LLMoxie: Exploring Agentic AI for Scientific Software Development

Landung Setiawan
eScience Institute
University of Washington
Seattle, Washington, USA
landungs@uw.edu

Anant Mittal
eScience Institute
University of Washington
Seattle, Washington, USA
anmittal@uw.edu

Cordero Core
eScience Institute
University of Washington
Seattle, Washington, USA
cdcore@uw.edu

Anshul Tambay
eScience Institute
University of Washington
Seattle, Washington, USA
anshul37@uw.edu

Carlos Garcia Jurado Suarez
eScience Institute
University of Washington
Seattle, Washington, USA
carlosg@uw.edu

David Beck
eScience Institute
Dept of Chemical Engineering
University of Washington
Seattle, Washington, USA
dacb@uw.edu

Andy Connolly
eScience Institute
Dept of Astronomy
University of Washington
Seattle, Washington, USA
ajc@astro.washington.edu

Vani Mandava
eScience Institute
University of Washington
Seattle, Washington, USA
vani1@uw.edu

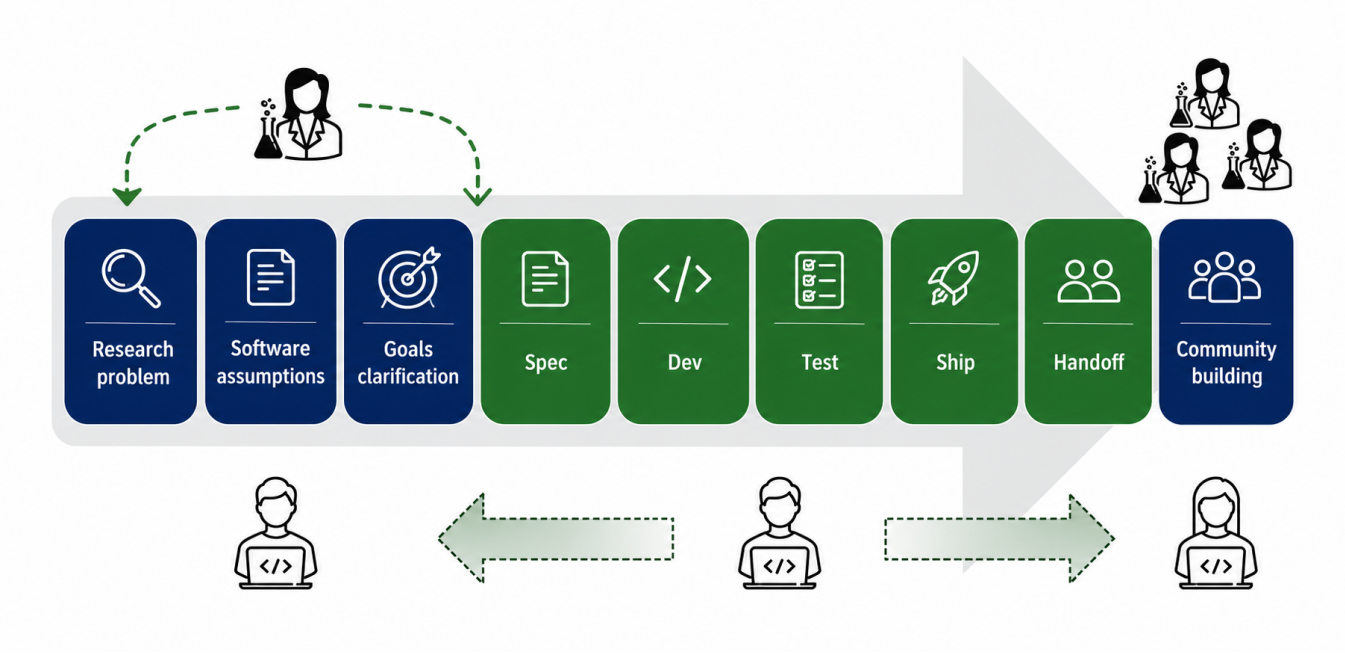

Figure 1: An RSE-oriented scientific software development lifecycle. Conventional engineering stages (Spec, Dev, Test, Ship) are bracketed by science-specific phases: upstream framing of the research problem, software assumptions, and goals clarification, and downstream handoff and community building. Scientists drive the bookend phases while research software engineers carry the work across the middle, illustrating where AI-assisted coding agents must integrate to support scientific practice.

Abstract

In this paper, we describe *LLMoxie*, an institutional AI platform whose three-tiered architecture supports multi-cloud and on-premise inference, a LiteLLM/MLflow control plane for authentication, budgeting, PII masking, and observability, and an application augmentation layer for AI coding agents. Layered on top, an open-source *RSE-Plugins* ecosystem encodes accumulated RSE knowledge as a Plugin-Agent-Skill hierarchy spanning scientific Python practice, domain-specific knowledge, a six-phase research-and-implement workflow, and project lifecycle management. Scientific software is judged less by raw code quality than by whether it can be cited, audited, reproduced, and extended. Off-the-shelf AI coding agents, optimized against commercial software benchmarks, are poorly calibrated for this setting: they ignore the conventions of the scientific Python libraries they invoke, mishandle sensitive or embargoed data, and leave decision trails that are difficult to reconstruct after the fact. We report on twenty months of practice at a university-based research software engineering (RSE) center, where RSEs embedded across astronomy, earth and climate science, agriculture, and health projects worked to close this gap. We characterize the recurring infrastructure, governance, and process challenges of adopting Agentic AI inside a multi-domain RSE center, describe the platform and plugin design, and distill operational lessons from real scientific software deployments. Together, the platform and plugins shift AI coding agents from generic code generators into domain-aware collaborators that respect community norms and produce auditable provenance of technical reasoning.

1 Introduction

Our research software engineering center, Scientific Software Engineering Center, is a professional software engineering organization embedded within the University of Washington, Seattle. It serves as a campus-wide collaborator for domain scientists, translating research questions into durable software systems by clarifying scientific goals, designing reproducible workflows, testing underlying assumptions, and delivering maintainable tools. Since its establishment, the center has delivered more than 20 multi-institution scientific software projects, convened workshops and events reaching over 1,000 participants, sustained a graduate scholars program, and partnered with principal investigators across 32 organizations worldwide. Situated within University of Washington's data-science institute, i.e., eScience Institute whose mission centers on data-intensive science and responsible AI practice, the center's research software engineers (RSEs) work alongside research scientists and data scientists to deliver software that satisfies both engineering quality bars and the standards of scientific practice.

