Developing a Llama-Based Chatbot for CI/CD Question Answering: A Case Study at Ericsson

by

Daksh Chaudhary

Thesis submitted to the University of Ottawa

in partial fulfillment of the requirements for the degree of

Master of Computer Science (Concentration in Applied Artificial Intelligence)

in

School of Electrical Engineering and Computer Science

University of Ottawa

Ottawa, Ontario, Canada

© Daksh Chaudhary, Ottawa, Canada, 2024

Abstract

This thesis presents our experience developing a Llama-based chatbot for question answering about continuous integration and continuous delivery (CI/CD) at Ericsson, a multinational telecommunications company. Our chatbot is designed to handle the specificities of CI/CD documents at Ericsson, employing a retrieval-augmented generation (RAG) model to enhance accuracy and relevance. Our empirical evaluation of the chatbot on industrial CI/CD-related questions indicates that an ensemble retriever, combining BM25 and embedding retrievers, yields the best performance. When evaluated against a ground truth of 72 CI/CD questions and answers at Ericsson, our most accurate chatbot configuration provides fully correct answers for 61.11% of the questions, partially correct answers for 26.39%, and incorrect answers for 12.50%. Through an error analysis of the partially correct and incorrect answers, we discuss the underlying causes of inaccuracies and provide insights for further refinement. We also reflect on lessons learned and suggest future directions for further improving our chatbot's accuracy.

Acknowledgements

If I have contributed in any way to research at the intersection of Software Engineering and Machine Learning, I owe all the credit to Professors Mehrdad Sabetzadeh and Shiva Nejati. This thesis would not have been possible without their continued support and guidance over the past few years. I would also like to thank my colleagues at Ericsson for providing the necessary tools and a conducive environment for conducting my research.

I am grateful to Professor Olga Baysal for serving as an examiner for my thesis defense. I also extend my thanks to the anonymous reviewers of the ICSME 2024 Industry Track for their insightful feedback on the research paper.

Finally, I want to take this opportunity to thank my loving family: my mom, Raj Chaudhary; dad, Rajendra Kumar Chaudhary; sister, Vrinda Chaudhary; brother, Maulik Chaudhary; and partner, Vaishnavi Prabhu. I owe my achievements and the person I have become to their unwavering love, support, and belief in me.

Table of Contents

List of Tables viii

List of Figures ix

Abbreviations xi

1 Introduction 1

1.1 Overview 1

1.2 Industrial Context and Motivation 3

1.3 Novelty 5

1.4 Significance 5

1.5 Thesis Structure. 6

2 Background 7

2.1 Continuous Integration and Continuous Delivery (CI/CD) 8

2.1.1 Introduction to CI/CD 8
2.1.2 CI/CD pipeline architecture 9
2.1.3 Deployment Pipeline Components and Tools 14
2.1.4 Benefits and Challenges of CI/CD 17
2.1.5 CI/CD in Ericsson 20
2.2 Deep Learning 21
2.2.1 Introduction to Deep Learning 21
2.2.2 Perceptron 22
2.2.3 The Multi-Layered Perceptron (MLP) 23
2.2.4 Neural Networks (NNs) 24
2.2.5 Recurrent Neural Networks (RNNs) 24
2.2.5.1 Long Short-Term Memory (LSTM) Networks 27
2.2.6 Transformer Model 29
2.2.6.1 Model Architecture 30
2.2.6.2 Scaled Dot-Product Attention 31
2.2.6.3 Multi-Head Attention 33
2.2.6.4 Encoder and Decoder Stacks 34
2.3 Natural Language Processing (NLP) 35
2.3.1 Introduction to NLP 35

2.3.2 Word Embeddings 36
2.3.3 Language Models 37
2.4 Retrieval-Augmented Question-Answering 39
2.5 Related Work 41
2.5.1 Chatbots in Software Engineering 41
2.5.2 Large Language Models (LLMs) in Software Engineering 43

3 Approach **45**
3.1 Corpus Creation 46
3.2 Chatbot Design 50

4 Empirical Evaluation **59**
4.1 Research Questions 59
4.2 Implementation 60
4.3 Experimental Setup 61
4.4 Ground Truth 61
4.5 Domain Corpus 62
4.6 Metrics 62
4.6.1 Metrics for RQ1 62
4.6.2 Metric for RQ2 63

4.7 Evaluation Procedure 64
4.8 Answers to RQs 66
4.9 Limitations and Validity Considerations 72

5 Conclusions **75**
5.1 Lessons Learned 75
5.2 Closing Thoughts 78

References **81**

Appendix **92**
A.1 Parameters for Microsoft Teams Messages 92
A.2 Parameters for Microsoft Teams Replies 93

List of Tables

4.1 Automatically Computed Accuracy Results 66

4.2 Results of Manual Analysis 67

List of Figures

2.1 Architecture of CI/CD Pipeline, as originally presented in [81] 10

2.2 An overview of the deployment pipeline components and associated tools, as originally presented in [75] 14

2.3 RNN architecture and its corresponding computational graph used for computing the training loss [29] 25

2.4 Block diagram of the LSTM recurrent network cell, as originally presented in [29] 28

2.5 Architecture of the Transformer model, as originally presented in [82] . . . 30

2.6 Scaled Dot-Product Attention, as originally presented in [82] 31

2.7 Multi-Head Attention which consists of several attention layers running in parallel, as originally presented in [82]. 33

3.1 Steps for Creating a (Domain-specific) CI/CD Corpus 46

3.2 Overview of Our Chatbot Design 50

3.3 Prompt Template for Query Rewriting 53

3.4 Examples of Query Rewriting 54

3.5 Example of Contextual Compression 55

3.6 Prompt Template for Answer Generation 57

3.7 Example of a User Query and Response Interaction 58

4.1 Results of Error Analysis, including Root Causes of Inaccuracies and Their Prevalence in Our Case Study 68

4.2 Response Times for Chatbot Instances Using Different Retrievers 70

Abbreviations

AI Artificial Intelligence 7, 35–37

API Application Programming Interface 47

BERT Bidirectional Encoder Representations from Transformers 37, 38, 40, 42

CD Continuous Delivery 3, 9, 16

CI Continuous Integration 3, 4, 8, 9, 11, 13, 15–17

CI/CD Continuous Integration and Continuous Delivery 1–5, 7–11, 14, 17, 18, 20, 21, 44–46, 62, 71, 78, 80

CloudRAN Cloud Radio Access Network 3, 4

CNN Convolutional Neural Network 24, 29

CSP Communication Service Provider 20

CSV Comma-separated values 47

GPT Generative Pre-trained Transformer 40, 41

HTML Hypertext Markup Language 47

IaC Infrastructure as Code 12

IR Information Retrieval 37

LLM Large Language Model vi, 1, 2, 5, 38–45, 47–49, 51, 52, 56, 57, 60, 64, 69, 71–73, 76, 79

LSTM Long Short-Term Memory 24, 27–29, 38

MLP Multi-Layered Perceptron 23, 24, 38

NLP Natural Language Processing 7, 29, 35, 36

NN Neural Network 22, 24, 38

QA Question-Answering 39–42

RAG Retrieval-Augmented Generation 2, 3, 5, 39–42, 44, 60, 77

RAN Radio Access Network 3

ReLU Rectified Linear Unit 23, 34

REST Representational State Transfer 47

RNN Recurrent Neural Network 24–27, 29, 35, 38

Chapter 1

Introduction

In this chapter, we lay the foundation for this thesis by offering a general overview of the domain, our chatbot development methodology, the underlying technologies, and the evaluation results in Section 1.1. Section 1.2 provides an overview of Ericsson and the specific unit where the work was conducted, highlighting the motivation for developing a chatbot within the Continuous Integration and Continuous Delivery (CI/CD) domain. We highlight the novelty and significance of this work in Sections 1.3 and 1.4, respectively. Finally, Section 1.5 outlines the structure of the thesis.

1.1 Overview

With advances in LLMs, the high-tech industry is increasingly looking into how chatbots can improve software engineering practices. There have been existing attempts to employ LLM-based chatbots in the domain of software engineering. Among others, Abdellatif et

al. [1, 2] and Daniel & Cabot [14] explore various facets of integrating chatbots into software engineering workflows, highlighting the potential for chatbots to streamline processes, enhance communication, and assist in tasks ranging from bug tracking and documentation to code generation and quality assurance.

This thesis presents our experience developing a Llama-based chatbot for answering questions related to CI/CD at Ericsson, a multinational telecommunications company. Ericsson employs agile development and DevOps across various projects, requiring many engineers to work efficiently with CI/CD processes.

CI/CD enables automated testing, integration, and deployment of code changes [6, 39, 71]. CI/CD is intrinsically linked to software maintenance and evolution, ensuring that software remains functional and up-to-date as new features and fixes are continuously integrated into the codebase. Our chatbot is designed to handle the contextual factors specific to CI/CD documents at Ericsson. This includes the evolving content of guidelines and team conversations about CI/CD.

To build an accurate chatbot, we opt for a Retrieval-Augmented Generation (RAG) model [52]. RAG enhances chatbot capabilities by combining retrieval of relevant documents with the generative power of LLMs, thereby providing more accurate and relevant responses. RAG presents two main advantages over fine-tuning a model on domain-specific corpora [22]: First, fine-tuning requires a labeled dataset, which can be expensive to build. Second, a fine-tuned model is prone to outdated knowledge; this issue can be mitigated through a RAG model that continuously accesses and retrieves up-to-date information.

We evaluate our chatbot on industrial CI/CD questions. We experiment with alter-

native retriever models for instantiating a RAG pipeline over Llama 2 and empirically evaluate the accuracy of the resulting pipelines. Our evaluation indicates that an ensemble retriever, combining BM25 [70] and embeddings [7, 63], leads to the best overall outcome. Specifically, by using an ensemble retriever and evaluating the chatbot against a ground truth of 72 CI/CD questions and answers at Ericsson, we obtain fully correct answers for 61.11% of the questions, partially correct answers for 26.39%, and incorrect answers for 12.50%. Following this evaluation, we conduct an error analysis on partially correct and incorrect answers to identify the root causes of the inaccuracies.

1.2 Industrial Context and Motivation

Ericsson is a multinational company specializing in providing ICT services and equipment to telecommunications operators and enterprises worldwide. Our chatbot was developed within Ericsson's Cloud Radio Access Network (CloudRAN) unit. This unit focuses on online and virtualized central system observability and monitoring solutions across cloud-native Radio Access Network (RAN) deployments, including software microservices on containers-as-a-service (CaaS) infrastructure.

CloudRAN employs CI/CD to automate code integration, testing, and delivery. The CI/CD process at CloudRAN adheres to industry best practices and follows a standard workflow [17]: All code is stored in a version-control system, with developers working on feature branches. When code is committed, a Continuous Integration (CI) server triggers automated builds and tests to provide instantaneous feedback. Successful builds produce artifacts for deployment. During the Continuous Delivery (CD) process, code that passes

CI tests is deployed to a staging environment for additional testing and manual review. CloudRAN has an approval process in place before deploying to production. Monitoring tools and centralized logging systems track application performance and detect issues. This iterative process ensures frequent, reliable code integration and delivery, reducing errors and downtime while enabling rapid feedback and improvements.

Our chatbot aims to improve CI/CD at CloudRAN by enabling software engineers, both within the unit and at client sites, to obtain answers to their CI/CD-related questions. A few examples of queries that CloudRAN would like the chatbot to be able to respond to are: *(1) What are the steps to release a microservice? (2) How can I modify test targets during staging? (3) How can I migrate from [cluster 1] to [cluster 2]? (4) How do I add a test channel to a Jenkins pool?*

We note that, both in the above queries and in the examples provided throughout the rest of the thesis, we have altered the content from its original form to preserve confidentiality, while ensuring that the substance remains unchanged. Any redacted text is indicated by square brackets ([]).

By handling routine queries, such as our illustrative examples above, the chatbot offers the potential to free up expert engineering resources to address more complex issues. This reduces operational costs, speeds up issue resolution, and allows experts to focus on tasks that require specialized skills.

The dynamic nature of CloudRAN's operations, which includes reliance on constantly evolving internal documents and team communications, is an important contextual factor to consider in the design of the chatbot to ensure its longevity. To this end, we employ

RAG to incorporate up-to-date information from team workspaces and messaging channels, providing answers based on the most recent knowledge.

1.3 Novelty

The novelty of our work lies in providing practical insights into the readiness of chatbot technologies for question answering in a complex and specialized yet dynamic setting, where the content from which answers are derived is fluid and changes over time. To the best of our knowledge, we are the first to report on the construction and evaluation of a chatbot for answering CI/CD questions in an industrial context.

This thesis builds upon the research presented in the paper, *'Developing a Llama-Based Chatbot for CI/CD Question Answering: A Case Study at Ericsson'* [11], where I served as the first author and major contributor. The paper was presented at the industry track of the *International Conference on Software Maintenance and Evolution* 2024, further validating the novelty of this work.

1.4 Significance

Despite recent advances in generative LLMs that have made chatbot construction more accessible, the field remains marked by hype, generally lacks empirical analysis of accuracy, and does not sufficiently elaborate on technical considerations that could have a make-or-break effect on chatbot efficacy. Our work highlights our main technical choices

for chatbot design, aiming to assist other researchers and practitioners facing similar challenges. Furthermore, we provide an empirically grounded examination of chatbot accuracy for a software-engineering problem in industry. This contributes to the development of a scientific body of knowledge that facilitates wider deployment of software-engineering chatbots.

1.5 Thesis Structure.

The rest of this thesis is structured as follows: Chapter 2 provides background information. Chapter 3 describes the technical approach that underlies our chatbot. Chapter 4 reports on the evaluation of our chatbot. Chapter 5 concludes the thesis by reflecting on the lessons learned throughout the course of this R&D project and discussing potential future work.

Chapter 2

Background

This chapter provides essential knowledge of CI/CD in software development and relevant Artificial Intelligence (AI) topics. We cover key developments in deep learning leading up to the Transformer model and offer a comprehensive overview of the Transformer model itself. Understanding this model, along with the provided Natural Language Processing (NLP) concepts, equips the reader to grasp the underlying functionality of both the retrieval and generation components of our chatbot. Additionally, we provide an overview of retrieval-augmented question-answering and discuss existing chatbots in software engineering in the related work section.

Please note that Section 2.2 is intended to serve as a detailed reference for the generative AI mechanisms in our work. Readers more interested in the practical applications of the thesis may wish to skip this section.

2.1 Continuous Integration and Continuous Delivery (CI/CD)

2.1.1 Introduction to CI/CD

Continuous Integration and Continuous Delivery are software engineering practices that enable organizations to build, test and deploy software features at an accelerated pace. CI/CD practices allow for improved product quality and faster time to the production environment, which ultimately leads to enhanced customer satisfaction. Although Continuous Delivery and Continuous Deployment differ in how they handle final deployments - manually in Continuous Delivery and automatically in Continuous Deployment - we use these terms interchangeably in our study [75] [81].

