

A Generative AI Based Architecture for Data Seeding in Software Testing

Itamir de Morais Barroca Filho(✉), Ramon Santos Malaquias, Jean Mário Moreira de Lima, André Morais Gurgel, Ney Pimentel Targino de Oliveira, Murilo Henrique Dantas de Oliveira Bezerra, Gabriel Paes Landim Lucena, and Vinicius Oliveira da Silva

Digital Metropolis Institute, Federal University of Rio Grande do Norte, Natal 59078970, Brazil
{itamir.filho,malaquias,jean.lima}@imd.ufrn.br, andre.gurgel@ufrn.br
https://portal.imd.ufrn.br/portal/

Abstract. In software engineering, the testing phase is critical to ensuring application quality, reliability, and adherence to defined requirements. Effective testing requires realistic, consistent data that reflects real-world conditions and exists in sufficient volumes. This makes the data seeding process indispensable. However, traditional approaches to data creation—manual or automated—face challenges such as ensuring data diversity and compliance with business rules, often resulting in inadequate test coverage with consequences in undetected bugs, reduced reliability, higher costs and compromised software quality. Thus, the goal of this paper is to present a novel generative AI-based architecture for advancing data seeding practices in software testing. The proposed architecture integrates user interaction, data embedding, and retrieval-augmented generation (RAG) to form a seamless pipeline for generating contextually relevant and realistic data in real-time. For the purposes of validating the architecture, a Proof of Concept (POC) was implemented focused on generating fake data from Brazilian customers to assess the quality of the data generated and the effectiveness of the implemented solution, in order to evaluate the potential of the architecture for the development of broader applications and in different contexts.

Keywords: Data seeding · Generative AI · Software testing · Retrieval-augmented generation

1 Introduction

Software testing is a critical phase of software engineering. It is defined as the process of evaluating and verifying that a software application or system meets the specified requirements and functions correctly. It involves executing a program to identify errors, ensuring that the software behaves as expected under various

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025
O. Gervasi et al. (Eds.): ICCSA 2025, LNCS 15649, pp. 234–249, 2025.
https://doi.org/10.1007/978-3-031-96997-3_15

conditions, and validating that it satisfies the user's needs [32]. Testing can be performed at different levels, including unit testing, integration testing, system testing, and acceptance testing, each focusing on various aspects of the software. The goal is to ensure quality, reliability, and performance before deploying the software in production.

In the system testing process, the testers usually have to create some data that will be input into the digital forms of the application that is being tested. This task can be challenging, and the more realistic this data is, the more accurate the test can be.

In this context, data seeding is populating a database with initial data to simulate real-world scenarios for testing or development purposes [25]. This data can include sample records, relationships, and configurations representing typical or edge-case situations. It can be complex and time-consuming. The challenges are numerous, including data volume and data realism. In terms of volume, generating substantial data to test applications properly can be laborious, especially when dealing with large databases. As for data realism, ensuring that the data generated accurately represents real-world scenarios is crucial. Manually creating such data can lead to consistency and possible inaccuracies.

Alternatively, we can leverage Artificial Intelligence (AI) for data seeding solutions, specifically generative AI such as Generative Adversarial Networks (GANs) [30] or Natural Language Processing (NLP) [8]. First, data generation itself: we can train it to generate diverse and realistic data that fit specific testing scenarios [22]. For example, it can create customer profiles, transaction histories, or product catalogs with varying attributes. Second, data variability: AI can provide diverse data sets, enabling thorough testing of edge cases and scenarios that may not be easy to create manually. Finally, data validation: AI can also assist in data validation by identifying anomalies and inconsistencies in the generated data to improve quality [1].

Adopting generative AI for data seeding can significantly improve the efficiency and effectiveness of testing processes for systems of information. This innovation can make testing more accurate, scalable, and secure, ultimately leading to higher-quality software products. Although we have tools to automate software testing, such as Cypress [9] and Selenium [27], to the best of our knowledge, they do not have features that can help with the test case data generation and data seeding based on a singular context, and not making usage of Generative AI.

Thus, this paper presents an architecture based on Generative AI to help develop tools for data seeding for software testing. We will present this architecture and a tool for data test generation for a customer application. This paper is structured as follows: in Sect. 2, we report a literature review presenting related works that address technologies, cases, and approaches to using Generative AI in software testing. Section 3 describes the proposed architecture for data seeding based on Generative AI, while Sect. 4 describes the implementation of a data seeding solution for generating Brazilian customer data based on the proposed

architecture. Finally, in Sect. 5, we present the concluding remarks and the future works.

2 Related Work

The application of Generative AI in software testing is an emerging research area aimed at improving software quality assurance through automated test data generation. Various approaches have been explored to enhance the diversity, efficiency, and coverage of test cases. For instance, Generative Adversarial Networks (GANs) have been used to synthesize realistic and diverse datasets, enabling more robust validation of machine learning models [15]. Additionally, evolutionary algorithms, such as genetic algorithms, optimize test coverage by iteratively evolving test inputs based on predefined functions [2]. Deep reinforcement learning has also been investigated for dynamically generating test cases by adapting to system behavior and constraints [33].

