Model Context Protocol (MCP) at First Glance: Studying the Security and Maintainability of MCP Servers

MOHAMMED MEHEDI HASAN, Queen's University, Canada
HAO LI, Queen's University, Canada
EMAD FALLAHZADEH, Queen's University, Canada
GOPI KRISHNAN RAJBAHADUR, Queen's University, Canada
BRAM ADAMS, Queen's University, Canada
AHMED E. HASSAN, Queen's University, Canada

Although Foundation Models (FMs), such as GPT-4, are increasingly used in domains like finance and software engineering, reliance on textual interfaces limits these models' real-world interaction. To address this, FM providers introduced tool calling—triggering a proliferation of frameworks with distinct tool interfaces. In late 2024, Anthropic introduced the Model Context Protocol (MCP) to standardize this tool ecosystem, which has become the de facto standard with over eight million weekly SDK downloads. Despite its adoption, MCP's AI-driven, non-deterministic control flow introduces new risks to sustainability, security, and maintainability, warranting closer examination.

Towards this end, we present the first large-scale empirical study of MCP servers. Using state-of-the-art health metrics and a hybrid analysis pipeline, combining a general-purpose static analysis tool with an MCP-specific scanner, we evaluate 1,899 open-source MCP servers to assess their health, security, and maintainability. Despite MCP servers demonstrating strong health metrics, we identify eight distinct vulnerabilities—only three overlapping with traditional software vulnerabilities. Additionally, 7.2% of servers contain general vulnerabilities, and 5.5% exhibit MCP-specific tool poisoning. Regarding maintainability, while 66% exhibit code smells, 14.4% contain ten bug patterns overlapping with traditional open-source software projects. These findings highlight the need for MCP-specific vulnerability detection techniques while reaffirming the value of traditional analysis and refactoring practices.

CCS Concepts: • **Software and its engineering** → **Empirical software validation**.

Additional Key Words and Phrases: Model context protocol, MCP, security, code smell, software bugs, maintainability

Authors' Contact Information: Mohammed Mehedi Hasan, mohammedmehedi.hasan@queensu.ca, Queen's University, Kingston, ON, Canada; Hao Li, Queen's University, Kingston, ON, Canada, hao.li@queensu.ca; Emad Fallahzadeh, Queen's University, Kingston, ON, Canada, emad.fallahzadeh@queensu.ca; Gopi Krishnan Rajbahadur, Queen's University, School of Computing, Kingston, Ontario, Canada, grajbahadur@acm.org; Bram Adams, Queen's University, Kingston, ON, Canada, bram.adams@queensu.ca; Ahmed E. Hassan, Queen's University, Kingston, ON, Canada, ahmed@cs.queensu.ca.

1 Introduction

Foundation Models (FMs), such as GPT-4 [99] and LLaMA [133], have revolutionized Artificial Intelligence (AI) applications through their powerful capabilities of language understanding, generation, and reasoning. Their widespread adoption is evident as they are increasingly integrated into real-world products and applications across domains, e.g., finance [145], healthcare [46], and software development [142], fundamentally transforming how we interact with technology and information. Despite this momentum, FMs inherently operate through isolated "textual" interfaces, limiting their ability to directly interact with dynamic, real-world data sources, APIs, and systems.

As an initial step to bridge this gap, FM providers have introduced capabilities such as function calling [85, 97] and tool calling [13], commonly known as *tool calling*, enabling real-world interactions—for example, booking an appointment using calendar tools. Leveraging these capabilities, AI and Agent frameworks, e.g., LangChain [73], Autogen [16], Pydantic-AI [3], CrewAI [40], Dify [47] and LlamaIndex [81], have introduced FM-based tool workflows for building AI applications that can reason about tasks, plan tool usage, and coordinate tool invocation. However, a significant challenge has emerged from this proliferation: each framework typically defines its own tool interface and data structure, e.g., a tool defined in one AI framework is not directly portable in another AI framework, leading to fragmentation that hampers interoperability, scalability, and consistent security practices.

To address these limitations, Anthropic introduced the Model Context Protocol (MCP) [10]—a universal, client-server protocol standardizing how AI applications expose tools to FMs. Analogous to how package managers (e.g., NPM, PyPI [23]) standardized software component reuse across diverse domains, MCP provides a unified, open standard allowing compliant clients to discover and invoke tools from a diverse ecosystem of MCP servers, thereby enhancing interoperability in AI tool usage. Since its introduction in late November 2024, major stakeholders—including Microsoft [86], OpenAI [98], Google [82], and Cloudflare [38]—have all embraced MCP to support AI applications. For instance, Microsoft's Azure MCP server[1] allows FMs to interact with Azure services, e.g., cloud storage and databases. As of May 2025, MCP's PyPI package receives approximately 1.8 million weekly downloads [12], and the corresponding NPM package sees about 6.9 million downloads per week [11]. This swift uptake by industry leaders and strong community engagement through open-source channels highlight MCP's growing centrality and further motivate the need to examine MCP servers through the lens of open-source software concerns such as health, sustainability, vulnerability, and maintainability.

Similar to other reusable software packages deployed in production environments, e.g., PyPI and NPM packages, the health, sustainability, security, and maintainability of MCP servers are critical to ensure long-term growth. Experience from other software ecosystems suggests that communities prioritize secure, widely adopted, well-documented packages [92]. However, early studies have demonstrated that FMs can be coerced into utilizing tools from vulnerable MCP servers to compromise user systems, commonly known as `tool poisoning`, leading to attacks such as credential theft and malicious code execution [62, 95]. For example, an MCP server can have a tool with file write permission, and by manipulating the FM, it can be tricked into adding some malicious code snippets in the `.bashrc` file to open a backdoor port for attackers [106]. Given that the whole point of FM tools is to give FMs autonomy in deciding when and which tool to use, the presence of potential security (and maintainability) issues in said tools has raised concerns among industry stakeholders [87].

However, despite these concerns, the current state of publicly available open-source MCP servers remains unexplored, mainly in terms of critical software dimensions, e.g., health, sustainability,

---

[1]https://github.com/Azure/azure-mcp

security, and maintainability. Specifically, there is limited empirical understanding of (i) how healthy and sustainable the MCP ecosystem is, (ii) whether different security vulnerability patterns are prevalent in these MCP servers, and (iii) what maintainability challenges, e.g., code smells and bugs, exist in those servers. These gaps are concerning, given the potential impact of unaddressed software issues. For instance, historically, we know that one Heartbleed vulnerability alone caused an estimated USD $500 million in damages globally [124] in 2014, and recent industrial reports suggest that mature organizations may spend up to £250,000 (USD $332,500) per system annually on software maintenance [54]. Without rigorous empirical analysis of these three dimensions, the community lacks the measures to mitigate such risks and issues proactively.

To bridge this gap, we conduct the first large-scale empirical study on health, sustainability, security vulnerabilities and maintainability issues across 1,899 MCP servers. To date (Jun 1, 2025) our dataset consists of 343 MCP servers from the official MCP collection [105] complemented by 1,556 MCP servers mined from open-source repositories hosted on GitHub. By analyzing these MCP servers, we aim to answer the following Research Questions (RQs):

RQ-0: How healthy and sustainable are MCP servers?

Motivation. As MCP is becoming a crucial part of the AI application ecosystem, it is essential to understand how healthy and sustainable the MCP ecosystem currently is. Prior studies have reported that different development (commits, CI adoption, etc.) and community metrics (contributor count, contributor reputation, issue activity, etc.) can act as indicators of health and long-term sustainability [39, 53, 146]. As development metrics reflect continuous maintenance activity and community metrics capture stakeholder engagement, we can understand the current state-of-the-art health and sustainability characteristics of MCP servers by studying these metrics and interpreting them in contrast to general OSS and Machine Learning(ML) software metrics.

Findings. MCP servers demonstrate strong development and community metrics with an average of **5.5 commits/week** (vs. traditional software 2.5 commits/week) and a slightly **higher CI adoption rate** (42.2% vs. traditional 40.3%), comparable median contributor count (2.0 vs. 2.0), but **significantly higher social reach** in terms of median follower count of the contributors (**129.6 followers** vs. traditional 37.3), and a slightly slower median issue resolution time (5.6 days vs. traditional 4.0 days).

RQ-1: To what extent do MCP servers contain security vulnerabilities?

Motivation. Prior studies have reported that at least 46% of Python packages contain security vulnerabilities [113] and the **average cost of some security incidents, e.g., data breach, can be USD up to $4.5 million** [63]. The potential risk may be even higher in the context of MCP servers, as these systems bridge foundation models with databases, file systems, and APIs, with humans handing off control to FM models. To understand the actual risks, we need to characterize the vulnerability patterns of MCP servers, analyze those against the known traditional software engineering vulnerabilities, and evaluate the effectiveness of current vulnerability detection tools in identifying MCP-specific issues.

Findings. To explore the vulnerability landscape of MCP servers, we apply a traditional vulnerability detector, SonarQube, and identify that **7.2% of MCP servers are affected by eight distinct vulnerability patterns**, with credential exposure being the most prevalent (3.6%). Only three of those vulnerabilities overlap with known ecosystem vulnerabilities. However, by applying an MCP-specific scanning tool [71], we uncover **5.5% MCP servers suffer from tool poisoning**, which indicates that traditional tools only find part of the whole landscape and we need more robust MCP-specific vulnerability detection techniques and tools.

RQ-2: To what extent do MCP servers contain maintainability issues?

Motivation. Prior studies have reported that maintainability issues, e.g., code smells, can be linked to bugs or fault-proneness [100]. Software development teams can spend up to 39% of maintenance effort on fixing these in mature projects [9]. As an emerging new technology, the maintainability landscape of MCP servers, e.g., code smells and bugs, is still unknown to the community. To understand the impact of these maintainability issues on MCP servers, we need to characterize the code smells and bugs and analyze those against the known traditional software engineering issues to evaluate the long-term impact.

Findings. MCP servers exhibit **similar code smell and bug patterns** like traditional software engineering domain. In the long term, we suspect these code smells, and bugs can incur significant maintainability issues in the MCP domain, as observed in traditional software engineering [54]. We report a taxonomy of ten code smells and nine bugs in MCP code. We also find that **66% MCP servers** have critical or blocker-level code smells, where **high cognitive complexity is the most prevalent smell** and **14.4% MCP servers have critical or blocker level-bugs**.

The main contributions of our paper are as follows:

(1) **Dataset**: We present the first curated dataset of open-source MCP servers [57], collected from both officially listed and mined GitHub repositories which can serve as a foundational asset for future research on MCP.

(2) **Analysis Framework and Baselines:** We develop and apply a hybrid analysis framework combining general-purpose static analysis tools (e.g., SonarQube) with emerging MCP-specific scanners (e.g., mcp-scan). Beyond tooling, we establish interpretive baselines for the health, security, and maintainability of emerging open-source projects, e.g., for MCP servers, through a structured literature review, which can serve as a baseline for future empirical studies across other new open-source software domains.

(3) **Implications**:
  • We present the first large-scale empirical study of the MCP ecosystem, examining its health, sustainability, security, and maintainability. We uncover general vulnerabilities in 7.2% of MCP servers and MCP-specific vulnerabilities in 5.5% of them. Additionally, we report the prevalence of maintainability issues, e.g., 66% of MCP servers have code smells, and 14.4% have bugs.
  • Although MCP-specific vulnerabilities can be more prevalent than traditional ones, they are hard to detect by both traditional and emerging MCP-specific tools, highlighting the need for dedicated research on MCP-specific vulnerability detection and mitigation techniques.
  • Maintainability issues of MCP servers, e.g., code smells and bugs, are closely related to traditional software engineering, suggesting that existing refactoring and debugging tools and techniques can be adapted for the MCP ecosystem.

The remainder of this paper is outlined as follows. Section 2 provides a motivating example for MCP. Section 3 provides background information. Section 4 gives an overview of related work. Section 5 describes the design of our study. Section 6 presents the results. Section 7 discusses the implications of our findings. Section 8 outlines threats to the validity of our study, and Section 9 concludes the paper.

2 A Motivational Example

Developing real-world AI applications with FMs demands seamless orchestration of diverse components. To motivate the role of MCP and it's challenges studied in this work, we present a common scenario faced by AI application developers while integrating FMs in the real world.

(a) AI Application developed with framework A using built-in search and stripe tool.

(b) A's tool is incompatible with framework B, needing to implement a custom stripe tool.

(c) Custom stipe tool made for framework B is incompatible with framework C.

(d) MCP: decoupling tools, prompts and resources from frameworks
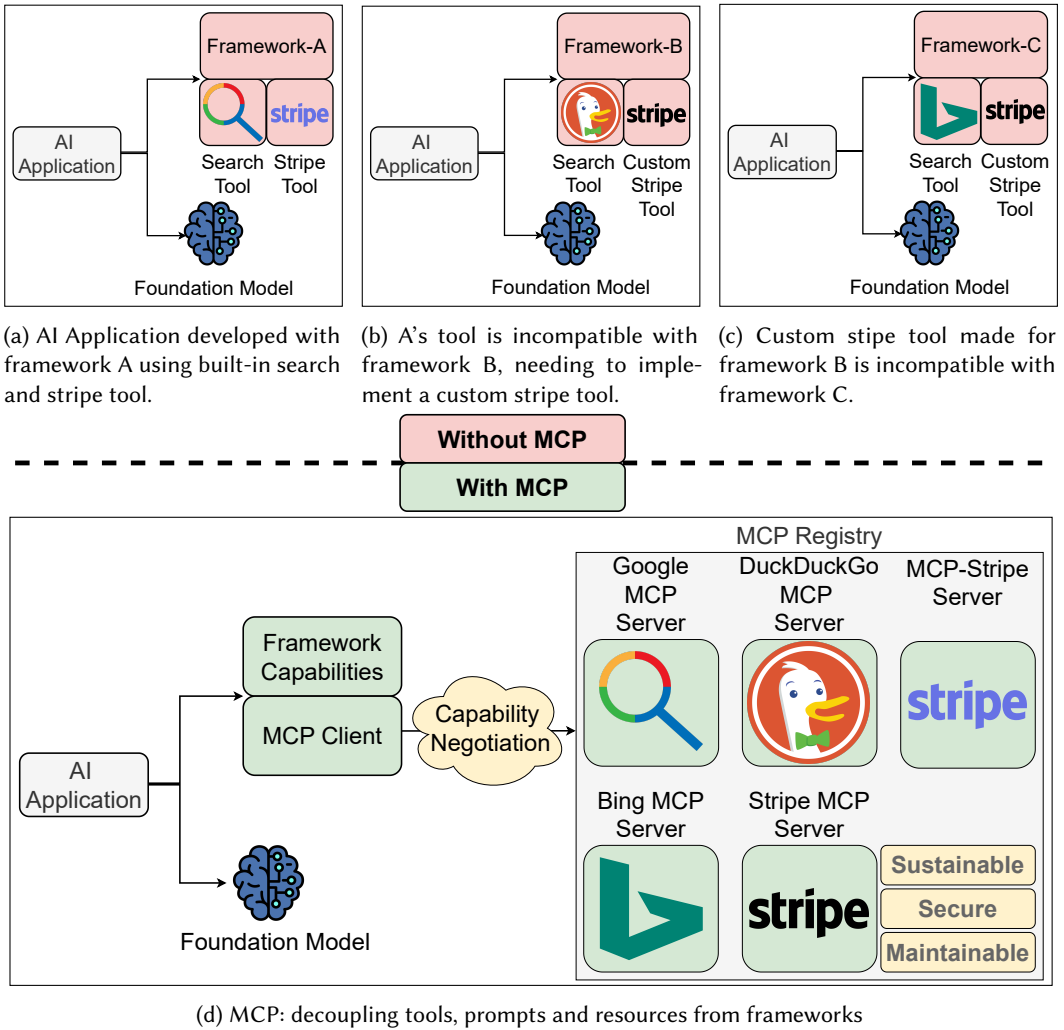
Fig. 1. A motivating example of developing FM-based AI applications. In (a), Alex developed an AI application using framework A and did not need any custom tools. In (b), when enhancing an existing application written in framework B, they had to build a custom stripe tool because B does not support A's built-in tools. In (c), they had to re-implement the same stripe tool again with a different interface to integrate it into the C framework. In contrast (d), where MCP servers offer a way to decouple tools from frameworks and enable interoperability — but raise new questions around sustainability, security, and maintainability.

