Everyday AI for Statistical Programmers: Global Implementation of a Clinical Analysis Assistant

Mathieu Cayssol Dr. Christoph Centner

Hoffmann–La Roche, Basel, Switzerland

Abstract

The increasing complexity of clinical trial design and statistical analyses has significantly raised the expectations placed on statistical programmers. With the growing number of therapeutic areas, study designs, and regulatory requirements, programmers are tasked with producing high-quality, validated code under tight timelines. In many organizations, code bases are distributed across multiple projects and repositories, often resulting in fragmented knowledge and duplicated effort, particularly when creating critical Tables, Listings, and Graphs (TLGs). Re-using code remains a time-consuming, manual task. The lack of a centralized, intelligent mechanism to retrieve and reuse previously validated code fragments exacerbates inefficiencies and increases the risk of inconsistencies across studies.

Enhancing code reusability is therefore a critical priority. By enabling programmers to quickly locate, adapt, and apply existing validated code, organizations can reduce redundancy, ensure consistency across analyses, and accelerate time-to-market. Traditional keyword-based search is inadequate for this task, failing to match a programmer's semantic intent with validated code snippets that may use different variable or function names.

This work introduces the Code Search Agent, a novel AI-powered solution that directly addresses the code fragmentation challenge. The Agent combines generative AI with a large-scale, metadata-driven code search capability. By indexing over 70,000 programs repositories and enriching code chunks with LLM-generated summaries, the tool makes historical programs accessible through natural language queries, thereby lowering the barrier to reuse. This structured approach enhances efficiency by reducing redundant programming, while also improving quality, consistency, and standardization across projects. The Code Search Agent is delivered as a specialized component of the existing Clinical Analysis Assistant RAG framework, providing an essential intelligent layer for scaling statistical programming capacity.

Keywords: AI, RAG, MCP, LLM, Agents, statistical programming, clinical analysis, Chatbot, Clinical Study, Clinical programming

1 Introduction

The operational landscape for statistical programmers is characterized by increasing expectations, driven by the expansion of therapeutic specialization, the complexity of study protocols, and increasingly stringent compliance frameworks. In many organizations, a prevalent challenge is the low rate of code re-utilization across disparate therapeutic areas and operational projects. While centralized repositories might be available, knowledge is fragmented and the reuse of existing code is often a manual and time-consuming effort. The lack of a centralized, intelligent mechanism to retrieve and reuse previously validated code fragments exacerbates inefficiencies and increases the risk of inconsistencies across studies. Enhancing code reusability is therefore a critical priority. By enabling programmers to quickly locate, adapt, and apply existing validated code, organizations can reduce redundancy, ensure consistency across analyses, and accelerate time-to-market. Furthermore, reusing proven code not only saves time but also improves quality by leveraging programs that have already been tested and reviewed in prior studies. Lastly, also harmonization of outputs across studies plays a critical role. The Code Search Agent directly addresses these challenges by combining generative AI with a large-scale, metadata-driven code search capability. By indexing over 70,000 programs, the tool makes historical programs accessible through natural language queries, thereby lowering the barrier to reuse. This structured approach enhances efficiency by reducing redundant programming, while also improving quality, consistency, and standardization across projects. Ultimately, such an intelligent assistant is a key enabler for scaling statistical programming capacity in the era of increasingly data-intensive clinical trials.

2 Background

2.1 OCEAN Platform Overview

The OCEAN (One CEntralised ANalytics) platform at Roche provides an integrated, governed, and scalable environment for large-scale clinical programming. It establishes a modern analytics ecosystem that combines secure storage, containerized workbenches, and version-controlled development in GitLab. Within this architecture, the full analytical workflow (from raw data ingestion to submission-ready deliverables) is standardized and traceable.

OCEAN introduces several key evolutions compared to legacy environments:

- A transition from traditional SAS-only workflows to a **multi-language ecosystem** that supports both SAS and R, enabling hybrid analyses and modernized pipelines.
- A **Git-based workflow** [1] for version control, branching, and collaboration, replacing older file-based paradigms.
- Integration of **GitLab** [2] for repository management, access control, and traceability.
- **Automated orchestration** of analyses via Snakemake, ensuring reproducible and parallel execution of pipelines across containerized environments.
- A clear separation between code, data, and results to reinforce compliance, scalability, and governance.

As illustrated in figure 2, data flows from raw data stores into governed clinical data repositories, that allows analysts to work within isolated, containerized workspaces supporting SAS, R, and Python. Snakemake orchestrates pipeline execution, while GitLab ensures full versioning, auditability, and collaboration. Together, these components provide a robust foundation for reproducibility, scalability, and cross-functional collaboration across therapeutic areas.

Figure 1: Overview of the clinical programming environment under OCEAN

2.2 Challenges

While OCEAN provides a technically advanced and future-proof foundation for clinical programming, its adoption introduces a new level of complexity for users. The shift from traditional, GUI-based workflows to a multi-language, containerized, and Git-driven ecosystem brings modern software engineering concepts that are powerful, but initially unfamiliar, to many statistical programmers. This transformation requires both technical upskilling and cultural adaptation, as teams move toward collaborative, code-centric practices. Once mastered, these approaches deliver substantial gains in reproducibility, automation, and analytical quality, yet the transition itself poses several challenges.

During the rollout of OCEAN, five primary challenges were identified:

- **Steep learning curve:** Users must acquire new skills in Git operations (branching, merging, version control), container management, and pipeline orchestration. Mastery of these concepts takes time but is essential for unlocking the platform's full potential.
- **Ecosystem transition:** Teams accustomed to single-tool environments (e.g., SAS-only workflows) must adapt to a broader, integrated toolchain spanning GitLab, Snakemake, R, and Python.
- **Increased technical depth:** Modern workflows demand familiarity with command-line interfaces, package management, and multi-language debugging. While initially daunting, these skills enable more reproducible and efficient analyses.
- **Fragmented information access:** Guidance, examples, and validated code are distributed across multiple repositories, wikis, and documentation systems, making it difficult for users to locate the right information quickly.
- **Code discoverability and reuse:** A major advantage of OCEAN lies in its modern Git-based workflow-code is version-controlled and centrally managed in GitLab, providing unprecedented accessibility and traceability across projects. However, even though validated programs are now available in one unified environment, identifying, comparing, and reusing relevant code fragments across repositories remains a time-consuming, manual task. This gap underscores the need for intelligent tooling to make the wealth of validated code truly reusable, consistent, and standardized across studies.

