HPC-AutoResearch: Adapting Autonomous Research Systems for HPC through Split-Phase Execution and LLM-Based Code Generation

Takanori Kotama[1,2], Shun-ichiro Hayashi[1], Daichi Mukunoki[3], Rio Yokota[2], Satoshi Ohshima[4], Tetsuya Hoshino[3] and Takahiro Katagiri[3]

[1] *Graduate School of Informatics, Nagoya University, Aichi, Japan*
[2] *RIKEN Center for Computational Science, Hyogo, Japan*
[3] *Information Technology Center, Nagoya University, Aichi, Japan*
[4] *Research Institute for Information Technology, Kyushu University, Fukuoka, Japan*

Abstract

Recent LLM-based autonomous research systems have demonstrated the ability to conduct the full scientific cycle—idea generation, experiment execution, paper writing, and peer review—without human intervention. However, these systems have primarily been demonstrated in Python-based machine learning environments, leaving open the question of how to automate research workflows that depend on compiled-language code (C, C++, Fortran, CUDA) and shared cluster environments managed by job schedulers. Such compiled-code workflows form a sequential pipeline—environment setup, code generation, compilation, execution, and analysis—in which a failure at any stage prevents subsequent stages from running, often forcing the entire process to restart. We present HPC-AutoResearch, a proof-of-concept system for the autonomous execution of compiled-code research workflows in HPC-like environments. Rather than attempting to cover the full HPC optimization loop, we focus on stabilizing the sub-pipeline from code generation to executable results. The key contribution is the Split-Phase execution model, which divides this sub-pipeline into five phases—planning, environment setup, coding, compilation, and execution—localizing failures within each phase and enabling iterative repair via rootless Singularity container-based isolation. A MemGPT-based compressed inference memory mechanism (Core, Recall, and Archival memory) records prior errors and recovery attempts, reducing repeated failures across iterations. In a proof-of-concept case study on the Himeno Benchmark, our system was the only configuration to achieve 3/4 on all seven LLM review criteria, while a general-purpose agent (Codex) failed entirely at environment setup. The memory-enabled run achieved a 47.0% node success rate vs. 40.5% without memory, with Stage 2 success rate improving from 29.2% to 54.2%. We position these results as a demonstration of feasibility rather than a comprehensive performance comparison across diverse HPC workloads.

Keywords

Autonomous Research, LLM-Based Code Generation, High-Performance Computing, Multi-Agent Systems, Tree Search

1. Introduction

Recent advances in large language models (LLMs) have spurred end-to-end autonomous research systems such as AI Scientist [1] and Agent Laboratory [2] that conduct the full scientific cycle—idea generation, experiment execution, paper writing, and peer review—without human intervention, while AIDE [3] provides tree-search-based code exploration for machine-learning engineering. A central capability of these systems is *generative code intelligence*: using LLMs to generate, debug, and refine code as part of the research process. However, existing end-to-end autonomous research systems have primarily been demonstrated in Python-based machine learning environments, leaving a critical gap in domains that require compiled-language code generation.

GeCoIn 2026: Generative Code Intelligence Workshop @ IJCAI-ECAI 2026, August 15–17, 2026, Bremen, Germany

kotama.takanori.j6@s.mail.nagoya-u.ac.jp (T. Kotama)

Autonomous systems for compiled-code research workflows require more than code generation. Such workflows typically involve a sequential pipeline consisting of environment setup, code generation, compilation, execution, and result analysis. A failure at any stage prevents subsequent stages from being executed, often forcing the entire process to restart. This observation motivates a system design that can (i) localize failures, (ii) recover from them within individual stages, and (iii) avoid repeatedly making the same mistakes. The execution environment imposes additional constraints—rootless containers and job schedulers (e.g., SLURM)—that general-purpose code generation systems cannot handle.

