On Automating Configuration Dependency Validation via Retrieval-Augmented Generation

Sebastian Simon*, Alina Mailach*†, Johannes Dorn*, Norbert Siegmund*†
*Leipzig University, Leipzig, Germany
†ScaDS.AI Dresden/Leipzig, Leipzig, Germany
{sebastian.simon, alina.mailach, johannes.dorn, norbert.siegmund}@cs.uni-leipzig.de

***Abstract*—Configuration dependencies arise when multiple technologies in a software system require coordinated settings for correct interplay. Existing approaches for detecting such dependencies often yield high false-positive rates, require additional validation mechanisms, and are typically limited to specific projects or technologies. Recent work that incorporates large language models (LLMs) for dependency validation still suffers from inaccuracies due to project- and technology-specific variations, as well as from missing contextual information.**

In this work, we propose to use retrieval-augmented generation (RAG) systems for configuration dependency validation, which allows to incorporate additional project- and technology-specific context information. Specifically, we evaluate whether RAG can improve LLM-based validation of configuration dependencies and what contextual information are needed to overcome the static knowledge base of LLMs. To this end, we conducted a large empirical study on validating configuration dependencies using RAG. Our evaluation shows that vanilla LLMs already demonstrate solid validation abilities, while RAG has only marginal or even negative effects on the validation performance of the models. By incorporating tailored contextual information into the RAG system–derived from a qualitative analysis of validation failures–we achieve significantly more accurate validation results across all models, with an average precision of 0.84 and recall of 0.70, representing improvements of 35 % and 133 % over vanilla LLMs, respectively. In addition, these results offer two important insights: Simplistic RAG systems may not benefit from additional information if it is not tailored to the task at hand, and it is often unclear upfront what kind of information yields improved performance.

***Index Terms*—RAG, LLMs, Configuration Dependencies, Dependency Validation**

I. Introduction

Modern software development involves coordinating a wide range of technologies, including code frameworks, build tools, databases, and *continuous integration and delivery* (CI/CD) pipelines [1]. Each technology typically requires configuration in its own format. For instance, container configurations are defined in a `Dockerfile`, whereas Java applications may be configured in a YAML-file. Such configuration files encode hundreds of configuration options in their own structure, syntax, and semantic [2], resulting in a vast and complex configuration space. Most of these diverse technologies must be configured jointly to ensure their correct interplay. Consider a typical Spring Boot project: the port values specified in the `application.yml` in Listing 1 and in the `Dockerfile` in Listing 2 must match to ensure the application is accessible from outside the container. This constraint constitutes a configuration dependency. Such dependencies can occur within a single technology (*intra*-technology) and across multiple technologies (*cross*-technology).

```yaml
server:
  port: 8080
spring:
  application:
    name: app
```

Listing 1: Application port configured in Spring Boot's application.yml.

```dockerfile
FROM openjdk:17
COPY target/myapp.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Listing 2: A Dockerfile exposing the port of the application.

Naturally, developers cannot meticulously manage all the frameworks and tools that potentially interact in their technology stack. As a result, configuration dependencies are rarely fully documented [3], leading to incomplete and often outdated documentation [4, 5, 6]. The 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 increases the risk of severe misconfigurations due to the violation of dependencies [7, 8, 9]; (2) misconfigurations often remain unnoticed until production because they emerge from complex interactions between multiple technologies [2]; (3) resulting configuration errors are complex and far-reaching, involving several technologies and diverse configuration artifacts, such that detection becomes time-consuming and costly [10, 11]; (4) resolving these configuration errors has found to be more challenging than fixing software bugs, making misconfigurations one of the most common causes of software failures in production today [12, 13].

Several dependency detection approaches have been proposed, using heuristics [14, 3, 15, 16], domain-specific languages [17, 18], and machine learning techniques [19, 20, 21]. Unfortunately, these approaches often suffer from a high rate of false positives, where, for example, independent configuration options are incorrectly linked due to coincidental similarities. To address these false positives, further validation mechanisms have been integrated to additionally judge configuration dependencies. However, these validation techniques

typically require significant technical effort and are usually limited to individual projects, a small set of technologies, or specific dependency types [18].

Given these challenges, recent advances in large language models (LLMs) [22, 23, 24, 25, 26, 27] present an opportunity to improve the validation of configuration dependencies in an automated and technology-agnostic way. In fact, Lian et al. [28] have successfully validated configuration constraints in individual configuration files of a *single* technology using LLMs. Nevertheless, LLMs still struggle to validate dependencies across *multiple* technologies due to their project- and technology-specific variations and a lack of contextual information. Here, key limitations mainly include outdated training data of LLMs (e.g., due to changes in configuration formats and frameworks), the unavailability of project-specific information (e.g., project structure or available resources), and ambiguous naming schemes for configuration options.

Retrieval-augmented generation (RAG) [29] offers a solution to the lack of information available in pre-trained LLMs by providing additional context during generation. However, selecting relevant context sources is particularly challenging, as information about configuration dependencies is rarely available and often scattered across multiple sources. For example, user manuals typically describe configuration options in terms of their names, functionality, and occasionally their allowed parameter ranges. But they almost never include explicit dependency information to other options, neither within the same technology nor across technologies. So, developers must often consult different sources, such as official technology documentation, Stack Overflow posts, GitHub repositories, or Web searches, to find real-world use cases of configuration options in which configuration dependencies are explicitly or implicitly discussed when validating dependencies.

In this work, we propose the use of RAG for configuration dependency validation. Specifically, we evaluate the validation effectiveness of vanilla LLMs compared to RAG and what types of information LLMs actually require for effective dependency validation. Specifically, we develop a novel RAG system for dependency validation, which integrates four widely-used context sources (i.e., technology docs, Stack Overflow posts, GitHub repositories, and Web search results). Our approach is guided by three primary objectives: (1) reducing false positives by providing accurate classification capabilities in a automated and technology-agnostic way, (2) assessing the usefulness of RAG for dependency validation, and (3) identifying what information LLMs actually require for effective dependency validation.

In a series of experiments, we first validate 350 configuration dependencies from ten real-world open-source software projects using six state-of-the-art LLMs and eight RAG variants (see Section IV-B) with the same underlying models. We then manually analyze validation failures of vanilla LLMs and the best found RAG variant to derive key information LLMs require for effective dependency validation. Finally, we integrate this special information into the best found RAG variant and assess its validation effectiveness on a previously unseen dataset of 150 configuration dependencies.

Our results show that vanilla LLMs already exhibit solid validation abilities, while RAG, contrary to its promise, only has marginal or even negative effects on their validation effectiveness. A closer look at the retrieved context for this task reveals that it often lacks dependency-related information and thus increases noise for LLMs rather than providing meaningful information, negatively affecting their validation performance. In a subsequent qualitative analysis of validation failures, we identified eight distinct failure categories, most of which are project- and technology-specific and were previously unknown to the literature. From these categories, we derived key information necessary for effective dependency validation, ranging from project-specific details over precise prompts to similar examples of classified dependencies. Integrating this special information into the best found RAG variant significantly improves validation performance over vanilla LLMs, achieving an average precision of 0.84 and recall of 0.70—improving by 35 % and 133 %, respectively.

In addition to these improvements in dependency validation, the key takeaway is that merely adding contextual information does not guarantee better validation effectiveness of LLMs. Instead, a systematic refinement of RAG components and particularly context sources is essential to unleash the full potential of RAG systems for dependency validation, and thus needs to be done in any RAG development activity.

In summary, we make the following contributions:

- A novel RAG system for configuration dependency validation, achieving an improvement of precision by 35 % and recall by 133 % over vanilla LLMs.
- The largest dataset of cross-technology configuration dependencies manually validated that we are aware of.
- An empirical assessment on the effectiveness of RAG for dependency validation, confirming that RAG with special information is beneficial for this task.