Alex is an AI engineer responsible for building and maintaining FM-based AI applications. They have been tasked with developing a conversational shopping assistant that lets users chat about products, receive recommendations, and complete purchases via Stripe. Alex begins by exploring whether a single FM can handle the entire flow—from product discovery to payment—but quickly realizes that:

(1) **Finding products** requires a dedicated search mechanism (e.g., web search APIs).
(2) **Recommending products** needs an LLM to synthesize and rank the discovered products

(3) **Collecting payments** needs a payment gateway, e.g., Stripe integration.

Alex learned they need `tools`, a self-contained implementation of some functionalities that connect a model to existing APIs or data sources, to accomplish 1 and 3. After exploring several AI application frameworks that can coordinate different tools, Alex opted for framework A (for example, LangChain) as it has an extensive library of pre-built tools, including various search integrations (e.g., Google Search, Bing Search) and, importantly, a tool for Stripe integration.

Scenario 1: The Good. Using framework A, Alex builds an AI application to orchestrate a user purchase workflow. When a user expresses a requirement, the system: (1) sends the query along with available tools (e.g., search and Stripe) to the FM; (2) invokes the search tool when the FMs instructs to do so (e.g., with keywords, brands, sites) and returns the results; (3) presents the FM's synthesized results to the user and collects the product choice; and (4) executes the Stripe tool, based on the FM's instruction to complete the payment.

Scenario 2: The Bad. The shopping assistant quickly gained popularity, particularly its integrated payment feature, which encouraged a new requirement: integrating Stripe-based donation collection to an existing fundraising AI application built with coordination framework B, for example LlamaIndex. As B did not have a Stripe tool, Alex attempted to reuse the A's Stripe tool but found it incompatible; A's tool interfaces, invocation methods, and data handling mechanisms fundamentally differed from B 's expectation. Consequently, Alex ended up re-implementing a custom Stripe tool for B.

Scenario 3: The Ugly. Soon after, another team required to add Stripe payments to their AI analytics app built with framework C, e.g., Pydantic-AI, to collect premium membership fees. Again, no built-in tool existed. Alex attempted to adapt their custom B Stripe tool, but C's tool interface prevented reuse, forcing them to consider a third bespoke Stripe implementation.

Through this series of experiences, Alex realized the critical root problem: a fundamental lack of tool interoperability and standardization within the FM-based AI application ecosystem.

Scenario 4: The Promise. To address this, Alex explores the emerging Model Context Protocol (MCP), which standardizes how FMs discover and invoke external capabilities via standard interfaces without any hard coupling with the framework behind the AI application. After MCP's release, Alex finds several Stripe MCP servers from open-source community. They realize that they can now integrate these directly into any of their applications, as long as the applications leverage the MCP protocol, finally eliminating duplicated effort and code.

Scenario 5: The Challenge. However, payment processing is highly sensitive for both customers and Alex's company. As a revenue-generating application, they require the Stripe MCP server to be sustainable, e.g., regularly developed and bug-fixed. Moreover, they must guard against security vulnerabilities compromising sensitive customer data (e.g., credit card numbers) and the maintenance issues of downstream applications. Given the availability of multiple Stripe MCP servers from various communities, they are now perplexed about how to determine:

(1) Which MCP servers are the most healthy and sustainable for their critical AI applications? (**RQ-0**)
(2) How can they verify an MCP server's security to protect sensitive customer data (e.g., credentials, credit card details)? (**RQ-1**)
(3) How can they assess an MCP server's maintenance quality, such as its likelihood of having bugs or smells? (**RQ-2**)

In this study we want to explore these challenges through quantative and qualitative analysis.

3 Background

3.1 Tool Landscape for FM-based AI Applications

3.1.1 Tools for FM. Foundation models (FMs) are pre-trained on massive datasets to internalize broad linguistic patterns and world knowledge. They rely entirely on their statically encoded parameters (weights) at inference time and cannot access information beyond their pre-training data [24, 26]. Though Retrieval-Augmented Generation (RAG) augments FMs with domain-specific textual context [77], FMs are still confined to generating only text output. To interact with the real world, FM-based AI applications use tools, which are external interfaces performing operations on behalf of the model, ranging from web search, database queries, and API invocation to code execution and hardware control. These primitives enable AI applications to fetch real-time data, interact with user interfaces, and trigger real-world actions. However, setting up those tools for proper use can be challenging and time-consuming [58].

3.1.2 AI Frameworks. Several tool-centric frameworks have emerged in recent years to support the development of FM-based AI applications and AI agents. Examples include LangChain [73], Autogen [16], and LlamaIndex [81]. These frameworks provide infrastructure, abstractions, and orchestration mechanisms for building AI applications and agents to reason about tasks, plan tool usage, and coordinate tool invocation to pursue user-defined goals.

3.1.3 Tool Workflow. Most frameworks offer built-in tools and provide support for creating custom tools to enable AI applications to specify a step-by-step tool workflow. As illustrated in our motivating example Figure 1a, once the user expresses an "intent," the AI application formats the request, retrieves the available tools, and bundles them into a prompt for the FM. The FM then plans an execution path and guides the AI application through a series of tool calls and interactions to fulfill the tasks, e.g., using a search tool to find products, collecting user preferences via interaction, and calling a payment API through Stripe tool to complete the purchase journey.

3.1.4 Shortcomings of Framework-Specific Tools. While these frameworks support FM-tool integration, they also introduce significant interoperability challenges due to their siloed operation [58]. Each framework defines its schema for tool registration, invocation, and response formatting, leading to compatibility issues. For example, a Stripe tool implemented for LangChain may not function seamlessly with AutoGen or LlamaIndex because of differences in input specifications and data contracts. Moreover, tools are often tightly coupled with their originating frameworks and lack standardized discoverability mechanisms, making reuse difficult. So, the application built on a specific framework is limited to using the tools provided only by that framework and misses the opportunity to integrate any other outside tools. This fragmentation mirrors early challenges in software engineering, which were eventually addressed through standardized packaging systems like NPM [152] or PyPI [104], enabling consistent reuse, interoperability, and lifecycle management.

3.2 Model Context Protocol(MCP)

To address the lack of standardization and enable reusable tools across frameworks, Anthropic introduced the **Model Context Protocol (MCP)** [10], a universal protocol for tool discovery and invocation in FM-based AI applications and AI agents. MCP decouples tools from frameworks by introducing a client-server architecture where tools are hosted in standalone "MCP servers", and AI applications or agents interact with them through a standardized "MCP client" as illustrated in Figure 2. MCP empowers tool vendors to publish any services with a defined interaction method, e.g., SDK, API, or any other protocol, as standalone MCP servers just once, instantly making those capabilities discoverable and invocable by any MCP-compliant AI application. This plug-and-play model eliminates code duplication, drastically lowering development overhead and accelerating

the adoption of new tools across FM-based AI application ecosystems. Additionally, MCP enables AI application developers to use a variety of tools with any MCP-aware model, ensuring high reusability.
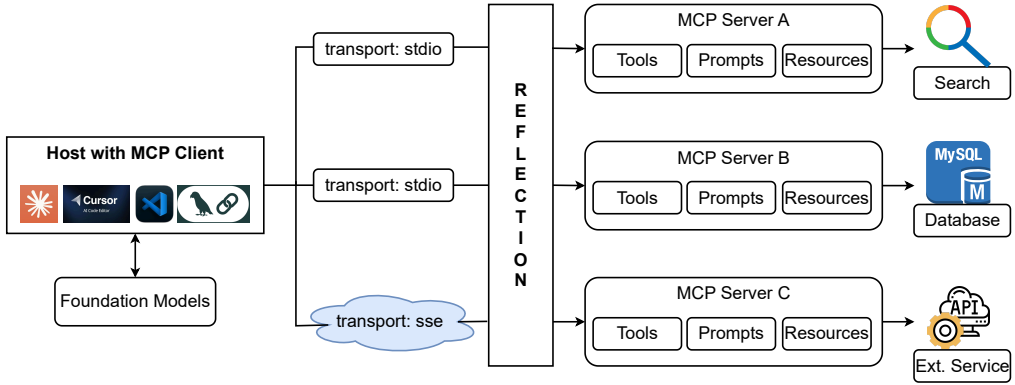

Fig. 2. High-level overview of MCP client-server architecture

3.2.1   MCP Workflow. In a typical MCP workflow, the AI application delegates the tool invocation responsibilities to an MCP client instead of using framework-specific tools. The MCP client begins by querying all installed MCP servers using reflection (more details below) to discover available tools, e.g., search tools and Stripe payment tools from our motivating example. It then extracts the tools' data, e.g., name, description, and interface signature, constructs a prompt incorporating these, and finally sends that to the FM. The FM analyzes the request, determines the sequence of tool invocations, and instructs the client about which tool to invoke with what parameters. During tool orchestration, the MCP client obtains the user's permission before invoking any tool, e.g., searching for products or making payments. Importantly, there is no tight coupling between tools and or AI applications, allowing tools to be reused and accessed across different AI applications and frameworks.

3.2.2   MCP Server. An MCP Server wraps the functionalities of one or more external services or data sources and exposes those in a standardized manner via the MCP protocol. It handles input validation, tool execution, and response formatting. Each server can host one or more tools and run locally or remotely. For example, a real-world Stripe MCP [131] server needed for our motivational example exposes tools for reading balance, creating invoices, paymentLinks and issuing refund.

3.2.3   Reflection. Reflection refers to an MCP server's ability to dynamically expose its capabilities and metadata obtained from the docstring or schema description of tools to the client, enabling runtime capability discovery. For instance, a client can query the Stripe MCP server [131] to discover available tools ("balance.read", "payment links.create", "refunds.create" etc.), read the description of those tools, inspect required parameters, and validate data types—without any prior hard-coding.

3.2.4   Transport. MCP supports local ("StdIO") and remote ("SSE over HTTPS") communication protocols. This makes it adaptable to various environments, from embedded use in desktop applications to cloud-based server deployments.

3.2.5   MCP Client. An MCP Client manages the communication between the FM and one or more MCP servers. It retrieves tool schema descriptions, mediates FM-guided calls, and ensures user approval for different tool execution.

3.2.6   MCP Registries. Inspired by the success of package managers (PyPI, NPM), the MCP community has developed several MCP Registries, e.g., Smithery [126], Glama [2], and Cloudflare Workers [38]. These registries serve as searchable registries where developers can publish and discover MCP servers. For example, at least two "Stripe MCP Servers" are available on Glama, providing similar payment integration with Stripe, which can be utilized by any AI application to accomplish the payment example we have described in the motivating example[2], [3].

4   Related Work
4.1   Health and Long-Term Sustainability of OSS

Since the introduction of the bazaar model of software development [109], open-source software (OSS) has evolved from a fringe movement to a foundational element of global software software ecosystem [36]. Despite its success, concerns around the long-term sustainability of OSS projects have persisted [39]. While there are community initiatives (e.g., CHAOSS [53]) to identify and consolidate the health and sustainability metrics for open-source projects, academic research has also sought to identify early warning signs of project failure, e.g., lack of contributions, insufficient community engagement, or maintainability bottlenecks [17, 39]. On the other hand, multiple studies identify factors like user base, language translation, responsibility distribution, and modularity to exert a positive impact [88] in the open-source software lifecycle. More recent work emphasizes early-stage repository metrics, including commit frequency, contributor count, contributor reputation, issue responsiveness, build frequency, and CI adoption as strong predictors of long-term healthy sustainability [52, 115, 146].

However, the strength of these predictors varies by project type and maturity, and very few studies have evaluated them in the context of rapidly evolving software paradigms like MCP servers. While MCP servers functionally resemble traditional reusable OSS components (PyPI or NPM packages), they operate under unique constraints, e.g., integration with foundation models and system-level tool orchestration. While sustainability and evolution have been extensively studied in PyPI [23, 135] and NPM [92], to the best of our knowledge, no prior study has empirically assessed the sustainability of MCP servers. Our work addresses this gap by adapting established metrics to this emerging domain and assessing their applicability.

4.2   Security Vulnerabilities in OSS and Package Ecosystems

On the other hand, security remains a critical concern in open-source ecosystems. Historically, we have observed that one single vulnerability in an open-source project, e.g., Heartbleed from OpenSSL, can bring the whole software industry down globally [138]. So, it is imperative to detect and resolve vulnerabilities in open-source software. To proactively detect vulnerabilities, researchers have developed a wide range of approaches, including static analysis tools [66, 68, 125], machine learning-based classifiers [120], and more recently, foundation model (FM) based tools [119, 148]. Empirical studies using these techniques have revealed persistent security risks across major OSS ecosystems, including NPM and PyPI [6, 45, 154]. Although many of these vulnerabilities are of moderate severity [112], their impact can be significant when embedded in popular transitive dependency chains [117].

---

[2]https://glama.ai/mcp/servers/@stripe/agent-toolkit
[3]https://glama.ai/mcp/servers/@atharvagupta2003/mcp-stripe

As an emerging domain, we also observe growing research to identify potential security loopholes in the MCP ecosystem. Common threats proposed include tool poisoning, command injection, installer spoofing, and configuration drift [62, 70]. More severe exploits demonstrating Malicious Code Execution (MCE), Remote Access Control (RAC), Credential Theft (CT), and novel Retrieval-Agent Deception (RADE) attacks have been demonstrated to bypass standard LLM guardrails [106]. However, despite these demonstrations, a wide gap exists in understanding the real-world prevalence of such vulnerabilities across the MCP ecosystem. Existing studies have primarily focused on controlled examples and proof-of-concept attacks, but no empirical investigation has yet examined how frequently these issues occur in publicly available MCP servers. Our study aims to bridge that gap by analyzing open-source MCP servers using general-purpose software engineering tools and MCP-specific vulnerability scanners.

4.3 Maintainability of OSS

For MCP servers to be broadly adopted in software ecosystems, they must be reusable and maintainable without imposing undue developer burden. Like traditional OSS, MCP servers are community-maintained and thus susceptible to the similar long-term maintainability risks [27, 118]. In OSS research, two significant maintainability dimensions have been extensively studied: (i) code smells, e.g., structural issues that hinder code readability and evolution, and (ii) bugs, e.g., functional errors that can lead to failures [34, 100, 121, 134, 150]. Together, these issues can consume up to 39% of team effort in software projects [9].

Prior work has established code smells, e.g., high cognitive complexity, as positively associated with low maintainability [44], and that low maintainability can lead to project mortality [39]. Additionally, studies have correlated higher bug severity with increased maintainability issues [34] in open-source projects. Similar to vulnerability, to detect such maintainability issues, researchers have developed a broad range of techniques, including static analysis [56, 75], machine learning-based approaches [147], and more recently, foundation model-based techniques [144].

Despite these advances, no prior study has evaluated maintainability risks in the MCP ecosystem, e.g., code smells and bugs. To address this gap, we analyze MCP servers using static analysis tools to assess the prevalence of code smells and bugs.

5　Methodology

This section presents the methodological framework of our study for evaluating the health and sustainability metrics, vulnerabilities, code smells and bugs of open-source MCP servers. We begin by constructing a comprehensive dataset of MCP server repositories. First, we collect officially listed MCP servers from Anthropic's GitHub repository. We then mine GitHub to identify additional repositories that use official MCP SDKs and enhance Anthropic's list with the mined results. To assess health and sustainability, we collect repository-level metadata (e.g., commit activity, contributor count) from GitHub for each MCP server. For security and maintainability evaluation, we perform static analysis using the state-of-the-art SonarQube [29], focusing on vulnerabilities, code smells, and bugs. Additionally, to identify MCP-specific security risks, e.g., tool poisoning, we apply the mcp-scan tool [71] to a representative sample of MCP servers.

