a broad range of built in tools [12]; however, even this may fail to cover the dynamic use cases pursued by expert simulation scientists.

This limitation may be stated more concretely through the example of molecular atomization energies. Computing an atomization energy is conceptually simple, but the workflow requires coordinated calculations for the molecule and its isolated constituent atoms, with a consistent choice of exchange-correlation functional, spin configuration, boundary conditions, numerical parameters, and postprocessing conventions. Small methodological changes, such as switching pseudopotential families or tightening convergence parameters, can alter the resulting quantity appreciably.

Within an MCP-based framework, one could expose a tool for building a molecular system, another for running a BIGDFT calculation, and a final one for aggregating energies. Such a setup is sufficient for a single preplanned workflow. But as soon as the scientific question evolves to include alternative pseudopotentials, open-shell atomic references, all-electron variants, or convergence scans over grid spacing and localization radii, the number of required tool variants grows rapidly. The agent can only choose among the interfaces that were explicitly designed in advance; it cannot introduce a new methodological branch on its own.

For this reason, we regard the MCP plus `remotemanager` layer not as a complete solution, but as the indispensable execution backbone on top of which a more flexible system can be built. In the next section, we introduce LARA-HPC, which retains the controlled HPC interaction offered by `remotemanager` while replacing static workflow definitions by validated, retrieval-guided code generation.

III. LARA-HPC ARCHITECTURE

The central limitation of the HPC tools approach, as discussed in the previous section, is its reliance on predefined actions. While this ensures safety and reproducibility, it restricts the ability of AI agents to construct workflows dynamically. In practice, simulation-based research often requires the composing of new steps and logic to encompass an expanding set of hypotheses and scientific questions. This motivates the introduction of a higher-level reasoning layer

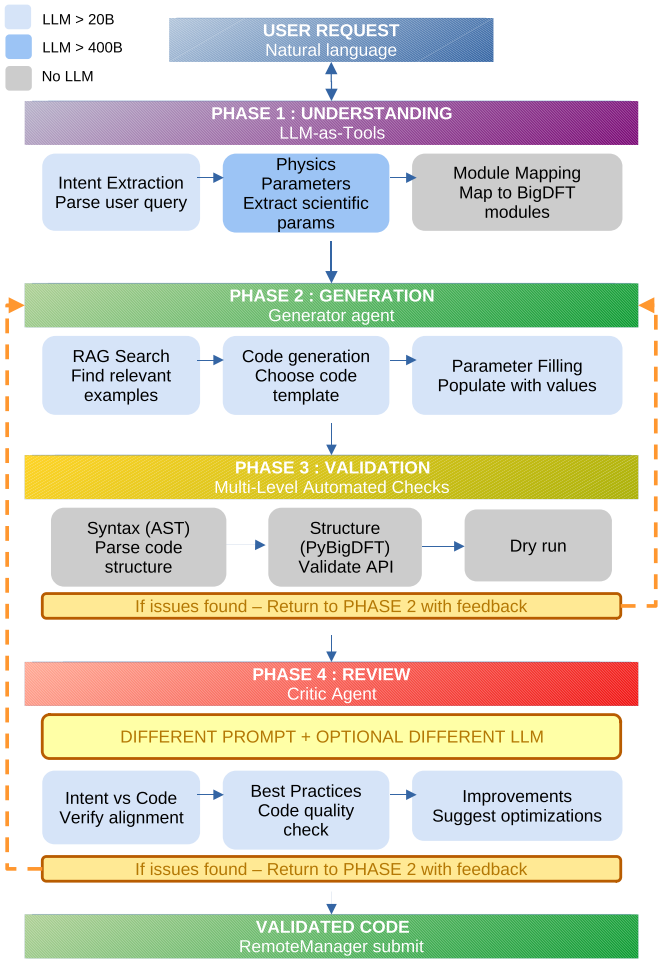


FIG. 2. LARA-HPC architecture. A user request is transformed through successive phases of understanding, generation, validation, and review, before producing validated code executed via `remotemanager`.

capable of generating and refining workflows while preserving the controlled interaction with HPC resources.

Our architectural choice is guided by a principle of minimal sufficient autonomy [24]: The agent should be as flexible as needed for scientific reasoning, but no more autonomous than is operationally safe in an HPC setting. A fully deterministic pipeline is too rigid for open-ended scientific requests, while a fully autonomous agent with direct execution authority would be difficult to justify in a high-consequence environment where an incorrect job may waste substantial computational resources [25,26]. LARA-HPC therefore adopts a staged design in which creative reasoning, deterministic validation, and controlled execution are explicitly separated. This positioning is also consistent with recent practical guidance on agent construction [24,27], improving, in passing, also the traceability of the approach.

A. Overview of the LARA-HPC workflow

LARA-HPC is structured as a multiphase pipeline that transforms a natural-language request into a validated and executable workflow. The architecture is illustrated in Fig. 2. The workflow is decomposed into four main phases:

(1) Understanding: extraction of scientific intent and mapping to domain-specific modules.

