SAPIENZA
UNIVERSITÀ DI ROMA

Automating Cloud Service Specification: AI-Driven Approach to Requirement Engineering

Faculty of Information Engineering, Informatics and Statistics
Master's Degree in Computer Science

Alberto Cotumaccio
ID number 1852040

Advisor
Prof. Emiliano Casalicchio

Co-Advisor
Dr. Danilo Magliarisi

Academic Year 2023/2024

Automating Cloud Service Specification: AI-Driven Approach to Requirement Engineering

Master's Thesis. Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: cotumaccio.1852040@studenti.uniroma1.it

Abstract

The evolution of cloud computing has marked a revolution in the world of technology and information, offering access to IT resources that are flexible, scalable, cost-effective, and easily accessible via the Internet. The cloud computing market is huge, and the diversity and specificity of products from various vendors lead to difficulties in selecting appropriate cloud services.

This thesis proposes an innovative approach that automates, through artificial intelligence, the Requirements Engineering (RE) process to translate high level cloud needs into low-level specifications. This process is expected to be integrated into a module within a cloud service broker.

To establish knowledge boundaries for the project, a new ontology for cloud computing is proposed, with the goal of clearly and hierarchically organizing all possible cloud requirements. It has a three-level hierarchical structure: main categories, subcategories and specific cloud features.

For automating the classification of sentences into main categories of the ontology, a transformer-based multi-label BERT model is proposed. It is chosen for its superiority in understanding natural language, as reported in literature. The model is evaluated through typical metrics such as precision and recall, it performs excellently proving effective in identifying cloud requirements and assigning correct labels.

For fine-tuning the model, a custom dataset is created by collecting public requests for proposals (RFPs) and exploiting generative artificial intelligence to simulate the writing of a few documents representative of our scenario, since no sample documents were provided. This dataset, containing sentences outlining both cloud and non-cloud requirements, is manually annotated and explored.

In the final steps, an NLP method based on the use of keywords is presented to extract cloud-specific features from the categorized requirements, and then match these features to the corresponding products offered by major cloud service providers, such as Amazon AWS, Google Cloud and Microsoft Azure.

The efficacy of the proposed approach is validated through a simulation of a real-world scenario, culminating in a final complete report outlining the various steps of the methodology.

Contents

1 Introduction **1**
1.1 Background and Motivation 1
1.2 Structure of the Thesis 2

2 Overview of Requirement Engineering **3**
2.1 Introduction to Requirement Engineering 3
2.2 Review of Specification Languages and Tools 4
2.3 Artificial Intelligence in Requirements Engineering 6

3 Methodology **9**

4 Ontology for Cloud Computing Concepts **11**
4.1 Review of Existing Cloud Ontologies 11
4.2 Ontology Design 12

5 Data Collection Procedure **17**
5.1 Review of Existing Datasets 17
5.2 Constructing the Dataset 18
5.3 Dataset Preprocessing 19
5.4 Dataset Exploration 20

6 Fine-Tuning BERT for Classifying Sentences **25**
6.1 Technical Background 25
6.2 Experimental Setup for Fine-Tuning 27
6.3 Training Implementation 29
6.4 Model Evaluation 31

7 Methodology Validation **35**
7.1 Parsing the User-Supplied Document 35
7.2 Sentences Classification 36
7.3 Keyword Matching: Cloud Features Extraction 39
7.4 Mapping to Cloud Products 42

8 Conclusion and Future Work **43**

Bibliography **47**

Chapter 1

Introduction

1.1 Background and Motivation

The evolution of cloud computing has marked a revolution in the world of technology and information, offering access to IT resources that are flexible, scalable, cost-effective, and easily accessible via the Internet. This innovation transforms the way organizations access and use technology infrastructure by allowing them to consume resources as services, similar to the way they consume household utilities such as gas and electricity, with a flexible pricing model that allows them to pay only for the resources they actually use (*pay-as-you-go* model). This eliminates the need for companies to invest in and maintain on-premise computing infrastructures, leading to a significant reduction in operating and capital costs [1].

Cloud computing consists of several service models, each of which provides access to different types of resources. The Infrastructure as a Service (IaaS) model provides access to virtualized computing resources, such as servers, networks, and storage systems, without the need to purchase or manage physical hardware. Platform as a Service (PaaS) provides a development and deployment environment in the cloud, including operating systems, execution environments for programming languages and databases, facilitating application development without the complexity of managing the underlying infrastructure. Software as a Service (SaaS), finally, allows users to access application software and services over the Internet, eliminating the need to install and manage applications on their IT infrastructure.

The cloud computing market is huge, and the variety and complexity of services offered pose considerable challenges in their selection. In addition, each cloud provider, such as Amazon AWS, Google Cloud, and Microsoft Azure, offers a particular description of its services, which is usually incompatible with other service providers. This accentuates the problem of interoperability, leading to the phenomenon of vendor lock-in, which limits users in choosing a vendor [2].

In the dynamic and constantly changing environment of cloud computing, it becomes critical to have tools to guide users in accurately identifying cloud service requirements. Hence, the integration of a cloud service specification module into a service broker [3] plays an important role. By acting as an intermediary, the broker improves efficiency in the selection and management of cloud services by facilitating the matching of user requirements with vendor offerings. Interaction

with the broker through the service specification module enables the transformation of needs expressed in natural language into detailed technical specifications, ensuring clarity and accuracy, and facilitating communication with other modules of the broker, such as those dedicated to service recommendation and discovery, which are used for proposing the most appropriate cloud solutions. Figure 1.1 shows the crucial role of the service specification block.