To bridge these gaps and accelerate user adoption, the Clinical Analysis Assistant (CAA) was developed as an intelligent conversational interface on top of OCEAN. By allowing users to interact

with the system through natural language, the assistant simplifies access to documentation, coding standards, and reusable programs, thereby reducing the cognitive load during onboarding. Beyond easing the learning curve, CAA enhances long-term productivity by enabling programmers to focus on scientific analysis rather than technical troubleshooting. Within this framework, the newly introduced **Code Search Agent (CSA)** plays a pivotal role, addressing one of the most persistent challenges in clinical programming: efficiently finding, comparing, and reusing validated code across projects to promote harmonization, standardization, and faster delivery of analysis outputs.

3 Solution

3.1 Clinical Analysis Assistant

The Clinical Analysis Assistant (CAA) is an AI-powered conversational layer built on top of the OCEAN platform that helps clinical programmers efficiently navigate the environment, access documentation, and reuse validated code. Through a chat-based interface, users can interact with specialized virtual agents supporting different aspects of the clinical programming workflow. CAA is implemented using Chainlit to provide the conversational interface and is securely hosted on Posit Connect [9], leveraging OCEAN's existing authentication and access control framework.

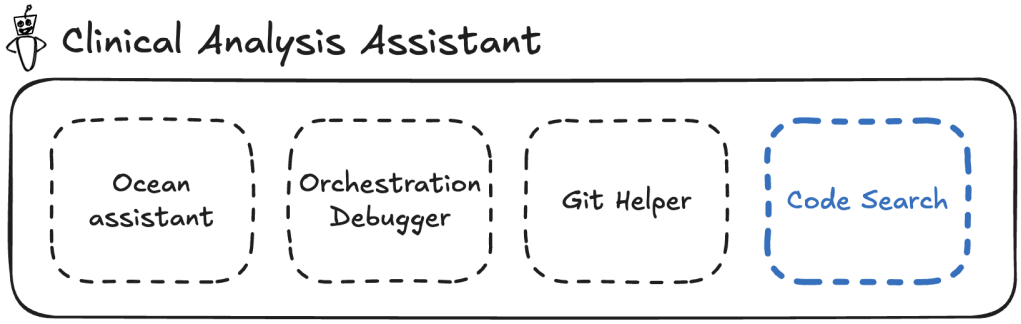

Figure 2: Clinical Analysis Assistant (CAA) and its 4 agents

The first three agents focus on platform navigation and troubleshooting: 1) the *Ocean Assistant* provides guidance on environment usage and best practices; 2) the *Orchestration Debugger* assists in diagnosing pipeline or container execution errors; and 3) the *Git Helper* simplifies version-control operations such as branching and merging. Together, these agents streamline onboarding and daily operations within OCEAN.

The fourth and most recent addition (the **Code Search Agent (CSA)**) is the focus of this work. CSA introduces metadata- and semantic-driven search across thousands of GitLab repositories, enabling programmers to locate validated and reusable code fragments using natural language queries. By returning results enriched with contextual metadata (e.g., study ID, description, activity name), CSA transforms code reuse from a manual, time-consuming process into an efficient and standardized practice across studies.

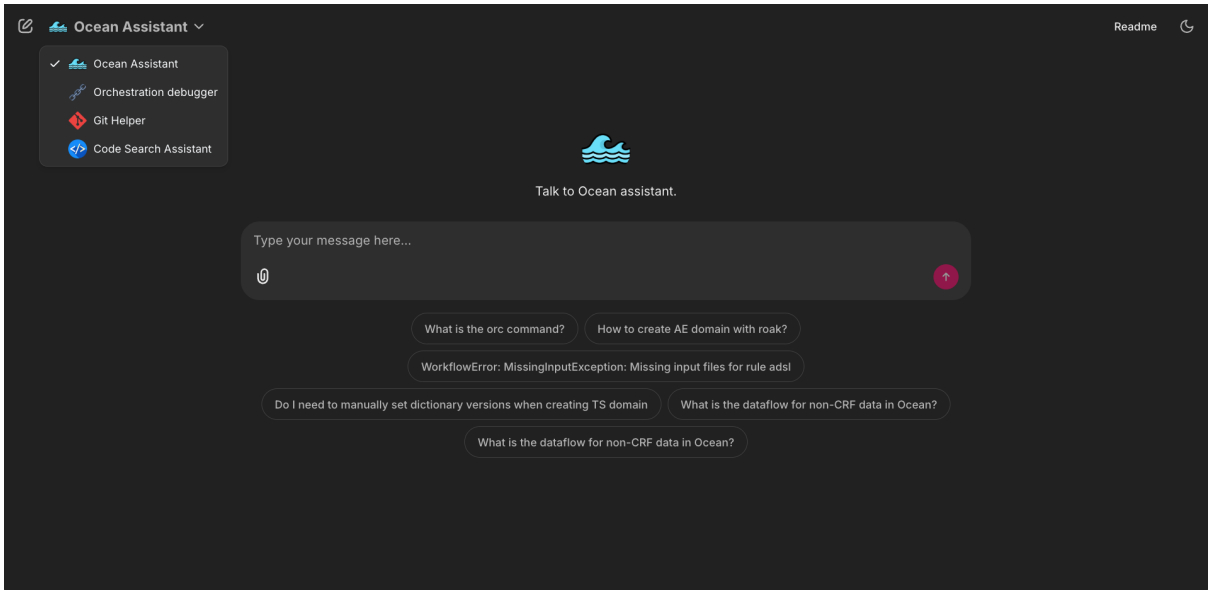

Figure 3: Clinical Analysis Assistant user interface with integrated Code Search Agent.

3.2 Code Search Agent

The Code Search Agent (CSA) is the newest feature or the CAA and enables users to query large code bases using natural language. It translates the user's input into a semantic query enriched with metadata filters (e.g., repository, function name, programming language) to ensure precise retrieval. The query is then executed against a vector database containing indexed source code, and the most relevant results are returned and summarized. This workflow accelerates navigation of complex code bases and improves transparency in retrieval. The overall process is illustrated in figure 4.