That dual mandate of production-quality engineering and scientific reproducibility is what makes scientific software a distinctive setting for AI-assisted development. Within professional communities that work at the intersection of software and science, there is growing recognition that the RSE profession is positioned to become central to the responsible integration of AI into research [8, 12, 21]. Unlike most commercial software, research code is judged less by its raw quality and robustness than by whether it can be cited, audited, reproduced, and extended by other scientists, often years after the original authors have moved on [13]. It typically operates over specialized data formats (e.g., `FITS`, `NetCDF`, `Zarr`, `HDF5`), relies on a tightly coupled scientific Python stack, runs on heterogeneous infrastructure ranging from laptops to high-performance computing clusters, and is written by teams in which professional software engineers are the exception rather than the rule. The sociotechnical landscape of scientific research software is complex and was not designed for engineering resilience or governance [10, 21]. Contemporary AI coding assistants, whose training data and evaluation benchmarks are dominated by commercial software development, are poorly calibrated and under-studied for research software. Specialized scientific-coding benchmarks [5, 31] have emerged in response, yet even frontier models perform poorly on them and their findings are not systematically incorporated back into frontier model development.

While postdoctoral researchers report using generative AI tools for a range of research tasks [2, 22, 23], AI-assisted research software engineering specifically remains an emergent field [3, 6]. A summer 2025 study of generative AI adoption among scientists who write code [24] found that respondents overwhelmingly prefer general-purpose, proprietary conversational tools such as ChatGPT accessed through web browsers over developer-specific tools integrated into programming environments. A second 2025 study of more than 1,500 research software repositories [6] found that only 17% exhibited both high AI usage and high software engineering maturity. These patterns have raised concern that over-reliance on general-purpose conversational models places scientific code at risk [3, 15, 25], with downstream consequences for open-science integrity and reproducibility [11].

From September 2024 to May 2026, our center engaged directly with this landscape to close the widening gap between AI-led software creation in industry and scientific software creation in academic research environments. Early efforts framed the problem as one of context: Retrieval-Augmented Generation (RAG) over scientific literature and pre-publication data [28] enabled general-purpose models to speak credibly about a given research domain, though even long-context RAG pipelines exhibit characteristic failure modes as corpora and contexts grow [4]. As frontier models improved and tool-using agents matured around the Model Context Protocol (MCP) [1, 7], the binding constraint shifted from raw model capability to the absence of structured, domain-aware context and process scaffolding around the model. Off-the-shelf coding agents produce running Python while ignoring the conventions of the communities whose libraries they invoke [3, 17], mishandle sensitive or embargoed data [11], and leave decision trails that are difficult to reconstruct after the fact [30]. This is a particularly poor

fit for scientific work, where the reasoning process is itself part of the artifact. Recent evaluations of agentic systems on realistic tasks reach a consistent conclusion: benchmark performance routinely overstates how reliably and auditably these systems behave in deployment [16, 18].

RSE centers are unusually well positioned to address this gap [8, 21]. Embedded across many scientific domains simultaneously, an RSE center observes the same failure modes recur across geoscience, astronomy, climate, agriculture, and health projects, and can therefore design a response that generalizes rather than overfits to any single laboratory. The substance of that response is to encode accumulated RSE knowledge into reusable context that AI agents can consume: packaging discipline, testing regimes for numerical code, documentation practice, reproducibility tooling, and structured research workflows [3, 13].

This paper reports on the result: **LLMoxie**, an institutional AI platform, together with **RSE-Plugins**, a hierarchical ecosystem of Claude-compatible plugins, agents, and skills that encode RSE practice as composable context for AI coding agents. LLMoxie provides a governed, multi-cloud inference layer with authentication, budgeting, PII masking, and observability; RSE-Plugins layer scientific Python conventions, domain-specific knowledge, and a structured research-and-implement workflow on top of a coding agent (Claude Code in our deployments). Together, they reposition the LLM from a generic code generator into a domain-aware collaborator that respects community norms and emits auditable trails of technical reasoning. Here we describe:

- A characterization of the recurring infrastructure, governance, and process challenges of adopting agentic AI inside an RSE center serving multiple scientific domains, drawn from a portfolio of active projects.
- The design of LLMoxie, a three-tiered platform that separates inference, governance, and application-level augmentation, with rationale for each layer (Section 3).
- RSE-Plugins, a Plugin-Agent-Skill hierarchy that encodes scientific Python practice, domain-specific knowledge, and a six-phase research-and-implement workflow as reusable context for coding agents (Section 4).
- Operational lessons from deploying this stack on real scientific software projects (Section 6).

Unlike generic model-routing and inference optimization AI infrastructure tools [19], LLMoxie shapes a broader research AI orchestration platform focused on integrating retrieval systems, scientific tools, scientific research knowledge bases, and extensible plugins into coherent research workflows. Its capabilities include tools, datasets, workflows, and execution environments with the LLM acting as only one component inside a larger research oriented ecosystem.

The remainder of the paper traces our adoption path from RAG-era experiments to the current LLMoxie design (Section 2), presents the three-tier platform and the RSE-Plugins hierarchy (Sections 3 and 4), surveys capability surfaces, production deployment, and engineering practices on active projects (Section 5), and closes with implications, limitations, and conclusions (Sections 6, 7, and 8).

2 Background & Motivation

Between September 2024 and May 2026, our center engaged with the AI-assisted RSE landscape through a portfolio of active scientific software projects. Our traditional delivery model—six-to-nine-month engagements with science principal investigators (PIs) covering requirements gathering, specification, implementation, code review, testing, documentation, and packaging—provided a structured baseline against which to assess where generative AI tooling helped, where it failed, and what institutional scaffolding was missing.

The first concrete pilot embedded scientific literature and pre-publication data into a vector database and exposed it to general-purpose models through Retrieval-Augmented Generation (RAG), in collaboration with researchers at a large astronomical survey project [28]. Consistent with our open-science mandate, the initial deployment used only open-source models olmo [9]. Once the survey data left embargo, open-weight models served through community runtimes (e.g., `ollama`) reached state-of-the-art performance on the survey's question-answering tasks, eliminating the RAG advantage for already-public corpora. RAG remained valuable, however, for sensitive and pre-publication data where the underlying corpus could not be shipped to a hosted model.