- An extensive qualitative analysis of validation failures, resulting in a novel taxonomy of dependency validation failures, thereby enabling future work in this area.

We provide a supplementary website with all code scripts, dependency datasets, prompt templates, and validation results at https://github.com/AI-4-SE/rag-config-val.

II. Related Work

In this section, we discuss related work on dependency detection, configuration validation, and LLMs for tackling software configuration.

Configuration Dependency Detection: The challenge of detecting configuration dependencies in software systems has inspired several research approaches. A number of approaches rely on machine learning. For example, ConfigC [20] and ConfigV [19] use an advanced form of association rule learning to learn different kinds of configuration rules. These approaches identify both coarse- and fine-grained value dependencies between configuration options in MySQL configuration files.

Chen et al. [3] studied configuration dependencies in 16 major software projects using metadata and manuals to iden-

tify common code patterns and existing dependency-checking practices. They developed cDep, a framework that identifies configuration dependencies within and across software components using static taint analysis. Simon et al. [14] conducted a systematic literature study, identifying value-equality as a significant and common form of cross-technology dependency in academic literature. They propose CfgNet, a framework that models a software project and its surrounding ecosystem (i.e., technologies such as CI/CD pipelines, build tools, and code frameworks) as a configuration network, enabling the early detection of dependency violations by checking the changes in a configuration network. Ramachandran et al. [30] utilized APIs to determine configuration dependencies, developing a heuristic to rank critical dependencies. Rex [9] employs association rule learning to identify correlations between code and configuration files in large services and suggest required changes. These approaches offer valuable solutions for detecting configuration dependencies in software systems, but often require extensive additional validation efforts. By contrast, our goal is to enable an automated and technology-agnostic validation of dependencies using RAG.

Configuration Validation: Several research approaches aim to validate configurations to proactively detect misconfigurations. A common method relies on domain-specific languages to describe configuration specification rules. For example, ConfValley [17] includes a declarative language for describing configuration specifications, an inference engine for automatically generating these specifications, and a checker engine for determining if a given configuration matches its specification. Similarly, ConfigValidator [18] employs a declarative language to write rules for detecting various misconfigurations. Both approaches require manual engineering to define specification rules. By contrast, we focus on an automated validation of configuration dependencies using LLMs, eliminating the need for manual rule engineering.

The work most closely related to ours is that of Lian et. al [28], which investigates the effectiveness of using LLMs for configuration validation. They introduce Ciri, an LLM-based framework, to detect constraint and dependency violations in single configuration files. Their experiments on ten open-source software systems show promising results in validating constraints, but difficulties in recognizing certain misconfigurations, such as dependency violations. While constraint violations (e.g., syntax and value range) are more common, dependency violations are often project- and technology-specific and thus harder for LLMs to capture without fine-tuning on dependency-related data or additional relevant contextual information. The key difference to our work is the focus on single-file constraint and dependency violations. So as a novel contribution, we provide for the first time a RAG system that integrates additional context information to validate configuration dependencies across multiple files, which sets our work apart from Lian et al.[28].

LLMs for Tackling Software Configuration: LLMs are promising tools for handling software configuration, demonstrated by various LLM-based tools. For instance, DB-Bert [31] extracts domain knowledge gained from relevant text sources, such as manuals, to identify database systems parameters for tuning and recommended parameters values. GPTuner [22] optimizes database management system (DBMS) tuning using an LLM-based pipeline and a prompt ensemble algorithm. LLMTune [23] recommends high-quality initial configurations tailored to a new database tuning task based on previous tuning tasks of DB administrators. LLM-CompDroid [24] uses LLMs alongside traditional bug resolution tools to detect and repair configuration compatibility bugs in Android apps. LogConfigLocalizer [25] leverages LLMs to localize root-cause configuration properties through log analysis. PerfSense [26] identifies performance-sensitive configurations using LLMs. These approaches highlight LLMs' potential in handling software configuration problems. We extend this domain by exploring RAG for dependency validation.

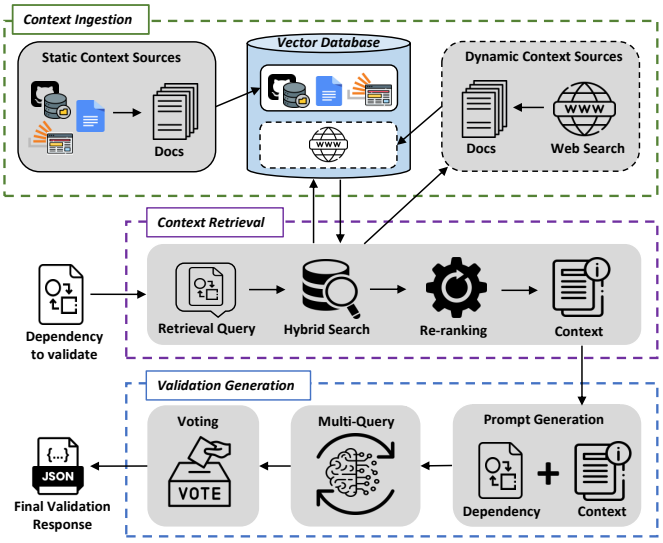

Fig. 1: Overview of the architecture of the RAG system

III. Dependency Validation with RAG

We propose a novel RAG system to validate configuration dependencies. Figure 1 gives an overview of the architecture from indexing data from different sources as context into a vector database, over retrieving relevant context from it, to querying an LLM with an augmented prompt. Next, we describe these steps in detail.

A. Context Ingestion

As the knowledge foundation of RAG, we populate a vector database with *static* and *dynamic* context information. *Static* context information is stored at system initialization and originates from: technology documents, Stack Overflow posts, and GitHub repositories. For each validation query, we extract *dynamic* context information via Web search and then index the content into the vector database. We rationalize the selection of context sources in Section IV-B.

To index content derived from the context sources, we first generate documents with metadata, such as the origin of a

context source. The resulting documents are then split into smaller chunks. Based on preliminary experiments aimed at balancing context granularity and length, we selected a chunk size of 256 tokens and an overlap of 10 tokens, ensuring sufficient context capture. These chunks are converted into embeddings, a vector representation of the documents, using an embedding model. Finally, the resulting embeddings are indexed in the vector database Pinecone [32].

B. Context Retrieval

The RAG system receives a configuration dependency as input from a dependency-detection tool. Here, a dependency must include the involved option names, their values, configuration files, and associated technologies. We first transform this dependency into a structured *dataclass* instance that encapsulated these attributes, enabling a consistent handling throughout context retrieval and validation generation. Based on this structured representation, we construct a retrieval query using a predefined template that embeds the option names, values, and associated technologies of the dependency. To obtain an initial set of context, we conduct a hybrid search in the vector database with the retrieval query. With hybrid search, we perform a search with sparse-dense vectors, combining sparse and dense embeddings in a single vector. This allows us to find context based on configuration option names via keyword matching (sparse vectors) and relevant context of dependencies via semantic matching (dense vectors). After obtaining a large initial set of context, we re-rank the elements of the initial set according to their relevance. Finally, we use the re-ranked context to augment the prompt for dependency validation.

C. Validation Generation

We created a prompt for dependency validation using a collective prompt design process, in which the first author drafted an initial version of the prompt, which was then reviewed, discussed, and adjusted with the other authors to improve its phrasing. 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 type that it has to validate, (3) the retrieved context information (see Section III-B), (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 2.

A central challenge addressed in the dependency definition (2) is the potential of configuration option values to be equal by coincidence. We explicitly accounted for this, as early observations revealed that some options, particularly with boolean and numeric values, can coincidentally match without indicating a dependency.