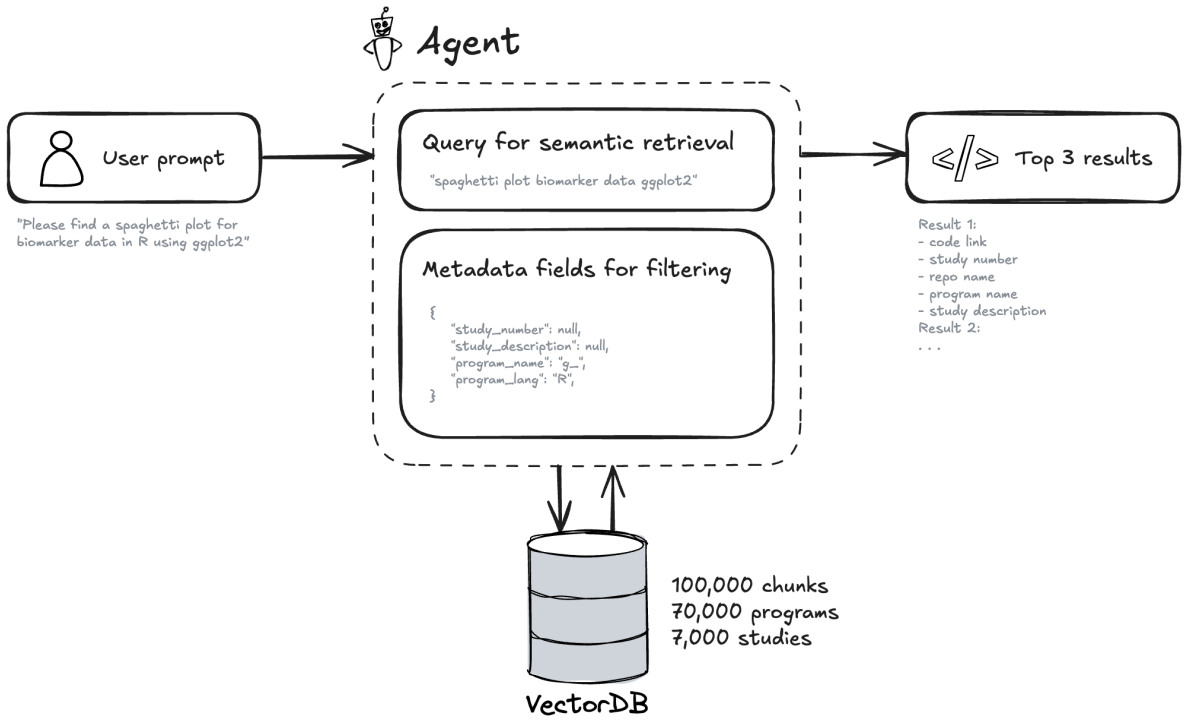

Figure 4: High-level workflow of the Code Search Agent, from user query to retrieved results.

The workflow proceeds in three steps:

1. **User Input:** The user provides a natural language query.
2. **Query Transformation:** The Agent converts the query into a semantic representation, augmented with metadata filters, to retrieve contextually relevant information from the vector database.
3. **Result Retrieval and Display:** The system returns the most relevant code snippets, which are summarized and presented to the user along with the semantic query and applied metadata filters for full transparency.

From a UI perspective, the interaction is shown in figure 5, where users can enter a query, inspect the results, and view the underlying semantic search parameters.

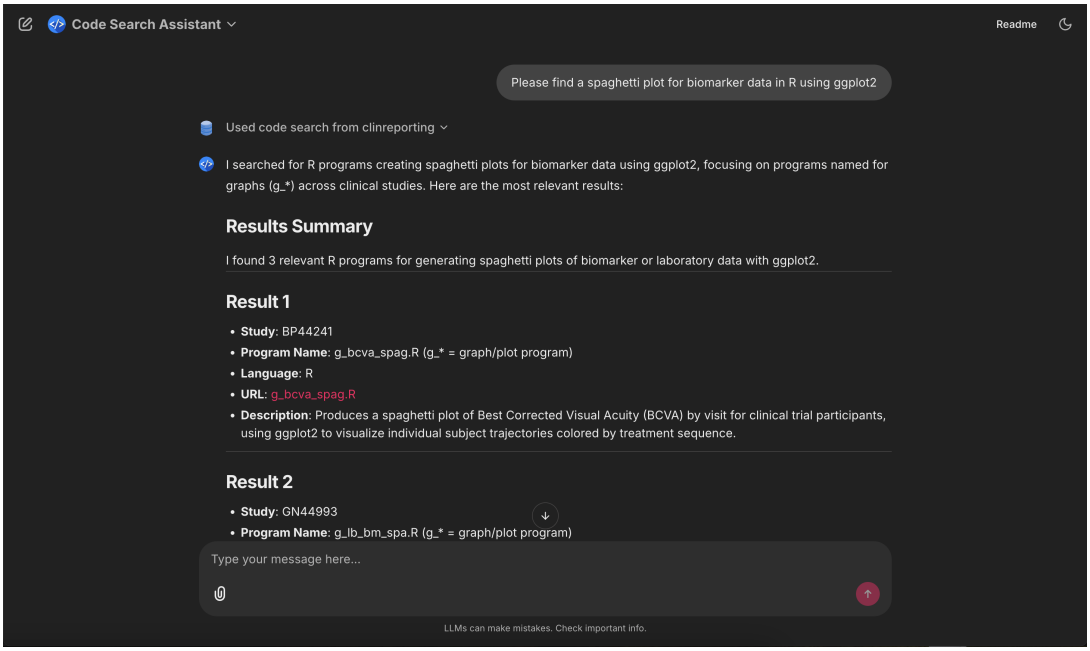

Figure 5: User interface for the Code Search Agent

The Agent executes code search operations in the background, ensuring responsiveness. By collapsing intermediate steps, the interface provides users with immediate access to: (i) the semantic query used for retrieval, and (ii) the metadata fields applied to refine results. This design increases transparency and fosters confidence in the accuracy of retrieved information.

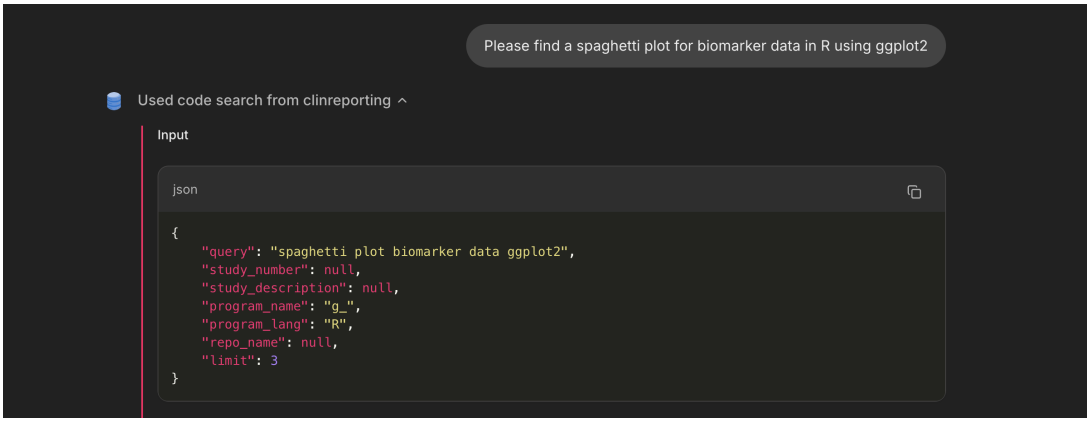

Figure 6: Tool invocation used to query the vector database containing code from the `clinreporting` GitLab group.