CI is a software development practice that involves frequently updating a shared version control system with new code changes. This practice automates the build and test processes for each code change made in the main branch, ensuring that new code integrates reliably with existing code and that any integration issues are detected early. Early detection of issues provides faster feedback to developers via a feedback mechanism (e.g., an email containing a detailed report on the latest build) and improves resolution time. In addition to creating a build and running tests, the CI process can involve further automated steps to ensure adherence to code quality standards set by the team, such as adherence to a code coverage metric threshold. CI requires the following four features: (1) a version control repository, (2) a build script, (3) a feedback mechanism, and (4) a process for integrating the source code changes (either manual or using CI/Integration server). Typically, a team

following CI practices should have all the engineers push their code changes to the main branch (not the feature branch) on a daily basis. Moreover, every code change should trigger automated unit tests and a build process that stores a build artifact in the central repository. Therefore, CI leads to faster development cycles by ensuring that the code changes can be made frequently in a reliable manner within the development environment [81] [16] [38] [39].

CD extends CI to automate the delivery of code changes from the main branch of the version control system to the production environment. CD is based on the principle that the mainline code, i.e., the code in the main branch of the version control repository, is always in a production-ready state and can be deployed to the users. Therefore, CI serves as a fundamental principle for achieving CD. CD primarily utilizes the deployment pipeline pattern to automate the execution of build processes and various types of tests, such as unit tests, functional acceptance tests, component tests, and system tests. A deployment pipeline should build packages only once per code change and use them for testing in all environments - development, testing, staging, and production. This ensures that the build deployed in the production environment has undergone comprehensive testing and meets both the functional and non-functional requirements of the product [81] [38] [39].

2.1.2 CI/CD pipeline architecture

Figure 2.1 illustrates the architecture of a generic CI/CD pipeline flow as described in [81]. Steps belonging to the continuous integration process are highlighted in blue, while steps belonging to the continuous delivery phase are highlighted in green. Each step of this

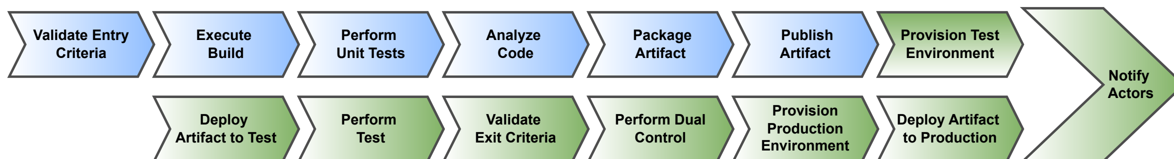

Figure 2.1: Architecture of CI/CD Pipeline, as originally presented in [81]

pipeline is discussed in detail below.

Step 1: Validate Entry Criteria. A CI/CD pipeline can be executed manually or triggered by external events in different systems. These triggers can be scheduled events or code change activities such as push, merge, or tag operations in version control systems like Git [26]. The validate entry criteria step ensures that the pipeline is executed by the right trigger with valid trigger data and correct pipeline configurations. Therefore, this validation step allows for a more reliable and consistent pipeline. Additionally, early detection of bottlenecks helps mitigate risk by addressing issues before they propagate through the pipeline. Examples of entry validation steps include verifying that all pipeline configuration variables are properly configured, required external systems are reachable, and the trigger is valid [81].

Step 2: Execute Build. This automated step involves the compilation and build process of the code to create artifacts, such as .jar files for *Java* code and .exe files for *C++* code [81].

Step 3: Perform Units Tests. This automated step involves testing individual units of the product/application, such as functions, to ensure that each unit performs as expected. Unit tests focus on verifying the functionality of each isolated unit without considering the

overall functionality of the application. If any unit test fails, the CI pipeline halts and the developers receive immediate feedback about the specific units causing an issue [81].

Step 4: Analyze Code. In addition to testing units of the code, CI pipelines often include certain code quality standards that are set by organizations. These standards may include bug detection and code quality checks using static analyzer tools like SonarQube [77], software composition analysis to identify vulnerable open-source packages using tools like JFrog Xray [42], scans to identify credentials hard-coded in the application, and validation of pipeline scripts [81].

Step 5: Package Artifact. This step involves all the activities performed to deliver the build artifact to different environments such as testing and production. One of these activities includes generating and storing a digital signature with this artifact. This process allows software engineers to track the artifact and ensure that it has undergone all the required testing and validation steps before it can be deployed in the production environment [81].

Step 6: Publish Artifact. Once the build artifact has been suitably packaged for different environments, it is stored in an immutable binary repository. This stored artifact is then retrieved by the subsequent steps in the CI/CD pipeline. Publishing an artifact is generally considered to be the last step in the continuous integration phase, with the next steps belonging to continuous delivery. The process of storing the build artifact centrally (along with a digital signature) and using it for the subsequent testing and validation in continuous delivery ensures that the build is generated once per code change and used

throughout the pipeline before deployment to the production environment [81].

Step 7: Provision Test Environment. This step involves provisioning an environment that can be used to perform various tests on the build artifact before it is deployed to production. The test environment can be either ephemeral or permanent. Ephemeral test infrastructure is generally created using Infrastructure as Code (IaC) and is temporary, meaning the environment can be created and destroyed for each test run. This approach is cost-efficient as the software engineering team pays only for the infrastructure they utilize. However, ephemeral infrastructure may not be suitable for certain test types, such as load and performance tests, which often require a large database that can take a long time to set up and tear down. In such cases, software engineers might benefit from a more permanent testing environment. Permanent testing environments are beneficial for tests that involve large datasets or long setup times and for tests that are run frequently and do not require frequent creation and deletion of the environment [81].

Step 8: Deploy Artifact to Test. In this step, the build artifact is deployed to all provisioned test environments for the required testing processes [81].

Step 9: Perform Tests. In this step, the artifact is subjected to various tests to ensure that the application is working correctly and fulfills all functional and non-functional requirements. This step can include different types of testing, such as component testing, integration testing, load and performance testing, usability testing, and pre-production testing. These tests can be executed independently, either sequentially or in parallel, to streamline the validation process. Note that unit testing is not performed in this stage, as

it has its own dedicated step in the CI phase [81].

Step 10. Validate Exit Criteria. This step validates the artifact against certain conditions set by the team before it can be deployed to the production environment. Examples of such conditions include: (a) The artifact must be tagged and versioned so that it can be properly identified after being deployed in production. (b) The artifact must have been built from the code present in the main branch or the release branch of the version control system. (c) The artifact must have passed all test and validation steps. (d) The artifact must not be expired. Even if an old artifact passes the test and validation steps, deploying it in production might pose risks. (e) The artifact's version should be greater than the version of the artifact currently deployed in the production environment [81].

Step 11: Perform Dual Control. Dual Control refers to the process where a team member (usually the manager or the product owner) is designated as the approver of artifacts before they can be deployed in production. This manual step allows for final verification by the designated person. Moreover, this step differentiates continuous delivery from continuous deployment, as having an automated step to deploy the artifact in production would classify this pipeline as a continuous deployment pipeline [81].

Step 12: Provision Production Environment. This step involves performing similar activities as during the provisioning of the test environment, but with specific configurations tailored to the production environment [81].

Step 13: Deploy Artifact to Production. If all the previous steps are completed successfully, the selected artifact is installed in the production environment [81].

Step 14: Notify All Actors. This final step involves notifying all the concerned members

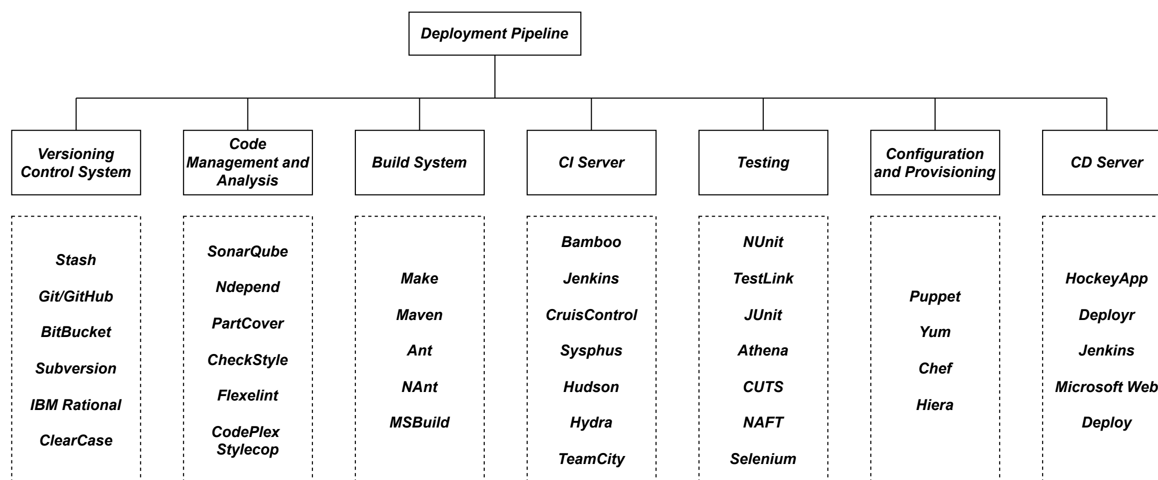

Figure 2.2: An overview of the deployment pipeline components and associated tools, as originally presented in [75]

associated with the product regarding the success or failure of the deployment pipeline [81].

2.1.3 Deployment Pipeline Components and Tools

In this section, we examine the different components of CI/CD and the commonly utilized tools associated with each component, as outlined by Shahin et al. [75]. Note that this list is not exhaustive, and the selection of components and their associated tools may vary based on the specific use case. Figure 2.2 displays the seven pipeline components: (1) Version Control System, (2) Code Management and Analysis, (3) Build System, (4) CI Server, (5) Testing, (6) Configuration and Provisioning, and (7) CD Server. Below, we provide an overview of each component's functionality and the associated tools.

(1) Version Control System. A version control system manages changes to source code

over time. It enables multiple developers to collaborate on a project by tracking individual changes, maintaining a history of code revisions, and facilitating the merging of code from different branches. Git/GitHub [26] and Subversion [79] are the most popular version control systems.

(2) Code Management and Analysis. Code management and analysis tools are essential for ensuring code quality. These tools offer functionalities such as static code analysis, measurement of code coverage, and code reviews. By automatically validating code against organizational standards and identifying potential issues, these tools enable software developers to maintain high code quality and enhance overall software maintainability. SonarQube [77] is a commonly used code analysis tool, while Gerrit [25] is popular for code management.

(3) Build System. Build systems automate the process of compiling source code and generating executable artifacts, such as '.jar' and '.exe' files. They manage dependencies, perform code compilation, and package the build outputs for subsequent stages in the pipeline. Maven [60], Ant [4], and Gradle [30] are among the popularly employed build tools in software development.

(4) CI Server. A CI server continuously monitors the mainline branch of version control system for code changes and orchestrates the process of generating code artifacts, executing unit test cases, and providing feedback to the developers. Jenkins [41] is a commonly

used CI server tool. In addition to triggering the CI activities, Jenkins provides the ability to deploy artifacts in different environments such as testing, staging, and production.

(5) Testing. The testing component enables the automation of various tests, including unit tests, integration tests, and end-to-end tests. Automated testing plays a crucial role in verifying that new code changes do not introduce defects, thereby maintaining software quality and reliability throughout the development lifecycle. Popular tools include JUnit [43], NUnit [64], and Selenium [73]

(6) Configuration and Provisioning. Configuration and provisioning tools manage the setup and maintenance of environments that are used for building, testing, and deploying software. Popular tools include Puppet [67] and Chef [12].

(7) CD Server. The CD Server automates the deployment of software artifacts to various environments, such as staging and production. Due to its advanced functionality, Jenkins [41] is also commonly used as a CD server.

2.1.4 Benefits and Challenges of CI/CD

In this section, we briefly present the general benefits and challenges associated with employing CI/CD in the software engineering field.

Benefits.

According to Duvall et al. [16], CI/CD offers the following high-level benefits:

- **Reduce Risks:** CI/CD executes frequent tests and validations on code changes, enabling early detection of software defects. This prevents issues from escalating in later deployment stages, where resolving them becomes more complex. Additionally, CI practices help engineers monitor the health attributes of their software, such as code complexity. This constant insight enables proactive monitoring of software quality and performance indicators.

- **Reduce Repetitive Processes:** CI/CD automates several repetitive processes throughout the software development cycle, saving time and cost.

- **Generate deployable software at any time and at any place:** With frequent code updates to the version control system and automated validation steps, the core CI/CD principle of always maintaining the code in a production-ready state is fulfilled. This ensures faster time to production and maximizes the value provided to customers.

- **Enable Better Project Visibility:** The frequent testing and integration in CI/CD results in the generation of substantial data, including build status information and

project metrics. This data enables the team to identify project bottlenecks and make informed decisions.

Challenges.

Laukkanen et al. [49] provide a systematic overview of the common CI/CD adoption challenges reported in the bibliographical database. These challenges can be grouped into the following themes:

- **Build design problems:** This theme is associated with issues caused by poor build design. It comprises two types of issues:
  (a) Complex build: The build process is complex, and
  (b) Inflexible build: The build process is inflexible.

- **System design problems:** This theme is associated with issues caused by poor system design decisions and consists of the following four types of issues:
  (a) System modularization: The system is decomposed into multiple units,
  (b) Unsuitable architecture: The system architecture is unsuitable and leads to time consuming testing, greater development efforts, and problematic deployment.
  (c) Internal dependencies: The software components have internal dependencies, and
  (d) Database schema changes: Any software change requires changes in the database schema itself

- **Integration problems:** This theme is associated with challenges that occur when the developers integrate their source code into the main branch of the version control system. Different types of issues within this theme are as follows:

(a) Large commits: Large commits require a substantial amount of time and effort to review and verify before they can be merged. Common reasons for large commits include a time-consuming testing process, slow code review and integration, and the development of large features.

(b) Long running branches: Code developed in long-lived branches can lead to a slower and more problematic integration process. Ideally, branches should be created for individual features and merged back upon their completion. Combined with shorter and more frequent commits, this practice can mitigate certain integration issues.

(c) Merge conflicts: Merge conflicts occur when integrating code commits from several developers into the main branch. Large commits and long-running branches are common causes of merge conflicts.

(d) Broken build: In cases where the build design is complicated, broken builds pose a significant issue, as maintaining them in a functional state can be time-consuming and challenging. Moreover, without a robust build process, there is a risk of integrating issues from subsequent code changes into the main codebase.

(e) Slow integration approval: Slow code reviews and integration approval processes delay feedback to developers and lead to larger commit sizes. Therefore, a timely approval process is essential to help mitigate integration challenges.

(f) Work blockage: A slow integration approval process can lead to the accumulation of multiple integration tasks in the queue, resulting in blocked work and decreased efficiency. Broken builds are another primary contributor to work blockage. The engineering team often must allocate valuable resources to fixing these builds, resources

that could otherwise be used for more productive tasks.

- **Testing problems:** This theme groups the common testing issues that occur during the CI/CD process. Common issues within this theme are as follows:

  (a) Ambiguous test result: Ambiguous test results occur when automated tests in the CI/CD process do not clearly indicate a pass or fail. Instead, these tests generate logs that developers receive via feedback mechanisms for manual analysis. This manual process of identifying results and debugging actual failures is time-consuming and practically involves only a minority of the team.

  (b) Flaky tests: Flaky tests are unreliable test cases that fail intermittently, failing to consistently capture expected behavior or stability. Possible causes for such unreliability in testing include test environment issues, concurrency bugs, and timing issues.