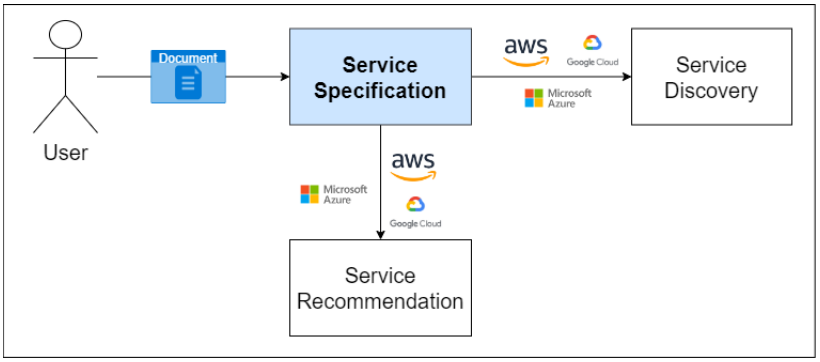

Figure 1.1. Portion of Cloud Service Broker: Key Role of the Service Specification

The motivation behind this thesis stems from the observation that, despite the availability of tools and methodologies for the selection and management of cloud services, a significant gap persists in the initial user specification and requirements definition phase. Consequently, this research focuses on developing an advanced cloud service specification module that can be effectively integrated within a cloud service broker, making the cloud service selection process more accessible, accurate and efficient for the end user. The project aims to create an interdisciplinary framework to break down barriers between different fields of study, such as requirements engineering, artificial intelligence and cloud computing.

1.2 Structure of the Thesis

The thesis is structured to interweave theoretical foundations and practical applications. Chapter 2 introduces the field of Requirements Engineering, emphasizing the role of artificial intelligence (AI) in this context. It also delves into the exploration of specification languages and tools. Chapter 3 outlines the methodology employed in this study, followed by a discussion of the ontology creation in chapter 4. This chapter includes a review of existing cloud ontologies and the creation of a custom one that organizes cloud computing-related concepts in a hierarchical manner. Chapter 5 focuses on reviewing existing datasets and constructing a customized one. Chapter 6 describes the process of developing a multi-label BERT model for sentences categorization, providing insights about the training and evaluation. Validation of the developed methodology is presented in chapter 7, where a real-world scenario is simulated. This scenario includes parsing user-supplied documents, classifying sentences into main categories, extracting specific cloud features, and mapping those features to cloud products, all leading up to the generation of a final report. Chapter 8 offers a summary of the results and conclusions.

Chapter 2

Overview of Requirement Engineering

2.1 Introduction to Requirement Engineering

Requirements Engineering (RE) represents an iterative process that deals with the identification, elaboration, structuring, specification, analysis, and management of requirements expressed by stakeholders in software projects [4]. This process plays a crucial role in identifying stakeholder goals, needs and expectations. Effective requirements management is critical to minimizing the risk of project failure and improving the quality of the final product.

Studies in the field point out that most errors in software development projects originate in the RE phase, with requirements ambiguity recognized as a major cause of failure. As pointed out by Bohem [5], correcting errors in requirements at advanced stages of the project can be up to 200 times more expensive than correcting them during the requirements definition phase. This problematic issue stems mainly from the fact that requirements are often expressed in natural language, which increases the complexity of requirements engineering because of its intrinsic ambiguity, incompleteness, and imprecision.

A requirement [6] is defined by the IEEE (Institute of Electrical and Electronics Engineers) as:

1. a condition or capability necessary for a user (person or system) to solve a problem or achieve a goal;

2. a condition or capability that must be provided by a system or part of a system to satisfy a contract, standard, specification, or any other formal document;

3. a documented representation of a condition or capability, as in (1) or (2).

This definition highlights the importance of accurately identifying and documenting the requirements and capabilities required to ensure the success of systems, emphasizing the need for clarity and precision in defining requirements.

Figure 2.1 illustrates the iterative process of Requirements Engineering (RE), commonly proposed by most studies, which consists of 4 different phases:

1. **Elicitation**: this first phase involves gathering requirements from the various stakeholders. The goal is to thoroughly understand the users' needs and expectations and build a common vision for the project.

2. **Analysis**: in this phase, requirements are reviewed, classified and detailed. This phase focuses on organizing requirements by type and refining them to avoid ambiguity and repetition, and ensure that they are complete, consistent, verifiable, and traceable.

3. **Specification**: in this phase, the analyzed requirements are formalized into a clear and unambiguous document. The intent is to make the specification understandable, ensuring that all requirements are precisely defined.

4. **Validation**: the last phase is concerned with verifying and validating the requirements to ensure that they have been understood and documented correctly. Any errors or misunderstandings in the specified requirements are then identified to ensure that they correctly express the needs of stakeholders. Any discrepancies require a return to the elicitation and analysis stages for further refinement.

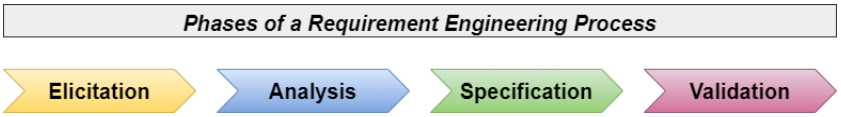

Figure 2.1. Iterative Requirements Engineering (RE) Process

2.2 Review of Specification Languages and Tools

This section provides an in-depth review of the research conducted on specification languages and tools used in the field of Requirements Engineering. Are analyzed safety-critical systems, graphical and mathematical notations and low-code platforms. Additionally, tools designed to facilitate the deployment of cloud infrastructures are examined. To conclude, an analysis is presented to identify the most widely used requirements management tools in the industry.

