

A Methodology for Evaluating RAG Systems: A Case Study On Configuration Dependency Validation

SEBASTIAN SIMON, Leipzig University, Germany

ALINA MAILACH, Leipzig University, ScaDS.AI Dresden/Leipzig, Germany

JOHANNES DORN, Leipzig University, Germany

NORBERT SIEGMUND, Leipzig University, ScaDS.AI Dresden/Leipzig, Germany

Retrieval-augmented generation (RAG) is an umbrella of different components, design decisions, and domain-specific adaptations to enhance the capabilities of large language models and counter their limitations regarding hallucination and outdated and missing knowledge. Since it is unclear which design decisions lead to a satisfactory performance, developing RAG systems is often experimental and needs to follow a systematic and sound methodology to gain sound and reliable results. However, there is currently no generally accepted methodology for RAG evaluation despite a growing interest in this technology.

In this paper, we propose a first blueprint of a methodology for a sound and reliable evaluation of RAG systems and demonstrate its applicability on a real-world software engineering research task: the validation of configuration dependencies across software technologies.

In summary, we make two novel contributions: (i) A novel, reusable methodological design for evaluating RAG systems, including a demonstration that represents a guideline, and (ii) a RAG system, which has been developed following this methodology, that achieves the highest accuracy in the field of dependency validation. For the blueprint's demonstration, the key insights are the crucial role of choosing appropriate baselines and metrics, the necessity for systematic RAG refinements derived from qualitative failure analysis, as well as the reporting practices of key design decision to foster replication and evaluation.

CCS Concepts: • Information systems → Information retrieval; • Software and its engineering;

Additional Key Words and Phrases: Retrieval-Augmented Generation, Large Language Models, Configuration Dependencies

1 INTRODUCTION

Retrieval-augmented generation (RAG) is an emerging novel approach, driven by recent advancements in large language models (LLMs), which have enhanced their capabilities in various software engineering tasks, such as code generation [6, 49], configuration validation [26], database tuning [16, 25], and program repair [10, 22, 48]. Despite these advancements, LLMs still face challenges, such as hallucination [20, 58], outdated data [45, 57], and untraceable reasoning [14, 23], which undermine their ability to provide accurate and reliable outputs. RAG offers a promising solution to these challenges by integrating techniques from information retrieval with the generative capabilities of LLMs. This enables the delivery of contextually relevant, up-to-date, and factually accurate information to an LLM, effectively mitigating its limitations.

Typically, a RAG system comprises of an ingestion part where data is preprocess, embedded, and stored as context documents in a vector database including meta-data, a retrieval part where context documents are retrieved and ranked according to their relevance to a query, and a query part where a prompt with a query is combined with the retrieval results and sent to the LLM. So, RAG is not a single technology, but an umbrella of different components, design decisions, and domain-specific adaptations. It can be composed from different embedding models, search types, filters, and re-rankers. Developers must define which data sources are relevant and how

Authors' addresses: Sebastian Simon, Leipzig University, Germany, sebastian.simon@cs.uni-leipzig.de; Alina Mailach, Leipzig University, ScaDS.AI Dresden/Leipzig, Leipzig, Germany, alina.mailach@cs.uni-leipzig.de; Johannes Dorn, Leipzig University, Leipzig, Germany, johannes.dorn@cs.uni-leipzig.de; Norbert Siegmund, Leipzig University, ScaDS.AI Dresden/Leipzig, Leipzig, Germany, norbert.siegmund@cs.uni-leipzig.de.

to preprocess them prior to ingestion into a vector database. Moreover, how to prompt the LLM, how to rank which contextual information is most beneficial, and whether and how to provide the LLM with examples to the query at hand, often requires multiple evaluation and refinement steps [2]. The final RAG system is, thus, a product of several design decisions and iterations based on empirical evidence about the RAG’s effectiveness collected on the way.

Such an iterative, and empirical-driven development process requires a sound research methodology when a RAG system is not only built but also evaluated in a sound way. Without a proper methodology, we may end up in missing opportunities (e.g., due to unconsidered RAG variants), unsound results (e.g., by not being able to causally trace evaluation outcomes to individual RAG design decision), or non-generalizable claims (e.g., by overfitting the RAG adaptions in the iterative development process to the used benchmark). With such a heterogeneity and number of aspects, it can be difficult to not miss important considerations in the study design and evaluation as we will describe in Section 2.2. Although a generally accepted methodology for RAG evaluation clearly calls for a community endeavor, we aim to propose a first blueprint of how such a methodology may look like and apply it on a real-world software engineering research task.

With this paper, we make two novel contributions: (i) a validated blueprint of a reusable empirical study design for evaluating RAG systems in software engineering, and (ii) a RAG system that accurately validates configuration dependencies. For (i), the blueprint outlines several empirical key considerations that have to be considered and reported when evaluating RAG, including considerations for independent (context sources, RAG variants) and dependent variables (metrics) variables, as well as aspects of evaluation (baseline, benchmark) and refinement (failure analysis and improvements). This way, we guide researchers and practitioners alike in building a methodological sound study design to evaluate RAG systems. We demonstrate the application and validity of our blueprint through an empirical study on configuration dependency validation. For (ii), we propose a RAG system (as part of the demonstration) for the specific scenario of dependency validation. Validating configuration dependencies is an important, non-trivial task in practice that requires substantial contextual information due to its project- and technology-dependent variants. Thus, it represents an ideal demonstrator for how to evaluate and conduct experiments with RAG systems and the proposed system is valuable on its own.

These two contributions go hand in hand: It is unsound to propose a blueprint of a methodology without a *real* case of evaluating it. So, rather than using a toy example, we chose an application scenario, where the introduction of a RAG system can make an actual contribution. In other words, the contribution in the area of configuration dependency validation is a significant byproduct of the blueprint’s demonstration. Our reporting on the study design, as well as on the obtained results will therefore take both perspectives. Combined, this serves also as a guide for using the proposed blueprint.

The main result is that by following the blueprint, we were able to systematically develop and evaluate a RAG system that achieves the highest accuracy in the field of configuration dependency validation (e.g., compared to [26] on a similar task). A key insight of our blueprint is the importance of selecting the right baseline and metrics to determine whether the introduction of RAG is truly beneficial for a given task, because we found in our study that baselines can surpass RAG systems. Moreover, it emphasizes the need for systematic refinements (e.g., derived from failure analysis) to unlock RAG’s full potential as we found significant improvements when refining the RAG based on a qualitative failure analysis. Finally, the blueprint ensures that key design decisions are reported, facilitating better replication, evaluation, and identification of best practices and common failure points in RAG systems.

In summary, we make the following contributions:

- A blueprint for a sound empirical study design to evaluate RAG systems;
- A demonstration of how this study design can be applied;
- A highly accurate RAG system for configuration dependency validation.
- A replication package containing all code scripts, datasets, prompt templates, and validation results (see Data Availability section for details).

2 SCENARIO AND RELATED WORK

This section introduces configuration dependency validation, reviews existing approaches, and identifies real-world challenges where a RAG system could be beneficial. We then present related work on RAG for software engineering tasks, RAG system evaluation, and prompt engineering.

2.1 Scenario: Configuration Dependency Validation

Modern software development requires coordinating a wide range of technologies, such as code frameworks, build tools, and databases [40]. This coordination usually revolves around configuring each technology in its own format. For instance, container configurations are defined in a Dockerfile, whereas Java application options are set in a application.yml. These configuration files often encode hundreds of configuration options in their own structure, syntax, and semantic [42]. Most of these diverse technologies' configurations need to be harmonized to ensure their correct interplay and interoperability. In the exemplary Spring Boot application scenario shown in Figure 1, the port configurations in the application.yml in Listing 1 and in the Dockerfile in Listing 2 must match for the application to work. This constraint constitutes a *configuration dependency*. Such dependencies can exist both within a single technology (*intra-technology*) and across multiple technologies (*cross-technology*).

```

1 server:
2   port: 8080
3 spring:
4   application:
5     name: app

```

Listing 1. Spring Boot's application.yml specifying the port of the web server.

```

1 FROM openjdk:11-jdk-slim
2 ARG JAR_FILE=target/*.jar
3 ADD ${JAR_FILE} app.jar
4 EXPOSE 8080
5 CMD ["java", "-jar", "/app.jar"]

```

Listing 2. A Dockerfile exposing the port of the web server.