More recently, Large Language Models (LLMs) have gained attention for their ability to generate contextually relevant text-based data [21], facilitating the validation of natural language processing applications and other software requiring structured textual inputs. This section reviews these approaches, highlighting their strengths, limitations, and contributions to automated test data generation in software testing.

2.1 Generative Adversarial Networks for Test Data Generation

Generative Adversarial Networks (GANs) have become one of the leading approaches for generating test data due to their ability to produce realistic and diverse data sets. X. Guo in [15] proposed utilizing GANs as a test case generator, specifically enhancing the test coverage by generating new data instances that reflect complex software behaviors. Despite his work achieving promising results–particularly in unit testing–the approach was limited by its evaluation on systems with only a few hundred branches, leaving questions about its generalizability and scalability.

Another study by Guo *et al.* [16] introduced significant improvements through a GAN-based framework for automatic test data generation that targets branch coverage in both unit and integration testing. The later research expanded the experimental scope by:

1. **Comparing Multiple GAN Architectures:** The study evaluated various GAN models, including WGAN-GP, BiGAN, and the standard GAN, revealing that WGAN-GP, with its enhanced training stability and smoother gradient properties, provided superior performance in terms of path prediction accuracy and test coverage.
2. **Enhancing Data Selection Strategies:** A novel selection mechanism was introduced to identify generated test cases that target unexecuted branches, thereby systematically increasing branch coverage even in more complex software modules.

3. **Scaling to Complex Systems:** By applying the framework to software modules with a more substantial number of branches compared to previous studies, the research demonstrated initial scalability potential. However, while the study expanded on previous evaluations, it was still constrained to modules with branch counts in the hundreds, leaving the question open for generative capabilities to large-scale enterprise applications with thousands of execution paths.

However, despite these improvements, GAN-based approaches remain fundamentally constrained in large-scale enterprise applications due to their reliance on execution feedback rather than semantic validation. Furthermore, both [15,16] require instrumenting the software and using execution feedback (e.g., branch hits) to train the GAN generator, following a typical coverage-driven fuzz testing cycle. While these methods generate diverse test inputs for structural execution paths, they lack business-rule enforcement.

This distinction is crucial: while prior GAN-based approaches optimize for execution path diversity, they do not assess whether generated inputs are meaningful within real-world constraints. For example, [16] generates test inputs that explore various execution paths, yet these inputs may consist of arbitrary numerical values lacking contextual validity. In contrast, our method ensures that generated test data represents structured, business-compliant customer profiles, aligning with real-world software requirements. Thus, structural test coverage alone does not guarantee realistic data—an overlooked gap that our RAG/LLM-based architecture directly addresses.

2.2 Machine Learning and Neural Networks in Test Data Generation

Machine Learning (ML) has emerged as a powerful tool for automating test data generation in software testing. Traditional fuzz testing, though effective at finding vulnerabilities, often suffers from low test coverage due to its random nature. To overcome this, researchers have developed neural network-based generative models that produce structured and diverse test inputs.

Paduraru and Melemciuc [26] introduced an open-source tool using recurrent neural networks (RNNs) and Long Short-Term Memory (LSTM) models for automatic test data generation. The tool clusters input data and trains separate generative models per cluster, effectively capturing complex dependencies via sequence-to-sequence architectures. Experiments on XML, HTTP, and PDF parsers showed that this method achieved about 20% higher branch coverage than genetic algorithm-driven fuzz testing, aided by strategies such as *SampleSpace* for maintaining structural correctness with controlled randomness.

[26] emphasize the importance of generating structured and valid inputs, mentioning the *SampleSpace* strategy to ensure structural correctness. However, their method still relied on manual interventions, such as custom hooks that incorporated expert knowledge into the generator. While this approach required specific configurations to enforce patterns, our architecture automates this

integration via RAG, systematically incorporating domain knowledge through retrieval and dynamic generation. This advancement eliminates the need for manual adjustments, making the process more efficient and scalable.

However, the approach depends heavily on the quality and diversity of the training corpus, introduces significant computational overhead during training, and utilizes clustering based on file type rather than semantic similarity. In contrast, our proposed method integrates Retrieval-Augmented Generation (RAG) with Large Language Models (LLMs) to dynamically refine test data based on real-time inputs and business rules.

2.3 Genetic Algorithms for Test Data Generation

Genetic Algorithms (GAs) have been widely explored as an effective means for automating test data generation, particularly in structural software testing. The key advantage of GAs is their ability to search large input spaces efficiently, optimizing test case selection for maximum coverage. However, traditional GA-based test generation methods face challenges related to computational cost, fitness function design, and adaptability to complex software environments.