(2) Generation: construction of candidate workflows using templates and retrieval.

(3) Validation: multilevel verification including syntax, API, physical constraints, and dry-run execution.

(4) Review: higher-level critique and optimization of the generated workflow.

This structured decomposition allows separating concerns between semantic interpretation, code generation, correctness verification, and optimization.

The decomposition also reflects distinct failure modes. Misunderstanding the user request leads to an inappropriate workflow family; hallucinations during generation lead to incorrect code or invalid API usage; subtle scientific inconsistencies survive syntax checks but are exposed by domain validation; and workflows that are formally correct may still be suboptimal or misaligned with the original intent. The four-phase structure is intended to address each of these failure modes with a dedicated mechanism rather than a single monolithic agent prompt. A similar phasing strategy has been adopted by the agentic system "The AI Scientist"—a pipeline that achieves the vision of full end-to-end automation of the scientific process [9]. The AI Scientist sequentially completes four main phases covering idea generation, tree-based experimentation, and manuscript writing and reviewing.

A further design principle of LARA-HPC is the separation of cognitive labor across models of different sizes (highlighted by different colors in Fig. 2). The most demanding reasoning step—translating a scientific problem into physically meaningful parameters, understanding domain equations, and selecting the right computational strategy—is handled by a single large model. This frontier model brings the broad scientific knowledge and deep reasoning capacity required to interpret quantum chemistry or electronic-structure problems correctly. Leveraging different LLM models at various stages and for different agent tasks is a common practice in agentic systems to best utilize specific skills (reasoning, coding, vision-language tasks) and to optimize costs [9]. A model with strong tool-calling capabilities can also be utilized to dynamically orchestrate the four stage workflow. Once this high-level physical reasoning is complete and the simulation parameters are established, subsequent well-scoped subtask steps are delegated to smaller, faster, and cheaper models.

B. Phase 1: scientific understanding

The first phase transforms the user request, expressed in natural language, into structured scientific information. This includes extraction of intent (e.g., energy calculation, structural relaxation), identification of relevant physical parameters, and mapping to available modules in the target simulation framework (e.g., BIGDFT). This step is critical because scientific workflows cannot be constructed solely from syntactic patterns: They require a correct interpretation of domain-specific concepts. The output of this phase is a structured representation of the problem, which serves as input for workflow generation.

In practical deployments, this phase also provides a natural location for clarification when the user request is underspecified. For example, an atomization-energy calculation may require the system to determine whether geometry optimization is intended, which pseudopotential family should be used, or whether open-shell reference atoms must be considered explicitly. Resolving such ambiguities before generation prevents avoidable downstream errors and is consistent with recent guidance on human-in-the-loop agent design [24,25]. Early appraisal of whether the simulation framework has the capability to perform the requested study further steers agents or human users toward hypotheses that are testable by the available tools.

C. Phase 2: workflow generation

The generation phase constructs candidate workflows by combining retrieval and templating mechanisms. It consists of three main steps:

(1) Retrieval (RAG): Relevant examples and documentation are retrieved from a curated knowledge base.

(2) Code generation: A candidate function is generated according to a specific code template.

(3) Parameter filling: The source code is incorporated into a structured message incorporating output from the previous stages.

This approach ensures that generated workflows remain anchored to known usage patterns, reducing the likelihood of hallucinated or inconsistent code. A common approach for grounding LLM responses in domain-specific knowledge is retrieval-augmented generation, where relevant documentation chunks are fetched at inference time and injected into the model's context. This strategy works well when the knowledge base is large and dynamic, as it avoids overwhelming the context window with irrelevant information. Another viable solution is represented by always-on memory (see Refs. [28,29] for discussions of agent memory). In this approach, domain tutorials are manually ingested into the model's persistent memory, tested against representative tasks, and refined until the model consistently produces correct results. This validated memory is then deployed as a static knowledge context throughout the workflow.

However, at this stage, correctness is not guaranteed. The generated workflow is only a candidate that must be validated before execution. This phase is intentionally more expressive than static tool calling. It allows the system to construct new workflow logic when required by the scientific question, for example, by adding atomic reference calculations, convergence loops, or parameter comparisons that were not hard coded as independent tools.

D. Phase 3: multilevel validation

The validation phase is the core innovation of LARA-HPC. It introduces a layered verification process that progressively ensures the correctness of the generated workflow. The role of validation is therefore not to replace generation, but to make this added expressivity safe enough for scientific use. In this sense, the architecture is closer to an agentic RAG workflow than to a static retrieval pipeline [30]. The validation pipeline includes

(1) syntactic validation: parsing of the abstract syntax tree;

(2) structural validation: verification of API usage (e.g., through BigDFT's Python interfaces);

(3) dry-run execution: evaluation of the workflow and its resource requirements using simulation code validation modes.

The dry-run step is particularly important, as it enables the detection of run-time and semantic errors without incurring the cost of full HPC execution. If issues are detected, structured feedback is generated and the workflow is returned to the generation phase for correction. This creates an iterative refinement loop that continues until the workflow satisfies all validation criteria.

