Detecting Violations of NST Theorems with LLMs through Retrieval-Augmented Generation

Danilo Peeters
Research & Development
NSX bv
Niel, Belgium
danilo.peeters@nsx.normalizedsystems.org

Geert Haerens
Management Information Systems
University of Antwerp
Antwerp, Belgium
geert.haerens@uantwerp.be

Herwig Mannaert
Management Information Systems
University of Antwerp
Antwerp, Belgium
herwig.mannaert@uantwerp.be

***Abstract*—While it is widely accepted that AI tools have the ability to significantly increase the productivity of software development, this contribution focuses on its use to improve software evolvability. More specifically, an artifact is presented that leverages *Large Language Models (LLMs)* through *Retrieval-Augmented Generation (RAG)* to automatically detect violations of *Normalized Systems Theory (NST)* evolvability theorems in source code. This artifact is based on a RAG architecture, where a dedicated knowledge base is created with practical examples of NST violations, and an *Abstract Syntax Tree (AST)* representation is used to convey code structure into natural language. An experimental setup to validate the artifact is described, allowing participants to engage in three experiments through a web application, and the empirical results are presented. Though the limitations of this validation are acknowledged, the results are deemed positive, and some options for future research are identified.**

***Keywords*—Large Language Models, Retrieval-Augmented Generation, Normalized Systems Theory, Software Evolvability.**

I. Introduction

Following the sweeping success of generative AI tools in the early 2020s, AI-based code generation has become widespread as well. Though the use of machine learning models trained to automatically create or complete source code may be new, the practice of code generation is not. Indeed, the automated generation of source code, often referred to as automatic programming, has been pursued and applied for decades [1]. AI-based code generation can therefore be considered as another technology or methodology toward automatic programming, following in the footsteps of techniques like *Generative programming*, *Model-Driven Engineering (MDE)*, *Model-Driven Architecture (MDA)*, *Low-Code Development Platforms (LCDP)*, and *No-Code Development Platforms (NCDP)*.

While it is obvious that AI-based code generation can increase the programming productivity, neither the goals nor the possible value of automatic programming techniques were limited to the increase in productivity. Other potential benefits pursued by automatic code generation, besides the elimination of human errors, include improvements in traceability, repeatability, architectural structure, and evolvability. It seems clear that the positive contribution of AI-based code generation toward these other goals, certainly regarding error elimination and repeatability, is not trivial.

However, AI-based code generation is not the only way to take advantage of AI tools during the software development process. Though many use cases can be identified, for instance related to the creation of documentation and requirements, this paper builds upon our previous work and focuses on software evolvability. More specifically, this paper, largely based on the master's thesis of the first author [2], investigates the use of generative AI tools to automatically evaluate the adherence of software source code to the evolvability theorems as proposed by *NST* [3].

The remainder of this paper is structured as follows. In Section II, we discuss the notion and measurement of software evolvability, and the NST approach based on evolvability theorems. Section III explains the use of *RAG* to supply *LLMs* with additional context. In Section IV, we describe the artifact that has been constructed to detect violations of the NST theorems. The preliminary results are presented and discussed in Section V. Finally, we present some conclusions and discuss future work in Section VI.

II. Software Evolvability

A. Defining and Measuring Evolvability

Breivold et al. describe characteristics that an evolvable system should have: analyzability, architectural integrity, changeability, extensibility, portability, testability and domain specific attributes [4]. Przybylek explains software evolvability as the ease of software to be updated in order to fulfill new requirements, and states that changes to a system are inevitable [5]. However, implementing such changes affects the modularity of the software, leading to code tangling and scattering, therefore lowering the adaptability, and defeating the initial purpose [5]. This aligns with Lehman's *Law of increasing complexity*, stating that continuous change results in an increased complexity due to structure degradation [6].