Safety-Critical Systems Architecture Analysis and Design Language (AADL) [7] is a formal modeling language used in the field of software engineering. It provides a framework for describing the architecture of safety-critical systems. This framework emphasizes the separation and distinct interactions among the various system components. AADL incorporates ReqSpec, a textual language for specifying requirements, which allows stakeholders to more clearly articulate their goals and needs. The main purpose of ReqSpec is to facilitate the iterative process of eliciting, defining and modeling requirements, particularly for real-time embedded systems. It provides support for progressive requirements refinement in conjunction with system design. In addition, ReqSpec enables qualitative and quantitative analysis of requirements specifications. Finally, it enables verification of corresponding system architecture models, ensuring that they are in line with established requirements.

Graphical Notations Graphical notations such as UML (Unified Modeling Language) [8], SysML (Systems Modeling Language) [9] and ArchiMate [10] offer visual representations of requirements, systems and their interactions. UML is predominantly used for software modeling, SysML extends UML for broader engineering applications, and ArchiMate focuses on enterprise architecture modeling. Models in these notations are geared toward refining requirements or design activities rather than specifying initial requirements. They are also limited in specifying nonfunctional requirements. Using one of these nontext notations often requires a complex translation of the source requirements, which can introduce additional errors.

Mathematical Notations One of the main mathematical notations is the formal language Z [11], which provides a solid basis for formal verification and validation of software systems. It is based on first-order logic and provides a mathematical notation for representing requirements, such as sets, functions, relations, constraints, and axioms. The use of Z facilitates precise understanding of system properties and demonstration of correctness with respect to specified requirements.

Low-Code Platforms Low-code platforms are a revolution in software development, facilitating the rapid creation of applications in line with market needs and reducing the need for in-depth technical expertise. These tools are characterized by their ability to integrate requirements as early as the prototyping stage, thus overcoming several limitations associated with traditional development methods. The languages used by low-code platforms sit somewhere in between natural and programming languages. A research project introduced a method based on Model-Driven Development to semi-automatically generate software applications using a language called ITLingo ASL (Application Specification Language) [12]. Through the integration of ITLingo ASL with the Quidgest Genio platform [13], it is possible to transform detailed requirements specifications into low-code designs, enabling automatic code generation for the application. A key advantage of this approach is the ability for developers to write precise specifications in a language that is platform independent, while maintaining the high performance in code generation.

Tools for Cloud Infrastructure Management In the field of cloud computing, Model-Driven approaches have been proposed to simplify the management of cloud infrastructures. Among these, the Argon tool [14] and the European Piacere project [15] stand out. Both enable high-level modeling of the final state of the desired cloud infrastructure, which can then be specialized for different cloud providers. They allow developers to focus on logic rather than low-level technical details. Using graphical notations, they allow services, defined graphically, to be automatically transformed into Infrastructure as Code (IaC) scripts, for example in Terraform. IaC is an IT infrastructure management practice that automates the configuration and deployment of resources and services through code, rather than through manual processes. Although these projects represent major initiatives, they do not achieve complete automation and are primarily aimed at developers with a technical background. In fact, they assume a detailed and quantified definition of initial requirements, which is necessary to properly proceed with manual infrastructure modeling.

Requirement Management Tools in Industry In industry, requirements management tools provide a solid platform for requirements gathering, analysis, documentation and tracking, ensuring that all stakeholders are aligned and that the final product accurately reflects the initial needs.

The breadth of solutions available in the market demonstrates the growing importance of effective requirements management. Solutions range from cloud-based solutions, which allow easy real-time access and collaboration, to on-premise solutions, ideal for organizations that require greater control over data security. Common features include requirements tracking, version control, integration with other software development platforms, and tools for analyzing the impact of changes.

A distinctive aspect of modern requirements management tools is their ability to facilitate effective collaboration among various project stakeholders, including developers, managers and customers. This is made possible through features such as discussion boards, automated notifications, and customizable dashboards that allow all participants to stay up-to-date on project status and actively contribute to requirements definition and review. Some of the main requirements management tools available in the market are shown in Figure 2.2.

Figure 2.2. Top Requirements Management Tools

2.3 Artificial Intelligence in Requirements Engineering

In the field of computer science, Artificial Intelligence (AI) is defined as a system capable of acting and thinking both human and rational [16]. The use of AI in RE aims to automate the requirements management process to solve the challenges associated with manual analysis of natural language that can be time-consuming, error-prone, and variable in interpretation. Much research has been conducted in the field of NLP for Requirement Engineering (NLP4RE) [17], where information extraction and classification emerge as the most popular activities, also benefiting from the presence of a very active community.

Natural Language Processing (NLP) [18] is a set of computational techniques for analyzing and representing natural texts at one or more levels of linguistic analysis in order to achieve human-like language processing for a variety of tasks or applications. The concept of "levels of linguistic analysis" refers to the phonetic, morphological, lexical, syntactic, semantic, discursive, and pragmatic analysis of language.

In the 1980s, the first proposed NLP tools were based on extracting relevant entities from requirements text based on simple syntactic rules, assuming that natural language requirements are expressed in a constrained and predictable format. This approach, however, often proved inadequate because of the complexity and variability of natural language.

Around the 1990s, the first statistical NLP methods based on Machine Learning (ML) were developed. The first real adoption of ML in the RE field dates back to a 2007 study by Cleland-Huang et al. [19], in which an approach was presented to automatically identify and classify nonfunctional requirements (NFRs) from requirements specifications. Conventional ML techniques are limited in their ability to process data in their raw form. This means that building these systems requires careful engineering and considerable domain expertise to design a feature extractor that transforms the raw text into an appropriate internal representation, i.e., a feature vector. The most widely used supervised learning algorithms are Naïve Bayes (NB), Support Vector Machine (SVM) and Decision Tree (DT). Unsupervised ones include Latent Dirichlet Allocation (LDA), K-means, Hierarchical Agglomerative Clustering (HAC) [20].