The JSON response format prompts the LLM to fill in three key fields: (1) *plan*: a string that describes the plan 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 each other or not; and

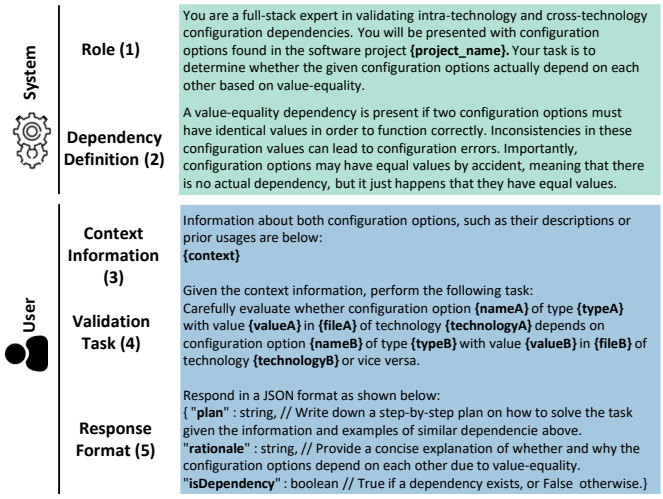

Fig. 2: The prompt template and its components used for dependency validation.

(3) *isDependency*: a boolean indicating the final decision of the LLM whether the given dependency is a true dependency or a false positive.

This response structure aligns with chain-of-thought prompting [33], enhancing the LLM's analytical capabilities. By decomposing the validation process into sequential steps, the LLM ensures a logical progression, first outlining a validation strategy, then providing reasoning, and finally making a decision. This aligns with recent reasoning research to enable more test-time compute to reach a final decision. Moreover, we initially pursue a zero-shot strategy to assess the general capability of RAG in validating dependencies without influencing results through task-specific examples.

As LLMs are prone to hallucinations and inconsistencies [34, 35], we adopt a multi-query strategy. By querying the LLM multiple times with the same prompt and temperature and aggregating the responses, we obtain a final result that better reflects the model's understanding of the configuration dependency and enhances its consistency compared to a single try. Specifically, we apply a frequency-based voting strategy to select the response that recurs most often across the generated responses [36, 28].

IV. Experiment Setup

Our experiment setup follows a proposed methodology for evaluating RAG systems [37]. We present our research questions and design decision for conducting our experiments.

A. Research Questions

RAG for Dependency Validation: We evaluate whether and to what extent RAG improve over vanilla LLMs for dependency validation. To this end, we conduct an experiment in which we validate 350 real-world configuration dependencies with six state-of-the-art LLMs and eight different RAG variants with the same underlying LLMs. Thereby, we assess the validation effectiveness of vanilla LLMs compared to

RAG and analyze the kind and amount of retrieved contextual information. We state the following research questions:

$RQ_{1.1}$	*How effective are vanilla LLMs compared to RAG in validating configuration dependencies?*
$RQ_{1.2}$	*What information does RAG retrieve for dependency validation?*

Types of Information Needed for Dependency Validation: To contribute further to the automated validation of configuration dependencies, we determine when and why LLMs fail. With a RAG system at our hand, we can qualitatively analyze the causes of failures as we have explicit contextual information available. This is hardly achievable with pure LLMs since the information is inherently stored in the model's parameters. Thus, we state the following research questions:

$RQ_{2.1}$	*What kind of failures occur in LLM-based dependency validation?*
$RQ_{2.2}$	*What types of information do LLMs require for accurate dependency validation?*

B. Operationalization

Procedure: To address $RQ_{1.1\text{-}1.2}$, we validated 350 real-world configuration dependencies using six state-of-the-art LLMs and eight RAG variants with the same underlying models. We began by validating the dependencies using the vanilla LLMs without any additional context. Next, we validated the dependencies with each RAG variant as follows: (1) We set up the vector database according to the specific RAG variant and ingested previously scraped *static* context information. (2) For each dependency, we then scraped the Web for additional *dynamic* context and ingested it into the vector database. (3) Next, we retrieved context information from the vector database for each dependency, including now both *static* and *dynamic* context information. We stored the retrieved context along with its metadata, such as the source and relevance score. (4) We joined the dependency and context information and generated validations for all dependencies using the different LLMs. We repeated these steps for all RAG variants. This procedure ensured that all LLMs received exactly the same context for a given dependency, avoiding order and time bias and, thus, enabling a sound comparison of the validation performance across different LLMs and RAG variants. Finally, we answered $RQ_{1.1}$ by comparing the validation performance using precision, recall, and F1-score, and analyzed the context information retrieved by the different RAG systems to answer $RQ_{1.2}$.

To answer $RQ_{2.1\text{-}2.2}$, we conducted a qualitative analysis of failures made by the vanilla LLMs and the best found RAG system. Specifically, we inspected all instances in which validation fails by extracting the involved configuration options, the retrieved context, and the reasoning of the LLM. We examined names and values of configuration options, as well as the technologies involved and assessed whether the LLM's reasoning aligns with its incorrect final assessment. By comparing this reasoning to our own reasoning during manual classification, we analyzed whether validation failures originate from missing information, the presence of false information, noise, or ambiguities. We then categorized failures based on recurring mistakes or underlying causes ($RQ_{2.1}$). From these failure categories, we derived what type of information an LLM needs to improve dependency validation. To ensure that this information, in fact, leads to improved validation performance, we integrated it into the best found RAG variant and, if applicable, to the vanilla LLMs, and assessed the validation effectiveness of the refined RAG variant on an unseen dataset of 150 real-world configuration dependencies ($RQ_{2.2}$).

TABLE I: Subject systems with version (SHA), number of stars, and number of commits

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

Datasets: A sound evaluation requires a dataset containing configuration dependencies and ground truth labels that can be used to assess the validation effectiveness of RAG. We extracted software projects that integrate technologies with potential configuration dependencies spanning an entire technology stack and that are covered by a previously published dependency-detection tool. We decided to use *CfgNet* [14] to extract potential dependencies since it is technology-agnostic and supports many state-of-the-art technologies.

For the technology stack, we decided to select software projects that incorporate a Spring Boot microservice architecture as it represents a technology stack of significant practical relevance. Using GitHub Topics [38], we obtained over 60 000 public repositories in the Spring Boot ecosystem. We sorted these projects by stars and included them if they incorporate at least the following technologies: Spring Boot, Maven, Docker, and Docker Compose. We selected the top 10 software projects that met these criteria to ensure a manageable yet diverse, and practically important set of subject systems. We list the resulting subject systems and their characteristics in Table I.

To obtain a ground truth dataset of cross-technology configuration dependencies, we ran *CfgNet* on the subject systems. For each software project, we sampled 50 potential dependencies, leading to a final set of 500 dependency candidates — the largest cross-technology dependency dataset we are aware of. Note that the resulting dataset is not restricted to the technologies used for selecting the projects, but includes all technologies that *CfgNet* captures from a repository, including build tools, CI/CD pipelines, databases, code, and testing frameworks.

To ensure the correctness of our ground truth dataset, we employed a manual multi-stage annotation process: First, the main author manually reviewed and classified each candidate in this dataset into one of three classes: (1) *true* for a valid dependency, (2) *false* for a false positive, and (3) *borderline* for unclear cases based on domain/technology expertise and inspection of related configuration options and values. Next, a second author independently reviewed all labels, conducting sanity checks and flagging inconsistencies. Finally, all borderline and conflicting cases were jointly discussed by the authors until a final consensus has been reached. This ensured a high-quality, human-validated dataset for our evaluation. Out of the 500 candidates, we finally classified 199 (40 %) as *true* and 301 (60 %) as *false* dependencies.

