AI-Assisted REST API Design with Large Language Models

Jorge Martinez-Gil[1], Christoph Daxerer[1], Mario Winterer[1], Cornelia Neumüller[2], and Matthias Krump[2]

[1] Software Competence Center Hagenberg GmbH
Softwarepark 32a, 4232 Hagenberg, Austria
[2] Raiffeisen Software GmbH
Goethestraße 80, 4020 Linz, Austria
{jorge.martinez-gil, christoph.daxerer, mario.winterer}@scch.at
{cornelia.neumueller, matthias.krump}@r-software.at

Abstract. This work presents an AI-assisted environment that helps developers design REST APIs using Large Language Models. The system integrates retrieval-augmented generation with OpenAPI's interface to deliver a structured, dialogue-based design process. Developers can define endpoints, describe functionality, and generate specifications, while the assistant searches for similar existing APIs to promote reuse. The approach is evaluated using complex, domain-specific specifications from a real-world project to assess its ability to produce valid and complete API components. The solution has been applied to the needs of a large financial institution to create compliant APIs for operations processing, enabling rapid identification of existing internal endpoints and avoiding duplicated work. The aim is to demonstrate practical value in supporting developers through faster design cycles and more consistent specifications in industrial environments.

Keywords: REST API design, Retrieval-augmented generation, AI-assisted development

1 Introduction

Designing high-quality RESTful APIs is a critical yet challenging task in modern software engineering. The OpenAPI specification has emerged as a de facto standard for describing REST APIs in a machine- and human-readable format. Drafting these specifications by hand before implementation is common to ensure better-designed services. However, it is error-prone due to the verbose, rigid syntax of `YAML` and `JSON`.

Recent advances in Large Language Models (LLMs) have shown promise in assisting developers with coding tasks [10]. However, existing LLM-based assistants often struggle with less common formats, such as OpenAPI definitions, leading to incorrect suggestions [20]. Organizations with large API ecosystems also face reuse difficulties [1]. With hundreds of internal APIs, developers may

create duplicate services simply because they are unaware of existing ones. Some companies maintain internal APIs with limited documentation, leaving teams uncertain whether an API for a specific requirement already exists.

The work described in this paper was conducted within a large financial institution that manages an extensive collection of internal APIs to support operations and related services. Designing new APIs in this setting requires strict compliance with security and regulatory requirements. Duplicate functionality can arise when design teams cannot locate relevant endpoints. Rapid identification of existing internal APIs during the design phase reduces redundancy, lowers maintenance effort, and helps maintain consistent adherence to established standards.

To tackle these challenges, we propose an AI-assisted REST API Design framework that uses LLMs to accelerate the creation of new API specifications and promote the reuse of existing APIs. Building on recent advances in LLM-based agents for automation and knowledge-intensive tasks [25], our approach integrates a Retrieval-Augmented Generation (RAG) pipeline with OpenAPI-aware prompting and function-calling capabilities. The system can retrieve relevant API documentation from a knowledge base of API specifications and provide it to an LLM. The model then assists the user in drafting or refining an OpenAPI specification for a new service. The assistant produces structured output to ensure that the generated specifications are syntactically valid and conform to a predefined schema. This approach reduces the risk of creating API descriptions that fail to meet the required standards.

This work also presents an implementation that uses open-source tools and custom modules. A vector database indexes a collection of OpenAPI documents, enabling semantic search to locate relevant APIs. The LLM is configured with OpenAI's function-calling API to generate `JSON` snippets that match the OpenAPI schema; these snippets are then combined into a working specification. Automated validation is applied through linting rules and `JSON Schema` checks to catch errors early in the process. The main contributions of this work are:

- We review the related work in software engineering AI assistants, API documentation and specification tools, automated API reuse, and LLM-powered developer tools, positioning our approach within the state-of-the-art.
- We present a novel combination of RAG, semantic chunking, and structured output prompting for API design. Our framework uses vector-based semantic search over API specifications to inject relevant context. Furthermore, it uses function calls to produce OpenAPI-compliant outputs, with iterative refinement and validation in a loop.
- We demonstrate the implementation of an AI-assisted REST API design approach that combines semantic chunking, retrieval-augmented generation, and OpenAPI function-calling to both accelerate specification drafting and actively promote reuse in a real-world setting in the financial sector with an existing large-scale API ecosystem.
- We evaluate this setting with initial results using quantitative metrics and qualitative developer studies.

The remainder of this work is structured as follows: Section 2 presents the background and related work. Section 3 describes the system architecture and design. Section 4 details the implementation of our prototype. Section 5 discusses evaluation design, developer studies, productivity impact, and limitations. Section 6 concludes the paper.

