# CFD Workflow Automation with Generative AI and Specialized Approaches for Storing and Querying Data

Dr. M. Bonner
*Siemens Digital Industries Software, Belgium*

## Abstract

Today's software tools are extensive and continually expanding, incorporating more and more features from the CFD area, which makes the tools powerful, but often leaves users lost in the innumerable settings. Consequently, users have to deal with customizable repetitive tasks operating on SW tools, invest significant amount of time to search for a proper API to write and to adjust macros, which is a backbone of any execution action. Additionally, due to the fact that the tool undergoes continuous updates and improvements, there are deprecated API, which have to be adjusted manually after a user switch to a new version. With the raise of Generative AI and Large Language Model (LLM) users expect an LLM-based assistant for code generation, enhancement and explanation operations and, surely, dealing with deprecated code. Indeed, the integration of Generative AI and LLM agents presents significant opportunities for workflow automation and enhancing efficiency. However, there are challenges in application of Generative AI to the area of code generation. First, APIs are kept confidential, which means that advertised LLMs are not familiar with the syntax. The second challenge is organising the existing data to feed AI to get sophisticated results. By employing specialized approaches for storing and querying source data, such as Vector-based DataBases (Vector DB) and Graph-based DataBases (Graph DB), overall precision of generated code can be improved. This article explores the innovative application of Generative AI combining it with advanced data retrieval mechanisms from Vector DBs and Graph DBs to automate the CFD workflow. It specifically focuses on the use case of employing LLM agents for macro generation, macro adjustments and macro updates within CAE Tools.

## 1. Introduction

As software tools evolve and expand their capabilities, they incorporate increasingly complex features that address CFD challenges, but simultaneously introduce a layer of complexity for users. This complexity arises from the vast number of settings and options that users must navigate, often leading to difficulties in managing and optimizing repetitive tasks. A significant portion of the user experience CAE software involves writing, adjusting, and maintaining macros, which serve as the backbone for executing repetitive actions within the tools. The continuous software updates and improvements lead to modifications in APIs, which result in the necessity to deal with deprecated APIs. In these scenarios macros must be manually adjusted each time a new version is released.

Recent advancements in Generative AI and LLMs offer innovative solutions to address these challenges by automating macro generation, macro adjustment, and macro maintenance tasks. Users now expect these intelligent agents to assist in the initial creation of code, refining it, explaining its functionality, and updating it to comply with newer software versions. The potential of Generative AI and LLMs is particularly significant in reducing the effort required for manual interventions, thereby optimizing the workflow and increasing productivity. The integration of such technologies promises a paradigm shift in how users interact with complex software tools.

Despite the promising capabilities of Generative AI and LLMs, several challenges delay their seamless integration into automated code generation for specific software environments. One significant challenge is the proprietary nature of commercial software APIs and existing examples, which limits the familiarity and training of widely advertised LLM models [1] with the specific syntax and functionality of these APIs. This confidentiality restricts the effectiveness of LLMs when generating or adjusting macros tailored for these software tools.

Another critical challenge lies in the organization and accessibility of existing data required to train the AI models to produce sophisticated and accurate results. Effective code generation by LLMs relies heavily on the quality and structure of the available data. Poorly organized data, which might include missing descriptions, missing examples, lack of documentation on deprecated APIs, storage in the format hard for the perception without additional context, can significantly compromise the precision of generated code, resulting in suboptimal or erroneous outputs. Thus, it is essential to employ advanced data retrieval mechanisms to enhance the data retrieval process.

This article addresses these challenges by exploring approaches to improve the integration of Generative AI in automating CFD workflows. Specifically, it investigates the combination of Generative AI with advanced data management techniques, such as Vector and Graph DBs, and the new trend of multi-agent systems to optimize the retrieval and utilization of source data. Vector DBs can store and retrieve data effectively based on similarity, while Graph DBs provide a structured and interconnected way to manage the relationship between data points. Together, these technologies can enhance the accuracy and relevance of the generated code.

The focus of this study is on the practical application of these combined technologies within CAE Tools. By employing LLM agents for macro generation and adjustments, the study aims to demonstrate the architecture of a research prototype for automated code generation, adjustment and update. The methodologies and results presented in this article offer valuable insights on practical solutions for effectively integrating Generative AI into CFD workflows.

