Code2MCP:
Transforming Code Repositories into MCP Services

Chaoqian Ouyang[1*] **Ling Yue**[2*] **Shimin Di**[3†] **Libin Zheng**[1†] **Linan Yue**[3]
Shaowu Pan[2] **Jian Yin**[1] **Min-Ling Zhang**[3]
[1]Sun Yat-sen University [2]Rensselaer Polytechnic Institute [3]Southeast University
shimin.di@seu.edu.cn zhenglb6@mail.sysu.edu.cn

Abstract

The Model Context Protocol (MCP) aims to create a standard for how Large Language Models use tools. However, most current research focuses on selecting tools from an existing pool. A more fundamental, yet largely overlooked, problem is how to populate this pool by converting the vast number of existing software projects into MCP-compatible services. To bridge this gap, we introduce Code2MCP, an agent-based framework that automatically transforms a GitHub repository into a functional MCP service with minimal human intervention. Code2MCP employs a multi-agent workflow for code analysis, environment setup, tool function design, and service generation, enhanced by a self-correcting loop to ensure reliability. We demonstrate that Code2MCP successfully transforms open-source computing libraries in scientific fields such as bioinformatics, mathematics, and fluid dynamics that are not available in existing MCP servers. By providing a novel automated pathway to unlock GitHub, the world's largest code repository, for the MCP ecosystem, Code2MCP serves as a catalyst to significantly accelerate the protocol's adoption and practical application. The code is public at `https://github.com/DEFENSE-SEU/Code2MCP`.

1 Introduction

The landscape of artificial intelligence is increasingly defined by autonomous agents that leverage Large Language Models (LLMs) to interact with external tools (Wang et al., 2024; Xi et al., 2024; Bubeck et al., 2023). To overcome the inherent limitations of LLMs in tasks requiring real-time information or precise computation, the paradigm of tool-augmented reasoning has become central (Huang et al., 2024; Hao et al., 2023; Yue et al., 2024). Seminal works have demonstrated that models can effectively learn to invoke external functions (Schick et al., 2023; Qin et al., 2023; Parisi et al., 2022).

However, this burgeoning ecosystem faces a fundamental scalability challenge: the $N \times M$ integration problem (Li et al., 2023; Liang et al., 2023; Qin et al., 2023; Anthropic, 2023). Each of the N models or agent applications often requires a bespoke connector for each of the M tools it must access. This results in a fragmented and inefficient system where development effort is duplicated and innovation is stifled by high integration costs (Qu et al., 2025; Shen, 2024). To address this, MCP is proposed as a universal standard that specifies how agents and tools should communicate, enabling an interoperable "plug-and-play" ecosystem (Anthropic, 2023).

In response to this integration challenge, the community's efforts have evolved, inadvertently revealing a deeper, more foundational bottleneck (Yue et al., 2025a). The initial challenge is to establish

*Equal contribution.
†Corresponding authors; and the work was done when C. Ouyang was an intern mentored by S. Di.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Scaling Environments for Agents (SEA).

the fundamental feasibility of tool use, where a limited set of tools proves the feasibility of the paradigm (Schick et al., 2023; Patil et al., 2023; Ding et al., 2025). To break past the inherent scarcity of these platforms, the focus shifts to the vast landscape of open-source repositories (Wang et al., 2025; Xie et al., 2023). This move, however, trades a scarcity problem for a chaos problem, exposing the wild non-standardization of real-world code. The MCP emerges as a direct answer to this chaos, promising a universal interface (Zhang et al., 2025; Anthropic, 2023). Yet, this leads to the critical gap: research is focused almost exclusively on the consumption side of MCP, using services from a presumed-to-exist pool (Gan & Sun, 2025), while the foundational "supply-side" problem of how to populate this pool from existing software is largely unaddressed.

However, while these efforts advance the consumption side of the problem, how agents can better use tools, they largely overlook a more fundamental bottleneck on the supply side. This supply bottleneck is not a theoretical concern but a stark reality preventing standards like MCP from achieving widespread adoption. For example, RAG-MCP (Gan & Sun, 2025) utilizes over 4,400 servers on `mcp.so`, but there are 268 million public GitHub repositories. The critical question of how to create a large and diverse pool of these standardized, agent-ready tools has been left unaddressed. This creates a major adoption gap, effectively locking away the largest software repository, GitHub, from this emerging ecosystem.

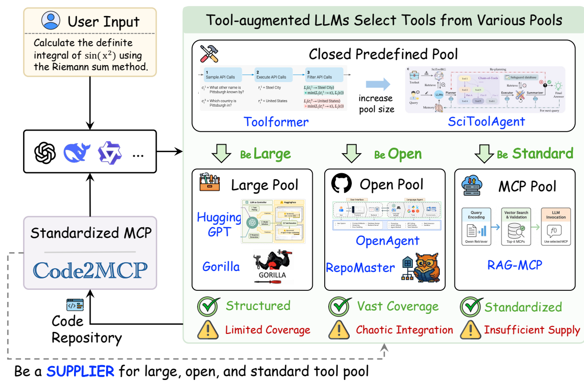

Figure 1: While most research focuses on the consumption of tools (right side), one bottleneck is their supply (left side). Code2MCP solves the supply problem by converting the code repository into a standardized MCP-compliant tool.

In this paper, we introduce Code2MCP, a new framework designed to bridge the critical tool supply gap. Code2MCP presents a blueprint for transforming any GitHub repository into a functional and documented MCP service with minimal human intervention. However, this transformation is a complex endeavor encompassing four pivotal challenges: (1) deep code comprehension to identify core functionalities, (2) reliable environment replication to ensure executability, (3) intelligent tool abstraction to design useful and valid service interfaces, and (4) robust self-correction to handle the inevitable errors throughout the process. To systematically address these challenges, as shown in Figure 2, Code2MCP implements a collaborative multi-agent system (Park et al., 2023). Unlike general-purpose coding agents (Cognition, 2024; Jimenez et al., 2024), different agents in our framework are specialized for the distinct stages of code analysis, environment setup, and API design. Crucially, the overall reliability of this workflow is ensured by an integrated Run-Review-Fix self-correction cycle, which endows the system with the ability to autonomously debug and refine the entire conversion process. The key contributions of this work are listed as follows:

- To solve the fundamental tool supply bottleneck hindering the adoption of the MCP standard, we propose a novel automated framework, Code2MCP, the first framework to systematically transform code repositories into agent-ready MCP services.
- The key challenge in converting code into a service is the inherent fragility of the multi-stage automation process, where an error at any step can derail the entire workflow. Thus, we introduce a novel multi-agent architecture governed by a Run-Review-Fix cycle, a self-correcting mechanism designed to systematically debug and refine the process, ensuring end-to-end reliability.
- We demonstrate the effectiveness and scalability of our framework by converting highly complex and diverse scientific libraries, covering Protein Design, Symbolic Mathematics, and Computational Fluid Dynamics, into fully functional MCP services. This provides a concrete and practical pathway to enrich the MCP ecosystem with specialized, high-value tools.

2 Related Work

As summarized in Section 1 and Table 1, the pioneering works focus on progressively expanding the scope of tool use, from initial feasibility studies using a few predefined APIs to leveraging large,

Table 1: A comparative summary of related works in tool-augmented LLMs.