In this work, we focus on automating the critical sub-pipeline from code generation to executable results, rather than attempting to cover the full HPC optimization loop (profiling-guided tuning, performance modeling, multi-node execution, accelerator-specific optimization). To this end, we develop **HPC-AutoResearch**, a proof-of-concept system that integrates split-phase execution, rootless container isolation, tree-search-based code generation, and hierarchical memory for compiled-code research workflows. We adapt AI Scientist [1] and introduce the *Split-Phase execution model*, which divides the code-generation-to-execution sub-pipeline into five isolated phases enabling stage-wise failure isolation and iterative retry. We further introduce Singularity container-based environment isolation for rootless dependency management, and a compressed inference memory mechanism that records prior errors and recovery attempts, allowing the system to reduce repeated failures across iterations.

The contributions of this study are:

1. **Problem identification**: We identify failure propagation in compiled-code research workflows as a key obstacle for autonomous research systems.
2. **Split-Phase execution framework**: We propose a split-phase execution framework that isolates failures across planning, environment setup, code generation, compilation, and execution.
3. **Proof-of-concept implementation**: We implement a proof-of-concept system that integrates container-based execution, a three-store hierarchical memory extending MemGPT [4], and iterative retry mechanisms.
4. **Case study**: We demonstrate that the proposed design can execute an end-to-end compiled-code workflow more robustly than a general-purpose coding agent, with hierarchical memory reducing repeated code generation errors in our case study on the Himeno Benchmark.

2. Proposed Method

2.1. Challenges of LLM-Based Code Generation for HPC

AI Scientist [1] automates the research cycle through LLM-based code generation and Best-First tree search from AIDE [3]. However, it is designed for Python-based ML experiments. Applying LLM-based code generation to HPC reveals fundamental challenges:

1. **Multi-stage failure propagation**: Compiled languages require an environment setup → compilation → execution pipeline. A single error in LLM-generated code (e.g., a missing compiler flag or incorrect library linkage) terminates the entire process.
2. **Environment constraints**: Shared HPC clusters require rootless container isolation and job scheduler interaction, which existing LLM-based code generation systems do not sufficiently support.
3. **Lack of cross-phase memory**: Without structured memory, the LLM repeats the same code generation errors across tree-search nodes.

Table 1 compares HPC-AutoResearch with AI Scientist.

Table 1
Comparison of autonomous research systems.

	AI Scientist [1]	HPC-AutoResearch
Target domain	ML	General (incl. HPC)
Languages	Python	C/C++/Fortran/CUDA
Exec. model	Tree search	Split-Phase + tree
Env. isolation	None	Singularity
Dep. resolution	Pre-installed	Iterative ReAct
Memory	Parent only	Core/Recall/Archival

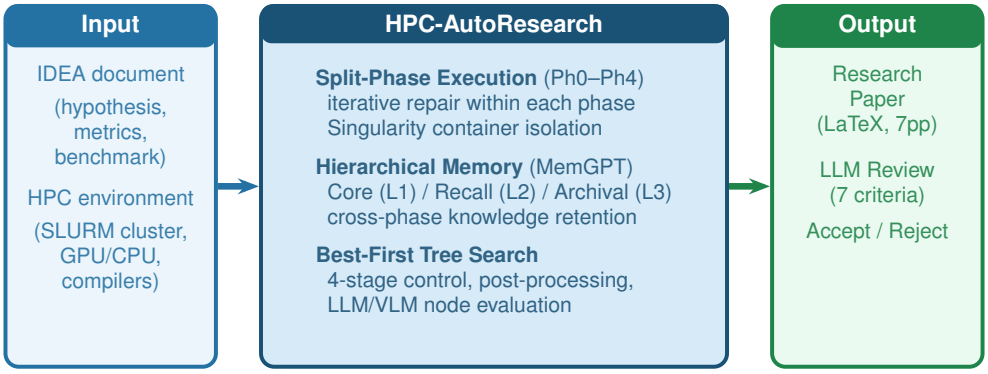

Figure 1: Overview of HPC-AutoResearch. The system inherits theme generation and Best-First tree search from AI Scientist, while introducing Split-Phase execution (Phases 0–4) and compressed inference memory.

2.2. Split-Phase Execution Model

The Split-Phase execution model divides the code generation and execution pipeline into five phases, each with failure localization and iterative repair. Figure 1 shows the system architecture, and Figure 2 shows the phase flow.