[13]

4 Technical Architecture

The Clinical Analysis Assistant is built on three core pillars: reliable data pipelines, retrieval-augmented generation (RAG), and the Model Context Protocol (MCP). Together, these elements ensure scalability, compliance, and robust data quality, while enabling seamless integration across systems. Within this framework, the code search capability is designed according to the same architectural principles, providing statistical programmers and developers with a consistent and dependable way to access, query, and understand source code across repositories. We will begin by briefly introducing the concepts behind RAG and MCP. Next, we will present the overall chatbot architecture. Finally, we will explain how the dedicated data pipeline enables efficient and reliable retrieval of data.

4.1 What is RAG?

Retrieval-Augmented Generation (RAG) is a method that combines the strengths of search and generative AI to deliver more accurate and grounded answers. Instead of relying only on what a language model has memorized during training, RAG introduces an additional step: bringing in relevant, external knowledge at the moment a query is asked.

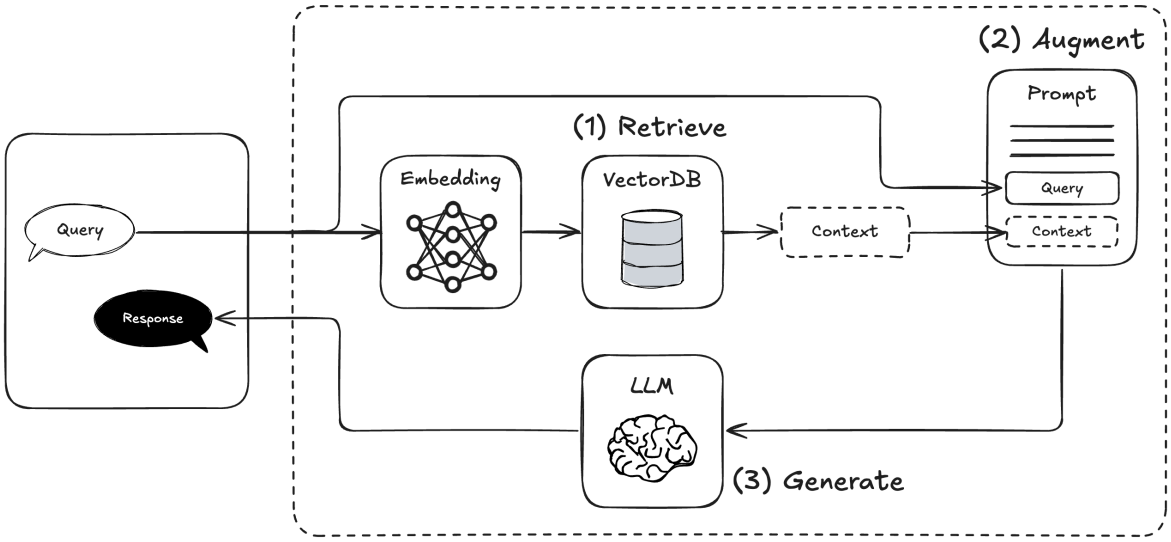

Figure 7: Retrieval Augmented Generation workflow (RAG)

This process unfolds in three main stages:

1. **Retrieve:** The user's query is transformed into an embedding and searched against a vector database to find the most relevant pieces of information (context).
2. **Augment:** The retrieved context is combined with the original query to form a richer, more informative prompt.
3. **Generate:** The Large Language Model (LLM) uses this augmented prompt to produce a response that is both coherent and factually grounded.

By connecting these steps together, RAG enables AI systems to dynamically pull in external knowledge, ensuring their answers are not only fluent but also accurate and up to date. [3] [4]

4.2 What is MCP?

The Model Context Protocol (MCP) was designed to make it easier for AI systems to work with external tools, databases, and APIs. Normally, every system has its own interface, which makes integration complicated and fragile. MCP fixes this by providing a standard way for AI to "plug in" to many different services without custom connectors each time.

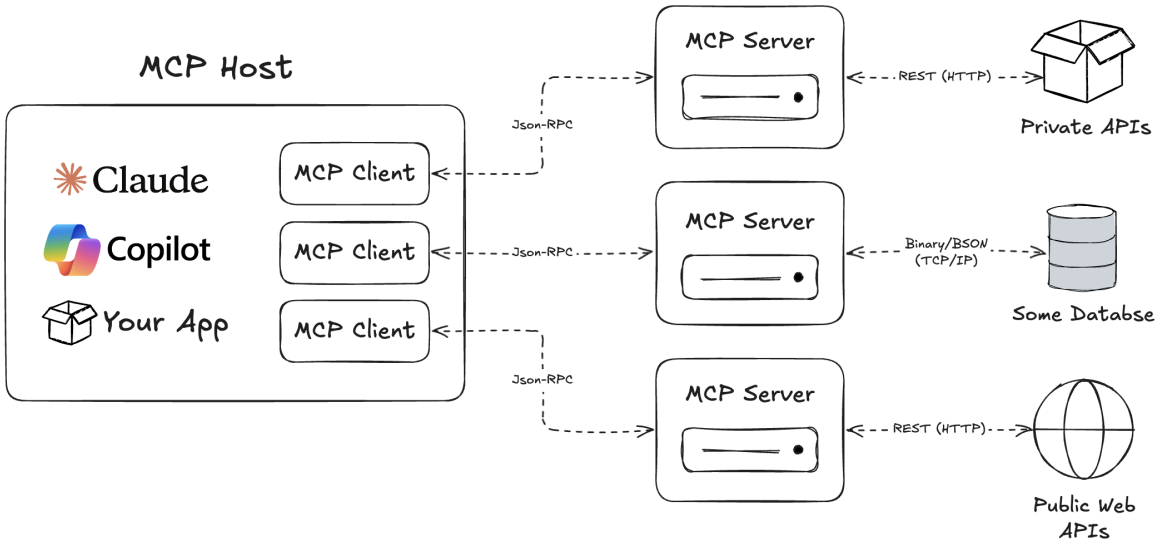

Figure 8: Model Context Protocol architecture

MCP works through three main components:

1. **MCP Host:** The core environment where the AI application (such as Claude, Cursor, or a custom app) runs and orchestrates requests.
2. **MCP Client:** Embedded in the host, it translates AI or user requests into MCP calls, manages communication, and handles responses.
3. **MCP Server:** Connected to external systems, such as private APIs, databases, or public web services. It executes requests and sends back results in a consistent format.