E. Phase 4: critical review

In addition to automated validation, LARA-HPC includes a review phase driven by a dedicated agent. This phase operates with a potentially different prompt or even a different model, and focuses on higher-level aspects, such as alignment between user intent and generated code, adherence to best practices, or identification of possible optimizations. This step is not strictly required for correctness, but improves the overall quality and robustness of the workflow. It introduces a second level of reasoning that complements the validation phase. Although the present version does not yet implement this phase end to end, we keep it in the architecture because scientific acceptability is broader than executability. A workflow can pass dry-run validation and still be methodologically weak, unnecessarily expensive, or poorly aligned with the actual question being asked. The review phase is intended to capture precisely this distinction, and follows the broader generator-critic pattern explored in recent self-refinement work [31].

F. Key design principles

Once a workflow has successfully passed validation and review, it is considered safe for execution. At this stage, the code is submitted to the HPC environment through `remotemanager`. This final step closes the loop between agentic reasoning and production execution. Importantly, the agent never interacts directly with the HPC system: All operations are mediated through the `remotemanager` layer described in Sec. II. Everything happens on the local user's workstation.

The architecture of LARA-HPC is guided by three key principles:

(1) Separation of concerns: Reasoning, validation, and execution are handled by distinct components.

(2) Validation-first design: Workflows must be verified before execution.

(3) Controlled interaction: All HPC operations are mediated through well-defined interfaces.

These principles ensure that flexibility in workflow generation does not compromise the reliability and reproducibility required for HPC-based scientific computing.

IV. DRY-RUN-DRIVEN VALIDATION

A central component of the LARA-HPC framework is the integration of a *dry-run* execution mode, which enables the validation of simulation workflows prior to their actual

execution on HPC resources. Unlike traditional static validation approaches, the dry-run mechanism in BIGDFT is implemented directly within the execution pipeline of the simulation code, allowing for domain-aware verification of inputs. In BIGDFT, the dry-run mode can be triggered in two complementary ways:

(1) through the Python API, by calling the `run()` method of the calculator with the keyword argument `dry_run=True`;

(2) through the environment, by setting the variable `BIGDFT_MPIDRYRUN`, which redirects execution to a dedicated dry-run executable. The value of the variable corresponds to the number of MPI processes the simulation will use.

In both cases, the dry-run mode modifies the execution pathway of the calculation. When dry run is activated, the standard BIGDFT executable is replaced by a specialized tool that performs a simulation of the execution, including input parsing and internal setup, but does not allocate the computational resources required for a full self-consistent field (SCF) calculation.

The dry-run mechanism is not an external checker, but is embedded within the same execution pipeline used for actual simulations. It thereby provides a rich set of validation capabilities that go beyond simple syntactic checks. These include

(1) Input validation: parsing and verification of input files;

(2) Structural consistency: validation of parameter dictionaries and required fields;

(3) Physical setup: construction of internal data structures such as grids, basis functions, and pseudopotentials;

(4) Resource estimation: evaluation of memory requirements based on system size and parallel configuration.

In particular, the memory estimation step provides a quantitative assessment of the feasibility of the simulation under given HPC conditions. The resulting logs contain a *Memory Consumption Report*, which is explicitly checked in the calculator implementation to confirm that the dry run completed successfully. This level of validation enables the detection of errors that would otherwise only appear during execution, such as inconsistent parameter combinations or infeasible resource requirements. It further provides feedback to the agentic system concerning the optimal allocation of HPC resources.

The dry-run execution produces standard output and log files that are processed in the same way as full runs. The calculator inspects debug files and log timestamps to determine whether the execution was successful. In case of failure, detailed error information is retrieved from dedicated debug files and reported back to the user or calling system. This mechanism provides structured feedback that can be directly used by the LARA-HPC validation loop to refine the workflow. Unlike generic error messages, these diagnostics are grounded in the internal logic of the simulation code, making them particularly valuable for automated correction.

A key advantage of the dry-run mode is its low computational cost. Since no full SCF cycle is performed, the execution time is negligible compared to a production run, even for large systems. This makes it possible to invoke dry run repeatedly within an iterative loop without impacting HPC resource usage. This property is essential for LARA-HPC, where validation is performed multiple times during the generation-refinement cycle. The dry run thus acts as a fast and reliable oracle for workflow correctness.

Within the LARA-HPC architecture (Sec. III), the dry run constitutes the final stage of the validation phase. It complements earlier checks such as syntax parsing and API validation by providing an execution-level verification grounded in the simulation code itself. This integration enables a shift from *static validation* to *simulation-native validation*. Instead of attempting to predict correctness from code structure alone, LARA-HPC relies on the simulation engine to evaluate the validity of the workflow. As a result, only workflows that pass the dry-run validation are submitted to execution through `remotemanager`, guaranteeing a high level of reliability in production campaigns.