Fig. 1. An exemplary cross-technology configuration dependency between Spring Boot and Docker, both specifying the port of the Web server.

Naturally, developers of these technologies cannot meticulously manage all the frameworks and tools that potentially interact in their technology stack. Consequently, configuration dependencies are rarely fully documented [7] and the corresponding documentations are commonly incomplete and outdated [18, 21, 33]. This lack of a comprehensive overview about how and where configuration dependencies manifest within a software project poses several challenges in practice: (1) the ever-evolving configuration landscape of software projects bears the risk of introducing severe misconfigurations due to the violation of dependencies [29, 30, 44]; (2) misconfigurations may go unnoticed into production due to the interplay of multiple technologies that manifest themselves only in combination; (3) resulting configuration errors are complex and far-reaching since several technologies and diverse configuration artifacts are involved [51, 53]; (4) detecting and resolving these misconfigurations is often more challenging than fixing software bugs, making misconfigurations one of the most common causes of software failures in production today [13, 31, 34].

Several dependency-detection approaches have been developed, primarily leveraging heuristics [7, 8, 43, 50], domain-specific languages [3, 15], or machine learning techniques [38, 39, 55]. However, the detection process of configuration dependencies is prone to producing many false positives by incorrectly linking configuration options that are identical only coincidentally, but, in fact, independent from one another. This significantly impairs the practicality of dependency-detection approaches, because developers need to be able to trust the accuracy of warnings and errors produced by tools to support them. The goal of our proposed RAG system is to overcome the challenges of false positives by providing accurate classification capabilities in a technology-agnostic way. So, configuration validation represents a prime example of a complex software engineering task that could be improved by RAG and we will use it to demonstrate how RAG systems can be systematically evaluated.

2.2 Related Work

Next, we discuss existing approaches that use a RAG system for software engineering task and that are concerned with the evaluation of RAG systems, including prompt engineering techniques.

RAG for Software Engineering Tasks. RAG is on the brink to become a major technique for solving software engineering tasks. This is reflected in the high number of papers and pre-prints that are just being released. Although partially not peer reviewed, at least, we can derive what topics are addressed and for which use cases RAG provides a potential improvement. For instance, despite the wide adoption of LLM-based tools to generate code [32], they frequently hallucinate APIs that do not exist and lead to errors [27]. Jain et al. [19] substantially reduce hallucination by providing the LLM with additional documentation context. Improving LLM generations using retrieved context has also been proposed to improve commit message generation [41, 52, 56], test case generation [28], and to establish traceability between natural language requirements and code artifacts [1].

Although these studies present advances achieved by leveraging RAG, they do not follow a standard methodology or terminology: Some vary different components of the RAG system and some vary the LLM that is used for generation. Similarly, a wide range of metrics is used to measure effectiveness, ranging from task-specific metrics to general NLP metrics. Many papers omit details on iterative system refinement. Some mention experimental development phases (e.g., prompt improvement [1] or RAG architecture configuration [28]) while others fail to report systematic approaches or measures to ensure result validity. As there are numerous design decisions within a RAG, it is hard to decide upfront what factors to vary and so they become naturally part of the study. However, when this is not systematically done and properly reported, validity and replicability is threatened. Our work aims at supporting researchers with this decision process by providing a blueprint of a methodology on how to conduct a sound empirical evaluation of a RAG system.

Evaluating RAG. Evaluating RAG systems, particularly their retrieval and generation components, has been addressed in several recent studies. For example, Yu et al. [54] provide a comprehensive overview about the evaluation and benchmarks of RAG systems, highlighting challenges in evaluating RAG systems and presenting existing quantifiable metrics of the retrieval and generation components. Es et al. [9] propose RAGAS, a framework introducing a suite of metrics to evaluate the different dimensions in a RAG pipeline, with an emphasis on retrieval and generation metrics, such as context or answer relevance. Salemi and Zamani [37] focus on evaluating retrievers in RAG systems. They propose eRAG, an evaluation approach for retrievers, where the LLM in the RAG system is applied on each document in the retrieval list and set-based or ranking metrics are used as aggregation function to obtain an evaluation score for each retrieval list. Their approach significantly offers computational advantages compared to end-to-end evaluations. Furthermore,

Chen et al. [5] conduct a comprehensive evaluation of RAG for current LLMs. They introduce Retrieval-Augmented Generation Benchmark (RGB), a new corpus for RAG evaluation on abilities of LLMs, such as noise robustness, negative rejection, information integration, and counterfactual reasoning. While all these approaches provide valuable solutions to the evaluation of RAG systems, particularly their retrieval and generation components, they do not help researchers with designing sound experiments. That is, they are concerned with the RAG components rather than how to address a research question in a sound way. We propose a blueprint of a methodology on how to conduct a sound empirical evaluation of RAG systems, supporting developers and researchers alike making sound and reasonable decisions.

Prompt Engineering. Prompts are specific instructions provided to an LLM to generate desired responses from it. Empirical investigations have shown that providing different prompts to an LLM has a significant impact on its performance [24, 60]. That is, ad-hoc developed prompts do not consistently lead to desired results, while carefully crafted prompts can unleash the vast capabilities of an LLM to deliver remarkable results. Systematically crafting and designing the best prompt to accomplish a given task is known as *prompt engineering* [36]. The probably most popular and widely used prompting strategies are zero-shot [35], few-shot [4] and chain-of-thought prompting [47]. Zero-shot prompting relies on carefully crafted prompts that guide an LLM towards a specific task while few-shot prompting involves the introduction of few input-output example of the given task to demonstrate the LLM how to solve a task. Prompt engineering is thus also a crucial part of a RAG system to improve its output quality [59].

3 A REUSABLE METHODOLOGY FOR RAG EVALUATION

In this section, we present a blueprint of a sound empirical study design for evaluating RAG systems. Following, we describe the key considerations of our blueprint, which have to be discussed and reported when evaluating RAG. We not only propose the blueprint, but also provide a concrete demonstration on how to implement and report decisions in the remainder of the paper.

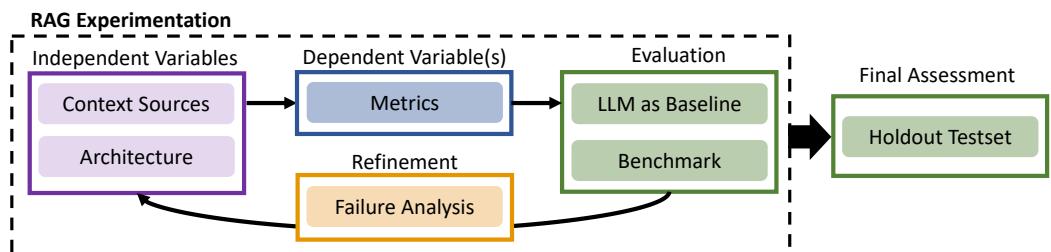


Fig. 2. Key considerations for sound empirical evaluations and refinements of RAG systems

Figure 2 highlights the key aspects to consider when conducting an empirical evaluation. We use the empirical terminology of dependent and independent variables to define the factors influencing the evaluation process and outcomes. For each consideration, we will now provide guiding questions and discussions.

Independent variables: What do we vary?

Context resources. Providing different context resources is the key difference between a RAG system and a vanilla LLM as they represent the factual and timely data on which an LLM can reason on instead of only using pretrained weights based on the training data. It is, thus, crucial to select resources that are relevant to the task at hand, which usually requires human judgment. Moreover,

as different resources, such as Web searches, database entries, user feedback, and historical data, all can contribute differently to the output accuracy, it is important to evaluate their individual and combined contribution to the independent variables.

Architecture. RAG is an umbrella of different components, design decisions, and domain-specific adaption. For example, there are hundreds of different embedding models to encode the semantics of textual information for later search [46], there are multiple ways for searching relevant documents in a multitude of vector stores [17], and there are plenty of ways on how to preprocess information (e.g., splitting documents), as well as how to filter and re-rank retrieved documents [12]. All these *architectural* decisions in a RAG system can influence result accuracy and therefore need to be considered in a sound empirical study. As discussed in Section 2.2, there is a current lack in reporting on design decisions, as well as improvements and refinements on them. Whether a proposed RAG system is a first, naive version or a carefully crafted one based on multiple iterations and feedback runs remains unclear, but is important on estimating the (un)tapped potential of the RAG system.