Ahmed and Hermadi [2] propose a GA-based test data generator that improves upon previous methods by synthesizing multiple test cases in a single GA run, rather than generating one test datum per iteration. Their approach focuses on path coverage criteria and incorporates various fitness function designs, including weighting schemes and path traversal techniques. Experimental results demonstrate improved efficiency over traditional single-path GA-based methods; however, the study primarily evaluates mathematical and sorting algorithms, leaving questions regarding scalability to enterprise applications.

Sofokleous and Andreou [31] present a more advanced dynamic test data generation framework that integrates control flow graph (CFG) analysis with two GA-based optimization algorithms: Batch-Optimistic (BO) and Close-Up (CU). The BO algorithm evolves test case sets to maximize edge/condition coverage, while the CU algorithm refines test generation by targeting uncovered paths. Although their framework achieves higher test adequacy compared to traditional random or gradient-descent methods, it suffers from high computational costs, limited handling of object-oriented features, and reliance on static CFG analysis.

Compared to these studies, our proposed approach extends beyond structural test generation by integrating Retrieval-Augmented Generation (RAG) and Large Language Models (LLMs). While GA-based approaches optimize test input selection at the control flow level, they do not incorporate semantic understanding or business rule constraints. In contrast, our architecture dynamically generates test data that aligns with domain-specific requirements by leveraging contextual retrieval from a vector database and LLM-based generative capabilities. This enables the generation of diverse, realistic, and semantically valid test cases, addressing limitations in existing GA-based methods. Moreover, our approach enhances adaptability through real-time user interaction, ensuring flexibility in enterprise applications where business logic and data constraints are crucial. This hybrid methodology represents a significant advancement toward

intelligent, context-aware test data generation, bridging the gap between automated structural testing and domain-driven test scenario validation.

2.4 Deep Learning and Reinforcement Learning Approaches

Deep learning and reinforcement learning have emerged as powerful methodologies for automated software testing. These approaches enable the generation of structured test inputs, optimize test selection, and improve software validation through intelligent search mechanisms.

Bayrı and Demirel [4] explore the role of Large Language Models (LLMs) in modern software testing. Their study demonstrates that models like ChatGPT and Codex can automatically generate test cases, optimize test suites, and assist in bug localization. By leveraging Reinforcement Learning from Human Feedback (RLHF), LLMs refine their understanding of software requirements and improve test adequacy. However, the effectiveness of LLM-generated test cases remains highly dependent on prompt engineering and domain adaptation, highlighting a need for further refinement in enterprise applications.

Tsai and Taylor [33] introduce DeepRNG, a deep reinforcement learning-assisted framework for software testing. The framework augments traditional random number generators (RNGs) with an RL agent, using call stack representations as input states. Experimental results on the Cosmos SDK, a blockchain framework with over 350,000 lines of code, demonstrate statistically significant improvements in test coverage. Unlike genetic algorithm-based fuzzing, DeepRNG dynamically adapts test input generation based on learned policies.

Beyond coverage improvements, DeepRNG was able to identify previously unknown real bugs in the Cosmos SDK, reinforcing the impact of diverse test data generation on defect detection. However, despite these advantages, DeepRNG's reliance on call stack representations limits its ability to incorporate higher-level business logic, and its computational complexity makes it less scalable for enterprise-scale applications.

While both studies contribute to advancing AI-driven test data generation, our proposed approach differs itself by integrating Retrieval-Augmented Generation (RAG) and Large Language Models (LLMs) into a unified architecture that ensures both structural coverage and domain-specific relevance. Unlike DeepRNG, which optimizes random test input selection through reinforcement learning, our framework retrieves contextually relevant data from a vector database and combines it with LLM-based generative capabilities to produce realistic, business-rule-compliant test cases.

Additionally, unlike prior LLM-driven approaches, which rely solely on static prompt engineering, our architecture enables dynamic and user-driven test case refinement, allowing for greater flexibility in enterprise-scale software testing. This hybrid methodology bridges the gap between structural test optimization and intelligent, context-aware validation, ensuring that AI-generated test cases are not only diverse but also aligned with real-world business logic–leading to more reliable and actionable software testing.

2.5 Retrieval-Augmented Generation (RAG)

Recent advancements in combining retrieval mechanisms with LLMs have further expanded their capabilities. Retrieval-Augmented Generation (RAG) separates the task of retrieving relevant context from generating responses, ensuring more grounded and relevant outputs [21].

Jeong [19] explored an enterprise RAG model, demonstrating its effectiveness in mitigating hallucinations and improving domain adaptation. Mao *et al.* [23] introduced Generation-Augmented Retrieval (GAR), which enhances retrieval accuracy by generating enriched queries. However, these studies do not address domain-specific retrieval for test data generation.

Our proposed architecture builds on top of those concepts to leverage RAG with a real-time retrieval for test data generation. Unlike previous works, our approach ensures domain-aware retrieval and validation mechanisms, improving software testing accuracy while maintaining contextual consistency.

The state-of-art for AI applications, LLMs are able to follow rules and use the provided context to generate output by leveraging both pre-trained knowledge and dynamically retrieved contextual information. Attention mechanisms [35] enable these models to focus on relevant portions of input data while adhering to structured patterns learned during training. Rule-following behavior is reinforced through fine-tuning on domain-specific datasets, where explicit constraints and best practices are encoded [17].