This first wave clarified that the binding constraint was not model quality but the absence of governed, science-aware infrastructure surrounding it. In mid-2024 our team was awarded federally-funded national AI compute resources [20] to build domain-agnostic AI infrastructure for scientists; the original proposal centered on hosted inference plus RAG. Over the following year, the release of the Model Context Protocol (MCP) and the rapid maturation of agent- and skill-based tooling shifted the locus of context engineering from retrieval over a single corpus to composable tools, agents, and skills coordinated by a coding agent. In response to changing landscapes, we evolved our effort to an agent- and skill-centric platform rather than a RAG service. Section 3 describes the resulting platform.

3 System Architecture

LLMoxie implements a three-tiered architectural framework that separates the model-serving inference layer, the control plane for access control and governance, and the application-level agents and skills, as illustrated in Figure 2.

3.1 Inference Layer

The foundational tier abstracts the underlying computational substrate, permitting transparent routing across heterogeneous inference backends. The system supports cloud-native deployment on Azure (via Microsoft Foundry Models), Amazon Web Services (via Amazon Bedrock), and Google Cloud Platform (via Vertex AI), as well as on-premise deployment via vLLM in high-performance computing (HPC) environments. This flexibility allows institutional deployment decisions to track existing infrastructure investments, data governance requirements, and cost considerations, including support for credit-based cloud subscriptions.

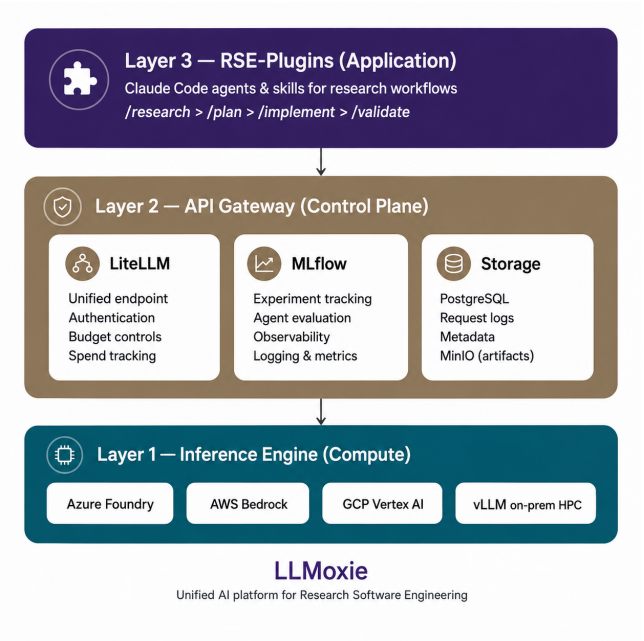

Figure 2: LLMoxie's three-tiered architecture for AI-enabled research software engineering. The inference layer routes requests across cloud (Azure Foundry, AWS Bedrock, GCP Vertex AI) and on-premise (vLLM/HPC) backends; the LiteLLM/MLflow control plane provides a unified OpenAI-compatible gateway with authentication, rate limiting, budget enforcement, PII masking, and observability; and the application augmentation layer delivers domain-specific workflows to Claude Code via the RSE-Plugins ecosystem.

3.2 Control Plane and Governance Layer

An intermediate gateway layer, implemented with LiteLLM and MLflow, provides unified API access to the inference backends through a single OpenAI-compatible endpoint. This abstraction permits transparent provider switching without application-level modifications. The control plane implements the governance mechanisms listed below:

- Authentication and authorization
- Request rate limiting (requests per minute and tokens per minute)
- Per-user and per-team budget enforcement
- Expenditure tracking
- Personally identifiable information (PII) masking via Microsoft Presidio
- Comprehensive user instrumentation to persistent storage (PostgreSQL)

MLflow integration provides model evaluation and observability primitives that downstream consumers use to analyze usage and improve reliability and cost-efficiency.

3.3 Application Augmentation Layer: RSE-Plugins Ecosystem

The application layer implements domain-specific research workflows through RSE-Plugins, an ecosystem designed for research software engineering (RSE) and scientific computing tasks. Rather than relying on generic LLM assistance, RSE-Plugins provide Claude Code with specialized knowledge modules that encode established practices from the scientific Python community and the professional RSE discipline. The ecosystem is organized as a three-level Plugin-Agent-Skill hierarchy of installable, shareable packages, in which each level supplies progressively narrower and more situated guidance. Section 4 details this internal structure and the four plugins currently maintained by our team.

4 RSE-Plugins Architecture and Design

Building on the hierarchy introduced in Section 3.3, an *Agent* is a domain-expert persona that drives multi-step interactions, a *Skill* is a focused, reusable knowledge module invoked on demand, and *slash commands* provide explicit entry points to recurring workflows. Plugins bundle these components into installable packages that can be shared across projects, teams, and institutions. Rather than closing the gap between general-purpose coding agents and scientific Python practice through retraining or fine-tuning, this approach encodes accumulated RSE craft as composable context that any sufficiently capable agent can consume at inference time. The design philosophy follows the Scientific Python Development Guide: collaborate with community conventions rather than work around them, refactor confidently under the protection of comprehensive testing, and prefer wide, reusable solutions over deep, monolithic ones.

4.1 Scientific Python Development Plugin

The Scientific Python Development Plugin is the foundational layer, encoding the conventions and tooling on which the scientific Python community has converged over the past decade. Unlike generic Python guidance, which tends to optimize for web and application idioms, this plugin treats reproducibility, numerical correctness, and packaging discipline as first-class concerns.

4.1.1 Scientific Python Expert Agent. Drawing on the Scientific Python Development Guide, this agent offers guidance across the core scientific Python stack (NumPy, Pandas, Matplotlib, and SciPy); package architecture using src-layout conventions, `pyproject.toml`, and the Hatchling build backend; reproducible environment management with `pixi` (unified Conda and PyPI resolution with multi-platform lockfiles); testing with `pytest`, NumPy testing utilities for numerical comparisons, and Hypothesis for property-based testing; code quality tooling such as `ruff` for linting and formatting, `mypy` for static type checking, and `pre-commit` hooks; and numerical computing best practices including edge case handling (NaN, inf, empty arrays), separation of I/O from computation logic, and NumPy-style docstrings. Rather than dispensing isolated point recommendations, the agent applies a structured decision-making framework that weighs scientific context, reproducibility requirements, and community conventions when guiding implementation choices.