  (c) Time consuming testing: Long-running test cases delay feedback to developers, contributing to challenges such as prolonged commit times, disrupted development flow, and diminished procedural discipline.

2.1.5 CI/CD in Ericsson

As one of the leading global Communication Service Providers (CSPs), Ericsson relies on CI/CD as a critical component in its goal of designing, manufacturing, and delivering robust telecommunication products, particularly for *5G* networks. CI/CD facilitates the agile

delivery of telecommunication software components by automating the software lifecycle. The CI/CD approach, which emphasizes delivering small, frequent updates to the software, reduces the risk of outages, minimizes the overall testing burden, and quickly provides customers with highly customized network services [18].

With the accelerated software release cycles typical of *5G* networks, which can range from 3 to 4 weeks, CI/CD ensures that development and deployment remain agile and responsive. By efficiently managing the technological complexity and granular changes inherent in *5G*, CI/CD enables service diversity and innovation. While it is theoretically possible to deploy a *5G* Core network without CI/CD, doing so would be significantly more costly and resource-intensive. Therefore, CI/CD not only optimizes operational efficiency but also plays a key role in unlocking new revenue opportunities by supporting the rapid and continuous life cycle management of *5G* services [19].

Importantly, the CI/CD pipeline implemented at Ericsson aligns closely with the pipeline architecture discussed earlier in Section 2.1.2. This alignment illustrates how CI/CD practices can be adapted to meet the specific demands of *5G* development, further emphasizing the relevance of the concepts presented in this thesis.

2.2 Deep Learning

2.2.1 Introduction to Deep Learning

Deep learning is a subset of the machine learning field. Unlike traditional machine learning models that require features to be extracted manually, deep learning models learn the

features and representations of raw input data automatically. Inspired by the human brain, these models consist of multiple layers of connected units, often referred to as neurons, forming what are called Neural Networks (NNs). Each neuron applies a simple but non-linear transformation to the representation at one level, thereby producing a representation at a higher and a more abstract level. These non-linear transformations taking place within multiple layers of the neural network enable deep learning methods to discover complex representations and structures from large datasets [50].

2.2.2 Perceptron

A perceptron is a mathematical model representing the simplest form of an artificial neural network, comprising a single neuron that uses a threshold **activation function**. Specifically, the perceptron produces an output signal of 1 if the weighted sum of its inputs exceeds the threshold value; otherwise, it generates an output signal of 0. In this manner, a perceptron loosely replicates the on/off behavior of a neuron in the human brain [10].

An activation function is a mathematical function that maps the weighted sum of inputs to the neuron's output value. For binary classification, the activation function of a perceptron can be formulated as shown in equation 2.1:

$$f(\mathbf{x}) = \begin{cases} 1, & \mathbf{w} \cdot \mathbf{x} + b > 0, \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

where $\mathbf{w}$ refers to the weight associated with an input, $\mathbf{x}$ represents the input feature,

and **b** represents the bias value. The weight associated with an input reflects the strength of the connection between that input value and the neuron. The bias term, which is added to the product of weights and input features, shifts the decision boundary away from the origin, thereby facilitating the modeling of more complex patterns in the data [46].

Since a perceptron consists of only one neuron and uses a simple *Heaviside* step function as its activation function, it can only model linearly separable data and cannot handle more complex patterns [62].

2.2.3 The Multi-Layered Perceptron (MLP)

The Multi-Layered Perceptron (MLP) is a type of neural network that consists of multiple layers of neurons and is capable of learning complex patterns in data for various supervised tasks, including classification and regression. A MLP includes one input layer, one or more hidden layers, and an output layer. Each hidden layer performs non-linear transformations on the data using activation functions such as **tanh**, **sigmoid**, and Rectified Linear Unit (ReLU), among others, which allow the network to capture complex relationships in the data.

A MLP is trained using an input dataset and an objective function (also known as a loss function). The objective function represents the difference between the predicted values of the network and the actual values. The training objective for a MLP, like any neural network algorithm, is to optimize the state of its parameters (weights and biases) for minimizing this loss function.

2.2.4 Neural Networks (NNs)

Since the MLP is a specific type of neural network, it can considered a subset of the broader NNs category [72]. In addition to MLP, NNs consist of several other types of architectures, such as the popular Recurrent Neural Network (RNN), Convolutional Neural Network (CNN), and Long Short-Term Memory (LSTM) architectures.

As highlighted in earlier sub-sections, a NN consists of multiple layers of neurons that map the input into output using non-linear activation functions. These non-linear transformations enable the network to discover complex patterns in data. Each neuron in the network consists of adjustable parameters called weights and biases. During the training process, these parameters are adjusted using optimization algorithms such as gradient descent and Adam to minimize the loss function, i.e. the error between the predicted values and the ground-truth [50].

2.2.5 Recurrent Neural Networks (RNNs)

RNNs are a type of neural network specialized in processing sequential data, where the components of the sequence are interconnected. Examples of such data include words, sentences, and time-series data. Like other neural networks, RNNs consist of neurons and have input, hidden, and output layers. However, unlike traditional neural networks, RNNs feature hidden layers that are recurrently connected, allowing them to maintain and update a state across time steps [29].

RNNs process input data sequentially, passing it from the input layer through the

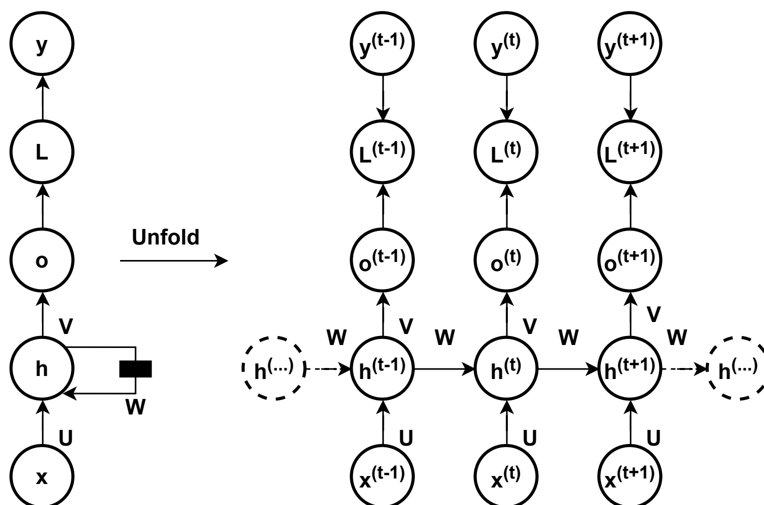

Figure 2.3: RNN architecture and its corresponding computational graph used for computing the training loss [29]

hidden layers one time step at a time. Unlike traditional feedforward networks, which process each input independently, RNNs have recurrent connections that enable them to retain and utilize information from previous time steps through a hidden state. This mechanism allows RNNs to consider prior inputs when processing and understanding the current input in sequential data [29].

Another key difference between traditional feedforward networks and RNNs is that recurrent networks are capable of processing sequences of variable lengths. This capability stems from their recurrent structure, along with parameter sharing across different time steps. This design allows RNNs to generalize across different sequence lengths and positions in time. To illustrate this difference between feedforward and recurrent networks, consider the following two sentences of equal length: (a) 'I went to Nepal in 2009,' and (b) 'In 2009, I went to Nepal.' A traditional feedforward network trained on these sentences would use

separate parameters for each word in the sentences. Since there is no parameter sharing across positions, the network would need to learn the language rules independently at each position. For instance, in sentence (a), even if the network identifies '2009' as a year based on its position, it would need to learn new parameters for the word '2009' in sentence (b) due to its different position. On the other hand, since an RNN shares the same parameters across different time steps, it does not need to relearn the rules of language at each position in the sentence. Instead, it can apply the same learned rules regardless of where the words appear in the sentence [29].

Equation 2.2 presents the equation of hidden states within a recurrent network. Variable $\mathbf{h}^t$ represents the hidden state of the network at time $\mathbf{t}$ and is defined as a function of the hidden states at previous time step $\mathbf{h}^{t-1}$, the external input $\mathbf{x}^t$ at time $\mathbf{t}$, and the shared parameters $\boldsymbol{\theta}$. In practice, a typical RNN also includes an output layer that processes information from these hidden states to produce a final result [29].

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \boldsymbol{\theta}) \tag{2.2}$$

A RNN can be designed in multiple ways. Figure 2.3 illustrates a commonly used RNN architecture, where an output is generated at each time step $\mathbf{t}$, and the hidden units contain recurrent connections. This means that information from the hidden layers at time step $\mathbf{t-1}$ is provided to the hidden layers at time step $\mathbf{t}$ along with the input $\mathbf{x}$ at time $\mathbf{t}$. Variables $\mathbf{L}$ and $\mathbf{y}$ represent the 'loss' and the 'target output values' respectively [29].

Training and associated challenges.

Training a RNN follows the typical neural network procedure, consisting of a **forward pass** and a **backward pass**. During the forward pass, data is propagated from the input layer to the output layer, with intermediate hidden layers processing the data through non-linear activation functions. In the backward pass, gradients (or derivatives) of the loss function are computed with respect to each network parameter. These gradients are then propagated backward from the output layer to the input layer, allowing the network to update its parameters to optimize the loss function, also known as the objective function [50].

However, this training procedure can be problematic in RNNs due to the tendency of propagated gradients to either grow or diminish with each time step. Over many time steps, this tendency can lead to the issue of **vanishing gradients**, where the gradients become too small to effectively update the network's parameters, or **exploding gradients**, where the gradients become excessively large, resulting in unstable parameter updates. These issues contribute to the long-term dependency problem, where the RNN struggles to learn and retain information over long sequences of data [50].

2.2.5.1 Long Short-Term Memory (LSTM) Networks

The LSTM architecture [32] is a type of RNN specifically designed to address the issue of **vanishing gradients** and to enable the network to learn long-term dependencies across multiple time steps in data. This is achieved by augmenting the standard RNN with a

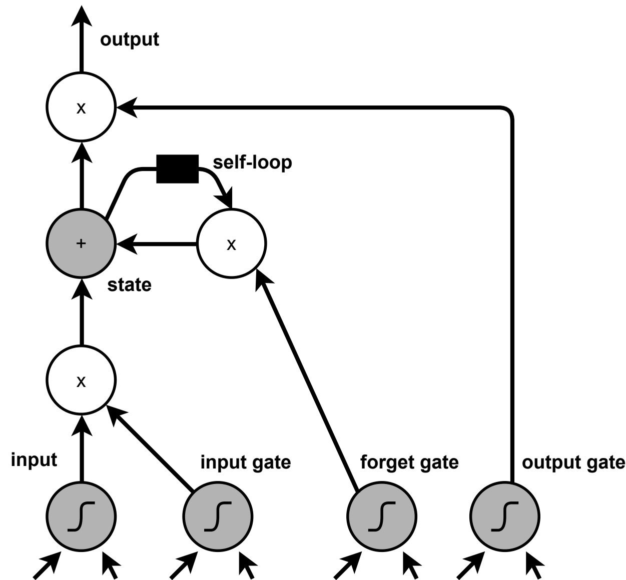

Figure 2.4: Block diagram of the LSTM recurrent network cell, as originally presented in [29]

mechanism that allows the network to retain information over extended periods. Unlike traditional units that simply apply non-linear transformations to the input data, the LSTM architecture consists of specialized "LSTM cells." These cells not only apply non-linear transformations using regular neurons but also accumulate information over time into the cell's internal state. Each LSTM cell includes three gates: the input gate, the forget gate, and the output gate, which collectively control the flow of information into and out of the cell state. The input gate utilizes the sigmoid function, which ranges from 0 to 1, to determine the extent to which new input information should be accumulated in the cell state. The forget gate regulates the self-loop of the cell state, determining how much information from the previous time step should be retained or forgotten. This gate also employs a sigmoid function, which evaluates the current input along with information from

previous time steps to decide whether the information should be retained (with a value closer to 1) or forgotten (with a value closer to 0) from the cell state. The output gate controls whether the cell's internal state should be used to influence the output of the LSTM cell at the current time step. Like the other gates, the output gate uses a sigmoid function to decide whether to allow the state to pass through to the output or to suppress it [29] [50].

Figure 2.4 illustrates a typical LSTM cell, consisting of an internal state, input gate, forget gate, and output gate. The self-loop represents the internal recurrence of the LSTM cell, while the black square indicates a delay of a single time step.

2.2.6 Transformer Model

Introduced in 2017, the Transformer architecture is a state-of-the-art model that relies solely on attention mechanisms, eliminating the need for recurrence, as seen in RNNs and CNNs. These mechanisms, combined with positional encodings, enable the architecture to process input data in parallel, which contrasts with the sequential nature of RNNs. Moreover, while standard RNNs are unidirectional, the Transformer model's self-attention mechanism allows it to capture relationships across the entire sequence simultaneously, enabling it to consider context from both directions. Although the original research paper focused on the machine translation task, the ability of Transformers to learn complex patterns in data through self-attention and to process this data efficiently in parallel has allowed for the training of large-scale language models. These models have since been applied successfully to a variety of NLP tasks, including question answering, text summarization,

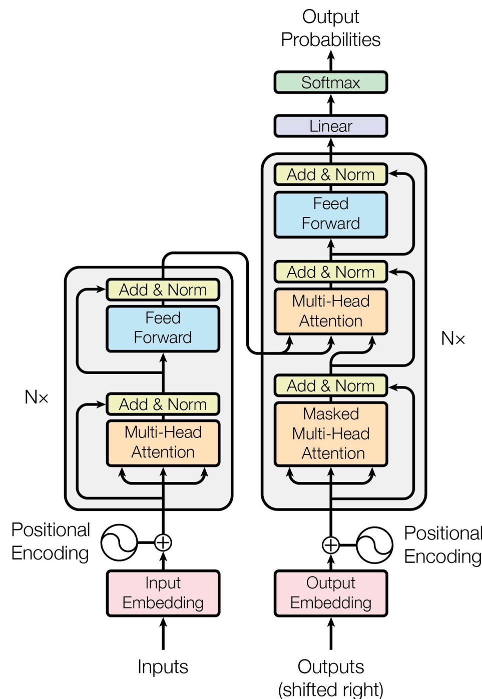

Figure 2.5: Architecture of the Transformer model, as originally presented in [82]

text generation, classification, and machine translation [82].

2.2.6.1 Model Architecture

Figure 2.5 presents the architecture of the Transformer model. Similar to other competitive transduction models, the Transformer model consists of an encoder-decoder architecture. The encoder layer takes an input sequence ($\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$, ..., $\mathbf{x}_n$) and generates representations for it. The decoder layer takes the generated representations and produces an output sequence ($\mathbf{y}_1$, $\mathbf{y}_2$, $\mathbf{y}_3$, ..., $\mathbf{y}_n$) one step at a time. The decoder component in the Transformer model is auto-regressive in nature, meaning that at each time step, it takes the previously generated symbols into consideration for producing the next output. We further describe the encoder-decoder stacks of the model in the subsequent section. However, before explaining this structure, we provide information on the important scaled dot-product

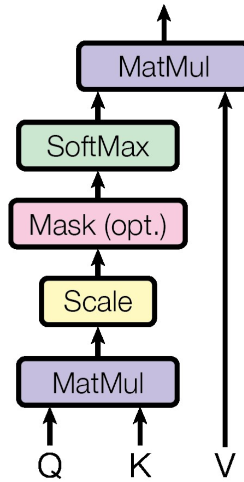

Figure 2.6: Scaled Dot-Product Attention, as originally presented in [82]

attention and multi-head attention mechanisms [82].