Dependent variables: What and how to measure?

Metrics. Every study begins with a research question, which then must be operationalized. If the goal is to assess the *effectiveness* of a RAG system for a specific task, this is often translated into task-specific metrics. For classification, metrics, such as precision or recall may be suitable, whereas generation tasks typically lack straightforward measures. RAG-specific metrics, such as context relevancy, assess how relevant retrieved documents are to the generation task. Although more relevant documents should improve RAG effectiveness, it is essential to empirically verify that a metric correlates with the desired outcome. So, RAG-specific metrics may be a part of the experiment, but due to their independence to the task, are usually insufficient to answer the research questions. Instead, defining metrics that truly reflect the system’s performance on the intended task should be the main priority.

Evaluation: How to evaluate the RAG system?

Baseline. A *baseline* provides a point of reference to determine whether the introduction of a RAG system is beneficial at all. Research has shown that simple machine learning baselines, when handled correctly, often outperform more complex models [11]. While creating a baseline can sometimes be challenging, it is crucial to ensure that system improvements genuinely enhance effectiveness and that unnecessary complexity is avoided when simpler approaches yield similar results. For RAG systems, using a vanilla LLM as a baseline is often appropriate, but the baseline must be refined alongside the RAG system for a fair comparison (e.g., if context-independent prompting of the RAG system is improved, the baseline prompting has to be improved as well). An analogy is hyperparameter tuning. When tuning only the proposed approach, but not the baseline, it remains unclear whether differences are due to the tuning or the approach itself. In cases where retrieval is essential, a basic RAG version can serve as the baseline instead of a vanilla LLM. One final aspect is the need to report the version number of an LLM. When only the architecture, such as GPT-4 or GPT-3.5 is reported, but not the version, we will not be able to reproduce the results as updates on the models change substantially their behavior and quality.

Benchmark. To compute the metrics for the dependent variable, both the RAG system and the baseline must be evaluated on a *benchmark*. A benchmark provides a standardized basis for comparing different approaches. For well-researched use cases, established benchmarks may already exist. In other cases, it may be necessary to create or adapt a benchmark. To allow future researchers to compare their approaches to the presented one, a benchmark should be well-documented and

publicly available. This ensures transparency and facilitates reproducibility in future work. Of course, this is not always possible. So, at least, the evaluation data set should be made public.

Refinements: How to systematically refine the RAG system?

Failure Analysis. A RAG system may benefit from several iterations on the task evaluation, in which each iteration may come with a *refinement* of all RAG components, their data sources, and used prompt templates. To gain meaningful insights into how to refine the system and identify the most promising changes, analyzing failure cases is essential. Depending on the use case, errors may require different analytical approaches. For classification problems, performance across classes can provide valuable insights, but in most cases, manual qualitative inspection will be necessary for deeper understanding. This analysis can be guided by questions such as: Is the context properly utilized? What additional information would the LLM need to solve certain cases successfully? Are the tasks clearly specified for the LLM? Such a failure analysis should be properly reported in the study to enable traceability of RAG decisions. As said before, it is important to apply changes (if possible) to the vanilla LLM as well.

Final Assessment: How to ensure validity and soundness after refinement and re-evaluation?

Holdout Testset. Each refinement requires an evaluation again. The benchmark or evaluation set from the first evaluation are likely candidates, but since the failure analysis has been conducted on the same data, we risk overfitting the system to this dataset. Then it becomes unclear whether observed improvements generalize to new data, essentially limiting generalizability and threatening external validity. Maintaining a separate *holdout testset* serves as an unseen, labeled dataset that helps to assess the effect of the *final* RAG system. As soon as this holdout testset is used and a refinement follows, we have to create a new testset. The final reporting about the RAG's performance should be mainly focused on the results of this testset.

4 DEMONSTRATION OF EVALUATING RAG FOR CONFIGURATION DEPENDENCY VALIDATION

In this section, we introduce an approach to leverage RAG for configuration validation and demonstrate how our reusable research methodology can be applied to evaluate this approach.

4.1 Configuration Dependency Validation with RAG

We propose a novel RAG system to validate configuration dependencies. In the following section, we describe its architecture from indexing data as context into a vector database to retrieving relevant context from it, and querying an LLM with an augmented prompt.

Context Ingestion. We populate a vector database with *static* and *dynamic* context information. *Static* context information is stored prior to the evaluation queries. *Dynamic* context information is dynamically extracted, processed, and indexed into the vector database on-the-fly for each dependency during validation.

The indexing process starts with transforming the content of *static* and *dynamic* context sources into documents, which are generic containers that stores text along with metadata. Technically, we use a data loader from llamaindex¹. The resulting documents are then split into smaller chunks with a certain amount of overlap. We choose a chunk size of 512 with an overlap of 50 in order to encompass all necessary information in the retrieved chunks. After splitting, the resulting chunks

¹<https://www.llamaindex.ai/>

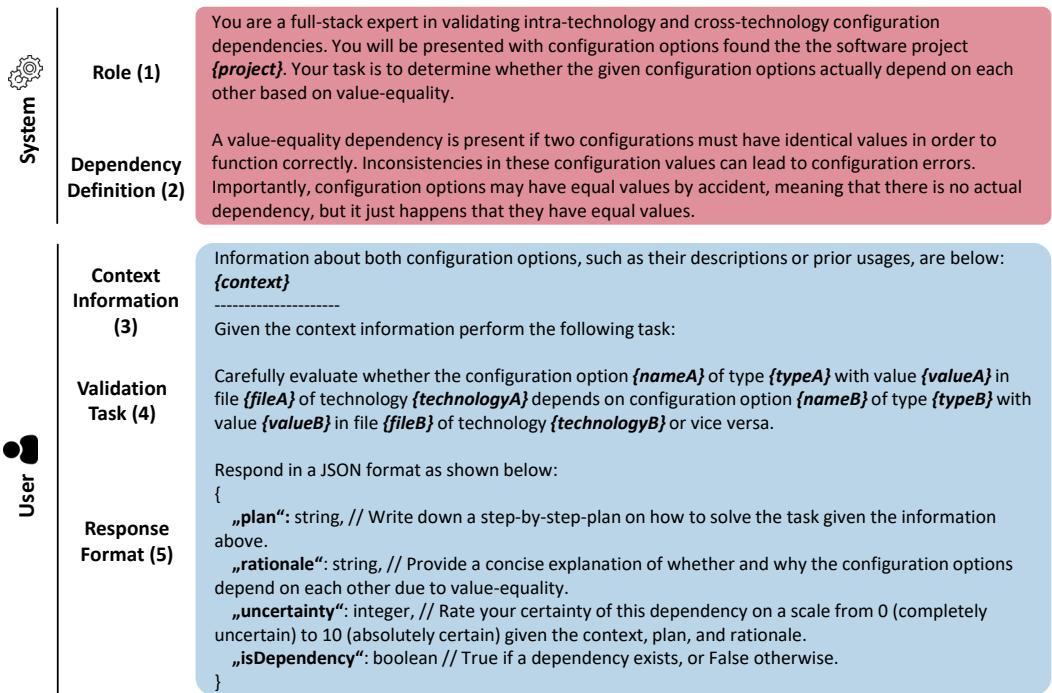


Fig. 3. The prompt template and its components used for dependency validation.

are then converted into embeddings, a vector representation of the documents, using an embedding model. Finally, the resulting embeddings are indexed in the vector database Pinecone².

Context Retrieval. We incorporate several retrieval strategies to enhance retrieval quality of our RAG system. First, we optimize the original query by rewriting. We then perform a hybrid search in the vector database with the retrieval query to obtain an initial set of contextual information. Hybrid search combines semantic search with keyword search in one query, which allows us to find context information based on configuration option names via keyword search and relevant context of dependencies via semantic search. After obtaining a large initial set of context information, we re-rank the elements of this set according to their relevance. Finally, we use the re-ranked contextual information to augment the prompt for dependency validation.