We split the dataset into two subsets using a 70/30 ratio, ensuring stratification such that each subset maintains proportional representation of dependencies across different projects. The larger dataset, consisting of 350 configuration dependencies, is used to investigate whether RAG enables effective dependency validation. The smaller dataset, containing 150 configuration dependencies, serves as an unseen test set to evaluate the refined RAG variant's ability to reliably validate configuration dependencies. Both subsets maintain a distribution of approximately 40 % true positives and 60 % false positives. As a notable contribution of this work, we make this entire dataset available.

RAG Variants: A possible confounding factor in our study represents the assembly of the RAG system, as alternative components can be chosen and may affect result accuracy. Hence, we implemented eight different RAG variants differing in three components (embedding model, re-ranking algorithm, and number of chunks). The RAG variants are shown in Table II. In our study, we conduct all experiments on all variants, but report only on the best due to space constraints.

Baselines: To asses the effectiveness of our RAG variants, we compare their validation abilities against vanilla LLMs. We leverage six state-of-the-art LLMs, including two proprietary LLMs (GPT-4o and GPT-4o-mini) and four open-source LLMs (Deepseek-r1:70b, Deepseek-r1:14b, Llama3.1:70b, Llama3.1:8b). A summary of the LLMs with their alias and other properties can be found in Table III. Throughout the remainder of the paper, we refer to each model by its alias. To balance creativity and determinism, we set the temperature of all models to 0.5 and repeated each prompt with identical content multiple times. This frequency voting mechanism ensures a higher robust and consistency than any single deterministic response [39], following best practices of self-consistency chain of thought [36] and multi-path reasoning [40].

A comparison against existing dependency-detection tools [3, 9] is, unfortunately, not possible as there is no publicly available tool applicable that explicitly targets configuration dependencies across multiple technologies. Moreover, we are not aware of any existing ML method (e.g., Bert) that is applied to dependency validation. Implementing such a baseline would require a ground truth of validated dependencies for each technology pair for fine-tuning. This is practically infeasible to achieve due to the combinatorial explosion of cross-technology dependencies.

TABLE II: Comparison of RAG variants by embedding model, embedding dimensionality (Embed. Dim.), reranking method (Reranking), and top-N retrieved results (Top N).

ID	Embedding Model	Embed. Dim.	Reranking	Top N
R1	text-embed-ada-002	1536	Sentence Transformer	5
R2	text-embed-ada-002	1536	Sentence Transformer	3
R3	text-embed-ada-002	1536	Colbert Rerank	5
R4	text-embed-ada-002	1536	Colbert Rerank	3
R5	gte-Qwen2-7B-instruct	3584	Sentence Transformer	5
R6	gte-Qwen2-7B-instruct	3584	Sentence Transformer	3
R7	gte-Qwen2-7B-instruct	3584	Colbert Rerank	5
R8	gte-Qwen2-7B-instruct	3584	Colbert Rerank	3

TABLE III: Comparison of LLMs by parameter count (# Params, in billions), maximum context length (Context Len., in tokens), and open-source status (Open Src., ✓= yes).

Alias	Full Model Name	# Params	Context Len.	Open Sourc.
4o	gpt-4o-2024-11-20	-	128 k	✗
4o-mini	gpt-4o-mini-2024-07-18	-	128 k	✗
DSr:70b	deepseek-r1:70b	70 B	131 k	✓
DSr:14b	deepseek-r1:14b	14 B	131 k	✓
L3.1:70b	llama3.1:70b	70 B	8 k	✓
L3.1:8b	llama3.1:8b	8 B	8 k	✓

Context Sources: Data on configuration dependencies is typically scarce, as dependencies are rarely explicitly documented and almost never specify dependencies between options, neither within the same technology nor across different technologies [41, 6]. Hence, developers often resort to alternative sources, such as man pages, community forums, such Stack Overflow, or broader Web searches [42, 43]. Beyond public documentation, project-specific details, such as environment settings or best practices, are sometimes embedded in project repositories, issue trackers, or discussion threads within version control platforms.

To address the challenges in selecting relevant context sources, we incorporate multiple widely used context sources, including official technology documentation, Stack Overflow posts, GitHub repositories, and Web search results. We systematically scraped data from the initial technologies used for project selection, focusing on their configuration files, their GitHub repositories, and the top 100 Stack Overflow posts for each technology pair. All these sources were collected prior to generation queries (i.e., representing *static* context), whereas the Web (i.e., *dynamic* context) was scraped dynamically for each generation query. The rational of this selection is to replicate developer behavior when validating configuration dependencies. We provide all retrieved context sources, including Stack Overflow posts, tech documents, and GitHub repositories, on our supplementary website.

Evaluation Metrics: We measure the effectiveness (or performance) of our RAG system in validating configuration dependencies using widely used classification metrics: precision, recall, and F1-score. 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 therefore crucial for trustworthy and reliable validation with LLMs. The F1-score, the harmonic mean of precision and recall, provides a single scalar for better comparability.

To measure relevance of context information for a configuration dependency, we use the relevance score provided by Pinecone. The metric is calculated using the dot product from the dense and sparse vectors of the retrieval query and the vector database (the embedded static and dynamic context information). This way, we obtain a measure of how closely these vectors are in terms of their direction and, thus, how similar they are. More similar vectors get higher scores and the context information is, thus, interpreted as more relevant for the configuration dependency it was retrieved for.

We refrain from using RAGAS metrics [44] in our evaluation for three reasons: (1) Our primary focus is to asses the validation effectiveness in a typical classification task. Hence, classification metrics capture the true goal of the task. (2) Instead of relying on a set of quantitative metrics for failure analysis and context relevance evaluation, we employ an in-depth qualitative manual analysis, which yields concrete actionable insights and root causes of failures. (3) RAGAS incurs a high cost when evaluating a large number of LLM queries. Our preliminary studies indicated that its metrics are highly sensitive to the choice of LLM and often produce unreliable results due to LLMs' partial non-compliance with the expected data format of the RAGAS framework.

V. Results

We first present the results of $RQ_{1.1\text{-}1.2}$ regarding the validation effectiveness of vanilla LLMs compared to RAG and the retrieved contextual information. We then present validation failure categories to identify key information for effective dependency validation, answering $RQ_{2.1\text{-}2.2}$. Note that we only report the results for the best performing RAG variant (*R1*), while the validation results for the remaining RAG variants are available on our supplementary website.

TABLE IV: Validation effectiveness of vanilla LLMs ("w/o") and the best performing RAG variant *R1*.

Model	Precision		Recall		F1-Score	
	w/o	R1	w/o	R1	w/o	R1
4o	0.89	0.86	0.46	0.61	0.61	0.71
4o-mini	0.76	0.60	0.18	0.78	0.29	0.68
DSr:70B	0.76	0.74	0.59	0.73	0.66	0.74
DSr:14B	0.84	0.66	0.56	0.46	0.67	0.54
L3.1:70b	0.70	0.65	0.45	0.34	0.55	0.45
L3.1:8b	0.52	0.53	0.52	0.34	0.52	0.41
Mean	**0.75**	0.67	**0.46**	0.54	**0.55**	0.59
Best	**0.89**	0.86	**0.59**	0.78	**0.67**	0.74

A. RAG for Dependency Validation

$RQ_{1.1}$: Validation Effectiveness of Vanilla LLMs and RAG: We validated 350 manually labeled configuration dependencies of a cross-technology stack from ten real-world software projects with six state-of-the-art vanilla LLMs and eight different RAG variants. Table IV shows the precision, recall, and F1-score for each vanilla LLM and the best performing RAG variant *R1* with the same underlying models.

Overall, our results demonstrate solid validation capabilities of vanilla LLMs for configuration dependencies, indicated by F1-scores ranging from 0.52 to 0.67, with both *DeepSeek-R1* models outperforming the other models. However, the results are not consistent: all models exhibit higher precision than recall, except *L3.1:8b*. That is, we observe a precision mostly in the ranges between 0.6 and 0.8, whereas *4o* achieves the highest precision at the cost of recall and *DSr:70B* achieves the highest recall at the cost of precision. Notably, *DSr:14B* matches and even surpasses the validation capabilities of larger models in terms of F1-score without additional context information.