	Work	Core Contribution	Tool Pool	Tool Selection	MCP
Consumer	**Toolformer**	Teaching LLM to use external tools	5 predefined tools	Fine-tuning	×
	SciToolAgent	Domain-specific enhancement for scientific tool utilization	KG of Scientific Tools (500+ tools)	Retrieval on KG	×
	HuggingGPT	Increase the size of tool pool	`huggingface.co`	LLM task planning	×
	Gorilla		TorchHub, TensorHub (1600+ tools)	Retriever-aware training	×
	OpenAgents	Tool use from open-source beyond a closed pool	`github.com`	Multi-agent planning	×
	RepoMaster			Rule-based deep search	×
	RAG-MCP	RAG for tool selection from MCP	`MCP.so` (4,400+)	Retrieval on MCP	✓
Supplier	**Code2MCP**	GitHub Repo to standardized MCP	`github.com`	N/A	✓

curated tool platforms, and ultimately, to the ambitious goal of directly interfacing with unstructured open-source repositories. Thus, the current bottleneck lies not in how LLMs consume tools, but in how such tools are supplied and created. In this paper, Code2MCP is designed to solve this fundamental "supply-side" problem.

Initial explorations in LLM Tool Use. The initial challenge is to establish the fundamental feasibility of tool use. Toolformer (Schick et al., 2023) demonstrates that an LLM can learn to invoke simple, well-defined tools like a calculator via simple APIs in a self-supervised way. SciToolAgent (Ding et al., 2025) leverages knowledge graphs to orchestrate 500+ scientific tools. This proves the concept and opens a new paradigm. However, its reliance on a small, predefined set of tools is inherently unscalable and insufficient for addressing the diverse needs of real-world tasks.

Scaling Tool Availability via Structured Platforms. To overcome the limitation of fixed toolsets, subsequent research turns to large, curated platforms. These approaches significantly expand the number of available tools. For instance, Gorilla (Patil et al., 2023) fine-tunes models on a massive corpus of API calls from hubs like TorchHub and TensorFlow Hub. Similarly, HuggingGPT (Shen et al., 2023) positions an LLM as a controller to delegate tasks to specialized models within the Hugging Face ecosystem. While powerful, their success hinges on environments where tools are well-documented and standardized.

Exploring Unstructured Open-Source Repositories and Challenges. A more ambitious paradigm shift involves treating the entirety of open-source code repositories as a virtually infinite tool source. Frameworks like OpenAgents (Xie et al., 2023) and RepoMaster (Wang et al., 2025) empower agents to directly parse, reason about, and execute code within GitHub repositories. These works confront the complexity of real-world code but expose the core bottleneck: the vast majority of these repositories are not designed for programmatic use by LLM agents. They lack standardized interfaces (Zhang et al., 2024; Jin et al., 2024; Ray, 2025), forcing the agent into an ad-hoc, brittle, and unreliable process of reverse-engineering the code, setting up its environment, and managing dependencies for every single task (Zeng et al., 2024; Olausson et al., 2023). This chaotic integration process illustrates a critical failure on the tool supply side.

The Emergence of Standardization and Unaddressed Gap. Recognizing this chaos, the community has moved towards standardization, exemplified by the MCP. For example, RAG-MCP (Gan & Sun, 2025) explores how an agent can effectively retrieve and select the most appropriate MCP service from a pool of 4400+ available options. This approach is promising, but it presumes the existence of a rich ecosystem of MCP-compliant services (Hasan et al., 2025). This highlights a crucial, unaddressed gap: *how is this ecosystem of MCP services populated in the first place?*

3 Methodology: The Code2MCP Framework

To achieve the goal of automatically transforming an arbitrary GitHub repository into a fully functional and reliable MCP service, we design Code2MCP, an automated framework driven by the collaboration of seven specialized agents. The entire conversion process, as depicted in Figure 2, is a multi-agent

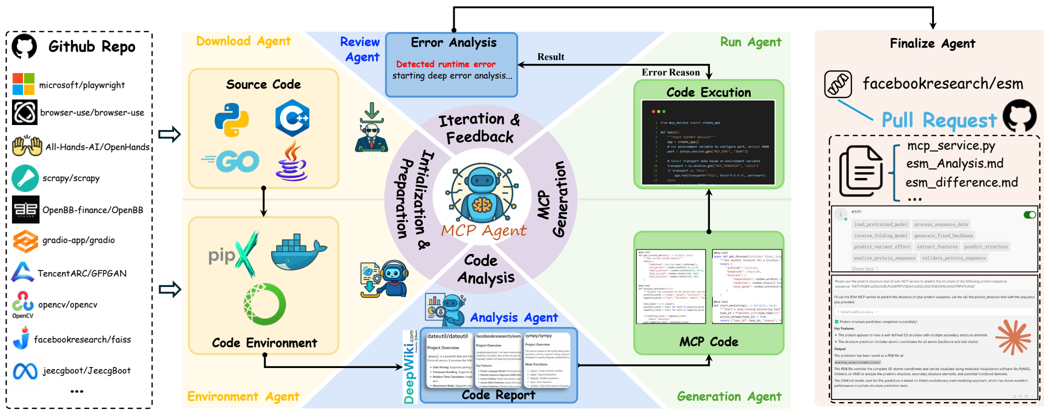

Figure 2: Overview of the Code2MCP framework. The system takes a GitHub repository URL as input and automatically generates a complete MCP service through a multi-agent workflow.

workflow that begins with code analysis, proceeds through a core Run-Review-Fix self-correction loop, and culminates in the generation of a merge-ready pull request.

Suppose there exists a consumer-side work listed in Table 1 that finds a suitable GitHub repository that may solve the user's query. Code2MCP converts this repository into MCP that LLMs can call and use. This is the core difference between this "supply-side" work and the consumer-side works.

Initialization and Analysis. First, the `Download Agent` is responsible for cloning the specified repository, identified by its URL u, and creating an isolated local workspace. Subsequently, the `Environment Agent` begins its work by addressing environmental dependencies, one of the most common failure points in code conversion. It meticulously replicates the original code's runtime environment by intelligently parsing dependency files or Dockerfiles. This step is the cornerstone for ensuring the reliability of all subsequent code generation and testing.

Once the environment is ready, the core challenge is identifying valuable, tool-worthy functionalities within a complex codebase. This is the task of the `Analysis Agent`. It moves beyond traditional static analysis by leveraging the DeepWiki tool for a deep semantic understanding of the code. By associating code entities with their intent from documentation and comments, the agent understands the code's "purpose", not just its "structure". The final output is a detailed Code Report, which serves as the strategic blueprint for all subsequent stages.

Generation, Execution, and Self-Correction. After obtaining a clear conversion blueprint, the framework enters its core iterative loop, which is designed to tackle the challenge of translating abstract functionalities into concrete, executable, and correct MCP services.

The loop commences with the `Generation Agent`. It takes the Code Report from the Analysis Agent and uses the code generation capabilities of an LLM to intelligently abstract and encapsulate

Algorithm 1 The Code2MCP Framework

Input: GitHub repository URL u
`Download Agent`: Clone repository into an isolated workspace.
`Environment Agent`: Replicate runtime environment from dependency files.
`Analysis Agent`: Analyze codebase $\longrightarrow$ generate detailed Code Report.
`Generation Agent`: Synthesize initial MCP files (`mcp_service.py`, `adapter.py`, tests) based on Code Report.
$r \leftarrow 0$; $success \leftarrow$ false
while $\neg success \wedge r < B$ **do**
  `Run Agent`: Execute test suite; collect error traceback τ on failure.
  if all tests pass **then**
    $success \leftarrow$ true
  else
    `Review Agent`: Analyze $\tau \longrightarrow$ generate correction plan δ.
    `Generation Agent`: Re-synthesize MCP files using Code Report and correction plan δ.
    $r \leftarrow r + 1$
  end if
end while
`Finalize Agent`: Package service files, generate README, and create Pull Request.
Output: A merge-ready Pull Request containing the functional MCP service.

the identified core functionalities into MCP-compliant interfaces. This includes creating essential files like the tool interface definition and adapter file, which connect the original code to the MCP interface, along with a basic test suite.