Software evolability is intertwined, and often confused, with software maintainability, as both metrics share characteristics [7]. In general, software evolvability is less standardized and harder to quantify. The issue of software maintainability is nearly as old as software engineering itself, and as early as the 70's, researchers have attempted to understand the underlying factors through metrics like cyclomatic complexity [8]. As software evolved, more sophisticated metrics emerged, such

as an overall *Maintainability Index (MI)*, often integrated into software tools. The use of such an integrated MI has been criticized by several authors, and while some have been contemplating other models than simple polynomials [9], others have even suggested that there seems to be no consistency between the different models [10].

B. Toward Evolvable Software Systems

NST proposes the concept of *combinatorial effects* as a key mechanism that causes the lack of software evolvability [3]. A combinatorial effect occurs when the impact of a change is not only dependent on the change itself, but also on the size of the system [3] [11]. Postulating that these combinatorial effects should not exist in information systems, NST derives four theorems. Every theorem entails a principle or necessary condition that software needs to adhere to avoid such combinatorial effects [12].

- *Separation of Concerns*
  All software constructs, such as methods or classes, shall only contain a single concern or change driver.
- *Data Version Transparency*
  All data passed between software constructs shall not require to adapt the code to support a new data version.
- *Action Version Transparency*
  All tasks or action constructs shall not require to adapt the calling code to use a new action version.
- *Separation of States*
  All software that calls tasks or action constructs, shall provide state keeping external to the calling construct.

While these principles are relatively straightforward, it is difficult and cumbersome for developers to adhere to these principles at all time in today's often complicated programming environments. Moreover, it is time consuming and often difficult, even for experienced developers, to evaluate in actual programming code whether any of these principles has been violated. Nevertheless, as these violations are — according to NST — one of the main causes limiting evolvability, an automated mechanism to detect these violations could be of great value. In this contribution, we explore the use of AI tools to perform such an automated detection.

III. Setting up Retrieval Augmented Generation

It is easy to verify through a simple question that most LLMs are already aware of NST in general, and its evolvability theorems in particular. However, it seemed highly unlikely that these LLMs already have the knowledge required to perform a sound analysis of programming code to investigate whether actual source code violates one or more of these theorems. This assumption, based on the fact that the most common literature found around NST contains only a very limited amount of code examples, was confirmed in several initial experiments. At first, this may seem to present a challenge as retraining an LLM requires in general a significant amount of hardware resources.

However, instead of retraining the entire model, it is also possible to extend the model with new information. A possible way to extend the model is through *RAG*. Using RAG, it is possible to supply additional context by searching a local knowledge source based on the input prompt, and adding this information to the query for the LLM [13]. The flow of such a *contextualized prompt* [14], providing contextualized information to an LLM, is schematically represented in Figure 1. For the implementation, we have combined the *OpenAI API* [15], based on the *GPT-4o-mini* model, with the *LangChain* framework [16]. While both frameworks provide a *Python API* hiding unnecessary complexity, their integrated use offers sufficient functionality for our purposes.

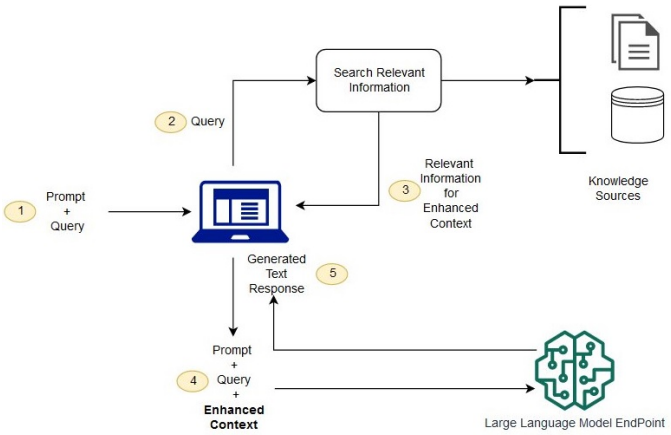

Fig. 1. Schematic model of the flow of a contextualized prompt [14].