These components communicate using a lightweight protocol (JSON-RPC). The host and client handle AI interactions, while servers connect to real-world resources such as a database query, an internal API, or a public web endpoint. This means the AI does not need to know the details of each service: it simply communicates through MCP. The benefits are clear: MCP makes integrations simpler, more reliable, and easier to maintain. Developers can scale their systems without constantly re-inventing connections, and in multi-agent setups, MCP allows agents to share tools and workflows smoothly. In short, MCP reduces complexity and accelerates innovation by giving AI a common language for working with the broader ecosystem. [5] [6] [7]

4.3 CAA and CSA Architecture

In the context of the CAA, and more specifically the CSA, we have designed an architecture based on the Model Context Protocol (MCP). The objective of this section is to describe the current MCP Host and MCP Server, and explain how they have been implemented.

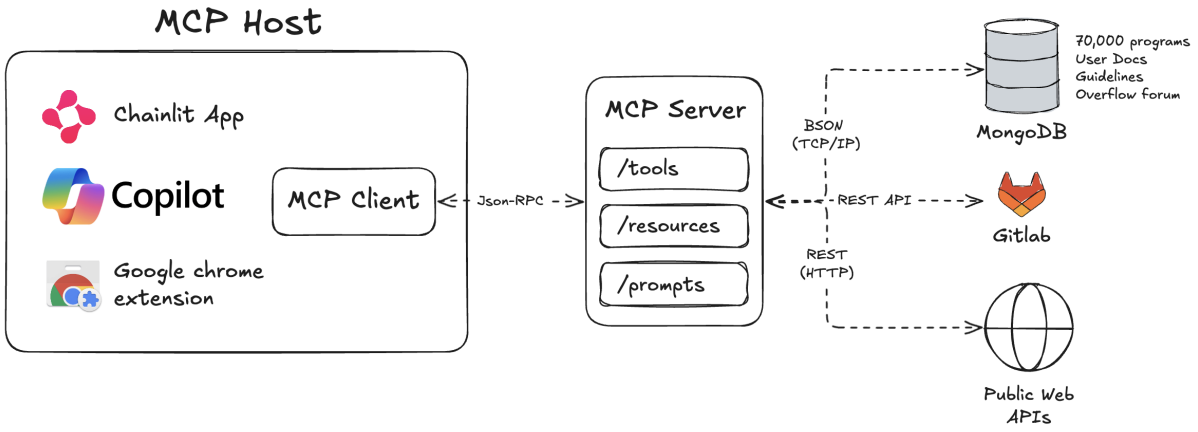

Figure 9: MCP architecture for clinical analysis assistant

MCP Host

The main MCP Host is a Chainlit [8] application deployed on Posit Connect [9]. This setup was chosen to leverage SSO authentication and ensure application security. The host integrates the OpenAI Agent SDK, which connects to the MCP Server via the MCP client.

To enhance usability, we provide a Google Chrome extension that avoids frequent tab switching, thereby improving the overall user experience. By adopting the MCP protocol, we maintain flexibility and agility in the fast-moving AI landscape. This allows us to integrate the assistant into any MCP Host.

In the future, our goal is to support direct integration within IDEs (such as GitHub Copilot, Cursor, and others), thereby embedding the assistant closer to developers' workflows.

MCP Server

The MCP Server has been implemented using `fastMCP`, which is built on FastAPI [10]. FastMCP enables the rapid definition of the three MCP primitives:

- **Resources:** External knowledge bases or APIs exposed to the agent. For example, documentation repositories, clinical datasets, or code repositories that the assistant can query.
- **Tools:** Functions or APIs that the assistant can call to perform specific tasks. For the Code Search Assistant, a tool has been defined to retrieve code snippets from the vector database. This allows the agent to return accurate and contextually relevant code fragments in response to user queries.
- **Prompts:** Pre-defined prompt templates that guide the behavior of the agent in specific contexts (e.g., code explanation, error debugging, or query reformulation).

The MCP Server is deployed as an ASGI web server (via `uvicorn`) [11] and hosted on Posit Connect [9]. This ensures scalability, performance, and easy maintainability.

4.4 Data Pipeline

The data pipeline for clinical analysis assistant and more specifically for the code search is a standard ETL (Extract, Transform, Load). The architecture can be seen as:

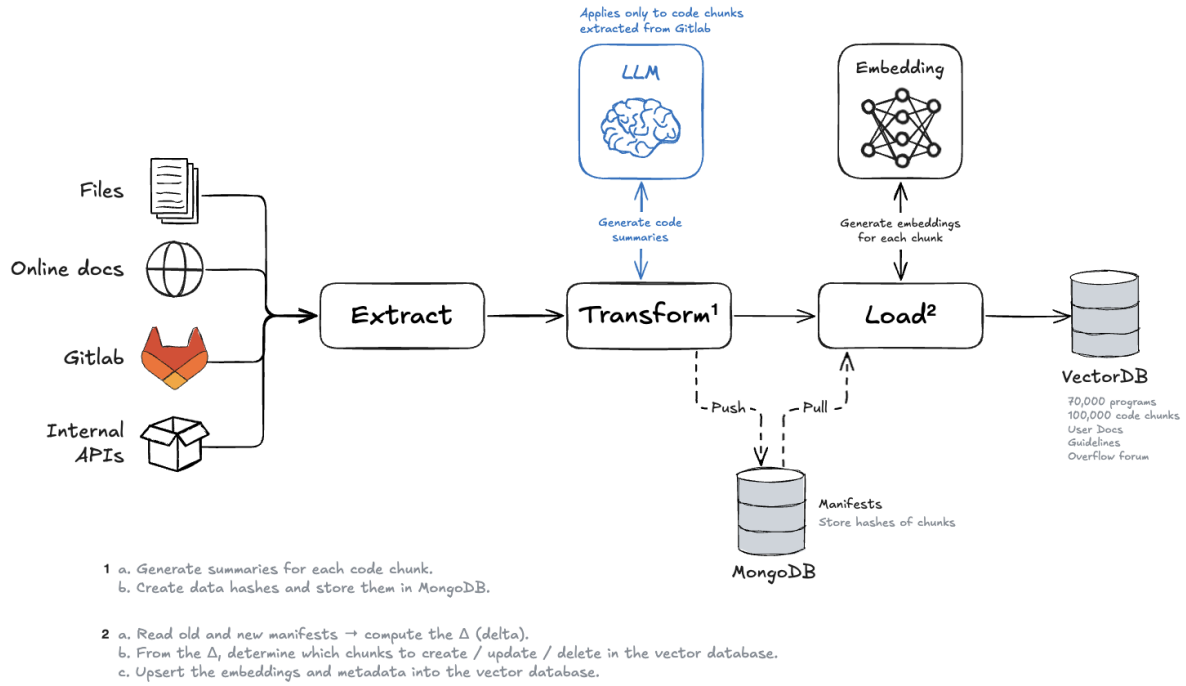

Figure 10: ETL for the CAA (the specifications for CSA are highlighted in blue color)

4.4.1 Extract

The Extract stage is responsible for gathering data from a broad and heterogeneous set of sources. In total, more than fifteen distinct data sources are integrated, including package documentation, Excel workbooks (`xlsx`), Quarto Markdown files [12] (`qmd`), code repositories stored on GitLab, internal guidelines, user forum for troubleshooting, and data pulled from internal APIs. Each source is accessed using dedicated connectors or APIs. At this stage, the system focuses solely on collecting raw content and associated metadata such as file path, repository, programming language, document type, and update timestamp. No data segmentation or transformation is yet performed. The output of this step is a unified pool of raw, structured and unstructured content ready for downstream processing.