Next, we apply an LLM-based jury technique [80] to synthesize and cluster the static analysis results into interpretable vulnerability, smell, and bug patterns and validate these patterns through manual analysis on randomly selected samples. Finally, we conduct a structured literature review to extract baseline metrics and defect (e.g., vulnerabilities, code smells, and bugs) taxonomies from related ecosystems (e.g., PyPI, NPM, ML codebases). These baselines are used to contextualize and contrast our findings on MCP servers. An overview of our study design is presented in Figure 3.
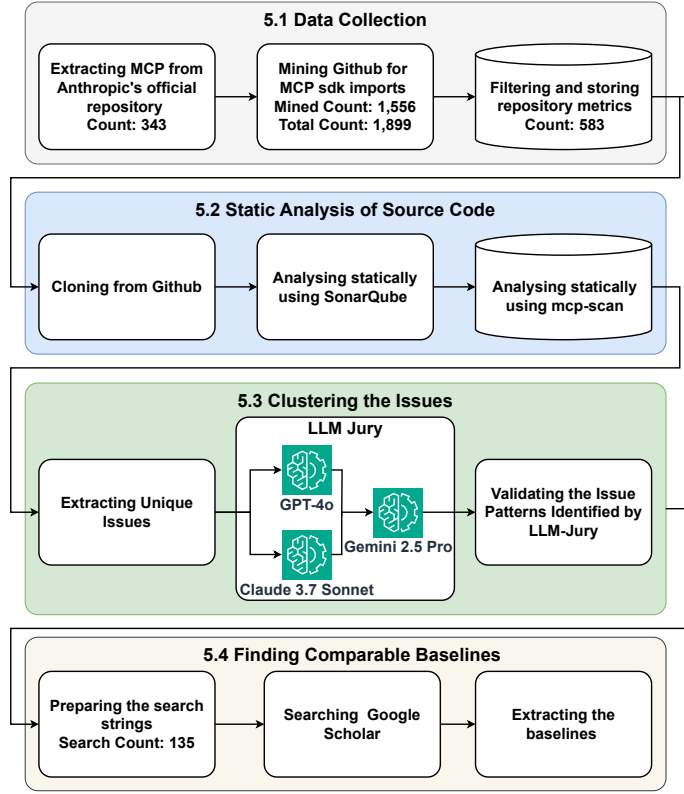
Fig. 3. Overview of the study design.

5.1 Data Collection

5.1.1 Extracting from Anthropic's official repository. Anthropic has published a list of MCP servers in their official repository and maintains the list actively. We start with this list of MCP servers maintained by Anthropic in their official Model Context Protocol repository[4]. In this repository, Anthropic classifies MCP servers into two major categories:

- **Official Integrations:** MCP servers maintained by organizations that build production-ready integrations for their platforms. For example, the AWS MCP server[5] is maintained by Amazon's AWS Labs.
- **Community Servers:** MCP servers developed and maintained by independent community members or contributors for various use cases. One such example is the MCP server built for the DeepSeek R1 model[6].

We identify 88 official and 255 community integrations listed in Anthropic's repository on Mar 19, 2025. For each identified repository, we record the repository name and GitHub URL in an Elasticsearch database.

---

[4]https://github.com/modelcontextprotocol/servers?tab=readme-ov-file
[5]https://github.com/awslabs/mcp
[6]https://github.com/66julienmartin/MCP-server-Deepseek_R1

Table 1. Distribution of integration types across programming languages

Language	Official	Community	Mined	Total Count
Python	20	73	103	196
JavaScript	13	39	63	115
TypeScript	22	52	153	227
Others	6	20	19	45
Total	61	184	338	583

5.1.2   Mining GitHub for SDK imports. We anticipate that, in addition to the repositories listed by Anthropic, many other MCP servers have been developed by independent contributors. To expand the coverage of our dataset, we mine GitHub for open-source MCP server implementations. Specifically, we search for the usage of MCP SDKs within publicly available source code. Alongside the introduction of the MCP server framework, Anthropic released SDK libraries for several popular programming languages, including Python, JavaScript, and TypeScript. These SDKs enable community members to build MCP servers using a standardized interface.

To identify relevant repositories, we use targeted search terms that correspond to known MCP SDK import patterns. For JavaScript and TypeScript, we search for the string `@modelcontextprotocol/sdk`, and for Python, we use the term `from mcp.server`. These queries are executed using the GitHub Code Search API,[7] following practices used in prior studies [78].

Using this approach, we identify a total of 1,715 repositories that import MCP server SDKs. Cross-referencing these with the repositories listed in Anthropic's official documentation, we find that 159 are already included as either official or community integrations. We label the remaining 1,556 repositories as **mined** MCP servers and store them in our Elasticsearch index. The cut-off date for MCP server mining is Mar 20, 2025.

5.1.3   Filtering the Toy Repositories. To ensure the quality of the MCP servers in our dataset, we filter out toy projects based on GitHub popularity. Following established practice in prior work [42], we exclude MCP server repositories with fewer than ten stars, resulting in a collection of 583 MCP servers. Although our mining process targets MCP servers using Python, JavaScript, or TypeScript SDKs, we observe several repositories whose dominant language fell outside these categories. This discrepancy arises because the Github API reports one dominant programming language in API from any polyglot project, e.g., projects containing code in multiple programming languages. We classify these under the label "Others". Table 1 shows the final distribution across integration types(e.g., official, community, and mined) and programming languages.

5.1.4   Mining and storing repository metrics. After mining all the repositories, we identify a consolidated set of 14 state-of-the-art metrics as operational proxies from Community Health Analytics in Open Source Software (CHAOSS) project [31] and prior literature [52, 53, 61, 130, 146] for evaluating the health and sustainability of MCP ecosystem. We then develop a script to automatically collect these metrics using the GitHub REST API for each of our dataset's 583 MCP server repositories. To ensure traceability and data reuse, we store the raw and processed metric values in an Elasticsearch-based data store, with one document per repository containing all extracted fields.

5.2   Static Analysis of Source Code

5.2.1   Analyzing with Traditional Tool. To analyze software vulnerabilities, code smells, and bugs in MCP servers, we apply static analysis using SonarQube [29], a widely used open-source tool.

---

[7]https://docs.github.com/en/rest/search/search?apiVersion=2022-11-28#search-code

Compared to FindBugs [43] and PMD [102], SonarQube offers broader language support (over 30 languages) and does not require compiled bytecode. SonarQube evaluates code using a rich set of predefined rules aligned with industry-standard security benchmarks such as MITRE CWE Top 25 [91], OWASP Top 10 [7], and PCI DSS [96]. Beyond vulnerabilities, SonarQube is also popular for identifying maintainability concerns such as code smells and bugs [151].

Given the scale of our dataset (583 repositories), we develop an automated workflow that (i) clones the target repository from GitHub; (ii) analyzes it using a Docker-based SonarQube CLI setup; and (iii) extracts issue-level metadata via the SonarQube Web API[8], and (iv) stores results in Elasticsearch for efficient retrieval. The scan time for individual repositories ranges from 30 seconds to 5 minutes, depending on project size. All analyses were completed within four days on a dedicated M3 Macbook Air machine.

5.2.2 Analyzing with MCP-specific Tool. In addition to general software vulnerabilities, MCP-specific vulnerabilities, e.g., tool poisoning, are increasingly discussed in recent literature [95]. Industry efforts have also emerged to address these concerns, with companies like Invariant Labs releasing dedicated MCP scanners, such as mcp-scan [71].

However, scanning with this tool requires a rigorous setup with steps like installing the MCP server along with the relevant infrastructure components such as Docker [48], collecting configuration-specific environment variables, e.g., API tokens and credentials, and installing dependencies. As this process is laborious and time-consuming, we decided to scan a subset of representative MCP servers. Using a 95% confidence level and 10% margin of error, we randomly sample 83 repositories from our complete set of 583 MCP servers. We then manually scan these 83 repositories using mcp-scan to identify potential instances of tool poisoning.

5.3 Clustering the Issues

5.3.1 Extracting Unique Issues and Violated Rules. We analyze the findings generated by Sonar-Qube across MCP server projects, focusing on identified security vulnerabilities, code smells, and bugs. Any violation of SonarQube's rules[9] is termed an "issue" with a type of "vulnerability" or "code_smell" or "bug", and an experience "severity" is assigned to that issue. SonarQube uses five major severity types [127]:

(1) *Blocker*: Problems that can cause severe unintended consequences, such as crashes, security breaches, and require immediate resolution.
(2) *Critical*: An issue with a critical impact on the application that should be fixed as soon as possible.
(3) *Major*: An issue with a major impact on the application.
(4) *Minor*: An issue with a minor impact on the application.
(5) *Info*: There is no expected impact on the application. For informational purposes only.

We extract the issue instances including metadata such as issue type, severity level, violated rule identifier, and rule description. We then synthesize the findings by deduplicating the rule violations and organizing them into three distinct sets corresponding to vulnerabilities, code smells, and bugs. This categorization enables further statistical and qualitative analysis in subsequent phases of our study.

5.3.2 Applying LLM Jury. To uncover broader recurring patterns across these issues, we cluster each set of unique issues into higher-level abstract patterns. For this clustering task, we adopt an LLM-Jury approach following prior works [79]. We use LLM-Jury, as this type of system has

---

[8]https://docs.sonarsource.com/sonarqube-server/10.4/extension-guide/web-api/
[9]https://rules.sonarsource.com/

Table 2. Search strategies used in our literature review to identify comparable baselines for four review items: health and sustainability metrics, security vulnerabilities, code smells, and bugs. For each item, we crafted structured search strings using key variables, and systematically explored all combinations of their values in Google Scholar.

Review Dimension	Search String	Variables
RQ-0: Health & Sustainability Metrics	<measurement_type> + <metric_name> + "empirical study"	**measurement_type**: median, mean, average, prevalence **metric_name**: RQ-0 Metrics (Table 4)
RQ-1: Vulnerabilities	"vulnerability" + "security" + "prevalence" + <domain_name>	**domain_name**: PyPI, NPM, IaC, Python, JavaScript
RQ-2: Code Smells	"code smell" + "prevalence" + <domain_name>	**domain_name**: PyPI, NPM, Python, JavaScript
RQ-2: Bugs	1. "bugs" + "prevalence" + <domain_name> 2. "static analysis" + "defects" 3. <bug_pattern> + <domain_name>	**domain_name**: Python, JavaScript, None **bug_pattern**: RQ-2 Bugs (Table 9)

already shown significantly close human-model inter-rater agreement in contrast to human-human agreement in different software engineering tasks, e.g., labeling or annotations, and reduces human effort [1, 67]. We follow the pairwise judge cognitive architecture [80], where each of the worker LLMs assigns a cluster label for each identified issue independently based on the violated rule name, description, impact, and language. Finally, the judge LLM reviews the same artifacts along with the outputs from each worker and determines the most appropriate thematic pattern for each issue. We use `claude-3.7-sonnet` and `gpt-4o` as worker LLMs, and `gemini-2.5-pro` as the judge LLM. Through this LLM-Jury process, we generate high-level thematic patterns that characterize the most common types of vulnerabilities, code smells, and bugs in the MCP server ecosystem.

5.3.3 Validating the Issue Patterns Identified by LLM-Jury. While prior work [80] reports a sufficient inter-rater agreement between LLM-Jury and human annotators, we independently validate its clustering results on our dataset. We randomly sample 25 issues from each category (vulnerabilities, code smells, and bugs), resulting in 75 samples. Each of the first two authors of this study works as a human reviewer and independently labels each issue. Then, two human reviewers resolve the disagreements, e.g., around naming and granularity, through discussions. Finally, to quantify agreement, we compute Fleiss' Kappa [51] across all three raters: the two human reviewers and the LLM-Jury system. We observe that all three raters have a perfect agreement for vulnerabilities ($\kappa = 1.0$) and near-perfect agreement for code smells and bugs ($\kappa = 0.9$), indicating high confidence in the quality of the issue patterns derived by the LLM-Jury system.

5.4 Finding Comparable Baselines

To contextualize our findings on MCP server health, security, and maintainability, we systematically review prior literature in order to extract comparable baselines across our study dimensions: health and sustainability metrics, vulnerabilities, maintainability issues, e.g., code smells, and software bugs. Given the novelty of the MCP domain and the breadth of our measurement targets, locating baselines for all dimensions is non-trivial.

We construct structured search strings and perform 135 literature searches on Google Scholar, as summarized in Table 2.

For each search, we varied key variables (e.g., metric names, bug patterns, domains), resulting in 96 searches for metrics, 5 for vulnerabilities, 4 for code smells, and 30 for bugs. We systematically examine the top 50 results per search and include studies based on the following criteria: (i) peer-reviewed venue, (ii) empirical focus on open-source ecosystems, and (iii) domain relevance (e.g., Python, JavaScript, ML, IaC). After detailed analysis and examination based on these filters, we find a total of 40 prior studies to contrast the baselines we need for our three research questions. We summarize the selected studies in Table 3.

From the studies related to RQ-0 in Table 3, we systematically extract health and sustainability metrics values (e.g., median commit frequency). For RQ-1, we identify vulnerability taxonomy and prevalence in prominent ecosystems, e.g., PyPI, NPM, and IaC, from the associated studies. Similarly, from the studies associated with RQ-2, we first extract the taxonomies and quantitative prevalence of code smells and bugs in traditional software domains; these studies also allow us to ascertain the prevalence of specific bug types (later identified in MCP servers) within these other domains. The collective data on metrics, taxonomies, and prevalence from these diverse sources form the baseline for our comparative analysis. This review process consumed approximately 60 person-hours, performed by the first two authors, and ensured that our comparisons to traditional software ecosystems are grounded in representative and methodologically sound sources.

6 Results

6.1 RQ-0: How healthy and sustainable are MCP servers?

Motivation. Open-source software (OSS) projects often face sustainability challenges, with studies reporting that over 20% become inactive within the first year and nearly 50% by the fourth year [4]. This attrition poses risks to downstream applications that depend on these projects. Prior studies have found that some early-stage metrics can work as indicators of health and sustainability: development metrics (e.g., commit frequency, adoption of continuous integration) reflect ongoing maintenance, while community metrics (e.g., number of contributors, contributor reputation, and issue activity) signal developer engagement [39, 53, 130, 146].

As MCP adoption grows, understanding the ecosystem's sustainability becomes increasingly critical. However, no prior work has investigated how these established metrics behave in MCP servers. In this study, we use these metrics to study MCP servers, assessing their health and sustainability from development activity and community involvement points of view. To interpret our findings, we contrast the development and community metrics of MCP servers with applications from general OSS and machine learning (ML) domains and provide a comparative baseline that clarifies whether MCP development trajectories align with healthy and sustainable software ecosystems.

Approach. We identify 14 repository-level metrics from existing literature that can be used to assess the health and sustainability of MCP servers, and mine those using the GitHub REST API as described in Section 5.1.4. Automated scripts fetch metrics related to commit frequency, contributors, stars, forks, project size, issue lifecycle, and builds. From existing literature, we gather similar metrics for general OSS and ML applications. In case when we did not find some metrics' value for the ML domain in the existing literature, we still interpret the findings relative to general OSS metrics. Additionally, we compare the metrics across the three MCP integration types, i.e., official, community and mined, to understand whether some metrics are better for some types of MCP servers. To analyze this, we apply the Kruskal-Wallis H Test [84] followed by pairwise Mann-Whitney U tests with Bonferroni correction [114]. We also measure the effect size through Cliff's delta [83].

Findings. MCP servers demonstrate higher or equal median values in 9 out of 14 key development and community metrics in comparison to OSS and ML baselines, indicating

Table 3. Summary of the 40 selected studies reviewed to identify comparable baselines for our three research questions, ordered by publication year.

