Improving Usability and Productivity of PETSc with Agent-Based Workflows

BARRY SMITH, HONG ZHANG, JUNCHAO ZHANG, SATISH BALAY, LE CHEN, MURAT KEÇELI, and LOIS CURFMAN MCINNES, Argonne National Laboratory, USA

Scientific computing software such as PETSc embodies deep expertise in numerical methods, solver configuration, and scalable implementation, yet this knowledge remains difficult for both users and large language models (LLMs) to access and apply effectively. As a result, even experienced researchers spend significant time selecting solvers, debugging configurations, and validating results. In this work, we present initial experiences in developing an AI-assisted, agent-based ecosystem to improve PETSc usability and productivity for scientific applications. Our approach integrates retrieval-augmented generation (RAG), grounded in PETSc manual pages and curated documentation, with modular services for code generation, compilation, execution, and validation, all exposed through lightweight agent interfaces. We position PETSc as a domain-aware component in multi-step AI workflows that span question answering, code development, execution, and verification. We describe the infrastructure and prototype services that support these capabilities and outline their potential to enable more effective, reliable, and scalable AI-assisted workflows in scientific computing.

CCS Concepts: • **Software and its engineering → Software libraries and repositories**; • **Mathematics of computing → Solvers**; • **Computing methodologies → Planning and scheduling**.

Additional Key Words and Phrases: Agents, PETSc

ACM Reference Format:

Barry Smith, Hong Zhang, Junchao Zhang, Satish Balay, Le Chen, Murat Keçeli, and Lois Curfman McInnes. 2026. Improving Usability and Productivity of PETSc with Agent-Based Workflows. 1, 1 (April 2026), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Complex hierarchical numerical simulation software systems are used daily across science and engineering. Their use requires deep expertise in mathematical modeling, numerical algorithms, and software engineering. Such software often has thousands of API entry points and (functions, data structures, class definitions, etc.), and the underlying mathematical models and numerical algorithms are built on thousands of abstractions, paradigms, and terminologies. Teams developing and using these simulation systems often include dozens to hundreds of contributors, each with specialized expertise. As a result, efficiency is low: adding features can take months or years; verifying correctness can take months, and debugging a single crash may take hours or days.

Recent advances in generative AI have demonstrated strong capabilities in code generation [2, 5, 8], high performance computing (HPC) software development [3], and compiler validation [4]. Agentic approaches have recently emerged as a promising paradigm for tackling complex software engineering tasks. By orchestrating multiple agents, these systems exhibit system-level code generation capabilities that go beyond what a single LLM can achieve. However, these advances also introduce new challenges in AI-assisted scientific computing. Due to a more structured and constrained problem space, naively applying general-purpose agentic pipelines in scientific computing may lead to suboptimal or even incorrect solutions. This highlights the need for domain-specific system design, where agent architectures, tool integration, and feedback mechanisms are tailored to the unique requirements of scientific software.

This work complements two other approaches for leveraging LLMs: domain-specific fine-tuning [9] and structured context (for example, skills.md files) for task guidance. We present preliminary work on a framework for model context

Authors' Contact Information: Barry Smith, bsmith@petsc.dev; Hong Zhang, hongzhang@anl.gov; Junchao Zhang, jczhang@anl.gov; Satish Balay, balay@anl.gov; Le Chen, lechen@anl.gov; Murat Keçeli, keceli@anl.gov; Lois Curfman McInnes, mcinnes@anl.gov, Argonne National Laboratory, Argonne, IL, USA.

protocol (MCP) services (also represented as Agent2Agent, A2A, agents) to support the use and development of the Portable Extensible Toolkit for Scientific computing (PETSc) numerical solver library [1]. Our goal is to expose PETSc's knowledge and capabilities through modular services that can be used directly by LLMs and composed into agentic workflows.

We describe the technologies used to implement these services and the common software infrastructure we are developing to reduce maintenance costs. MCP services include both simple tools and LLM-based capabilities, and some function as A2A-style agents exposed through MCP interfaces. The infrastructure enables flexible composition and reuse across workflows. These services may incorporate advanced technologies such as code verification tools and numerical algorithm verification methods, including grid convergence analysis, the method of manufactured solutions, and stability analysis. While commonly used by experts, they are often applied inconsistently due to workflow overhead. Exposing them as services enables more systematic use within LLM-driven workflows. We believe our agentic system will lead to systematic, standardized use of validation processes that are too often currently skipped. Although rapid advances in LLM ecosystems may subsume some capabilities, domain-specific infrastructure remains essential for reliable scientific workflows.

This paper is organized as follows. We present the MCP infrastructure design, followed by current services and methods for quantitative evaluation. In scientific computing, anecdotal evaluation is insufficient; results must be established through rigorous, multi-faceted analysis.