The reviewed works demonstrate the evolution of AI focusing in software testing, with various technologies contributing to automated test data generation. GANs, neural networks, genetic algorithms, and reinforcement learning have shown promising results in improving test coverage, efficiency, and scalability. However, challenges such as ensuring data quality, mitigating bias, achieving domain-specific adaptability and generating contextually relevant test data in near real-time remain and present opportunities for further research.

Building on these insights, this paper introduces a novel architecture that leverages Generative AI for data seeding in software testing. Unlike prior approaches, our architecture integrates RAG paradigm to dynamically combine contextual knowledge retrieval with generative capabilities, ensuring that the data is both realistic and tailored to diverse testing scenarios. This dual-layer approach enhances quality and relevance while reducing reliance on extensive pre-training or fine-tuning for domain-specific applications. Additionally, it provides a flexibility to overcome the concept drifts [34], particularly when the training data used by a machine learning model differs from the data which it may encounter in production. We note that no work solves the problem of concept drift – e.g., [26] would need to be manually retrained for new input formats. Recognizing this, our architecture adopts RAG, capable of updating the context in real time, mitigating concept drift.

By addressing these challenges, the proposed architecture not only improves the reliability and scalability of test data generation but also establishes a foundation for broader applications in automated software testing frameworks.

3 Solution Architecture

The software architecture of a system refers to the process of defining the structures and components that compose it, considering their interactions and evolution perspectives [3]. The architecture is a critical aspect in software systems, as it defines the structural basis that supports the behavior, performance, and maintenance of complex systems. In solutions that use Large-Language Models (LLMs), such as the one described in this article, it is necessary to consider architectures that integrate data retrieval and generation to maximize the effectiveness of these models and of the solution as a whole.

In this sense, the Retrieval-Augmented Generation (RAG) paradigm can be an important tool in the data recovery and generation process. RAG is an advanced technique in Natural Language Processing (NLP) that combines information retrieval mechanisms from external databases with natural language generation models, allowing the model to access contextually relevant information during the text generation process [21]. When using RAG, the system initially uses a recovery component that searches for relevant information in a knowledge base relevant to the application context. Then, the relevant information is sent along with the prompt to the LLM, which in turn is responsible for generating a response based on both the retrieved information and its own knowledge, generating a contextualized response [21].

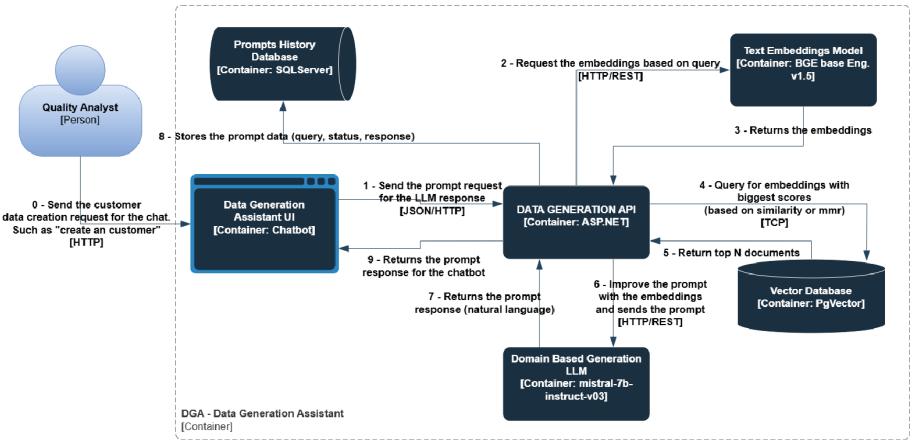

Fig. 1. Solution Architecture - C4 Model Container Diagram

Therefore, for this solution, an architecture based on RAG was defined, mainly considering the data generation requirement for the data seeding process. The generated data has specific characteristics, and for this reason, it is necessary to maintain a database with information relevant to the context. Figure 1 illustrates the Application Container Diagram, using the C4 Model notation. C4

Model is an approach used to document the architecture of software systems in a visual way, using four levels of abstraction (context, containers, components and code) to represent the structure and details of the system in a way that is understandable to different audiences [6].

To illustrate the proposed solution, Fig. 1 is the container level of the solution. In the C4 Model, the container level provides an intermediate view between the context diagram (higher level) and the component diagram (more detailed), where each container can be described in terms of its functionality, technologies used, main responsibilities and interaction with other containers. And Fig. 1 presents the solution containers and the way they communicate between them. In addition, each container is described below.