V. USE CASE: VALIDATION-DRIVEN WORKFLOW GENERATION

To illustrate the capabilities of the LARA-HPC framework, we present concrete use cases (atomization energy and surface binding) based on an implementation of the architecture described in Sec. III. This implementation focuses on the generation and validation of atomistic simulation workflows using BIGDFT, and is designed as a minimal yet representative instance of the overall architecture. We focus on highlighting representative challenges addressed by LARA-HPC in practice, rather than presenting a benchmark dataset, as the models available today almost never correctly generate BIGDFT workflows. Hence, our results focus on how LARA-HPC improves code generation, rather than highlighting the dramatically higher success rate of LARA-HPC that is a byproduct of BIGDFT being poorly represented in the training data of current LLMs. The `remotemanager` package [32] and LARA-HPC [33] used for these tests are both available as open source software.

The implementation considered in this work realizes the LARA-HPC pipeline up to Phase 3 (Validation) of Fig. 2. The RAG component is implemented using materials from a BIGDFT training event, which provide representative examples of input configurations and workflows. This choice ensures that generated code remains consistent with established usage patterns while keeping the implementation lightweight. We use Google's GEMINI 2.5 Flash [34] as the smaller, stage-specific LLM; *gemini-embedding-001* as the embedding model; and Anthropic's Sonnet 4.6 as the frontier model. This section demonstrates that scientific workflow generation can be coupled to simulation-native validation in a way that catches meaningful errors before production execution. The goal of the use case is not to claim exhaustive automation, but to show that the validation loop materially changes what kinds of workflows can be trusted.

A. Illustrative error detection and correction

To provide a concrete understanding of the validation-driven workflow, we present representative examples of errors generated during the atomization workflow construction and their subsequent correction through the LARA-HPC pipeline.

API misuse. A common class of errors arises from incorrect usage of the simulation API. For instance, the generated workflow may import a nonexisting/misspelled Python module. During the validation phase, this error is detected either at the structural validation level or during dry-run execution, producing an error message indicating that the module does not exist. Another concrete example concerns dynamic self-correction after an `AttributeError`. Starting from a natural-language request to construct a water molecule in a periodic box and return its center of mass, the agent first produced an invalid call to `System.get_center_of_mass()`. The dry-run stage intercepted the error, after which the generation phase consulted the indexed documentation and replaced the invalid method by the valid `System.centroid` attribute. The corrected workflow then executed successfully and returned a coordinate vector. This example shows the intended control loop: Failure is not hidden, but converted into structured feedback that triggers a focused repair.

Missing input parameters. Another frequent issue concerns incomplete or inconsistent input definitions, such as missing keys in the input data. The dry-run execution detects missing required fields (e.g., atomic positions or pseudopotentials), leading to a failure at the input parsing stage.

Physically inconsistent parameters. More subtle errors involve inconsistent configurations. A representative example is a spin-polarized calculation on a single hydrogen atom with an impossible total magnetic moment:

```
from BigDFT.Database.Molecules import
get_molecule
from BigDFT.Inputfiles import Inputfile

sys = get_molecule("H")
inp = Inputfile()
inp.set_xc("PBE")
inp.set_hgrid(0.4)
inp.spin_polarize(2)
```

This configuration is syntactically valid Python and uses the correct API, but it violates a physical constraint because a one-electron system cannot sustain a total spin polarization of two. In the executable validation path, BigDFT reports the issue explicitly:

```
ERROR: some problem occured during the
execution of the command The error occured
is BIGDFT_INPUT_VARIABLES_ERROR Additional
Info: Spin-Polarized calculation (nspin=
2). The mpol polarization should be less
than the number of electrons (mpol=2 and
nel= 1)
```

The corrected workflow simply restores a physically admissible polarization.

These examples highlight the multilevel nature of the LARA-HPC validation process. Errors are not only syntactic but also structural and physical. The dry-run mechanism plays a central role in detecting issues that cannot be inferred from code inspection alone. By iteratively correcting such errors, the system converges toward workflows that are both syntactically valid and scientifically meaningful. This illustrates the transition from a generative paradigm to a validation-driven construction process. The spin polarization failure raises at the same time also a general crucial point: LARA-HPC relies on the robustness of a code with respect to exception handling and error descriptions. LARA-HPC was able to set the proper spin for hydrogen since BIGDFT makes users aware of the proper correlation of spin state and available electrons. This example points to the need of having a *specific* dry-run tool, designed by the developers of the code into which the HPC campaign should run, that will be consumed by the agent to meet required validation criteria. The ability of the software developer will be then in being capable to design such a tool in a suitable way to inform HPC campaigns and relevant workflow actions.

B. Why atomization energy matters as a use case

Beyond these local error-correction examples, the broader scientific motivation for LARA-HPC is the handling of workflows whose difficulty lies in methodological branching rather than in syntax alone. Atomization energies provide a compact example. To compute them correctly, the agent must coordinate molecular and atomic calculations, preserve the same exchange-correlation and numerical settings across all runs, choose appropriate reference spin states for isolated atoms, and often explore variants such as pseudopotential family or convergence parameters.