Once the code is generated, the `Run Agent` immediately executes the test suite within the prepared environment. Its role is to verify the practical executability of the generated code. If the tests pass, the workflow exits the loop and proceeds to the finalization stage. However, if execution fails, the `Run Agent` captures a detailed error traceback, denoted as τ, and passes it to the `Review Agent`, triggering the framework's critical self-correction mechanism. The `Review Agent` acts as a "senior debugging engineer", performing an in-depth error analysis. Its analysis goes beyond simply parsing the traceback τ; it correlates the error message with the generated code, the original repository's structure outlined in the Code Report, and the specific test case that failed. This contextual reasoning allows it to diagnose the root cause with high precision, identifying issues such as code logic error, a missing dependency, or an interface mismatch. After diagnosis, it formulates a correction plan, δ. The resulting plan is a structured, actionable directive that specifies the exact file and code block to be modified and provides a clear description of the required change. This precise instruction is then fed back to the Generation Agent to re-synthesize the corrected code. This closed loop of "run-review-fix," formed by the `Run Agent` and `Review Agent`, iterates until all issues are resolved and the generated code passes all tests. To prevent infinite loops, the process is bounded by a maximum of B attempts. This robust self-correction mechanism is the key to Code2MCP's end-to-end reliability.

Finalization and Delivery. Upon the successful completion of the core loop, the `Finalize Agent` handles the concluding tasks. It organizes and packages all the validated, functional MCP service files. To facilitate review and adoption by the original repository maintainers, it also automatically generates a clear README file explaining how to use the newly added MCP service. Finally, the `Finalize Agent` packages all generated artifacts into the clear directory structure shown in Figure 3. Operating as a fully automated contributor, it then uses the GitHub API to submit these additions as a new Pull Request to the original repository. This act completes the code-to-service transformation and builds a direct bridge connecting the open-source project to the growing MCP ecosystem.

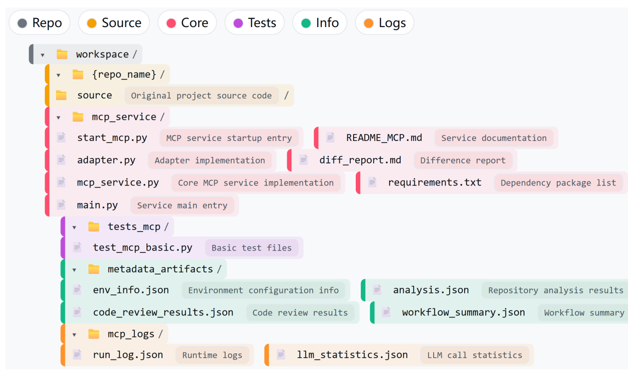

Figure 3: The complete output directory structure generated by the Code2MCP framework. The top-level `workspace` contains the original repository alongside all generated artifacts within the `mcp_output` directory.

4 Evaluation

4.1 Experimental Setup

Task. The evaluation task is to convert diverse GitHub repositories into validated MCP services, assessing both the efficiency in reducing conversion time and cost compared to manual effort, and the effectiveness demonstrated by the functional success of the generated tools.

Demonstrated Tools. As shown in Table 2, we demonstrate 6 representative open-source repositories to cover a spectrum of complexity and domains. The first group includes three widely-used utility libraries for simpler conversion tasks: `TextBlob` for natural language processing, `dateutil` for datetime parsing, and `searchkick` for intelligent search. These serve to demonstrate the significant speedup Code2MCP provides for standard conversion tasks. The second group includes three complex scientific libraries to test Code2MCP's ability to handle domain-specific codebases: `ESM` for protein science, `SymPy` for symbolic mathematics, and `Foam-Agent` for computational fluid dynamics. Their primary purpose is to serve as the subjects of our in-depth case studies. These cases provide the qualitative evidence for its success on complex software. This capability is crucial for addressing the tool supply bottleneck.

Table 2: Statistics on the time required for Code2MCP to convert code into MCP and the token consumption. The manual time is an average of estimates from three senior developers.

Repository	Manual Time (Est. hours)	Code2MCP Time (min)	Speedup Factor	LLM Cost (Tokens)
sloria/TextBlob	1.0 – 1.5	12	~6x	24.7k
dateutil/dateutil	0.7 – 1.2	15	~4x	19.2k
ankane/searchkick	1.0 – 1.5	13	~6x	28.9k
facebookresearch/esm	4.0 – 5.0	50	~5x	58.5k
sympy/sympy	2.5 – 3.5	32	~6x	78.3k
csml-rpi/Foam-Agent	2.0 – 3.0	17	~8x	41.4k

Evaluation. We evaluate the effectiveness and generality of Code2MCP through a two-pronged approach: quantitative evaluation and in-depth case studies. The quantitative evaluation measures key performance metrics, such as conversion time and LLM token cost, against the estimated time for manual conversion to validate Code2MCP's efficiency.

Implementation Details. By default, Code2MCP utilizes gpt-4o-2024-05-13 as its core reasoning engine. The temperature for all models is set to 1. Code2MCP leverages `gitingest`[3] to ingest repositories into contextual prompts and fetch pre-analysis reports from `deepwiki`[4]. Case studies are conducted on servers equipped with 8 NVIDIA H100 80 GB GPUs.

4.2 Quantitative Evaluation

A core motivation of Code2MCP is to alleviate the significant manual effort of turning code repositories into MCP services. Therefore, we report the time and LLM tokens consumed by Code2MCP in Table 2 and compare this with the estimated manual effort for a human expert. The manual time is estimated by three senior developers familiar with the respective libraries and the MCP standard, and the average value is reported. The manual task includes code comprehension, code wrapping, protocol implementation, testing, and documentation. These tasks are tedious and time-consuming, posing a significant challenge even for experienced developers. From Table 2, we observe that the estimated manual time far exceeds that of Code2MCP. This disparity arises because the manual task is a complex endeavor, encompassing challenges like deep code comprehension and environment replication, which extend far beyond simple implementation and testing.

Having established the framework's general efficiency, we now proceed to the case studies of the three scientific libraries. These studies on `ESM`, `SymPy`, and `Foam-Agent` will provide a qualitative analysis of the framework's core capabilities.

4.3 Case Studies: Protein, Math, and Computational Fluid Dynamics

Bridging Protein Science with AI Agents. Advanced computational models, exemplified by AlphaFold (Jumper et al., 2021), have revolutionized structural biology with breakthroughs in protein structure prediction and sequence analysis. However, their practical application is hindered by steep learning curves and substantial engineering effort, creating a high barrier to entry for many scientists, especially those without deep computational backgrounds. For molecular biologists or drug discovery researchers focused on wet-lab experiments, writing Python scripts to analyze protein variants is often a significant distraction from their research. While AlphaFold is now present in the MCP ecosystem [5], `ESM` provides complementary capabilities such as zero-shot variant effect prediction to assess how mutations impact proteins, and it offers higher efficiency and accuracy. Thus, we select `ESM` as our conversion target. More broadly, scientific discovery depends on a diverse toolbox rather than a single universal solution, and the current scarcity of tool variety creates a critical "supply-side" bottleneck.