Validation Generation. We created a prompt for dependency validation using a collective prompt design process, in which all authors discussed and adjusted the prompt and its components in multiple iterations. Our final prompt for dependency validation includes five elements: (1) a system message, which defines the role of the LLM, (2) a definition of the dependency that it has to validate, (3) the retrieved context information, (4) the actual instruction to validate the given dependency, and (5) a description of the JSON format in which the LLM should respond. The validation prompt and its components are shown in Figure 3.

The JSON response format prompts the LLM to fill in four primary fields: (1) *plan*: a string that describes the plan on how to validate the dependency based on the contextual information step-by-step; (2) *rationale*: a string that explains whether and why the configuration options depend on

²<https://www.pinecone.io/>

each other or not; (3) *uncertainty*: an integer from 0 (completely uncertain) to 10 (absolutely certain) indicating the LLM’s uncertainty about the dependency based on the context information; and (4) *isDependency*: a boolean indicating the final decision of the LLM whether the given dependency is a true dependency or a false positive.

The proposed response structure follows the idea of chain-of-thought prompting [47], aiming to increase the understanding and analytical capabilities of LLMs in validating configuration dependencies. That is, we prompt the LLM to lay out a step-by-step strategy on how to validate the given dependency and to provide an explanation and rating of its uncertainty of before making the final decision. By breaking down the validation task into several steps, each building on the previous one, we ensure a logical progression of actions that the LLM should perform.

4.2 Research Questions

To evaluate the effectiveness of the proposed RAG system and identify key design decisions that enhance dependency validation, we conduct an experimental study. This study aims to address the following two research questions:

- **RQ₁**: How effective are vanilla LLMs and unrefined RAG in validating configuration dependencies?
- **RQ₂**: What validation failures occur with vanilla LLMs and unrefined RAG and what refinements can be derived from that?

By asking and answering these questions, we contribute to advancements in configuration dependency validation while also completing a full cycle of a RAG evaluation and refinement, demonstrating the applicability of our proposed blueprint.

4.3 Applying the Blueprint: Evaluating RAG for Configuration Dependency Validation

By using our blueprint, we can give a concise overview in Figure 4 of how we addressed several design choices for conducting the evaluation. We will now discuss each decision in detail.

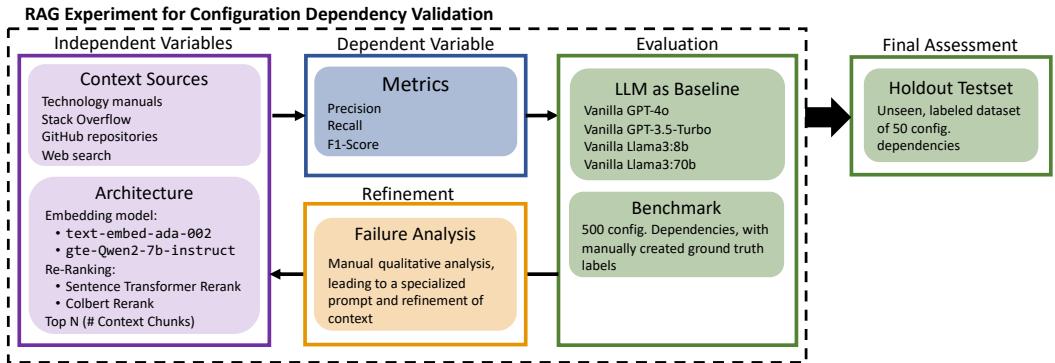


Fig. 4. Study design decisions based on the blueprint for evaluating RAG for configuration dependency validation.

4.3.1 Independent variables.

Context Sources. Data on configuration dependencies is rarely available and usually not explicitly stated, which makes deciding on context sources especially challenging. For example, manuals typically describe configuration options in terms of their names, abbreviations, functionality, and (rarely) their allowed parameter ranges. But the description almost always lacks explicit dependency

information to other options of that technology and entirely to other technologies. Hence, there is no clear source for reliable information on configuration dependencies. Developers often turn to the man pages, Stack Overflow, or Web searches that provide real-world use cases of configuration options in which concrete dependencies are explicitly or implicitly discussed. Furthermore, project-specific information, such as project and environment settings in which the configuration resides, may be found in the corresponding project’s repository. Based on these considerations, we decide to collect information from the same sources that also developers use, that is, man pages, Stack Overflow posts, GitHub repositories, and Web search results.

RAG Architecture. We vary between several core elements, such as the embedding model (ada2 vs. Qwen2), the re-ranking algorithm (Colbert vs. Sentence Transformer), and the final number of context chunks provided to the LLM (3 vs. 5). By varying only one component at a time, we obtain four different RAG variants, shown in Table 1. Note, that we clearly could manipulate many more aspects of the RAG pipeline. However, this is the first and unrefined setup with which we start the evaluation, these configurations might be refined during experimentation if the failure analysis reveals significant room for improvement.

4.3.2 Dependent Variable.

Metrics. The dependent variable in our demonstration is the effectiveness of the approach in validating configuration dependencies. We measure effectiveness using traditional classification metrics: precision (1a), recall (1b), and F1-score (1c). Precision represents the proportion of true dependencies (TP) among positively labeled ones (TP + FP), while recall reflects the proportion of true dependencies among all actual dependencies (TP + FN). These metrics highlight the trade-off between identifying as many true dependencies as possible (recall) and ensuring the correctness of positively labeled dependencies (precision). High precision and recall are crucial for trustworthy validation. The F1-score, the harmonic mean of precision and recall, provides a single scalar value while allowing for more detailed analysis when examining each metric individually.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (1a)$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (1b)$$

$$\text{F1} = \frac{2 \cdot (\text{Precision} \cdot \text{Recall})}{(\text{Precision} + \text{Recall})} \quad (1c)$$

4.3.3 Evaluation.

Metrics. To assess the effectiveness of our RAG variants for dependency validation, we compare their validation abilities against vanilla LLMs. Throughout our empirical study, we leverage four state-of-the-art LLMs, including two proprietary LLMs developed by OpenAI³ (i.e., GPT-4o and GPT-3.5-Turbo) and two open-source LLMs developed by Meta⁴ (i.e., LLama3:8b and LLama3:70b (non-quantized)). The open-source LLMs are executed on our own hardware, a MacBook Pro with an Apple M3 chip and 128GB of RAM. A summary of the studied LLMs with their properties can be found in Table 2. For all LLMs, we set the temperature to 0 in order to improve determinism and consistency of LLM outputs. Note that the LLMs play a second role here. They are not only baselines, but also independent variables as we vary and compare them (to each other).

³<https://openai.com/>

⁴<https://llama.meta.com/llama3/>

Table 1. Variants of RAG used in the first experiment with text-embed-ada-002 (ada2) and gte-Qwen2-7B-instruct (Qwen2).

ID	Embed. Model	Embed. Dim.	Reranking	Top N
1	ada2	1536	Colbert	5
2	Qwen2	3584	Colbert	5
3	Qwen2	3584	Sentence Transf.	5
4	Qwen2	3584	Colbert	3

Benchmark. To evaluate the effectiveness of our approach in validating configuration dependencies, we collected a dataset of potential configuration dependencies. We ran *CfgNet*, an existing dependency detection framework from the literature [43], on ten real-world software projects. The selected projects used at least the following technologies: Spring Boot, Maven, Docker, and Docker Compose, and were among the most starred on GitHub. An overview of the systems is provided in Table 3. For each software project, we sampled 50 potential dependencies, leading to a final set of 500 candidates. The rational for this sampling is that all dependencies must be manually checked in detail to create a trustworthy and correct ground truth. We manually labeled each dependency as either *true* if it correctly represents a dependency, or *false* if it is a false positive. This way, we created the largest ground truth cross-technology configuration dependency dataset that we are aware of and that has been used in any study. As a notable contribution of this work, we make this dataset available for other researchers and tool developers.

4.3.4 RAG Refinement.

Failure Analysis. We conducted a qualitative analysis of the validation failures of vanilla LLMs and the best-performing RAG variant to identify failure patterns. Specifically, we inspected all validation failures by extracting the involved configuration options, the retrieved context, and the reasoning of the LLM. For example, we reviewed the options' names, their values, and the involved technologies. Then, we checked whether the reasoning of the LLM was consistent with its final (wrong) assessment. We put this reasoning with our own manual correct classification in relation to find, for instance, whether additional information was not available to the LLM to make a correct judgment or whether false information, noise, or ambiguities caused the failure. We then categorize failures by similar mistakes or causes of mistakes. The following refinement step used these categorizes to implement measures for avoiding the failure causes.