2 Background and Related Work

This section reviews prior research on AI-driven developer assistants, automated API documentation and specification tools, and techniques for semantic API retrieval.

2.1 LLMs in Software Engineering

The application of LLMs to software engineering tasks has seen rapid growth [5, 8, 28]. For example, recent studies on the specialization of code models [17] suggest that augmenting pre-trained models can improve developer satisfaction [21] and effectiveness in software engineering tasks [8].

Beyond code completion [6], researchers and practitioners have explored LLMs in testing [2]. For example, LLMs can help generate formal specifications from natural language requirements [14], bridging the gap between high-level needs and implementation. Nonetheless, challenges such as domain-specific language understanding and hallucination persist [7].

Software engineering assistants such as GitHub Copilot have been studied extensively [4]. Recent research evaluated Copilot's code generation and found both benefits and limitations [18]. There are alternative approaches that mined Stack Overflow and GitHub Discussions, reporting that while Copilot often provides useful code and boosts developer satisfaction, integration into existing workflows can be complex [27]. Amazon's CodeWhisperer[3] offers similar AI capabilities. These tools excel at suggesting code in well-known languages but tend to perform poorly on less common tasks. A recent study quantified this gap by benchmarking Copilot on OpenAPI completion and found that it underperforms [20].

2.2 API Documentation and Specification Tools

Documenting and discovering APIs has been a long-standing challenge. Traditional API documentation tools, such as Swagger[4], focus on rendering human-readable documentation from machine-readable specifications [3]. At the same time, code-first frameworks generate OpenAPI specifications from source-code annotations [12]. These approaches help synchronize implementation and specification, but do not alleviate the initial design effort if starting from scratch.

[3] https://workshops.aws/categories/CodeWhisperer
[4] https://swagger.io/

To improve API quality, organizations often enforce guidelines for their API spec linting tools and perform static analysis on documents to flag guideline violations or inconsistent styles [22]. Some tools offer customizable rules that ensure internal best practices are adhered to. We integrate such linting into our workflow to provide feedback on the AI-generated specs.

Recent research has shown interest in automating the creation of API specifications. One notable work is SpeCrawler [13], which aims to generate OpenAPI specifications from unstructured API documentation by combining rule-based extraction and LLM generation. Prior rule-based methods struggled with the heterogeneity of API docs, but LLMs enable more flexible parsing and generalization. Then, a pipeline uses an LLM to interpret various sections of API documentation and compose a formal specification. Our work involves the interactive creation of new APIs guided by high-level intents rather than generating them from existing documentation.

2.3 API Search and Reuse

Discovering existing APIs that could be reused is a form of information retrieval that historically relied on keyword searches in API directories [15, 16]. API Harmony [23] was an early attempt to facilitate API reuse in large organizations. API Harmony built a graph of APIs and their relations (e.g., data models, functionality overlap) and provided a search interface to find APIs by functionality. The goal was to help developers identify, select, and consume internal or public APIs that fit their needs.

Our approach modernizes API search by using semantic vector search. Instead of manually maintained graphs or simple text matching, we use embeddings of API specifications to capture their functionality. This allows a developer to query in natural language and retrieve relevant API definitions even if exact keywords differ. RAG is a natural fit here since it has been widely applied to knowledge-intensive NLP tasks [24]. It involves fetching documents relevant to the query and feeding them into an LLM's answer generation context [11, 9].

Moreover, treating OpenAPI specs as knowledge documents allows RAG to be applied to the API reuse problem. This finds potentially useful APIs, and the retrieved content can be directly cited or integrated, grounding the assistant's suggestions in real API contracts. RAG has the added benefit of reducing hallucinations and providing traceability for the outputs [26], which is essential in a critical domain like API design, where factual accuracy is important.

2.4 Contribution over the State-of-the-Art

Our work intersects with the broader landscape of LLM-based developer tools. The reason is that LLMs are being integrated into IDEs and DevOps pipelines for various tasks [7], including generating commit messages, providing automated code review comments, and generating test cases from requirements [19]. The idea of constraining the model to produce a `JSON` object that matches a provided schema is to obtain output that is directly parsed and used by programs [2]. We

employ this approach by defining a function schema for OpenAPI components and instructing the LLM to invoke this function.

Early experiments required the model to follow a carefully crafted system prompt to generate `YAML` output. When calling functions, our assistant can generate structured data that we then render into valid `YAML`. Recent OpenAI updates even enable strict schema enforcement, ensuring the model's output conforms to the schema.

3 System Architecture

Figure 1 illustrates the architecture of our AI-assisted REST API design tool. Our implementation comprises two key modules: (1) the API Retrieval Module and (2) the API Design Assistant Module, supported by a (3) validation component and (4) two user interfaces.