4.1.2 Scientific Documentation Architect Agent. Documentation quality is critical to scientific software discovery, adoption, and long-term maintenance, yet documentation is consistently the work researchers most often defer. This agent produces technical documentation for scientific Python codebases, organized according to the Diátaxis framework. Core competencies include Diátaxis-structured documentation (tutorials, task-oriented how-to guides, API reference, and conceptual or architectural explanation); Sphinx and MkDocs configuration with `pydata-sphinx-theme`, `furo`, `numpydoc`, `autodoc`, `napoleon`, and `intersphinx` cross-referencing; NumPy-style docstrings and API reference generation for scientific libraries; visual communication through architecture diagrams and algorithm flowcharts; and multi-audience technical writing for researchers, developers, and domain experts. The agent proceeds through a four-phase workflow (Discovery, Planning, Structuring, and Writing) intended to produce documentation that is complete, reproducible, and accessible across experience levels.

4.1.3 Scientific Python Skills. Five specialized skills support the plugin's agents, each encoding a focused slice of scientific Python practice:

`pixi-package-manager`*:* Managing dependencies with `pixi`, which unifies Conda and PyPI in a single resolver. Covers multi-platform lockfiles (Linux, macOS, Windows), feature-based environment configurations (development, testing, GPU/CPU), and task automation.

`python-packaging`*:* Modern packaging conventions: src-layout, `pyproject.toml`, the Hatchling build backend, CLI entry points, PyPI distribution, and version and dependency specification appropriate to scientific packages.

`python-testing`*:* `pytest`-based testing for scientific code: NumPy numerical comparisons with floating-point tolerances, fixtures and parametrization, property-based testing with Hypothesis, coverage measurement, and CI setup.

`code-quality-tools`*:* `ruff` for unified linting and formatting (replacing `flake8`, `black`, and `isort`), `mypy` for static type checking, and `pre-commit` hooks for automated quality gates, with CI integration.

`scientific-documentation`*:* Sphinx and MkDocs configuration, NumPy-style docstrings, the Diátaxis framework, accessibility standards, and documentation hosting on Read the Docs.

4.2 Scientific Domain Applications Plugin

Where the Scientific Python plugin captures cross-cutting practice, the Scientific Domain Applications Plugin addresses the data formats, conventions, and computational patterns that distinguish specific research communities. Astronomy and Earth/climate science are the first instantiations, reflecting where our center has accumulated the deepest engagement; agriculture, health, and additional geoscience subfields named in our portfolio (Section 1) are an active workstream and we are actively working on representing these as dedicated plugins.

4.2.1 Astronomy & Astrophysics Expert Agent. This agent consolidates expertise in astronomical computing workflows (`FITS` file handling, celestial coordinate transformations, photometric and spectroscopic pipelines, and physical unit management) anchored in the Astropy ecosystem. It guides users through workflows in which heterogeneous data types and successive transformations would otherwise invite errors [14].

4.2.2 Scientific Domain Skills. Two focused skills support domain-specific expertise:

`xarray-for-multidimensional-data`*:* Labeled multidimensional arrays with Xarray: `NetCDF`, `HDF5`, and `Zarr` I/O; Dask integration for out-of-core datasets; DataTree for hierarchical organization; and `rioxarray` for geospatial rasters. Particularly valuable for climate, Earth science, and satellite workflows.

`astropy-fundamentals`*:* `FITS` I/O, celestial coordinate transformations and catalog cross-matching, physical units and quantities, multi-scale time handling, aperture and PSF photometry with `photutils`, and 1D spectroscopy with `specutils`.

4.3 AI Research Workflows Plugin

Unstructured AI assistance has a documented tendency to produce bloat, undisciplined refactoring, and implementation approaches that are difficult to justify after the fact [18]. The AI Research Workflows Plugin imposes a structured methodology for complex software tasks, with explicit decision-making at each stage and auditable trails of technical reasoning emitted as a routine byproduct.

4.3.1 Research Workflow Orchestrator Agent. This agent guides users through a rigorous six-phase development methodology in which each phase produces durable artifacts consumed by subsequent phases:

(1) **Research Phase** (`/research` command): Systematic documentation of existing code, architectural patterns, and dependencies. Findings are saved as structured markdown to the `.agents/` directory for reference in subsequent phases. This enables a spec-driven development [26] workflow where specifications become executable, directly generating working implementations.
(2) **Planning Phase** (`/plan` command): Creation of detailed, testable implementation plans through interactive research. The agent decomposes complex tasks into phased units with measurable success criteria split into automated and manual checks.
(3) **Plan Iteration** (`/iterate-plan` command): Refinement of plans based on user feedback or changed requirements, maintaining plan consistency and completeness while accommodating evolving understanding.
(4) **Experimentation Phase** (`/experiment` command): Optional exploration of multiple implementation approaches before commitment, enabling informed selection of optimal strategies.
(5) **Implementation Phase** (`/implement` command): Execution of the plan with systematic verification checkpoints. Implementation proceeds phase-by-phase with intermediate validation against plan specifications.

(6) **Validation Phase** (`/validate` command): Systematic verification that implementation satisfies plan criteria by running all automated checks defined in the plan, identifying required manual testing steps, and generating a structured validation report.

4.3.2 Research Workflow Skills. Supporting the orchestrator agent, the plugin provides one specialized skill:

`research-workflow-management`*:* Templates and frameworks supporting the six-phase methodology, including research documentation templates capturing domain analysis and architectural decisions; plan templates specifying implementation phases, measurable success criteria, and file-level references; experiment templates for controlled comparison of implementation approaches; implementation templates with per-phase verification checkpoints; and handoff templates supporting knowledge transfer between collaborators and workflow sessions.

By documenting decisions at each step of research, planning, and implementation, the workflow addresses a critical limitation of unstructured AI assistance—the absence of decision transparency—and extends scientific reproducibility from data and code to the software engineering decisions themselves. Section 5 situates this workflow alongside the other capability surfaces of LLMoxie.