1. **Quality Analyst:** Person responsible for planning, writing, and executing end-to-end tests. He accesses the system's graphical interface (2) and requests, using natural language, the generation of data to perform the test.
2. **Data Generation Assistant UI:** Graphical interface where, from a chat, the user can request the generation of data for the test. This interface was implemented using a SPA (Single-Page Application) [11] and communicates with the API developed for the data generation process.
3. **Data Generation API:** RESTFul API responsible [10] for receiving the user request (prompt) and performing the data generation process based on the RAG paradigm. This API is responsible for transforming the prompt into an embedding vector by executing the text embedding model (4), verifying which embeddings have the greatest similarity with the specific customer data (5), executing the LLM to generate the customer from the augmented prompt (6), and returning the data to the graphical interface in natural language.
4. **Text Embeddings Model:** Machine Learning Model responsible for transforming text content into a vector representation, which will be used to analyze similarity with specific customer data (5).
5. **Vector Database:** Vector database [36] that stores customer data with the exact characteristics that must be considered for generating new customers from the LLM. It is used to return the top-n customers that have a greater degree of similarity to what was requested by the user, from the vectorized prompt (4).
6. **Domain Based Generation LLM:** Large-Language Model used to generate the desired data from the refined prompt, considering the most relevant data (5) and the context that should be considered in data generation. The model's response must be returned to the Data Generation API (3) so that it can return the result to the Data Generation Assistant graphical interface.

Finally, by using the RAG paradigm, the solution architecture allows integration between retrieving contextual information and generating personalized data specific to the desired context. In addition, the use of embedding models to transform documents into indirect representations for storage in the vector database allows searching for similarities between the prompt sent and the data indexed in the database, returning relevant information to feed the LLM. This

feature allows responses to be more accurate and contextualized. Thus, by allowing data relevant to the context to be stored in the vector database, the architecture allows generating data for and supporting the data seeding process in different use cases, thus meeting the flexibility and scalability requirements of the solutions.

4 Proof of Concept (POC)

In order to validate the proposed architecture for data seeding in software testing, we implemented a proof of concept. The study consisted of developing an Intelligent Platform for generating fictitious customer data for the context of Brazilian companies, based on the National Registry of Legal Entities (CNPJ). This domain was chosen due to the need to ensure realistic and diversified data for automated testing in information systems.

The solution was developed following the software components described in the previous section. The development and instantiation of each of the components proposed by the architecture will be described below.

- The **"Data Generation Assistant UI"** component was developed from a web system (due to the ease with which the user can access applications of this nature, from internet access) using a SPA (Single-Page Application) [11] using React [24] and the Gatsby framework [12];
- The **"Data Generation API"** component was developed from a Restful API using the Python programming language and the FastAPI Framework [28]. This component is responsible for receiving requests (prompts), processing them with the Text Embeddings Model, sending the embeddings to the Vector Database and also making the request to the LLM service;
- The **"Text Embeddings Model"** component was contemplated from the use of AI services provided by the Hugging Faces framework [18] to implement the $BAAI/bge-m3$ embedding model [7]. This model was chosen because it provides multi-lingual support, allows easy integrations and presents high accuracy in semantic search, a key requirement for our solution;
- The **"Vector Database"** component was implemented using a PostgreSQL database [14], using the "pg_vector" feature [20]. The vector database was initially populated with a company database made available from the Brazilian Federal Revenue Service portal [5]. This database includes information about companies operating in Brazil, considering details such as company share capital, unique identification numbers, and corporate purposes. In addition, the dataset provides company names and size classifications, offering valuable insights for use in LLM;
- The **"Domain Based Generation LLM"** component was contemplated by using AI services provided by Google [13]. In this experiment, we used the Gemini 1.5 Flash Model, one of the models provided by Google, designed for applications that require faster response times [29] - such as the chatbot developed by us in this POC.

Based on these definitions, the implementation of this POC followed a structured approach, starting with the extraction of data from public government sources. After this, the data underwent a normalization and cleaning process to ensure consistency and eliminate duplicate or inconsistent records. The data was then converted into vector representations to facilitate semantic search and inserted into PG Vector, enabling integration with the generative AI model. The data flow of this application will be described in the following subsection.

4.1 Data Flow

The application was developed following the proposed architecture and a data flow structured in the Retrieval-Augmented Generation (RAG) paradigm, which combines information retrieval with synthetic data generation to optimize the data seeding process in software testing. Figure 2 illustrates the application's data flow, which is composed of the following steps:

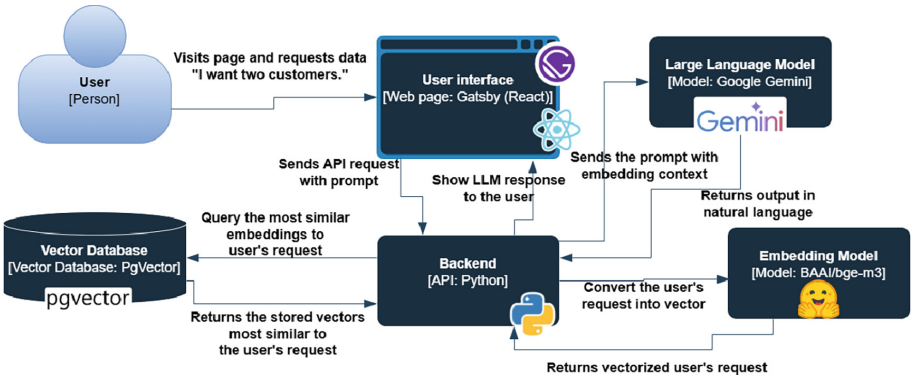

Fig. 2. Flowmap showing the flow of data within the application

1. **User Interaction**:
   The flow begins with the user's interaction, who accesses the web interface and requests the generation of data using natural language, from a chatbot. Figure 3 illustrates an interaction where the user requests the creation of a customer and the chatbot responds with the data of a fake customer, but with valid data for testing.
2. **Request Processing**:
   After the user sends the request, the interface forwards the prompt to the RESTFul API (Data Generation API), which manages the data generation. At this stage:
   - The API receives the request and converts it into a numeric vector using a Text Embeddings model;

- This vector is used to search for contextually relevant records stored in the vector database.

3. **Contextual Data Retrieval**:
   The Vector Database contains structured information about Brazilian companies, extracted from government sources. Using vector similarity measures (such as cosine similarity), the most relevant records to the context of the request (described in item 2) are retrieved and sent to the next step in the flow.

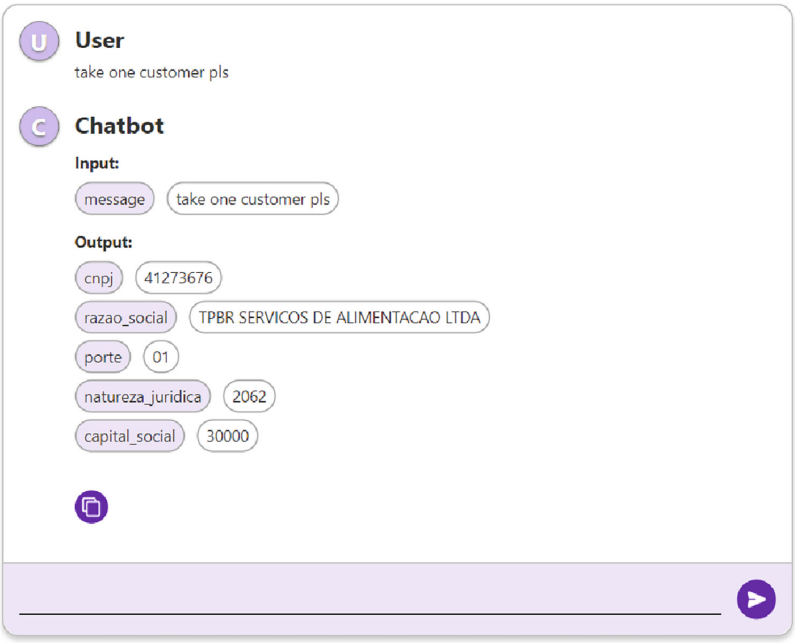

Fig. 3. User data request

4. **Synthetic Data Generation**:
   The data retrieved from the similarity search (described in item 3) is incorporated into the original prompt and sent to a Large Language Model (LLM) model, which processes the input and generates new synthetic data. The model is instructed to follow specific formatting rules, ensuring that the generated data is realistic and consistent with the application domain. The output can be formatted in JSON or natural text, depending on the user request.
5. **Data Delivery and Validation**:
   The data generated by LLM is returned to the API and forwarded to the graphical interface. The interface displays the results to the user in a structured manner, allowing the user to view and copy the generated data for use in tests. Figure 4 illustrates a case where the user wishes to refine the data,

requesting more specific information, such as extracting only the CNPJ or the company address. In this case, the API reprocesses the request, refining the data generation as necessary.