The integration consists of three steps: the creation of a knowledge base, converting the contents of the knowledge base, and using the combined information to generate a response. The *Knowledge Base (KB)* consists of the additional information that contextualizes future queries to the LLM, i.e., in our case, information related to NST. We decided to use a simple and general way to capture information in the knowledge base, constructing it as a set of text entries containing information. As LLMs use embeddings to quickly generate an answer to the provided queries, we converted the text entries of the generated knowledge base into a similar format, using the OpenAI API to create the embeddings and store them in a *float32 array*.

A schematic representation of embedding the additional knowledge and storing the contextualized request is presented in Figure 2. To store vectors, we made use of the *Facebook AI Similarity Search (FAISS)* index [17]. The use of this index has several advantages. First, it allows for the efficient searching of dense vectors due to the possibility for batch processing and its efficient similarity search. Second, it allows searching for k-nearest neighbours instead of simply 1-nearest neighbour, enabling the retriever to gather more contextualized information [17]. The FAISS index was then used in conjunction with a retriever class, its responsibility limited to gathering data from the index. It exposes one method $retrieve(query, top\_k)$ which requires the query data to search on, and an optional parameter for the number of nearest neighbours to return.

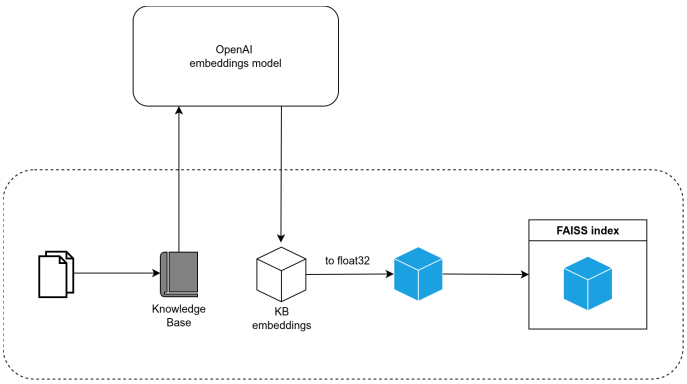

Fig. 2. Representation of the knowledge base integration using FAISS.

The RAG setup was validated by adding a definition of the fictitious word *squirrelthunk*. Without additional information from the KB, the LLM provides the correct answer when we ask what a squirrelthunk means, i.e., that squirrelthunk does not have a widely recognized meaning. We then added an entry in the knowledge base with a fictitious definition, i.e., *A squirrelthunk is the sound that a squirrel makes when it falls out of a tree*. After embedding this definition in the knowledge base, the LLM provided our definition, followed by its own interpretation based on the semantics of the word. This validates the setup to extend the LLMs knowledge base by constructing an additional KB, embedding its contents, and prompting the LLM with both context and the initial query.

IV. Detecting Violations of NST Theorems

In this section, we set forth the construction of the actual artifact to detect possible violations of the NST evolvability theorems based on the RAG architecture.

A. A Naive Approach

A very straightforward solution towards integrating NST into an LLM is to provide the LLM with the theoretical foundations. More specifically, the main book on NST contains the essence of the theory that could aid an LLM in understanding concepts such as combinatorial effects [3]. To store the book in the KB, we divided it into chunks of text, each containing 1000 characters with overlaps of 200 characters. Every chunk is first embedded using text embeddings from OpenAI. Once embedded, they are stored in *Chroma*, an open source vector database provided by LangChain, which natively works with the framework, and allows for easy document embedding storage. Due to the more native support for storing and retrieving documents, we considered Chroma to be a more optimal solution for larger documents. Finally, the standard retriever function provided by the framework is used to compute similarities and retrieve relevant pieces of code. This approach improves upon the general design of using FAISS that was previously explained in Section III. Finally, the queries were created, and divided into a system query, containing instructions for the behaviour of the LLM based on *LangSmith* [18], and a user query, asking to find violations of the NST theorems.