4.4 Project Management Plugin

The Project Management Plugin addresses the full lifecycle of research software, from inception through mature community development and eventual transition to new maintainers—a recurring need in academic settings where contributors graduate, move institutions, or rotate off grants.

4.4.1 Project Onboarding Specialist Agent. This agent handles project initialization, contributor onboarding, and knowledge transfer for open-source projects in any language. It scaffolds community health files (`README`, `CONTRIBUTING`, `LICENSE`, `CODE_OF_CONDUCT`, `SECURITY`, `CITATION.cff`), GitHub issue and pull request templates, and onboarding documentation. For handoff, it documents institutional knowledge, audits open issues and experimental branches, and assembles transfer packages for incoming maintainers.

4.4.2 Documentation Validator Agent. This agent audits documentation quality and validates setup instructions, combining automated tooling with manual, step-by-step tracing of setup instructions. The automated tooling includes Vale for prose, `markdownlint`, HTMLProofer and `lychee` for links, `doc8` for reStructuredText, language-specific doctest runners such as `pytest –doctest-glob` and `cargo test –doc`, and `nbval` for Jupyter notebooks. Findings are categorized by severity (critical, important, recommended, minor) and reported with file and line references.

4.4.3 Project Lifecycle Commands. Three slash commands provide explicit entry points to recurring project-lifecycle workflows:

`/setup-project`*:* Scaffolds new projects with community health file templates, standard directory structure, and development tooling configuration.

`/project-handoff`*:* Assesses project readiness for transition to new maintainers, evaluating documentation completeness, contributor experience, and knowledge transfer requirements.

`/validate-project-handoff`*:* Systematically tests that setup instructions and documentation function as written, catching discrepancies between documented and actual processes.

4.4.4 Project Management Skills. Two skills provide reusable knowledge modules for the plugin's agents:

`community-health-files`*:* Templates for `README`, `CONTRIBUTING`, `LICENSE`, `CODE_OF_CONDUCT`, `SECURITY`, `CITATION.cff`, GitHub issue and pull request templates, and changelogs, with conventions appropriate for academic and research contexts.

`documentation-validation`*:* Vale prose linting, `markdownlint`, HTMLProofer link checking, `pytest` doctest for code examples, container-based validation of setup instructions, and CI integration for automated documentation checks.

5 Research Capabilities and Operational Modes

LLMoxie exposes two complementary capability surfaces, each addressing a distinct stratum of the research software lifecycle and each adoptable independently. Where unified "AI research platforms" commonly conflate engineering assistance and infrastructure provisioning into a single stack, we have kept these concerns orthogonal: researchers can adopt the RSE-Plugins ecosystem with any sufficiently capable coding agent, deploy the Azure substrate as a self-hosted inference and governance plane, or combine both. The remainder of this section details these two capability surfaces and the software engineering practices that underpin both.

5.1 Engineering Assistance via RSE-Plugins

The RSE-Plugins ecosystem (Section 4) supplies structured context to coding agents at inference time. Through the Plugin-Agent-Skill hierarchy, researchers obtain scientific Python development discipline, domain-situated guidance for fields including astronomy and earth/climate science, a structured AI research-and-implement workflow, and project lifecycle management. Unlike approaches that pursue domain alignment through retraining or fine-tuning, this capability encodes accumulated RSE craft as composable context that any sufficiently capable coding agent can consume.

5.2 Production Deployment on Azure

The infrastructure layer provisions the cloud resources required to operate LLMoxie at scale and is available to teams whose primary use of LLMoxie is the RSE-Plugins ecosystem as well as to teams operating LLMoxie as a self-hosted inference and governance plane. LLMoxie provides infrastructure-as-code capabilities through Pulumi, enabling declarative specification and automated provisioning of cloud-native resources on Azure. The deployment system creates a comprehensive infrastructure stack, including:

- **Virtual networking infrastructure** with delegated subnets for Container Apps and PostgreSQL, and a private DNS zone for database connectivity

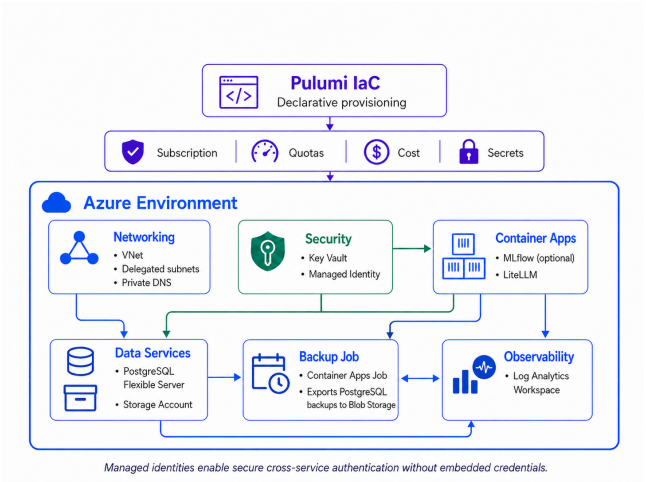

Figure 3: LLMoxie Azure deployment architecture. Pulumi declaratively provisions a virtual network with delegated subnets and private DNS, an Azure Key Vault for secrets and managed identities, a PostgreSQL Flexible Server with scheduled backups to blob storage, and Azure Container Apps hosting the LiteLLM proxy and MLflow experiment tracking, all instrumented through a Log Analytics workspace.

- **Cryptographic key management** via Azure Key Vault with access-policy-based authorization, coupled with a secrets workflow that generates the PostgreSQL administrator password and persists derived connection strings as managed secrets
- **Relational database services** providing PostgreSQL Flexible Server instances with configurable resource tiers, configurable backup retention, and optional zone-redundant high availability
- **Object storage** via an Azure Storage Account with blob containers used for database backups
- **Container orchestration** via Azure Container Apps with optional support for MLflow experiment tracking and LiteLLM proxy services
- **Scheduled database backups** implemented as an Azure Container Apps Job that exports PostgreSQL databases to blob storage on a recurring schedule
- **Observability infrastructure** including Log Analytics Workspace for monitoring and diagnostics
- **Managed identities** for secure cross-service authentication without credential exposure

The deployment system includes validation workflows that verify Azure subscription access, validate resource quotas, estimate operational costs, and perform secrets validation before infrastructure provisioning.