2.2.6.2 Scaled Dot-Product Attention

Attention mechanisms facilitate the modeling of dependencies within a sequence without regard to the distance between components in the input or output sequences, enabling the model to capture both short-term and long-term dependencies. This means that the attention mechanism allows the model to relate different components in the sequence to one another, enhancing the understanding of context and improving the learned representation [82].

The Transformer model employs a specific type of attention mechanism known as "scaled dot-product attention." This mechanism calculates attention weights by mapping

a query vector and a set of key vectors. The query vector ($\mathbf{Q}$) represents the component currently in focus within the sequence and determines how much attention should be allocated to each component. The key vectors ($\mathbf{K}$) represent all components of the sequence and assess the strength of the relationships between the current component and the others. The value vectors ($\mathbf{V}$) contain the actual data that the model will attend to, with the final output derived by computing a weighted sum of these value vectors based on the calculated attention weights. The query, key, and value vectors are computed using their respective weight matrices $\mathbf{W}^Q$, $\mathbf{W}^K$, $\mathbf{W}^V$, along with the input matrix $\mathbf{X}$. These weight matrices are initialized randomly and are optimized during the training phase. The accuracy of these weight matrices directly impact the model's ability to learn accurate representations from the data [82] [68].

The resultant attention output is computed as a weighted sum of the value vectors, with the weights determined by the relevance of each key vector to the query, as computed by the attention mechanism. The specific steps involved in generating the attention output are as follows:

- Compute the dot product of all the queries $\mathbf{Q}$ and keys $\mathbf{K}$ of dimension $\mathbf{d}_k$ to get similarity scores.
- Divide this dot product $\mathbf{QK}^T$ by the dimension of the key vector $\sqrt{\mathbf{d}_k}$. This scaling factor helps in mitigating unstable gradients that are caused by large dot product values.
- Apply the **softmax** function to normalize the scores and obtain the score matrix.

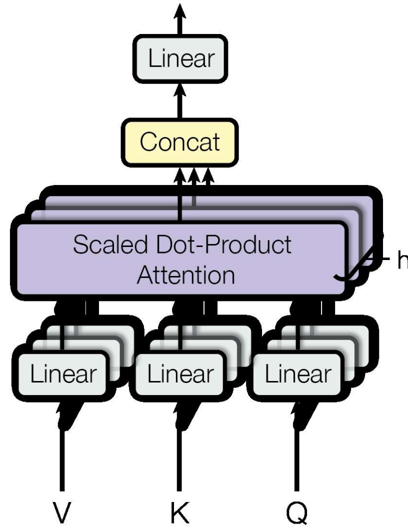

Figure 2.7: Multi-Head Attention which consists of several attention layers running in parallel, as originally presented in [82].

- Compute the attention matrix for each component by multiplying the score matrix with the value matrix.

Equation 2.3 and Figure 2.6 present the mathematical and pictorial representation of the scaled dot-product attention respectively [82].

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.3}$$

2.2.6.3 Multi-Head Attention

In scaled-dot product attention (Section 2.2.6.2), a single set of queries (**Q**), keys (**K**), and values (**V**), each represented as vectors with dimensionality $\mathbf{d}_{model}$, is used to compute

a single attention matrix. However, Vaswani et al. [82] extend this approach by linearly projecting the queries, keys, and values h times into dimensions $\mathbf{d}_q$, $\mathbf{d}_k$, and $\mathbf{d}_v$ using different learned linear projections. Each of these projected versions is then processed in parallel through the attention function, yielding $\mathbf{d}_v$-dimensional output values. These output values are subsequently concatenated and projected once more to obtain the final attention values. Figure 2.7 illustrates this process. The vectors $\mathbf{V}$, $\mathbf{K}$, and $\mathbf{Q}$ are linearly projected h times and then processed in parallel by h scaled dot-product attention layers.

Multi-head attention enables the model to simultaneously attend to information from different representation subspaces at different positions, resulting in more accurate learning [82].

2.2.6.4 Encoder and Decoder Stacks

The Transformer model is composed of a stack of encoder and decoder layers (see Figure 2.5). The output of each encoder/decoder layer is utilized as input for the subsequent encoder/decoder layer [68].

The encoder stack receives source sentences as input and generates their representations as output. Prior to feeding the sentences into the encoder, they are converted into embeddings and supplemented with positional encoding. This positional encoding enhances the input data by incorporating information about the sequence order, thus facilitating parallel processing. Each encoder layer comprises a multi-head attention sub-layer, followed by a fully connected feedforward network sub-layer, which consists of two linear transformations with ReLU activation in between [82].

Similar to the encoder stack, the decoder stack is composed of several decoder units, where each decoder's output serves as input to the subsequent decoder layer. In addition to the sub-layers found in each encoder layer, the decoder incorporates a multi-head attention sub-layer that operates over the output of the encoder stack. Moreover, the self-attention sub-layer in the decoder is modified to ensure that predictions at position i are based solely on outputs from positions earlier than i [82].

2.3 Natural Language Processing (NLP)

2.3.1 Introduction to NLP

NLP is a subset of AI that focuses on enabling computers to understand and process human language. NLP encompasses a range of applications in day-to-day life, including automatic machine translation on the web, text classification for spam detection, text summarization, text generation, and dialogue systems such as chatbots. Traditionally, NLP approaches were predominantly rule-based or statistical. However, the advent of neural-based models, such as RNNs (see Section 2.2.5) and Transformer architectures (see Section 2.2.6), has significantly advanced language modeling capabilities. The Transformer model, in particular, has become widely utilized in chatbot systems to effectively assist users with their queries.

2.3.2 Word Embeddings

AI models cannot directly process raw text. To enable these models to understand and work with word meanings, words must be represented in a machine-interpretable format. In NLP, **vector semantics** is the standard method for this purpose, allowing us to capture various aspects of word meaning, such as lemmas, senses, synonymy, and similarity. Vector semantics defines a word's meaning based on its distribution in a language, modeling it according to its typical neighboring words and grammatical contexts. For example, from the following sentences, we can infer that *ongchoi* refers to a leafy green solely from its contextual usage and the similarity to other words in similar contexts [44].

Example sentences (A):

- Ongchoi is delicious sauteed with garlic
- Ongchoi is superb over rice
- ...ongchoi leaves with salty sauces...

Example sentences (B):

- ...spinach sauteed with garlic over rice...
- ...chard stems and leaves are delicious...
- ...collard greens and other salty leafy greens...

In vector semantics, words are represented as vectors in a multidimensional space derived from the distributions of neighboring contexts. These vectors, known as **embeddings**, can be classified into two types: static and dynamic. **Static embeddings** assign a

fixed vector to each word in the vocabulary, as seen in models like **word2vec**. In contrast, **dynamic embeddings** generate context-specific vectors for each word, with Bidirectional Encoder Representations from Transformers (BERT) being a prominent example of this approach. Since embeddings effectively measure semantic similarity between words or sentences, they are widely used in Information Retrieval (IR) tasks to identify the document **d** from a set of documents **D** that best matches a query **q** [44].

2.3.3 Language Models

Developing artificial intelligence that can understand and process language has historically been a challenging task due to the complex nature of languages and their varied rules. Language modeling is a key approach that has proven effective in developing such AI models. This approach focuses on modeling the generative likelihood of word sequences and predicting the likelihood of future tokens. Language models achieve this by analyzing the context and determining the next token based on the probabilities governed by the rules of the language [20].

Zhao et al. [20] further categorize language models into the following four main types based on their underlying technology.

- **Statistical Language Models:** These models are based on *statistical learning* methods and rely on the Markov assumption to build prediction models. Examples of such language models include n-gram models, such as bigram and trigram models.

- **Neural Language Models:** These models generate the probabilities of word sequences using NNs (see Section 2.2.4), such as MLP (Section 2.2.3) and RNN (Section 2.2.5). Bengio et al. [8] introduced the important concept of *distributed representation* of words and developed word prediction using aggregated contextual features, which served as the backbone for future models.

- **Pre-trained Language Models:** These models aim to learn the underlying representation of language during a pre-training stage, where they are trained on large corpora. They are then fine-tuned on downstream tasks to adapt their behavior or impart specific knowledge. An early attempt at modeling context-aware representations was made using a bidirectional LSTM (see Section 2.2.5.1). The introduction of the highly parallelizable Transformer model (Section 2.2.6) led to the development of powerful language models such as BERT [15].

- **Large Language Models:** Scaling pre-trained language models by increasing the number of their parameters has led to significant improvements in performance. The term LLMs refers to this group of pre-trained language models that have been scaled to include a large number of parameters. In our study, we utilize the Llama2-Chat 7 billion parameters model [80], which is a LLM that has been optimized for dialogue use cases.

2.4 Retrieval-Augmented Question-Answering

Our chatbot falls under the umbrella of retrieval-augmented Question-Answering (QA) techniques. Retrieval-augmented QA involves integrating a retrieval mechanism to extract pertinent information from a given set of documents, thereby enhancing the accuracy and completeness of answers to queries. Retrieval augmentation has been explored for both extractive QA [22] and generative QA [55, 74], with significant accuracy improvements shown for both types of QA. For generative QA, which is the focus of our work, the associated prompt engineering is one of the most critical steps. OpenAI provides several general guidelines on how to build effective prompts [66] for RAG tasks. We follow these guidelines for building retrieval augmentation into our chatbot.

A common approach for implementing retrieval augmentation is through a *retriever and reader architecture* [22]. The retriever is responsible for efficiently selecting a subset of relevant documents from a larger corpus, acting as an initial filter to narrow the search space. Subsequently, the reader – typically an LLM – is tasked with comprehending and extracting/generating information based on the retrieved documents. The retriever component can be configured to supply the reader with the latest documents pertinent to the user's query, addressing the challenge posed by the LLMs' knowledge cut-off. Moreover, providing pertinent context helps reduce LLMs' tendency to hallucinate [76].

Enhancement Strategies for RAG.

Several enhancements can be considered to further increase the accuracy of RAG. We explored two such enhancements: end-to-end training and query rewriting. Below, we

outline these enhancements and explain our rationale for their inclusion or exclusion.

End-to-end training involves jointly training the retriever and the reader on domain-specific data [40,52]. Previous attempts at end-to-end training have employed LLM readers such as BERT [15] and BART [51]. However, implementing this approach with readers like Llama 2 [80] and Generative Pre-trained Transformer (GPT)-3 [9] remains prohibitively expensive. Since our chatbot is based on Llama 2, we do not pursue end-to-end training. Furthermore, we note that while end-to-end training has been shown to lead to improvements in studies with BERT and BART, these improvements are comparatively modest [52]. This suggests that end-to-end training would be worthwhile only if its cost is sufficiently low, which is currently not the case for the newer generation of LLMs.

A second possible enhancement to consider is query rewriting. In a RAG pipeline, the retriever step fetches documents similar to the user query. A well-written and self-contained query is thus critical for this step. Nonetheless, real-world user queries are not always optimal and may require adjustments to improve the retriever's accuracy. Ma et al. [59] demonstrate the usefulness of adding a query rewriter to RAG. Motivated by their results, we adapt and extend their guidelines to integrate a query rewriter into our chatbot. Although query rewriting inevitably increases QA execution time, our overall pipeline's execution time remains acceptable, as we show in Section 4.

2.5 Related Work

In software engineering, chatbots and LLMs are increasingly used to assist with tasks such as code understanding, code generation, and quality assurance [1]. This section reviews recent relevant strands of work on chatbot-enabled software engineering and contrasts them with our research. Following this, we examine recent applications of LLMs designed to assist with various software engineering tasks

2.5.1 Chatbots in Software Engineering

A prominent example of a widely used conversational software development tool is GitHub Copilot [27]. Copilot builds on top of the Codex model [65], a fine-tuned version of GPT-3 [9], to provide support in various tasks such as code completion, code generation from natural-language input, code migration, and answering coding questions. Within Copilot, the task most similar to our chatbot function is answering coding questions. Nevertheless, Copilot's QA capabilities are focused on code-related queries [28]. In contrast, our chatbot does not have a code-centric focus and is complementary; its primary goal is to assist systems engineers with domain-specific technical questions concerning integration, testing, deployment, and troubleshooting.

QAssist [22] employs a RAG architecture similar to ours to help stakeholders analyze and improve the quality of software requirements specifications. Specifically, this tool uses requirements-relevant content alongside generic domain material sourced from Wikipedia to answer questions about natural-language requirements. While QAssist uses RAG and

shares a broad-spectrum QA objective similar to ours, it implements extractive QA using BERT variants. As such, QAssist can only highlight passages containing the answer, without the ability to engage with users in a conversational and context-aware manner. In contrast, our chatbot uses Llama 2, a generative LLM, to produce coherent and context-aware answers. These answers are based on knowledge extracted from relevant passages and chat history. Furthermore, in terms of design, our chatbot has a more advanced technology chain than QAssist, aligning with the latest advances in chatbot development.

A recent study by Abedu et al. [3] employs a RAG-based chatbot to facilitate user access to information within software repositories. This approach allows users to input the URL of the target repository in the chatbot interface and then query the repository's content. While the ability to dynamically input the desired repository provides flexibility, it also complicates the implementation of tailored preprocessing to enhance retrieval performance. In our chatbot design, we separate the domain-corpus creation process from the chatbot pipeline. This separation enables us to preprocess documents based on their content type. Furthermore, while Abedu et al. only experiment with a fixed choice of retriever (embeddings-based), we conduct a comparative analysis of four different types of retrievers to identify the most suitable one for our problem context, as we discuss in Section 4.7.

In summary, while chatbots have been deployed to support various software engineering tasks, none are specifically designed for the analytical goals that our chatbot addresses, nor do they have the exact same design considerations as ours.

2.5.2 LLMs in Software Engineering

In addition to their use in chatbots, LLMs are widely applied across various software engineering domains, including requirements engineering, software design, software maintenance, and software development.

One specific application is automatic generation of code comments, as explored by Geng et al. [24]. Using the Codex model [65], they aim to generate accurate code comments to support software development. Their multi-shot learning approach, combined with re-ranking strategies, generates code comments tailored to different developer intents, enhancing relevance through the careful selection of examples. Their method leverages the fact that these LLMs are pre-trained on code-comment pairs, thus establishing a strong connection between code and natural language. With real-world developer comments encompassing multiple intents, this study achieves state-of-the-art results through effective multi-shot prompting.

Another approach, CodeEditor by Li et al. [53], focuses on code editing tasks. This study presents a model pre-trained on GitHub code snippets and their modified versions, evaluated across three configurations: fine-tuning, few-shot prompting, and zero-shot prompting. Empirical results indicate that CodeEditor outperforms current state-of-the-art baselines, demonstrating its effectiveness in supporting code modification tasks.

Beyond code editing and comments generation, recent studies also utilize LLMs for maintenance and quality assurance tasks. For instance, the CrashTranslator approach by Huang et al. [33] automates the process of reproducing mobile crashes directly from stack traces. Specifically, this approach leverages a LLM to backtrack from the stack trace and

predict the series of events leading to a mobile application crash.