This theoretical approach produced unsatisfactory results regarding both correctness and consistency. By analyzing the results of the search function during the experimental sessions with this model, we identified two major issues.

1) *The data in the knowledge base was too generic.* This was reflected in the fact that the answers from the LLM seemed to rely more on its internally trained data, than on the additional information from the knowledge base. As such, the consistency of the answers varied widely between identical prompts. In addition, the correctness of the answers varied significantly.
2) *The representation of the code did not sufficiently convey its structure.* In most cases, the LLM would identify "separation of concerns", or a related issue such as low cohesion, as the violated theorem, and failed to detect obvious violations of other theorems. We speculate that this could be due to the internal workings of the LLM, causing it to focus on the linguistic aspect of the code. The LLM seemed to determine violations of "separation of concerns" by investigating the naming of variables and methods in the class.

B. A More Practical Knowledge Base

To provide a more useful knowledge base to analyze source code, the first step is to change the knowledge base from theoretical knowledge to practical examples. More specifically, the knowledge base was updated to include the following types of information.

1) Individual Java source files, where each file primarily violates one NST theorem.
2) A metadata file, containing an entry per file containing a combinatorial effect. This entry contains the file name, the violated NST theorem, and a description, in natural language, of the reason for the violation.

One could argue that most real-life code that introduces combinatorial effects would violate multiple theorems to an extent. Though we could have supported this by annotating the files with all tags of the violated theorems, and adding multiple entries per file in the metadata, we decided to limit the scope of the initial artifact to the use of code examples that violate one theorem specifically.

C. Conveying Code Structure in Language

The second issue discussed in Section IV-A is related to the inherent use of natural language in LLMs. A natural language processing engine seems to distinguish the interdependency of variables and methods based on words, i.e., the naming of the variables. This would obviously impede its capability to detect violations of NST theorems. To convey code structure in natural language, we propose to introduce an additional layer to translate that code structure into natural language. This approach is similar to platforms that use AI to analyze or generate code, as they seem to believe as well that parsing code beforehand yields better results [19].

To make the code structure more explicit, we convert the code into an *AST* representation. This transformation makes the code structure visible, while preserving the naming of both the different code constructs and the variables. Though ASTs are most commonly visualized in graphs, it is possible to extract a text-based representation. As represented in Figure 3, the code structure is visualized through the use of programming language independent constructs. For example, an $if(a > 2)$ statement in Java will be translated to an $IfStatement$ with a $BinaryOperation$ containing the operands of the statement. In addition to code constructs, naming is preserved, e.g., the variable name $orderDetails$ is captured correctly.

To fully leverage the capabilities of RAG, both the knowledge base and the user prompted code need to be transformed into a text-based AST representation. In addition to solving the issues discussed above, we also believe that this guides the LLM to focus more on the information in the additional knowledge base, rather than on its own training data containing mostly source code in natural language. This hybrid approach, leading toward the LLM leveraging both the code structure and the names of these constructs in natural language, will be used in the remainder of this contribution.

D. An Integrated Hybrid Implementation

The integrated hybrid implementation to detect violations of NST theorems goes through the following steps.

1) Read the examples and metadata from the file system, and initialize the knowledge base.
2) Iterate over these examples, transform them into a text-based AST representation, and apply serialization.
3) Embed the serialized AST representations using the OpenAI embeddings.
4) Store the embeddings with metadata in the vector store.
5) Read the source code file to be investigated.
6) Transform the source code file into a text-based AST representation, and apply serialization.
7) Retrieve similar code examples and their metadata from the vector store.
8) Prompt the LLM with the context examples and the user query, i.e., source code and request to detect violations.