4.4.2 Transform

The Transform stage is where the raw data undergoes normalization, segmentation, enrichment, and embedding preparation. The process begins with text cleaning (removal of markup, formatting symbols, and irrelevant tokens) and metadata validation to ensure integrity and consistency across diverse data sources. The cleaned text or code is then decomposed into smaller, semantically coherent *chunks*. These chunks represent the smallest retrievable units in the system and provide the granularity needed for efficient vector-based search. Each chunk is subsequently assigned a unique *data hash*, which is used for version control and change detection. While the hashing logic may vary depending on the data source (for example, based on commit ID and file path for GitLab code, or checksum and timestamp for documents), the manifest structure stored in MongoDB remains consistent. These manifests are later used to compute deltas and maintain synchronization with the vector database.

For CSA, code chunks are extracted from GitLab repositories, which necessitates an additional transformation step. A LLM generates a natural language summary for each code segment, describing its functionality, inputs, and outputs from a statistical programmer's perspective. Prompt engineering has been performed in collaboration with statistical programmers to ensure

that these descriptions are meaningful and relevant within the context of clinical studies. These summaries improve semantic retrieval by allowing natural language queries to match relevant code snippets, even when explicit keywords differ.

Finally, embeddings are generated for every chunk, regardless of type. Each chunk is converted into a high-dimensional vector representation capturing its semantic meaning. This embedding process enables context-aware retrieval and ensures that similar content across heterogeneous sources can be surfaced efficiently during query time.

4.4.3 Load

The load stage performs the final synchronization between the transformed dataset and the operational VectorDB. This process is incremental and governed by manifest comparison: old and new manifests are retrieved from MongoDB, and their differences (Δ) are computed to determine which chunks must be created, updated, or deleted. This delta-driven mechanism ensures the vector database remains continuously aligned with upstream sources while minimizing redundant processing. In addition to maintaining data integrity, the delta-based approach also provides significant cost and compute optimization. By reprocessing only the modified or newly added chunks, the system avoids unnecessary recomputation of embeddings and redundant I/O operations. These optimizations make the ETL pipeline both economically sustainable and operationally scalable for continuous ingestion from multiple data sources. After the delta computation, all relevant embeddings and metadata are upserted into the VectorDB. The database currently indexes more than 70,000 programs, 100,000 code chunks, and numerous other content types such as documentation, guidelines, and forum discussions. By maintaining a unified semantic index, the system enables high-performance, context-aware retrieval for the Code Search Assistant and related AI agents, while preserving full traceability and version integrity across all data sources.

5 Results and Evaluation

The analyzed time period was 18$^{\text{th}}$ of March 2024 - 19$^{\text{th}}$ of October 2025 for Clinical Analysis Assistant and 15$^{\text{th}}$ of September 2025 - 19$^{\text{th}}$ of October 2025 for Code Search. Weekends were excluded in the overall analysis due to the expected decline of usage during those days. Adoption statistics are presented as mean $\pm$ SD if not indicated otherwise.

5.1 Adoption and Engagement

For the Clinical Analysis Assistant, a total of 116,507 questions were recorded over the entire analysis period, with an average of 670 ± 338 weekly conversations and $1{,}399 \pm 682$ questions. For the Code Search Agent, the total count reached 490 questions, with 42 ± 14 weekly conversations and a mean of 92 ± 43 questions. We recorded **600 unique users** for the Clinical Analysis Assistant and **45 unique users** for Code Search. Approximately 92% of all questions during the period were directed to Ocean Assistant. Figure 11 demonstrates the engagement over time for Clinical Analysis Assistant and figure 12 for Code Search.