Historically, performing even foundational protein characterization with a library like `ESM` is a multi-step, technical endeavor. Users have to write custom scripts, navigate the library's internal

[3] `https://github.com/coderamp-labs/gitingest`

[4] `https://deepwiki.org`

[5] `https://github.com/augmented-nature/alphafold-mcp-server`

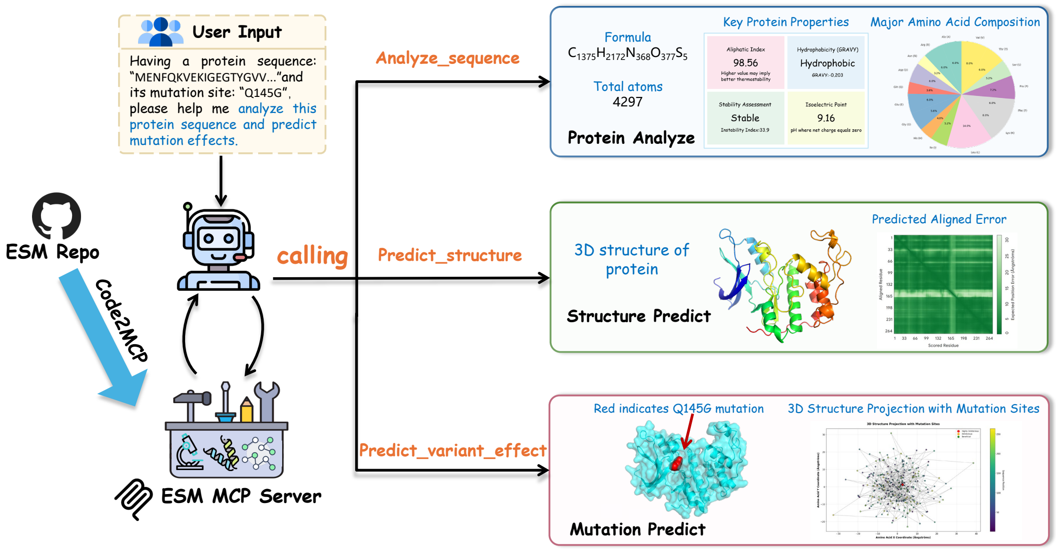

Figure 4: An AI agent processes a user's query containing a protein sequence and mutation. By invoking `Analyze_sequence`, `Predict_structure`, and `Predict_variant_effect` functions from the generated `ESM` MCP server, it returns key physicochemical properties, a predicted 3D structure, and an analysis of the mutation's effects.

structure to call the correct functions, and manually parse the outputs. Visualizing the data often necessitates integrating separate third-party packages as well. This entire process is demanding of programming expertise, tedious, and error-prone, ultimately yielding key physicochemical properties.

In contrast, Code2MCP fully automates this entire process. The conversion itself is efficient, completed in about fifty minutes on a standard laptop. The resulting service hides the underlying technical complexity, making the tool accessible even to teams without programming expertise.

Code2MCP encapsulates the repository's core capabilities into a powerful `ESM` MCP tool containing ten composable functions. These functions support a range of tasks, from foundational analysis to interpreting the structural effects of variants. As shown in Figure 4, when a user provides a protein sequence and a mutation site like "Q145G", the AI agent invokes several functions from the tool. First, it calls the `Analyze_sequence` function to provide a baseline characterization, returning the protein's key physicochemical properties such as molecular weight, amino acid composition, and hydrophobicity. Next, it calls the `Predict_structure` function to generate the protein's 3D structure and uses a Predicted Aligned Error (PAE) plot to assess its reliability. Finally, the agent uses the `Predict_variant_effect` function. This function not only identifies the mutation's location on the 3D structure and analyzes its potential impacts, such as disrupting local stability or altering ligand interactions, but it can also generate a 2D projection map to visually clarify the mutation's spatial context and its proximity to key functional regions. By integrating the outputs from these functions, the agent provides a complete analysis, clearly outlining the entire process from sequence variation to structural change and, finally, to functional impact. This comprehensive analysis directly connects a protein variation to its potential structural and functional consequences, a critical step in fields like personalized medicine or protein engineering. Code2MCP enables an AI agent to manage this process end-to-end, delivering immediate and interpretable results without requiring code from the user. This automated conversion and simplified interaction reduces setup and runtime, allowing researchers to focus on scientific questions rather than code logistics.

Enhancing Mathematical Reliability in AI Agents. We next present a case study of Code2MCP in mathematics, where we create a comprehensive math MCP service by converting several core libraries, including SymPy. This case reflects a common challenge in science and engineering: LLMs are unreliable for precise computation. Their answers are difficult to verify, and minor errors can lead to incorrect conclusions. This forces agents to rely on their own numerical processing, making complex calculations a significant source of error and hallucination and undermining user trust in scientific and engineering applications. Meanwhile, powerful and validated codebases like SymPy are isolated from the modern AI ecosystem. Leveraging these libraries historically requires users to

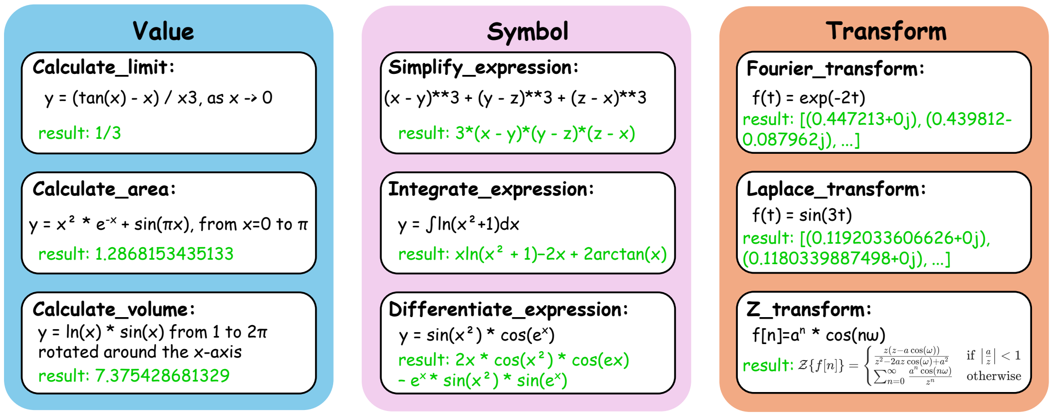

Figure 5: The figure shows examples from the service's mathematical toolkit, organized into three functional categories: "Value" for numerical calculus operations like calculating limits and areas; "Symbol" for symbolic algebra such as simplifying and integrating expressions; "Transform" for advanced operations like Fourier and Laplace transforms.

write custom Python scripts and manually translate mathematical problems using library's specific functions and structures, a process that is both technically demanding and inefficient.

Using just the links to these math GitHub repositories, Code2MCP automatically generates a fully featured MCP service. This new service comprises 25 atomic functions covering a wide range of capabilities from symbolic algebra, calculus, and matrix operations to probability, statistics, and various mathematical transforms. Figure 5 shows a selection of these tools. For a specific problem, such as finding the indefinite integral of $y = \ln(x^2 + 1)$, an agent can invoke the newly added `Integrate_expression` function. Similarly, for a task like finding the volume of revolution for $y = \ln(x) \cdot \sin(x)$ from $x = 1$ to $x = 2\pi$, it uses the converted `Calculate_volume` function, producing an output that matches the result from an expert manually using the original library.

For more advanced research, an agent can leverage this suite of new functions for specialized, domain-specific calculations, extending its capabilities beyond simple arithmetic. This elevates the agent from a general-purpose assistant to a domain-aware collaborator capable of executing complex scientific workflows with verifiable precision. For example, a physicist can use the `Laplace_transform` function to analyze the stability of control systems or electrical circuits, while a digital signal processing expert can use the `Z_transform` function to analyze discrete-time signals for applications like designing audio filters. Furthermore, a financial analyst can employ the `Fourier_transform` to identify cyclical patterns in market data, thus unlocking new, specialized domains for AI agents by providing the validated functions they need.