Next to the transformation from the code to a text-based AST representation, we also introduced a serialization step. This step makes the format more human readable by adding indentation and removing unnecessary clutter from the translation step. After we retrieve the examples and their metadata from the vector store, we explicitly add them and their metadata to the user query. This gives the processing LLM direct access to the code, the violated theorem, and the violation reason. In the final query, it is stated explicitly that the LLM should find combinatorial effects in the user code, as we found the LLM to sometimes simply return combinatorial effects from the examples in the knowledge base. Furthermore, a suggested fix is included in the answer. As the current version of the knowledge base does not contain information on how to solve the combinatorial effects, this suggested fix will mostly be generated from the LLMs internal knowledge base.

V. Results and Discussion

In this section, we present the experiment that we performed to validate the artifact, and discuss the results.

A. Experiment Setup

To allow users to test the artifact, a web application was created that functions as a proxy layer between the user and the algorithm. The flow of the web application is quite straightforward. It offers a simple input form, consisting of an entry field for the participant number, and an upload field to upload a file that needs to be analyzed. When the file has been uploaded, the application starts the analysis script to execute the algorithm, requests the user to wait, and presents the results when they become available. An example of such a presented result is shown in Figure 4.

The script containing the algorithm was not shared directly with the participants for two reasons. First, sharing the API key of the script could lead to abuse, which could negatively impact the experience of other participants and the quality of the results. Second, providing the script directly to the participants would require them to set up a development environment, which could lead to unforeseen issues.

The knowledge base was extended with more code examples, partly from real world applications, and partly crafted by the first author. This resulted in a knowledge base with four examples related to action version transparency, four examples for data version transparency, nine examples for separation of concerns and six examples for separation of states.

The invitation to participate in the experiment was sent out to 48 eligible candidates of which 14 responded positively. After reaching the deadline, 13 participants had submitted results which were deemed valid, as we required the questionnaire to be completed in its entirety. Each participant had a varying level of expertise using NST software development, with a minimum of 0.5 *Years of Experience (YoE)*, a median of 3 YoE and a maximum of 9 YoE. Every participant was given an anonymous participant number, which also served to authenticate the participants when using the web application.

B. Analyzing Perceived Accuracy

The analysis of perceived accuracy was made using three different experiments. The first experiment was based on a Java file with a single method, the second experiment on a Java file with two interacting methods, and the third experiment on a Java file containing an entire class. For every experiment, the participants were asked to rate three different aspects of the response generated by the artifact, i.e., which theorem was being violated, the artifact's explanation, and the suggested fix. Participants were asked to choose between the ratings *correct*, *partially correct*, and *incorrect*.

For all three questions in all three experiments, 7 or more participants rated the artifact's anwser as *correct*, and the number of participants rating an answer as *incorrect* was never

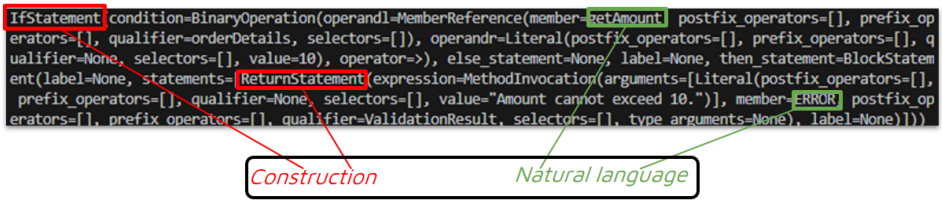

Fig. 3. Example of an AST source code representation combining both structure and natural language.

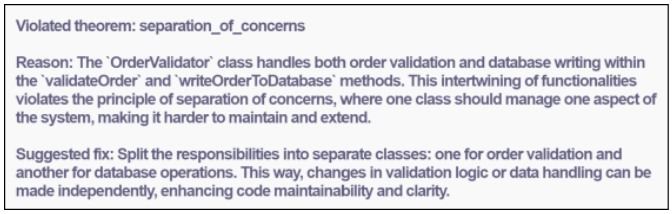

Fig. 4. An example of an analysis result presented by the system.