Additionally, Ma et al. introduce KnowLog [58], a pre-trained LLM aimed at improving log understanding. KnowLog combines a contrastive pre-training task with domain-specific information to generate universal representations, achieving state-of-the-art results in log analysis.

In contrast to these studies, our work targets CI/CD-specific question-answering, leveraging a RAG architecture to provide precise, contextually relevant responses.

Chapter 3

Approach

In this chapter, we describe the approach implemented by our chatbot. The chatbot takes as input a collection of documents – in the context of our industry collaboration, a collection of Ericsson documents related to CI/CD – alongside a natural-language user query. Subsequently, the chatbot uses an LLM to generate a natural-language response to the query. Section 3.1 describes the process of creating a domain-specific corpus from Ericsson's CI/CD documents. Section 3.2 provides an overview of our chatbot's architecture and discusses the steps involved in generating an answer based on the domain-specific corpus and a given query.

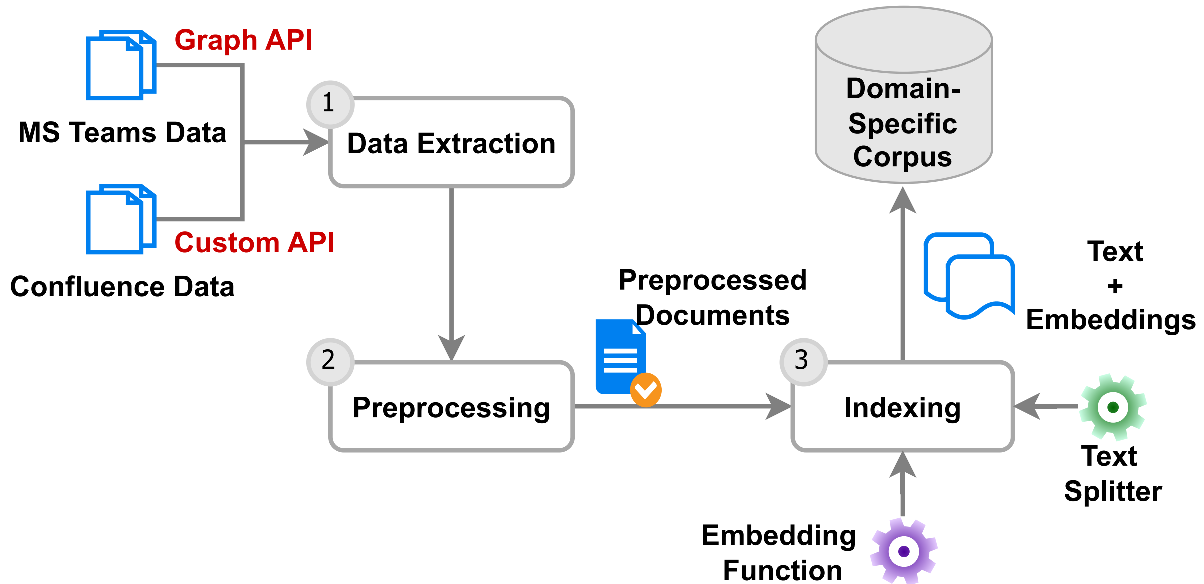

Figure 3.1: Steps for Creating a (Domain-specific) CI/CD Corpus

3.1 Corpus Creation

Figure 3.1 presents the steps we follow for transforming Ericsson's CI/CD documents into a (domain-specific) corpus.

Step 1: Data Extraction. We gather CI/CD domain knowledge from two Ericsson-specific sources: (1) messages exchanged among software engineers in CI/CD channels on internal Microsoft Teams, and (2) Confluence [5] web pages that contain information about common troubleshooting procedures for CI/CD-related tasks within the organization. These data sources enable our chatbot to respond to Ericsson-specific technical questions. Noting that both data sources evolve frequently over time, we first extract the most recent data from these sources. This extraction step is decoupled from the chatbot itself (Section 3.2), allowing updates to be performed offline and on a regular basis (e.g.,

overnight) without disrupting the chatbot's function.

The Teams data, comprised of messages and replies, is collected through the Microsoft Graph Representational State Transfer (REST) Application Programming Interface (API) [61]. Using this API, we generate separate tables (in Comma-separated values (CSV) format) for storing messages and replies, with one table dedicated to messages and another to replies. The messages table has 35 parameters (fixed columns) such as message ID, creation time, content, sender's username, mentions, and reactions. The replies table consists of 43 parameters such as reply ID, reply's parent message ID, content, mentions, and reactions. A complete list of parameters for Microsoft Teams messages and replies is provided in Appendices A.1 and A.2, respectively.

The Confluence data consists of a set of Hypertext Markup Language (HTML) pages, with each page containing a troubleshooting topic followed by content outlining the troubleshooting procedure. We extract these pages using a custom REST API developed by Ericsson.

Step 2: Preprocessing. We process the data extracted in Step 1 into a suitable format for use by the LLM.

For the Teams data, we strip the HTML formatting from the 'content' column of the messages and replies tables discussed earlier and convert the content to plain text. Subsequently, each reply in the replies table is linked to its corresponding parent message in the messages table through the unique parent message ID. This process connects each message to its replies in the same order as they were originally posted, thus preserving the chronological order of messages. To protect privacy, as we reconcile the messages and

the responses to them, we remove personal information such as employee names and email addresses from both the content and the associated metadata. Finally, we store every message along with all the responses to it in a plain-text document.

As for the extracted Confluence pages, the preprocessing is straightforward: we store the title and content of each page in a plain-text document.

To facilitate more accurate interpretation of the Teams and Confluence data by the LLM, we augment this data with prefixes. Specifically, each Teams message is prefixed with the phrase "Message:", while replies to the message are prefixed with "This message has the following responses:". Similarly, the title of each Confluence page is prefixed with "Page Title:", and the content following the title is prefixed with "The content of this page is as follows:".

Step 3: Indexing. In this step, we embed and store the preprocessed documents obtained from Step 2 (Figure 3.1) in a vector database. This vector database serves as the domain-specific corpus for the information retrieval step of our chatbot (Step 2 in Figure 3.2, discussed later). Since the preprocessed documents are ultimately supplied to the LLM as relevant context, our objective is to maximize their length. However, two important considerations arise when determining the ideal length. On the one hand, we must ensure that the documents fit within the context length (token limit) accepted by the LLM, as exceeding this limit could result in context loss, runtime errors, or incoherent output. On the other hand, supplying long documents to the LLM can lead to a "needle in the haystack" problem [31], where relevant information is lost amongst the noise. Therefore, determining the optimal length of embedded documents becomes an important factor in maximizing

chatbot efficacy, as we aim to maximize the amount of relevant information while limiting the noise.

Our choice of how to split the data in a preprocessed document depends on the source from which the document originates. We have custom splitters for each Teams and Confluence. For Teams, based on actual data and the experience of collaborating engineers at Ericsson, the combination of a message and all its replies is anticipated to always be well below the contextual length of modern LLMs. Therefore, we have determined that each individual message, alongside all the replies to it, could be embedded directly as one unit in our vector database. This means that the units fetched by the retriever step of the chatbot will constitute one message and all the associated replies.

For the Confluence data, the length of a preprocessed document could exceed the context length of the LLM. Therefore, we need to apply document chunking before embedding large Confluence documents in the vector database. To determine the optimal chunk size for Confluence data, we conducted exploratory experimentation with values ranging from 200 to 1000 tokens (increasing the chunk size by 100 tokens in each iteration). Based on this experimentation, we observed that a chunk size of 800 tokens led to the best question-answering results. Furthermore, we followed the best practice of making adjacent chunks overlapping [22], maintaining an overlap of 200 tokens (25%) between adjacent chunks. To further preserve context, all chunks belonging to the same Confluence document were prefixed with the document title and the respective chunk number.

Following the splitting of the Teams and Confluence data, we send the resulting chunks to an embedding function to generate text embeddings. To ensure consistent terminology,

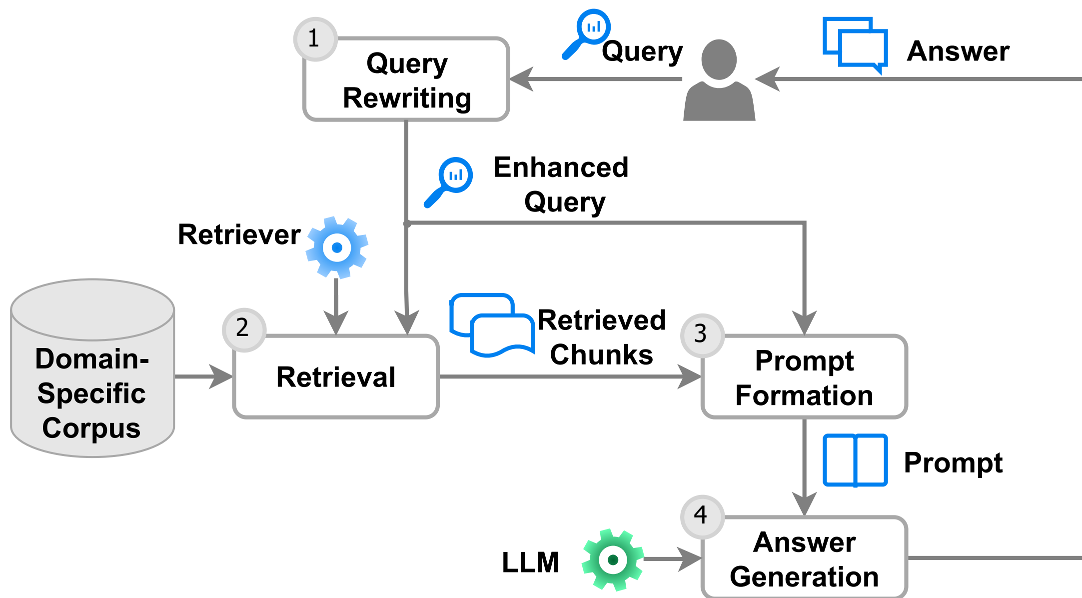

Figure 3.2: Overview of Our Chatbot Design

we refer to these chunks as *context items* rather than "documents", to avoid ambiguity between the chunks and the original (Confluence) documents. We store the embeddings for each context item in the vector database alongside the item's original text. The domain-specific corpus depicted in Figures 3.1 and 3.2 is realized by this vector database.

3.2 Chatbot Design

Figure 3.2 shows the steps implemented in our chatbot. We will discuss each of these steps below.

Step 1: Query Rewriting. Given that user queries may not always be clear or well-

structured [59], our query rewriting module prompts an LLM [1] to enhance the query into a more effective search query. To achieve this, we use the query enhancement guidelines of Ma et al.'s [59]. Specifically, given a (frozen) LLM and a user query, we prompt the LLM to restate the query in more precise terms before the query is used for question answering. We augment Ma et al.'s prompt template for query rewriting by instructing the model to also analyze the user query in the context of the conversation history and determine if the current query is a follow-up. In response, the LLM either forms an improved question or returns the user query verbatim as the question to pose to the LLM.

In addition to attempting to construct a better query based on Ma et el.'s prompt template, we incorporate various prompt engineering techniques to improve the quality of the LLM's output. Figure 3.3 presents our query-rewriting prompt and highlights the various considerations involved.

We employ the same prompt template that Meta utilized for pre-training Llama 2 [80]. This helps ensure that the model is able to understand the prompt structure and instructions clearly. In the prompt template, the instructions provided between the `<<SYS>>` tokens convey a system message to the model, instructing it on its task and intended behaviour. To improve results, we explicitly define the task of the LLM (Figure 3.3, ❶). We then employ zero-shot chain-of-thought prompting [83] (❷) to decompose the task into intermediate steps. We mitigate ambiguity in the model's understanding of the task by explicitly handling possible scenarios (❸). Further, we instruct the model to retain important key terms from the query (❹) and to strictly follow the desired output format (❺).

[1] Note that the LLM used for query rewriting does not necessarily have to be the same one used for answer generation in Step 4 of Figure 3.2, which will be discussed later. In our current approach, nevertheless, we use Llama 2 as the LLM of choice to implement both Steps 1 and 4 of our chatbot design.

We illustrate Step 1 using the two examples, Example 1 and Example 2, in Figure 3.4. Example 1 demonstrates the ability of query rewriting to filter irrelevant information from the user query and generate a better query that focuses on the important aspects. Example 2 highlights how query rewriting can identify the question implied by the user statement. Moreover, in the case of the follow-up question in Example 2, query rewriting uses the conversation history to derive a complete query.

Step 2: Retrieval. Following the retrieval-augmented generation approach [52], we employ a retriever component to identify domain-specific knowledge that should be imparted to the LLM for answer generation. In our design, the retriever uses the enhanced query obtained from query rewriting (Step 1) to retrieve relevant context items from the domain-specific corpus, constructed using the process described in Section 3.1.

Recognizing that different data sources may necessitate different retrievers for better question-answering accuracy, our approach offers the flexibility to specify the retriever to be used. The selected retriever is responsible for picking the top-k context items to feed to the LLM as relevant information for answer generation. Determining the optimal value of k is crucial and should consider the nature and size of the context items. Our exploratory experimentation revealed that setting $k = 3$ yields the best results in our application context. This choice also happens to be consistent with the k value recommended by Ezzini et al. [22] based on systematic experimentation. However, we observe that forcing k context items to be considered at all times has the potential to introduce noise in cases where the number of relevant items in the corpus is less than k. To filter irrelevant information in the top-k context items, we consider a context item only if it has a higher cosine similarity value

```
<s>[INST] <<SYS>>                                    1

You will be provided with a user question. Your task is to rewrite this question. Approach this
task step-by-step, take your time and do not skip steps: 2

1. Analyze the current user question in the context of the provided conversation history.
2. Determine if the current user question is a follow-up based on the conversation history.
3. If the current user question is a clear follow-up, rewrite it as a standalone question. If the
conversation history is not clearly relevant or you are unable to rewrite the user question,
return the user question itself as the standalone question.   3
4. Ensure that the standalone question from Step 3 is clear, precise, and retains the original
meaning and key terms (like URLs, proper nouns, document titles, etc). 4
5. Your output must be a JSON object only. Do not include any explanation or any other text.
You must strictly follow this JSON schema: 5
{{
  "standalone_question": "*Your Step 3 result will go here*",
}}

<</SYS>>


<conversation_history>:
{chat_history}
</conversation_history>


<user_question>:
{question}
</user_question>

[/INST]
```

Figure 3.3: Prompt Template for Query Rewriting

to the query than a configurable threshold. This threshold helps prevent the selection of irrelevant context items merely to meet the specified quota of k items. We set this threshold to 0.7, based on exploratory experimentation and informed by our experience in setting

Query Rewriting Example 1

User Query:

I have been trying to load [tool] reports since morning but am unable to do so due to an error. How can I solve this?

Query Rewriting Output:

```
{
  "standalone_question": "How can I solve an error when loading [tool] reports?"
}
```

Query Rewriting Example 2

User Query:

I want to add design rules check for container images

Query Rewriting Output:

```
{
    "standalone_question": "How can I add design rules check for container images?"
}
```

{# Follow-up question by user #}

User Query:

Are there any pre-requisites?