Over the past two decades, with the increased availability of data, such as app reviews and social media interactions, Deep Learning (DL) methods using deep neural networks with multiple layers have emerged in the NLP scene. The central idea of DL is that it allows a machine to be fed a large amount of raw data and automatically discover representations or features needed for detection or classification. In this way, DL requires minimal manual feature engineering. In addition, features learned from DL models are high-level and allow for better generalization even to new and unseen data. Among the most widely used neural networks are Convolutional Neural Networks (CNNs), Feedforward Neural Networks (FNNs) and Recurrent Neural Networks (RNNs).

The architecture of transformers [21] has revolutionized the NLP field because of its ability to handle sequences of data with an attention mechanism to capture long-range relationships. In 2018, Google introduced BERT (Bidirectional Encoder Representations from Transformers) [22], a transformer-based deep learning model pre-trained on huge text corpora, which has achieved cutting-edge results in language comprehension tasks. This model can be specialized through fine-tuning on specific datasets, effectively adapting it to particular needs. Some recent studies highlight the clear superiority of this approach in the field of Requirement Engineering, compared to previous methodologies [23] [24] [25].

Another important innovation was GPT (Generative Pre-trained Transformer), an advanced generative language model developed by OpenAI. Studies have investigated the applicability of ChatGPT, through prompt writing, in the software development cycle, highlighting its potential [26]. It was found that the model stands out for its ability to operate with high autonomy in activities such as requirements analysis, domain modeling, and implementation. However, some significant limitations were identified, including lack of traceability between phases, inconsistencies in the generated artifacts, and a sometimes excessive application of domain knowledge that can lead to unnecessary expansion of requirements. It is also crucial to monitor model "hallucinations," i.e., the generation of inaccurate information, and to carefully evaluate the generated output before its use.

Chapter 3

Methodology

This chapter presents in detail the methodology adopted to implement the project. The proposed approach aims to incorporate artificial intelligence, specifically Deep Learning (DL) and Natural Language Processing (NLP) techniques, to automate the analysis of documents provided by users. Figure 3.1 illustrates a schematic graphical representation of a black-box view.

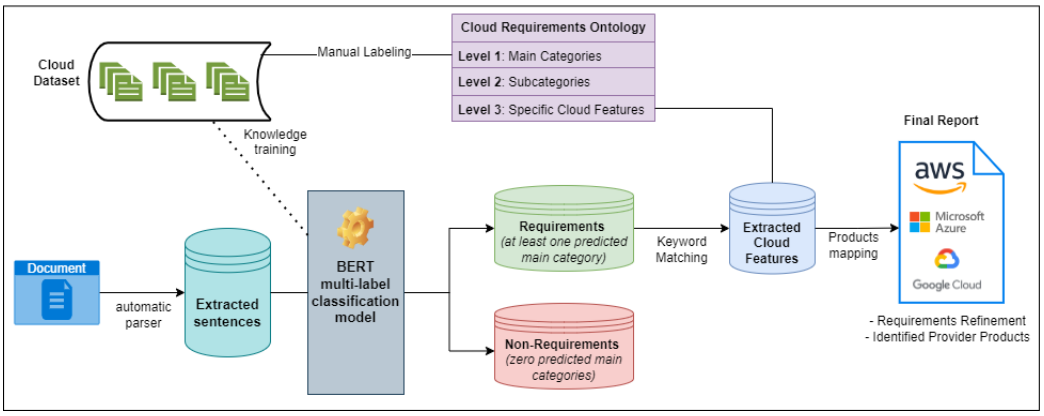

Figure 3.1. Outline of The Methodology With a Black Box View

The core of the project is the development of an ontology that clearly, completely, and in a non-ambiguous manner organizes all potential cloud requirements within a hierarchical structure divided into 3 levels: main categories, subcategories, and specific cloud features.

Users provide a document where they express, in English semi-technical natural language, their goals of the desired cloud migration. After extracting all sentences by parsing the document, the objective is to automate their classification according to the main categories treated. The document may also include sentences containing irrelevant information that, having no practical value, should be discarded.

A requirement sentence could refer to more than one cloud service at the same time. To give an example, the requirement *"Our application requires scalability to handle variable loads efficiently and monitoring capabilities to observe its performance continuously"* encompasses both the computing and management aspects.

For this purpose, we plan to train a multi-label classifier that allows each sentence

to be associated with one or more predefined labels, none if the sentence is not a requirement. After examining several techniques, was chosen the approach that achieved the best results in language comprehension tasks according to the literature: the use of a pre-trained model based on the Transformer architecture, such as BERT (Bidirectional Encoder Representations from Transformers). BERT represents an extremely effective model capable of analyzing text by considering the context from both directions of a word, which significantly improves the ability to understand the meaning of words in various contexts.

Automatic classification of statements into specific categories represents a supervised learning technique. This type of learning allows an algorithm to learn from an already labeled dataset in order to make predictions about new, unlabeled data [27]. To train the classifier, it is therefore essential to provide it with a set of labeled sentences. Due to the lack of a suitable pre-existing data corpus, an ad hoc one was created consisting of sentences collected from various cloud documents. Sentences were manually labeled by referring to the main cloud categories of the ontology.

Once the model has identified relevant sentences by framing them into macro categories, these are analyzed in more depth. Cloud-specific features are extracted through a keyword-oriented NLP approach, assuming that each cloud feature can be expressed in different ways. After that, each one is mapped to the corresponding products offered by major cloud service providers, such as AWS, Google Cloud, and Microsoft Azure.