RQ	Study	Purpose
RQ-0	Herraiz et al., 2009 [60]	An empirical study on OSS analyzing their evolution
	Kerzazi et al., 2014 [69]	A study to measure the impact of build breakage
	Borges et al., 2016 [25]	A study on the popularity of software systems hosted at GitHub
	Hilton et al., 2016 [61]	Understanding the usage of CI systems
	Coelho et al., 2017 [39]	Reasons Behind the Failure of Modern Open Source Projects
	Baltes et al., 2018 [19]	A study on the Influence of CI on Commit Activity
	Zou et al., 2019 [155]	An empirical study on branch usage in GitHub projects
	Bao et al., 2019 [20]	Predicting newcomers' transition to long-term contributors
	Chen et al., 2020 [33]	A study on characterizing real-world build times
	Goggins et al., 2021 [53]	Exploring the metrics related to health and sustainability
	Moid et al., 2021 [90]	A study to predict repository stars using smart models
	Ait et al., 2022 [4]	Assessing survival rate of GitHub projects
	Xiao et al., 2023 [146]	Exploring the long-term project sustainability on GitHub
	He et al., 2023 [59]	A study to evaluate the effectiveness of Dependabot
	Idowu et al., 2024 [64]	A study on OSS ML projects, focusing on evolution
	Lai et al., 2024 [72]	Comparison between ML and non-ML issues in OSS AI projects
	Bernardo et al., 2024 [22]	Exploring CI adoption practices in ML projects
RQ-1	Rahman et al., 2019 [107]	An empirical study of security smells in IaC scripts
	Wist et al., 2021 [143]	An empirical study on vulnerabilities in Docker Hub images
	Ruihonen et al., 2021 [113]	A security-oriented static analysis of Python packages in PyPI
	Latendresse et al., 2022 [74]	Analyzing security risks of JavaScript dependencies in NPM
	Zerouali et al., 2022 [153]	A study on vulnerabilities affecting NPM and RubyGems packages
	Alfadel et al., 2023 [6]	An analysis of security vulnerabilities in Python packages
RQ-2	Ayewah et al., 2007 [18]	A study of warnings found by FindBugs in Java programs
	Yamashita et al., 2012 [150]	How well code smells reflect factors affecting maintainability
	Park et al., 2015 [101]	An analysis of HTML and CSS syntax errors
	Tufano et al., 2015 [134]	Understanding when and why bad smells are introduced
	Saboury et al., 2017 [116]	An empirical study of code smells in JavaScript projects
	Rice et al., 2017 [110]	An algorithm to detect method argument selection bugs
	Castagna et al., 2017 [30]	A type system for functional languages to support gradual typing
	Chen et al., 2018 [34]	An empirical study on how defects impact maintainability
	Palomba et al., 2018 [100]	Relationship between code smells and fault/change proneness
	Wang et al., 2019 [140]	An approach to automatically repair buggy loops
	Munoz et al., 2020 [94]	Validating cognitive complexity's impact on code understandability
	Amit et al., 2021 [9]	Measuring the effort invested in bug fixing
	Van Oort et al., 2021 [136]	Studying the prevalence of code smells in ML projects
	Siddiq et al., 2022 [121]	A study of code smells in transformer-based code generation
	Gupta et al., 2023 [55]	A severity assessment of Python code smells
	Arteca et al., 2023 [15]	A study on detecting incorrect property accesses in JavaScript
	Souza et al., 2024 [129]	Detecting exception-handling anti-patterns in Java, TS, and Python

promising sustainability. For instance, as shown in Table 4, MCP servers have a **higher median commit frequency** (5.5 commits/week) compared to general OSS projects (2.5 commits/week). In terms of CI adoption (e.g., using tools to automate the compilation, building, and testing of software), we observe **42.2% MCP servers have adopted CI**, which is slightly higher than both the

Table 4. Comparison of development and community metrics of MCP servers with General OSS and ML domains. The bold ones are the age-normalized values of time-dependent metrics, e.g., which grow over time.

Metric Name	MCP Server	General OSS Domain	ML Domain
Median Total Commit Count	36.3	608.0 [25]	110.0 [64]
Median Commits/Week	5.5	2.5 [19]	-
Median Github Contributor Count	2.0	41.0 [25]	2.0 [64]
Norm. Median Github Contributor Count/year	4.0	61.2 [25]	-
Median Follower Count Of Contributors	129.6	37.3 [90]	-
Norm. Med. Follower Count Of Contributors/year	259.2	17.0 [90]	-
Median Star Count	39.3	66.0 [59]	-
Norm. Median Star Count/year	79.0	34.7 [59]	-
Median Forks Count	9.0	51.0 [155]	-
Norm. Median Forks Count/year	18.0	7.5 [155]	-
Median Lines Of Code	925.2	21,168.0 [60]	2,849.0 [136]
Median File Count	9.0	142.0 [60]	26.0 [64]
Median Total Github Issue Count	2.0	673.0 [20]	-
Median Issue Lifetime in Days	5.6	4.0 [32]	25.0 [72]
CI Adoption Rate (%)	42.2	40.3 [61]	37.2 [115]
Build Success Rate(%)	90.0	70.0 [33]	-
Median Build Duration in Mins	1.9	9.3 [33]	21.4 [22]
Median Time To Fix a Broken Build in Mins	13.9	46.0 [69]	-

General OSS domain (40.3%) and the ML domain (37.2%). While the difference compared to general OSS is not substantial, this adoption rate is notable because prior work reports that open-source projects typically adopt CI only after one year [61], whereas our findings indicate that MCP servers often adopt CI within six months of their initial release.

MCP servers also exhibit a higher median build success rate, shorter median build times, and faster resolution of broken builds compared to the baselines as shown in Table 4. According to prior research, better build-related metrics indicate that **MCP servers are capable of doing more frequent releases** [61], and release frequency can positively impact the development and sustainability of projects in their early stage [52].

MCP servers exhibit higher age-normalized growth in some metrics, e.g., stars and forks, despite appearing to lag behind OSS baselines in raw counts. As shown in Table 4, the median star count and fork count of MCP Servers are lower than the baselines. However, the MCP protocol was introduced only six months ago, whereas the baseline projects are much older, e.g., the median age of the projects for fork count and star count are 6.8 and 1.9 years, respectively. Hence, normalizing these metrics by project age, MCP servers demonstrate an exceptionally fast growth trajectory. Specifically, MCP servers average approximately 79 stars and 18 forks per year, surpassing the normalized rates of 34.7 stars and 7.5 per year observed in the OSS baselines. Additionally, we observe higher community reach of MCP contributors in both raw and age-normalized count of their followers. These accelerated early-stage trends suggest a promising early trajectory for sustainability within the MCP ecosystem.

We find that mined MCP servers receive 101.4% more commits than community MCP servers. The median total commit count in mined, official, and community MCP servers is 44.3, 42.0, and 22.0, respectively. We use a Kruskal–Wallis H-test to confirm that the differences in total

commit count among the three integration types are statistically significant ($H = 22$, $p = 0.000$), and a post-hoc Mann-Whitney U test with Cliff's Delta reveals that the difference is only significant for community vs mined ($Pcorr = 0.000$ and $d = -0.243$, *small effect*). The difference between the other two combinations is not significant indicating that mined MCP servers have more development activities than only community servers.

We also find that mined MCP servers are 56% larger than the official MCP server in terms of LoC. We observe median LoC in mined, official, and community MCP servers are 1,445.5, 929, and 548, respectively. A Kruskal–Wallis H-test confirms that the differences in lines of code across the three integration types are statistically significant ($H = 44.4976$, $p < 0.0001$). Post-hoc Mann–Whitney U tests with Cliff's Delta reveal that mined MCP servers contain more lines of code than official MCP servers ($p_{\text{corr}} = 0.008$, $d = -0.244$, *small effect*) and community MCP servers ($p_{\text{corr}} = 0.000$, $d = -0.345$, *medium effect*). There is no significant difference between official and community MCP servers. Similar to the previous finding, a larger project size in mined MCP servers again demonstrates more development activities.

> **Summary of RQ-0**
>
> (1) MCP servers demonstrate healthy development behaviors in terms of early-stage health and sustainability indicators.
> (2) Mined MCP servers are more active and larger in size, suggesting early adopter momentum.

6.2 RQ-1: To what extent do MCP servers contain security vulnerabilities?

Motivation. Vulnerabilities are widespread in open-source ecosystems. For example, 46% of Python packages and 40% of JavaScript packages contain at least one known security issue [74, 113]. We observe widespread adoption of these languages to build MCP servers. e.g., millions of weekly downloads of the MCP packages [11, 12], raising immediate concerns about their security posture. Moreover, the vulnerability landscape is evolving with the rise of FM-based AI tools. For instance, a recent attack targeting the FM-based code editor and MCP client "Cursor" [10] leveraged three malicious NPM packages to exfiltrate credentials from over 4,200 users.[11]

This example highlights the broader risks of MCP servers as they mediate access between FMs and external systems, a dimension that has not existed before. In particular, MCP servers, deployed locally or remotely, act as intermediaries connecting FMs with sensitive resources, e.g., file systems, databases, and API endpoints. As a result, MCP servers often handle confidential data, including credentials, API keys, and user information. This tight coupling with critical infrastructure makes MCP servers attractive targets for exploitation. Despite this, the extent to which MCP servers are vulnerable remains unknown.

Motivated by these emerging threat landscapes, we investigate the extent and nature of vulnerabilities present in MCP servers. Specifically, this research question aims to characterize the prevalence and patterns of security vulnerabilities in MCP servers, comparing those with the reported vulnerabilities from other domains in previous literature and assessing whether current tools and techniques are sufficient to detect the unique vulnerability landscape of MCP servers.

Approach. To extract vulnerability issues from MCP servers, we perform static analysis on their codebases using SonarQube, as detailed in Section 5.2.1. Out of five major severity categories of SonarQube, in this RQ, we focus on the first four severity levels: *Blocker*, *Critical*, *Major*, and *Minor*.

---

[10]https://www.cursor.com/en
[11]https://thehackernews.com/2025/05/malicious-npm-packages-infect-3200.html

We extract all vulnerabilities within these categories across the analyzed repositories and use the LLM-Jury methodology to derive high-level vulnerability patterns.

To ground the identified vulnerability patterns, we map each to its closest corresponding Common Weakness Enumeration (CWE) [89], a widely adopted taxonomy maintained by the security community. The mapping process involves examining the definition of the violated SonarQube rule, referencing the official SonarQube rule documentation [128], and selecting the CWE that best describes the underlying weakness. For instance, when plaintext credentials such as OPENAI_API_KEY are detected in the source code, we map this pattern to CWE-798: *Use of Hard-coded Credentials*, as described in SonarQube rule S7219[12].

To assess the real-world impact of these weaknesses, we further check the Common Vulnerabilities and Exposures (CVE) database [37] to identify previously reported security incidents associated with each mapped CWE. This helps contextualize the potential risk of MCP vulnerabilities by linking them to known exploits and publicly disclosed attacks.

To capture MCP-specific vulnerabilities we additionally analyze a representative subset of MCP repositories using the mcp-scan tool [71] as described in Section 5.2.2. Finally, since our dataset includes three integration types—*official*, *community*, and *mined*, we apply the Kruskal-Wallis H test followed by pairwise Mann-Whitney U tests with Bonferroni correction to statistically compare the distributions of vulnerability counts across these groups, following the methodology used in prior software engineering studies [49].

Findings. MCP servers exhibit "credential exposure" as the most common vulnerability as opposed to "cross-site-scripting" in PyPI or "malicious package" of NPM ecosystem. We report the eight vulnerability patterns identified using SonarQube, the traditional vulnerability detector, in MCP servers in Table 5 along with the top vulnerability patterns available for PyPI, NPM, and the IaC ecosystem. Among these patterns, only two—authentication issues and improper input validation–commonly appear as top vulnerabilities in PyPI packages [6], while none are prominently reported in NPM packages [153]. Credential exposure, which we identify as the most frequent vulnerability in MCP servers, maps to the hard-coded secrets vulnerability pattern of Infrastructure-as-Code domain [107]. We also observe a lack of access control, improper resource management, and transport security issues as other prevalent vulnerabilities in MCP servers, yet no prior work has found these as top vulnerabilities in other ecosystems. These results demonstrate that MCP servers face a broader and more distinctive set of vulnerabilities, underscoring the need for ecosystem-specific analysis and tooling.

7.2% of MCP server repositories contain at least one security vulnerability, with half of these affected by credential exposure. We summarize the distribution of vulnerability patterns in Table 6. We detect 277 vulnerabilities across 42 unique MCP servers, which are related closely to 13 CWEs. Many of those CWEs are the root cause of previously reported CVEs in other domains, as reported in Table 6, indicating that the vulnerabilities detected in the MCP server are non-trivial.

Traditional tools (e.g., SonarQube) detect substantially fewer vulnerabilities in MCP servers than in ecosystems such as PyPI and NPM. For instance, prior studies report that 46% of Python packages contain at least one security vulnerability [113], and 40% of NPM packages depend on code with known vulnerabilities [74]. Furthermore, researchers have identified 119 distinct Common Weakness Enumerations (CWEs) across the PyPI ecosystem [6] which is significantly higher than the CWEs found in MCP servers.

MCP-specific analysis, particularly with emerging tools like mcp-scan, presents significant engineering challenges. As detailed in our methodology (Section 5.2.2), running mcp-scan necessitated a complex setup, requiring not only the installation of the MCP servers but also its

---

[12]https://next.sonarqube.com/sonarqube/coding_rules?open=secrets:S7219&rule_key=secrets:S7219

Table 5. Vulnerability patterns across MCP servers, PyPI packages, NPM packages, and IaC scripts, sorted in descending order of prevalence. Highlighted patterns indicate cross-domain similarities, with superscript numbers and color denoting the closest semantic match. For example: (1) Credential Exposure in MCP closely resembles Hard-coded Secrets in IaC; (2) Authentication Issues in MCP align with Access Restriction Bypass in PyPI; (3) Input Validation Issues in MCP are similar to Improper Input Validation in PyPI.

MCP Servers	PyPI Packages [6]	NPM Packages [153]	IaC Script [107]
Credential Exposure[1]	Cross-Site-Scripting (XSS)	Malicious Package	Admin by default
Lack of Access Control	Denial of Service (DoS)	Directory Traversal	Empty password
CORS Issues	Information Exposure	Cross-site Scripting	Hard-coded secret[1]
Improper Resource Management	Arbitrary Code Execution	Resource Downloaded over Insecure Protocol	Invalid IP address binding
Transport Security Issues	Access Restriction Bypass	Regular Expression Denial of Service	Suspicious comment
Authentication Issues[2]	Regular Expression Denial of Service (ReDoS)	Denial of Service	Use of HTTP without TLS
Insecure File Creation	Improper Input Validation[3]	Prototype Pollution	Use of weak crypto. algo.
Input Validation Issues[3]	Directory Traversal	Command Injection	
	Remote Code Execution (RCE)	Arbitrary Code Execution	
	Authentication Bypass[2]	Arbitrary Code Injection	

Table 6. Prevalence of vulnerability patterns in MCP servers, along with their closest related CWEs and example CVEs caused by those CWEs.

MCP Vulnerabilities	% of MCP Servers	Related CWEs	Example CVEs Caused by CWEs
Credential Exposure	3.6%	CWE-259, CWE-798	CVE-2022-29964
Lack of Access Control	1.4%	CWE-306, CWE-284	CVE-2022-24985
CORS Issues	1.2%	CWE-345	–
Improper Resource Management	1.0%	CWE-770	CVE-2022-23471
Transport Security Issues	0.7%	CWE-295, CWE-297, CWE-327	CVE-2021-22909
Authentication Issues	0.5%	CWE-347	CVE-2002-1796
Insecure File Creation	0.2%	CWE-377	CVE-2022-41954
Input Validation Issues	0.2%	CWE-611	CVE-2022-42745

complete runtime configuration, including API credentials and auxiliary services. Despite following these steps, in our initial attempt to scan a representative sample of 83 MCP servers, only 60 scans were successful, with the remainder failing due to an issue within the tool. After we reported this to the maintainers, they released a fixed version that enabled us to successfully scan an additional 13 servers, bringing our total to 73. These events highlight that such MCP-specific tools are still evolving in their early lifecycle.