larger than 2. For the first question, i.e., which theorem was violated, the accuracy increased per experiment, as *partially correct* responses gave way to *correct* answers. This seemed remarkable as we found during initial experimentation that the hallucination levels of the LLM tended to increase when attempting to analyze larger code fragments. This observation aligned with other research showing that larger, more complex contexts seem to degrade the accuracy of the LLM's output significantly [20]. After analyzing the outputs of the LLM for the participants' input code for the third experiment, we found that the LLM determined the violated theorem to be *separation of concerns* in every case. This behavior, which could explain why the performance seemingly increases between the different experiments, seems logical. As the source code grows, the number of methods typically increases, which leads in general to more inter-module dependencies, that could exhibit more undesirable types of coupling, and therefore less adequate separation of concerns. Moreover, the LLM's bias toward separation of concerns, which was present to some extent in the earlier experiments as well, could be consistent with the LLM's own training data as it is the one theorem most closely aligned with common design knowledge.

C. Intention to Use, Trust, and Perceived Usefulness

In a second part of our experiment, the participants were asked to answer a number of questions based on the use of a 5-point Likert scale. Table I presents the answers of the participants for the questions related to their intention to use, trust, and perceived usefulness regarding the artifact.

The results seem to indicate that the participants do not seem to feel that they are able to depend on the artifact, and clearly seem to have doubts that the artifact can increase their speed and productivity. On the other hand, they seem convinced that the artifact can contribute to their effectiveness, and that it definitely has the potential to be a useful tool. This seems to be in line with a rather widespread belief that people do not feel that LLMs can be blindly trusted, but that they do appreciate them as an assistant or companion.

VI. Conclusion and Future Work

While it is widely accepted that AI tools may support and improve the software development process in many ways, we have focused in this contribution on its use regarding software evolvability. More specifically, we have presented an artifact that leverages LLMs through RAG to automatically detect violations of software source code with respect to the evolvability theorems proposed by *NST* [3]. The presented RAG-based artifact uses an hybrid architecture, with a knowledge base containing practical examples, and the use of an AST representation to convey code structure into natural language. We acknowledge the limitations of the presented artifact, such as the limited number of practical examples, and their constraint to having only a single theorem violation in one file.

To validate the artifact, an experimental setup was created. Thirteen experienced NST software developers participated in this validation, engaging in three separate experiments through a web application. Though the participants did not feel that they could depend on the artifact, they were convinced that the artifact could be a useful tool to improve their effectiveness. We recognize the margin of error for the quantitative analysis, and the possible bias due to the fact that all participants worked as software developers for the same company. Therefore, the results in this work should be interpreted carefully. Nevertheless, the use of LLMs with RAG seems a promising way to leverage AI tools to improve the quality of software development in general, and software evolvability in particular. Moreover, as the presented artifact is designed to provide developers with advice and suggestions, the lack of repeatability, a well known issue with current AI tools, would not be a major problem.

Various options are being considered to improve the current artifact, and/or increase the value that it could provide during software development. First, more practical examples could be added to the knowledge base. Besides increasing the number of examples, limitations on the internal complexity could be lifted, e.g., more theorem violations in a single file, and examples in other languages could be included. Second, the artifact could be modified to use multiple available AI models, and return a summary of the various results. Third, the artifact could be integrated into the environment of the software developer. This integration could be implemented in real-time,

TABLE I
PARTICIPANTS' INTENTION TO USE, TRUST, AND PERCEIVED USEFULNESS [2].

Question	Strongly disagree	Somewhat disagree	Neither agree nor disagree	Somewhat agree	Strongly agree
Q07: I can depend on this system	0	5	2	2	1
Q09: I intend to use this system in the next 6 months	1	2	3	6	1
Q11: By using this product at my job, I could execute tasks more quickly	1	8	2	2	0
Q12: The use of this product would increase my productivity	0	3	7	2	1
Q13: The use of this product would increase my effectiveness at work	0	2	2	7	2
Q14: I find this product useful at my job	0	1	0	7	5

using a plugin for an *Interactive Development Environment (IDE)*, or in batch mode, as part of a software factory control system coordinating the DevOps environment [21]. Finally, it could be investigated to transition from the RAG-based setup to the training of a dedicated AI model, that would possess a deeper level of NST knowledge, both practical and theoretical.