This procedure outputs a final PDF that contains analysis info, a structured requirements refinement, and a mapping to the various cloud products.

Chapter 4

Ontology for Cloud Computing Concepts

An ontology of cloud concepts is necessary, as it will allow us to decide which cloud concepts to extract from user documents and how to classify them accurately, laying the foundation for a clear and unambiguous requirements specification. It is essential to address the challenge posed by the variety of service descriptions, inconsistent naming conventions and heterogeneity in terms of types and characteristics of cloud services. Therefore, it is necessary to adopt a uniform conceptual model that facilitates the delineation of domain knowledge boundaries.

Ontology is described as "a formal specification of a conceptualization" [28]. This means that it provides a framework for representing a given domain by featuring key concepts in a way that is clear and understandable to both machines and humans. These concepts are typically structured in an organizational gerarchy. An analysis of cloud ontologies in the literature was conducted; however, noting the absence of one suitable for our scenario, it was decided to develop a custom ontology.

4.1 Review of Existing Cloud Ontologies

Initial research focused on the current definition of ontologies in the context of cloud computing. Several surveys, each with advantages and limitations, have highlighted attempts to standardize descriptions of cloud services, with the goal of improving discovery and helping users navigate their way through a wide variety of offerings.

One area of great interest in the cloud is the security, where numerous ontologies have been developed to address compliance issues [29]. However, analyses have revealed that existing ontologies tend to be insufficiently specific and have several weaknesses, including limited coverage of cloud service types (often focused exclusively on IaaS), a lack of definition of non-functional properties such as cloud technical characteristics and Quality of Service (QoS) parameters, as well as the absence of key elements for service level agreement (SLA) specifications.

Some studies have attempted to extend the application of ontologies to other areas of software engineering, such as change control, process support and requirements engineering. Despite the progress made, it is clear that the characteristics of cloud services covered by current ontologies are extremely limited and incomplete.

4.2 Ontology Design

This section discusses the creation of a custom ontology which should be free from certain errors, such as redundancy, inconsistency and missing information.

The goal in developing an ontology for the cloud context is to organize and precisely define the whole concepts related to this domain, ranging from computational hardware resources to security and compliance policies. Such a model is intended to embrace both the functional and non-functional characteristics of the cloud, while maintaining a vendor-independent approach. However, it is essential to ensure that the developed taxonomy accurately represents the current scenario, placing particular emphasis on services offered by Amazon AWS [30], Google Cloud [31] and Microsoft Azure [32].

The defined ontology is structured according to a hierarchical model with three levels of detail: at the top are the main categories identified, which serve as the fundamental pillars. At the second level are the subcategories, which delineate more specific domains within each main category. Finally, for each subcategory, is defined a set of cloud-specific features.

Since the ontology will form the foundation for an upcoming deep learning model dedicated to multi-label classification, it is essential to minimize the number of main categories. In this regard, identifying a limit of six categories has proven to be an optimal balance. Below a general overview is presented for each main category, including a graphical representation:

- **Compute**. This category represents concepts related to computing operations:
  - Resource types such as virtual machines, containerization, serverless computing, dedicated servers. Elasticity concepts like scaling, load balancing.
  - Host provisioning modes, including self-provisioning and OS options.
  - Various types of requirements based on processor, memory, and GPU.

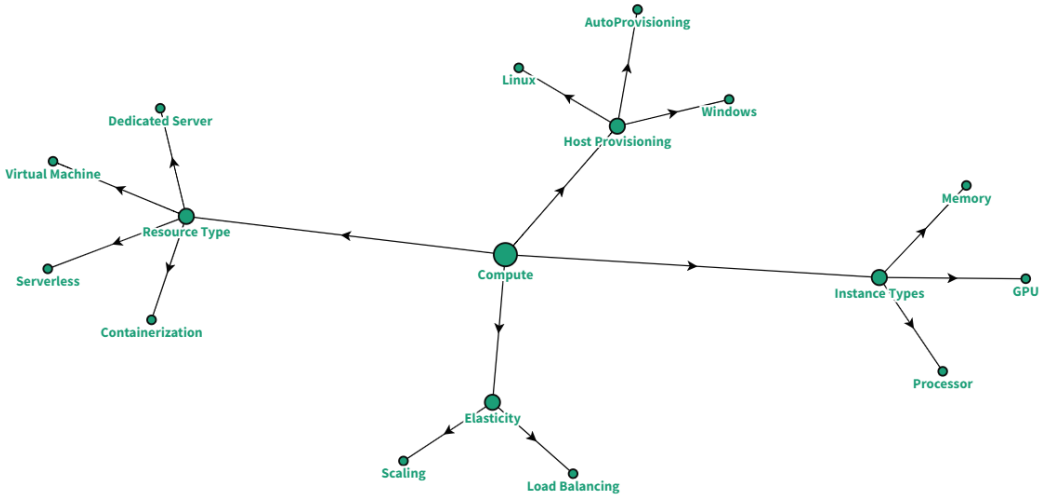

Figure 4.1. Network Graph for Compute Category

- **Data Handling**. Focuses on data management and optimization through:
  - Data storage solutions.
  - Database systems, both relational and NoSQL.
  - Data optimization techniques such as disk utilization and caching.