2 Design

The primary options for implementing MCP servers for scientific computing are TypeScript and Python, both with mature and rapidly evolving ecosystems. We adopt Python FastMCP as our infrastructure because it offers more functionality than the MCP Python module and we have no experience with TypeScript.

For deployment simplicity, each MCP server runs on its own HTTP(S) port. For development and debugging, we support the MCP "stdio" protocol. We also provide an automatically generated command-line client for each server to facilitate testing and tutorials. The handling of server tools within these clients is fully automated, as described below.

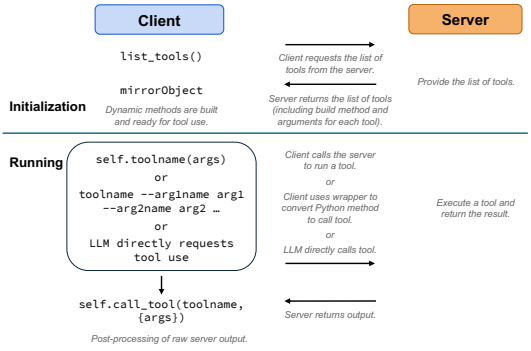

Fig. 1. PETSc MCP agents: Dynamic mirroring of server tools into command line or client methods.

MCP servers can be accessed in two primary ways. First, they can be invoked directly by an LLM, which selects a tool (i.e., a server function) and constructs its arguments. In Python, this interaction is supported through the "call_tool()" API. However, when used directly in Python code, this API results in difficult-to-read code. This limitation is problematic

because our design includes hierarchical workflows in which higher-level MCP services invoke lower-level services, effectively treating MCP servers as remote procedure call (RPC) components. In these cases, clear and maintainable Python interfaces are essential, and reliance on "call_tool()" alone is insufficient.

While it is possible to manually wrap the "call_tool()" invocation with a client-side mirror, this approach does not scale. Instead, we use Python's introspection capabilities to automatically generate client-side method wrappers. When a client connects to a server and retrieves the list of available tools via "tool_list()", corresponding methods are dynamically created, enabling natural and readable Python usage. We demonstrate in Fig. 1 the mirroring concept for the Python binding and command line approaches, and we show fragments of Python code that enable the automation for the Python binding in Fig. 2. Because we anticipate maintaining multiple MCP servers, this automation is critical for scalability. While Fig. 1 shows the client using a single server, in general, a client will interact with multiple servers, as shown in Fig. 3.

```python
"""Generate mirror method for each tool"""
for tool in await self.session.list_tools():
  method = functools.partial(self.call_tool_wrapper, tool.name, tool.inputSchema)
  setattr(self, tool.name, method) # adds tool.name to the class methods
async def call_tool_wrapper(self, tool_name, inputSchema, **kwargs):
  """The internal executor for the dynamically created methods."""
  ... argument error checking based on inputSchema
  return await self.call_tool(tool_name, kwargs)
```

Fig. 2. Mirroring server tools on client.

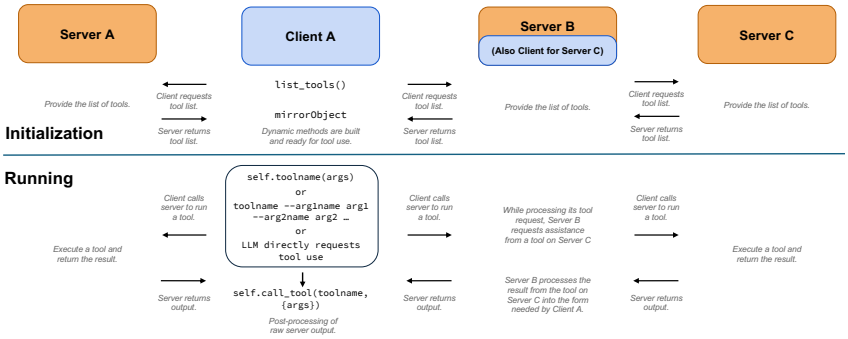

Fig. 3. Client interacting with multiple independent servers.

To remain protocol-neutral, we have augmented our MCP-based infrastructure by using the Google A2A Python module to wrap each MCP server as an A2A agent, enabling systems designed for A2A agents to easily use our MCP servers.

3 Specific Servers

We currently provide four MCP servers, https://mcp.petsc-ai.org. Discussed in Section 3.1 are a PETSc documentation server and a server that provides retrieval-augmented generation (RAG) context for PETSc users and developers. Discussed in Section 3.2 are a server that generates and tests PETSc examples, along with a server that compiles and runs PETSc programs.

```python
# Server code
@mcp.tool()
def make(executable: str = '', dependencies: str = '')-> dict:
  """Run the bash make command to ...."""
  ...
# Standard LLM style client code
session.call_tool('make', {"executable": "test", "dependencies": "test.kokkos.cxx"})
# Dynamic mirrored client code
client.make(executable = "test", dependencies= "test.kokkos.cxx")
```

Fig. 4. Example of using the mirrored client method with the standard approach and the mirror method.