## 2. Automation Process with Macros

Automation capabilities as available in many CAE software tools have significantly transformed user engagement with CFD simulations. This advancement offers a more streamlined and efficient methodology for addressing engineering challenges and is valued by customers [2]. The software allows extensive customization capabilities through its APIs, enabling users to tailor the simulation environment to their specific needs. The software's scripting capabilities allow users to automate everything from geometry import through to post-processing. Some users heavily rely on these automation tools (see **Error! Reference source not found.**), creating macros and scripts to handle recurring tasks and to reduce user errors. This massively enhances productivity and guarantees consistency across projects.

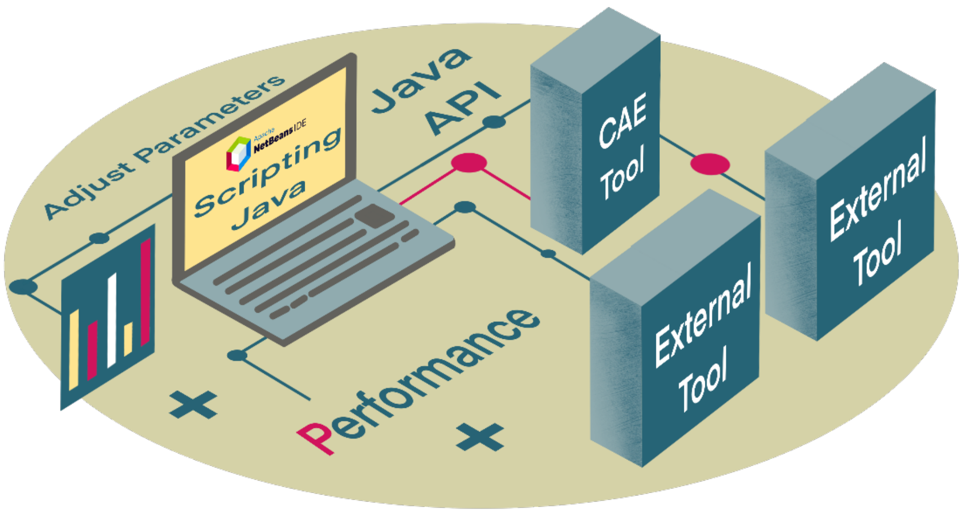


*Figure 1. Automation in CAE Tools.*

Any sophisticated software undergoes continuous updates and improvements. As new features are introduced and existing functionalities are enhanced, certain API methods and classes become outdated and are marked for future removal. One common challenge users face is dealing with such deprecated code.

For users, this means that their existing scripts and customizations may contain deprecated code that needs to be revised. The accumulation of such code is a common scenario in long-term projects, where initial scripts were written using older versions of the API. As the software evolves, these scripts must be updated to align with the current standards, ensuring optimal performance and avoiding potential issues.

Another frequently encountered challenge in the use of automation is the generation and, more commonly, the modification of macro code. Typically, a user initially records a macro and subsequently adapts it for a specific application. The documentation may contain elements from more than 10000 of classes and methods, which have often insufficient documentation and are lacking application examples for effective semantic search.

Despite of all inconveniences mentioned above, over time, utilizing macros in CAE software proves to be significantly more time-efficient compared to the repetitive process of manual clicking (Figure 2. Comparison of time spent: manual processes vs. Java MacrosFigure 2). However, one must consider the ongoing cost of maintaining these scripts, as adjustments are often necessary due to the deprecation of code in software updates. This could result in additional workload to ensure that scripts remain functional, potentially offsetting some of the initial time savings. Nevertheless, the efficiency gains in long-term project execution generally outweigh the periodic maintenance efforts required to keep the automation up-to-date.

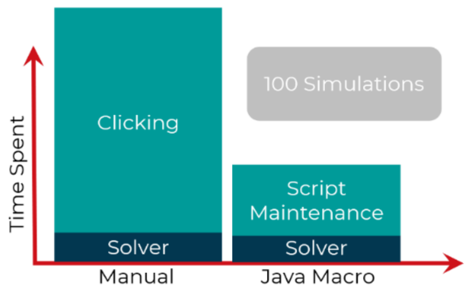


*Figure 2. Comparison of time spent: manual processes vs. Java Macros.*

## 3. Related Work

The application of LLMs to automate code generation has gathered substantial attention across various domains, such as code generation, code refinement, legacy code modernization, and their integration into automated workflows.

One significant area of research involves generating code by retrieving relevant documentation. [3] introduced 'DocPrompting,' a technique, where LLMs dynamically query code snippets from documentation to generate the code. This approach highlights the potential of LLMs to bridge the gap between LLMs trained on publicly available code and extensive, often complex, confidential API documentation, exemplifying how leveraging existing documentation can enhance the accuracy of generated code.