Despite the operational challenges and early stage of the tool, we still detect potential tool poisoning in 5.5% of MCP servers, which is more prevalent than credential exposure. The ability of an early-stage tool, deployed with considerable effort on a limited sample, to uncover

this rate of a critical MCP-specific vulnerability strongly underscores the likelihood of more hidden issues that existing tools are currently unable to detect.

While mcp-scan is able to detect tool poisoning, it misses other security concerns, such as excessive permission requirements and insecure default behaviors. During the setup process, we manually uncovered several concerning patterns that were not flagged by the scanner. For instance, the `apple-notes-mcp` server[13] requires full disk access on macOS to interact with the native Apple Notes SQLite database highlighting an overly privileged configuration that which can introduce a significant attack surface. Similarly, `godot-mcp`[14] was configured with auto-approval enabled for sensitive operations such as stopping projects or modifying project identifiers, potentially allowing unvetted commands to be executed. These issues are missed by mcp-scan because it relies on tool descriptions obtained through reflection rather than analyzing the source code, limiting its ability to catch deeper or context-specific security flaws.

Pure MCP servers are more prone to credential exposure and transport security issues than the MCP servers derived from other applications, in which 85% of the identified vulnerabilities are found in deployment files. To better understand MCP server vulnerabilities, we analyze five random MCP servers that are "pure" MCP implementations without inherited legacy code or multifunctional roles. The most common vulnerabilities in these projects are credential exposure and transport security issues. For instance, we identify transport security issues, e.g., SSL/TLS verification bypasses in `sooperset/mcp-atlassian` and `tuanle96/mcp-odoo`, while credential exposure was prevalent in `amornpan/py-mcp-mssql`, `kiliczsh/mcp-mongo-server`, and `Matmax-Worldwide/payloadcmsmcp`. Then, we analyze the projects with more than five identified vulnerabilities. We found that only five MCP servers—`SciPhi-AI/R2R`, `alibaba/higress`, `devflowinc/trieve`, `get-convex/convex-backend`, and `anaversity/learn-agentic-ai`—fit this criterion and in these servers 85% vulnerabilities are found in ".yaml" files. At the same time, all the MCP servers with more than five vulnerabilities have implemented MCP as an additional feature in addition to their current functionalities. This highlights that the vulnerabilities in pure MCP repositories and other repositories where MCP is a derived feature need to be studied differently.

The traditional vulnerability scanner SonarQube cannot detect any vulnerabilities in official MCP servers. Figure 4 illustrates the distribution of vulnerability counts per server, grouped by integration type (`official`, `community`, and `mined`) where both community and mined MCP servers have a median vulnerability count of 2, while no vulnerability is found in official MCP servers. Interestingly, this mirrors findings from the Docker ecosystem, where official images have been shown to exhibit fewer vulnerabilities compared to community-maintained ones [143].

We detect exposed OpenAI and Gemini API keys, Google Cloud service account certificates, and GitHub tokens in community and mined MCP server repositories, posing significant risks of financial loss and unauthorized access. Figure 5 presents three representative examples of such credential exposures across JSON, Python, and certificate files from real-world repositories. Leaked API keys for platforms like OpenAI and Google Cloud can be exploited by malicious actors to initiate high-volume API calls, potentially resulting in substantial financial charges for the affected account owners. Likewise, exposed GitHub tokens may allow unauthorized access to private repositories or CI/CD pipelines. As shown in Table 6, these are indicative of CWE-798 (Use of Hardcoded Credentials), which has been associated with several previous high-impact security incidents, including CVE-2022-29964.

---

[13]https://github.com/sirmews/apple-notes-mcp
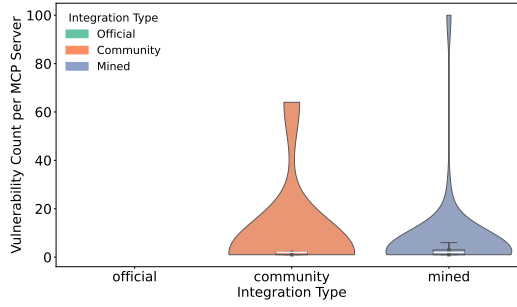[14]https://github.com/Coding-Solo/godot-mcp

Fig. 4. Vulnerability count distribution per MCP server grouped by Integration Type.

```
1  OPENAI_API_KEY": "sk-Wo********g5"
```

(a) Hardcoded OpenAI API key in a JSON configuration file.

```
1  # Configure Gemini
2  genai.configure(api_key='AIza****-****1zn4')
3  model = genai.GenerativeModel('gemini-2.0-flash-001')
```

(b) Gemini API key exposed directly in Python source code.

```
1  "type": "service_account",
2  "project_id": "***",
3  "private_key_id": "d4d****4a4",
4  "private_key": "-----BEGIN PRIVATE KEY-----\***\n-----END PRIVATE KEY
       -----\n",
5  "universe_domain": "googleapis.com"
```

(c) Google Cloud service account private key exposed in a certificate file.

Fig. 5. Examples of credential exposure across different code and configuration formats. As these are sensitive credentials and keys we have obfuscated those.

Summary of RQ-1

(1) MCP servers exhibit distinct vulnerability patterns compared to other domains of software engineering. Out of eight vulnerability patterns detected in MCP servers, three are common with other domains.
(2) MCP-specific vulnerabilities can be highly prevalent as even an early stage tool could already detect 5.5% MCP-specific issues.
(3) Credential exposure, e.g., API Keys from FM service providers, GitHub tokens, can cause significant financial loss and major data breaches.

RQ-2: To what extent do MCP servers contain maintainability issues?

Motivation. Maintainability remains a pressing concern in modern software systems, especially in the era of FM-based AI applications. According to the *State of Software-2025* report [54], 73% of AI and big data systems fall below the industry benchmark for maintainability. In mature organizations,

overheads to tackle maintainability issues can reach up to £250,000 (USD $332,500) per system annually. Two primary sources of those maintainability issues are code smells [150] and software bugs [34]. Prior research has demonstrated that code smells (e.g., high cognitive complexity) are strongly correlated with increased debugging time and elevated software error rates [94]. In addition, many code smells are often linked to bugs or fault-proneness of the systems [100], while software bugs themselves impose a considerable engineering burden—up to 39% of developer effort in leading GitHub projects is spent solely on bug fixing [9].

These concerns are particularly relevant to MCP servers, as this is a relatively new software paradigm used to connect FMs with external tools and data sources in comparison to other similar domains, e.g., PyPI or NPM packages. As prior studies suggest that most code smells are introduced during initial code creation rather than being accumulated over time [134], MCP server implementations may be particularly susceptible to maintainability issues, e.g., code smells, making early assessment critical. Motivated by these risks, this research question aims to systematically characterize the types and prevalence of code smells and software bugs in MCP servers. We also explore whether the prevalence of code smells and bugs has any statistical difference across various programming languages and integration types. By analyzing these systems at an early stage, we seek to provide actionable insights into their maintainability profile and identify patterns that may guide future development and tooling efforts.

Approach. We use the static analysis tool SonarQube to identify both code smells and bugs. We filter and retain only the critical and blocker-level code smells and bugs, and then use an LLM-Jury to categorize these issues into higher-level patterns as described in Section 5. We analyze these clusters to understand the most recurrent maintainability issues in the early-stage MCP servers and compare those with other software domains.

In addition, we conduct Kruskal-Wallis H-tests followed by pairwise Mann-Whitney U tests with Bonferroni correction to compare code smell and distributions across different programming languages (e.g., Python, JavaScript, TypeScript) and integration types (official, community, mined). We also compute Cliff's Delta (d) for each pairwise comparison to quantify the effect size.

Findings. We report our findings for code smells and bugs in the following subsections.

6.2.1 Prevalence of code smells in MCP server repositories. **All high-frequency code smells from traditional and ML systems also appear in MCP servers, indicating that existing techniques can be reused to improve MCP maintainability.** Since different analyzers define code smells at varying levels of granularity (e.g., SonarQube's *high cognitive complexity* vs. PyLint's *line-too-long, complex code*), we align SonarQube-detected smells in MCP servers with top-ranked smells in prior studies on Python ML project [136] and FM-generated code [121] based on semantic similarity and summarize this in Table 7. To contextualize our findings, we use these semantically similar categories for comparison. We find strong parallels when comparing code smells commonly studied in the JavaScript ecosystem [116]. Specifically, top JavaScript smells such as lengthy lines, chained methods, and long functions correspond closely to the *high cognitive complexity* patterns observed in MCP implementations. These overlaps suggest that refactoring techniques from existing literature, e.g., extraction, move [111, 122], de-compositional object techniques [5, 123] can also benefit the MCP domain to refactor smelly code.

66% of MCP servers contain at least one critical or blocker-level code smell, with some of those code smells, e.g., import & dependency issues, variable declaration, and usage issues, present in 100% Python ML projects. We identify a total of 17,832 critical and blocker-level code smells across 385 MCP servers and present the distribution of these issues across different programming languages in Table 8. Notably, the median number of critical code smells in MCP

Table 7. Code smell patterns across MCP Servers, ML Projects and FM Generated Code sorted by prevalence in descending order. The first column (MCP server) includes prevalence percentages. Highlighted patterns indicate cross-domain similarities, with superscript numbers and color denoting the closest semantic match.

MCP Servers	ML Projects [136]	FM-Generated Code [121]
High Cognitive Complexity (59.7%)[1]	unused-wildcard-import[5]	Undefined-variable[4]
Code Duplication-Redundancy[2] (21.4%)	bad-indentation[1]	Line-too-long[1]
Function Structure Issues (19.4%)[3]	invalid-name[4]	Unused-argument[3]
Variable Declaration and Usage Issues (11.8%)[4]	line-too-long[1]	Pointless-statement[1]
Asynchronous & Concurrency Issues (10.8%)	missing-function-docstring[3]	Pointless-string-statement[4]
Runtime Issues (8.7%)	no-member[3]	No-member[3]
JavaScript/TypeScript Specific Issues (4.1%)	duplicate-code[2]	Used-before-assignment[4]
Type Safety and Correctness (2.7%)	trailing-whitespace[1]	Superfluous-parenthesis[1]
Import & Dependency Issues (1.4%)[5]	redefined-outer-name[4]	Duplicate-code[2]
Python Specific Issues (1.2%)	missing-module-docstring[1]	Consider-using-enumerate[4]

Table 8. Distribution of critical and blocker code smells across programming languages. **Critical smell %** and **Blocker smell %** indicate the percentage of projects where at least one critical or blocker-level code smell is present. **Median Critical** and **Median Blocker** represent the median number of critical and blocker code smells per project, respectively.
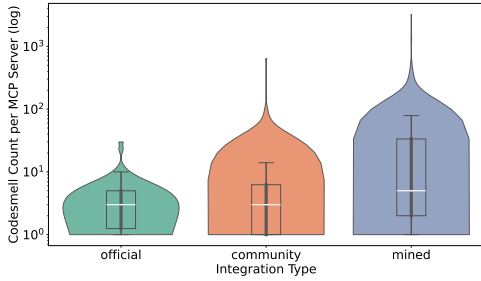
Language	Critical smell %	Median Critical	Blocker smell %	Median Blocker
Python	68.1	1.0	5.6	0
JavaScript	39.8	2.0	1.3	0
TypeScript	61.1	4.0	5.8	0
Others	47.3	12.0	4.4	0

servers ranges between 2 and 4 in the most commonly used programming languages, e.g., Python, JavaScript, and TypeScript, while the median number of blocker-level code smells is zero across all these languages. In contrast, traditional software engineering studies have reported that certain code smells can be present in nearly 100% of the studied Python ML projects and traditional Python projects [35, 136] and up-to 97% FM generated code can contain code smells [121].
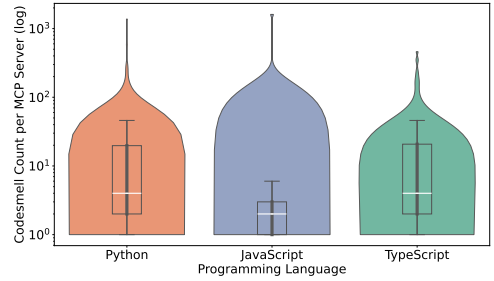
59.7% of MCP servers suffer from high cognitive complexity, which is also considered as the one of the most severe code smells in Python ecosystem. As summarized in Table 7, we observe that high cognitive complexity is almost three times more prevalent than the second most common code smell, e.g., code duplication-redundancy, in MCP servers. Cognitive complexity is a widely used metric for modeling and estimating the functional complexity, size, and effort required for software development [141]. While prior studies suggest a threshold of 15 for cognitive complexity [94], violation of this threshold is considered the one of the most severe code smells in the Python ecosystem [55]. Similarly, this threshold is violated in a substantial portion (59.7%) of MCP servers, which can lead to increased comprehension time, reduced understandability, and higher debugging time and error rates [94].

Mined MCP servers contain 66% more code smells than both official and community servers. Figure 6a presents the distribution of code smell counts per MCP server across different

integration types on a logarithmic scale. We observe a median code smell count of 5 in mined MCP servers, compared to 3 in official and community MCP servers. A Kruskal-Wallis H-test reveals a significant difference in code smell counts among the three integration types (H = 23.2936, p < 0.001). Post-hoc Mann-Whitney U tests indicate that mined servers have more code smells than both official ($P-value$ = 0.004 and d = −0.322, *small effect*) and community ($P-value$ = 0.000 and d = −0.280, *small effect*) servers. However, the difference between official and community MCP servers is not statistically significant. We can partially explain by the larger size of mined MCP servers found in Section 6.1, as prior work [149] has shown a positive correlation between code size and code smells.

(a) Code smell count distribution by integration type in logarithmic scale.

(b) Code smell count distribution by programming language in logarithmic scale.

Fig. 6. Comparison of code smells by integration type and programming language.

JavaScript MCP servers tend to have 50% fewer code smells compared to both Python and TypeScript MCP servers. Figure 6b shows the distribution of code smell counts per MCP server across different programming languages. We observe median code smell counts of 2 in JavaScript MCP servers, compared to 4 in both official and community MCP servers. A Kruskal-Wallis H-test reveals significant differences in code smell counts among the three groups(H = 26.531, p < 0.001). Post-hoc Mann-Whitney U tests show that both Python($Pcorr$ = 0.000 and d = 0.395, *medium effect*) and TypeScript($Pcorr$ = 0.000 and d = 0.415, *medium effect*) servers have significantly more code smells than JavaScript servers whereas the difference between Python and TypeScript is not statistically significant.

6.2.2 Prevalence of bugs in MCP server repositories. **Despite language and domain differences, at least 30% of MCP servers' most frequent bugs overlap with known bug patterns in Java applications, indicating that established debugging techniques can be reused with adaptation.** We report and compare the most frequent bugs in MCP servers with those reported in Java applications analyzed using FindBugs [18], as shown in Table 9. While our sample contains no Java-based MCP servers, we used this baseline due to the scarcity of large-scale static bug analyses in Python and JavaScript ecosystems. Unlike vulnerabilities (standardized via CWEs) and code smells (driven by consistent static rules), bugs are influenced by diversified factors, e.g., language and development context [108, 132]. Nonetheless, we find three overlapping bug patterns, suggesting partial applicability of conventional bug mitigation strategies in the MCP ecosystem.