5.3 Software Engineering Practices and Quality Assurance

Both LLMoxie and RSE-Plugins adhere to open-source software engineering standards [3, 13] appropriate for research software.

Figure 4: LLMoxie Analytics snapshot. LLMoxie platform spend for 2026-05-13, showing 2,236 API requests (99.6% success rate), 45.2M tokens consumed, and a total spend of $214.14 distributed across five teams, with External Collaborators comprising the dominant share ($176.32).

Automated testing with `pytest` and coverage analysis is paired with `pre-commit` hooks that enforce code quality (`flake8` for linting, `prettier` for formatting consistency, `codespell` for documentation accuracy). Environments are managed through `pixi`, enabling consistent development, testing, and deployment across heterogeneous host systems while integrating both Conda and PyPI package ecosystems. Comprehensive logging and monitoring support diagnosis of production issues, with PostgreSQL integration providing persistent audit trails. Figure 4 shows a high level snapshot of daily usage. Further drill down reports show detailed usage by provider, model and key. Documentation is generated automatically from source code—OpenAPI schemas for REST APIs and docstring extraction for Python modules—reducing documentation drift.

6 Implications and Future Work

Several lessons emerge from our practice of incorporating LLMs and agents with PIs across a dozen research teams. In addition to creating and using LLMoxie for scientific software development within the academic research and open-source/open-science ecosystem, we observed measurable gains in our capacity for impact.

Invest early in onboarding and documentation. Initial training sessions revealed substantial gaps in user understanding of AI-assisted scientific software development. Subsequent guided exposure helped our collaborators develop more accurate mental models of what AI, specifically the LLMoxie infrastructure and plugins, could and could not reliably do. Documentation quality proved to be a primary lever: clear, detailed documentation measurably reduced errors and improved adoption rates across teams.

Prioritize modular infrastructure design. Modularity (e.g., reusable skills and plugins) allowed teams to integrate LLMoxie plugins and skills into diverse workflows as reusable, domain-specific components rather than conforming to a single prescribed approach. Model flexibility, interoperability, and an

openly documented design improved debuggability and made teams more likely to adopt and sustain agentic-AI-generated code, and in several cases to contribute community plugins back to the LLMoxie open-source project.

Calibrate best practices to user expertise. Greenfield development tasks showed clear and consistent gains, with LLMoxie accelerating prototyping and enabling non-experts to implement complex functionality. Complex debugging, legacy codebase modification, and tasks requiring fine-grained correctness presented a more complicated picture. Students without strong domain knowledge who used agentic AI through LLMoxie sometimes unknowingly incorporated errors into their code, echoing broader findings that novice users are most susceptible to over-reliance and to skill-formation deficits when AI does the heavy lifting [15, 25, 29, 32]. Large undifferentiated context windows degraded performance, consistent with reports that long-context retrieval and multi-turn agent interactions exhibit characteristic degradation modes [4, 18]; preprocessing and selectively curating relevant information substantially improved output quality.

Plan for cost, observability, and long-term sustainability. Centralized infrastructure providing unified access across multiple models substantially lowered barriers to entry, with shared resource allocations enabling broader participation, particularly among students and junior researchers. When LLMoxie infrastructure is funded through credit programs with expiration dates such as federally funded national AI compute pilots [20], system design must account for resource constraints and plan for sustainability. Observability and telemetry proved essential: our team used LiteLLM and MLflow to monitor usage, diagnose failures, and understand adoption patterns. These signals guided per-user-group configuration choices that optimized both token usage and corresponding spend.

Address the governance gap. LLMoxie shifts developer focus from code generation toward higher-order evaluation and design, enabling more rapid prototyping and a greater willingness to explore and discard approaches. It also improved interaction with legacy systems by reducing the cognitive load of navigating unfamiliar code, making maintenance of complex legacy codebases more approachable. However, current agentic AI practices lack mechanisms analogous to version control or commit history for tracking the provenance of AI-assisted contributions, accumulating what has been termed *intent debt*: the absence of externalized rationale that developers and AI agents alike need in order to safely evolve a system [30]. While some communities have begun to establish standards for reviewing AI-assisted software [11, 27], this remains a rapidly evolving area. We have used this gap to scope future work on LLMoxie centered on provenance tracking, reproducibility, and auditability of AI-assisted contributions.

Build feedback loops between practice and infrastructure. The most successful deployments followed a pattern of iterative co-evolution rather than linear tool adoption. LLMoxie infrastructure and plugins were built by our team to address the pain points encountered in day-to-day work on scientific software development. Evidence-based best practices—structured development phases, context-curation methods, and quality-assurance mechanisms—once designed as reusable components, became available to subsequent teams without requiring them to independently traverse the same learning curve. Institutional investment should therefore focus not only on tooling itself but on the feedback mechanisms that allow learned practices to be continuously refined and shared.

7 Limitations

Thorough utilization across domains, user experience levels, and use cases has revealed areas of improvement. First, the experience reported here is drawn from a single multi-domain RSE center; while the same failure modes recurred across diverse projects, generalization to other institutional settings remains to be established. Second, the operational lessons in Section 6 are based on practitioner observation and project artifacts rather than a controlled user study; we do not report quantitative measures of productivity, code quality, or scientific output, and a more formal evaluation is an obvious next step. Third, the application-augmentation layer is currently anchored on a specific coding agent (Claude Code), and although the plugin design targets portability across sufficiently capable agents, that portability has not been exercised at scale. Fourth, as an open-source framework for running, managing, and benchmarking LLM experiments and deployments, the repository has few active maintainers; substantial investment is needed to grow the contributor base, otherwise issues and pull requests risk accumulating and external users may fork rather than contribute. Finally, LLM-based software engineering is a rapidly evolving field with an accelerated pace of technological breakthroughs and model evolution [21]. While RSEs can keep up with and continually evolve tooling such as LLMoxie, prioritization of the engineering backlog must be thoughtfully balanced against unforeseen developments from frontier model laboratories.

8 Conclusion

LLMoxie offers a systematic approach to integrating large language models into scientific research workflows while preserving researcher autonomy, institutional data governance, and scientific reproducibility. Its three-tiered architecture separates inference, governance, and application augmentation, enabling flexible deployment across institutional contexts under uniform governance, while the Plugin-Agent-Skill hierarchy provides a scalable framework for encoding RSE expertise across diverse scientific computing challenges.