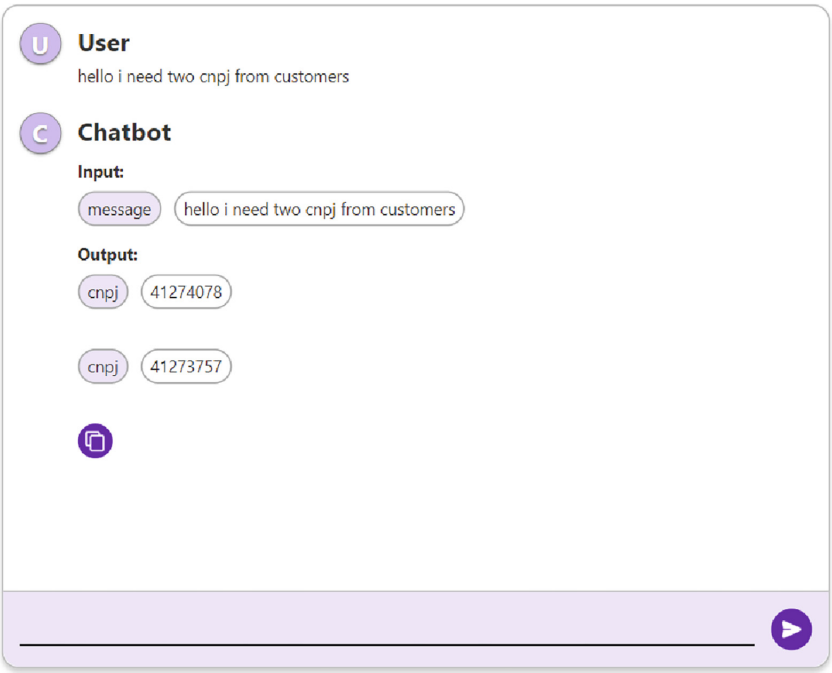

Fig. 4. User data request for specific attributes

Finally, to avoid irrelevant responses, the system was configured to handle messages outside the context of data generation. When the user sends a command unrelated to the application's purpose, the API responds with a message informing that the request cannot be fulfilled. This data flow ensures that the generated information is contextually relevant, flexible, and adaptable to different testing scenarios, optimizing the data seeding process in software applications.

4.2 Results

The POC results demonstrated the effectiveness of the solution in generating test data in an automated and efficient manner. The developed tool was able to generate a large volume of reliable data in a short space of time, optimizing the software development process and reducing the need for manual data entry.

The architecture evaluation showed that the integration of the AI model with the vector database provided an improvement in the relevance and accuracy of

the generated data. In addition, the tool's ability to understand and respond to specific user requests allowed for customization in the data generation process, increasing its applicability in different testing contexts.

Among the benefits observed, the solution's ability to generate diverse data stands out, covering typical scenarios and edge cases in an automated manner. This contributed to expanding test coverage and reducing the incidence of previously unidentified errors.

5 Conclusion and Future Work

This paper presents an architecture based on Generative Artificial Intelligence to improve the data seeding process in software testing. To validate the proposed approach, a Proof of Concept was implemented to generate data from Brazilian customers, demonstrating the solutions and benefits of the solution in a specific context. However, the architecture was designed to be expandable, allowing its application in other domains and contexts of software testing that require realistic and contextualized synthetic data.

In addition to the results obtained in the case study, the tool developed demonstrates an impact on the data seeding process by enabling the generation of a larger volume of reliable data in less time. This not only optimizes the software development process, reducing the need for manual data entry, but also contributes to improving the quality of tests, expanding coverage and facilitating the identification of extreme cases.

Future work will involve further validation of the architecture in more diverse scenarios, ensuring its robustness and adaptability to different industry needs. This includes evaluating its performance in different domains and investigating possible optimizations in the data generation process to ensure that the information generated is always aligned with the specific requirements of each application.

Finally, new validation strategies for the generated data will be explored to mitigate the risk of inconsistencies and ensure that the artificial data is reliable enough to replace real data in software testing. The possibility of improving the integration of the solution with other test automation tools will also be analyzed, making the process even more efficient and accessible for development and quality teams.

References

1. Agrawal, S., Agrawal, J.: Survey on anomaly detection using data mining techniques. Proc. Comput. Sci. **60**, 708–713 (2015)
2. Ahmed, M.A., Hermadi, I.: Ga-based multiple paths test data generator. Comput. Oper. Res. **35**(10), 3107–3124 (2008) 10.1016/j.cor.2007.01.012, https://www.sciencedirect.com/science/article/pii/S0305054807000251, part Special Issue: Search-based Software Engineering
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice: Software Architect Practice_c3. Addison-Wesley (2012)