We identify nine distinct bug categories affecting 14.4% of MCP servers, all of which have been studied before in Java, Python or JavaScript ecosystems. From Table 9, we observe that the top three bug patterns in MCP servers, i.e., `Array Manipulation Issues`, `Parameter Mismatches`, and `Type/Structure Issues`, do not appear in the Java ecosystem's reported bug

Table 9. Top bug types and their distribution across MCP servers and Java projects. MCP server percentages are shown in parentheses. The table is sorted in descending order of prevalence. Highlighted patterns indicate cross-domain similarities, with superscript numbers and color denoting the closest semantic match.

MCP Servers	Java Projects [18]
Array Manipulation Issues (6.2%)	Nullcheck of value previously dereferenced
Parameter Mismatch (3.1%)	Possible null pointer dereference
Type/Structure Issues (2.9%)	Unwritten field
Mishandled Exceptions (2.2%)[1]	Invocation of toString on an array[3]
Object Property and Method Design Issues (1.7%)[2]	Class defines field that masks a superclass field
Iterable and Collection Issues (1.7%)[3]	Method call passes null for unconditionally dereferenced parameter[2]
Infinite Loop (1.7%)	Possible null pointer dereference in method on exception path[1]
Style Issues (0.9%)	Method ignores return value
Getter/Setter and Object Property Access (0.7%)	Field only ever set to null
	Suspicious reference comparison

Table 10. Distribution of critical and blocker bugs across programming languages. **Critical bug %** and **Blocker bug %** indicate the percentage of projects where at least one critical or blocker-level bug is present. **Median Critical** and **Median Blocker** represent the median number of critical and blocker bugs per project, respectively.

Language	Critical bug %	Median Critical	Blocker bug %	Median Blocker
Python	3.6	0	8.1	2.0
JavaScript	2.6	2.0	1.5	0
TypeScript	13.1	3.0	1.8	0
Others	7.0	46.0	3.0	3.0

taxonomies. Of the other six bug categories, three (i.e., `Mishandled Exceptions`, `Collection Misuse`, and `Object Method Design Issues`) have been spotted before in the Java ecosystem. Those not found in the Java ecosystem (including the most prevalent three), do occur in other ecosystems, and hence are known to practitioners and the research community. For example, `Array Manipulation Issues`, which often arise when practitioners do not pass any comparator function while sorting an array, can lead to unintended behavior such as lexicographic sorting in the JavaScript community [15]. Other than that, parameter mismatch where a function is called with incorrect arguments in terms of name or number [110], type/structure issues originating from wrong type intersection [30], infinite loop [140], style issues related to invalid CSS property [101] and getter/setter and object property access issue caused by unintentional write or read behavior in getter/setter methods [15] have been identified in the Python and JavaScript literature. These overlaps suggest that existing debugging practices can address many MCP-specific bugs, at least at the detection and fix level.

Bug percentages in MCP servers do not vary across different programming languages whereas mined MCP servers contain 67% more bugs than community MCP servers. We detect a total of 523 bugs across 84 unique MCP projects. We illustrate the language-wise bug distribution in Table 10 and also the corresponding violin plot distributions in Figure 7a. Although

---

[15]https://v8.dev/blog/array-sort

(a) Bug distribution per MCP server grouped by language.

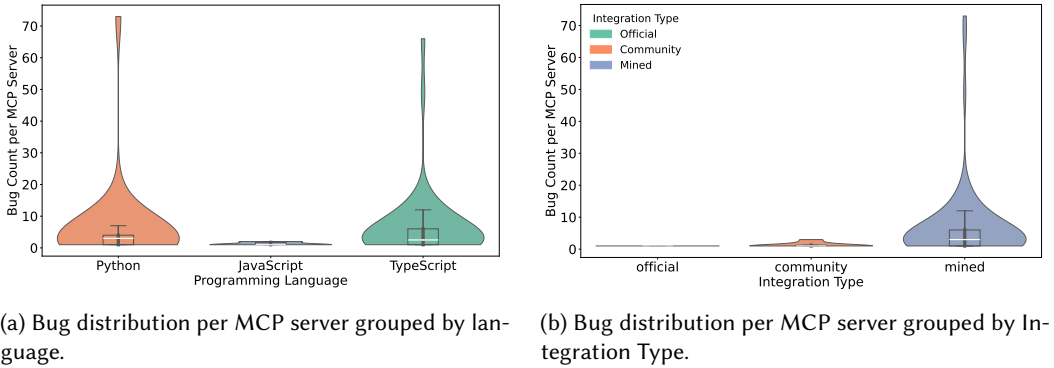(b) Bug distribution per MCP server grouped by Integration Type.

Fig. 7. Comparison of bugs by integration type and programming language across MCP servers.

from the table it appears that TypeScript can be more susceptible to critical bugs and Python is more susceptible to blocker bugs, a Kruskal-Wallis statistical test indicates that there is no significant difference in the prevalence of bugs with respect to the programming language of MCP servers ($H = 4.9$, $p = 0.178$).

In contrast, bug prevalence across integration types as shown in Figure 7b. A Kruskal-Wallis H-test reveals significant differences in bug counts among the three groups with ($H = 9.6$, $p = 0.008$). Post-hoc Mann-Whitney U tests with Cliff's delta show that mined servers have significantly more bugs than community servers ($p - value = 0.026$ and $d = -0.557$, *large effect*) where the differences in other pairs are not significant. We suspect that the higher commit activity reported in Section 6.1 can have an association with this behavior as previous studies indicated that a higher frequency of commits, especially those not adhering to contribution norms, was associated with an increased likelihood of introducing bugs [50].

Summary of RQ-2

(1) 66% of MCP servers contain at least one critical or blocker-level code smell, and 14.4% exhibit at least one statically detected bug, indicating widespread maintainability concerns.

(2) The most prevalent code smell is high cognitive complexity, affecting 59.7% of MCP servers—three times more frequent than the next most common smell—and strongly linked to reduced understandability and higher error risk.

(3) Mined MCP servers exhibit more code smells and bugs than official or community servers, possibly due to their higher development activity or less structured maintenance practices.

(4) All identified MCP bug types overlap with known bugs from Java, Python, and JavaScript ecosystems, suggesting that existing debugging and refactoring tools may be reused or adapted for MCP development.

---

7 Implications

MCP servers demonstrated a strong indication of long-term sustainability regarding different metrics we studied in Section 6.1. Hence, we must make sure the MCP ecosystem grows in a secure and maintainable manner. In this section, we discuss the implications of our study on the security and maintainability dimensions of MCP servers. Specifically, we have three target audiences in

the MCP ecosystem: (i) Researchers, (ii) Practitioners, and (iii) Ecosystem, e.g., MCP Registry, Maintainers. In the following subsections, we discuss the implications of our findings for each of these target audiences.

7.1 Implication for Researchers

Security researchers should expand standard vulnerability databases, e.g., CWE, OWASP, to include MCP-specific threats. Our study reveals that common MCP vulnerabilities, e.g., such as tool poisoning, credential exposure, and transport security issues, are largely absent from existing vulnerability taxonomies (e.g., CWE, OWASP). While we observed 13 distinct CWEs in MCP servers, only one (CWE-306) appears in the current MITRE Top 25. On the other hand, while we have OWASP top-10 for GenAI[16], we do not have anything similar for the MCP domain. We urge the security research community to formalize MCP-specific threat models and advocate for their inclusion in standard databases and taxonomies.

Researchers should develop MCP-specific vulnerability analysis tools to uncover hidden risks beyond what general-purpose scanners can detect. While traditional tools like SonarQube successfully identify some MCP vulnerabilities, e.g., credential exposure and transport security issues, our findings showed that these issues appear in only a small fraction of repositories. This low prevalence suggests an unusually secure ecosystem or, more likely, an under-detection problem rooted in the limitations of current tools. Additionally, existing MCP-specific tools like mcp-scan remain in early development stages and currently target only one class of vulnerability (tool poisoning), with limited reliability and coverage. These gaps underscore the need for new MCP-specific tools that account for the unique operational characteristics of MCP servers—including dynamic tool invocation, runtime dependency management, and FM-specific interaction patterns. Inspired by our hybrid analysis approach, future research can explore integrated frameworks that combine static and dynamic analysis, program instrumentation, and FM-aware code inspection to surface latent vulnerabilities specific to the MCP domain.

7.2 Implication for Practitioners

We can classify MCP practitioners into two groups : (i) MCP Developers who build and manage MCP servers and tools and (ii) MCP Users who use MCP servers to build FM-based AI applications. Each group can benefit by interpreting the results of this study and applying those in real-life scenarios.

MCP developers must adopt proactive security practices to prevent credential exposure and related vulnerabilities. Our findings reveal that sensitive credentials, e.g., API keys, service account secrets, and access tokens, are frequently left exposed in MCP server codebases (Section 6.2). These exposures pose immediate threats, including unauthorized system access, service abuse, and financial losses. Developers should integrate static application security testing (SAST) into their CI pipelines and adopt secure coding practices to mitigate these risks, as suggested in prior research [41]. While tools like SonarQube can detect many low-hanging issues, we recommend extending this with MCP-specific tools (e.g., mcp-scan) to catch issues unique to the domain. In addition, teams should implement secure development workflows incorporating peer-reviewed pull requests, automated secret detection (e.g., GitLeaks), and environment-based credential management to reduce exposure risks [21, 103].

MCP developers should leverage proven ML and LLM-based techniques to detect and remediate code smells and bugs. Our study finds that MCP servers exhibit code smells and bugs that closely mirror those observed in traditional software domains (Section 6.2). This similarity

---

[16]https://genai.owasp.org/llm-top-10/

allows practitioners to apply well-established techniques from the broader software engineering literature. Prior works demonstrate that FMs have potential to identify smells, generate human-like refactorings, and even provide semantic bug fixes [8, 144]. In the context of MCP development, these findings suggest multiple actionable paths: developers can integrate LLM-based tools for code smell and bug detection, adopt LLM-enhanced IDE assistants for reducing code smell while writing code, and leverage automated refactoring tools to resolve the existing code smells and bugs for long-term code quality. Given the structural and behavioral parallels between MCP servers and traditional software systems, developers are encouraged to integrate these advanced tools into their workflows.

MCP users should audit third-party MCP servers for security vulnerabilities before deployment to avoid downstream risk propagation. Our findings indicate that vulnerabilities from MCP servers, e.g., lack of access control, improper resource management, transport security issues, and insecure file creation, may propagate into downstream applications and compromise the host environment. In addition, emerging threats like tool poisoning introduce new risks whereby FMs could be coerced into executing malicious toolchains on user infrastructure, leading to persistent compromise or data exfiltration. To mitigate these risks, MCP users should incorporate proactive auditing workflows. While still evolving, tools such as SonarQube and mcp-scan offer a foundational level of detection and can surface common vulnerabilities. Although these tools may not detect the full spectrum of MCP-specific threats, our study shows that they can still identify some key issues, which makes them a good starting point. MCP users should have this bare minimum audit process until the research community develops better, state-of-the-art tools.

7.3 Implication for Ecosystem Maintainers

Ecosystem (e.g., MCP registry) maintainers should establish standardized governance mechanisms and provide clear developer guidelines to ensure baseline security and quality. MCP registries, e.g., Smithery [126], Glama [2], or Cloudflare Workers [38], serve as centralized hubs for discovering and deploying MCP servers. As our study already highlighted security vulnerabilities and maintainability issues in MCP servers, this calls for broader governance initiatives from registry maintainers. While some registries implement basic review processes (e.g., isolated manual testing or simple pull requests), these are often limited to infrastructure setup or package-level scans. For instance, Glama performs manual review in an isolated VM and scans the server for Python and NPM dependencies[17] and Smithery performs automated analysis of the MCP server code and raises simple pull requests to create Dockerfiles or any obvious missing dependencies. However, these checks fail to capture issues like credential exposure or MCP-specific tool poisoning. To mitigate these risks, registries should formalize governance protocols, including submission checklists, required permission disclosures, and security guideline conformance similar to other mature registries, e.g., Google Play Store or iOS app store [14, 65].

MCP registries should incorporate automated vulnerability and sustainability scanning and revocation workflows to prevent the propagation of insecure MCP servers. As centralized distribution platforms, registries are critical in ecosystem-wide security. However, without automated pre-publication scanning and post-deployment monitoring, they risk serving as vectors for vulnerability propagation. Inspired by mature ecosystems like the Google Play Store, Docker Hub, and the iOS App Store, registry maintainers should establish continuous monitoring and enforce automated revocation policies for non-compliant or vulnerable servers [28, 139]. These policies and workflows reduce the burden on end users and create a scalable security model that will evolve with the growing adoption of MCP infrastructure.

---

[17]https://www.reddit.com/r/mcp/comments/1hm3g2s/glama_mcp_server_directory/

8   Threats to Validity

8.1   External validity

In this study, we analyzed open-source MCP servers, acknowledging that they may not fully represent the entire population of MCP servers. To mitigate this threat to external validity, we constructed our dataset using a diverse set of sources, including Anthropic's official curated list and additional MCP servers mined from GitHub using a search-based approach.

Similarly, the vulnerabilities identified through our SonarQube analysis may not constitute an exhaustive list of all potential vulnerabilities in the MCP servers. Nevertheless, to the best of our knowledge, SonarQube's security rules cover all categories outlined in the MITRE CWE Top 25, OWASP Top 10, and PCI DSS standards [128]. Therefore, we expect that the most prevalent and critical software vulnerabilities are reflected in our analysis. Additionally, we utilized mcp-scan to detect MCP-specific vulnerabilities (e.g., tool poisoning). While mcp-scan is a relatively new and continuously evolving tool, it was the most widely recognized and actively maintained option available at the time of our study.

We report the prevalence of code smells and bugs based on SonarQube analysis in RQ-2. Although prior research has noted that SonarQube-reported "bugs" may not always correspond to actual faults, and that some "code smells" may exhibit a stronger correlation with real defects [76], our focus in RQ-2 is on the maintainability of MCP servers. Accordingly, we report both bugs and code smells as indicators of maintainability concerns.

Also, as our mining technique heavily relies on the import patterns of official MCP SDKs, repositories using MCP without explicit imports, e.g., via other third-party SDKs, may be missed from our analysis. However, as the primary mining could gather 1,899 MCP servers, the search patterns of official MCP SDKs are sufficient at the time of the study.

8.2   Construct validity

As MCP servers represent a nascent technology space, many open-sourced projects may be experimental or built by early adopters. Though we applied a star-based filter to exclude the toy projects, we acknowledge previous works stating that star count may not be a good proxy for repository quality or relevance [93]. However, as the MCP concept itself is less than a year old, we could not go for a more rigorous filtering and chose a 10-star threshold as a reasonable compromise between the data quality and the quantity, as suggested in other previous works [42].

The metrics used in this study to assess future sustainability, as discussed in Section 6.1, may not represent an exhaustive set of possible indicators. Additionally, for some metrics, median values were not available in other domains. Nevertheless, we made extensive efforts to gather metric definitions from existing literature [39, 146] and reviewed all possible empirical studies to obtain corresponding values. It is important to note that our goal in presenting these metrics across different ecosystems is not to offer an absolute comparison, but rather to provide an indication of whether an ecosystem appears to be in a generally healthy state.

The vulnerability patterns clustered by LLM-Jury could be a source of threat, as LLMs are often susceptible to hallucination. However, we mitigated this threat by grounding the patterns with the support of CWEs, as CWEs are often used to ground the vulnerability detected through open coding [107] and manual analysis [137]. We also added a manual human verification for random 25 samples not only for vulnerability but also for code smells and bugs.

8.3 Internal validity

In this study, we investigated open-source MCP servers' health, sustainability, security vulnerabilities, and maintainability issues. A key methodological challenge lies in establishing appropriate baselines for comparison, especially given the young age of the MCP ecosystem.