In a static tool-calling framework, each of these scientific variants requires an explicit new tool or a more complex interface. In a generation-based framework, the agent can instead synthesize the required workflow structure on demand. The price of this flexibility is that the resulting code must be validated much more carefully. For this reason, atomization energy is not merely a chemistry example—it is a representative stress test for the central claim of this paper, namely, that scientific usefulness requires more expressivity than fixed tools, but HPC safety requires more discipline than unconstrained code generation.

Using LARA-HPC to generate a code that converges the atomization energies revealed another limitation of a predefined tool approach. BIGDFT represents the electronic structure using wavelets, which support a multiresolution approach. Around each atom, a coarse-grained region with lower resolution reduces computational cost, and its size can be adjusted via the `set_rmult()` function. The key point is not the specific parameter, but the fact that the system had to modify workflow logic rather than merely fill slots in a preexisting interface (Listing 4).

Listing 4. Excerpt of a validated code modification generated by LARA-HPC when extending a BigDFT workflow to accept `crmult` through `set_rmult()`.

```
def input_template(xc='PBE', hgrid=0.35,
**kwargs):
  from BigDFT.Inputfiles import Inputfile
  inp0 = Inputfile()
  inp0.set_xc(xc)
  inp0.set_hgrid(hgrid)
  inp_spec = {'XC': xc, 'h': hgrid}

  if 'crmult' in kwargs:
      crmult_value = kwargs.pop('crmult')
      inp0.set_rmult(crmult_value)
      inp_spec['rmult'] = crmult_value

return inp0, inp_spec
```

One possible solution is to create a fully general interface exposing all possible input options. Another would be to allow the agent to first generate an input file that is sent to the MCP server. Both of these solutions place a heavy burden on prompt engineering to ensure the agent properly manages an enormous array of possible input options and all of their dynamic constraints. Thus, the LARA-HPC approach is to enable the generation of validated, programmatic workflows in the context of the most relevant documentation and physics considerations. Once the workflow passes all validation steps, it can be safely executed on HPC resources using the `remotemanager` interface described in Sec. II. At this stage, the workflow is no longer a speculative generated code, but a verified simulation procedure that has been tested under realistic conditions.

C. Validation-driven HPC

To connect AI agents to HPC resources for predefined workflows, it is sufficient to use `remotemanager` as glue code through the scheme in Listings 2 and 3. However, the purpose for LARA-HPC is to enable autonomous AI assistants to run novel, computationally demanding calculations having validated the workflow locally. As a demonstration of this point, the orchestrating agent is prompted with the following query:

```
Inside the data directory there is a XYZ
file. I want to calculate the binding energy
of the water molecule to the surface in that
geometry using the PBE functional.
```

In this scenario, the LLM is provided with an anatase/water structure composed of 291 atoms [35].

The agent reaches the dry-run phase and then responds to the user with the following information:

```
The slab calculations are large(~200 GB
peak per process at hgrid=0.37). This is
a 291-atom TiO2 slab - you will need a
significant HPC allocation with many MPI
processes to make the memory per process
tractable.
```

The memory requirements generated in the LLM's response are listed in Table I. As the dry-run tool allows the agent to specify the number of MPI ranks, we then ask the orchestrating agent:

```
I plan to run this on supercomputer Fugaku.
The calculation should be run with 4 MPI per
node. Each node can safely use approximately
26 GB. Using dry run, tell me how many nodes
I should use for this calculation.
```

The agent performs two dry-run tool calls to generate the required data. It then responds to the user:

```
Recommendation: use 10 nodes (40 MPI
processes, 4 per node). This gives ~1.2
GB of headroom per process on the worst-case
calculation (slab + water). If you want a
more comfortable margin, 12 nodes (48 MPI)
would bring the peak to ~4,560 MB/process,
leaving ~2.1 GB free per process.
```

Thus, the dry-run approach provides a guard against the agent executing underprovisioned calculations. This saves the high latency work of finding a suitable calculation setup with each experiment going through the supercomputer queue and wasting computational resources.

VI. DISCUSSION: TOWARD A COPILOTED HPC ECOSYSTEM

The results presented in this work highlight a key limitation of current AI-assisted scientific workflows: While modern large language models are highly effective at generating code, they lack the mechanisms required to ensure correctness and reproducibility in high-performance computing environments. In contrast, scientific simulations demand strict guarantees on input validity, resource usage, and physical consistency. In this work, we introduced LARA-HPC, a validation-driven agentic framework designed to enable reliable generation of atomistic simulation workflows in HPC environments. By combining

TABLE I. Memory analysis produced by the AI agent for calculating the slab system. The subcaptions, headers, and entries are all verbatim taken from the agent's output.