Furthermore, the inherent capability of LLMs to act as few-shot learners, as demonstrated by [4]. By leveraging minimal examples, LLMs can adapt to new tasks with surprisingly high proficiency, which is particularly beneficial for generating code in diverse domains without extensive retraining.

In addition to generating initial code snippets, recent studies have focused on refining and improving the generated code using AI agents. For instance, [5] proposed 'CRITIC,' a system where LLMs iteratively interact with tools to critique and enhance the generated code. Similarly, [6] introduced 'Cross-Refine,' an approach employing tandem learning to generate and iteratively refine natural language explanations, which can be analogously applied to code generation. Additionally, motivated by how humans refine their written text, [7] introduced Self-Refine, an approach for improving initial outputs from LLMs through iterative feedback and refinement. The main idea is to generate an initial output using an LLM; then, the same LLM provides feedback for its output and uses it to refine itself, iteratively.

Modernizing legacy code is another critical area, where LLMs demonstrate significant potential. [8] explored the challenges and opportunities in utilizing LLMs for updating and maintaining legacy codebases, focusing on the generation of updated documentation and code adjustments.

Overall, the related work in leveraging LLMs for code generation spans multiple innovative directions, from initial code generation based on document retrieval and few-shot learning to iterative refinement and legacy code modernization. In our work we are forced to incorporate several techniques in the form of multi-agent system.

## 4. Data Storage and Retrieval Mechanisms

LLMs have demonstrated remarkable capabilities in generating code, but their performance can be significantly enhanced by incorporating retrieval-augmented generation (RAG) [9], [10] and few-shot prompting techniques [4]. These approaches are particularly crucial, when LLMs are not trained on confidential or permanently extended data, as they enable the models to dynamically access relevant information from external sources. Vector and Graph DBs play an important role in this context, enabling efficient and fast retrieval of embeddings that capture semantic similarities within the data.

RAG techniques enhance LLM performance by supplementing their generative capabilities with dynamically fetched, contextually relevant information. For instance, when an LLM is not pre-trained on specific confidential datasets, RAG allows it to retrieve and integrate up-to-date, relevant data during the generation process. This dynamic retrieval ensures that the LLM's responses remain accurate and contextually appropriate, providing an effective workaround for scenarios, where pre-training on certain data is not feasible.

Few-shot learning enables LLMs to quickly adapt to new tasks and domains based on just a few examples, without the need for extensive task-specific training. Vector and Graph DBs optimize this process by efficiently storing and indexing examples, allowing the LLM to access relevant resources quickly. This capability is especially valuable, when the LLM lacks prior exposure to specific confidential data, ensuring context-aware learning.

Vector DBs are good in handling high-dimensional data and computing semantic similarities between data points. They convert data into embeddings, which are numerical representations of the data in high-dimensional space. This allows similarity search using metrics like cosine similarity or Euclidean distance. Their primary strength lies in managing and querying large volumes of unstructured data, making them ideal for tasks like document similarity search.

On the other hand, Graph DBs are designed to model and navigate relationships between data points through nodes and edges. Their strength lies in representing and querying relationships and structures within the data. This makes them useful for tasks like exploring connections within the API elements. Thoughtful extraction from the API documentation can also add additional connections, when mapping out the relationships between different entities for a more in-depth contextual comprehension. For example, when mapping out the relationships between elements for a more in-depth contextual comprehension more direct links could be retrieved.

Overall, the integration of Vector and Graph DBs is essential for LLMs to leverage RAG and few-shot learning techniques effectively, maintaining performance and scalability even, when trained data is limited or unavailable.

## 5. Multi-Agent System Architecture

Multi-Agent Systems (MAS) provide a framework for addressing complex tasks through the utilization of multiple interacting intelligent agents. These agents, which may comprise software autonomous units, which operate independently within a given environment. The fundamental characteristics of MAS include autonomy, social ability to interact with other agents, reactivity to environmental changes, and proactiveness in goal achievement.

The advantageous features of MAS include scalability, robustness, and flexibility. MAS system can be easily scaled by adding more agents, which enhances its capacity to handle complex tasks. Furthermore, the failure of a single agent does not affect the overall performance, ensuring robustness. The flexibility of MAS allows for dynamic adaptation to changing environments and tasks, making them suitable for systems that require collaborative autonomy to achieve collective goals. This paradigm is particularly effective for complex problem-solving while maintaining user-friendly interaction through natural language interfaces.