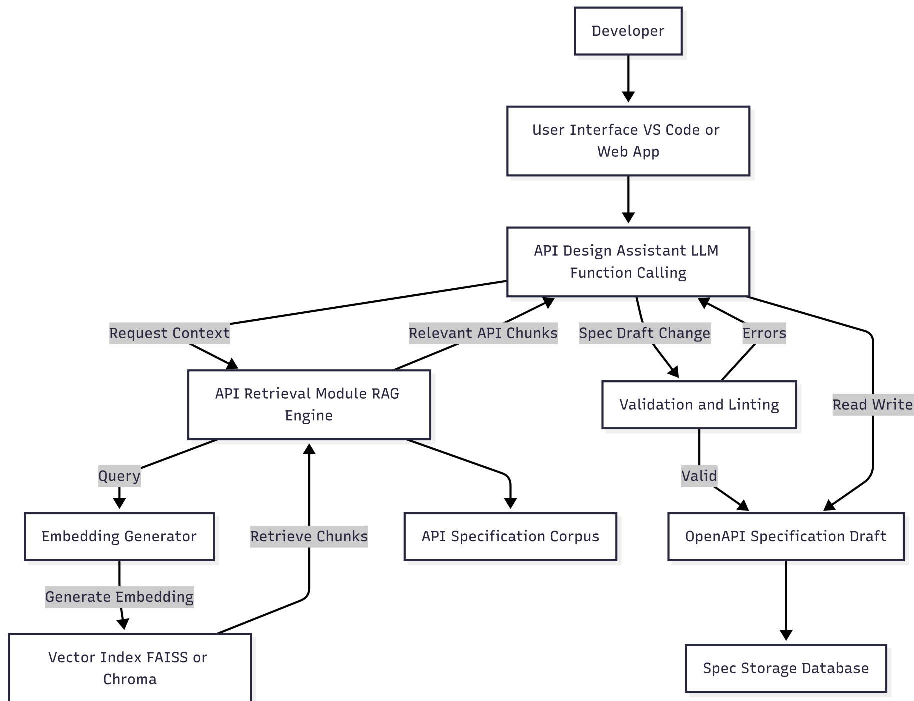

Fig. 1: System architecture of the AI-assisted REST API design tool

3.1 API Retrieval Module (RAG Engine)

Since a large number of REST APIs already exists and can be reused, this module is responsible for finding relevant existing APIs based on the user's query or design intent. It contains a Vector Index of API specifications: we internally collected an OpenAPI corpus. We embedded them into vector space using OpenAI's text-embedding model. Each document is represented as a high-dimensional vector such that similar APIs are nearby in that space. We use `FAISS` as the backend for efficient k-NN search over these vectors. Given a user's prompt, the query is embedded, and the nearest neighbor specs are retrieved.

A naive approach of chunking by fixed tokens proved suboptimal since many chunks contained only structural information (e.g., a list of parameter names) with little semantic signal, leading to irrelevant retrieval results. Instead, we implement a semantic chunking strategy: each API spec is parsed into logical units (the high-level description, each resource path with all its methods, shared components schema, etc.). For each unit, we generate a concise summary using an LLM (`GPT 4o`). This is a natural language description of what that part of the API does. These summaries are then embedded to populate the vector index.

Figure 2 shows how user queries are matched to relevant parts of `YAML` files using vector embeddings and similarity search. Retrieved chunks are then provided to an LLM assistant for accurate, context-aware responses.

This technique helped to increase retrieval precision. Even specs that lacked explicit documentation could be found because the LLM-generated summary captured the intent of the endpoints. The retrieved results are passed to the next module as contextual knowledge. We limit the number of retrieved documents to avoid overloading the LLM context window; typically, the top-k relevant API specs (or specific endpoints from them) are included.

3.2 API Design Assistant Module

The interactive assistant is powered by an LLM (`GPT-4o`). It operates in a conversational loop with the user, similar to ChatGPT but specialized for API design. We defined a dedicated system prompt to establish the assistant's role:

> *You are an API design assistant helping the user create or modify an OpenAPI specification. You can access documentation snippets of potentially relevant APIs and an editable draft of the user's API spec.*

The assistant is given two forms of context at each turn: (a) the retrieved API snippets and (b) the current draft specification (if one exists). The draft spec context is included by serializing the in-progress `YAML` into the prompt or using function arguments. Hence, the assistant knows the API's context and what is already defined. The assistant decides how to act based on the user's request. We implemented two modes of assistant responses:

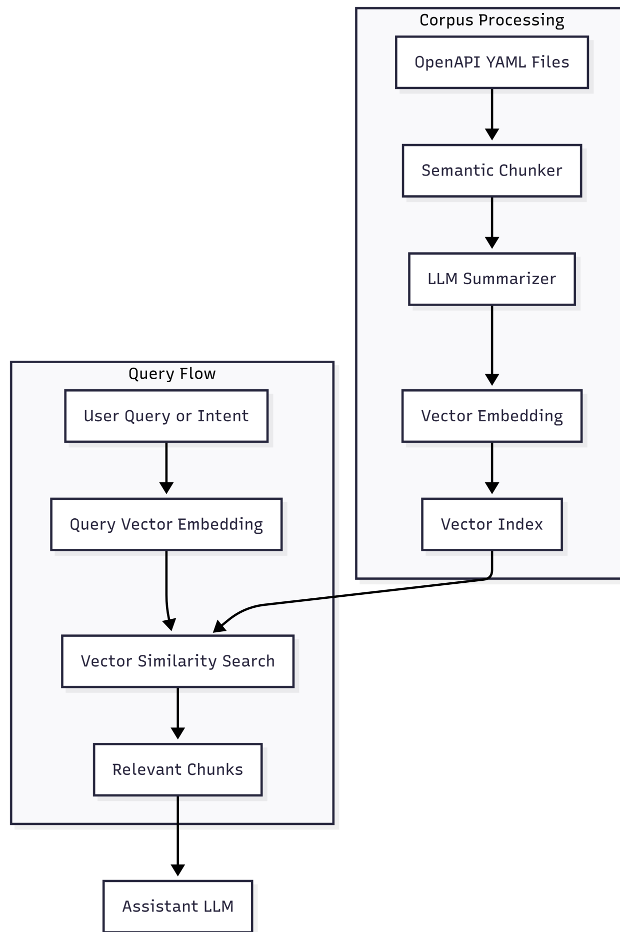

Fig. 2: RAG workflow adopted in our framework

1. **Natural Language Guidance**: In this mode, the assistant responds with an explanation or suggestion, potentially referencing a retrieved API as inspiration. For instance: *It looks like `ImageService` API has a similar upload endpoint. You might consider a `POST /images` endpoint that takes an image file as form data and returns a URL.*
2. **Spec Generation**: In other cases, the assistant outputs OpenAPI specification snippets (`YAML` or `JSON`) to be merged. This is enabled by function calling[5] that allows models to interface with external systems (tools). When the assistant decides to propose a concrete change, it calls the function with structured arguments.

[5] OpenAI function calling: *https://platform.openai.com/docs/guides/function-calling*

The chat completion API returns a structured `JSON` object describing the new endpoint. Our system intercepts this response and transforms it into the appropriate internal format. The function output is also shown to the user for transparency. Using function calling with a strict schema guarantees that the model's output is syntactically valid and complete for endpoint generation.

3.3 Iterative Validation

After the assistant proposes changes, the modified OpenAPI spec is run through a validation pipeline. We automatically check JSON schema validity against the OpenAPI 3.1 schema. We also run a linter with a company-specific ruleset. Any errors or warnings are fed back into the conversation.

For example, if the assistant generated a snippet without a response for `HTTP 400`, the linter issues a warning that a default error response is missing. The assistant then suggests adding it, and the improved snippet can be regenerated (the assistant is instructed to explain the issue and correct the spec). This validation loop ensures that each iteration produces a syntactically correct spec that conforms to best practices. The assistant is instructed to apologize and correct the spec if validation finds an issue (this mimics how a human would respond to failing tests).

3.4 User Interface Integration

We integrated this system into two front-ends for evaluation. First, a **Visual Studio Code extension** was developed. It provides a chat sidebar where the developer can converse with the API Design Assistant. The extension syncs the OpenAPI file with the assistant: the user can approve suggestions to apply them to the file, and the assistant can read the current file to inform its following answers. This tight IDE integration aims for a seamless workflow.

The second front-end is a **web application** built with React and Flask backend. This was used to pilot the tool in a team setting. In the web UI, multiple users can collaboratively brainstorm an API design with the assistant. The Flask backend orchestrates calls to the retrieval module and OpenAI API, and the React UI displays the current spec (rendered visually using a Swagger-UI component) alongside the chat interface. This setup is helpful for demonstrations, as it does not require installing an IDE plugin and works in any browser.

4 Implementation Details and Lessons Learned

Our implementation required effort and custom development of (1) an indexed OpenAPI corpus for the RAG, (2) prompt engineering with tailored templates, and (3) a web-based multi-user interface to the AI-based assistant.