The best performing RAG variant *R1* shows clear improvements for *4o*, *4o-mini*, and *DSr:70B*, with the most substantial gain observed for *4o-mini*, where the F1-score more than doubles. Conversely, the vanilla LLMs outperform *R1* for *DSr:14B*, *L3.1:70b*, and *L3.1:8b*. Moreover, we observe an interesting pattern: the precision tends to drop for all models with additional context, but recall only increases for both *GPT* models. This means that all LLMs perform slightly worse in correctly identifying true dependencies with additional context. Only both *GPT* models no longer miss as much dependencies than before. This mixed picture is a strong indicator that even plausible context information is not always helpful for this task and does not generally improve the validation abilities of LLMs. It also demonstrate that this task is quite challenging for LLMs as factual relevant information seem to be not explicitly available such that the context may increase noise for an LLM rather than providing meaningful information.

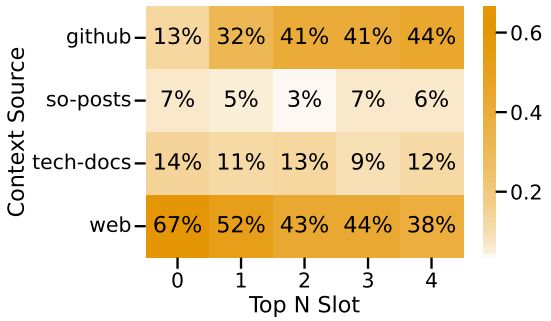

Fig. 3: Usage of context sources for RAG variant *R1*.

$RQ_{1.2}$: Retrieved Contextual Information: In Figure 3, we highlight the fraction of sources that the best performing RAG variant *R1* has deemed relevant for the query and submitted to one of five context slots. Web search dominates, especially

in the first two slots, likely due to its broad coverage and up-to-date information. GitHub information appears in the latter context slots, indicating that RAG deems project-specific information to be relevant for dependency validation. Interestingly, technology documentation appear less frequently, suggesting that they lack details needed for validation. Similarly, the low selection rate of Stack Overflow posts implies that user discussions and troubleshooting insights may be less aligned with explicit configuration dependencies or they do not align with the search process of vector databases. This underpins the scarcity of explicit dependency-related information in these context sources.

We also calculated the average relevance scores for each context slot in *R1*. Notably, *R1* shows continuously negative relevance scores, ranging from -2.91 in the first slot to -7.68 in the fifth slot. This may indicate that the metric does not capture the essence of relevance to this task. Since much of the retrieved context is irrelevant, it likely introduces noise rather than meaningful information, regardless of the source from which it was obtained. Also interesting, when comparing these values to other RAG variants (not shown here), we find that other variants have a positive relevance scores despite the lower accuracy and recall. Hence, we refrain from over-interpreting this metric and point to the discussion.

Summary: Answering $RQ_{1.1}$, vanilla LLMs have a high precision for identifying valid configuration dependencies, however, at the cost of missing many true dependencies. By contrast, the additional context of a RAG system leads to a higher recall but comes with a reduced precision. So, both approaches shine for different aspects. For $RQ_{1.2}$, we found that Web search and GitHub sources are most often used as a context sources. This diminishes, at least partly, the importance of Stack Overflow posts and technology documentation.

B. Types of Information Needed for Dependency Validation

RQ$_{2.1}$: Validation Failures: In total, we reviewed 1192 validation failures from vanilla LLMs and *R1* and derived eight distinct failure categories. We summarize the failure categories with a brief description in Table V and present their distribution across the vanilla LLMs and *R1* in Table VI. Due to space constraints, we refer for concrete examples to the supplementary website.

A key observation is that some failure categories are more prevalent in certain models, indicating varying degrees of competency in dependency validation. Notably, *4o* demonstrates a strong ability to validate dependencies in the *Port Mapping* and *Independent Technologies and Services* categories, whereas all other LLMs struggle with these dependencies. This suggests that larger and more advanced models may possess a better grasp of infrastructure-level dependencies, likely benefiting from broader and more comprehensive training data. However, there are also failure categories in which all LLMs encounter significant difficulties, particularly in the *Inheritance and Overrides* and *Configuration Consistency* category. These categories consistently exhibit the highest failure counts across all models, highlighting a fundamental limitation in how LLMs capture hierarchical relationships in software projects and configuration dependencies due to their highly project- and technology-specific nature. Such information is often encoded in internal documentation, such as wikis or internal knowledge bases. This suggest that pre-training data may be insufficient in covering project-specific nuances, necessitating the integration of project-specific details.

TABLE V: Eight failure categories derived from a qualitative analysis of failures from vanilla LLMs and RAG variant *R1*.

Category	Description
Inheritance and Overrides	This category includes validation failures due to Maven's project inheritance, which allows modules to inherit and override configurations from a parent module, such as general settings, dependencies, plugins, and build settings.
Configuration Consistency	Often configuration values are the same across different configuration files, which often leads to dependencies, but sometimes only serves the purpose of consistency. In this category, LLMs confuse equal values for the sake of consistency with real dependencies.
Resource Sharing	Resources, such as databases or services can be shared across modules or used exclusively by a single module. Without additional project-specific about available resources, LLMs struggle to infer whether resources are shared or used exclusively by a single module.
Port Mapping	Ports of services are typically defined in several configuration files of different technologies, creating equality-based configuration dependencies. However, not all port mappings have to be equal (e.g. a container and host port in docker compose).
Naming Schemes	Software projects often use ambiguous naming schemes for configuration options and their values. These ambiguities result from generic and commonly used names (e.g., project name) that may not cause configuration errors if not consistent but can easily lead to misinterpretation by LLMs.
Context (Availability, Retrieval, and Utilization)	Failures in this category are either because relevant information is missing (e.g. not in the vector database or generally not available to vanilla LLMs), available in the database but not retrieved, or given to the LLM but not utilized to draw the right conclusion.
Independent Technologies and Services	In some cases (e.g. in containerized projects) different components, such as services, are isolated by design. In these cases the configuration options between these components are independent, if not explicitly specified.
Others	This category contains all validation failures where the LLMs fail to classify the dependencies correctly that can not be matched to any other category and share no common patterns.

When comparing vanilla LLMs to their RAG counterpart, we observe that *4o*, *4o-mini*, and *DSr:70b* produce less failures, as they benefit from additional context indicated by an increase in F1-score by 16.4 %, 134.5 %, and 12.1 %. The remaining models (*DSr:14b*, *L3.1:70b*, and *L3.1:8b*) show an increase in validation failures. This aligns with our earlier observations, where validation effectiveness improves for larger LLMs and degrades for smaller ones.

RQ$_{2.2}$: Special Information for Dependency Validation: Based on the failure categories' characteristics, we derived

TABLE VI: Statistics of failure categories of vanilla LLMs and the RAG variant *R1*.