3.1 Providing a gold standard for PETSc documentation/source

The web contains abundant information about PETSc. Copies of PETSc documentation from 20 years ago are hosted on some obscure university department's web server. Information is embedded directly in LLMs from two-year-old training data. Since LLM-based agents can search the web and generate new text, it is important to provide direct access to gold-standard PETSc documentation and source code. Access must be straightforward for LLMs, and it must be the first place they search before performing their tasks. Thus, we provide two MCP servers that act as front-ends to the gold-standard material. The first is a **documentation server** that returns manual pages for the entire PETSc API as well as sections from the user's guide. The second provides **RAG searches** [6] across all PETSc documentation (including a separate section that provides material useful to those developing PETSc code, such as style guides). These servers could also provide responses for particular versions of PETSc as needed. Additional servers, such as those for the source code and the PETSc publications, are also planned.

In [6], we showed that RAG with reranking can mitigate the well-known hallucination problem of LLMs. On a benchmark of 37 real PETSc user questions about Krylov methods, GPT-4o without RAG, scored poorly and hallucinated frequently. Adding RAG dramatically improved 20 of the 37 scores while decreasing only 3 scores. A key technical insight of this work is that general-purpose LLMs alone are insufficient for accurate scientific computing support, but their performance improves dramatically when grounded in the PETSc knowledge base via RAG with high-quality sources such as manual pages and curated documentation, which provide critical, authoritative signals that reduce hallucinations and enable near-expert responses. However, RAG alone is not sufficient: embedding-based retrieval can introduce noisy or partially relevant context, so adding a reranking stage is essential to filter and prioritize the most relevant information. The combination of RAG, manual-page grounding, and reranking yields consistently high-quality answers, often approaching expert-level correctness, demonstrating that effective AI for scientific software depends not just on stronger models but on carefully engineered, domain-specific knowledge pipelines.

The RAG MCP server exposes this pipeline as two tools. `get_documentation_rag_prompt` returns context from user-level documentation (manual pages, user's guide, examples); `get_developer_rag_prompt` returns context from the developer manual that provides style guides, coding conventions, design information, etc. Both tools retrieve the top k chunks by similarity from a vector database, apply a neural reranker to keep the top n ($n < k$), and return the results with PETSc-specific system instructions.

A legitimate question is whether these tools are or will remain necessary. With ever-improving LLMs and systems such as Context7 [7] that aim to provide gold-standard information, do subcommunities even need to roll their own tools? At the moment, it seems prudent for the PETSc community to curate and provide this information.

3.2 Servers for working with code

LLM-based environments that interact directly with developers, such as Claude Code, Gemini-CLI, Warp, and VS Code, have direct access to the file system, the source code, and all compiler and development tools. This is powerful and convenient because these environments can compile code, inspect compiler warnings and errors, and revise code interactively without requiring users to run compilers manually. But what if one wants to host a remote code generator system that compiles and runs code and runs sophisticated verification tools to ensure code correctness before returning the verified code to the calling client, which may itself be another agent?

To support the development of such MCP servers, we have developed a **PETSc-specific "compile-run" server** that includes PETSc and compilers, etc. (perhaps hosted in the cloud). The **code generation server** (which may also run in the cloud) uses an iterative loop to prompt an LLM to generate code, then dispatches compilation and execution to the compiler-run server. This design "separates the concerns" of the mechanics of compiling and running code from the concerns of generating code. The compile-run server could, and likely would, serve as the backend for multiple code-generation servers.

In addition, as a proof of principle, we provide an LLM-based code generator that asks Claude to generate code and use the compile-run server to verify that the code compiles and runs. If the code does not compile or run, the Claude-based agent iteratively updates the code and tries again. This demonstrates how multiple agents manage end-to-end complexity by separating concerns into distinct subprocesses.

A representative use of the PETSc compile-run MCP server is PETScAgent-Bench [10], an agentic benchmark for evaluating AI-generated PETSc programs. Each benchmark problem is given as a natural-language description. The current problem set covers a range of typical PETSc workloads, including stiff ODE systems, advection and Darcy-flow PDEs, incompressible Navier–Stokes, and Rosenbrock optimization.

A *code-generation agent* receives a problem description and returns one or more source files, the number of MPI processes to use, and command-line arguments for the PETSc solver. A separate *evaluator agent* takes these outputs and uses the compile-run MCP server to assess them. The evaluator agent checks whether the code compiled, whether it ran without errors, and compares its numerical output against reference values, and passes the source code to further evaluators that assess code quality and PETSc usage. For example, a generated program that omits a required PETSc header or misconfigures a solver produces a specific compiler or runtime diagnostic that the evaluator can record and attribute, rather than reporting only a pass or fail.