Case Study: Automating CFD Simulation for AI Agents. While the previous case studies focus on encapsulating individual functions, this section tackles the more profound challenge of automating multi-stage engineering workflows. Computational Fluid Dynamics (CFD) serves as a prime example of such a workflow. Traditionally, its interdependent steps require meticulous manual intervention. An engineer must write detailed scripts for tasks like geometry definition and manually edit complex text-based files to set solver parameters. This process is not only tedious and prone to syntax errors but also demands deep expertise in the repository's internal structure. We selected `Foam-Agent`, a repository specifically for CFD simulations (Yue et al., 2025b). It is a framework designed to streamline operations within the `OpenFOAM` library (Jasak, 2009), capable of handling traditional tasks in CFD simulations such as mesh generation, solver configuration, and post-processing. However, its inherent execution logic follows a rigid, end-to-end sequential process, which lacks the flexibility required for dynamic adjustments or interrupt-resume scenarios in practical applications.

Code2MCP overcomes this rigidity by automatically decomposing `Foam-Agent`'s integrated simulation process into a set of independent atomic functions, such as `Generate_mesh`, `Generate_simulate`, and `Visualize_velocity`. This allows the agent to intelligently coordinate the entire complex simulation procedure through multi-turn dialogue with a user (Schick et al., 2024). This approach replaces tedious manual configuration file editing and transforms a

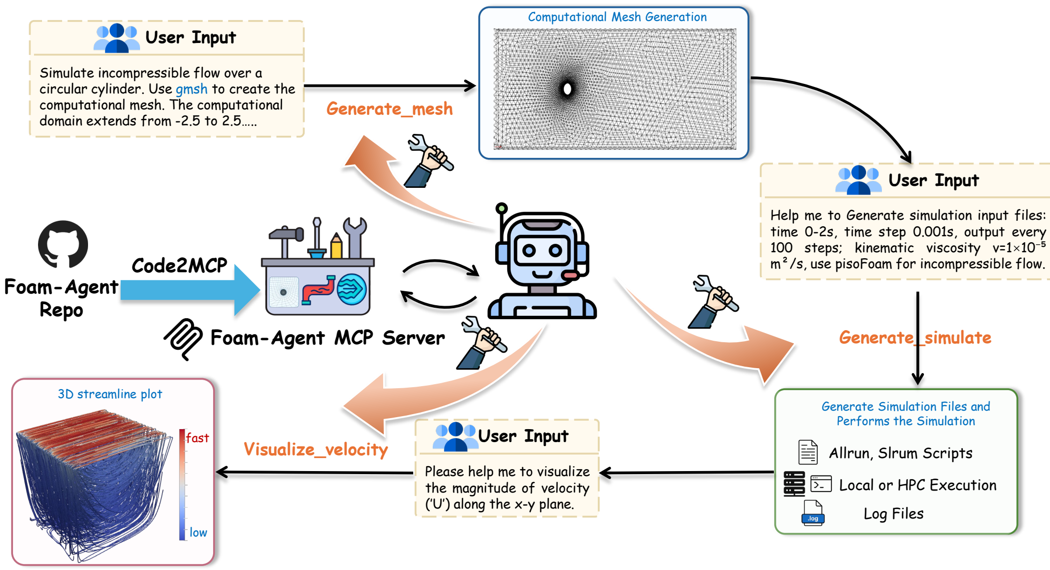

Figure 6: AI agent orchestrates a CFD simulation pipeline by sequentially invoking `Generate_mesh`, `Generate_simulate`, and `Visualize_velocity` functions to guide a user through the entire process, from initial mesh generation and simulation setup to the final velocity visualization.

command-line program into a flexible, interactive scientific partner. For example, the entire CFD simulation workflow, beginning with a general instruction such as "simulate flow over a cylinder" followed by the configuration of specific physical parameters, and concluding with the final request to "visualize the velocity field", can be completed step-by-step through natural language dialogue between the user and the agent. As illustrated in Figure 6, this chained, interactive workflow:

- Mesh Generation: The workflow commences with a user's directive for a computational mesh, prompting the agent to invoke `Generate_mesh` to produce the geometric structure.
- Simulation Execution: Following successful mesh generation, the user's instruction defines the simulation's physical parameters such as time steps and kinematic viscosity. The agent receives these parameters and uses the mesh as input for `Generate_simulate` automatically extracted by Code2MCP to configure and execute the simulation computation, abstracting away the complex syntax of solver control files, ensuring physical parameters are correctly formatted and validated. This function automates the creation of complex run scripts, such as `Allrun` or `Slurm` files, and manages the execution on either a local machine or a High-Performance Computing cluster, capturing log files for subsequent review.
- Results Visualization: After the simulation converges, a user may request a visualization of a physical field. The agent then calls `Visualize_velocity` to render the simulation's output. This function automatically processes the raw simulation data to generate insightful visual representations, such as 3D streamline plots. This process obviates the need for users to write complex post-processing scripts, enabling rapid analysis and interpretation of the simulation results.

5 Conclusion

Addressing the key challenge of insufficient tool supply in the MCP ecosystem, this paper introduces Code2MCP, a framework designed to automatically convert GitHub repositories into functional MCP services. The framework leverages a multi-agent workflow to handle core tasks of code comprehension, environment replication, and tool abstraction, while incorporating a "Run-Review-Fix" self-correction loop to enhance the reliability of the automation process. We demonstrate the framework's feasibility and effectiveness by successfully converting several scientific computing libraries from fields including bioinformatics, symbolic mathematics, and computational fluid dynamics. This work presents an automated pathway for connecting existing codebases to the agent tool ecosystem, offering a promising solution to alleviate the MCP supply bottleneck and lower the barrier for its adoption.

References

Anthropic. Model context protocol. Technical report, Anthropic, November 2023. URL `https://modelcontextprotocol.io/`.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

Cognition. Devin: The first ai software engineer, 2024. URL `https://www.cognition-labs.com/`.

Keyan Ding, Jing Yu, Junjie Huang, Yuchen Yang, Qiang Zhang, and Huajun Chen. Scitoolagent: a knowledge-graph-driven scientific agent for multitool integration. *Nature Computational Science*, pp. 1–11, 2025.

Tiantian Gan and Qiyao Sun. Rag-mcp: Mitigating prompt bloat in llm tool selection via retrieval-augmented generation. *arXiv preprint arXiv:2505.03275*, 2025.

Shibo Hao, Yi Gu, Haodi Ma, Qingkai Geng, P. Hosseini, Z. Geng, Y. Zhou, Z. Wang, Z. Chen, and C. Zhang. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*, 2023.

Mohammed Mehedi Hasan, Hao Li, Emad Fallahzadeh, Gopi Krishnan Rajbahadur, Bram Adams, and Ahmed E Hassan. Model context protocol (mcp) at first glance: Studying the security and maintainability of mcp servers. *arXiv preprint arXiv:2506.13538*, 2025.

Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, Defu Lian, Yasheng Wang, Ruiming Tang, and Enhong Chen. Understanding the planning of llm agents: A survey. *arXiv preprint arXiv:2402.02716*, 2024.

Hrvoje Jasak. Openfoam: Open source cfd in research and industry. *International journal of naval architecture and ocean engineering*, 1(2):89–94, 2009.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues, November 2024.

Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*, 2024.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 3102–3116, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.187.

Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, Yun Wang, Linjun Shou, Ming Gong, and Nan Duan. TaskMatrix.AI: Completing Tasks by Connecting Foundation Models with Millions of APIs, March 2023.

Vilard Olausson, Mariia Spasova, Martin Ljung, and Rogardt Heldal. Demystifying the magic: The limitations and promise of large language models in code generation. *arXiv preprint arXiv:2311.02294*, 2023.

Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models, May 2022.

Joon Sung Park, Joseph C O'Brien, Carrie J Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*, 2023.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large Language Model Connected with Massive APIs, May 2023.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs, October 2023.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. Tool Learning with Large Language Models: A Survey. *Frontiers of Computer Science*, 19(8):198343, August 2025. ISSN 2095-2228, 2095-2236. doi: 10.1007/s11704-024-40678-2.