All phases that require command execution run inside Apptainer [5] (formerly Singularity) containers, which support rootless execution on shared clusters without requiring root privileges, unlike conventional Docker deployments commonly restricted on multi-tenant HPC systems. Dependencies built in Phase 1 are baked into per-worker SIF images, eliminating reinstallation in subsequent phases.

- **Phase 0 (Planning)**: Environment information (CPU/GPU architecture, NUMA topology, compiler versions, job scheduler configuration) is automatically collected within the container. Based on the collected information, guidance for subsequent code generation phases is produced.
- **Phase 1 (Environment Setup)**: Iterative dependency resolution based on the ReAct pattern [6]. The LLM generates and executes installation commands, attempting alternative approaches upon failure. Up to 100 iterations are possible, supporting package installation, source builds, and data retrieval.
- **Phase 2 (Coding)**: The LLM generates multiple source files as structured JSON output. Multi-file configurations including header files and Makefiles are supported. The structured output format ensures that the file system layout is explicitly controlled.
- **Phase 3 (Compilation)**: The generated code is built using gcc/g++/nvcc. Compiler error messages are parsed and saved to memory, enabling targeted fixes in debug nodes. This phase is skipped for interpreted languages.
- **Phase 4 (Execution)**: The program is executed, and output is collected upon completion. Timeout handling and error detection are built in.

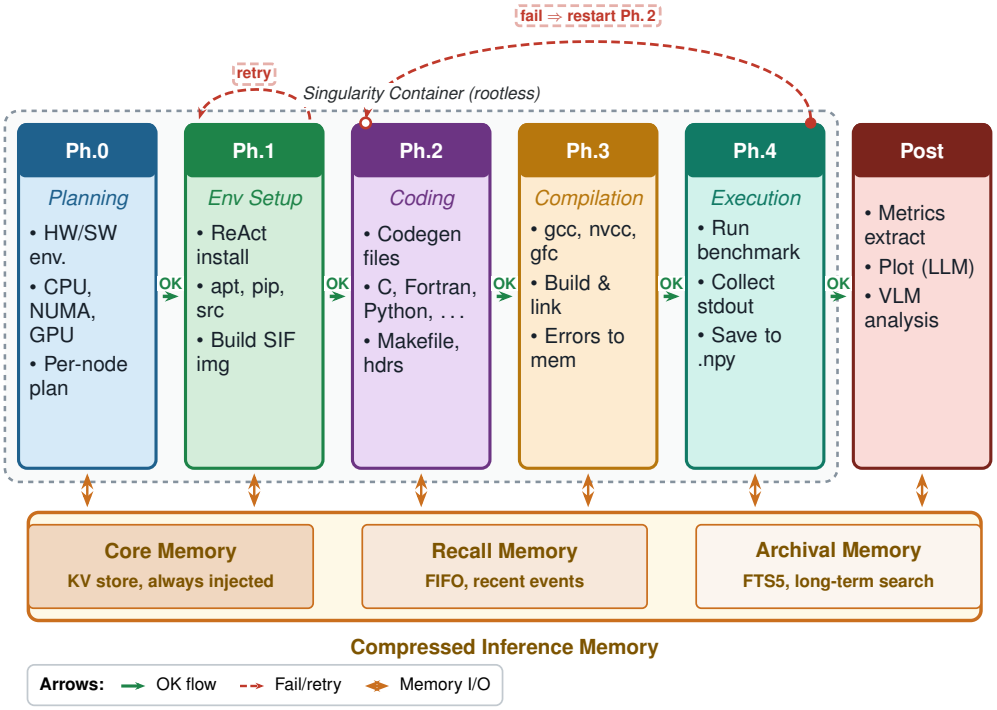

Figure 2: Split-Phase execution model. Phases 0–4 form an isolated, iteratively repairable pipeline inside a Singularity container. The compressed inference memory is queried and updated at every phase.