Here, we outline one way of how MAS implemented to enable the dynamic creation, modification, and updating of code through the coordination of several specialized agents, each fulfilling a distinct role. Figure 3 illustrates the organization and interaction among these agents.

Agent A is responsible for classifying user prompts by identifying specific use-cases. Based on the classification provided by Agent A, a sequence of subsequent agents is activated to execute their respective tasks. Agent B searches the Vector Database for API entries relevant to the user's query, while Agent E queries the Vector Database for macros relevant to the use-case. For the update of deprecated code, Agent F identifies modifications within the Vector Database Macro API affecting the specified APIs.

Agent C synthesizes code by utilizing the relevant APIs retrieved by Agent B and the examples provided by Agent E. This generated code is then refined by Agent D, which executes the code with a compiler, processes any resulting error messages, and interacts with the Graph Database API to address and correct issues. For the update of deprecated code, Agent G updates any uploaded macros based on the information gathered by Agent F.

Finally, Agent H compiles a summary of all changes, facilitating user review and enabling the generation of comprehensive user-guide documentation. Note that Agents B, C, E, G, and F employ Retrieval-Augmented Generation (RAG) and few-shot learning techniques.

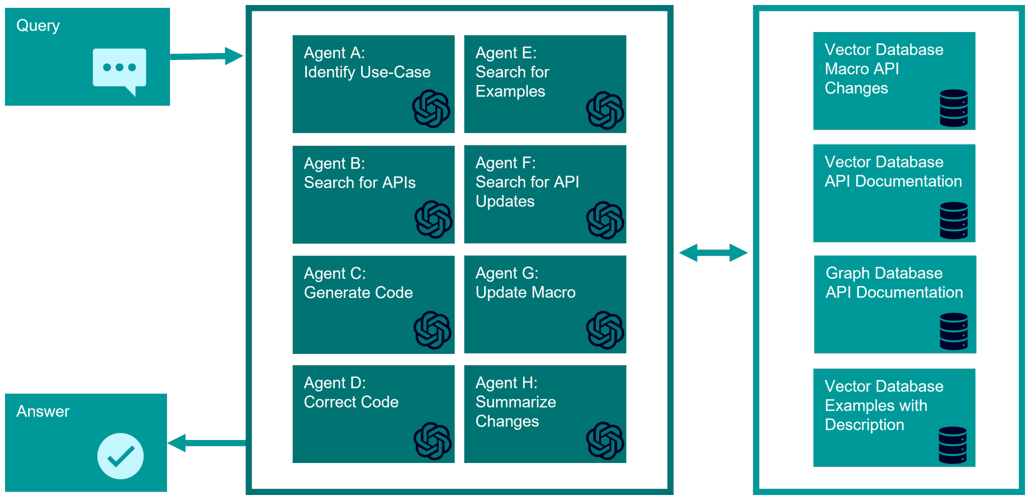
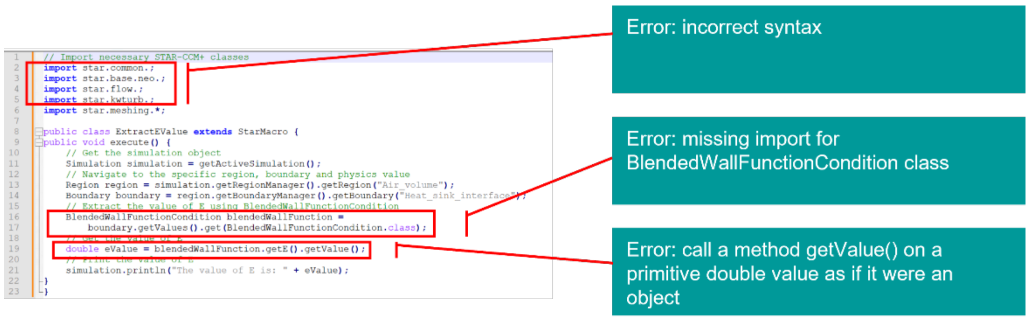

Query
Agent A: Identify Use-Case
Agent E: Search for Examples
Agent B: Search for APIs
Agent F: Search for API Updates
Agent C: Generate Code
Agent G: Update Macro
Agent D: Correct Code
Agent H: Summarize Changes
Answer
Vector Database Macro API Changes
Vector Database API Documentation
Graph Database API Documentation
Vector Database Examples with Description

*Figure 3. Multi-Agent System architecture for automated code generation, adjustment, and update.*

We elaborate on the functionality of the Agent D, which incorporates a self-correcting mechanism, on a concrete example. Drawing upon methodologies from both the CRITIC [5], Cross-Refine [6] and Self-Refine [7] frameworks, our MAS allows agents to interact with external tool, Java compiler, to iteratively verify and correct the outputs autonomously. This multi-step approach enhances the accuracy of generated Java code and responses, ensuring continuous improvement in performance.