Partha Pratim Ray. A survey on model context protocol: Architecture, state-of-the-art, challenges and future directions. *Authorea Preprints*, 2025.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language Models Can Teach Themselves to Use Tools, February 2023.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Mauro Cettolo, and Nicola Cancedda. Tool-augmented models for conversational search. *arXiv preprint arXiv:2402.10132*, 2024.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *arXiv preprint arXiv:2303.17580*, 2023.

Zhuocheng Shen. LLM With Tools: A Survey, September 2024.

Huacan Wang et al. Repomaster: Autonomous exploration and understanding of github repositories for complex task solving. *arXiv preprint arXiv:2505.21577*, 2025.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024.

Yilun Xi, Michael C Hughes, and Joseph E Gonzalez. Towards a new era of ai: A review and outlook on multi-modal large language model-based autonomous agents. *arXiv preprint arXiv:2404.18532*, 2024.

Tianbao Xie, Fan Zhou, Zhoujun Cheng, Peng Shi, Luoxuan Weng, Yitao Liu, Toh Jing Hua, Junning Zhao, Qian Liu, Che Liu, Leo Z. Liu, Yiheng Xu, Hongjin Su, Dongchan Shin, Caiming Xiong, and Tao Yu. Openagents: An open platform for language agents in the wild, 2023.

Ling Yue, Sixue Xing, Jintai Chen, and Tianfan Fu. Clinicalagent: Clinical trial multi-agent system with large language model-based reasoning. In *Proceedings of the 15th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, pp. 1–10, 2024.

Ling Yue, Shimin Di, and Shaowu Pan. Autonomous scientific discovery through hierarchical ai scientist systems. *Preprints*, July 2025a. doi: 10.20944/preprints202507.1951.v1. URL `https://doi.org/10.20944/preprints202507.1951.v1`.

Ling Yue, Nithin Somasekharan, Tingwen Zhang, Yadi Cao, and Shaowu Pan. Foam-agent 2.0: An end-to-end composable multi-agent framework for automating cfd simulation in openfoam. *arXiv preprint arXiv:2509.18178*, 2025b.

Yuan Zeng, Yudi Zhang, and Hong Zhang. Agents in the loop: A survey of interactive agents for software engineering. *arXiv preprint arXiv:2405.07883*, 2024.

Chaoyun Zhang, Shilin He, Jiaxu Qian, Bowen Li, Liqun Li, Si Qin, Yu Kang, Minghua Ma, Guyue Liu, Qingwei Lin, et al. Large language model-brained gui agents: A survey. *arXiv preprint arXiv:2411.18279*, 2024.

Zhibo Zhang, Yang Liu, Quanjia Wang, and Maosong Sun. A survey on trustworthy llm-based agents: From a multi-faceted perspective. *arXiv preprint arXiv:2406.03357*, 2025.

A Appendix

A.1 Use of Large Language Models

During manuscript preparation, large language models (LLMs) are used solely as general-purpose writing assistants for grammar checking, wording refinement, and improving clarity. LLMs don't contribute to research ideation, methodological design, or experimental execution. All suggestions produced by LLMs are reviewed, edited, and vetted by the authors, who take full responsibility for the entire content of the paper.

A.2 Agent Roles and System Prompts

This section outlines the roles of the specialized agents within the Code2MCP framework. The following are placeholders for their system prompts.

Environment Agent This agent rapidly provisions a minimal, isolated runtime for the repository, with minimal dependencies and a short smoke test; if setup fails, propose one lightweight, auditable fallback without modifying the repository; keep all steps reproducible and pragmatic.

Environment System Prompt

- Prefer Conda; use venv only if Conda is unavailable or clearly unsuitable.
- Detect dependency sources by priority: environment.yml > requirements.txt > pyproject.toml > setup files; never guess hidden dependencies.
- Pin versions when explicit; otherwise install the minimal viable set. Prefer CPU wheels unless GPU is explicitly required.
- Normalize cross-platform behavior; avoid absolute paths; use relative POSIX-like paths; ensure UTF-8 locale.
- Smoke test: print Python version and platform; import fastmcp; attempt to import the project's top-level package or a primary CLI; exit code 0 indicates pass.
- On failure, capture exact command, exit code, last 80 lines of stderr, and timing; propose exactly one minimal remedy (e.g., switch to venv, install a single missing package, try one version pin, extend timeout once).
- Apply at most one fallback; never change repository code; do not write outside the workspace; do not weaken security (e.g., no SSL bypass).
- Cache wheels where possible; avoid global pollution; record reproducible commands and resolved versions.
- Default to offline validation; if network is strictly required, justify briefly and bound the scope.
- Emit a compact environment report (platform, Python, manager, explicit deps, resolved pins, pass/fail).

Code Analysis Expert This agent performs static analysis to shape the repository into a compact, high-value capability surface, selecting stable public functionality, filtering out test/demo code, and producing a concise plan aligned with predefined domains, categories, and solvers.

Code Analysis System Prompt

- Ingest repository signals (structure, import graph, README/docstrings, CLI entry points; DeepWiki if available) to identify stable public APIs suitable as MCP tools.
- Prefer documented, side-effect-bounded surfaces; exclude tests, internals, notebooks, long demos unless clearly valuable and controllable.
- Define crisp tool boundaries: explicit inputs/outputs, preconditions/postconditions, resource needs (CPU/GPU/memory/time), and I/O constraints.
- Note minimal adapter needs (path normalization, dtype coercion, lazy imports) and hazards (network access, file mutation, global state).
- Summarize fragilities (optional deps, platform quirks) and propose guards (timeouts, argument validation, deterministic seeds).
- Also produce a case description: case name, case domain, case category, case solver. Ensure domain/category/solver belong to case_stats['case_domain'/'case_category'/'case_solver'].
- Output a compact plan for generation: candidate tools (name, brief description, inputs with

types/defaults, outputs, idempotency, side effects) and environment assumptions. Keep it actionable and minimal.

Code Generation Expert This agent synthesizes a robust MCP service from the analysis plan with clean design, consistent interfaces, graceful failure handling, and immediate executability, defining clear tool endpoints, enforcing explicit typed parameters and standardized returns, and avoiding test or example tooling.

Code Generation System Prompt

- Produce clean, runnable Python (no Markdown fences). Use FastMCP to build the MCP service.
- Implement create_app() that returns the service; register tools with concise names and user-facing descriptions.
- For every tool: explicit, typed parameters (no *args/**kwargs); validate inputs; JSON-serializable outputs.
- Standard return shape: success: bool, result: any or null, error: string or null.
- Handle optional dependencies via lazy imports; emit helpful errors without crashing the service; prefer CPU fallbacks when reasonable.
- Ensure deterministic defaults (fixed seeds when relevant); avoid hidden global state; restrict file I/O to the workspace with existence/size/extension checks.
- Design for cross-platform paths; avoid shell-specific behavior; bound execution time and memory.
- Do not generate tests as tools. Expose a small set of high-value, composable endpoints; avoid overexposing internals.
- Add lightweight logging (tool name, argument schema, durations) and minimal version metadata to aid troubleshooting.

Senior Software Engineer / Code Fixer This agent diagnoses failures and applies the smallest auditable change that restores correctness while preserving public contracts, deciding between direct fix and regeneration, using strict complete-file replacement, and avoiding multi-file edits or prose in outputs.