Through the open-source RSE-Plugins ecosystem, the platform shifts LLMs from generic code generators into domain-aware collaborators that respect community conventions, follow established scientific computing practices, and produce auditable trails of technical reasoning. By combining open-source development, modular design, and reusable RSE knowledge, LLMoxie provides a foundation for accessible, high-quality AI-augmented scientific software that allows researchers and RSEs to pair computational and human expertise without compromising the rigor and reproducibility that scientific work demands.

9 Acknowledgments

The authors thank the National AI Research Resource (NAIRR) Pilot program funded by the National Science Foundation (NSF) that provided cloud credits (Award 240292) using which the LLMoxie stack was created. We thank HuggingFace and Microsoft Azure

for credits. In addition we thank support from Sarah Stone and the CloudBank program for helping sustain the development of LLMoxie. We thank Amazon and AWS Bedrock for credits to eScience Institute using which we utilized Anthropic models. The University of Washington Scientific Software Engineering Center (SSEC) is supported by Schmidt Sciences, as part of their Virtual Institute for Scientific Software (VISS) program.

References

[1] Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. 2025. Small Language Models are the Future of Agentic AI. doi:10.48550/arXiv.2506.02153

[2] Stephanie A. Besser, Eric A. Jensen, and Daniel S. Katz. 2026. How Generative AI Is Shaping Research Software Development and Maintenance at a Research-Intensive University. *Open Research Europe* 6 (2026), 56. doi:10.12688/openreseurope.22009.1

[3] Eric W. Bridgeford, Iain Campbell, Zijao Chen, Zhicheng Lin, Harrison Ritz, Joachim Vandekerckhove, and Russell A. Poldrack. 2025. Ten Simple Rules for AI-Assisted Coding in Science. doi:10.48550/arXiv.2510.22254

[4] Databricks. 2024. Long Context RAG Performance of LLMs. https://www.databricks.com/blog/long-context-rag-performance-llms

[5] Titouan Duston, Shuo Xin, Yang Sun, Daoguang Zan, Aoyan Li, Shulin Xin, Kai Shen, Yixiao Chen, Qiming Sun, Ge Zhang, Jiashuo Liu, Huan Zhou, Jingkai Liu, Zhichen Pu, Yuanheng Wang, Bo-Xuan Ge, Xin Tong, Fei Ye, Zhi-Chao Zhao, Wen-Biao Han, Zhoujian Cao, Yueran Zhao, Weiluo Ren, Qingshen Long, Yuxiao Liu, Anni Huang, Yidi Du, Yuanyuan Rong, and Jiahao Peng. 2025. AInsteinBench: Benchmarking Coding Agents on Scientific Repositories. (2025). doi:10.48550/ARXIV.2512.21373

[6] Siamak Farshidi, Kwabena Bennin, Önder Babur, June Sallou, Ayalew Kassahun, and Bedir Tekinerdogan. 2025. Advancing Research Software Engineering With AI: A Research Framework. doi:10.21203/rs.3.rs-7178452/v1

[7] Tiantian Gan and Qiyao Sun. 2025. RAG-MCP: Mitigating Prompt Bloat in LLM Tool Selection via Retrieval-Augmented Generation. doi:10.48550/arXiv.2505.03275

[8] Sandra Gesing. 2025. RSEs 2035: Surviving or Thriving in the Age of AI. In *2025 IEEE International Conference on eScience (eScience)*. 381–382. doi:10.1109/eScience65000.2025.00081 ISSN: 2325-3703.

[9] Dirk Groeneveld, Iz Beltagy, Pete Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, A. Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, Shane Arora, David Atkinson, Russell Authur, Khyathi Raghavi Chandu, Arman Cohan, Jennifer Dumas, Yanai Elazar, Yuling Gu, Jack Hessel, Tushar Khot, William Merrill, Jacob Daniel Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew E. Peters, Valentina Pyatkin, Abhilasha Ravichander, Dustin Schwenk, Saurabh Shah, Will Smith, Emma Strubell, Nishant Subramani, Mitchell Wortsman, Pradeep Dasigi, Nathan Lambert, Kyle Richardson, Luke S. Zettlemoyer, Jesse Dodge, Kyle Lo, Luca Soldaini, Noah A. Smith, and Hanna Hajishirzi. 2024. OLMo: Accelerating the Science of Language Models. In *Annual Meeting of the Association for Computational Linguistics*. https://api.semanticscholar.org/CorpusID:267365485

[10] Alexandre Hocquet, Frédéric Wieber, Gabriele Gramelsberger, Konrad Hinsen, Markus Diesmann, Fernando Pasquini Santos, Catharina Landström, Benjamin Peters, Dawid Kasprowicz, Arianna Borrelli, Phillip Roth, Clarissa Ai Ling Lee, Alin Olteanu, and Stefan Böschen. 2024. Software in Science Is Ubiquitous yet Overlooked. *Nature Computational Science* (July 2024), 1–4. doi:10.1038/s43588-024-00651-2

[11] Mohammad Hosseini, Serge P J M Horbach, Kristi L Holmes, and Tony Ross-Hellauer. 2024. Open Science at the Generative AI Turn: An Exploratory Analysis of Challenges and Opportunities. doi:10.1162/qss_a_00337

[12] Alyssa Hughes. 2022. AI4Science to Empower the Fifth Paradigm of Scientific Discovery. https://www.microsoft.com/en-us/research/blog/ai4science-to-empower-the-fifth-paradigm-of-scientific-discovery/

[13] Haley Hunter-Zinck, Alexandre Fioravante De Siqueira, Váleri N. Vásquez, Richard Barnes, and Ciera C. Martinez. 2021. Ten Simple Rules on Writing Clean and Reliable Open-Source Scientific Software. *PLOS Computational Biology* 17, 11 (Nov. 2021), e1009481. doi:10.1371/journal.pcbi.1009481

[14] Sebastian Antony Joseph, Syed Murtaza Husain, Stella S. R. Offner, Stéphanie Juneau, Paul Torrey, Adam S. Bolton, Juan P. Farias, Niall Gaffney, Greg Durrett, and Junyi Jessy Li. 2025. AstroVisBench: A Code Benchmark for Scientific Computing and Visualization in Astronomy. (2025). doi:10.48550/ARXIV.2505.20538