Agent D validates and refines code by compiling and assessing error messages. The agent uses iterative refinement techniques to address the issues identified during these verification steps, similar to the principles used in CRITIC and Self-Refine. This iterative process enables agent to produce outputs of higher quality without the need for extensive retraining or additional supervision. The agents utilize few-shot prompting and RAG to enhance their performance, much like in CRITIC and Self-Refine, ensuring that they can adapt to various tasks and environments dynamically.



```java
// Import necessary STAR-CCM+ classes
import star.common.;
import star.base.neo.;
import star.flow.;
import star.kwturb.;
import star.meshing.*;

public class ExtractEValue extends StarMacro {
public void execute() {
    // Get the simulation object
    Simulation simulation = getActiveSimulation();
    // Navigate to the specific region, boundary and physics value
    Region region = simulation.getRegionManager().getRegion("Air_volume");
    Boundary boundary = region.getBoundaryManager().getBoundary("Heat_sink_interface");
    // Extract the value of E using BlendedWallFunctionCondition
    BlendedWallFunctionCondition blendedWallFunction =
        boundary.getValues().get(BlendedWallFunctionCondition.class);
    // Get the value of E
    double eValue = blendedWallFunction.getE().getValue();
    // Print the value of E
    simulation.println("The value of E is: " + eValue);
}
}
```

Error: incorrect syntax

Error: missing import for BlendedWallFunctionCondition class

Error: call a method getValue() on a primitive double value as if it were an object

*Figure 4. Java Macro with syntactical errors generated by an Agent using the data from Vector DB, later corrected by another Agent using the data from Graph DB.*

## 6. Conclusion

In conclusion, this article has explored the topic of how Generative AI combined with advanced data management techniques can be applied to automating the CFD workflow within CAE tools. Additionally, the study demonstrated the current state of the MAS framework, which purpose is automatic macro generation, adjustment, and maintenance of macros. It describes alternative methods how to deal with challenges related to the confidentiality of API data and the organization of proprietary data. Important to notice that the framework is still in a prototype phase and remains under investigation.

Future research directions might focus on creating an evaluation framework to estimate the precision of generated macros. A promising direction for future research might be fine-tuning of LLMs to improve their ability to generate code based on the gathered examples [11].

## 7. References

[1] „ChatGPT," OpenAI, 2023. [Online]. Available: https://chat.openai.com/chat.

[2] „Best CFD Simulation Software," Siemens, 2024. [Online]. Available: https://blogs.sw.siemens.com/simcenter/best-cfd-simulation-software/),.

[3] Shuyan Zhou et al., „DocPrompting: Generating Code by Retrieving the Docs," in *arXiv*, 2023.

[4] Tom Brown et al., „Language Models are Few-Shot Learners," in *arXiv*, 2020.

[5] Zhibin Gou et al., „CRITIC: Large Language Models Can Self-Correct with Tool-Interactive Critiquing," in *arXiv*, 2024.

[6] Qianli Wang et al., „Cross-Refine: Improving Natural Language Explanation Generation by Learning in Tandem," in *arXiv*, 2024.

[7] Aman Madaan et. al., „Self-Refine: Iterative Refinement with Self-Feedback," in *arXiv*, 2023.

[8] Colin Diggs et al., „Leveraging LLMs for Legacy Code Modernization: Challenges and Opportunities for LLM-Generated Documentation," in *arXiv*, 2024.

[9] Yunfan Gao et al., „Retrieval-Augmented Generation for Large Language Models: A Survey," in *arXiv*, 2024.

[10] „A Simcenter personal consultant realized with Generative AI," Siemens, 2024. [Online]. Available: https://blogs.sw.siemens.com/art-of-the-possible/a-simcenter-personal-consultant-realized-with-generative-ai/.

[11] Maria Bonner et al., „LLM-based Approach to Automatically Establish Traceability between Requirements and MBSE,“ *INCOSE International Symposium,* Bd. 34, pp. 2542-2560, 2024.