	Vanilla LLMs						R1					
Failure Category	4o	4o-mini	DSr:70b	DSr:14b	L3.1:70b	L3.1:8b	4o	4o-mini	DSr:70b	DSr:14b	L3.1:70b	L3.1:8b
Inheritance and Overrides	33	54	34	34	48	41	32	27	26	38	55	56
Configuration Consistency	29	34	22	20	31	41	25	36	19	34	34	43
Resource Sharing	3	3	4	2	4	5	4	8	6	3	3	6
Naming Schemes	1	0	0	1	0	3	0	2	0	3	0	2
Port Mapping	0	7	6	2	2	8	1	8	4	2	2	5
Context (Availability, Retrieval, Utilization)	13	13	7	8	8	4	3	4	5	12	7	8
Independent Tech./Services	0	1	3	1	0	10	0	8	4	4	2	2
Others	3	10	7	8	10	21	3	8	8	8	13	11
Total	82	121	83	76	103	133	68	101	72	104	116	133

three types of information LLMs require to better validate dependencies: (1) project-specific information, such as details about the project structure, implementation details, and available resources; (2) a precise and unambiguous prompt; and (3) examples of correct and incorrect classifications of similar dependencies (i.e., few-shot prompting [45]). Project-specific information particularly may reduce failures in the categories *Inheritance and Overrides* and *Resource Sharing*. A revised prompt may avoid confusions about the consistency of configuration values. Since this refinement is also applicable to the vanilla LLMs, we include it in the evaluation, yielding a refined baseline. Finally, examples of similar dependencies may increases the understanding and validation abilities of LLMs for certain dependencies.

We refined RAG variant *R1* as follows: (1) For each subject system, we manually extracted the project structure, implementation details, and relevant resources from its GitHub repository. This information was then injected into the prompt at retrieval time, using the project name which is associated with the dependency under validation. (2) We refined the validation prompt to make the definition of configuration dependencies more precise and to avoid any ambiguity regarding the consistency of values. We apply this refinement also to the vanilla LLMs to have a sound comparison as the baseline may profit from this improvement as well. (3) For each failure category, we curated two representative dependencies, one correctly and one incorrectly classified, and retrieved the two most similar examples for each validation prompt using cosine similarity. We ensured that these dependencies were not present in the dataset of dependencies to prevent data leakage. To not jeopardize the validity of the results, we evaluate the final validation effectiveness of both the refined vanilla LLMs and the refined RAG variant *R1* on an unseen dataset of 150 real-world configuration dependencies. The final revised version of our validation prompt for the vanilla LLMs and RAG variant *R1* can be found on our supplementary website.

Table VII presents the final validation scores for all studied LLMs and the refined RAG variant *R1* on an unseen dataset. There are three notable observations: (1) The refined RAG variant *R1* consistently outperforms the refined vanilla LLM baseline in all cases, achieving improvements of precision by 35 % and recall by 133 % over vanilla LLMs. This clearly shows that the new information based on the failure categories

TABLE VII: Validation effectiveness of refined vanilla LLMs ("w/o") and the refined RAG variant *R1*. Δ represents the change in percent.

Models	Precision			Recall			F1-Score		
	w/o	R1	Δ	w/o	R1	Δ	w/o	R1	Δ
4o	0.91	0.94	3.3	0.34	0.80	135.3	0.50	0.87	74.0
4o-mini	0.00	0.77	N/A	0.00	0.59	N/A	0.00	0.67	N/A
DSr:70B	0.78	0.82	5.1	0.46	0.83	80.4	0.58	0.82	41.4
DSr:14B	0.66	0.87	31.8	0.31	0.67	116.1	0.42	0.75	78.6
L3.1:70b	0.75	0.80	6.7	0.25	0.74	196.0	0.37	0.77	108.1
L3.1:8b	0.59	0.83	40.7	0.43	0.39	-9.3	0.50	0.53	6.0
Mean	0.62	**0.84**	35.5	0.30	**0.70**	133.3	0.40	**0.74**	85.0
Best	0.92	**0.94**	2.2	0.46	**0.83**	80.4	0.58	**0.87**	50.0

significantly improve RAG's effectiveness for dependency validation. (2) *4o* has a high precision, but still fails to detect a large portion of actual dependencies. This discrepancy here to other models and the RAG system is substantial, emphasizing not only the importance of relevant context, but also that powerful models alone are insufficient for complex, real-world tasks. (3) *4o-mini* did not detect any true positives and false positives with the revised prompt, resulting in precision, recall, and F1-score values of zero. To verify this phenomenon, we validated the unseen configuration dependencies with *4o-mini* twice, but the results did not change. Such a catastrophic break down in performance is remarkable and we are not aware of any other study with a similar observation. Such a dependence on the prompt phrasing is dangerous especially for production systems.

Summary: Answering $RQ_{2.1}$, we identified eight distinct failure categories in validating dependencies, most of which are project- and technology specific. Hence, using information solely based on pre-training of LLMs is often insufficient in practice when specifics of projects must be taken into account. For $RQ_{2.2}$, we found that incorporating this specific information substantially increased the validation effectiveness of LLMs, with improvements of precision up to 35 % and recall up to 133 % over vanilla LLMs.

VI. Discussion

Dependency Validation Effectiveness of LLMs: The validation ability of vanilla LLMs varies significantly. But also RAG systems have, contrary to their praise, small or even negative impact on the validation performance compared to vanilla LLMs. This suggests that simply adding context information — despite being related and used by developers for the same task — does not guarantee a better validation performance. This in line with related work [46] that an unrefined RAG system (e.g., with suboptimal retrievers or context sources) can degrade the performance of LLMs. Moreover, we have not seen substantial changes in the validation performance among different RAG variants. Although we cannot rule out that our variations might be too conservative, we expect that replacing import retrieval components should have a recognizable effect on validation performance if they matter. However, it seems that the choice of contextual information is substantially more important than the RAG assembly itself.

Refinements of RAG for Dependency Validation: Poor performance of RAG systems can be caused by several factors: The retriever may fail to retrieve relevant context, the selected context sources may lack necessary information, or the prompt could be too vague, leaving room for misinterpretations. Identifying the sources of such failures is essential when developing RAG systems, which is why we emphasize the importance of a qualitative analysis of failure for every development of a RAG system. This also helps to derive specific measures and relevant changes for effective refinements of the RAG system. However, a qualitative analysis is inherently task-specific and requires domain knowledge, as well as manual effort to interpret nuanced failure patterns, making it a labor-intensive but essential component in the development of every RAG system. While partial automation (e.g., labeling support or clustering failure cases) may exist, the interpretative nature and required expertise for such qualitative analyses currently resists their full automation. For our task, clarifying the validation task alone does not guarantee better performance, it is the project-specific information and examples of similar classified dependencies available in a RAG system that achieves the highest scores.

In addition to a qualitative analysis, tuning key parameters in the RAG pipeline, such as chunk size, overlap, and splitting strategy, can also affect validation performance. In our experiments, we used a chunk size of 256 tokens with a 10-token overlap, chosen based on preliminary trials. Varying this parameter produced only marginal performance changes unless extreme values were used. Future work will explore more adaptive approaches, such as content-aware or recursive chunking, which can further improve the retrieval quality but not weaken our current results.

Temperature Sensitivity: Our results clearly show that RAG with the correct context information is a suitable technique to reliably validate configuration dependencies. However, specific factors such as the temperature may affect accuracy. To determine the temperature sensitivity of the best performing RAG system *R1*, we conducted a post-hoc sensitivity analysis. We use 10 equally distributed samples from the validation set and vary temperature from 0.0 to 1.0 in 0.2 steps. Table VIII shows the precision, recall, and F1-Score of *R1* across the LLMs and different temperature values. We see that most models exhibit stable performance with only minor fluctuations across different temperature values. In particular, both proprietary *GPT* models maintained perfect precision across all temperature values, with only slight recall variations leading to occasional perfect F1-scores. Open-source models showed greater variability, primarily in recall. For *DSr:70B* and *DSr:14B*, precision remained perfect across different temperature values, but recall fluctuated more noticeably with higher temperatures, indicating reduced sensitivity at intermediate temperatures but degradation at higher values. Similarly, *L3.1:70B* was steady until recall declined at higher temperature values, whereas the smallest model, *L3.1:8B*, remained largely stable.