2.3. Compressed Inference Memory

To retain knowledge across the five phases, we adopt a three-layer memory design extending MemGPT [4]:

- **Core Memory**: A key-value store always injected into the LLM's context. It holds current phase goals, critical facts (e.g., environment profile), and accumulated code generation best practices.
- **Recall Memory**: A FIFO queue retaining recent phase events so that immediately preceding failure patterns are available without full-context injection.
- **Archival Memory**: A long-term store with full-text search (SQLite FTS5). Code snapshots, error logs, and derived insights are saved with tags.

The HPC-specific extensions over vanilla MemGPT are: (1) Copy-on-Write inheritance across tree-search nodes, preventing failed code generation in one branch from polluting siblings; (2) automatic error-triggered Archival retrieval—when a non-zero exit code is observed, keywords from the error message are used to search Archival Memory, and the top-k ($k = 3$) matching entries are injected as context for the LLM's next code generation attempt; and (3) a four-level pressure management system designed for unattended batch execution.

Each node in the tree search maintains an independent memory branch. Child branches inherit the ancestor chain but writes are reflected only in their own branch (Copy-on-Write). This balances search diversity with knowledge reuse: successful code generation strategies propagate downward, while failed attempts do not pollute siblings.

2.4. Supporting Mechanisms

- **Best-First Tree Search**: Inherited from AI Scientist; Split-Phase execution is applied at each node. The search proceeds through four stages: initial implementation, baseline

tuning, creative improvement, and ablation. Debug nodes are generated with probability p_{debug} when a node fails, enabling iterative repair of LLM-generated code.
- **Resource File Mechanism**: External resources (datasets, templates) are declaratively defined in JSON. Each resource entry specifies a type, content summary for prompt injection, and a mount path within the container.
- **Generalized LLM Review**: The ML-specific review prompts from AI Scientist are rewritten to domain-agnostic language, enabling fair automated evaluation of HPC research papers.

3. Evaluation Experiments

3.1. Experiment Setup

We conducted a comparative experiment involving a performance analysis task based on the Himeno Benchmark [7]. It is designed for performance evaluation of incompressible fluid analysis codes, measuring the processing speed of the main loop when solving the Poisson equation using the Jacobi iterative method. It is a well-known benchmark in the HPC performance-engineering community, and its performance analysis requires iterative tuning of compiler flags, parallelization strategies, and memory access patterns. This makes it suitable for evaluating the effectiveness of LLM-based code generation and iterative repair mechanisms. The experimental task requires LLM-based code generation for performance measurement, statistical analysis, and visualization. Our evaluation is intended as a proof-of-concept demonstration of feasibility, rather than a comprehensive performance comparison across diverse HPC workloads.

For evaluation, we performed LLM-based review using the pos-bias mode of the prior AI Scientist [1]. The evaluation targets are the proposed method (HPC-AutoResearch) and three general-purpose coding agents. The LLM-based review rates seven criteria—originality, quality, clarity, significance, soundness, presentation, and contribution—on a 4-point scale, and produces an Overall score (out of 10) and an Accept/Reject decision.

The four conditions are:

- **Proposed (HPC-AutoResearch)**: All proposed features. LLM: OpenAI `gpt-5.2`.
- **Claude Code**: Anthropic's CLI agent (`claude-opus-4-6`).
- **Codex CLI**: OpenAI's Codex CLI agent (`gpt-5.2-codex`).
- **Gemini CLI**: Google's Gemini CLI agent (`gemini-3-pro-preview`).

All experiments were conducted in February 2026, when all model endpoints above were publicly available (notably, `gemini-3-pro-preview` was discontinued on March 26, 2026). All conditions used the same HPC environment (Dual-socket AMD EPYC 9554, 128 cores, 2 NUMA nodes, Ubuntu 22.04) and the Himeno Benchmark C source code. The proposed method takes only an IDEA document (research hypothesis, ∼900 words) as input; general-purpose agents received both IDEA and TASK (experimental procedure, ∼500 words) documents, with procedures explicitly instructed. For fair comparison, all outputs were processed through the same post-processing pipeline (metrics extraction, plot generation, paper generation) and evaluated by generalized LLM review.