4.1 OpenAPI Corpus and Indexing

We collected an initial set of `YAML` files from public repositories, excluding trivial or duplicate entries. We cleaned this dataset by filtering out invalid files and those that were not in the OpenAPI format. To provide high-quality data for retrieval, we linted all specs with OpenAPI ruleset; about 15% were discarded due to significant errors (unparseable or too incomplete). The final corpus spans diverse domains, providing a basis for reuse suggestions. We then constructed embeddings for these documents using OpenAI's `text-embedding-ada-002` model.

After that, we applied the semantic chunking strategy described earlier rather than embedding all specifications (which often exceed token limits). We developed a script to iterate over each spec and break it down into the following components: the info section, each tag group of endpoints, each path (if needed), and each schema component. For each chunk, we prompted `GPT-4o` with a specific instruction to generate a summary documentation.

These summaries and key metadata (such as the API file and section from which they originate) were then embedded in vectors. We used `FAISS` for fast similarity search due to its balance of speed and recall, which is particularly well-suited to our dataset size.

We also experimented with the Chroma vector DB, which wraps `FAISS` and provides persistence and embedding management. Both yielded similar performance, but we opted for `FAISS` for better control over indexing parameters.

4.2 Assistant Prompting and Tools

The design of the system prompt for the assistant went through many iterations. Our latest version that yielded the best results is:

> *You are an expert API designer AI. You will be given: 1. the current API draft or an empty draft, 2. user requests or questions, and 3. snippets from other APIs (as inspiration or reference). Your job is to help the user create a correct OpenAPI specification. You might explain the reasoning, but when appropriate, you should directly provide the spec changes (in YAML) required to implement the request. You have tools to output structured changes (endpoints, schemas, etc.). Do not produce code unrelated to the API spec. Ensure all YAML is valid and complete.*

In addition, we also provided several examples in the system message (few-shot learning) of how the assistant should behave. One example shows the assistant adding a new endpoint, and another shows it suggesting using an existing API instead.

We also maintain the state between user interactions. When the user accepts a suggestion, the spec state updates, and the latest specification must be provided in the conversation if the user requests a follow-up (e.g., if the user requests *Now add an endpoint to delete an image*). Therefore, our implementation stores the current spec in memory and extracts the relevant parts to include in the prompt.

The prompt template uses placeholder tags like `@paths.*.summary` to pull in summaries of all existing paths from the draft, so the assistant is aware of them. To achieve this, we developed a small templating system to map these placeholders to `JSONPath` queries on the spec.

4.3 Integration with Flask and React

The `Flask` backend wraps the retrieval and LLM calls behind a REST API. Key endpoints include: `/query` (for a user question, which triggers retrieval and LLM processing and returns the assistant's answer along with any spec changes), `/validate` (runs the linter and schema validator on the current spec), and `/spec` (to fetch or update the working specification). We also used `Flask-SocketIO` to push real-time updates to the client.

The `React` frontend consists of a chat panel and a preview panel. For the preview, we integrated the *Redoc* and *Swagger-UI* components, which render the OpenAPI document in a visually structured format. This allows the users to view the evolving specification as documentation—users can even try out endpoints if a mock server is configured.

A key implementation detail was supporting multi-user sessions: users were identified by a session `ID`, and we maintained separate spec states per session.

Figure 3 shows the workflow for our assistant to help developers design and validate OpenAPI specifications. User queries are processed through an RAG engine, and proposed spec changes are validated before being saved to the specification database.

5 Evaluation and Discussion

Evaluating an AI-assisted design tool involves assessing both the quality of the artifacts produced and the experience of developers who use it. Our first results from pilot testing in a lab setting indicate that the assistant can reduce the number of validation iterations by about 30–40%, with an expected time saving of 25–35% in producing a complete specification. For the involved company, these gains can help to considerably lower development costs and shorten deployment cycles. However, subsequent evaluation within the company is required to confirm these estimated gains in real-world scenarios.

5.1 Pilot Testing

We are currently performing the first pilot tests in a lab setting in close cooperation with the industry partner. The retrieval module is examined in isolation using a benchmark of query scenarios based on realistic reuse cases, as described in [11]. Each query has a ground truth set of relevant APIs identified by the authors. Standard metrics, such as recall, mean reciprocal rank, and semantic relevance scores from an independent reviewer, are applied.