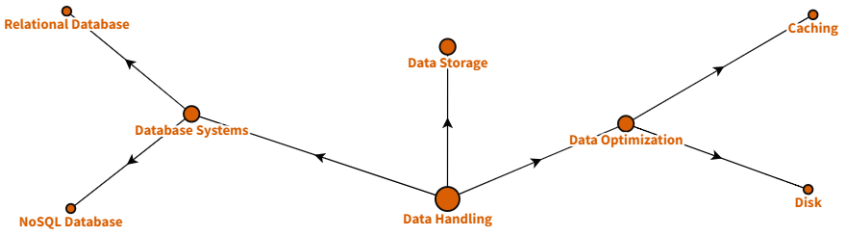

Figure 4.2. Network Graph for Data Handling Category

- **Network**. Addresses essential network infrastructure and services:
  - Local area network and WAN infrastructures.
  - Crucial protocols including TCP/IP, security protocols, and web protocols.
  - Supporting services such as DNS, CDN, and firewalls.
  - Network metrics such as bandwidth, latency, and throughput.

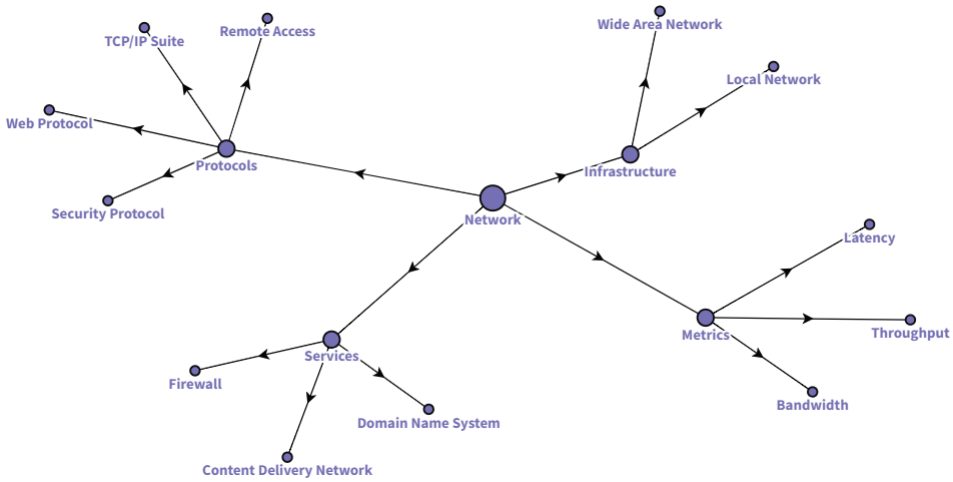

Figure 4.3. Network Graph for Network Category

- **Security & Compliance**. Covers critical aspects of security and regulations:
  - Identity and access management, multifactor and single sign-on.
  - Data protection, encryption, and maintaining confidentiality.
  - Regulatory compliance, threat management, phishing, vulnerability assessment, threat detection, and protection from cyber-attacks.

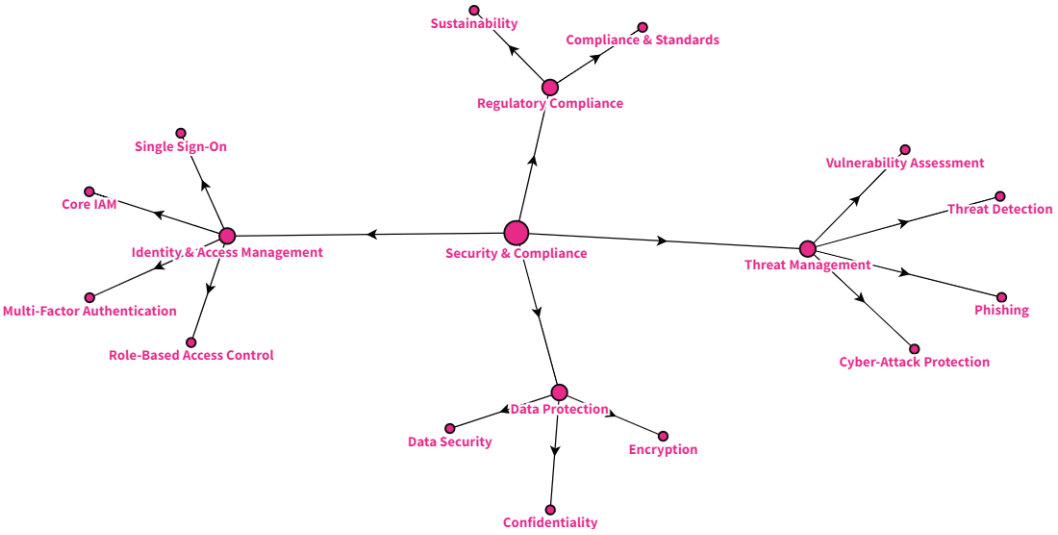

Figure 4.4. Network Graph for Security & Compliance Category

- **Management & Monitoring**. Emphasis on supervision and optimization of cloud resources:
  - Resources monitoring, dashboards, log management, audit tracking, alerts, and reporting.
  - AI-driven insights for analytics, performance optimization, and data warehousing.
  - Support services including helpdesk, maintenance, and data migration.
  - Financial management with focus on cost optimization and budgeting.