4.3.5 Final Assessment.

Holdout Testset. In addition to the benchmark consisting of 500 dependency candidates, we created a holdout testset of 50 potential dependencies by randomly sampling configuration dependencies from our subject systems and manually reviewing them similar to how we labeled the dependencies of the baseline. The final assessment of whether our proposed RAG system is able to accurately validate configuration dependencies is conducted and reported on this testset.

5 RESULTS

This section presents the results of our empirical study, which has two goals: First, we demonstrate the applicability and validity of our RAG methodology. Second, we evaluate the effectiveness of RAG and vanilla LLMs on the specific scenario of dependency validation, as an independent

Table 2. Overview of studied LLMs

Model Name	# Param (B)	Context Length	Open Source
gpt-4o-2024-05-13	-	128k	✗
gpt-3.5-turbo-0125	175	16k	✗
llama3:70b	70	8k	✓
llama3:8b	8	8k	✓

Table 3. Subject Systems with Version (SHA), Number of Stars (★), Number of Commits (Com.)

GitHub Project	Version	★	Com.
macrozheng/mall	70a226f	76.6k	1k
apolloconfig/apollo	49bd8cc	28.9k	2.8k
linlinjava/litemall	92ffc39	19k	1.2k
sqshq/piggymetrics	6bb2cf9	13.1k	290
codecentric/spring-boot-admin	60be5d1	12.2k	3.1k
macrozheng/mall-swarm	fd5246c	11.5k	463
wxiaoqi/Sping-Cloud-Platform	9aad435	6.3k	358k
pig-mesh/pig	8e10249	5.7k	1.6k
jetlinks/jetlinks-community	1ad1e44	5.2k	1.2k
Yin-Hongwei/music-website	12e1b0a	5.1k	385k

contribution on its own. Results are structured by the scenario’s research questions. We discuss the implications and insights for our proposed methodology alongside.

5.1 RQ1: Validation Effectiveness of Vanilla LLMs and unrefined RAG on Dependency Validation

We validated 500 manually labeled configuration dependencies of a cross-technology stack from ten real-world software projects with four state-of-the-art LLMs. Table 4 shows the number of validation failures (#Failures) as well as precision, recall, and F1-score for each vanilla LLM and unrefined RAG variant. Column “RAG ID” maps to the individual RAG systems given in Table 1. Vanilla LLMs without RAG are marked with “w/o”.

Overall, our results show solid validation capabilities of vanilla LLMs for configuration dependencies, indicated by F1-scores ranging from 0.59 to 0.73, with *GPT-4o* outperforming smaller models. When having a closer look, results are not consistent: *GPT-4o* and *Llama3:70b* have higher precision than recall scores, which indicates that these models generally perform well in correctly identifying valid and invalid dependencies, but simultaneously miss an important portion of dependencies. By contrast, *GPT-3.5-turbo* and *Llama3:8b* have a high coverage in identifying valid dependencies, but produce more false positives. Although the smallest model *Llama3:8b* performs best in recalling dependencies, it comes at the heavy cost of low precision. Thus, neither the vanilla LLMs nor the unrefined RAG variants provide a satisfying recall and precision.

Given the unrefined RAG systems that we implemented, our results indicate that adding additional contextual information does generally not improve the validation abilities of LLMs. On contrary, in almost all cases, vanilla LLMs slightly outperform the unrefined RAG variants with the same underlying LLMs in terms of F1-score, except for *Llama3:70b*. Specifically, *Llama3:70b* is the only LLM that shows slightly higher F1-scores (improvements range from 0.06 to 0.09) in all RAG variants than its vanilla version. However, these improvements are only marginal and are likely not meaningful. This trend across all unrefined RAG variants is a strong indicator that even plausible context information may be unhelpful for this task. Moreover, it demonstrate that this task is quite challenging as factual relevant information seem to be not explicitly available such that the context may increase noise for an LLM. In summary, we observe that the unrefined RAG systems have a small effect on all LLMs, except *GPT-4o*: they improve recall scores, meaning the models now identify more dependencies than before, but decreases precision, meaning the models now produce more false positives. For *GPT-4o* there are no improvements, as all scores decrease.

Table 4. Validation effectiveness of vanilla LLMs GPT-4o (4o), GPT-3.5-Turbo (3.5T), Llama3:8b (L3:8b), Llama3:70b (L3:70b), and different RAG variants on 500 configuration dependencies extracted from ten real-world software projects.

RAG ID	LLM	#Failures	Precision	Recall	F1-Score
w/o	4o	91	0.89	0.62	0.73
	3.5T	157	0.59	0.67	0.63
	L3:70b	154	0.63	0.56	0.59
	L3:8b	159	0.55	0.84	0.65
	mean	140	0.67	0.67	0.65
1	4o	112	0.81	0.57	0.67
	3.5T	196	0.51	0.74	0.60
	L3:70b	160	0.57	0.76	0.66
	L3:8b	196	0.46	0.90	0.61
	mean	166	0.59	0.74	0.64
2	4o	116	0.79	0.57	0.66
	3.5T	185	0.52	0.78	0.63
	L3:70b	152	0.59	0.78	0.67
	L3:8b	189	0.48	0.90	0.62
	mean	160	0.60	0.76	0.65
3	gpt-40	120	0.78	0.55	0.64
	3.5T	189	0.52	0.76	0.62
	L3:70b	158	0.58	0.76	0.66
	L3:8b	203	0.45	0.88	0.60
	mean	165	0.58	0.74	0.63
4	4o	116	0.79	0.56	0.66
	3.5T	188	0.52	0.72	0.60
	L3:70b	166	0.56	0.73	0.64
	L3:8b	169	0.48	0.83	0.61
	mean	160	0.59	0.71	0.63

We vary the architecture of the RAG systems in different core components to test whether these configurations affect validation effectiveness. Overall, there are only minimal variations in the validation effectiveness of LLMs. We observe only a small pattern in the fourth unrefined RAG variant, in which we decreased the number of context chunks that are provided to the LLMs. The recall of *GPT-3.5-Turbo*, *Llama3:8b*, and *Llama3:70b* drops compared to all other RAG variants for which we provide more context chunks to the LLMs, showing that, if we provide context at all, providing more context leads to better performance.

5.1.1 Discussion RQ1. The validation ability of vanilla LLMs varies significantly. While vanilla LLMs can already validate some dependencies and non-dependencies accurately, a substantial number of falsely validated dependencies remains, depending especially on the size of the LLM. Unrefined RAG systems have, contrary to their praise, no or mostly negative effects on the validation abilities of all LLMs, suggesting that merely adding context information does not guarantee a better outcome. This in line with related work [19], supporting the observation that an unrefined RAG system (e.g., with supoptimal retrievers or context sources) can degrade the performance of LLMs. So, simply implementing RAG systems for a concrete software engineering task is insufficient, at least, when omitting a refinement step. Poor performance with an unrefined RAG system can stem from several factors: the retriever may fail to retrieve relevant context information, the selected context sources may lack necessary information, or the prompt could be too vague, leaving room for misinterpretations. All these potential sources of failures have to be considered when evaluating a RAG system. Therefore, a qualitative analysis of the produced failures by the RAG system is an essential part of the evaluations of RAG systems, as it sheds light not only on possible causes of failures but also helps to derive specific measures and relevant changes for effective refinements of the RAG system.

Moreover, we have not seen substantial changes among different RAG variants. Although we cannot rule out that our changes might be too insignificant, we expect that replacing the embedding model, changing the reranker and varying the context size affect important components and should have a recognizable effect on validation performance. It seems, however, that the choice of contextual information is substantially more important than the architecture itself such that refinements should target these aspects first, instead of trying more RAG variations.

Answer RQ1: Most vanilla LLMs exhibit better validation performance than our unrefined RAG systems with the same underlying model. Adding additional context does not automatically improve performance, particularly for more advanced LLMs. Overall, our results indicate that simply implementing an unrefined RAG system is not enough: systematically improving and refining different RAG components and context to reduce the number of failures might be key to effectively validating dependencies.