(a) Memory estimates per component (one process dry run)

Calculation	Wavefunction memory	Estimated peak
Slab + water (291 atoms)	∼1784 MB	∼197 GB
Bare slab (288 atoms)	∼1629 MB	∼195 GB
Water molecule (3 atoms)	∼1.5 MB	∼136 MB

(b) Memory per MPI process—slab + water (the bottleneck)

MPI processes	Nodes (@ 4 MPI/node)	Peak/process	Limit/process	Fits?
32	8	6935 MB	6656 MB	✗
40	10	5473 MB	6656 MB	√

a controlled execution layer based on `remotemanager` with simulation-native validation through the dry-run capabilities of BIGDFT, we argued for a shift from generative AI workflows to a validation-first paradigm.

In principle, users may manage their atomistic simulations using any of the highly general agentic systems available today (coding agents, "claws," etc.). These agents can run arbitrary commands, connect arbitrary systems, and generate arbitrary code. However, this extreme degree of nondeterministic behavior risks the wasting of precious computational resources on the production of scientifically meaningless results. When correct results are achieved, they will only come after long and costly trial-and-error loops. In fact, this iterate until correctness approach is the common strategy of current AI agents—one that is effective only if the resource use, latency of trials, and safety risks are low. Even when such an approach finally succeeds, human in the loop would be a warranted feature. Preserving traceable tool calling and well-automated relationships between tools/phases of a production process will also ensure reproducibility of the scientific tasks.

For example, we noted in Sec. II A that `remotemanager` works by performing the same actions as a human user would; from this, we know that a general AI agent is also capable of connecting to a remote machine, creating and submitting jobscripts, checking for results, etc. For a given HPC resource or DFT program, when the underlying model is not sufficiently trained on the specific documentation, today's agents can search the web and gather the required information. Yet, each of these setup steps delays achieving the goals of the scientific project. The benefit of providing suitable tooling is to be sure the agent does not need to waste time nor its context window on discovering the correct set of workflow operations.

Another weak point of deploying general agent systems is that each step in the research process becomes dependent on a frontier quality model. This may not be desirable due to the high costs and security risks that come with using commercial providers. Such an approach is also sensitive to changes to models made outside of the user's control, which may disrupt a given workflow. The modular LARA-HPC architecture separates stages by the quality of engine required (Fig. 2), with tight inner reasoning loops accomplished using smaller models. The code that is generated becomes independent of any given model, allowing the workflow to be executed (even with a manual process) reproducibly in the future.

A simple solution to the weak points of general agents is to package predefined tools suitable for steps in a scientific workflow. For example, the composition of the model context protocol and `remotemanager` allows users to easily define tools that perform atomistic simulations using HPC resources. In addition to formalizing these steps to ensure immediate correctness and a separation of concerns, providing tools serves as a form of prompt engineering [36]—biasing agent actions toward approaches it may not immediately produce under standard operating conditions.

The drawback of this approach is that predefined tools are scientifically constraining. In research, the challenge is often not merely choosing among existing actions, but constructing a workflow variant that was not fully conceived in advance due to an evolving hypothesis. To allow for the full expressiveness of new procedural patterns, while ensuring controlled and correct execution, we introduce the LARA-HPC AI system based on a domain-specific, multiagent loop. In addition to the dynamic prompting made possible by a multiphase workflow incorporating physics reasoning and RAG, LARA-HPC is centered on a dry-run primitive that enables reliable validation of proposed workflows with low latency and resource use.

Based on these experiences, we argue that it is now the task of the broader computational physics community to develop shared knowledge on developing agentic systems suitable for its specific research requirements. These systems may come in the form of MCP servers, agent SKILL files, tools to be inserted in Agent Client Protocol schemes, or full packages like LARA-HPC. The community's frameworks would then be integrated into end-to-end agentic systems like The AI Scientist [9] and others [37–42], which are designed to autonomously conduct all distinct phases of scientific research—research ideas, write code, run experiments, plot and analyze data, write scientific manuscripts, and perform peer review. Recent research suggests that the length of tasks AI can reliably complete is doubling roughly every seven months [43]. This implies that many of today's implementation and debugging limitations could be overcome sooner than expected. Nevertheless, integrating domain-specific systems into these

advanced models will remain essential to reduce the increasingly dominant burden of tool calling on agentic science [44].

One such critical requirement will be the ability to operate these systems within a *sandboxed environment*. This environment must provide controlled access to simulation software and associated data, HPC resources and scheduling systems, external AI services, or locally deployed models. The sandbox ensures that all interactions are traceable, reproducible, and compliant with the constraints of the research context, including licensing restrictions and data confidentiality. Within this framework, LARA-HPC can be seen as an orchestration layer of the sandbox that connects state-of-the-art copilots, retrieval systems, and validation tools, without being tied to a specific provider or model. This flexibility is essential in scientific environments, where requirements vary widely across domains and institutions.

ACKNOWLEDGMENTS

W.D. acknowledges the FOCUS (Hyogo Prefecture, Kobe City) Project to Promote the Formation of a Research and Education Center (COE) in the Field of Computational Science. We acknowledge Eliott Jacopin for useful discussions. We thank E. Polack, D. Caliste, J. Janssen, C. Herrera-Contreras, T. Abui Degbotse, J. Briki, L. Didukh, and the organizers of the 2025 LLM Hackathon for Applications in Materials Science and Chemistry for stimulating discussions and for providing momentum to the initiative.