The Best-First tree search parameters: maximum 40 iterations per stage across 4 stages, draft count 3, maximum debug depth 3, debug probability 0.5, and 4 parallel workers. For simplicity, all LLMs were called with temperature 0.7; the proposed method used snapshot `gpt-5.2-2025-12-11`. A *node success* is defined as a node whose Phase 4 execution terminates with exit code 0 and produces parseable numeric output matching the expected metric schema.

3.2. Results

The proposed method, Claude Code, and Gemini CLI completed all stages. Codex failed to adapt to SLURM job submission at environment setup and could not obtain results.

Table 2
Comparison of LLM review scores (pos bias mode).

Criterion	Proposed	Proposed (w/o mem)	Claude	Codex	Gemini
Originality	3/4	3/4	2/4	2/4	2/4
Quality	3/4	3/4	3/4	1/4	2/4
Clarity	3/4	3/4	3/4	2/4	2/4
Significance	3/4	3/4	2/4	2/4	3/4
Soundness	3/4	3/4	3/4	1/4	3/4
Presentation	3/4	3/4	3/4	2/4	2/4
Contribution	3/4	3/4	2/4	1/4	3/4
Overall	7/10	7/10	6/10	3/10	7/10
Decision	Accept	Accept	Accept	Reject	Accept
Node success	47.0%	40.5%	—	—	—

Table 3
Metric availability per stage: with vs. without memory.

Stage	Cond.	Median	CV	p99/med	Pareto
St.1	W/ mem	✓	✓	✓	✓
	W/o mem	✓	✓	✓	–
St.2	W/ mem	✓	✓	✓	✓
	W/o mem	✓	△	✓	–
St.3	W/ mem	✓	✓	✓	✓
	W/o mem	✓	✓	✓	–
St.4	W/ mem	✓	✓	✓	✓
	W/o mem	✓	–	–	–

✓: present – : absent △: present but reinvented methodology

We use LLM-based review as a coarse-grained indicator of output quality, not as a substitute for expert evaluation. Table 2 shows the resulting scores. The proposed method was the only condition to achieve 3/4 on all seven criteria. Although Gemini CLI matched the Overall score (7/10), it fell short in originality and clarity. Codex received a Reject decision due to environment setup failure. These scores should be interpreted with caution given the coarse 4-point scale and the absence of human validation; a more rigorous evaluation with human expert review and multiple benchmarks remains an important direction for future work.

3.3. Effect of Compressed Inference Memory on Code Generation

The ablation run without memory achieved identical review scores but exhibited higher code generation failure rates: overall node success rate dropped from 47.0% to 40.5%, with Stage 2 (baseline tuning) falling from 54.2% to 29.2%. Without memory, the LLM repeated the same code generation errors (path mismatches, schema violations) across nodes.

Table 3 shows metric availability per stage. The memory-enabled run maintained consistent measurement methodology across all stages, while the no-memory condition independently reinvented its code structure at each stage—Pareto hypervolume was never computed, and Stage 4 omitted key metrics entirely. This demonstrates that cross-phase memory enables the LLM to maintain consistent code generation conventions.

3.4. Discussion

3.4.1. Effectiveness of Split-Phase Execution for Code Generation

The contrast between Codex and the proposed method is striking. Codex's LLM failed to generate correct SLURM job scripts at the environment setup stage, and without failure isolation, all subsequent code generation became inoperable. In contrast, Phase 1's iterative ReAct-based

repair autonomously resolved similar errors, demonstrating that phase-isolated failure recovery is essential for LLM-based code generation in HPC environments.

3.4.2. Compiled Language Code Generation

A fundamental motivation is that HPC numerical computation is overwhelmingly performed in compiled languages; Python-based simulation is practically nonexistent at scale. The Split-Phase model addresses this by explicitly incorporating compilation as a separate phase with its own failure localization. The current experiment uses C; extending to Fortran or CUDA is architecturally possible since the model abstracts the compile-execute pipeline, though these languages introduce distinct failure modes (e.g., implicit interfaces, kernel launch errors) that remain to be validated.