For RQ-0, where we studied development and community health, we derived baselines from a broad set of prior works focused on general open-source projects. These metrics are widely referenced in prior literature [39, 146], but no single prior study provided a unified baseline covering all metrics considered (e.g., commit frequency, CI adoption, contributor count). Hence, we synthesized baselines by aggregating metric distributions reported across diverse studies with varying sampling criteria, project maturity, and domains. For example, metrics for general OSS projects often reflect mature repositories with multi-year histories, while MCP servers are, on average, less than six months old. To mitigate this, we normalize time-dependent measures, perform statistical tests among the integration types and highlight relative positioning rather than making absolute judgments.

In RQ-1, we compared MCP vulnerabilities with those reported in PyPI, NPM, and infrastructure-as-code (IaC) ecosystems. However, the methodologies of these baseline studies differ from ours. For example, PyPI and NPM vulnerability counts are drawn from third-party vulnerability databases (e.g., Snyk), which rely on community reporting. In contrast, we employ static analysis via SonarQube and MCP-specific tooling. The prior IaC study [107] partially aligns methodologically through static analysis. To mitigate this, we also reported the static analysis focusing vulnerabilities [74, 113], which unfortunately reports only prevalence and does not prioritize taxonomy. As the motivation of this study is to understand the current state-of-the-art landscape of MCP servers without a comparable baseline, it is hard to relate the findings that justify the baseline choices.

In RQ-2, the bug taxonomy we use for comparison is drawn from FindBugs-based studies on Java applications [18]. Though none of the MCP servers analyzed are written in Java, we could not find any comprehensive, static-analysis-driven bug taxonomies for Python, JavaScript, or TypeScript—the dominant languages in the MCP ecosystem. Hence, using Java-based taxonomies as reference points represents an unavoidable compromise due to the limited availability of domain-specific baselines. To mitigate this, we also studied language-specific research on individual bug patterns observed in MCP servers and reported those findings separately to address this.

Finally, MCP is a rapidly evolving paradigm, and our baseline choices reflect the best available options at the time of study. As the ecosystem matures and more empirical research becomes available, we expect the relevance and precision of comparisons to improve.

9 Conclusion

Our paper presents the first large-scale empirical study of 1,899 MCP servers, evaluating their sustainability, security, and maintainability using a hybrid analysis approach that combines a general-purpose static analysis tool (i.e., SonarQube) with an MCP-specific tool (i.e., mcp-scan).

Our findings show that MCP servers generally exhibit strong signals of long-term sustainability, with higher or equal median values in 9 out of 14 key development and community metrics. Maintainability issues such as code smells and bugs—affecting 66% and 14.4% of MCP servers, respectively—occur at rates comparable to traditional open-source systems. Our findings suggest that established techniques for assessing sustainability and improving maintainability remain applicable in the MCP context.

In contrast, MCP servers diverge notably in their security profile. We identify eight distinct vulnerability patterns in 7.2% of MCP servers, only three of which overlap with common issues in

ecosystems like Python or Infrastructure-as-Code. Additionally, 5.5% of MCP servers exhibit tool poisoning—an emerging, MCP-specific threat.

These findings signal a shift in the risk surface of modern FM-based AI application infrastructure and highlight the need for dedicated tools and taxonomies to identify and address MCP-specific vulnerabilities. As MCP adoption accelerates, researchers, practitioners, and registry maintainers must invest in domain-specific security tooling, automated auditing, and robust governance to ensure the safe and reliable evolution of FM-based software systems.

Disclaimer

ChatGPT-4o and Gemini-2.5 Pro-Preview were used only for copy-editing and table formatting in compliance with IEEE and ACM policies on AI use in publications.

References

[1] Toufique Ahmed, Premkumar Devanbu, Christoph Treude, and Michael Pradel. 2025. Can LLMs Replace Manual Annotation of Software Engineering Artifacts?. In *IEEE/ACM International Conference on Mining Software Repositories*.

[2] Glama AI. 2025. Glama: Your #1 Platform for Discovering Every MCP Server. https://glama.ai/mcp, last visited: May 15.

[3] Pydantic AI. 2025. Pydantic-AI: Agent Framework / shim to use Pydantic with LLMs. https://ai.pydantic.dev/, last visited: May 22.

[4] Adem Ait, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2022. An empirical study on the survival rate of GitHub projects. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 365–375.

[5] Jehad Al Dallal and Anas Abdin. 2017. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering* 44, 1 (2017), 44–69.

[6] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2023. Empirical analysis of security vulnerabilities in python packages. *Empirical Software Engineering* 28, 3 (2023), 59.

[7] Malak Aljabri, Maryam Aldossary, Noor Al-Homeed, Bushra Alhetelah, Malek Althubiany, Ohoud Alotaibi, and Sara Alsaqer. 2022. Testing and exploiting tools to improve owasp top ten security vulnerabilities detection. In *2022 14th International Conference on Computational Intelligence and Communication Networks (CICN)*. IEEE, 797–803.

[8] Eman Abdullah AlOmar, Anushkrishna Venkatakrishnan, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. 2024. How to refactor this code? An exploratory study on developer-ChatGPT refactoring conversations. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 202–206.

[9] Idan Amit and Dror G Feitelson. 2021. Corrective commit probability: a measure of the effort invested in bug fixing. *Software Quality Journal* 29, 4 (2021), 817–861.

[10] Anthropic. 2025. Introducing the Model Context Protocol. https://www.anthropic.com/news/model-context-protocol, last visited: Apr 23.

[11] Anthropic. 2025. Model Context Protocol: NPM package. https://www.npmjs.com/package/%40modelcontextprotocol/sdk, last visited: May 18.

[12] Anthropic. 2025. Model Context Protocol: PyPi package. https://pypistats.org/packages/mcp, last visited: May 18.

[13] Anthropic. 2025. Tool Calling: Tool Usage with Claude. https://docs.anthropic.com/en/docs/agents-and-tools/tool-use/overview, last visited: May 15.

[14] Apple. 2025. App Review Guidelines. https://developer.apple.com/app-store/review/guidelines/, last visited: June 03.

[15] Ellen Arteca, Max Schäfer, and Frank Tip. 2023. A statistical approach for finding property-access errors. *arXiv preprint arXiv:2306.08741* (2023).

[16] Microsoft Autogen. 2025. AutoGen: A framework for building AI agents and applications. https://microsoft.github.io/autogen/stable/, last visited: Apr 23.

[17] Guilherme Avelino, Eleni Constantinou, Marco Tulio Valente, and Alexander Serebrenik. 2019. On the abandonment and survival of open source projects: An empirical investigation. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–12.

[18] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. 2007. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 1–8.

[19] Sebastian Baltes, Jascha Knack, Daniel Anastasiou, Ralf Tymann, and Stephan Diehl. 2018. (No) influence of continuous integration on the commit activity in GitHub projects. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics*. 1–7.

[20] Lingfeng Bao, Xin Xia, David Lo, and Gail C Murphy. 2019. A large scale study of long-time contributor prediction for github projects. *IEEE Transactions on Software Engineering* 47, 6 (2019), 1277–1298.

[21] Setu Kumar Basak, Lorenzo Neil, Bradley Reaves, and Laurie Williams. 2022. What are the practices for secret management in software artifacts?. In *2022 IEEE Secure Development Conference (SecDev)*. IEEE, 69–76.

[22] João Helis Bernardo, Daniel Alencar Da Costa, Sérgio Queiroz de Medeiros, and Uirá Kulesza. 2024. How do machine learning projects use continuous integration practices? an empirical study on GitHub actions. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 665–676.

[23] Ethan Bommarito and Michael Bommarito. 2019. An empirical analysis of the python package index (pypi). *arXiv preprint arXiv:1907.11073* (2019).

[24] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. 2021. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258* (2021).

[25] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the factors that impact the popularity of GitHub repositories. In *2016 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 334–344.

[26] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[27] Simon Butler, Jonas Gamalielsson, Björn Lundell, Christoffer Brax, Anders Mattsson, Tomas Gustavsson, Jonas Feist, Bengt Kvarnström, and Erik Lönroth. 2022. Considerations and challenges for the adoption of open source components in software-intensive businesses. *Journal of Systems and Software* 186 (2022), 111152.

[28] Paolo Calciati and Alessandra Gorla. 2017. How do apps evolve in their permission requests? a preliminary study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 37–41.

[29] G Ann Campbell and Patroklos P Papapetrou. 2013. *SonarQube in action*. Manning Publications Co.

[30] Giuseppe Castagna and Victor Lanvin. 2017. Gradual typing with union and intersection types. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–28.

[31] CHAOSS Project. 2025. Community Health Analytics in Open Source Software: Topic - All Metrics. https://chaoss.community/kbtopic/all-metrics/. Accessed: Jun 10, 2025.

[32] CHAOSS Project. 2025. Practitioner Guide: Responsiveness. https://chaoss.community/practitioner-guide-responsiveness/. Accessed: May 15, 2025.

[33] Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. 2020. Buildfast: History-aware build outcome prediction for fast feedback and reduced cost in continuous integration. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 42–53.

[34] Celia Chen, Shi Lin, Michael Shoga, Qing Wang, and Barry Boehm. 2018. How do defects hurt qualities? an empirical study on characterizing a software maintainability ontology in open source software. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 226–237.

[35] Zhifei Chen, Lin Chen, Wanwangying Ma, and Baowen Xu. 2016. Detecting code smells in python programs. In *2016 international conference on Software Analysis, Testing and Evolution (SATE)*. IEEE, 18–23.

[36] Henry Chesbrough. 2023. Measuring the economic value of open source. *San Francisco: Linux Foundation* (2023).

[37] Steve Christey and Robert A Martin. 2007. Vulnerability type distributions in CVE. *Mitre report, May* (2007).

[38] Cloudflare. 2025. Cloudflare Agents Docs: Model Context Protocol (MCP). https://developers.cloudflare.com/agents/model-context-protocol, last visited: Apr 23.

[39] Jailton Coelho and Marco Tulio Valente. 2017. Why modern open source projects fail. In *Proceedings of the 2017 11th Joint meeting on foundations of software engineering*. 186–196.

[40] CrewAI. 2025. CrewAI: The leading multi-agent platform. https://www.crewai.com/, last visited: May 27.

[41] Dinis Barroqueiro Cruz, João Rafael Almeida, and José Luís Oliveira. 2023. Open source solutions for vulnerability assessment: A comparative analysis. *IEEE Access* 11 (2023), 100234–100255.

[42] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling projects in github for MSR studies. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 560–564.

[43] Andrey Loskutov Keith Lea David Hovemeyer, Bill Pugh. 2025. An extensible multilanguage static code analyzer. https://findbugs.sourceforge.net/, last visited: May 18.

[44] DI De Silva, RD New Kandy, BLO Sachethana, SMDTH Dias, PYC Perera, ME Katipearachchi, and TDDH Jayasuriya. 2023. The Relationship between Code Complexity and Software Quality: An Empirical Study. *Journal of Software Engineering Research and Development* 11, 1 (2023), 1.

[45] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th international conference on mining software repositories*. 181–191.

[46] Kerstin Denecke, Richard May, LLMHealthGroup, and Octavio Rivera Romero. 2024. Potential of large language models in health care: Delphi study. *Journal of Medical Internet Research* 26 (2024), e52399.

[47] Dify. 2025. Dify: Build Production Ready Agentic Solution. https://dify.ai/, last visited: May 27.

[48] Inc Docker et al. 2020. Docker. *linea].[Junio de 2017]. Disponible en: https://www. docker. com/what-docker* (2020).

[49] Tore Dybå, Vigdis By Kampenes, and Dag IK Sjøberg. 2006. A systematic review of statistical power in software engineering experiments. *Information and Software Technology* 48, 8 (2006), 745–755.

[50] Filipe Falcão, Caio Barbosa, Baldoino Fonseca, Alessandro Garcia, Márcio Ribeiro, and Rohit Gheyi. 2020. On relating technical, social factors, and the introduction of bugs. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 378–388.

[51] Rosa Falotico and Piero Quatto. 2015. Fleiss' kappa statistic without paradoxes. *Quality & Quantity* 49 (2015), 463–470.

[52] Amir Hossein Ghapanchi. 2015. Predicting software future sustainability: A longitudinal perspective. *Information Systems* 49 (2015), 40–51.

[53] Sean Goggins, Kevin Lumbard, and Matt Germonprez. 2021. Open source community health: Analytical metrics and their corresponding narratives. In *2021 IEEE/ACM 4th International Workshop on Software Health in Projects, Ecosystems and Communities (SoHeal)*. IEEE, 25–33.

[54] Software Improvement Group. 2025. State of Software 2025: A Global Report on the Hidden Costs and Risks of Software. https://www.softwareimprovementgroup.com/wp-content/uploads/State-of-software-2025.pdf, last visited: May 08.

[55] Aakanshi Gupta, Rashmi Gandhi, Nishtha Jatana, Divya Jatain, Sandeep Kumar Panda, and Janjhyam Venkata Naga Ramesh. 2023. A severity assessment of python code smells. *IEEE Access* 11 (2023), 119146–119160.

[56] Md Shariful Haque, Jeff Carver, and Travis Atkison. 2018. Causes, impacts, and detection approaches of code smell: a survey. In *Proceedings of the 2018 ACM Southeast Conference*. 1–8.

[57] Mohammed Mehedi Hasan, Hao Li, Emad Fallahzadeh, Gopi Krishnan Rajbahadur, Bram Adams, and A. E Hassan. 2025. The replication package of our study on MCP Servers. https://github.com/SAILResearch/replication-25-mcp-server-empirical-study, last visited: Jun 11.

[58] Ahmed E Hassan, Gustavo A Oliva, Dayi Lin, Boyuan Chen, Zhen Ming, et al. 2024. Rethinking software engineering in the foundation model era: From task-driven ai copilots to goal-driven ai pair programmers. *arXiv preprint arXiv:2404.10225* (2024).

[59] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. 2023. Automating dependency updates in practice: An exploratory study on github dependabot. *IEEE Transactions on Software Engineering* 49, 8 (2023), 4004–4022.

[60] Israel Herraiz, Jesus M Gonzalez-Barahona, and Gregorio Robles. 2008. Determinism and evolution. In *Proceedings of the 2008 international working conference on Mining software repositories*. 1–10.

[61] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 426–437.

[62] Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. 2025. Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions. *arXiv preprint arXiv:2503.23278* (2025).

[63] IBM. 2025. Cost of a Data Breach Report 2024. https://www.ibm.com/reports/data-breach, last visited: May 27.

[64] Samuel Idowu, Yorick Sens, Thorsten Berger, Jacob Krüger, and Michael Vierhauser. 2024. A large-scale study of ml-related python projects. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*. 1272–1281.

[65] Alphabet Inc. 2025. Security checklist. https://developer.android.com/privacy-and-security/security-tips, last visited: June 03.

[66] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 6–pp.

[67] Jaehun Jung, Faeze Brahman, and Yejin Choi. 2024. Trust or Escalate: LLM Judges with Provable Guarantees for Human Agreement. *arXiv preprint arXiv:2407.18370* (2024).

[68] Arvinder Kaur and Ruchikaa Nayyar. 2020. A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code. *Procedia Computer Science* 171 (2020), 2023–2029.

[69] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. 2014. Why do automated builds break? an empirical study. In *2014 IEEE international conference on software maintenance and evolution*. IEEE, 41–50.

[70] Sonu Kumar, Anubhav Girdhar, Ritesh Patil, and Divyansh Tripathi. 2025. MCP Guardian: A Security-First Layer for Safeguarding MCP-Based AI System. *arXiv preprint arXiv:2504.12757* (2025).

[71] Invariant Lab. 2025. Introducing MCP-Scan: Protecting MCP with Invariant. https://invariantlabs.ai/blog/introducing-mcp-scan, last visited: May 29.

[72] Tuan Dung Lai, Anj Simmons, Scott Barnett, Jean-Guy Schneider, and Rajesh Vasa. 2024. Comparative analysis of real issues in open-source machine learning projects. *Empirical Software Engineering* 29, 3 (2024), 60.