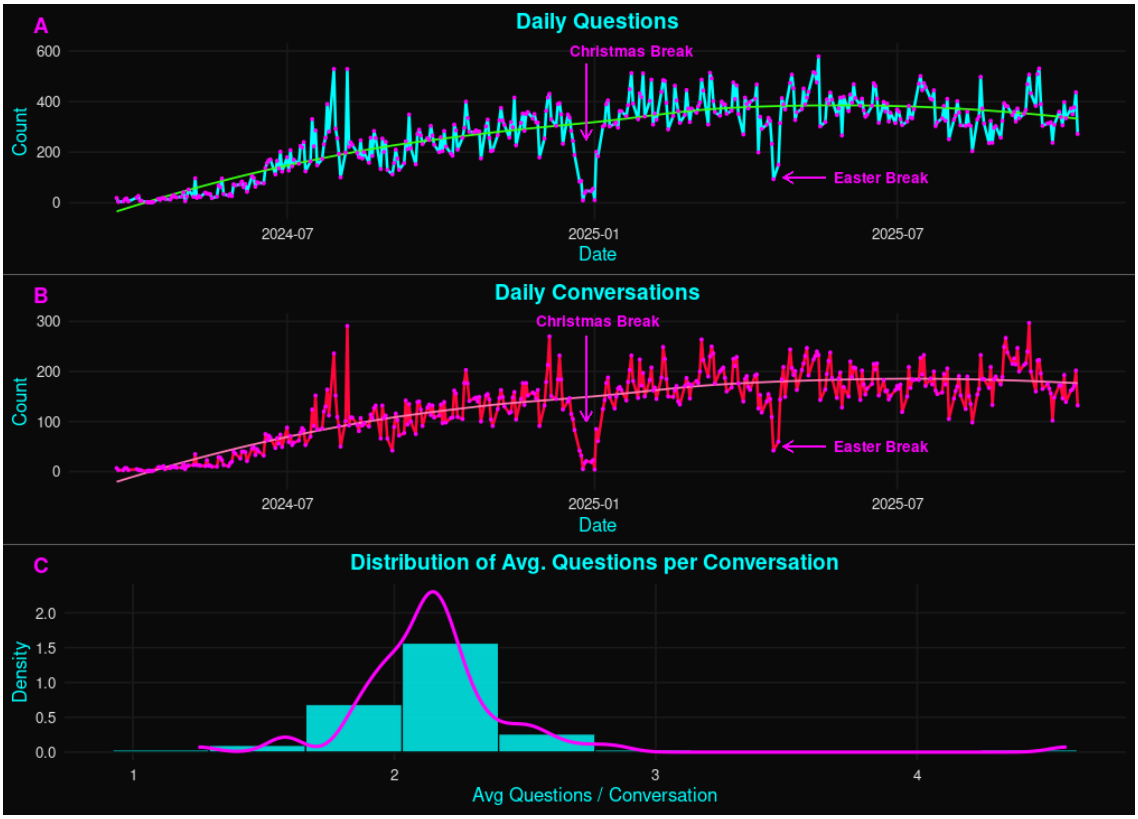

Figure 11: Engagement metrics for Clinical Analysis Assistant

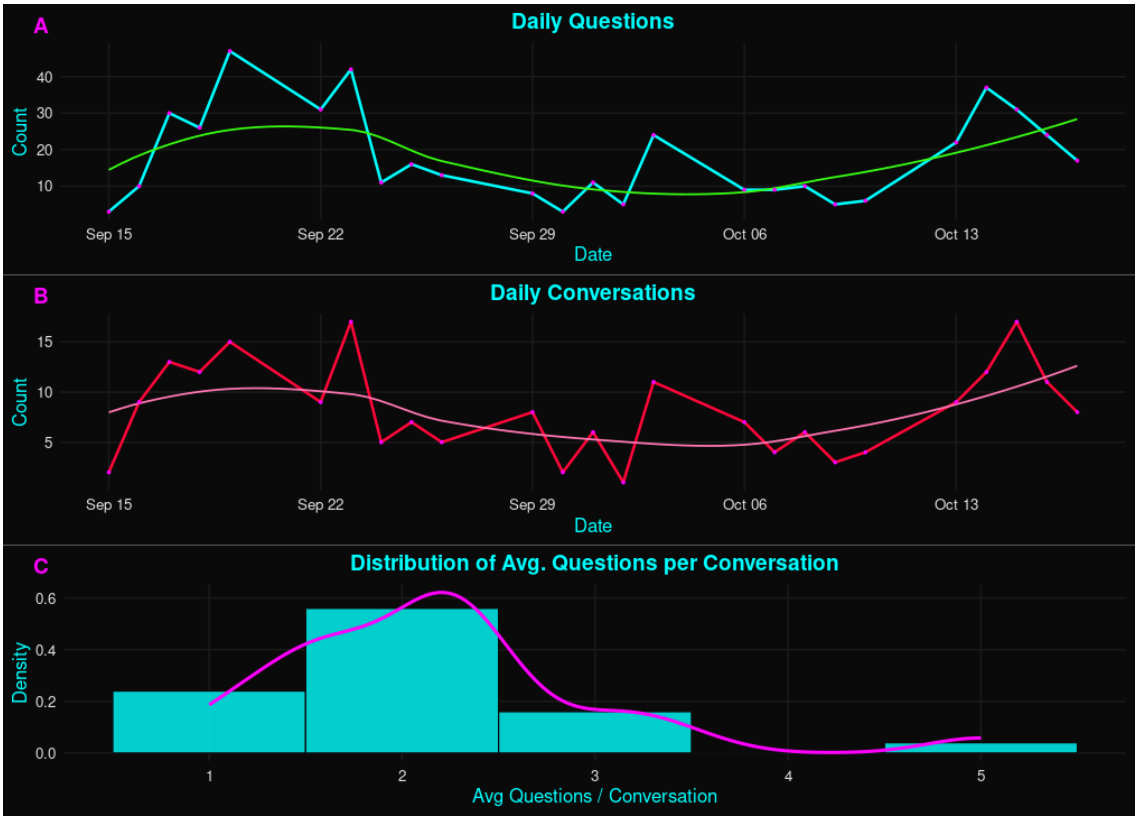

Figure 12: Engagement metrics for Code Search

The most frequently occurring words are illustrated in the bar chart presented in figure 13. While the Clinical Analysis Assistant serves as a comprehensive platform facilitating navigation and interaction within Roche's data analysis ecosystem, its vocabulary reflects a broader and more generalized usage. In contrast, the Code Search Agent exhibits a more specialized linguistic profile, emphasizing terminology related to code retrieval and programming across multiple languages.

Figure 13: Most frequent used words for prompting in Clinical Analysis Assistant (left) and Code Search tool (right)

5.2 Impact on Productivity

Preliminary observations suggest that the Code Search Agent can substantially enhance programmer productivity by streamlining information retrieval and reducing redundant work. Through rapid access to validated code snippets, documentation, and programming standards, users can resolve routine queries and coding challenges in a fraction of the time previously required. The integration of large-scale code search across historical repositories further amplifies these gains by enabling the discovery and reuse of existing, high-quality programs. This capability minimizes duplicate coding efforts, shortens development cycles, and promotes standardization across studies. Early internal estimates indicate potential time savings of 20–40% for common programming tasks, depending on project complexity and user experience. Collectively, these improvements contribute not only to greater operational efficiency but also to enhanced consistency and quality in clinical data analyses.

5.3 User Feedback

5.3.1 Clinical Analysis Assistant user Survey

To complement the usage analytics and performance indicators, a comprehensive survey was conducted at the end of Q1 2025 to assess both the quantitative and qualitative aspects of user experience with the CAA. The survey aimed to evaluate perceived usefulness, adoption frequency, and gather open-ended feedback for future improvement. A total of 174 responses were collected for the usefulness rating (figure 14, left) and 189 responses for the usage frequency (figure 14, right). The distribution of usefulness ratings (1–5 scale, where 5 represents most useful) yielded a mean rating of 3.39, indicating a generally positive perception of the Assistants' value in daily work activities. As shown in figure 14, most users rated the Assistants between 3 and 5, suggesting that it effectively supports their analytical or operational tasks.

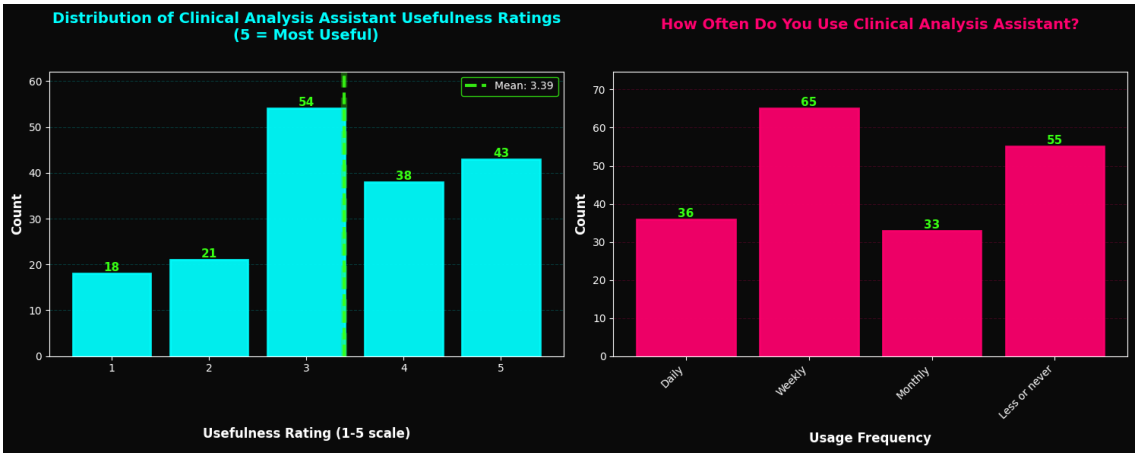

Figure 14: Survey results for CAA usefulness (left) and usage frequency (right)