3.4.3. Limitations

This work focuses on stabilizing the sub-pipeline from code generation to executable results. Full HPC optimization workflows often include profiling-guided tuning, performance modeling, multi-node execution, accelerator-specific optimization, and iterative refinement based on measured performance. Extending the proposed framework to such full optimization cycles is left for future work. Limitations of the present evaluation include dependence on the underlying LLM's capabilities (the proposed method uses `gpt-5.2` while competitors use different model families; isolating model contribution from system design requires further ablation), single-run statistics without multi-seed replication (reported success rates are from ≈160 node evaluations per condition), and evaluation on a single benchmark with automated LLM review only—the automated review is a black-box function whose validity as a substitute for expert human review has not been established. A scaffolded baseline providing agents with pre-written job templates is planned as future work. All prompts and generated papers are publicly available on Zenodo[1].

3.5. Security Considerations

Allowing an LLM to generate and execute arbitrary commands on an HPC cluster raises safety and security concerns directly relevant to generative code intelligence. HPC-AutoResearch mitigates these through three mechanisms: (1) *Apptainer sandboxing*—all LLM-generated commands execute inside a user-namespace container with no host root access, limiting the blast radius of malicious or erroneous installations; (2) *network isolation*—Singularity containers can optionally run with `-net none`, restricting supply-chain attacks via packages; (3) *resource quotas*—SLURM resource limits prevent runaway jobs from consuming shared resources. However, these mitigations are not exhaustive. The current system does not perform static analysis on LLM-generated code before execution, does not validate cryptographic signatures of downloaded dependencies, and does not enforce least-privilege network policies beyond optional flags. Future work should integrate static taint analysis or sandboxed pre-execution validation of generated code, aligning with the broader goal of ensuring safety of AI-generated code in production environments.

4. Related Work

LLM-Based Code Generation for HPC. HPC-Coder [8] fine-tunes LLMs on HPC and scientific codes, demonstrating improved parallel-code completion and OpenMP pragma generation over generic models. Godoy et al. [9] systematically evaluated LLM code generation for HPC kernels. LM4HPC [10] provides an evaluation framework for HPC-specific tasks. HPC-GPT [11] is fine-tuned on HPC data. chatHPC [12] builds conversational agents for HPC platforms. VibeCodeHPC [13] applies multi-agent LLM prompting to HPC code auto-tuning, but does not

[1] https://zenodo.org/records/18518994

address end-to-end autonomous research paper generation or split-phase execution with persistent memory. These systems address code generation or user assistance rather than end-to-end research automation.

Research Automation Systems. AI Scientist [1] automates the full research cycle and achieved peer-reviewed workshop acceptance, while Agent Laboratory [2] uses a phased architecture for ML experimentation; both have primarily been demonstrated in Python-based environments. Google AI Co-Scientist [14] focuses on biomedical hypothesis generation and candidate prioritization, with experimental validation performed outside the autonomous agent loop.

Memory and Tree Search for LLM Agents. MemGPT [4] applies virtual memory concepts to LLMs. LATS [15] applies Monte Carlo Tree Search to LLM agents. SWE-Search [16] integrates MCTS with software agents. CodeTree [17] uses tree search for code generation. Our work extends MemGPT to three tiers with Copy-on-Write inheritance and applies Split-Phase execution at each tree node.

Code Intelligence Agents. SWE-agent [18] and OpenHands [19] target software engineering but not compiled HPC workflows. AIDE [3] provides tree-search-based code generation for ML. Our contribution is the integration of phase-isolated code generation, compressed inference memory, and container-based isolation into a single pipeline for autonomous HPC experimentation.

5. Conclusion

We presented the design and an initial case study of HPC-AutoResearch, a proof-of-concept system that integrates split-phase execution, rootless container isolation, tree-search-based code generation, and hierarchical memory for autonomous execution of compiled-code research workflows. The central message is that, for autonomous research systems, LLM-based code generation alone is insufficient: the surrounding workflow must localize, absorb, and retry failures across stages. The Split-Phase execution model divides the code-generation-to-execution subpipeline into five isolated phases enabling localized failure recovery, and the compressed inference memory retains cross-phase knowledge to reduce repeated code generation errors.