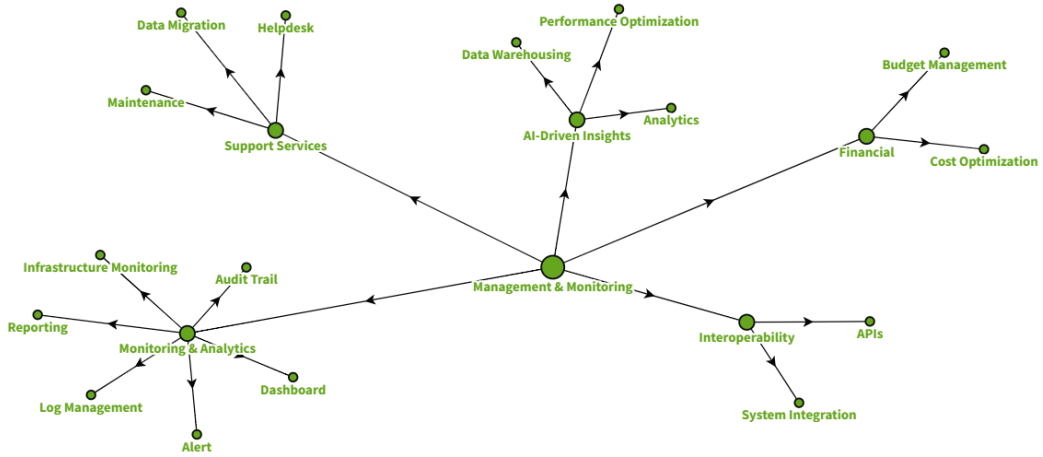

Figure 4.5. Network Graph for Management & Monitoring Category

- **Cloud Service Essentials**. Fundamentals for effective management of cloud services:
  - Performance management including high availability, downtime management, and response times.
  - Infrastructure management with emphasis on system isolation, multi-region, redundancy and replication, and data center management.
  - Backup and recovery strategies.
  - Cloud models (public, private, hybrid).

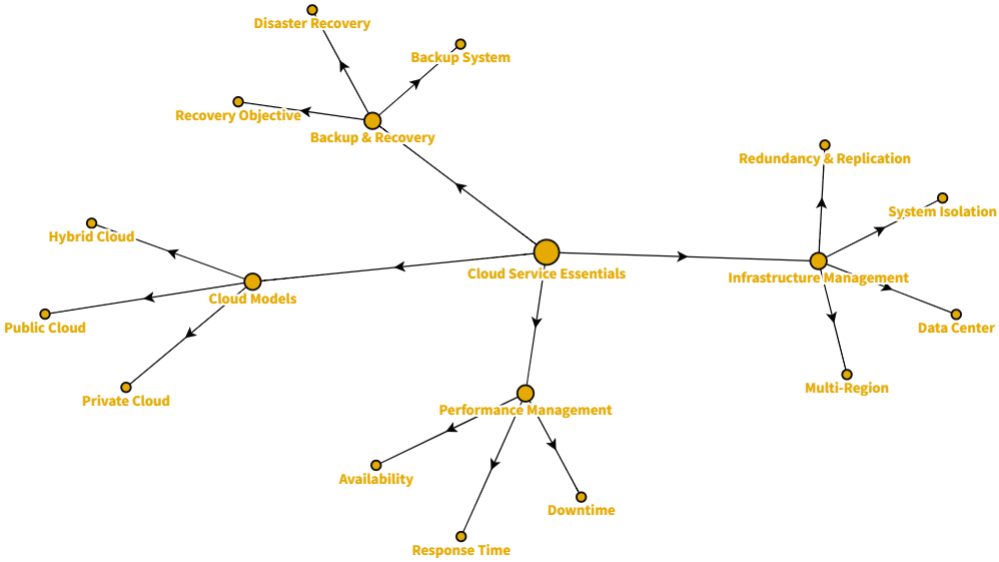

Figure 4.6. Network Graph for Cloud Service Essentials Category

Chapter 5

Data Collection Procedure

In this chapter, is described in detail the process that led to the creation of a custom dataset, needed for the fine-tuning of the BERT model. Is discussed in depth the methodology adopted for its construction, preprocessing and exploration, starting with the analysis of open source data corpus in the field of Requirements Engineering.

5.1 Review of Existing Datasets

After conducting a detailed research, it was revealed that most studies feed their models through the collection of own specific documents, thus creating a customized set of data tailored to their needs. In some studies in the RE field some corpora are used that have been made open source, among them are:

- The PROMISE Repository [33], published in 2005, which includes 625 labeled requirements, both functional and non-functional.
- The DePaul Corpus [19], presented in 2007, featuring 358 functional requirements (FR) and 326 non-functional requirements (NFR), classified into 10 categories and derived from 15 different requirements specifications, developed by DePaul University students.
- The SecReq Dataset [34], introduced in 2011 to support the analysis of security requirements, which includes 511 requirements from three separate projects, divided into security-related and non-security-related requirements.
- The PURE Dataset [35], made available in 2017, encapsulates about 35,000 unlabeled sentences extracted from 79 requirements documents in raw form. These requirements specifications are of different domains, have various levels of abstraction, and range from product standards to public corporate documents to academic projects. The authors offered the dataset for a wide range of uses. One particular study [23] manually labeled a subset of 7,745 sentences for fine-tuning BERT for requirements identification from unstructured documents, making the dataset available as an open source resource.

Despite the existence of few public and open source datasets in the field of software requirements, none of them looks adequate for our specific needs related to cloud requirements. This highlights the need to build a customized dataset.

5.2 Constructing the Dataset

The goal is to collect data and build a custom dataset that includes representative statements of both cloud requirements and non-requirements. To this end, public documents are consulted and the use of generative AI is explored to address the absence of provided document samples. Each sentence is analyzed to determine which cloud labels to assign, performing manual labeling for higher quality.

Public Documents Collection. First, a search was conducted for public documents about cloud computing available on the Internet. Typically, organizations or government agencies seek to acquire technology solutions by outlining the requirements needed in a public document called *Request for Proposal (RFP)*. In addition, tender documents may be used, which are a formal invitation to various potential suppliers or contractors to submit bids. These documents are written in natural language and can span hundreds of pages. They detail technical specifications, contractual commitments, and may provide additional background information. The entity issuing the solicitation will select the most appropriate bid for the development of the required solution. It is common for these large documents to contain both parts outlining cloud requirements and other purely informative sections. These documents are generally intended for public distribution and do not contain sensitive personal information.