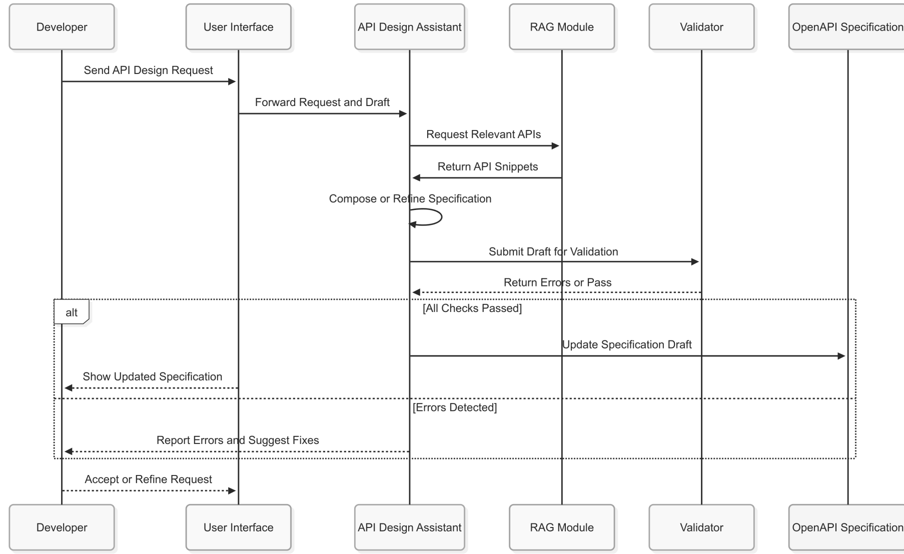

Fig. 3: Architecture for integrating RAG and automated specification validation

The quality of OpenAPI specifications created with the assistant's support is assessed through correctness and completeness. Correctness is determined by validation checks, recording the number of iterations required for a specification to pass all tests. Completeness is measured against predefined requirements for each task. A set of synthetic API design tasks is used, with reference specifications prepared manually. The assistant processes each task, and the resulting specifications are compared with the references using precision and recall metrics for required endpoints and error case coverage. Text similarity and schema similarity measures are applied to examine how closely the assistant's output matches the reference designs.

Currently, the pilot testing phase is run in a lab setting only, as it is out of scope for the company to use third-party-hosted LLMs for sensitive corporate API data. If our assistant uses proprietary APIs, sending their specs or descriptions to an external API could pose risks. While we use OpenAI's API, one could substitute it with a locally hosted LLM for enhanced privacy. The trade-off would be the cost of running and possibly lower quality for some tasks.

Another aspect is the correctness of the AI's knowledge. Suggesting an insecure design or referencing a deprecated API without warning is considered a blocking issue. We thus plan to incorporate some safety checks. For instance, our corpus could be annotated with deprecation info, and the assistant could be prompted to favor newer APIs. There is also the concern of overdependence: Developers should still be trained in API design fundamentals and not just accept output that they do not fully understand to avoid the propagation of errors.

5.2 User Study Design

As a next step, we plan to migrate the AI-assisted REST API design tool to a company-hosted environment, including an on-premise LLM solution. Based on this setting, we will perform a user study with developers and domain experts in a real-world project context. The design for this study is outlined below.

Developer Experience Evaluation. A key part of the planned evaluation is a user study with developers. It will assess the tool's impact on productivity and satisfaction. Participants will complete an API design task in a controlled setting. A within-subjects design will be used, in which each participant completes one task with the AI assistant and another without it, with the order randomized to reduce learning effects. The tasks will be of similar scope. One scenario will be based on the needs of a large financial institution that develops secure, compliant APIs for operational processing.

The evaluation will record the time taken to complete each task and the quality of the final specification, using either expert review or a checklist of required elements. A usability survey will be given, including questions such as: *Did the assistant help you find useful existing APIs? Did it save time when writing the specifications? How was the quality of its suggestions? Did you trust the suggestions or verify them before use?* Interaction data will also be captured, including the number of prompts, the number of iterations required to correct errors, and other relevant measures.

With the assistant, users are expected to complete tasks more quickly and produce specifications that are at least as complete as those created manually, with greater confidence in meeting format and compliance requirements. The study will also examine potential adverse effects, such as cases in which the assistant produces convincing but inaccurate suggestions. These qualitative aspects will be gathered through interviews conducted after the tasks.

Impact on Developer Productivity. An AI assistant could change how developers approach API design. Instead of navigating through internal wikis or asking, *Does an API for X already exist?* They can get an answer in seconds using our RAG approach. This lowers the barrier to reuse, ultimately reducing duplication across an organization's services.

One interesting side effect is the potential for such an assistant to serve as organizational memory: corporate APIs that have historically been underutilized may gain more adoption if the AI consistently surfaces them when relevant.

From a spec writing perspective, the assistant encourages best practices. However, there is a risk of over-reliance: developers might not learn the ins and outs of OpenAPI if the AI always does it for them, which could be problematic when the AI is unavailable or when debugging an issue. This parallels concerns in code autocomplete tools, where developers may lose some lower-level coding skills. The trade-off is worth it if it frees up time for higher-level design thinking.