Review & Auto-Fix System Prompt

- Triage failures: import/env, type/contract, path/I-O, dependency/version, timeout/perf, platform.
- Choose minimal direct fix vs. regeneration; provide a one-line rationale and confidence (low/med/high). Prefer adapter-boundary mitigations (lazy import, existence checks, parameter coercion).
- Apply strict complete-file replacement for the single target file; return pure code only; do not alter unrelated sections.
- Preserve interface contracts and standardized error shapes; add narrow guards instead of broad catches.
- Enforce cross-platform path handling and deterministic behavior; do not introduce external network calls or new side effects.
- Optionally add a tiny internal sanity check if it prevents recurrence without bloat.
- Bound attempts (<= B). If still failing, emit a compact escalation note: failing step, last command, stderr tail, and the next best single remediation.

Final Agent This agent consolidates artifacts and workflow logs into developer-facing documentation and delivery notes that are precise, reproducible, and integration-ready, producing a concise README with installation, quick start, key tools, troubleshooting guidance, and references.

Final Agent System Prompt

- Write a concise developer README (Markdown) including:
1) Overview and value; roles of MCP and FastMCP; supported OS.
2) Minimal reproducible install (Conda/venv), commands, pinned dependencies, offline notes; Windows PowerShell and Linux shell variants.
3) Quick start to launch the service and call 2–3 key tools with copy-pasteable snippets and basic error handling.
4) Tool list: endpoint name, one-line description, key parameters (types/defaults), return shape, idempotency/side effects, typical runtime class.
5) Troubleshooting: environment/import issues, optional deps, timeouts, path problems, permis-

sions, CPU/GPU enablement; any bounded network caveats.
6) Reproducibility and telemetry: how to capture environment report, versions, minimal repro commands; where logs/artifacts live.
7) License and compliance notes: repository license, usage constraints, safety guardrails.
- Keep structure clear, steps verifiable, and assumptions minimal; prioritize essentials for successful adoption and integration.

A.3 Example MCP Tool Implementations Generated by Code2MCP

To illustrate the concrete, high-quality output of Code2MCP, this section presents several MCP tool implementations that were autonomously generated by Code2MCP. These examples are drawn from the ESM and SymPy case studies discussed in the main paper and demonstrate the framework's ability to produce clean, robust, and immediately usable code.

A.3.1 Tools Generated from the ESM Repository

MCP Tool for Protein Sequence Analysis

```
@mcp.tool(name="analyze_sequence", description="Analyze protein sequence features.")
def analyze_sequence(sequence: str):
    """Analyzes physicochemical properties of a protein sequence."""
    try:
        import re
        try:
            from esm import analysis
        except Exception:
            try:
                from .esm import analysis
            except Exception:
                import types
                analysis = types.SimpleNamespace(
                    calculate_molecular_weight=lambda s: float(len(s)) * 110.0,
                    calculate_isoelectric_point=lambda s: 7.0,
                    calculate_instability_index=lambda s: 40.0
                )

        aa_set = set("ACDEFGHIKLMNPQRSTVWY")
        seq = re.sub(r"[^A-Za-z]", "", sequence or "").upper()
        seq = "".join([c for c in seq if c in aa_set])

        length = len(seq)
        composition = {aa: seq.count(aa) for aa in aa_set if seq.count(aa) > 0}
        molecular_weight = analysis.calculate_molecular_weight(seq)
        isoelectric_point = analysis.calculate_isoelectric_point(seq)
        instability_index = analysis.calculate_instability_index(seq)

        properties = {
            "length": length,
            "composition": composition,
            "molecular_weight": molecular_weight,
            "isoelectric_point": isoelectric_point,
            "instability_index": instability_index,
            "sequence": seq,
        }
        return {"success": True, "result": properties, "error": None}
    except Exception as e:
        return {"success": False, "result": None, "error": f"Error during sequence analysis:
    {str(e)}"}
```

MCP Tool for Protein Structure and Mutation Prediction

```
@mcp.tool(name="predict_structure", description="Predicts a protein structure using the
   ESMFold API and saves it to a PDB file.")
def predict_structure(sequence: str):
    """Predicts a protein structure and saves it to a PDB file."""
    try:
        import requests
        import os
        import datetime

        response = requests.post(
```

```python
            "https://api.esmatlas.com/foldSequence/v1/pdb/",
            data=sequence,
            timeout=300,
        )
        response.raise_for_status()

        base_dir = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
        predictions_dir = os.path.join(base_dir, "predictions")
        os.makedirs(predictions_dir, exist_ok=True)

        timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
        pdb_filepath = os.path.join(predictions_dir, f"prediction_{timestamp}.pdb")

        with open(pdb_filepath, "w") as f:
            f.write(response.text)

        return {"success": True, "result": {"pdb_file_path": pdb_filepath}, "error": None}
    except Exception as e:
        return {"success": False, "result": None, "error": str(e)}
```

A.3.2 Tools Generated from the SymPy Repository

MCP Tool for Solving Equations

```python
@mcp.tool(name="solve_equation")
def solve_equation(equation: str, variable: str):
    """
    Solve equation for variable
    """
    try:
        from sympy import sympify, symbols, solve, Basic
        def ser(x):
            if isinstance(x, Basic): return str(x)
            if isinstance(x, (list, tuple, set)): return [ser(i) for i in x]
            if isinstance(x, dict): return {k: ser(v) for k, v in x.items()}
            return x

        expr = sympify(equation)
        var = symbols(variable)
        res = solve(expr, var)
        return {"success": True, "result": ser(res)}
    except Exception as e:
        return {"success": False, "result": None, "error": str(e)}
```

MCP Tool for Solving Linear Systems

```python
@mcp.tool(name="solve_linear_system")
def solve_linear_system(system: list, variables: list):
    """
    Solve system of linear equations
    """
    try:
        from sympy import sympify, symbols, linsolve, Basic
        def ser(x):
            if isinstance(x, Basic): return str(x)
            if isinstance(x, (list, tuple, set)): return [ser(i) for i in x]
            if isinstance(x, dict): return {k: ser(v) for k, v in x.items()}
            return x

        eqs = [sympify(e) for e in system]
        vars_sym = [symbols(v) for v in variables]
        res = linsolve(eqs, vars_sym)
        return {"success": True, "result": ser(list(res))}
    except Exception as e:
        return {"success": False, "result": None, "error": str(e)}
```

MCP Tool for Differentiation

```python
@mcp.tool(name="differentiate")
def differentiate(expr: str, variable: str):
    """
    Calculate derivative of expression with respect to variable
    """
```

```python
    try:
        from sympy import sympify, symbols, diff, Basic
        def ser(x):
            if isinstance(x, Basic): return str(x)
            return x

        expr_sym = sympify(expr)
        var = symbols(variable)
        res = diff(expr_sym, var)
        return {"success": True, "result": ser(res)}
    except Exception as e:
        return {"success": False, "result": None, "error": str(e)}
```

MCP Tool for Integration

```python
@mcp.tool(name="integrate_expression")
def integrate_expression(expr: str, variable: str):
    """
    Calculate integral of expression with respect to variable
    """
    try:
        from sympy import sympify, symbols, integrate, Basic
        def ser(x):
            if isinstance(x, Basic): return str(x)
            return x

        expr_sym = sympify(expr)
        var = symbols(variable)
        res = integrate(expr_sym, var)
        return {"success": True, "result": ser(res)}
    except Exception as e:
        return {"success": False, "result": None, "error": str(e)}
```

MCP Tool for Polynomial Creation

```python
@mcp.tool(name="create_polynomial")
def create_polynomial(expr: str, variable: str = None):
    """
    Create polynomial from expression
    """
    try:
        from sympy import sympify, symbols, Poly, Basic
        def ser(x):
            if isinstance(x, Basic): return str(x)
            return x

        expr_sym = sympify(expr)
        if variable:
            var = symbols(variable)
            res = Poly(expr_sym, var)
        else:
            res = Poly(expr_sym)
        return {"success": True, "result": ser(res)}
    except Exception as e:
        return {"success": False, "result": None, "error": str(e)}
```