In a proof-of-concept evaluation on the Himeno Benchmark, the Split-Phase model enabled full-pipeline completion while a general-purpose agent (Codex) failed at environment setup, and the memory mechanism improved node success rate from 40.5% to 47.0%. These results are indicative rather than conclusive: they are based on a single benchmark, a single run, and automated review only, and they do not address the full HPC optimization loop.

Future work includes scaffolded baselines for fair comparison, application to diverse HPC workloads (MPI, CUDA, Fortran), multi-seed replication with statistical testing, fine-grained memory ablations, and evaluation with human HPC experts. The source code is available at https://github.com/kotama7/HPC-AutoResearch.

Acknowledgments

This study was supported by the JST Next-Generation Edge AI Semiconductor Research and Development Project JPMJES2511 and the Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures (JHPCN) (jh250015).

Declaration on Generative AI

During the preparation of this work, the authors used Claude (Anthropic) in order to: Drafting content, Generate images (figures), Text translation, and assistance in implementing the software presented in this work. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] C. Lu, C. Lu, R. T. Lange, Y. Yamada, S. Hu, J. Foerster, D. Ha, J. Clune, Towards end-to-end automation of AI research, Nature 651 (2026) 914–919. doi:10.1038/s41586-026-10265-5.

[2] S. Schmidgall, Y. Su, Z. Wang, X. Sun, J. Wu, X. Yu, J. Liu, M. Moor, Z. Liu, E. Barsoum, Agent laboratory: Using LLM agents as research assistants, in: C. Christodoulopoulos, T. Chakraborty, C. Rose, V. Peng (Eds.), Findings of the Association for Computational Linguistics: EMNLP 2025, Association for Computational Linguistics, Suzhou, China, 2025, pp. 5977–6043. URL: https://aclanthology.org/2025.findings-emnlp.320/. doi:10.18653/v1/2025.findings-emnlp.320.

[3] Z. Jiang, D. Schmidt, D. Srikanth, D. Xu, I. Kaplan, D. Jacenko, Y. Wu, Aide: Ai-driven exploration in the space of code, 2025. URL: https://arxiv.org/abs/2502.13138. arXiv:2502.13138, arXiv:2502.13138.

[4] C. Packer, S. Wooders, K. Lin, V. Fang, S. G. Patil, I. Stoica, J. E. Gonzalez, Memgpt: Towards llms as operating systems, 2024. URL: https://arxiv.org/abs/2310.08560. arXiv:2310.08560, arXiv:2310.08560.

[5] G. M. Kurtzer, V. Sochat, M. W. Bauer, Singularity: Scientific containers for mobility of compute, PLOS ONE 12 (2017) 1–20. URL: https://doi.org/10.1371/journal.pone.0177459. doi:10.1371/journal.pone.0177459.

[6] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, Y. Cao, React: Synergizing reasoning and acting in language models, 2023. Publisher Copyright: © 2023 11th International Conference on Learning Representations, ICLR 2023. All rights reserved.; 11th International Conference on Learning Representations, ICLR 2023 ; Conference date: 01-05-2023 Through 05-05-2023; arXiv:2210.03629.

[7] R. Himeno, Himeno benchmark, https://i.riken.jp/en/supercom/documents/himenobmt/, 1998. Originally released 1998, last upgraded 2001; accessed 2026-05-11.

[8] D. Nichols, A. Marathe, H. Menon, T. Gamblin, A. Bhatele, Hpc-coder: Modeling parallel programs using large language models, in: ISC High Performance 2024 Research Paper Proceedings (39th International Conference), 2024, pp. 1–12. doi:10.23919/ISC.2024.10528929.

[9] W. Godoy, P. Valero-Lara, K. Teranishi, P. Balaprakash, J. Vetter, Large language model evaluation for high-performance computing software development, Concurrency and Computation: Practice and Experience 36 (2024). doi:10.1002/cpe.8269, publisher Copyright: © 2024 John Wiley & Sons Ltd.