Quality of AI Suggestions. Our approach prevents hallucinations by grounding the model with real data and using function constraints. However, subtle issues can still arise. For instance, the assistant might suggest an existing, related API that is not a perfect fit, potentially leading a team down a path of trying to adapt something that is not worth adapting. Final decisions on reuse remain with the developer. For this reason, we see the assistant as augmenting, not replacing, the human in the loop.

In our experience, the best outcomes came when the user operated the framework as if it were a junior assistant, accepting its good suggestions and redirecting it when it was off track. If users accept everything, they might end up with an API that technically works but is not conceptually cohesive. Thus, tooling should encourage a critical review of the AI's output.

Workflow Integration. For real-world integration, this assistant must seamlessly integrate with existing tools and systems. Developers typically use a range of tools, including an IDE, a source control system, and CI/CD pipelines.

Our framework is a step in this direction, but further integration points could be explored for enhanced functionality. For example, a Git pre-commit hook could run the assistant in validation mode. Whenever an OpenAPI file changes, it could automatically suggest improvements or catch mistakes.

Similarly, within API management platforms, an AI assistant can help populate documentation or create initial specifications based on a few inputs. We also consider the possibility of an AI-assisted review process for APIs. When a team submits an API design for review, an AI agent could act as a reviewer, flagging potential issues or asking clarifying questions. Our current assistant already has some of this capability, as it is familiar with best practices.

5.3 Limitations

Our current work focuses on REST APIs using OpenAPI. A natural extension is to consider other interface description language for event-driven systems. The techniques would largely transfer, though the retrieval content and the output schema would differ.

Another limitation is that our framework does not directly integrate with actual code implementation. After designing the API specification, developers implement the service throughout the whole development lifecycle. However, we plan to address this in future work by incorporating tooling that connects the specification to code generation and validation processes.

A future system where the same assistant, having helped write the spec, can also generate skeleton code and possibly even some application logic. There is also room to use the source code when designing an API for an existing system, e.g., parsing code to extract potential API endpoints. Our documentation refinement sub-project attempted something related by linking source code to existing specifications, ensuring documentation and specification completeness. This also enables automated checks that keep the spec, code, and related artifacts aligned during design and maintenance.

6 Conclusions and Future Work

We have presented a novel framework for AI-assisted REST API design using LLMs. Our approach combines the retrieval of existing API knowledge with guided specification generation, addressing two key challenges: allowing developers reuse existing knowledge before reinventing it and easing the burden of writing correct OpenAPI specifications. The rationale for integrating techniques like RAG is to facilitate OpenAPI function calls for structured output and rigorous schema validation. Pilot testing indicates possible gains in speed and quality, with fewer risks of non-compliant or redundant endpoints. The method can be adapted to other sectors with large API ecosystems, offering a practical means to deliver APIs more efficiently in enterprise software engineering. It can even be integrated into enterprise API governance workflows, offering immediate validation and reuse recommendations within CI/CD pipelines.

Our approach helps improve software development practices by assisting teams in drafting consistent API specifications early in the development process. It works with existing developer environments, including editors and validation steps, lowering the chances of missing elements or duplicating prior work. Automated checks and reuse suggestions allow faster progress without reducing quality. This aligns with standard DevOps methods that favor short feedback cycles and repeatable steps. The system also helps teams stay coordinated during API design and implementation. Its structure and components are described in detail, along with related work on software engineering assistants, specification tools, reuse approaches, and the use of LLMs in development.

This work also opens up several lines for future exploration. One is the adaptation of software design to other domains. If LLMs can assist with API specs, they can also help with database schema design or the synthesis of configuration files, especially when there are extensive catalogs of existing artifacts to learn from. Another possible line is deeper IDE integration and even more interactive experiences. From a research perspective, our study contributes to understanding how AI can be effectively incorporated into software design workflows. This area will become increasingly important as AI capabilities continue to advance.

Acknowledgments

The research reported in this paper has been funded by the Federal Ministry for Innovation, Mobility and Infrastructure (BMIMI), the Federal Ministry for Economy, Energy and Tourism (BMWET), and the State of Upper Austria in the frame of the SCCH competence center INTEGRATE (FFG grant no. 892418) in the COMET - Competence Centers for Excellent Technologies Programme managed by Austrian Research Promotion Agency FFG.

References

1. Sebastien Andreo and Jan Bosch. Api management challenges in ecosystems. In *International Conference on Software Business*, pages 86–93. Springer, 2019.

2. Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
3. Hanyang Cao, Jean-Rémy Falleri, and Xavier Blanc. Automated generation of rest api specification from plain html documentation. In *Service-Oriented Computing: 15th International Conference, ICSOC 2017, Malaga, Spain, November 13–16, 2017, Proceedings*, pages 453–461. Springer, 2017.
4. Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
5. Usman Khan Durrani, Mustafa Akpinar, Muhammed Fatih Adak, Abdullah Talha Kabakus, Muhammed Maruf Ozturk, and Mohammed Saleh. A decade of progress: A systematic literature review on the integration of ai in software engineering phases and activities (2013-2023). *IEEE Access*, 2024.
6. Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.
7. Muhammad Hamza, Dominik Siemon, Muhammad Azeem Akbar, and Tahsinur Rahman. Human-ai collaboration in software engineering: Lessons learned from a hands-on workshop. In *Proceedings of the 7th ACM/IEEE International Workshop on Software-intensive Business*, pages 7–14, 2024.
8. Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79, 2024.
9. Gautier Izacard and Edouard Grave. Leveraging passage retrieval with generative models for open domain question answering. *arXiv preprint arXiv:2007.01282*, 2020.
10. Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.
11. Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick SH Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In *EMNLP (1)*, pages 6769–6781, 2020.
12. István Koren and Ralf Klamma. The exploitation of openapi documentation for the generation of web frontends. In *Companion proceedings of the the web conference 2018*, pages 781–787, 2018.
13. Koren Lazar, Matan Vetzler, Guy Uziel, David Boaz, Esther Goldbraich, David Amid, and Ateret Anaby-Tavor. Specrawler: Generating openapi specifications from api documentation using large language models. *arXiv preprint arXiv:2402.11625*, 2024.
14. Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*, 2023.
15. Chun-Yang Ling, Yan-Zhen Zou, Ze-Qi Lin, and Bing Xie. Graph embedding based api graph search and recommendation. *Journal of Computer Science and Technology*, 34:993–1006, 2019.

16. Jorge Martinez-Gil. Source code clone detection using unsupervised similarity measures. In Peter Bludau, Rudolf Ramler, Dietmar Winkler, and Johannes Bergsmann, editors, *Software Quality as a Foundation for Security - 16th International Conference on Software Quality, SWQD 2024, Vienna, Austria, April 23-25, 2024, Proceedings*, volume 505 of *Lecture Notes in Business Information Processing*, pages 21–37. Springer, 2024.
17. Jorge Martinez-Gil, Alejandra Lorena Paoletti, Gábor Rácz, Attila Sali, and Klaus-Dieter Schewe. Accurate and efficient profile matching in knowledge bases. *Data Knowl. Eng.*, 117:195–215, 2018.
18. Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. On the robustness of code generation techniques: An empirical study on github copilot. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 2149–2160. IEEE, 2023.
19. Deep Mehta, Kartik Rawool, Subodh Gujar, and Bowen Xu. Automated devops pipeline generation for code repositories using large language models. *arXiv preprint arXiv:2312.13225*, 2023.
20. Bohdan Petryshyn and Mantas Lukosevicius. Optimizing large language models for openapi code completion. *CoRR*, abs/2405.15729, 2024.
21. Christoph Treude and Marco A Gerosa. How developers interact with ai: A taxonomy of human-ai collaboration in software engineering. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, pages 236–240. IEEE, 2025.
22. Kuldeep Vayadande, Kuhu Mukhopadhyay, Vaishnavi Chaudhari, Shreyas Manwadkar, Tanmay Mutalik, and Ishan Gawali. Let us lint: a tool for code formatting and code enhancing. In *2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–8. IEEE, 2023.
23. Erik Wittern, Vinod Muthusamy, Jim Laredo, Maja Vukovic, Aleksander Slominski, Shriram Rajagopalan, Hani Jamjoom, and Arjun Natarajan. API harmony: Graph-based search and selection of apis in the cloud. *IBM J. Res. Dev.*, 60(2-3), 2016.
24. Shangyu Wu, Ying Xiong, Yufei Cui, Haolun Wu, Can Chen, Ye Yuan, Lianming Huang, Xue Liu, Tei-Wei Kuo, Nan Guan, et al. Retrieval-augmented generation for natural language processing: A survey. *arXiv preprint arXiv:2407.13193*, 2024.
25. Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101, 2025.
26. Hong Qing Yu and Frank McQuade. Rag-kg-il: A multi-agent hybrid framework for reducing hallucinations and enhancing llm reasoning through rag and incremental knowledge graph learning integration. *arXiv preprint arXiv:2503.13514*, 2025.
27. Beiqi Zhang, Peng Liang, Xiyu Zhou, Aakash Ahmad, and Muhammad Waseem. Practices and challenges of using github copilot: An empirical study. In Shi-Kuo Chang, editor, *The 35th International Conference on Software Engineering and Knowledge Engineering, SEKE 2023, KSIR Virtual Conference Center, USA, July 1-10, 2023*, pages 124–129. KSI Research Inc., 2023.
28. Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372*, 2023.