[15] Samia Kabir, David N. Udo-Imeh, Bonan Kou, and Tianyi Zhang. 2024. Is Stack Overflow Obsolete? An Empirical Study of the Characteristics of ChatGPT Answers to Stack Overflow Questions. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/3613904.3642596

[16] Sayash Kapoor, Benedikt Stroebl, Peter Kirgis, Nitya Nadgir, Zachary S. Siegel, Boyi Wei, Tianci Xue, Ziru Chen, Felix Chen, Saiteja Utpala, Franck Ndzomga, Dheeraj Oruganty, Sophie Luskin, Kangheng Liu, Botao Yu, Amit Arora, Dongyoon Hahm, Harsh Trivedi, Huan Sun, Juyong Lee, Tengjun Jin, Yifan Mai, Yifei Zhou, Yuxuan Zhu, Rishi Bommasani, Daniel Kang, Dawn Song, Peter Henderson, Yu Su, Percy Liang, and Arvind Narayanan. 2025. Holistic Agent Leaderboard: The Missing Infrastructure for AI Agent Evaluation. doi:10.48550/arXiv.2510.11977

[17] Miklós Koren, Gábor Békés, Julian Hinz, and Aaron Lohmann. 2026. Vibe Coding Kills Open Source. doi:10.48550/arXiv.2601.15494

[18] Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. 2025. LLMs Get Lost In Multi-Turn Conversation. doi:10.48550/arXiv.2505.06120

[19] Hao Li, Yiqun Zhang, Zhaoyan Guo, Chenxu Wang, Shengji Tang, Qiaosheng Zhang, Yang Chen, Biqing Qi, Peng Ye, Lei Bai, Zhen Wang, and Shuyue Hu. 2026. LLMRouterBench: A Massive Benchmark and Unified Framework for LLM Routing. (2026). doi:10.48550/ARXIV.2601.07206

[20] Vani Mandava. [n. d.]. NAIRR Current Projects: NAIRR240292. https://submit-nairr.xras.org/current-projects?_requestNumber=NAIRR240292. Accessed: 2026-06-28.

[21] Lois Curfman McInnes, Dorian Arnold, Prasanna Balaprakash, Mike Bernhardt, Beth Cerny, Anshu Dubey, Roscoe Giles, Denice Ward Hood, Mary Ann Leung, Vanessa Lopez-Marrero, Paul Messina, Olivia B. Newton, Chris Oehmen, Stefan M. Wild, Jim Willenbring, Lou Woodley, Tony Baylis, David E. Bernholdt, Chris Camano, Johannah Cohoon, Charles Ferenbaugh, Stephen M. Fiore, Sandra Gesing, Diego Gomez-Zara, James Howison, Tanzima Islam, David Kepczynski, Charles Lively, Harshitha Menon, Bronson Messer, Marieme Ngom, Umesh Paliath, Michael E. Papka, Irene Qualters, Elaine M. Raybourn, Katherine Riley, Paulina Rodriguez, Damian Rouson, Michelle Schwalbe, Sudip K. Seal, Ozge Surer, Valerie Taylor, and Lingfei Wu. 2025. Report of the 2025 Workshop on Next-Generation Ecosystems for Scientific Computing: Harnessing Community, Software, and AI for Cross-Disciplinary Team Science. doi:10.48550/arXiv.2510.03413

[22] Linda Nordling. 2023. How ChatGPT Is Transforming the Postdoc Experience. *Nature* 622, 7983 (Oct. 2023), 655–657. doi:10.1038/d41586-023-03235-8

[23] Gabrielle O'Brien. 2025. How Scientists Use Large Language Models to Program. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3706598.3713668

[24] Gabrielle O'Brien, Alexis Parker, Nasir Eisty, and Jeffrey Carver. 2026. A Survey of Generative AI Adoption and Perceived Productivity Among Scientists Who Program. doi:10.48550/arXiv.2512.19644

[25] Gabrielle O'Brien. 2025. Threats to Scientific Software From Over-Reliance on AI Code Assistants. *Nature Computational Science* 5, 9 (Sept. 2025), 701–703. doi:10.1038/s43588-025-00845-2

[26] Deepak Babu Piskala. 2026. Spec-Driven Development:From Code to Contract in the Age of AI Coding Assistants. doi:10.48550/arXiv.2602.00180 arXiv:2602.00180 [cs.SE].

[27] rOpenSci. 2026. Software Review in the Era of AI: What We Are Testing at rOpenSci. (Feb. 2026). doi:10.59350/5tavw-mke71

[28] Don Setiawan, Anant Mittal, Anuj Sinha, Anuj Khandelwal, Cordero Core, Madhav Kashyap, Niki Burggraf, Anshul Tambay, and Vani Mandava. 2024. Generative AI Copilot for Scientific Software – a RAG-Based Approach. https://uw-ssec-tutorials.readthedocs.io/en/latest/Archive/SciPy2024/README.html

[29] Judy Hanwen Shen and Alex Tamkin. 2026. How AI Impacts Skill Formation. doi:10.48550/arXiv.2601.20245

[30] Margaret-Anne Storey. 2026. From Technical Debt to Cognitive and Intent Debt: Rethinking Software Health in the Age of AI. doi:10.48550/arXiv.2603.22106

[31] Minyang Tian, Luyu Gao, Shizhuo Dylan Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yao Li, Shengyan Liu, Di Luo, Yutao Ma, Hao Tong, Kha Trinh, Chenyu Tian, Zihan Wang, Bohao Wu, Yanyu Xiong, Shengzhu Yin, Minhui Zhu, Kilian Lieret, Yanxin Lu, Genglin Liu, Yufeng Du, Tianhua Tao, Ofir Press, Jamie Callan, Eliu Huerta, and Hao Peng. 2024. SciCode: A Research Coding Benchmark Curated by Scientists. (2024). doi:10.48550/ARXIV.2407.13168

[32] Suqing Wu, Yukun Liu, Mengqi Ruan, Siyu Chen, and Xiao-Yun Xie. 2025. Human-Generative AI Collaboration Enhances Task Performance but Undermines Human's Intrinsic Motivation. *Scientific Reports* 15, 1 (April 2025), 15105. doi:10.1038/s41598-025-98385-2