The MCP server detaches the build and execution logic from the benchmark itself. Wrapping this logic in an MCP server makes it available as a reusable service that any agent can call; the service could also be used by a coding assistant, a debugging agent, or a tutorial system through the same interface. The same pattern also extends to iterative workflows where an agent generates code, calls the compile-run server to test it, reads the diagnostics, and refines its solution in a loop.

4 Conclusion

In this paper, we discuss preliminary work exploring agentic tooling to help LLM-based systems improve the development and use of the PETSc numerical solver library. We have developed a common infrastructure for deploying MCP servers/A2A agents that support and extend LLM-based coding agents for PETSc development and usage. We believe that such agents, in combination with skills files and possibly fine-tuned LLM models, are crucial for leveraging AI

in scientific computing. Finally, we briefly discussed prior work that quantitatively demonstrated the usefulness and added value of our tooling infrastructure. Future work will build on these foundations for more advanced functionality.

Acknowledgments

This material is based upon work partially supported by Laboratory Directed Research and Development (LDRD) funding from Argonne National Laboratory, provided by the Director, Office of Science, of the U.S. Department of Energy (DOE) under Contract No. DE-AC02-06CH11357. This material is based upon work partially supported by the U.S. DOE, Office of Science, Office of Advanced Scientific Computing Research, by the Competitive Portfolios for Advanced Scientific Computing Research Program, and by the Scientific Discovery through Advanced Computing (SciDAC) Program through the FASTMath Institute. Research was partially supported by the U.S. DOE Office of Science Distinguished Scientist Fellows Program. This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory.

References

[1] S. Balay et al. 2025. *PETSc Users Manual.* Technical Report ANL-21/39 - Revision 3.23.3. Argonne National Laboratory. https://petsc.org

[2] Le Chen, Arijit Bhattacharjee, Nesreen Ahmed, Niranjan Hasabnis, Gal Oren, Vy Vo, and Ali Jannesari. 2024. OMPGPT: A generative pre-trained transformer model for OpenMP. In *European Conference on Parallel Processing*. Springer, 121–134.

[3] William F Godoy, Pedro Valero-Lara, Keita Teranishi, Prasanna Balaprakash, and Jeffrey S Vetter. 2024. Large language model evaluation for high-performance computing software development. *Concurrency and Computation: Practice and Experience* 36, 26 (2024), e8269.

[4] Christian Munley, Aaron Jarmusch, and Sunita Chandrasekaran. 2024. LLM4VV: Developing LLM-driven testsuite for compiler validation. *Future Generation Computer Systems* 160 (2024), 1–13.

[5] Daniel Nichols, Aniruddha Marathe, Harshitha Menon, Todd Gamblin, and Abhinav Bhatele. 2024. HPC-Coder: Modeling parallel programs using large language models. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*. Prometeus GmbH, 1–12.

[6] Barry Smith, Junchao Zhang, Hong Zhang, Lois Curfman McInnes, Murat Keceli, Archit Vasan, Satish Balay, Toby Isaac, Le Chen, and Venkatram Vishwanath. 2025. AI Assistants to Enhance and Exploit the PETSc Knowledge Base. In *Workshop Proceedings of the 54th International Conference on Parallel Processing (ICPP Workshops '25)*. Association for Computing Machinery, New York, NY, USA, 9–17. doi:10.1145/3750720.3757281

[7] Upstash Team. 2026. Context7: Up-to-date code documentation for LLMs and AI code editors. https://context7.com/. Accessed: 2026-03-30.

[8] Pedro Valero-Lara, Alexis Huante, Mustafa Al Lail, William F. Godoy, Keita Teranishi, Prasanna Balaprakash, and Jeffrey S. Vetter. 2023. Comparing Llama-2 and GPT-3 LLMs for HPC kernels generation. arXiv:2309.07103 [cs.SE] https://arxiv.org/abs/2309.07103

[9] Pedro Valero Lara, Aaron Young, Jeffrey S Vetter, Zheming Jin, Swaroop Pophale, Mohammad Alaul Haque Monil, Keita Teranishi, and William F Godoy. 2025. ChatHPC: Building the Foundations for a Productive and Trustworthy AI-Assisted HPC Ecosystem. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 458–474.

[10] Hong Zhang, Barry Smith, Satish Balay, Le Chen, Murat Keceli, Lois Curfman McInnes, and Junchao Zhang. 2026. An Agentic Evaluation Framework for AI-Generated Scientific Code in PETSc. arXiv:2603.15976 [cs.AI] https://arxiv.org/abs/2603.15976