4. Bayrı, V., Demirel, E.: Ai-powered software testing: The impact of large language models on testing methodologies. In: 2023 4th International Informatics and Software Engineering Conference (IISEC), pp. 1–4 (2023). https://doi.org/10.1109/IISEC59749.2023.10391027
5. do Brasil, R.F.: Dados abertos cnpj - maio 2023 (2023). https://arquivos.receitafederal.gov.br/dados/cnpj/dados_abertos_cnpj/2023-05, http://leanpub.com/software-architecture-for-developers. Accessed 14 Nov 2024
6. Brown, S.: The c4 model for visualising software architecture (2019). https://c4model.com. https://huggingface.co. Acesso em: 7 nov. 2024
7. Chen, J., Xiao, S., Zhang, P., Luo, K., Lian, D., Liu, Z.: M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. In: Ku, L.W., Martins, A., Srikumar, V. (eds.) Findings of the Association for Computational Linguistics: ACL 2024. pp. 2318–2335. Association for Computational Linguistics, Bangkok, Thailand (Aug 2024). https://doi.org/10.18653/v1/2024.findings-acl.137, https://aclanthology.org/2024.findings-acl.137
8. Chowdhary, K.R.: Natural Language Processing, pp. 603–649. Springer India, New Delhi (2020). https://doi.org/10.1007/978-81-322-3972-7_19
9. Cypress.io: Cypress: Javascript end to end testing framework (2024). https://www.cypress.io/, available from https://www.cypress.io/. Accessed Nov 13 2024
10. Feng, X., Shen, J., Fan, Y.: Rest: An alternative to RPC for web services architecture. In: 2009 First International Conference on Future Information Networks, pp. 7–10 (2009). https://doi.org/10.1109/ICFIN.2009.5339611
11. Flanagan, D.: JavaScript: The Definitive Guide, 5th edn. O'Reilly, Sebastopol, CA (2006)
12. Gatsby, Inc: Gatsby (2015). https://www.gatsbyjs.com/, available from https://www.gatsbyjs.com/. Accessed Nov 27 2024
13. Google LLC: Google cloud gemini 1.5 flash (2024). https://github.com/google-gemini/generative-ai-python, https://github.com/google-gemini/generative-ai-python. Accessed Oct 29 2024
14. Group, P.G.D.: Postgresql: The world's most advanced open source relational database (2025). https://www.postgresql.org/. Accessed 23 Feb 2025
15. Guo, X.: Towards automated software testing with generative adversarial networks. In: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S), pp. 21–22 (2021). https://doi.org/10.1109/DSN-S52858.2021.00021
16. Guo, X., Okamura, H., Dohi, T.: Automated software test data generation with generative adversarial networks. IEEE Access **10**, 20690–20700 (2022). https://doi.org/10.1109/ACCESS.2022.3153347
17. Hadi, M.U., et al.: Large language models: A comprehensive survey of its applications, challenges, limitations, and future prospects (07 2023). https://doi.org/10.36227/techrxiv.23589741
18. Hugging Face: Transformers: State-of-the-art machine learning for NLP. (2024). https://huggingface.co. Accessed Oct 29 2024
19. Jeong, C.: A study on the implementation of generative AI services using an enterprise data-based LLM application architecture. Adv. Artif. Intell. Mach. Learn. **03**(04), 1588–1618 (2023). https://doi.org/10.54364/aaiml.2023.1191
20. Kane, A.: Pgvector: Open-source vector similarity search for postgres (2021). https://github.com/pgvector/pgvector. Accessed 23 Feb 2025

21. Lewis, P., et al.: Retrieval-augmented generation for knowledge-intensive NLP tasks. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) Advances in Neural Information Processing Systems. vol. 33, pp. 9459–9474. Curran Associates, Inc. (2020)
22. Liu, R., et al.: Best practices and lessons learned on synthetic data. arXiv preprint arXiv:2404.07503 (2024)
23. Mao, Y., He, P., Liu, X., Shen, Y., Gao, J., Han, J., Chen, W.: Generation-augmented retrieval for open-domain question answering. In: Zong, C., Xia, F., Li, W., Navigli, R. (eds.) Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pp. 4089–4100. Association for Computational Linguistics, Online (Aug 2021). https://doi.org/10.18653/v1/2021.acl-long.316, https://aclanthology.org/2021.acl-long.316
24. Meta Platforms, I.: React: A javascript library for building user interfaces (2025). https://github.com/facebook/react. Accessed 23 Feb 2025
25. Microsoft: Data seeding. https://learn.microsoft.com/en-us/ef/core/modeling/data-seeding, http://leanpub.com/software-architecture-for-developers. Accessed 24 Oct 2023
26. Paduraru, C., Melemciuc, M.C.: An automatic test data generation tool using machine learning. In: ICSOFT, pp. 472–481 (01 2018). https://doi.org/10.5220/0006836604720481
27. Project, S.: Selenium: Browser automation framework (2024). https://www.selenium.dev/. Accessed Nov 13 2024
28. Ramírez, S.: Fastapi (2025). https://fastapi.tiangolo.com
29. Reid, M., et al: Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. arXiv preprint arXiv:2403.05530 **abs/2403.05530** (2024). https://arxiv.org/abs/2403.05530
30. Saxena, D., Cao, J.: Generative adversarial networks (gans): challenges, solutions, and future directions. ACM Comput. Surv. **54**(3) (May 2021). https://doi.org/10.1145/3446374
31. Sofokleous, A.A., Andreou, A.S.: Automatic, evolutionary test data generation for dynamic software testing. J. Syst. Softw. **81**(11), 1883–1898 (2008) 10.1016/j.jss.2007.12.809, https://www.sciencedirect.com/science/article/pii/S0164121208000101
32. Sommerville, I.: Software Engineering, Global Edition. Pearson Education (2016). https://books.google.com.br/books?id=W_LjCwAAQBAJ
33. Tsai, C.Y., Taylor, G.W.: Deeprng: Towards deep reinforcement learning-assisted generative testing of software. ArXiv **abs/2201.12602** (2022). https://doi.org/10.48550/arXiv.2201.12602
34. Tsymbal, A.: The problem of concept drift: definitions and related work. Computer Science Department, Trinity College Dublin **106** (2004)
35. Vaswani, A., et al.: Attention is all you need. In: Guyon, I., et al. (eds.) Advances in Neural Information Processing Systems. vol. 30. Curran Associates, Inc. (2017)
36. Xie, X., Liu, H., Hou, W., Huang, H.: A brief survey of vector databases. In: 2023 9th International Conference on Big Data and Information Analytics (BigDIA), pp. 364–371. IEEE (2023)