Query Rewriting Output:

```
{
    "standalone_question": "What are the pre-requisites for adding design rules check for
    container images?
}
```

Figure 3.4: Examples of Query Rewriting

similar thresholds in our previous work [57].

Finally, we employ reordering and contextual compression techniques to further enhance

Question: What are the prerequisites for creating a microservice release pipeline?

Fetched Item without Contextual Compression:

Chunk number: 1

Title of document: How to create a Microservice Release pipeline
The content of this document is as follows:
General information: Information about the Microservice release pipeline is available here: technical description, sequence diagrams, etc- description of the Microservice template, including the Release pipeline. Please note, the validation step is added to the release pipeline as a gating function in the example now.

How to add a Release pipeline to your Microservice - Important! Pre-requisites: The following pre-requisites are needed before the Release pipeline can be added: Follow the instruction on How to add Product numbers to a Microservice. Step the Microservice version to at least [version number]. The version [version number] of the products in the Microservice needs to be manually registered in [tool] as an initial version. [Redacted part - Ericsson confidential information]. Add Release files: There are 2 files needed to implement the Release pipeline, and both are available in the Microservice template - [Redacted part - Ericsson confidential information]. Clone the template repo and copy the files into your Microservice root directory. [Redacted part - Ericsson confidential information]

Fetched Item with Contextual Compression:

Chunk number: 1

Title of document: How to create a Microservice Release pipeline
How to add a Release pipeline to your Microservice - Important! Pre-requisites: The following pre-requisites are needed before the Release pipeline can be added: Follow the instruction on How to add Product numbers to a Microservice. Step the Microservice version to at least [version number]. The version [version number] of the products in the Microservice needs to be manually registered in [tool] as an initial version. [Redacted part - Ericsson confidential information]. Add Release files: There are 2 files needed to implement the Release pipeline, and both are available in the Microservice template - [Redacted part - Ericsson confidential information].

Figure 3.5: Example of Contextual Compression

the quality of the retrieved context items. The reordering technique involves placing the most relevant context items at either the beginning or the end when feeding the material to the LLM. This approach was inspired by recent research, which indicates that LLMs utilize context most effectively when it is located at the beginning or end, with a decline in performance when the relevant context is situated in the middle of long contexts [56]. Moreover, since the information relevant to the user query might be buried within the fetched items, we attempt to compress the items using the query before feeding the items to the LLM. This process, known as contextual compression [47], helps reduce the amount of irrelevant information. Similar to the threshold discussed above, contextual compression is a noise-reduction measure; however, whereas the threshold filters out irrelevant items, contextual compression mitigates the noise present within the selected items. Figure 3.5 presents an example of contextual compression, illustrating that even though the retrieved item remains the same in both cases, contextual compression significantly reduces the noise present within the item with respect to the question.

Step 3: Prompt Formation. This step takes as input the enhanced query generated from Step 1 and the relevant items retrieved in Step 2, and formulates a question prompt for the LLM. Figure 3.6 presents our prompt template. In this template, we clearly define the environment in which the model is being deployed, as well as its responsibilities (Figure 3.6, ❶). This guides the model to refrain from answering unethical questions. Furthermore, the prompt includes a safeguard to mitigate model hallucinations. Specifically, we instruct the model to answer the query solely based on the retrieved documents (❷). In cases where the model is unable to provide an accurate answer, it is instructed to inform the user and either ask for more clarity (if the query is still unclear) or request more information (if

```
<s>[INST] <<SYS>>

You are a technical chatbot deployed in a telecom organization - Ericsson. Your role is to
assist developers with their technical queries based solely on the given context.

Do not rely on external knowledge sources. Ensure that your answer is complete, accurate,
and precise as per the given context documents.

If a user's question is unclear or factually incoherent, provide a clear explanation instead of
attempting to answer incorrectly.

<</SYS>>

**Context:**
{context}

**User Question:**
{question}

**Answer: [/INST]**
```

Figure 3.6: Prompt Template for Answer Generation

sufficient relevant context has not been retrieved). The subsequent portion of the template contains placeholders for the context items and the user question, which are populated at runtime. After these placeholders are populated, the final prompt is passed to the LLM.

Step 4: Answer Generation. In this step, we use the LLM, which, in our current implementation, is Llama 2, to generate a response to the prompt obtained in Step 3. Figure 3.7 illustrates an example response by the chatbot to the following query: "List the prerequisites for adding a release pipeline to microservice", which is a variant of the query shown earlier in Figure 3.5. The chatbot takes this query as input, retrieves relevant context items, and generates an answer based on them.

Hello! I am a CI/CD chatbot at Ericsson. How can I help you?

List the prerequisites for adding a release pipeline to microservice.

To add a release pipeline to a microservice, several prerequisites must be met first. These include:

1. The microservice version must be updated to at least version # redacted
2. The version version name redacted of the products in the microservice must be manually registered in tool name redacted as an initial version.
3. Follow the instructions on how to add product numbers to a microservice.

Figure 3.7: Example of a User Query and Response Interaction

Chapter 4

Empirical Evaluation

This chapter provides a detailed overview of the evaluation procedure and the results obtained. Structured into nine sections, it systematically addresses the research questions, outlines the implementation and experimental setup, and discusses the metrics and evaluation procedures. Additionally, the chapter presents the findings and offers an analysis of the evaluation results.

4.1 Research Questions

Our evaluation aims to answer the following two research questions (RQs):

RQ1. *How accurate is our chatbot?* RQ1 assesses the accuracy of our chatbot using a combination of automated metrics and a manual analysis of correctness. This manual analysis is followed by a root-cause analysis, which identifies the underlying reasons for the

inaccuracies in the chatbot's responses as observed in our case study.

RQ2. *What is our chatbot's response time?* RQ2 measures the execution time of different chatbot-pipeline instantiations.

4.2 Implementation

We implement our chatbot using Python (version 3.10) along with supporting libraries. Specifically, we utilize the Transformers library (version 4.31.0) [37] for loading the model and the language tokenizer. To reduce computational requirements, we load the model in a 4-bit quantized format using the BitsAndBytes library (version 0.41.0) [35]. HuggingFace serves as the model repository and provides a wrapper for the text-generation pipeline over the model.

For our experiments, we use the Llama2-chat 7B parameter model [80] released by Meta. Llama2-chat is an open-source model, and by hosting it locally, we ensure that Ericsson's confidential data remains secure. Furthermore, this model has been optimized for and has demonstrated strong performance in conversational applications [80]. Finally, the model is compliant with Ericsson's internal LLM usage policies.

We employ the BAAI/bge-base-en embedding model [34] for indexing documents as it has shown good performance for the retrieval task and is computationally inexpensive [36]. LangChain (0.0.240) [48] acts as the primary library, providing support and wrappers for the RAG functions. These include: (a) a wrapper over the Chroma embedding vector-store [13], (b) support for implementing the chat history window, (c) a wrapper over the

retriever, (d) support for various optimizations of the retrieval process, and (e) a wrapper for the question-answer pipeline. We employ the Ragas library (version 0.0.21) [21] for our evaluation process.

4.3 Experimental Setup

We deployed our chatbot and performed our experiments on a Kubernetes pod containing an Intel Xeon Gold 6230N CPU with 40 GB of RAM and an Nvidia Tesla T4 GPU with 16 GB of GDDR6 memory. The CUDA version was 12.4, and the OS used was Ubuntu 20.04. Model serving was performed using the RayServe software library [69].

4.4 Ground Truth

The ground truth for our evaluation consists of 72 representative question-answer pairs gleaned over a span of nearly a year from resources and internal communications among members of the CloudRAN team at Ericsson. For each question-answer pair, we recorded the basis (documents and/or messages) upon which the correct answer was articulated. To ensure accuracy and quality, the ground truth was validated by subject-matter experts at Ericsson.

4.5 Domain Corpus

To build the domain corpus used as input for the retrieval step of our chatbot (Step 2 in Figure 3.2), we followed the corpus creation process described in Section 3.1. The corpus, which forms the basis for our evaluation in this section, was generated in May 2024. It includes 4,169 Microsoft Teams messages, 18,389 responses to these messages, and 240 Confluence web pages. These CI/CD resources collectively resulted in a total of 4985 context items, which were then stored in our vector database along with their embeddings.

4.6 Metrics

4.6.1 Metrics for RQ1

To address RQ1, we evaluate the performance of the chatbot's retrieval component through context recall and assess the overall chatbot pipeline using answer similarity, as explained below.

Context Recall@k measures whether the correct answer to a user question is present within the top k context items fetched by a retriever. In other words, given a question and k retrieved context items, this metric checks if one of these items contains all or part of the answer to the question.

Answer Similarity evaluates the semantic resemblance between the generated answer and the ground-truth answer. The generated and ground-truth answers are first embedded using an embedding function to create vectors. We employ the same embedding function

that was used in Section 3.1 (Step 3) to create the document corpus. The semantic association between these vectors is then determined using cosine similarity. The metric value ranges between 0 and 1, with a higher value indicating greater resemblance and, thus, better accuracy.

In addition to the above automatically computed metrics, we manually evaluate a single iteration of all ground-truth questions answered by our best-performing pipeline (as per Recall@k and answer similarity), classifying the chatbot answers as correct, partially correct, or incorrect:

Correct. A (generated) answer is correct if it is semantically equivalent to the ground truth. An answer is deemed equivalent to the ground truth if (a) it does not omit any information present in the ground truth, *and* (b) it does not include any information absent from the ground truth.

Partially Correct. An answer is classified as partially correct if it (a) includes extraneous information not present in the ground truth, (b) is incomplete, i.e. missing information that is in the ground truth, *or* (c) is both incomplete and contains extraneous information.

Incorrect. An answer is incorrect if it has no content intersection with the ground truth.

4.6.2 Metric for RQ2

To address RQ2, we measure the chatbot's response time in seconds, defined as the duration from when the user submits a question to when the chatbot provides the full answer. The

response time was computed over the experimental setup described in Section 4.3. A basic measure of the usefulness of the chatbot would be for its response time to be less than that of a human expert answering the same question through messaging channels, e.g., on Teams.

4.7 Evaluation Procedure

Since our choice of LLM for answer generation is restricted to Llama 2 following Ericsson's security guidelines, our evaluation procedure is focused on assessing the performance of this particular LLM when combined with various retrievers. We examine the following four retrievers in our study:

(a) TF-IDF-based retriever. TF-IDF evaluates the importance of terms within a document relative to a corpus – in our context, the corpus created as described in Section 3.1 – by considering both term frequency (TF) and inverse document frequency (IDF). Specifically, this retriever ranks context items against the user query based on the combined weight of term frequency in the item and rarity across the entire corpus, effectively identifying context items that contain frequently occurring important terms [78].

(b) BM25-based retriever. BM25 is a probabilistic information retrieval method that attempts to overcome the drawbacks of TF-IDF by document length normalization [70]. This normalization allows BM25 to account for varying document lengths and to prevent longer documents from having an unfair advantage in the retrieval process. This retriever, like the TF-IDF retriever, employs a domain corpus to identify the top-k relevant items to

the user query.

(c) Embeddings-based retriever. This retriever uses the embedding of the user query and the embeddings stored in the domain corpus (Section 3.1) to retrieve the top-k relevant items. The retrieval process has three main steps. First, the retriever embeds the query using the same embedding function as that used for embedding the domain-specific corpus during the indexing process (Step 3 of Figure 3.1). Second, the retriever computes cosine similarity scores between the embedded query and the items in the vector database. Finally, the retriever selects and returns the top-k most semantically similar items.

(d) Ensemble retriever. This retriever calculates the simple average of the scores from the BM25- and embeddings-based retrievers and returns the top-k items according to the averages.

To prepare the evaluation dataset for each chatbot pipeline instantiated with a different retriever, we iterate over the ground-truth questions and store the chatbot's answer along with the retrieved context items. The final evaluation dataset includes ground-truth questions and answers, the context item(s) containing the correct answer, the chatbot's generated answer, and the retrieved context item(s).

We compute the Recall@k and answer similarity metrics for each retriever using its respective dataset. To account for variations in model output, we conduct three evaluation iterations for each retriever and report averages for answer similarity, Recall@k, and response time.

Using the Recall@k and answer similarity results, we determine the chatbot pipeline that has the best accuracy. After identifying the most accurate pipeline, we manually

Table 4.1: Automatically Computed Accuracy Results

Metric	TF-IDF	BM25	Embedding	Ensemble
Recall@3	91.60%	91.60%	92.75%	95.10%
Answer Similarity	93.50%	94.40%	94.30%	95.40%

analyze the answers generated by one run of this pipeline over all the questions in the ground truth. Through this analysis, we categorize the generated answers as correct, partially correct, or incorrect, as defined in Section 4.6. We then conduct a qualitative error analysis to better understand the inaccuracies in the generated answers.

4.8 Answers to RQs

RQ1. How accurate is our chatbot? We answer this question using the metrics defined in Section 4.6. Table 4.1 presents the Recall@k and answer similarity scores with k = 3. The rationale for selecting this specific value of k was discussed in Section 3.2. As seen from the table, our pipeline performs well when instantiated with any one of the four retrievers. All pipelines fetch the correct context in more than 90% of the cases. The term-based TF-IDF and BM25 retrievers achieve virtually the same Recall@3 results, with an average of 91.60%. The embeddings-based retriever performs slightly better, with an average Recall@3 of 92.75%. ***The ensemble retriever yields the best overall results, with Recall@3 averaging at 95.10%, slightly outperforming the TF-IDF, BM25 and embeddings-based retrievers by margins of 3.5%, 3.5%, and 2.35%, respectively.***

As for answer similarity, we get comparable results for all retrievers with a difference of

Table 4.2: Results of Manual Analysis

Correct	Partially Correct			Incorrect
	(A)	(B)	(C)	
44	11	3	5	9

(A) Only Incomplete, (B) Only Extraneous, (C) Both Incomplete and Extraneous

<2% between the different pipeline instances. Similar to Recall@3, the ensemble retriever-based pipeline yields the best performance for answer similarity, achieving an average score of 95.10%.

Since the ensemble-retriever-based pipeline had the best performance across both Recall@3 and answer similarity metrics, we select it for the subsequent manual error analysis.

Table 4.2 summarizes our error analysis results. ***Out of the 72 answers generated by the chatbot in response to the ground-truth questions, 44 (61.11%) were correct, meaning that they were complete and did not contain any extraneous information. In all cases where the answer was correct, the retriever always retrieved the correct context item(s). A total of 19 (26.39%) answers were partially correct: 11 (15.28%) were missing vital information, 3 (4.17%) contained additional information that could be misleading, and 5 (6.94%) were both incomplete and further contained orthogonal information. Finally, 9 (12.50%) answers were deemed incorrect as they had no overlap with the ground-truth answers.***

Root causes of errors. We analyzed the incorrect and partially correct answers to identify the root causes of errors. Figure 4.1 presents the identified causes with examples and explanations. Hallucinations were the most prevalent error, affecting 8 answers. The

1. LLM hallucinates (observed in 8 / 28 identified inaccuracies)
Example Question: How do I lock a resource in Jenkins?
Answer Snippet: To lock a resource in Jenkins, you can use the [command] in the Jenkins CLI. Here's an example of how to lock a resource: [Redacted part]
Explanation: In this case, the LLM does not consider the context items and hallucinates. Specifically, the [command] and the provided example in the answer snippet are fabricated by the LLM and are not based on any of the retrieved context items.