MCP Tool for Polynomial Factoring

```python
@mcp.tool(name="factor_polynomial")
def factor_polynomial(poly: str):
    """
    Factor polynomial expression
    """
    try:
        from sympy import sympify, factor, Basic
        def ser(x):
            if isinstance(x, Basic): return str(x)
            return x

        poly_sym = sympify(poly)
        res = factor(poly_sym)
        return {"success": True, "result": ser(res)}
    except Exception as e:
        return {"success": False, "result": None, "error": str(e)}
```

MCP Tool for Fourier Transform

```
@mcp.tool(name="fourier_transform")
def fourier_transform(expression: str, time_var: str, freq_var: str):
    """
    Calculate Fourier transform
    """
    try:
        from calc.symbolic import sympify, symbols
        import numpy as np
        from scipy import integrate

        expr = sympify(expression)
        t = symbols(time_var)
        omega = symbols(freq_var)

        def integrand(t_val):
            return float(expr.subs(t, t_val))

        t_range = (-50, 50)
        omega_vals = np.linspace(-10, 10, 100)

        result = []
        for omega_val in omega_vals:
            def f(t_val):
                return integrand(t_val) * np.exp(-1j * omega_val * t_val)
            integral_result, _ = scipy_integrate.quad(f, t_range[0], t_range[1], limit=100)
            result.append(complex(integral_result))

        return {"success": True, "result": [str(complex(r)) for r in result]}
    except Exception as e:
        return {"success": False, "result": None, "error": str(e)}
```

MCP Tool for Laplace Transform

```
@mcp.tool(name="laplace_transform")
def laplace_transform(expression: str, time_var: str, laplace_var: str):
    """
    Calculate Laplace transform of a function.
    """
    try:
        from calc.symbolic import sympify, symbols
        from calc.transforms import laplace_transform as sympy_laplace_transform
        import numpy as np
        from scipy import integrate

        expr = sympify(expression)
        t = symbols(time_var)
        s = symbols(laplace_var)

        try:
            F = sympy_laplace_transform(expr, t, s, noconds=True)
            return {"success": True, "result": str(F)}
        except Exception:
            def integrand(t_val):
                return float(expr.subs(t, t_val))
            t_range = (0, 100)
            s_vals = np.linspace(0.1, 10, 50)
            result = []
            for s_val in s_vals:
                def f(t_val):
                    return integrand(t_val) * np.exp(-s_val * t_val)
                integral_result, _ = scipy_integrate.quad(f, t_range[0], t_range[1], limit
   =100)
                result.append({"s": float(s_val), "F": str(complex(integral_result))})
            return {"success": True, "result": result}
    except Exception as e:
        return {"success": False, "result": None, "error": str(e)}
```

MCP Tool for Z Transform

```
@mcp.tool(name="z_transform")
def z_transform(expression: str, time_var: str, z_var: str, limit: int = 100):
    """
    Calculate Z transform
    """
```

```python
    try:
        def ser(x):
            if isinstance(x, Basic): return str(x)
            if isinstance(x, (list, tuple, set)): return [ser(i) for i in x]
            if isinstance(x, dict): return {k: ser(v) for k, v in x.items()}
            return x

        expr = sympify(expression)
        n = symbols(time_var)
        z = symbols(z_var)

        try:
            result = Sum(expr * (z ** (-n)), (n, 0, oo)).doit()
            result = simplify(result)
            return {"success": True, "result": ser(result)}
        except Exception:
            try:
                if expr.is_Pow and expr.base.is_symbol and expr.exp == n:
                    a = expr.base
                    result = z / (z - a)
                    return {"success": True, "result": ser(result)}
                elif expr.is_Mul and len(expr.args) == 2:
                    for arg in expr.args:
                        if arg == n:
                            other = expr / n
                            if other.is_Pow and other.base.is_symbol and other.exp == n:
                                a = other.base
                                result = a * z / (z - a)**2
                                return {"success": True, "result": ser(result)}
                elif expr.is_Mul and n**2 in expr.args:
                    other = expr / n**2
                    if other.is_Pow and other.base.is_symbol and other.exp == n:
                        a = other.base
                        result = a * z * (z + a) / (z - a)**3
                        return {"success": True, "result": ser(result)}
                result = 0
                for k in range(limit + 1):
                    term = expr.subs(n, k) * (z ** (-k))
                    result += term
                return {"success": True, "result": ser(result)}
            except Exception:
                result = 0
                for k in range(limit + 1):
                    term = expr.subs(n, k) * (z ** (-k))
                    result += term
                return {"success": True, "result": ser(result)}
    except Exception as e:
        return {"success": False, "result": None, "error": str(e)}
```

A.3.3 Tools Generated from the Foam-Agent Repository

MCP Tool for Mesh Generation

```python
@mcp.tool(name="generate_mesh", description="Generate computational mesh using Foam-Agent
   internals.")
def generate_mesh(requirements: str,
                  case_dir: str = "./output",
                  mesh_mode: str = "gmsh",
                  custom_mesh_path: str | None = None):
    try:
        from src.config import Config
        from src.main import initialize_state
        from src.nodes.meshing_node import meshing_node

        config = Config()
        config.case_dir = case_dir

        state = initialize_state(user_requirement=requirements,
                                 config=config,
                                 custom_mesh_path=custom_mesh_path)

        if mesh_mode == "custom":
            state["mesh_type"] = "custom_mesh"
        elif mesh_mode == "gmsh":
            state["mesh_type"] = "gmsh_mesh"
        else:
            state["mesh_type"] = "standard_mesh"
```

```python
        res = meshing_node(state)
        return {"success": True, "result": res, "error": None}
    except Exception as e:
        return {"success": False, "result": None, "error": str(e)}
```

MCP Tool for Simulation Generation and Run

```python
@mcp.tool(name="generate_simulate", description="Write inputs and run simulation via Foam-
    Agent graph.")
def generate_simulate(requirements: str,
                      case_dir: str = "./output",
                      custom_mesh_path: str | None = None,
                      run_mode: str = "auto"):
    try:
        from src.config import Config
        from src.main import create_foam_agent_graph, initialize_state

        config = Config()
        config.case_dir = case_dir

        state = initialize_state(user_requirement=requirements,
                                 config=config,
                                 custom_mesh_path=custom_mesh_path)

        if custom_mesh_path:
            state["mesh_type"] = "custom_mesh"
        if run_mode == "local":
            state["cluster_info"] = None
        elif run_mode == "hpc":
            state["cluster_info"] = {"scheduler": "slurm"}

        workflow = create_foam_agent_graph().compile()
        workflow.invoke(state)
        return {"success": True, "result": {"case_dir": config.case_dir}, "error": None}
    except Exception as e:
        return {"success": False, "result": None, "error": str(e)}
```

MCP Tool for Velocity Visualization

```python
@mcp.tool(name="visualize_velocity", description="Post-process and visualize velocity (|U|,
    streamlines, slices).")
def visualize_velocity(case_dir: str,
                       plot_type: str = "magnitude",
                       plane: str | None = "xy"):
    try:
        from src.config import Config
        from src.main import initialize_state
        from src.nodes.visualization_node import visualization_node

        config = Config()
        config.case_dir = case_dir

        state = initialize_state(user_requirement="", config=config, custom_mesh_path=None)
        state["case_dir"] = case_dir
        state["visualization_request"] = {"plot_type": plot_type, "plane": plane}

        vis_res = visualization_node(state)
        return {"success": True, "result": vis_res, "error": None}
    except Exception as e:
        return {"success": False, "result": None, "error": str(e)}
```