Across all models, precision remained consistently high, while recall was the primary source of variation. Higher temperature values tended to reduce recall rather than precision, meaning LLMs were more likely to miss relevant dependencies than to produce false positives. This effect was negligible for both *GPT* models but more pronounced for some open-source models, suggesting that low-to-moderate temperatures are preferable for stable performance. Overall, the proprietary *GPT* models were practically insensitive to temperature changes, maintaining consistently high precision and stable recall. By contrast, the open-source models exhibited greater susceptibility to higher temperatures, where recall fluctuations led to noticeable performance degradation. This indicates a gap in robustness: while proprietary models deliver stable accuracy across temperature settings, open-source models benefit from careful temperature tuning to ensure reliable performance.

Threats to Validity: A threat to *internal validity* is potential bias in our manual labeling of configuration dependencies. We mitigated this through a rigorous review processes involving multiple authors, especially for borderline cases. We further mitigated subjectivity in our qualitative analysis of failures by independent coding of multiple researchers. Another threat to *internal validity* concerns the dynamic Web scraping. As context in the Web can change during the time of our experiments, it might introduce a time-dependent factor in our experiments. To counter this, for each dependency in the experiment, we scrape the dynamic context once and reuse it for all LLMs. Moreover, the selected chunk size and overlap used for populating the vector database is based on preliminary experiments that aimed at balancing context granularity and length. However, this only may leave further improvement for a RAG system, underpinning our proposal to use this technology for dependency validation.

The main threat to external validity concerns the generalizability of our findings beyond the Spring Boot ecosystem. We carefully selected Spring Boot projects specifically to mitigate this exact threat for several reasons: First, Spring Boot projects incorporate a broad range of technologies and address practically relevant, industry-scale challenges. Evaluating this

TABLE VIII: Sensitivity analysis for *R1* with different temperature (T) values. P = Precision, R= Recall, and F1 = F1-Score.

Models	T = 0.0			T = 0.2			T = 0.4			T = 0.6			T = 0.8			T = 1.0		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
4o	1.00	0.80	0.89	1.00	0.80	0.89	1.00	0.80	0.89	1.00	1.00	1.00	1.00	0.80	0.89	1.00	0.80	0.89
4o-mini	1.00	0.60	0.75	1.00	0.60	0.75	1.00	0.60	0.75	1.00	0.60	0.75	1.00	1.00	1.00	1.00	0.60	0.75
DSr:70B	1.00	0.80	0.89	1.00	0.60	0.75	1.00	0.80	0.89	1.00	0.80	0.89	1.00	0.60	0.75	1.00	0.60	0.75
DSr:14B	1.00	0.60	0.75	1.00	0.60	0.75	0.75	0.60	0.67	1.00	0.60	0.75	1.00	0.40	0.57	1.00	0.60	0.75
L3.1:70B	1.00	0.80	0.89	1.00	0.80	0.89	1.00	0.80	0.89	1.00	0.80	0.89	1.00	0.60	0.75	1.00	0.40	0.57
L3.1:8B	1.00	0.60	0.75	1.00	0.60	0.75	1.00	0.60	0.75	1.00	0.60	0.75	1.00	0.60	0.75	1.00	0.80	0.89
Mean	1.00	0.70	0.82	1.00	0.67	0.80	0.96	0.70	0.81	1.00	0.73	0.84	1.00	0.67	0.79	1.00	0.63	0.77
Best	1.00	0.80	0.89	1.00	0.80	0.89	1.00	0.80	0.89	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.80	0.89

ecosystem is therefore comparable to other enterprise ecosystems, such as Python/Django, with their similarly structured *settings.py* and *.env* files. More importantly, providing a solution just for the Spring Boot ecosystem alone has already an impact on practice.

Second, focusing on a single ecosystem increases the reliability of our ground truth and ensures a sound evaluation. Since no validated dataset of configuration dependencies exists, creating one demands deep expert knowledge across multiple frameworks and a substantial number of repositories per ecosystem, a combination that is infeasible given the runtime and cost of evaluation. Our study thus delivers the largest manually annotated dataset of configuration dependencies for a practically relevant ecosystem.

Third, our entire RAG architecture is designed to be technology-agnostic: the ingestion, retrieval, and generation pipelines are not tied to any specific framework. This contrasts with classical ML or LM methods (e.g., BERT), which require ground truth training data for fine-tuning, which is not only non-existent but also infeasible for a practical approach, considering the number of technologies out there. By dynamically ingesting context sources, our approach can adapt to other ecosystems without retraining, and we, thus, argue that this even improves generalizability compared to existing techniques.

Another threat to *external validity* comes from the selected LLMs and embedding models. We selected models to provide a representative evaluation across the current state of the art, including flagship proprietary LLMs, leading open-source alternatives, and the best-performing embedding models available at the time of experimentation to ensure a robust and timely evaluation.

The threats to *construct validity* arise from using standard but popular classification metrics, which may not capture all nuances of validation quality. We addressed this by complementing quantitative analysis with qualitative examination of failures. A similar threat represents the relevance score provided by Pinecone, which computes the vector similarity of the query with the retrieved context. As the dot product is used for the relevance score, this might not always align with the semantic similarity, as recent studies on the normalized dot product has shown [47]. Since we also found a discrepancy between the score and the actual performance (i.e., negative relevance score for the best RAG), we are more in line with [47] and would resort from over-interpreting these numbers. This also supports our caution against automatically computed RAG metrics, such as RAGAs.

VII. Conclusion

Validating configuration dependencies is a complex real-world task. Recent advancements in LLMs offer a promising direction for automated, technology-agnostic dependency validation. However, prior work highlights that LLMs struggle in reliably validating configuration dependencies. We propose a novel RAG system for dependency validation that integrates project- and technology-specific context sources and evaluate whether it can improve LLM-based validation of configuration dependencies, as well as what contextual information are needed to overcome the static knowledge base of LLMs.

Our evaluation shows that vanilla LLMs miss many dependencies, but are less prone to misclassifications than RAG. However, when enriched with special information derived from a qualitative analysis of validation failures, such as project-specific details, precise and unambiguous prompts, as well as examples of similar dependencies, our proposed RAG system achieves substantially more accurate validation results for all models, with improvements of precision by 35 % and recall by 133 % over vanilla LLMs. The key takeaway is that merely adding context does not automatically enhance the validation effectiveness of LLMs. Instead, a systemic refinement of RAG components and particularly context sources is necessary to unleash the full potential of RAG for configuration dependency validation.

VIII. Acknowledgements

This work was supported by the Saxon State Ministry for Science, Culture and Tourism (SMWK) through the "Center for Scalable Data Analytics and Artificial Intelligence Dresden/Leipzig" (ScaDS.AI) and by the European Social Fund (ESF) together with the German state of Saxony under grant number A.100760692 (Project: Teaching-AI).

References

[1] M. Sayagh, N. Kerzazi, and B. Adams, "On cross-stack configuration errors," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, p. 255–265.

[2] N. Siegmund, N. Ruckel, and J. Siegmund, "Dimensions of software configuration: on the configuration context in modern software development," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2020, p. 338–349.

[3] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, "Understanding and discovering software configuration dependencies in cloud and datacenter systems," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2020, p. 362–374.

[4] A. Hubaux, Y. Xiong, and K. Czarnecki, "A user survey of configuration challenges in linux and ecos," in *Proceedings of the International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS)*. ACM, 2012, p. 149–155.

[5] D. Jin, X. Qu, M. B. Cohen, and B. Robinson, "Configurations everywhere: implications for testing and debugging in practice," ser. ICSE Companion 2014. ACM, 2014, p. 215–224.

[6] A. Rabkin and R. Katz, "Static extraction of program configuration options," in *Proceedings of the International Conference on Software Engineering*. ACM, 2011, p. 131–140.

[7] B. Maurer, "Fail at scale: Reliability in the face of rapid change," *Queue*, vol. 13, no. 8, p. 30–46, 2015.

[8] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic configuration management at facebook." ACM, 2015, p. 328–343.