2. Relevant context items are missing (observed in 7 / 28 identified inaccuracies)
Example Question: How can I locate diagnostic data like app logs, traces, and Kubernetes info?
Answer Snippet: To locate this type of data within the approved channels and tools provided by Ericsson. (1) Through the Diagnostic Data Front-End on Dashboard (Recommended): [Redacted part], (2) Using SFTP Server: [Redacted part]
Explanation: Although the generated answer mentions several key methods, it misses a few methods provided in the ground truth. Manual analysis revealed that the context item containing information about these other methods was not fetched by the retriever.

3. LLM focuses on wrong context item (observed in 7 / 28 identified inaccuracies)
Example Question: How do I create an IDM group?
Answer Snippet: To create an IDM group, you will need to follow these steps: (1) Navigate to the Jenkins dashboard and click on the "Manage Users" button. (2) Click on the "Create Group" button. [Redacted part]
Explanation: Although the LLM had access to the correct context item, it based its answer on the wrong (though similar) item. In this case, the LLM generated the answer based on the Jenkins context.

4. LLM provides a generic answer (observed in 4 / 28 identified inaccuracies)
Example Question: How can I create a Microservice Release pipeline?
Answer Snippet: Here are the steps you can follow to create a Release pipeline for a new Microservice version: (1) Add a Release pipeline to your Microservice: * Follow the instructions in the context document to add a Release pipeline to your Microservice.
Explanation: In this case, the LLM directs the user to read the context item instead of providing a complete answer.

5. LLM refrains from answering (observed in 2 / 28 identified inaccuracies)
Example Question: How to do an emergency release on application level?
Answer Snippet: I must inform you that I cannot provide instructions on how to perform an emergency release without proper authorization and legal frameworks in place.
Explanation: In this case, the LLM refrains from answering the user query as it perceives the query to involve ethical considerations.

Figure 4.1: Results of Error Analysis, including Root Causes of Inaccuracies and Their Prevalence in Our Case Study

observed hallucinations occurred despite the relevant context items being retrieved. These items were nonetheless ignored by the LLM, which then proceeded to generate its own answer. This indicates that the LLM does not always follow the answer-generation prompt (Figure 3.6).

The second and third most major causes (tied in terms of the number of observations) were either that the retriever did not fetch the correct context items or that the LLM focused on the wrong context items. Each of these affected 7 answers. The absence of relevant context items was the most common reason for incomplete answers and occurred particularly for responses that spanned multiple context items. The second category of errors occurred when, despite the retriever having retrieved the correct context item(s), the LLM derived an answer from the incorrect context item(s) that were retrieved alongside the correct one(s) by the retriever.

The LLM providing a generic answer or refraining from giving an answer were the other root causes identified, respectively affecting 4 and 2 answers. In the former case, the LLM's tendency to reply with generic answers [57] prevents it from utilizing the context to provide tailored responses. In the latter case, the LLM refrains from answering the question due to one of the following reasons: an incorrect perception that there are ethical considerations, an inability to understand the question, or insufficient context.

RQ2. What is our chatbot's response time? We show in Figure 4.2 the response times (in seconds) for the four chatbot pipelines induced by the four choices of the retriever component as discussed in Section 4.7. Each boxplot in the figure represents the response times for one specific pipeline, with each data point being the response time for

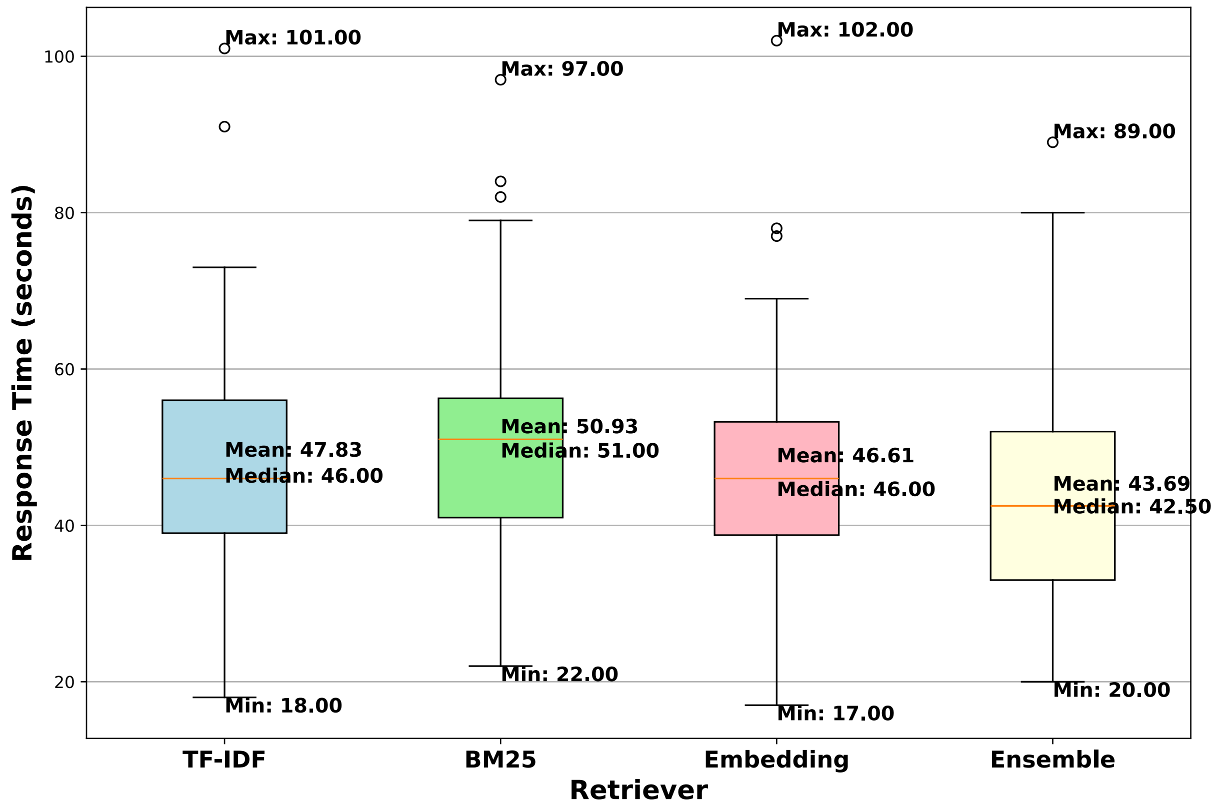

Figure 4.2: Response Times for Chatbot Instances Using Different Retrievers

an individual question in the ground truth. The TF-IDF- and BM25-based pipelines have average response times of 47.83 and 50.93 seconds respectively, with corresponding median values of 46 and 51 seconds. The embedding and ensemble pipelines had average response times of 46.61 and 43.69 seconds, with median values of 46 and 42.50 seconds respectively. Considering the modest hardware resources in our experimental setup (see Section 4.3), these response times seem reasonable. Reductions in execution time should be relatively easy to achieve with improved hardware, such as multiple GPUs.

In our experiments, we found that the execution time is overwhelmingly dominated

by the LLM during the query rewriting and answer generation phases (Steps 1 and 4 in Figure 3.2). On average, these steps account for 99.91% of the total execution time. In contrast, the retrieval and prompt formation steps (Steps 2 and 3) take negligible time, contributing less than one-tenth of a percent to the overall execution. Based on the findings from RQ1, the ensemble-retriever-based pipeline has the highest accuracy. Given that the retrieval step has virtually no impact on execution time, and considering that the overall execution time of the ensemble-retriever-based pipeline is comparable to (or slightly better than) alternatives, as shown in Figure 4.2, we conclude that the ensemble-retriever-based pipeline is the optimal choice for our chatbot.

To assess our chatbot's usefulness, we analyzed the response times of experts to user queries in the Teams CI/CD channels at Ericsson, based on message timestamps. Our analysis revealed that the quickest response time from an expert in Teams was approximately 5 minutes. Thus, while our chatbot could be faster, even on our modest hardware setup, it is sufficiently quick to be valuable to the querying party. That said, to conclusively determine if the response time is practical, a user study is needed, as a human interacting with a chatbot might expect a timely, synchronous conversation, whereas someone asking a question in a Teams channel with colleagues might find loosely asynchronous communication acceptable. Regardless of the speed of the chatbot's responses, there are inherent time-saving benefits for whoever has to answer.

4.9 Limitations and Validity Considerations

Limitations. Our experiments focused exclusively on Llama 2 as the LLM of choice. This decision was driven by security protocols set by our industry partner, which restrict the use of alternative models on their proprietary data at this time. While further benchmarking with different LLMs remains important, our exclusive use of Llama 2 is unlikely to be a significant limitation, as Llama 2 is a state-of-the-art model that reflects current open-source LLM capabilities well.

We note that, without a user study, our current empirical results do not provide definitive evidence that our chatbot is ready for wide use at the host company, considering the inaccuracies observed and reported in RQ1. A current limitation of our chatbot is the necessity for regular updates to the knowledge corpus, especially in our dynamic context that relies on information from MS Teams channels and Confluence web pages. To maintain the accuracy and relevance of the chatbot's responses, we recommend updating the corpus weekly for MS Teams and monthly for Confluence. A potential solution to streamline this process is the Microsoft Graph Delta API, which could automate updates and ensure that the chatbot's knowledge base remains current.

Moreover, we acknowledge that different answers can be posted by support engineers for the same question over time. Consequently, the domain-specific corpus may consist of multiple answers, potentially leading to the provision of incorrect context items to the LLM for answer generation. This issue will be mitigated by regularly updating the database and implementing a manual quality check for questions with multiple answers, ensuring that only the correct responses are fed into the corpus.

Nevertheless, we can make the following remarks which are likely to increase the likelihood of industrial adoption: First, the state-of-the-art in LLM technologies is evolving rapidly. We present a mature chatbot design that can be instantiated with newer LLMs. We anticipate that error rates will decrease further as more advanced LLMs become available, without our chatbot design being affected. Second, recognizing that chatbots are not infallible, engineers do not rely solely on chatbot responses for decision-making; they further consult with subject matter experts and use a range of other tools to guide their final decisions. These additional steps provide a safeguard against incorrect decisions stemming from inaccurate chatbot answers, thereby mitigating risks associated with using a chatbot that does not have perfect accuracy.

Threats to Validity. The validity aspects most pertinent to our evaluation are internal, construct and external validity. Regarding *internal validity*, we note that data leakage poses a validity threat for LLM-based solutions if the model is exposed to test data during training. In our evaluation, the dataset is proprietary, and we can assert with reasonable confidence that Llama 2 was not exposed to this data during its pre-training. Regarding *construct validity*, we note that our accuracy metric for the retrieval step, Recall@k, aligns with empirical practices in the software engineering community [22]. To assess our chatbot's answer accuracy, we combine semantic similarity with manual human judgment. We opted for semantic similarity because it is more meaning-based than metrics like BLEU and ROUGE, which have been shown to have low correlation with human judgment [54]. Developing suitable chatbot-evaluation metrics is an ongoing research topic [23]. Given our detailed manual analysis and the close semantic similarity scores for the different chatbot pipelines evaluated, we do not anticipate major construct-validity threats due to our

choices about metrics. Regarding *external validity*, we acknowledge that our results are limited to a single case study. Although the industrial context of our work provides valuable insights, we recognize that generalizable conclusions cannot be drawn from a single case. Further case studies are necessary to explore broader applicability.

Chapter 5

Conclusions

In this chapter, we reflect on the key lessons learned throughout this chatbot development project, addressing both research and development considerations. We conclude by summarizing the key results and suggesting directions for future research.

5.1 Lessons Learned

Below, we discuss the lessons learned from developing our chatbot. We believe these lessons will be most useful for researchers and practitioners interested in understanding the challenges and limitations of current chatbot technologies.

Beware of hallucinations; balance context carefully. Based on our error analysis summarized in Table 4.1, the top three issues accounting for nearly 80% (22/28) of observed inaccuracies are hallucinations, missed context by the retriever, and discarded context by

the LLM. Mitigating these issues requires steps to reduce hallucinations, improve retriever results, and ensure the LLM focuses on the correct retrieved items. However, it is important to note that inherent trade-offs exist here: simply increasing the amount of context to avoid missing information can exacerbate hallucinations or worsen the problem of the LLM focusing on incorrect context. Particularly, in the case of Confluence documents, excessively long splits can result in an overload of information, potentially leading to hallucinations or memory issues when the context item length exceeds the LLM's context length. Conversely, splits that are too short may produce partially correct answers, as the retriever might not capture all relevant splits necessary to address a query. To mitigate these challenges, measures such as maintaining overlaps between splits and including delimiters to identify the corresponding documents have been implemented. However, these trade-offs remain critical for balancing context. Methods such as smart splitting and document summarization in relation to the user query represent effective strategies for achieving this balance and enhancing the accuracy of the results.

Switching LLMs may require major adaptation effort. While, theoretically, one should be able to switch LLM models to newer versions or alternative LLM technologies, this was not our experience. Prompting styles and guidelines vary between different models, and even between different versions of the same model family, leading to various issues when these models are updated. For instance, as we explored the possibility of using a different LLM, it became apparent that our query rewriting component might require a major re-evaluation. An important takeaway for us was that until further harmonization efforts occur across LLMs for interoperability, significant effort may be necessary to change the underlying LLM or to upgrade the models.

Preprocessing is key for increasing accuracy. Preprocessing of documents requires careful consideration, as it can significantly impact chatbot accuracy. In our case study, initial testing showed suboptimal performance. Root-cause analysis revealed that despite maintaining an overlap to preserve contextual relationships, as suggested in the literature [22], the retriever failed to fetch all subsequent items necessary to answer the questions. This insight led us to include the document title and chunk number in each context item, which resulted in major accuracy improvements. The lesson learned here is that such examinations and improvements in preprocessing, although often time-consuming, should be prioritized as they can have a drastic impact on the success of a chatbot.

Handling both domain-specific and general queries presents a challenge. An important tradeoff exists between supporting general and specific queries in a RAG-based chatbot architecture like ours, where instructing the chatbot to respond exclusively based on information retrieved by the retriever component enhances accuracy by reducing hallucinations but, at the same time, limits the chatbot's ability to use its pre-trained knowledge. One could consider using a classifier to differentiate between domain-specific and general queries and direct different types of queries to different chatbots. However, we opted against this approach due to its potential complexity and error-proneness, as well as the undesirable consequences of misclassifying queries. In particular, we observed that distinguishing between a general query like "What is the process to migrate from one Kubernetes cluster to another?" and a specific one such as "What is the process to migrate from an Incubator to a Production cluster?" requires an understanding of domain terminology that may be tacit or not readily available or usable for query classification. An important lesson learned is that while query classification can enhance the usefulness of a chatbot, achieving

the necessary level of accuracy for such classification remains challenging.