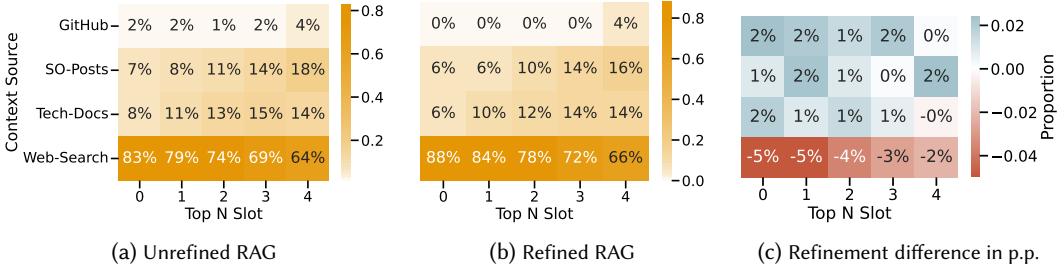


Fig. 5. Usage of context sources per context slot for all 500 dependencies

Reflection on the blueprint: Baseline, LLMs, and Context Sources

At this stage of the experiment, we can already reflect on three important aspects of the methodology: LLMs as baseline, RAG architecture, and context sources.

What if we had not compared the RAG system to the vanilla LLM baseline? The comparison to vanilla LLMs as baseline sets the RAG results into perspective. Looking only at different RAG variants, we could easily overestimating its impact without understanding its relative performance. Our real-world scenario prominently demonstrates that vanilla LLMs may even be superior to RAGs that include the very same LLMs. Thus, an assessment of vanilla LLMs seems essential.

What if only a single LLM would have been used? We see largely varying results not only among the vanilla LLMs, but also the RAG system. Also, we observe differences in LLMs depending on the RAG variant. For example, RAG variant 1 has a better recall for llama3:8b than RAG variant 4. So, even when these results are similar, we do observe interaction effects of LLMs and the RAG architecture, which can be more significant for other scenarios and, thus, should be evaluated.

What about context sources?

Our evaluation has not presented results differentiated according to different context sources. In our case, as the RAG variants were inferior to the vanilla LLMs, there was no benefit in evaluating which source has which effect. In general, however, when we observe improvements, it is important to trace causes of improvements to the contextual information. To demonstrate how such an evaluation gains for insights into the RAG mechanism, we highlight in Figure 5a the fraction of sources that the RAG has deemed relevant for the query and submitted to one of the five context slots. We see that most of the time, the results of the Web search are supplied to the LLM as context. Such insights represent means to evaluate the validity of the RAG system and to refine the system in an iteration.

5.2 RQ2: Validation Failures of Vanilla LLMs and unrefined RAG

We answer the research question on which failures do occur based on a qualitative analysis. We approach this by identifying common failure categories. To this end, we manually inspect every false classification of all vanilla LLMs and RAG variant 2, as it yield the highest validation score and lowest number of false classifications among the RAG systems (see Table 4 rows “ean”). In total, we reviewed 1 203 validation failures from which we derived eight failure categories. Table 5 depicts a summary of all categories and their distribution across the LLMs and RAG variant 2. We provide a detailed description for each category on our supplementary Web page. We see that some failures

are unique to specific LLM. For instance, while *GPT-4o* reliably validated dependencies in the *Port Mapping*, *Resource Sharing*, and *Independent Technologies/Services* category, all other LLMs still struggle with these dependency types. However, there are also failure categories in which all LLMs encounter difficulties, particularly in the categories *Inheritance and Overrides* and *Configuration Consistency*, which show the highest number of failures.

Table 5. Failure categories of vanilla LLMs GPT-4o (4o), GPT-3.5-Turbo (3.5T), Llama3:8b (L3:8b), Llama3:70b (L3:70b), and the best found RAG variant on dependency validation.

Failure Category	Vanilla LLMs				RAG ID 2			
	4o	3.5T	L3:8b	L3:70b	4o	3.5T	L3:8b	L3:70b
Inheritance and Overrides	44	52	26	50	49	42	30	36
Configuration Consistency	25	46	54	46	36	69	67	55
Resource Sharing	1	11	11	4	6	15	15	9
Ambiguous Options Values	1	3	11	4	2	8	12	7
Port Mapping	-	7	12	8	7	10	8	7
Context (Availability, Retrieval, Utilization)	17	9	4	11	15	3	6	18
Independent Technologies/Services	-	10	17	10	-	17	10	6
Others	3	19	24	21	1	21	41	14
Total	91	157	159	154	116	185	189	152

Across the failure categories, we developed specific RAG improvements (i.e., refinements) derived from the categories' characteristics. These refinements range from adding more context in the form of project-specific information, such as details about module organization and available resources over refined prompt customization to derived examples of correct and incorrect classification (i.e., few-shot prompting [4]). Providing project-specific information particularly may reduce failures in the categories *Inheritance and Overrides* and *Resource Sharing* as such information is needed to perform a correct validation. Notably, we found that improving the prompt to better specify different forms of dependencies may be helpful to avoid confusions of LLMs regarding the consistency of configuration values. So, this represents a refinement that is also applicable to the vanilla LLMs and so we evaluate with this refined baseline.

5.2.1 Refined RAG System and Baseline. We implemented all refinements derived from our qualitative failure analysis and applied them to the best-performing unrefined RAG system:

- We added project-specific details by extracting relevant information about the project, including the project structure, module organization, and implementation details.
- We collected exemplary dependencies alongside their correct classification and integrated them into the RAG system by adding the two most similar examples (based on cosine similarity) to the context of the LLM.
- We refined the validation prompt to provide a more precise definition of dependencies.

Note that we also applied the last refinement to the vanilla LLMs as this is clearly a measure, which can improve the baseline and is not RAG specific. We discuss this more in the reflections on the methodology. Next, we reran our evaluation on all prior failed dependencies using the refined vanilla LLMs and refined RAG variant 2. Not shown here, we found a substantial reduction of failures for all LLMs and the refined RAG. This rerun acts only as a sanity check since conducting an evaluation on the data from which we derived the refinements can clearly jeopardize the validity of the results. Thus, following our methodology, we assessed the final performance of both the refined vanilla LLMs and the refined RAG system on the unseen holdout set.

Table 6 presents the final validation scores for all evaluated models on the holdout set. There are two notable observations: (1) The small refined vanilla LLMs especially benefit from the improved prompting. We observe a *Llama3:8b* model that is now on-par with the most advanced refined model *GPT-4o* and even surpasses the unrefined *GPT-4o* model for the F1 score. (2) The refined RAG systems clearly outperform the refined vanilla LLM baseline in all cases. Most notably, one of the smaller open-source models, *Llama3:70b* now performs best compared to all other models with an F1 score of 0.89. Thus, both smaller open-source models combined with RAG are now a preferable option to the two large proprietary models.

Table 6. Validation effectiveness of vanilla LLMs GPT-4o (4o), GPT-3.5-Turbo (3.5T), Llama3:8b (L3:8b), Llama3:70b (L3:70b) and the refined RAG system with the same underlying LLMs on a holdout test set of 50 configuration dependencies.

RAG ID	LLM	Pre- cision	Recall	F1
w/o 2	4o	0.91	0.65	0.75
	3.5T	0.73	0.61	0.67
	L3:70b	0.75	0.58	0.65
	L3:8b	0.64	0.89	0.74
2	4o	0.95	0.68	0.79
	3.5T	0.79	0.84	0.81
	L3:70b	0.85	0.93	0.89
	L3:8b	0.80	0.77	0.79

5.2.2 Discussion of RQ2. With our refinement, we could clearly show that a RAG system is beneficial for configuration dependency validation. Although clarifying the task improves the usage of all LLMs (especially the smaller ones), it is the additional contextual information available in a RAG system that achieves the highest scores. Although beyond the focus of this work, a further refinement concentrating on the retrieval part seems promising to reach even higher F1-scores. Especially when looking at Figure 5c, we observe that the main context information is Web search, potentially leaving out other relevant information. Here, it makes sense to apply metrics, such as RAGAS [9] to find more failure categories that are concerned with retrieval relevance. Moreover, the failure analysis is a tedious manual process, which has to be taken into account when trying to automate the development of RAG systems for certain software engineering scenarios. An artifact is that the refinements could become very specific. For our case, this could mean to jeopardize the technology-independent part. Hence, we concentrate on making all refinements agnostic to the technology at hand, but this might not be possible in any case.