[73] LangChain. 2025. LangChain: composable framework to build with LLMs . https://www.langchain.com/, last visited: Apr 23.

[74] Jasmine Latendresse, Suhaib Mujahid, Diego Elias Costa, and Emad Shihab. 2022. Not all dependencies are equal: An empirical study on production dependencies in npm. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[75] Luigi Lavazza, Sandro Morasca, and Davide Tosi. 2021. Comparing static analysis and code smells as defect predictors: an empirical study. In *IFIP international conference on open source systems*. Springer, 1–15.

[76] Valentina Lenarduzzi, Francesco Lomio, Heikki Huttunen, and Davide Taibi. 2020. Are sonarqube rules inducing bugs?. In *2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 501–511.

[77] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[78] Hao Li and Cor-Paul Bezemer. 2025. Bridging the language gap: an empirical study of bindings for open source machine learning libraries across software package ecosystems. *Empirical Software Engineering* 30, 1 (2025), 6.

[79] Hao Li, Cor-Paul Bezemer, and Ahmed E Hassan. 2024. Software Engineering and Foundation Models: Insights from Industry Blogs Using a Jury of Foundation Models. *arXiv preprint arXiv:2410.09012* (2024).

[80] Jiahuei Lin, Dayi Lin, Sky Zhang, and Ahmed E Hassan. 2024. Engineering AI Judge Systems. *arXiv preprint arXiv:2411.17793* (2024).

[81] LlamaIndex. 2025. LlamaIndex: LlamaIndex is the leading framework for building LLM-powered agents over your data with LLMs and workflows. https://docs.llamaindex.ai/en/stable/, last visited: Apr 23.

[82] Google LLC. 2025. GenAI Toolbox: An introduction to MCP Toolbox. https://googleapis.github.io/genai-toolbox/getting-started/introduction/, last visited: May 27.

[83] Guillermo Macbeth, Eugenia Razumiejczyk, and Rubén Daniel Ledesma. 2011. Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica* 10, 2 (2011), 545–555.

[84] Thomas W MacFarland, Jan M Yates, Thomas W MacFarland, and Jan M Yates. 2016. Kruskal–Wallis H-test for oneway analysis of variance (ANOVA) by ranks. *Introduction to nonparametric statistics for the biological sciences using R* (2016), 177–211.

[85] Meta. 2025. Function Calling: Llama 3.1 models now officially supports function calling. https://github.com/abetlen/llama-cpp-python/issues/1618, last visited: May 15.

[86] Microsoft. 2025. Introducing Model Context Protocol (MCP) in Copilot Studio: Simplified Integration with AI Apps and Agents. https://www.microsoft.com/en-us/microsoft-copilot/blog/copilot-studio/introducing-model-context-protocol-mcp-in-copilot-studio-simplified-integration-with-ai-apps-and-agents, last visited: Apr 23.

[87] Microsoft. 2025. Plug, Play, and Prey: The security risks of the Model Context Protocol. https://techcommunity.microsoft.com/blog/microsoftdefendercloudblog/plug-play-and-prey-the-security-risks-of-the-model-context-protocol/4410829, last visited: May 18.

[88] Vishal Midha and Prashant Palvia. 2012. Factors affecting the success of Open Source Software. *Journal of Systems and Software* 85, 4 (2012), 895–905.

[89] Mitre. 2025. Common Weakness Enumeration: : A community developed list of SW & HW weakness that can become vulnerabilities. https://cwe.mitre.org/index.html, last visited: May 15.

[90] Mohammed Abdul Moid, Abdullah Siraj, Mohd Farhaan Ali, and Ahmed Osman Amoodi. 2021. Predicting Stars on Open-Source GitHub Projects. In *2021 Smart Technologies, Communication and Robotics (STCR)*. IEEE, 1–9.

[91] Dietmar PF Möller. 2023. NIST cybersecurity framework and MITRE cybersecurity criteria. In *Guide to Cybersecurity in Digital Transformation: Trends, Methods, Technologies, Applications and Best Practices*. Springer, 231–271.

[92] Suhaib Mujahid, Rabe Abdalkareem, and Emad Shihab. 2023. What are the characteristics of highly-selected packages? A case study on the npm ecosystem. *Journal of Systems and Software* 198 (2023), 111588.

[93] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating github for engineered software projects. *Empirical Software Engineering* 22 (2017), 3219–3253.

[94] Marvin Muñoz Barón, Marvin Wyrich, and Stefan Wagner. 2020. An empirical validation of cognitive complexity as a measure of source code understandability. In *Proceedings of the 14th ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)*. 1–12.

[95] Vineeth Sai Narajala and Idan Habler. 2025. Enterprise-Grade Security for the Model Context Protocol (MCP): Frameworks and Mitigation Strategies. *arXiv preprint arXiv:2504.08623* (2025).

[96] Mathew Nicho, Hussein Fakhry, and Charles Haiber. 2011. An integrated security governance framework for effective PCI DSS implementation. *International Journal of Information Security and Privacy (IJISP)* 5, 3 (2011), 50–67.

[97] OpenAI. 2025. Function Calling: Enable models to fetch data and take actions. https://platform.openai.com/docs/guides/function-calling, last visited: May 15.

[98]   OpenAI. 2025. OpenAI Agents SDK: Model context protocol (MCP). https://openai.github.io/openai-agents-python/mcp, last visited: Apr 23.

[99]   OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, et al. 2024. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2024).

[100] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. In *Proceedings of the 40th International Conference on Software Engineering*. 482–482.

[101] Thomas H Park, Brian Dorn, and Andrea Forte. 2015. An analysis of HTML and CSS syntax errors in a web development course. *ACM Transactions on Computing Education (TOCE)* 15, 1 (2015), 1–21.

[102] PMD. 2025. An extensible multilanguage static code analyzer. https://pmd.github.io/, last visited: May 18.

[103] Luís Prates and Rúben Pereira. 2025. DevSecOps practices and tools. *International Journal of Information Security* 24, 1 (2025), 1–25.

[104] G Priyalakshmi and R Latha. 2018. Evaluation of software reusability based on coupling and cohesion. *International Journal of Software Engineering and Knowledge Engineering* 28, 10 (2018), 1455–1485.

[105] Model Context Protocol. 2025. Model Context Protocol servers. https://github.com/modelcontextprotocol/servers, last visited: Apr 23.

[106] Brandon Radosevich and John Halloran. 2025. MCP Safety Audit: LLMs with the Model Context Protocol Allow Major Security Exploits. *arXiv preprint arXiv:2504.03767* (2025).

[107] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 164–175.

[108] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 155–165.

[109] Eric Raymond. 1999. The cathedral and the bazaar. *Knowledge, Technology & Policy* 12, 3 (1999), 23–49.

[110] Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. 2017. Detecting argument selection defects. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–22.

[111] Chanchal K Roy, Minhaz F Zibran, and Rainer Koschke. 2014. The vision of software clone management: Past, present, and future (keynote paper). In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 18–33.

[112] Jukka Ruohonen. 2018. An empirical analysis of vulnerabilities in python packages for web applications. In *2018 9th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. IEEE, 25–30.

[113] Jukka Ruohonen, Kalle Hjerppe, and Kalle Rindell. 2021. A large-scale security-oriented static analysis of python packages in pypi. In *2021 18th International Conference on Privacy, Security and Trust (PST)*. IEEE, 1–10.

[114] Graeme D Ruxton and Guy Beauchamp. 2008. Time for some a priori thinking about post hoc testing. *Behavioral ecology* 19, 3 (2008), 690–693.

[115] Dhia Elhaq Rzig, Foyzul Hassan, Chetan Bansal, and Nachiappan Nagappan. 2022. Characterizing the usage of ci tools in ml projects. In *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 69–79.

[116] Amir Saboury, Pooya Musavi, Foutse Khomh, and Giulio Antoniol. 2017. An empirical study of code smells in javascript projects. In *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 294–305.

[117] Haya Samaana, Diego Elias Costa, Ahmad Abdellatif, and Emad Shihab. 2025. Opportunities and security risks of technical leverage: A replication study on the NPM ecosystem. *Empirical Software Engineering* 30, 3 (2025), 96.

[118] Víctor Rea Sánchez, Pablo Neira Ayuso, José A Galindo, and David Benavides. 2020. Open source adoption factors—a systematic literature review. *IEEE Access* 8 (2020), 94594–94609.

[119] Aleksei Shestov, Rodion Levichev, Ravil Mussabayev, Evgeny Maslov, Pavel Zadorozhny, Anton Cheshkov, Rustam Mussabayev, Alymzhan Toleu, Gulmira Tolegen, and Alexander Krassovitskiy. 2025. Finetuning large language models for vulnerability detection. *IEEE Access* (2025).

[120] Nima Shiri Harzevili, Alvine Boaye Belle, Junjie Wang, Song Wang, Zhen Ming Jiang, and Nachiappan Nagappan. 2024. A systematic literature review on automated software vulnerability detection using machine learning. *Comput. Surveys* 57, 3 (2024), 1–36.

[121] Mohammed Latif Siddiq, Shafayat H Majumder, Maisha R Mim, Sourov Jajodia, and Joanna CS Santos. 2022. An empirical study of code smells in transformer-based code generation techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 71–82.

[122] Randeep Singh and Ashok Kumar. 2018. Identifying various code-smells and refactoring opportunities in object-oriented software system: a systematic literature review. *International Journal on Future Revolution in Computer Science & Communication Engineering* 8, March (2018), 62–74.

[123] Satwinder Singh and Sharanpreet Kaur. 2018. A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal* 9, 4 (2018), 2129–2151.

[124] sixsentix. 2025. Ten most expensive bugs in history (part 2). https://www.sixsentix.com/insights/ten-most-expensive-bugs-in-history-part-2, last visited: May 27.

[125] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2018. How developers diagnose potential security vulnerabilities with a static analysis tool. *IEEE Transactions on Software Engineering* 45, 9 (2018), 877–897.

[126] smithery. 2025. Smithery: Your Agent's Gateway to the World. https://smithery.ai/, last visited: May 15.

[127] SonarQube. 2025. SonarQube: Static Code Analysis. https://docs.sonarsource.com/sonarqube-server/latest/user-guide/code-metrics/metrics-definition/#se-severity-types, last visited: May 14.

[128] SonarQube. 2025. SonarQube: Static Code Analysis. https://docs.sonarsource.com/sonarqube-server/10.8/user-guide/rules/security-related-rules, last visited: May 14.

[129] Jairo Souza, Tales Alves, Robson Oliveira, Leopoldo Teixeira, and Baldoino Fonseca. 2024. Exception Miner: Multi-language Static Analysis Tool to Identify Exception Handling Anti-Patterns. In *Simpósio Brasileiro de Engenharia de Software (SBES)*. SBC, 741–747.

[130] Ştefan Stănciulescu, Likang Yin, and Vladimir Filkov. 2022. Code, quality, and process metrics in graduated and retired asfi projects. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 495–506.

[131] Stripe. 2025. The Stripe Model Context Protocol server allows you to integrate with Stripe APIs through function calling. https://github.com/stripe/agent-toolkit/tree/main/modelcontextprotocol, last visited: May 27.

[132] Sebastian Sztwiertnia, Maximilian Grübel, Amine Chouchane, Daniel Sokolowski, Krishna Narasimhan, and Mira Mezini. 2021. Impact of programming languages on machine learning bugs. In *Proceedings of the 1st ACM International Workshop on AI and Software Testing/Analysis*. 9–12.

[133] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *arXiv preprint arXiv:2302.13971* (2023).

[134] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 403–414.

[135] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. 2018. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 644–655.

[136] Bart Van Oort, Luís Cruz, Maurício Aniche, and Arie Van Deursen. 2021. The prevalence of code smells in machine learning projects. In *2021 IEEE/ACM 1st workshop on AI engineering-software engineering for AI (WAIN)*. IEEE, 1–8.

[137] Morteza Verdi, Ashkan Sami, Jafar Akhondali, Foutse Khomh, Gias Uddin, and Alireza Karami Motlagh. 2020. An empirical study of c++ vulnerabilities in crowd-sourced code examples. *IEEE Transactions on Software Engineering* 48, 5 (2020), 1497–1514.

[138] James Walden. 2020. The impact of a major security event on an open source project: The case of OpenSSL. In *Proceedings of the 17th international conference on mining software repositories*. 409–419.

[139] Haoyu Wang, Hao Li, Li Li, Yao Guo, and Guoai Xu. 2018. Why are android apps removed from google play? a large-scale empirical study. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 231–242.

[140] Weichao Wang, Zhaopeng Meng, Zan Wang, Shuang Liu, and Jianye Hao. 2019. LoopFix: An approach to automatic repair of buggy loops. *Journal of Systems and Software* 156 (2019), 100–112.

[141] Yingxu Wang and Vincent Chiew. 2013. Empirical studies on the functional complexity of software in large-scale software systems. In *Advances in Abstract Intelligence and Soft Computing*. IGI Global Scientific Publishing, 174–192.

[142] Thomas Weber, Maximilian Brandmaier, Albrecht Schmidt, and Sven Mayer. 2024. Significant productivity gains through programming with large language models. *Proceedings of the ACM on Human-Computer Interaction* 8, EICS (2024), 1–29.

[143] Katrine Wist, Malene Helsem, and Danilo Gligoroski. 2021. Vulnerability analysis of 2500 docker hub images. In *Advances in Security, Networks, and Internet of Things: Proceedings from SAM'20, ICWN'20, ICOMP'20, and ESCS'20*. Springer, 307–327.

[144] Di Wu, Fangwen Mu, Lin Shi, Zhaoqiang Guo, Kui Liu, Weiguang Zhuang, Yuqi Zhong, and Li Zhang. 2024. iSMELL: Assembling LLMs with Expert Toolsets for Code Smell Detection and Refactoring. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1345–1357.

[145] Shijie Wu, Ozan Irsoy, Steven Lu, Vadim Dabravolski, Mark Dredze, Sebastian Gehrmann, Prabhanjan Kambadur, David Rosenberg, and Gideon Mann. 2023. Bloomberggpt: A large language model for finance. *arXiv preprint*

arXiv:2303.17564 (2023).

[146] Wenxin Xiao, Hao He, Weiwei Xu, Yuxia Zhang, and Minghui Zhou. 2023. How early participation determines long-term sustained activity in github projects?. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 29–41.

[147] Pravin Singh Yadav, Rajwant Singh Rao, Alok Mishra, and Manjari Gupta. 2024. Machine learning-based methods for code smell detection: a survey. *Applied Sciences* 14, 14 (2024), 6149.

[148] Mehmet Ali Yalçinkaya and Ecir Uğur Küçüksille. 2024. Artificial Intelligence and Dynamic Analysis-Based Web Application Vulnerability Scanner. *ISeCure* 16, 1 (2024).

[149] Aiko Yamashita and Steve Counsell. 2013. Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software* 86, 10 (2013), 2639–2653.

[150] Aiko Yamashita and Leon Moonen. 2012. Do code smells reflect important maintainability aspects?. In *2012 28th IEEE international conference on software maintenance (ICSM).* IEEE, 306–315.

[151] Ping Yu, Yijian Wu, Jiahan Peng, Jian Zhang, and Peicheng Xie. 2023. Towards understanding fixes of sonarqube static analysis violations: A large-scale empirical study. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 569–580.

[152] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. 2018. An empirical analysis of technical lag in npm package dependencies. In *International conference on software reuse.* Springer, 95–110.

[153] Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. 2022. On the impact of security vulnerabilities in the npm and RubyGems dependency networks. *Empirical Software Engineering* 27, 5 (2022), 107.

[154] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX security symposium (USENIX Security 19).* 995–1010.

[155] Weiqin Zou, Weiqiang Zhang, Xin Xia, Reid Holmes, and Zhenyu Chen. 2019. Branch use in practice: A large-scale empirical study of 2,923 projects on github. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS).* IEEE, 306–317.