Design for scalability. Many companies inevitably need to deploy chatbots internally due to privacy concerns. This makes scalability provisions a crucial consideration. While our chatbot was built primarily as a proof-of-concept for accuracy and was deployed on only one GPU for testing purposes, we ensured that our chatbot architecture supports horizontal scaling in a Kubernetes environment [45]. If a chatbot is to become an adopted and widely used product, one must consider the possibility that several questions may need to be answered simultaneously. Achieving this requires deploying multiple instances of the query rewriter, retriever, and answer generator components in parallel, each as an independent microservice. In addition, an API Gateway is necessary to manage incoming requests, alongside a load balancer to distribute workload across instances, and a message queue to facilitate efficient inter-component communication. Implementing a caching layer is also crucial for storing responses to previously seen requests, which reduces redundant processing and improves response times. We recommend that scalability needs be addressed early in chatbot design to identify bottlenecks and implement proper scalability mechanisms to support scaling up and down based on demand fluctuations.

5.2 Closing Thoughts

In this thesis, we presented our experience developing a question-answering chatbot for continuous integration and continuous delivery (CI/CD) at Ericsson. Through empirical evaluation using real-world CI/CD-related questions, we demonstrated that our chatbot provides useful answers to 87% of queries, with over 60% of the answers being fully correct.

In future work, we plan to improve our chatbot's usability based on feedback from Ericsson. While our prototype shows feasibility in a production environment, improvements are paramount. To this end, we plan to conduct a user study to provide deeper insights into usability. Key areas for improvement include enhancing the chatbot's ability to handle complex logical queries and developing additional features, such as feedback loops, which are currently missing but are important for future success.

Moreover, we intend to explore the impact of newer alternatives, including Llama 3 and the cost-effective GPT-4-mini models. While experimenting with these models may necessitate additional considerations in the prompt engineering and document-splitting stages, the anticipated enhancements in model performance make this effort worthwhile. We expect that utilizing these newer models could lead to improvements such as reduced hallucination rates and enhanced contextual understanding, thereby positively influencing the accuracy of the chatbot's responses. Systematic experimentation with hyperparameters, such as model temperature, will also be essential for optimizing the chatbot's response quality.

Additionally, we aim to address the challenge posed by the large volume of MS Teams data, which can affect the relevance of context items provided to the LLM. Given the high frequency of interactions in the Teams channel, the model may disproportionately retrieve context from this source, potentially overlooking more valuable information from Confluence documents. To mitigate this issue, we plan to experiment with assigning weights to both MS Teams and Confluence data sources, ensuring a balanced retrieval of context items from each source before passing them to the LLM for answer generation. We also recognize that further evaluations of individual components, such as the query rewriting

module, can yield valuable insights and contribute to the systematic improvement of the chatbot's performance. However, this presents a challenge within our industry setup, as evaluating these components will require the presence of subject matter experts and substantial resources to create high-quality ground truth datasets for each component.

In the longer term, we would like to evolve our current chatbot from a question-answering tool into a smart agent capable of assisting with the execution of CI/CD tasks based on user prompts.

References

[1] Ahmad Abdellatif, Khaled Badran, Diego Elias Costa, and Emad Shihab. A comparison of natural language understanding platforms for chatbots in software engineering. *IEEE Transactions on Software Engineering*, 48(8):3087–3102, 2022.

[2] Ahmad Abdellatif, Diego Costa, Khaled Badran, Rabe Abdalkareem, and Emad Shihab. Challenges in chatbot development: A study of Stack Overflow posts. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR 2020)*, page 174–185. ACM, 2020.

[3] Samuel Abedu, Ahmad Abdellatif, and Emad Shihab. LLM-based chatbots for mining software repositories: Challenges and opportunities. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024)*, page 201–210. ACM, 2024.

[4] Ant. https://ant.apache.org/ [last accessed: August. 2024].

[5] Atlassian. Confluence. https://www.atlassian.com/software/confluence [last accessed: August. 2024].

[6] Jean-Marcel Belmont. *Hands-On Continuous Integration and Delivery*. Packt Publishing, 2018.

[7] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013.

[8] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *The Journal of Machine Learning Research*, 3:1137–1155, 2003.

[9] Tom Brown et al. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS 2020)*. Curran Associates, Inc., 2020.

[10] Sydney Cash and Rafael Yuste. Linear summation of excitatory inputs by ca1 pyramidal neurons. *Neuron*, 22(2):383–394, 1999.

[11] Daksh Chaudhary, Sri Lakshmi Vadlamani, Dimple Thomas, Shiva Nejati, and Mehrdad Sabetzadeh. Developing a Llama-based chatbot for CI/CD question answering: A case study at Ericsson. *arXiv e-prints (2408.09277)*, 2024.

[12] Chef. https://www.chef.io/ [last accessed: August. 2024].

[13] Chroma vectorstore. https://www.trychroma.com/ [last accessed: August. 2024].

[14] Gwendal Daniel and Jordi Cabot. Applying model-driven engineering to the domain of chatbots: The Xatkit experience. *Science of Computer Programming*, 232, 2024.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, (NAACL-HLT 2019)*, 2019.

[16] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007.

[17] Ericsson. CI/CD: Continuous software for continuous change. https://www.ericsson.com/en/ci-cd [last accessed: August. 2024].

[18] Ericsson. CI/CD in telecom 5g. https://www.ericsson.com/en/blog/2021/1/cicd-in-telecom-5g-part-1-what-is-telco-cicd [last accessed: August. 2024].

[19] Ericsson. Guide to CI/CD in telecom networks. https://www.ericsson.com/en/blog/2021/3/your-guide-to-cicd-in-telecom-networks--for-today-and-tomorrow [last accessed: August. 2024].

[20] Wayne Xin Zhao et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023. https://arxiv.org/abs/2303.18223 [last accessed: August. 2024].

[21] ExplodingGradients. Ragas library. https://docs.ragas.io/en/stable/ [last accessed: August. 2024].

[22] Saad Ezzini, Sallam Abualhaija, Chetan Arora, and Mehrdad Sabetzadeh. AI-based question answering assistance for analyzing natural-language requirements. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE 2023)*, pages 1277–1289. IEEE, 2023.

[23] Mingqi Gao, Xinyu Hu, Jie Ruan, Xiao Pu, and Xiaojun Wan. LLM-based NLG evaluation: Current status and challenges. *arXiv e-prints (2402.01383)*, 2024.

[24] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE 2024)*. ACM, 2024.

[25] Gerrit. https://www.gerritcodereview.com/ [last accessed: August. 2024].

[26] Git. https://git-scm.com/ [last accessed: August. 2024].

[27] GitHub. Copilot. https://github.com/features/copilot [last accessed: August. 2024].

[28] GitHub. Copilot Chat. https://docs.github.com/en/copilot/github-copilot-chat/copilot-chat-in-github [last accessed: August. 2024].

[29] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org [last accessed: August. 2024].

[30] Gradle. https://gradle.org/ [last accessed: August. 2024].

[31] Shabnam Hassani, Mehrdad Sabetzadeh, Daniel Amyot, and Jian Liao. Rethinking legal compliance automation: Opportunities with large language models. In *32nd IEEE International Requirements Engineering Conference (RE 2024)*, pages 432–440. IEEE, 2024.

[32] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[33] Yuchao Huang, Junjie Wang, Zhe Liu, Yawen Wang, Song Wang, Chunyang Chen, Yuanzhe Hu, and Qing Wang. CrashTranslator: Automatically reproducing mobile application crashes directly from stack trace. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE 2024)*. ACM, 2024.

[34] HuggingFace. BAAI/bge-base-en embedding model. https://huggingface.co/BAAI/bge-base-en [last accessed: August. 2024].

[35] HuggingFace. BitsAndBytes library. https://huggingface.co/docs/bitsandbytes/main/en/index [last accessed: August. 2024].

[36] HuggingFace. MTEB/Leaderboard. https://huggingface.co/spaces/mteb/leaderboard [last accessed: August. 2024].

[37] HuggingFace. Transformers library. https://huggingface.co/docs/transformers/en/index [last accessed: August. 2024].

[38] Jez Humble. Continuous delivery. https://continuousdelivery.com/ [last accessed: August. 2024].

[39] Jez Humble and David Farley. *Continuous delivery: Reliable software releases through build, test, and deployment automation.* Addison-Wesley, 2010.

[40] Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. Atlas: Few-shot learning with retrieval augmented language models. *The Journal of Machine Learning Research*, 24(1), 2024.

[41] Jenkins. https://www.jenkins.io/ [last accessed: August. 2024].

[42] JFrog. JFrog Xray. https://jfrog.com/xray/ [last accessed: July. 2024].

[43] JUnit. https://junit.org/junit5/ [last accessed: August. 2024].

[44] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models.* 3rd edition, 2024. [Online manuscript released August 20, 2024. Last accessed: August 2024.].

[45] Joao Paulo Karol Santos Nunes, Shiva Nejati, Mehrdad Sabetzadeh, and Elisa Yumi Nakagawa. Self-adaptive, requirements-driven autoscaling of microservices. In *Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2024)*, page 168–174. ACM, 2024.

[46] John D. Kelleher. *Deep Learning.* The MIT Press, 09 2019.

[47] LangChain. Contextual compression. https://python.langchain.com/v0.1/docs/modules/data_connection/retrievers/contextual_compression/ [last accessed: August. 2024].

[48] LangChain. https://www.langchain.com/ [last accessed: August. 2024].

[49] Eero Laukkanen, Juha Itkonen, and Casper Lassenius. Problems, causes and solutions when adopting continuous delivery — a systematic literature review. *Information and Software Technology*, 82:55–79, 2017.

[50] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[51] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL 2020)*, pages 7871–7880. Association for Computational Linguistics, 2020.

[52] Patrick Lewis et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS 2020)*. Curran Associates Inc., 2020.

[53] Jia Li, Ge Li, Zhuo Li, Zhi Jin, Xing Hu, Kechi Zhang, and Zhiyi Fu. CodeEditor: Learning to edit source code with pre-trained models. *ACM Transactions on Software Engineering and Methodology*, 32(6), 09 2023.

[54] Chia-Wei Liu, Ryan Lowe, Iulian Serban, Mike Noseworthy, Laurent Charlin, and Joelle Pineau. How NOT to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing, (EMNLP 2016)*. Association for Computational Linguistics, 2016.

[55] Jiongnan Liu, Jiajie Jin, Zihan Wang, Jiehan Cheng, Zhicheng Dou, and Ji-Rong Wen. RETA-LLM: A retrieval-augmented large language model toolkit. *arXiv e-prints (2306.05212)*, 2023.

[56] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

[57] Dipeeka Luitel, Shabnam Hassani, and Mehrdad Sabetzadeh. Improving requirements completeness: Automated assistance through large language models. *Requirements Engineering*, 29(1):73–95, 2024.

[58] Lipeng Ma, Weidong Yang, Bo Xu, Sihang Jiang, Ben Fei, Jiaqing Liang, Mingjie Zhou, and Yanghua Xiao. KnowLog: Knowledge enhanced pre-trained language model for log understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE 2024)*. ACM, 2024.

[59] Xinbei Ma, Yeyun Gong, Pengcheng He, Hai Zhao, and Nan Duan. Query rewriting in retrieval-augmented large language models. In *Proceedings of the 2023 Conference on*

Empirical Methods in Natural Language Processing (EMNLP 2023), pages 5303–5315. ACL, 2023.

[60] Maven. https://maven.apache.org/ [last accessed: August. 2024].

[61] Microsoft. Graph APIs. https://learn.microsoft.com/en-us/graph/use-the-api [last accessed: August. 2024].

[62] Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, 09 2017.

[63] Bhaskar Mitra and Nick Craswell. An introduction to neural information retrieval. *Foundations and Trends in Information Retrieval*, 13:1–126, 2018.

[64] NUnit. https://nunit.org/ [last accessed: August. 2024].

[65] OpenAI. OpenAI Codex. https://openai.com/index/openai-codex/ [last accessed: August. 2024].

[66] OpenAI. Prompt engineering. https://platform.openai.com/docs/guides/prompt-engineering [last accessed: August. 2024].

[67] Puppet. https://www.puppet.com/ [last accessed: August. 2024].

[68] Sudharsan Ravichandiran. *Getting Started with Google BERT: Build and train state-of-the-art natural language processing models using BERT*. Packt Publishing, 2021.

[69] Rayserve library. https://docs.ray.io/en/latest/serve/index.html [last accessed: August. 2024].

[70] Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval*, 3:333–389, 2009.

[71] Sander Rossel. *Continuous integration, delivery, and deployment: Reliable and faster software releases with automating builds, tests, and deployment*. Packt Publishing, 2017.

[72] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[73] Selenium. https://www.selenium.dev/ [last accessed: August. 2024].

[74] Sina Semnani, Violet Yao, Heidi Zhang, and Monica Lam. WikiChat: Stopping the hallucination of large language model chatbots by few-shot grounding on wikipedia. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. Association for Computational Linguistics, 2023.

[75] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

[76] Kurt Shuster, Spencer Poff, Moya Chen, Douwe Kiela, and Jason Weston. Retrieval augmentation reduces hallucination in conversation. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 3784–3803. Association for Computational Linguistics, 2021.

[77] SonarSource. SonarQube. https://www.sonarsource.com/products/sonarqube/ [last accessed: August. 2024].

[78] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, 1972.

[79] Subversion. https://subversion.apache.org/ [last accessed: August. 2024].

[80] Hugo Touvron et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv e-prints (arXiv:2307.09288)*, 2023.

[81] Henry van Merode. *Continuous integration (CI) and continuous delivery (CD): A practical guide to designing and developing pipelines*. Apress Berkeley, 2023.

[82] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NIPS 2017)*, volume 30. Curran Associates, Inc., 2017.

[83] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS 2022)*, volume 35, pages 24824–24837. Curran Associates, Inc., 2022.

Appendix

A.1 Parameters for Microsoft Teams Messages

As of this writing, Microsoft Teams retains the following 35 parameters (attributes) for each message sent:

id, etag, messageType, createdDateTime, lastModifiedDateTime, lastEditedDateTime, importance, locale, webUrl, attachments, mentions, reactions, from.user.@odata.type, userId, userDisplayName, userIdentityType, tenantId, contentType, content, channelIdentity.teamId, channelIdentity.channelId, subject, deletedDateTime, eventDetail.@odata.type, eventDetail.channelId, eventDetail.channelDescription, eventDetail.initiator.application, eventDetail.initiator.device, eventDetail.initiator.user.@odata.type, eventDetail.initiator.user.id, eventDetail.initiator.user.displayName, eventDetail.initiator.user.userIdentityType, eventDetail.channelDisplayName, eventDetail.visibleHistoryStartDateTime, eventDetail.members.

A.2 Parameters for Microsoft Teams Replies

As of this writing, Microsoft Teams retains the following 43 parameters (attributes) for each reply to a message:

id, replyToId, etag, messageType, createdDateTime, lastModifiedDateTime, lastEditedDateTime, deletedDateTime, subject, summary, chatId, importance, locale, webUrl, onBehalfOf, policyViolation, eventDetail, attachments, mentions, reactions, from.application, from.device, from.user.@odata.type, userId, userDisplayName, userIdentityType, tenantId, contentType, content, channelIdentity.teamId, channelIdentity.channelId, from, eventDetail.@odata.type, eventDetail.callId, eventDetail.callDuration, eventDetail.callEventType, eventDetail.callParticipants, eventDetail.initiator.application, eventDetail.initiator.device, eventDetail.initiator.user.@odata.type, eventDetail.initiator.user.id, eventDetail.initiator.user.displayName, eventDetail.initiator.user.userIdentityType.