Answer RQ₂: Following our blueprint, we derived simple refinements from a qualitative failure analysis and integrated them into a refined RAG variant. The refined RAG variant achieved substantial improvements compared the refined vanilla LLMs, with smaller open-source LLMs especially benefiting from these refinements, making them a preferable option when combined with RAG. Thus, knowing the failure modes of RAG allows for strong refinements and should be done in any RAG development activity.

Reflection on the blueprint: Refinement and Assessment

After introducing the blueprint and demonstrating its application on a concrete software engineering task, we can reflect on the remaining aspects of the methodology: Refinement and holdout testset.

Why refining the baseline is necessary? If we had not refined the baseline, our RAG system might have shown even greater improvements compared to the not refined vanilla LLMs. However, we would not be able to distinguish whether these improvements are genuinely due to enhancements in RAG-specific components or simply the result of clarifying general information in the system prompt independent of having a RAG, hence, potentially threatening the validity of our conclusions.

What if we had not refined the RAG system? Knowing in advance which contextual information actually improves the RAG performance is hard. In addition, RAG is an umbrella of different components, design decisions, and domain-specific adaptation, all of which are possible source of failures. So, unrefined RAG systems are likely to leave their potential untapped. Therefore, conducting a thorough qualitative analysis of the failures helps to incorporate the actually relevant context and select the most effective components. Furthermore, we see it important to report on the failure modes and refinements explicitly as part of a research endeavor so that others can better follow, judge, contextual, and replicate the work. Moreover, this improves also our general understand of what are best practices of RAG refinements.

What if we had not assessed the final performance on a holdout set? If we had not tested the configuration validation system on a holdout set, then we would seriously threaten our external validity and generalizability. This is a rather typical scenario in a machine-learning setting for overfitting, but applies here also for the iterative refinement of RAG systems. For a practical use case, it is the testing (i.e., generalizing) error that matters. So, we need to treat our evaluation in the same way and be transparent in reporting.

5.3 Threats to Validity

The main threat to *internal validity* is potential bias in our manual labeling of configuration dependencies. We mitigated this through rigorous review processes involving multiple authors, especially for borderline cases. We further mitigated subjectivity in our qualitative analysis of failure categories by independent coding of multiple researchers.

Regarding *external validity*, our focus on software projects in the Spring Boot ecosystem may limit generalizability. Nevertheless, we selected a large range of real-world technologies that are widely used in practice. Moreover, we selected diverse, popular open-source projects as a benchmark to test on real use cases. The relatively small holdout test set (50 dependencies) may limit *conclusion validity* and statistical power. However, the results agree with those for the large benchmark set. So, we expect no changes in data distribution.

Reflection on the blueprint: Validity, Reliability, Objectvity

Objectivity: Objectivity may be threatened because the researchers who developed the blueprint are partially the researchers conducting the empirical study in the scenario. We mitigate this threat in two ways: First, we defined the methodology upfront and followed it accordingly. Second, we composed a team where one person was mainly concerned with the blueprint, one with the study on dependency validation, and one with both of them. This way, we counter a possibly too optimistic application of the blueprint.

Reliability: Although it needs more studies applying our blueprint to obtain a final assessment, we have two strong indications of the blueprint's reliability: First, we derived a RAG system that challenges the state of the art in its area with only a single round of refinement – a success on its own. Second, we reviewed upcoming papers and found similarities with the blueprint, but also methodological issues for which our blueprint can provide support.

Validity: The success of the RAG system for dependency validation might also be achieved when not following our proposed research methodology. Importantly, the blueprint's goal is not to enable research that produces successful results, but sound and reliable one. That is, although if we would have reached similar results, it is due to our methodology that the results are sound and transparently reported such that one knows what measures caused them. A threat questioning the generalizability of the blueprint is certainly the demonstration of only a single scenario. However, we made sure that this scenario has characteristics typical for the usage of RAG and is also a wide-spread use case in practice in contrast to a toy or artificial one. Additionally, our blueprint uses generic empirical concepts, which should be key aspects of any experimental evaluation and are broad enough to adapt to many scenarios. Hence, we expect the blueprint to generalize to similar scenarios without excluding the possibility of adaptations and additions in the future.

6 CONCLUSION

The development of RAG systems is driven by numerous components, design decisions, and domain-specific adaptations. Without a systematic evaluation and reporting, the validity and replicability of RAG approaches are threatened. Thus, we propose a blueprint of a methodology for a sound and reliable evaluation of RAG systems. We demonstrate its applicability on a real-world software engineering task: the validation of configuration dependencies across software technologies.

Following our blueprint, we systematically developed a RAG system that achieves the highest validation accuracy in the field of configuration dependency validation—an important research contribution on its own. At the same time, we could show how to apply the blueprint at different steps and reason on its validity. Overall, the blueprint achieved to provide sound and reliable results. It also enables a transparent reporting of different steps in the process of developing and evaluating the RAG system, thereby improving replicability and trust in the results. As a final remark, the now heavily growing number of research and studies on RAG systems requires for a more standardized and validated methodological research design. We make an important contribution toward a viable design with this paper.

DATA AVAILABILITY

On our supplementary website at <https://github.com/simisimon/dependency-validation-rag> we provide the dataset of configuration dependencies, source code for our RAG implementation, experimentation scripts, validation results, and prompt templates.

REFERENCES

- [1] Syed Juned Ali, Varun Naganathan, and Dominik Bork. 2024. Establishing Traceability between Natural Language Requirements and Software Artifacts by Combining RAG and LLMs. (2024).
- [2] Scott Barnett, Stefanus Kurniawan, Srikanth Thudumu, Zach Brannelly, and Mohamed Abdelrazek. 2024. Seven Failure Points When Engineering a Retrieval Augmented Generation System. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI* (Lisbon, Portugal) (*CAIN '24*). Association for Computing Machinery, New York, NY, USA, 194–199.
- [3] Salman Baset, Sahil Suneja, Nilton Bila, Ozan Tuncer, and Canturk Isci. 2017. Usable declarative configuration specification and validation for applications, systems, and cloud. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track* (Las Vegas, Nevada) (*Middleware '17*). Association for Computing Machinery, New York, NY, USA, 29–35.
- [4] Tom B Brown. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).
- [5] Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. 2024. Benchmarking Large Language Models in Retrieval-Augmented Generation. *Proceedings of the AAAI Conference on Artificial Intelligence* 38, 16, 17754–17762. <https://ojs.aaai.org/index.php/AAAI/article/view/29728>
- [6] 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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [7] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. 2020. Understanding and discovering software configuration dependencies in cloud and datacenter systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) (*ESEC/FSE 2020*). Association for Computing Machinery, New York, NY, USA, 362–374.
- [8] Wei Chen, Heng Wu, Jun Wei, Hua Zhong, and Tao Huang. 2016. Determine Configuration Entry Correlations for Web Application Systems. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 42–52.
- [9] Shahul Es, Jithin James, Luis Espinosa-Anke, and Steven Schockaert. 2023. Ragas: Automated evaluation of retrieval augmented generation. *arXiv preprint arXiv:2309.15217* (2023).
- [10] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) (*ICSE '23*). IEEE Press, 1469–1481.
- [11] Wei Fu and Tim Menzies. 2017. Easy over hard: a case study on deep learning. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) (*ESEC/FSE 2017*). Association for Computing Machinery, New York, NY, USA, 49–60.
- [12] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* (2023).
- [13] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) (*SoCC '16*). Association for Computing Machinery, New York, NY, USA, 1–16.
- [14] Jie Huang and Kevin Chen-Chuan Chang. 2023. Towards Reasoning in Large Language Models: A Survey. (2023), 1049–1065. Publisher Copyright: © 2023 Association for Computational Linguistics.; 61st Annual Meeting of the Association for Computational Linguistics, ACL 2023 ; Conference date: 09-07-2023 Through 14-07-2023.
- [15] Peng Huang, William J. Bolosky, Abhishek Singh, and Yuanyuan Zhou. 2015. ConfValley: a systematic configuration validation framework for cloud services. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) (*EuroSys '15*). Association for Computing Machinery, New York, NY, USA, Article 19, 16 pages.
- [16] Xinmei Huang, Haoyang Li, Jing Zhang, Xinxin Zhao, Zhiming Yao, Yiyuan Li, Zhuohao Yu, Tieying Zhang, Hong Chen, and Cuiping Li. 2024. LLMTune: Accelerate Database Knob Tuning with Large Language Models. *arXiv preprint arXiv:2404.11581* (2024).
- [17] Yizheng Huang and Jimmy Huang. 2024. A Survey on Retrieval-Augmented Text Generation for Large Language Models. *arXiv preprint arXiv:2404.10981* (2024).
- [18] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. 2012. A user survey of configuration challenges in Linux and eCos. In *Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems* (Leipzig, Germany) (*VaMoS '12*). Association for Computing Machinery, New York, NY, USA, 149–155.
- [19] Nihal Jain, Robert Kwiatkowski, Baishakhi Ray, Murali Krishna Ramanathan, and Varun Kumar. 2024. On Mitigating Code LLM Hallucinations with API Documentation. *arXiv preprint arXiv:2407.09726* (2024).
- [20] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* 55, 12, Article 248