DATA AVAILABILITY

The `remotemanager` package [32] and LARA-HPC [33] used for these tests are both available as open source software.

[1] M. Chen *et al.*, Evaluating large language models trained on code, arXiv:2107.03374.

[2] J. Gottweis *et al.*, Towards an AI co-scientist, Nature (London) **6**55, 487 (2026).

[3] Y. Zimmermann *et al.*, 32 examples of LLM applications in materials science and chemistry: Towards automation, assistants, agents, and accelerated scientific discovery, Mach. Learn.: Sci. Technol. **6**, 030701 (2025).

[4] Y. Zimmermann *et al.*, Reflections from the 2024 large language model (LLM) hackathon for applications in materials science and chemistry, arXiv:2411.15221.

[5] N. Alampara, A. Aneesh, M. Ríos-García, A. Mirza, M. Schilling-Wilhelmi, A. A. Aghajani, M. Sun, G. Prastalo, and K. M. Jablonka, General-purpose models for the chemical sciences: LLMs and beyond, Chem. Rev. **126**, 2484 (2026).

[6] I. Mandal, J. Soni, M. Zaki, M. M. Smedskjaer, K. Wondraczek, L. Wondraczek, N. N. Gosvami, and N. M. Anoop Krishnan, Evaluating large language model agents for automation of atomic force microscopy, Nat. Commun. **16**, 9104 (2025).

[7] A. M. Bran, S. Cox, O. Schilter, C. Baldassari, A. D. White, and P. Schwaller, Augmenting large language models with chemistry tools, Nat. Mach. Intell. **6**, 525 (2024).

[8] A. Vriza, U. Kornu, A. Koneru, H. Chan, and S. K. R. S. Sankaranarayanan, Multi-agentic AI framework for end-to-end atomistic simulations, Digit. Discov. **5**, 440 (2026).

[9] C. Lu, C. Lu, R. T. Lange, Y. Yamada, S. Hu, J. Foerster, D. Ha, and J. Clune, Towards end-to-end automation of AI research, Nature (London) **651**, 914 (2026).

[10] S. Gao *et al.*, Democratizing AI scientists using tooluniverse, arXiv:2509.23426.

[11] Y. Qu, K. Huang, M. Yin, K. Zhan, D. Liu, D. Yin, H. C. Cousins, W. A. Johnson, X. Wang, M. Shah, R. B. Altman, D. Zhou, M. Wang, and L. Cong, CRISPR-GPT for agentic automation of gene-editing experiments, Nat. Biomed. Eng. **10**, 245 (2026).

[12] Y. Zou, A. H. Cheng, A. Aldossary, J. Bai, S. X. Leong, J. A. Campos-Gonzalez-Angulo, C. Choi, C. T. Ser, G. Tom, A. Wang, Z. Zhang, I. Yakavets, H. Hao, C. Crebolder, V. Bernales, and A. Aspuru-Guzik, El Agente: An autonomous agent for quantum chemistry, Matter **8**, 102263 (2025).

[13] T. D. Pham, A. Tanikanti, and M. Keçeli, Chemgraph as an agentic framework for computational chemistry workflows, Commun. Chem. **9**, 33 (2026).

[14] Q. Campbell, S. Cox, J. Medina, B. Watterson, and A. D. White, MDCrow: Automating molecular dynamics workflows with large language models, Mach. Learn.: Sci. Technol. **7**, 025037 (2026).

[15] Z. Wang, H. Huang, H. Zhao, C. Xu, S. Zhu, J. Janssen, and V. Viswanathan, Dreams: Density functional theory based research engine for agentic materials simulation, arXiv:2507.14267.

[16] E. Deelman, D. Gannon, M. Shields, and I. Taylor, Workflows and e-Science: An overview of workflow system features and capabilities, Future Gener. Comput. Syst. **25**, 528 (2009).

[17] S. P. Huber *et al.*, Aiida 1.0, a scalable computational infrastructure for automated reproducible workflows and data provenance, Sci. Data **7**, 300 (2020).

[18] R. M. Martin, *Electronic Structure: Basic Theory and Practical Methods* (Cambridge University Press, Cambridge, UK, 2020).

[19] V. Brázdová and D. R. Bowler, *Atomistic Computer Simulations: A Practical Guide* (Wiley-VCH, Weinheim, Germany, 2013).

[20] P. Lewis *et al.*, Retrieval-augmented generation for knowledge-intensive NLP tasks, in *Advances in Neural Information Processing Systems*, edited by H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Curran Associates, Inc., Red Hook, NY, 2020), Vol. 33, pp. 9459–9474.

[21] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, ReAct: Synergizing reasoning and acting in language models, in *Proceedings of the Eleventh International Conference on Learning Representations* (ICLR, 2023), pp. 30084–30116.