Regarding usage patterns, the right panel of figure 14 illustrates that weekly engagement was the most common behavior, followed by less frequent or occasional use. A smaller but consistent group of respondents reported daily interaction, reflecting steady adoption among core users. In addition to quantitative metrics, qualitative feedback was also collected through open-ended responses. These narratives provided valuable insights into user motivations, challenges, and feature requests, offering a nuanced understanding of how the CAA integrates into existing workflows. This feedback has been compiled and shared separately to inform ongoing enhancements, focusing on usability improvements, clearer guidance, and prioritization of high-impact features.

5.3.2 Code Search Agent User Survey

User evaluations of the CSA demonstrated a consistently positive reception. Participants reported that the feature was highly effective in supporting their daily programming tasks, particularly in locating and reusing existing code on GitLab from prior studies. On average, the tool received a rating of 4 out of 5 points, reflecting strong overall satisfaction with its usability, relevance of retrieved results, and integration into existing workflows. Users emphasized that the CSA not only accelerated the search for validated programs but also increased their confidence in code quality and standardization. This feedback reinforces the tool's perceived value as a practical and efficiency-enhancing component of the Clinical Analysis Assistant ecosystem.

6 Limitations and Future Work

While the current implementation of the Code Search Agent (CSA) demonstrates strong potential in accelerating code retrieval and enhancing reuse within the OCEAN ecosystem, several limitations and development opportunities remain. First, the integration of CSA is currently limited to the conversational interface within the Clinical Analysis Assistant. Although this interface has proven effective for exploratory search, future work aims to achieve deep integration with development environments through MCP-enabled plugins for RStudio and Visual Studio Code. Such direct embedding would enable statistical programmers to access semantic code search and retrieval capabilities within their native coding environments, reducing context switching and providing zero-friction access to relevant, validated code during active development. Second, while the current design focuses primarily on code discovery, upcoming iterations will expand toward code generation in clinical contexts. By integrating the Code Search Assistant with the AiR (AI for R) Agent, the system will be able to leverage both retrieved historical programs and generative AI capabilities to produce fit-for-purpose R code aligned with clinical study requirements. This combined approach will bridge retrieval and synthesis, allowing

programmers to adapt existing validated code into new, study-specific analytical pipelines more efficiently. Finally, future development will explore multi-agent collaboration and study-aware intelligence. The goal is to create agents that understand the context of a given clinical study, including its metadata, protocol specifications, and derivation requirements, by dynamically pulling information from multiple data sources such as the 70,000 indexed programs, study file systems, and repositories containing statistical guidelines and programming specifications. Such context-aware agents could autonomously propose or even generate compliant code for standard derivations and TLGs, accelerating delivery while maintaining quality and traceability. These planned evolutions will move the CSA beyond code retrieval toward a more proactive and autonomous assistant, capable of supporting end-to-end clinical programming tasks in alignment with Roche's vision for intelligent, reproducible, and scalable analytics.

7 Conclusion

The Codes Search Agent successfully addresses fragmentation and redundancy in statistical programming by deploying a sophisticated RAG-based, multi-agent framework. With proven user adoption and significant query volume, the platform has clearly demonstrated its value in streamlining access to documentation and reusable code, thereby improving quality and consistency. Looking forward, our roadmap centers on moving beyond static retrieval to achieve true code intelligence through execution agents and deep IDE integration, ultimately aiming to deliver autonomous, verifiable, and highly streamlined clinical analytics.

Acknowledgments

We thank Haoyang Ju for his major contributions to the project, and acknowledge the contributions of Vincent Shen, Pawel Rucki, Kathrin Flunkert, Liviu Neagu, Adam Forys, Jian Dai, Mahdi About, and Helene Royo.

Contact Information

`mathieu.cayssol@roche.com`
`christoph.centner@roche.com`

References

[1] Chacon, S., & Straub, B. (2014). *Pro Git* (2nd ed.). Apress.

[2] GitLab Documentation: Roles and Permissions. (2025). Retrieved October 28, 2025, from `https://docs.gitlab.com/ee/user/permissions.html`.

[3] Gupta, S., Ranjan, R., & Singh, S. N. (2024). A Comprehensive Survey of Retrieval-Augmented Generation (RAG): Evolution, Current Landscape and Future Directions. *arXiv preprint arXiv:2410.12837*.

[4] Li, S., Stenzel, L., Eickhoff, C., & Bahrainian, S. A. (2025). Enhancing Retrieval-Augmented Generation: A Study of Best Practices. *arXiv preprint arXiv:2501.07391*.

[5] Anthropic. (2024, November 25). Introducing the Model Context Protocol (MCP). Retrieved from `https://www.anthropic.com/news/model-context-protocol`.

[6] Model Context Protocol. (n.d.). What is the Model Context Protocol (MCP)? Retrieved from `https://modelcontextprotocol.io/docs/getting-started/intro`.

[7] Schultz, R. (2025, May 6). How the Model Context Protocol (MCP) Future-Proofs AI Agent Tool-Calling for Scalable, Secure, Interoperable Workflows. *Caiyman.ai*. Retrieved from `https://www.caiyman.ai/blog/model-context-protocol-mcp-ai-agents-standard`.

[8] Chainlit Documentation: Overview. (2025). Retrieved October 28, 2025, from `https://docs.chainlit.io/`.

[9] Posit Connect: User Guide, Version 2025.09.1. (2025). Retrieved October 28, 2025, from `https://docs.posit.co/connect/`.

[10] FastAPI Documentation. (2025). Retrieved October 28, 2025, from `https://fastapi.tiangolo.com/`.

[11] Uvicorn: An ASGI Web Server for Python. (2025). Retrieved October 28, 2025, from `https://www.uvicorn.org/`.

[12] Quarto Documentation: Guide. (2025). Retrieved October 28, 2025, from `https://quarto.org/docs/guide/`.

[13] Husain, H., Wu, H., Gazit, T., Allamanis, M., & Brockschmidt, M. (2019). CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv preprint arXiv:1909.09436*.