- (mar 2023), 38 pages.
- [21] Dongpu Jin, Xiao Qu, Myra B. Cohen, and Brian Robinson. 2014. Configurations everywhere: implications for testing and debugging in practice (*ICSE Companion 2014*). Association for Computing Machinery, New York, NY, USA, 215–224.
 - [22] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radićek. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5131–5140.
 - [23] Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large Language Models are Zero-Shot Reasoners. 35 (2022), 22199–22213.
 - [24] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2024. Large language models are zero-shot reasoners (*NIPS ’22*). Curran Associates Inc., Article 1613, 15 pages.
 - [25] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2023. GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. *arXiv preprint arXiv:2311.03157* (2023).
 - [26] Xinyu Lian, Yinfang Chen, Runxiang Cheng, Jie Huang, Parth Thakkar, and Tianyin Xu. 2023. Configuration Validation with Large Language Models. *arXiv preprint arXiv:2310.09690* (2023).
 - [27] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, and Li Zhang. 2024. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971* (2024).
 - [28] Adam Mackay. 2024. Test Suite Augmentation using Language Models -Applying RAG to Improve Robustness Verification. In *ERTS2024 (ERTS2024)*. ERTS2024, Toulouse, France.
 - [29] Ben Maurer. 2015. Fail at Scale: Reliability in the face of rapid change. *Queue* 13, 8 (sep 2015), 30–46.
 - [30] Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Maddila, B. Ashok, Sumit Asthana, Christian Bird, and Aditya Kumar. 2020. Rex: preventing bugs and misconfiguration in large services using correlated change analysis. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA, USA) (*NSDI’20*). USENIX Association, USA, 435–448.
 - [31] David Oppenheimer, Archana Ganapathi, and David A Patterson. 2003. Why do Internet services fail, and what can be done about it?. In *4th Usenix Symposium on Internet Technologies and Systems (USITS 03)*.
 - [32] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590* (2023).
 - [33] Ariel Rabkin and Randy Katz. 2011. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) (*ICSE ’11*). Association for Computing Machinery, New York, NY, USA, 131–140.
 - [34] Ariel Rabkin and Randy Howard Katz. 2013. How Hadoop Clusters Break. *IEEE Software* 30, 4 (July 2013), 88–94.
 - [35] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
 - [36] Laria Reynolds and Kyle McDonell. 2021. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) (*CHI EA ’21*). Association for Computing Machinery, New York, NY, USA, Article 314, 7 pages.
 - [37] Alireza Salemi and Hamed Zamani. 2024. Evaluating Retrieval Quality in Retrieval-Augmented Generation. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Washington DC, USA) (*SIGIR ’24*). Association for Computing Machinery, New York, NY, USA, 2395–2400.
 - [38] Mark Santolucito, Ennan Zhai, Rahul Dhodapkar, Aaron Shim, and Ruzica Piskac. 2017. Synthesizing configuration file specifications with association rule learning. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 64 (oct 2017), 20 pages.
 - [39] Mark Santolucito, Ennan Zhai, and Ruzica Piskac. 2016. Probabilistic automated language learning for configuration files. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17–23, 2016, Proceedings, Part II* 28. Springer, 80–87.
 - [40] Mohammed Sayagh, Noureddine Kerzazi, and Bram Adams. 2017. On cross-stack configuration errors (*ICSE ’17*). IEEE Press, 255–265.
 - [41] Ensheng Shi, Yanlin Wang, Wei Tao, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2022. RACE: Retrieval-augmented Commit Message Generation. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, 5520–5530. <https://aclanthology.org/2022.emnlp-main.372>
 - [42] Norbert Siegmund, Nicolai Ruckel, and Janet Siegmund. 2020. Dimensions of software configuration: on the configuration context in modern software development (*ESEC/FSE 2020*). Association for Computing Machinery, New York, NY, USA, 338–349.
 - [43] Sebastian Simon, Nicolai Ruckel, and Norbert Siegmund. 2023. CfgNet: A Framework for Tracking Equality-Based Configuration Dependencies Across a Software Project. *IEEE Transactions on Software Engineering* 49, 8 (Aug 2023), 3955–3971.

- [44] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic configuration management at Facebook (*SOSP ’15*). Association for Computing Machinery, New York, NY, USA, 328–343.
- [45] Song Wang, Yaochen Zhu, Haochen Liu, Zaiyi Zheng, Chen Chen, et al. 2023. Knowledge editing for large language models: A survey. *arXiv preprint arXiv:2310.16218* (2023).
- [46] Xiaohua Wang, Zhenghua Wang, Xuan Gao, Feiran Zhang, Yixin Wu, Zhibo Xu, Tianyuan Shi, Zhengyuan Wang, Shizheng Li, Qi Qian, et al. 2024. Searching for Best Practices in Retrieval-Augmented Generation. *arXiv preprint arXiv:2407.01219* (2024).
- [47] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. 35 (2022), 24824–24837.
- [48] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) (*ICSE ’23*). IEEE Press, 1482–1494.
- [49] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) (*MAPS 2022*). Association for Computing Machinery, New York, NY, USA, 1–10.
- [50] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farmington, Pennsylvania) (*SOSP ’13*). Association for Computing Machinery, New York, NY, USA, 244–259.
- [51] Tianyin Xu and Yuanyuan Zhou. 2015. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Comput. Surv.* 47, 4, Article 70 (jul 2015), 41 pages.
- [52] Pengyu Xue, Linhao Wu, Zhongxing Yu, Zhi Jin, Zhen Yang, Xinyi Li, Zhenyu Yang, and Yue Tan. 2024. Automated Commit Message Generation with Large Language Models: An Empirical Study and Beyond. *arXiv preprint arXiv:2404.14824* (2024).
- [53] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) (*SOSP ’11*). Association for Computing Machinery, New York, NY, USA, 159–172.
- [54] Hao Yu, Aoran Gan, Kai Zhang, Shiwei Tong, Qi Liu, and Zhaofeng Liu. 2024. Evaluation of Retrieval-Augmented Generation: A Survey. *arXiv preprint arXiv:2405.07437* (2024).
- [55] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. EnCore: exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) (*ASPLOS ’14*). Association for Computing Machinery, New York, NY, USA, 687–700.
- [56] Linghao Zhang, Hongyi Zhang, Chong Wang, and Peng Liang. 2024. RAG-Enhanced Commit Message Generation. *arXiv preprint arXiv:2406.05514* (2024).
- [57] Ningyu Zhang, Yunzhi Yao, Bozhong Tian, Peng Wang, Shumin Deng, Mengru Wang, Zekun Xi, Shengyu Mao, Jintian Zhang, Yuansheng Ni, et al. 2024. A comprehensive study of knowledge editing for large language models. *arXiv preprint arXiv:2401.01286* (2024).
- [58] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. 2023. Siren’s song in the AI ocean: a survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219* (2023).
- [59] Penghao Zhao, Hailin Zhang, Qinhan Yu, Zhengren Wang, Yunteng Geng, Fangcheng Fu, Ling Yang, Wentao Zhang, and Bin Cui. 2024. Retrieval-Augmented Generation for AI-Generated Content: A Survey. *CoRR* (2024).
- [60] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910* (2022).