[22] L. E. Ratcliff, W. Dawson, G. Fisicaro, D. Caliste, S. Mohr, A. Degomme, B. Videau, V. Cristiglio, M. Stella, M. D'Alessandro, S. Goedecker, T. Nakajima, T. Deutsch, and L. Genovese, Flexibilities of wavelets as a computational basis set for large-scale electronic structure calculations, J. Chem. Phys. **152**, 194110 (2020).

[23] W. Dawson, L. Beal, L. E. Ratcliff, M. Stella, T. Nakajima, and L. Genovese, Exploratory data science on supercomputers for

quantum mechanical calculations, Electron. Struct. **6**, 027003 (2024).

[24] A. Blount, A. Gulli, S. Saboo, M. Zimmermann, and V. Vuskovic, Introduction to agents, kaggle/Google Whitepaper, 2025, https://www.kaggle.com/whitepaper-introduction-to-agents, accessed: 11 April 2026.

[25] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J. Wen, A survey on large language model based autonomous agents, Front. Comput. Sci. **18**, 186345 (2024).

[26] Y. Gan *et al.*, Navigating the risks: A survey of security, privacy, and ethics threats in LLM-based agents, arXiv:2411.09523.

[27] OpenAI, A practical guide to building agents, OpenAI Documentation, 2024, https://platform.openai.com/docs/guides/agents, accessed: 11 April 2026.

[28] C. Packer *et al.*, MemGPT: Towards LLMs as operating systems, arXiv:2310.08560.

[29] W. Xu, Z. Liang, K. Mei, H. Gao, J. Tan, and Y. Zhang, A-MEM: Agentic memory for LLM agents, in *Advances in Neural Information Processing Systems*, edited by D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (Curran Associates, Inc., Red Hook, NY, 2025), Vol. 38, pp. 17577–17604.

[30] A. Singh, A. Ehtesham, S. Kumar, and T. T. Khoei, Agentic retrieval-augmented generation: A survey on agentic RAG, arXiv:2501.09136.

[31] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, S. Gupta, B. P. Majumder, K. Hermann, S. Welleck, A. Yazdanbakhsh, and P. Clark, Self-refine: Iterative refinement with self-feedback, in *Advances in Neural Information Processing Systems*, edited by A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Curran Associates, Inc., Red Hook, NY, 2023), Vol. 36, pp. 46534–46594.

[32] BigDFT developers, RemoteManager repository, canonical maintained project page, 2026, https://gitlab.com/l_sim/remotemanager, accessed: 16 September 2026.

[33] BigDFT developers, LARA-HPC: Lara agentic research assistant for HPC, canonical maintained project page, 2026, http://github.com/bigdft-group/lara-hpc, accessed: 16 September 2026.

[34] G. Comanici *et al.*, Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, arXiv:2507.06261.

[35] G. Fisicaro, S. Filice, S. Scalese, G. Compagnini, R. Reitano, L. Genovese, S. Goedecker, I. Deretzis, and A. La Magna, Wet environment effects for ethanol and water adsorption on anatase TiO_2 (101) surfaces, J. Phys. Chem. C **124**, 2406 (2020).

[36] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing, ACM Comput. Surv. **55**, 1 (2023).

[37] Y. Weng, M. Zhu, Q. Xie, Q. Sun, Z. Lin, S. Liu, and Y. Zhang, DeepScientist: Advancing frontier-pushing scientific findings progressively, in *Fourteenth International Conference on Learning Representations*, edited by C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (ICLR, 2026), Vol. 2026, pp. 47981–48037.

[38] Analemma Intelligence, FARS: Fully Automated Research System, 2026, https://analemma.ai/blog/introducing-fars/, accessed: 9 April 2026.

[39] Y. Lyu *et al.*, EvoScientist: Towards multi-agent evolving AI scientists for end-to-end scientific discovery, arXiv:2603.08127.

[40] Y. Li, X. Yang, X. Yang, X. Wang, W. Liu, and J. Bian, R&D-Agent-Quant: A multi-agent framework for data-centric factors and model joint optimization, in *Advances in Neural Information Processing Systems*, edited by D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (Curran Associates, Inc., Red Hook, NY, 2025), Vol. 38.

[41] S. Schmidgall, Y. Su, Z. Wang, X. Sun, J. Wu, X. Yu, J. Liu, M. Moor, and Z. Liu, Agent laboratory: Using LLM agents as research assistants, *Findings of the Association for Computational Linguistics: EMNLP 2025*, edited by C. Christodoulopoulos, T. Chakraborty, C. Rose, and V. Peng (Association for Computational Linguistics, 2025), pp. 5977–6043.

[42] F. Villaescusa-Navarro *et al.*, DENARIO Project: Deep Knowledge Artificial Intelligence Agents for Scientific Discovery, PRX Intelligence **1**, 021001 (2026).

[43] T. Kwa *et al.*, Measuring AI ability to complete long tasks, METR Blog, 2025, https://metr.org/blog/2025-03-19-measuring-ai-ability-to-complete-long-tasks/, accessed: 11 April 2026.

[44] R. Raj, H. Wang, and T. Krishna, A CPU-centric perspective on agentic AI, arXiv:2511.00739.