[10] L. Chen, P.-H. Lin, T. Vanderbruggen, C. Liao, M. Emani, B. de Supinski, Lm4hpc: Towards effective language model application in high-performance computing, in: OpenMP: Advanced Task-Based, Device and Compiler Programming: 19th International Workshop on OpenMP, IWOMP 2023, Bristol, UK, September 13–15, 2023, Proceedings, Springer-Verlag, Berlin, Heidelberg, 2023, p. 18–33. URL: https://doi.org/10.1007/978-3-031-40744-4_2. doi:10.1007/978-3-031-40744-4_2.

[11] X. Ding, L. Chen, M. Emani, C. Liao, P.-H. Lin, T. Vanderbruggen, Z. Xie, A. Cerpa, W. Du, Hpc-gpt: Integrating large language model for high-performance computing, in: Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W '23, Association for Computing Machinery, New York, NY, USA, 2023, p. 951–960. URL: https://doi.org/10.1145/3624062.3624172. doi:10.1145/3624062.3624172.

[12] J. Yin, J. Hines, E. Herron, T. Ghosal, H. Liu, S. Prentice, V. Lama, F. Wang, chatHPC: Empowering HPC users with large language models, J. Supercomput. 81 (2024). URL: https://doi.org/10.1007/s11227-024-06637-1. doi:10.1007/s11227-024-06637-1.

[13] S. ichiro Hayashi, K. Morita, D. Mukunoki, T. Hoshino, T. Katagiri, Vibecodehpc: An agent-based iterative prompting auto-tuner for hpc code generation using llms, 2026. URL: https://arxiv.org/abs/2510.00031. arXiv:2510.00031, arXiv:2510.00031.

[14] J. Gottweis, W.-H. Weng, A. Daryin, T. Tu, A. Palepu, P. Sirkovic, A. Myaskovsky, F. Weissenberger, K. Rong, R. Tanno, K. Saab, D. Popovici, J. Blum, F. Zhang, K. Chou, A. Hassidim, B. Gokturk, A. Vahdat, P. Kohli, Y. Matias, A. Carroll, K. Kulkarni, N. Tomasev, Y. Guan, V. Dhillon, E. D. Vaishnav, B. Lee, T. R. D. Costa, J. R. Penadés, G. Peltz, Y. Xu, A. Pawlosky, A. Karthikesalingam, V. Natarajan, Towards an ai co-scientist, 2025. URL: https://arxiv.org/abs/2502.18864. arXiv:2502.18864, arXiv:2502.18864.

[15] A. Zhou, K. Yan, M. Shlapentokh-Rothman, H. Wang, Y.-X. Wang, Language agent tree search unifies reasoning, acting, and planning in language models, in: Proceedings of the 41st International Conference on Machine Learning, ICML'24, JMLR.org, 2024. ArXiv:2310.04406.

[16] A. Antoniades, A. Örwall, K. Zhang, Y. Xie, A. Goyal, W. Wang, Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement, 2025. URL: https://arxiv.org/abs/2410.20285. arXiv:2410.20285, arXiv:2410.20285.

[17] J. Li, H. Le, Y. Zhou, C. Xiong, S. Savarese, D. Sahoo, CodeTree: Agent-guided tree search for code generation with large language models, in: L. Chiruzzo, A. Ritter, L. Wang (Eds.), Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), Association for Computational Linguistics, Albuquerque, New Mexico, 2025, pp. 3711–3726. URL: https://aclanthology.org/2025.naacl-long.189/. doi:10.18653/v1/2025.naacl-long.189.

[18] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, O. Press, Swe-agent: agent-computer interfaces enable automated software engineering, in: Proceedings of the 38th International Conference on Neural Information Processing Systems, NIPS '24, Curran Associates Inc., Red Hook, NY, USA, 2024. ArXiv:2405.15793.

[19] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, G. Neubig, Openhands: An open platform for AI software developers as generalist agents, in: The Thirteenth International Conference on Learning Representations, 2025. URL: https://openreview.net/forum?id=OJd3ayDDoF, arXiv:2407.16741.