REFERENCES

[1] H. Mannaert, K. De Cock, P. Uhnak, and J. Verelst, "On the realization of meta-circular code generation and two-sided collaborative metaprogramming," *International Journal on Advances in Software*, vol. 13, no. 3-4, pp. 149–159, 2020.

[2] D. Peeters, "Leveraging retrieval-augmented generation (rag) llms for custom code validation through normalized systems theory," Master's thesis, Antwerp Management School, 2025.

[3] H. Mannaert, J. Verelst, and P. De Bruyn, *Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design*. Koppa, 2016.

[4] H. P. Breivold, I. Crnkovic, and M. Larsson, "Software architecture evolution through evolvability analysis," *The Journal of systems and software*, vol. 85, no. 11, p. 2574–2592, 2012.

[5] A. Przybyłek, "An empirical study on the impact of aspectj on software evolvability," *Empirical software engineering : an international journal*, vol. 23, no. 4, p. 2018–2050, 2018.

[6] M. Lehman, "Programs, life cycles, and laws of software evolution," in *Proceedings of the IEEE*, vol. 68, 1980, pp. 1060–1076.

[7] S. Ciraci and P. van den Broek, "Evolvability as a quality attribute of software architectures," in *Proceedings ERCIM Workshop Software Evolution (EVOL '06)*, 2006, pp. 29–31.

[8] T. J. McCabe, "A complexity measure," *IEEE Transactions On Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[9] M. Riaz, E. Mendes, and E. Tempero, "A systematic review of software maintainability prediction and metrics," in *Proceedings Third International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 367–377.

[10] D. I. Sjøberg, B. Anda, and A. Mockus, "Questioning software maintenance metrics: A comparative case study," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2012, pp. 107–110.

[11] H. Mannaert, J. Verelst, and K. Ven, "Towards evolvable software architectures based on systems theoretic stability," *Software: Practice and Experience*, vol. 42, no. 1, pp. 89–116, 2012.

[12] ——, "The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability," *Science of Computer Programming*, vol. 76, no. 12, pp. 1210–1222, 2011, special Issue on Software Evolution, Adaptability and Variability.

[13] "What is retrieval-augmented generation?" URL: https://research.ibm.com/blog/retrieval-augmented-generation-RAG, IBM. 22 August 2023, [accessed: 2025-09-08].

[14] "What is RAG (Retrieval-Augmented Generation)?" URL: https://aws.amazon.com/what-is/retrieval-augmented-generation/, AWS. 16 July 2024, [accessed: 2025-09-08].

[15] "Openai developer platform," URL: https://platform.openai.com/docs/overview/, OpenAI. 2024, [accessed: 2025-09-08].

[16] "Contextualize your LLM App," URL: https://www.langchain.com/retrieval, LangChain. 2024, [accessed: 2025-09-08].

[17] "Welcome to Faiss Documentation," URL: https://faiss.ai, Meta. 2024, [accessed: 2025-09-08].

[18] "LangSmith Playground," URL: https://smith.langchain.com/hub/wfh/rag-prompt/playground, LangChain. 2025, [accessed: 2025-09-08].

[19] "Using code syntax parsing for generative ai," URL: https://windsurf.com/blog/using-codesyntax-parsing-for-generative-ai, Windsurf. 2025, [accessed: 2025-09-08].

[20] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157—-173, 2024.

[21] H. Mannaert, J. Verelst, K. De Cock, and J. Faes, "An integrated software manufacturing control system for a software factory with built-in rejuvenation," *International Journal on Advances in Software*, vol. 17, no. 1-2, pp. 90–99, 2024.