[9] S. Mehta, R. Bhagwan, R. Kumar, C. Bansal, C. Maddila, B. Ashok, S. Asthana, C. Bird, and A. Kumar, "Rex: preventing bugs and misconfiguration in large services using correlated change analysis," in *Proceedings of the Usenix Conference on Networked Systems Design and Implementation*. USENIX Association, 2020, p. 435–448.

[10] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the ACM Symposium on Operating Systems Principles*. ACM, 2011, p. 159–172.

[11] T. Xu and Y. Zhou, "Systems approaches to tackling configuration errors: A survey," *ACM Computing Surveys*, vol. 47, no. 4, jul 2015.

[12] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing? lessons from hundreds of service outages," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2016, p. 1–16.

[13] A. Rabkin and R. H. Katz, "How hadoop clusters break," *IEEE Software*, vol. 30, no. 4, pp. 88–94, July 2013.

[14] S. Simon, N. Ruckel, and N. Siegmund, "Cfgnet: A framework for tracking equality-based configuration dependencies across a software project," *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 3955–3971, Aug 2023.

[15] W. Chen, H. Wu, J. Wei, H. Zhong, and T. Huang, "Determine configuration entry correlations for web application systems," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, June 2016, pp. 42–52.

[16] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proceedings of the TACM Symposium on Operating Systems Principles*. ACM, 2013, p. 244–259.

[17] P. Huang, W. J. Bolosky, A. Singh, and Y. Zhou, "Confvalley: a systematic configuration validation framework for cloud services," in *Proceedings of the European Conference on Computer Systems*. ACM, 2015, pp. 1–16.

[18] S. Baset, S. Suneja, N. Bila, O. Tuncer, and C. Isci, "Usable declarative configuration specification and validation for applications, systems, and cloud," in *Proceedings of the ACM/IFIP/USENIX Middleware Conference: Industrial Track*. ACM, 2017, pp. 29–35.

[19] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac, "Synthesizing configuration file specifications with association rule learning," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–20, 2017.

[20] M. Santolucito, E. Zhai, and R. Piskac, "Probabilistic automated language learning for configuration files," in *Proceedings of the Internation Conference on Computer Aided Verification*. Springer, 2016, pp. 80–87.

[21] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "Encore: exploiting system environment and correlation information for misconfiguration detection," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2014, p. 687–700.

[22] J. Lao, Y. Wang, Y. Li, J. Wang, Y. Zhang, Z. Cheng, W. Chen, M. Tang, and J. Wang, "Gptuner: A manual-reading database tuning system via gpt-guided bayesian optimization," *arXiv preprint arXiv:2311.03157*, 2023.

[23] X. Huang, H. Li, J. Zhang, X. Zhao, Z. Yao, Y. Li, Z. Yu, T. Zhang, H. Chen, and C. Li, "Llmtune: Accelerate database knob tuning with large language models," *arXiv preprint arXiv:2404.11581*, 2024.

[24] Z. Liu, Y. Tang, M. Li, X. Jin, Y. Long, L. F. Zhang, and X. Luo, "Llm-compdroid: Repairing configuration compatibility bugs in android apps with pre-trained large language models," *arXiv preprint arXiv:2402.15078*, 2024.

[25] S. Shan, Y. Huo, Y. Su, Y. Li, D. Li, and Z. Zheng, "Face it yourselves: An llm-based two-stage strategy to localize configuration errors via logs," *arXiv preprint arXiv:2404.00640*, 2024.

[26] Z. Wang, D. J. Kim, and T.-H. Chen, "Identifying performance-sensitive configurations in software systems through code analysis with llm agents," *arXiv preprint arXiv:2406.12806*, 2024.

[27] C. Bird, D. Ford, T. Zimmermann, N. Forsgren, E. Kalliamvakou, T. Lowdermilk, and I. Gazit, "Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools," *Queue*, vol. 20, no. 6, p. 35–57, 2023.

[28] X. Lian, Y. Chen, R. Cheng, J. Huang, P. Thakkar, and T. Xu, "Configuration validation with large language models," *arXiv preprint arXiv:2310.09690*, 2023.

[29] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in neural information processing systems*, vol. 33, pp. 9459–9474, 2020.

[30] V. Ramachandran, M. Gupta, M. Sethi, and S. R. Chowdhury, "Determining configuration parameter dependencies via analysis of configuration data from multi-tiered enterprise applications," in *Proceedings of the International Conference on Autonomic Computing*. ACM, 2009, p. 169–178.

[31] I. Trummer, "Db-bert: A database tuning tool that "reads the manual"," in *Proceedings of the International Conference on Management of Data*. ACM, 2022, p. 190–203.

[32] Pinecone, "Pinecone vector database," https://www.pinecone.io/, 2025, accessed: 2025-05-28.

[33] J. Wei, X. Wang, D. Schuurmans, M. Bosma, b. ichter, F. Xia, E. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," vol. 35, pp. 24 824–24 837, 2022.

[34] O.-M. Camburu, B. Shillingford, P. Minervini, T. Lukasiewicz, and P. Blunsom, "Make up your mind! adversarial generation of inconsistent natural language explanations," *arXiv preprint arXiv:1910.03065*, 2019.

[35] Y. Elazar, N. Kassner, S. Ravfogel, A. Ravichander, E. Hovy, H. Schütze, and Y. Goldberg, "Measuring and improving consistency in pretrained language models," *Transactions of the Association for Computational Linguistics*, vol. 9, pp. 1012–1031, 2021.

[36] X. Wang, J. Wei, D. Schuurmans, Q. V. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," in *Proceedings of the International Conference on Learning Representations (ICLR)*. OpenReview.net, 2023. [Online]. Available: https://openreview.net/forum?id=1PL1NIMMrw

[37] S. Simon, A. Mailach, J. Dorn, and N. Siegmund, "A methodology for evaluating rag systems: A case study on configuration dependency validation," *arXiv preprint arXiv:2410.08801*, 2024.

[38] GitHub, "Github topics," https://github.com/topics, 2025, accessed: 2025-05-28.

[39] X. Lian, Y. Chen, R. Cheng, J. Huang, P. Thakkar, M. Zhang, and T. Xu, "Large language models as configuration validators," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2025, pp. 1704–1716.

[40] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J. Wen, "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, no. 6, p. 186345, 2024. [Online]. Available: https://doi.org/10.1007/s11704-024-40231-1

[41] T. Xu, V. Pandey, and S. Klemmer, "An hci view of configuration problems," *arXiv preprint arXiv:1601.01747*, 2016.

[42] X. Xia, L. Bao, D. Lo, P. S. Kochhar, A. E. Hassan, and Z. Xing, "What do developers search for on the web?" *Empirical Software Engineering*, vol. 22, pp. 3149–3185, 2017.

[43] J. Josyula, S. Panamgipalli, M. Usman, R. Britto, and N. B. Ali, "Software practitioners' information needs and sources: A survey study," in *International Workshop on Empirical Software Engineering in Practice (IWESEP)*. IEEE, 2018, pp. 1–6.

[44] S. Es, J. James, L. Espinosa-Anke, and S. Schockaert, "Ragas: Automated evaluation of retrieval augmented generation," *arXiv preprint arXiv:2309.15217*, 2023.

[45] T. B. Brown, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[46] N. Jain, R. Kwiatkowski, B. Ray, M. K. Ramanathan, and V. Kumar, "On mitigating code llm hallucinations with api documentation," *arXiv preprint arXiv:2407.09726*, 2024.

[47] H. Steck, C. Ekanadham, and N. Kallus, "Is cosine-similarity of embeddings really about similarity?" in *Companion Proceedings of the ACM Web Conference 2024*. ACM, 2024, p. 887–890.