For each collected document, were identified sections in which details of the required cloud infrastructure requirements are provided. The 15 public documents reviewed are shown in Table 5.1. For each document, the following are given: the unique ID to simplify traceability throughout the project, the source from which it was obtained and, finally, details regarding the sections, paragraphs and exact pages from which the sentences were extracted.

Table 5.1. Collected Public RFP Documents

ID	SOURCE	SECTIONS
RFP_1	nrcassam.nic.in	paragraph 5 (page 14-22)
RFP_2	bankofindia.co.in	section 3.2 (pages 13-34)
RFP_3	biosaline.org	section II (from page 3)
RFP_4	ibm.gov.in	section 5 (pages 8-16)
RFP_5	dgshipping.gov.in	Vol1: par.1.3, 1.4.3
RFP_6	sebi.gov.in	appendix VI-VII (pages 101-117)
RFP_7	meity.gov.in	section 5 (page 11-44)
RFP_8	odisha.gov.in	section 4.3 (pages 9-27)
RFP_9	oecm.ca	part 2 (pages 9-18)
RFP_10	gil.gujarat.gov.in	section 2.1, 2.2 & 4.1
RFP_11	cgtmse.in	par. 4.3 (pages 17-28)
RFP_12	inflibnet.ac.in	Annexure-T-II (pages 18-24)
RFP_13	acgov.org	section I (pages 5-11)
RFP_14	aptransport.org	pages 14-21
RFP_15	agriseta.co.za	pages 2-7

Initially, an attempt was made to extract sentences through the use of some automatic parsing and extraction engines. However, this strategy was abandoned due to the poor quality of the extracted texts and the structural complexity of the parsed documents. Therefore, manual processing of the documents was opted in order to preserve the original syntax and semantics in the extracted texts. Nevertheless, several challenges emerged that necessitated the adoption of certain conventions. Given the various sentence lengths and structures, it was chosen to treat each sentence as a separate data unit. Therefore, if a requirement consists of multiple sentences, it is segmented into multiple units. Sometimes requirements are written in the form of a list with many elements. In this case, the list is transformed into a single sentence separating the various elements with commas. In some special cases, such lists have been divided into multiple sentences.

Generating Documents For Our Scenario. Since no sample documents were provided, a generative AI technology, such as GPT, was used to fill this gap to try to reproduce the given scenario. A document writing prompt was devised, providing as background the fact that these are written by non-technical users who are defining their goals for migrating their IT infrastructure to the cloud. Using this trick, seven small different texts were automatically generated from which the sentences were extracted. This improves the representativeness of the dataset and allows us to train the model with instances that accurately reflect the described reference scenario.

Manual Labeling. Each sentence in the dataset was manually analyzed and classified according to the main cloud categories treated. As it was discussed in Chapter 4, these possible categories are: 'compute', 'data handling', 'network', 'security & compliance', 'management & monitoring', 'cloud service essentials'.

This process was done using the one-hot encoding technique, which consists of assigning a value of '1' if a statement belongs to a particular category, and '0' otherwise. The manual approach to labeling, although laborious and time-consuming, was preferred to ensure a high level of quality, avoiding the use of automatic parsing tools that could introduce errors. To mitigate risks such as subjectivity in label assignment and possible inaccuracies, the process was carried out by two annotators, ensuring more reliable results. In case of disagreement between annotators on a specific label, the text sample in question was excluded from the dataset to preserve integrity and consistency.

5.3 Dataset Preprocessing

Having done a careful manual labeling, the dataset is already considerably refined and did not need much preprocessing. However, the following actions were taken:

- **Harmonization of terminology**: considering that many statements are derived from RFP documents, the term "bidder" recurs frequently. In our application context, however, this term would be inappropriate, so it was replaced with "provider" to ensure greater alignment with our context. This serves to optimize the understanding and predictive accuracy of the model.

- **Cleanup of imperfections**: some statements in the dataset have imperfections. To resolve, unnecessary whitespace and end periods were removed, thus ensuring that there are no formatting disharmonies. This prevents possible negative impacts on model performance.
- **Removal of duplicates**: 103 duplicate statements were removed, without distinguishing between uppercase and lowercase. For each duplicate, the first occurrence was retained. The presence of duplicates can alter the training process and lead to overfitting, as the model can "learn" the same example multiple times. Interestingly, some sentences were found to be identical across different RFPs, this is a first evidence of potential language standardization.

5.4 Dataset Exploration

In this section is carried out an exploration of the resulting clean dataset, containing 2164 ennuples, to understand its characteristics. Its columns are as follows:

- **source**: The public document source from which the sentence was extracted
- **description**: Textual sentence
- **goal**: Binary variable indicating whether the statement is a cloud requirement
- **compute**: Indicates the presence (1) or absence (0) of compute-related content
- **network**: Indicates the presence (1) or absence (0) of network-related content
- **data handling**: Indicates the presence (1) or absence (0) of content related to data handling
- **security & compliance**: Indicates the presence (1) or absence (0) of content related to security & compliance
- **management & monitoring**: Indicates the presence (1) or absence (0) of content related to management & monitoring
- **cloud service essentials**: Indicates the presence (1) or absence (0) of content related to cloud service essentials

The pie chart in Figure 5.1 shows that 68.1% of the sentences (1474 records) were labeled as cloud requirements, while 31.9% (690 records) were labeled as non-requirements, indicating a greater presence of sentences labeled with at least one cloud category.

The bar plot in Figure 5.2 shows in more detail the distribution of requirements (green bars) and non-requirements (red bars) broken down by source. Significant variability in the proportion among them is illustrated, also reflecting the varying magnitude of the number of extracted sentences of the documents, some with a large number and others somewhat smaller.

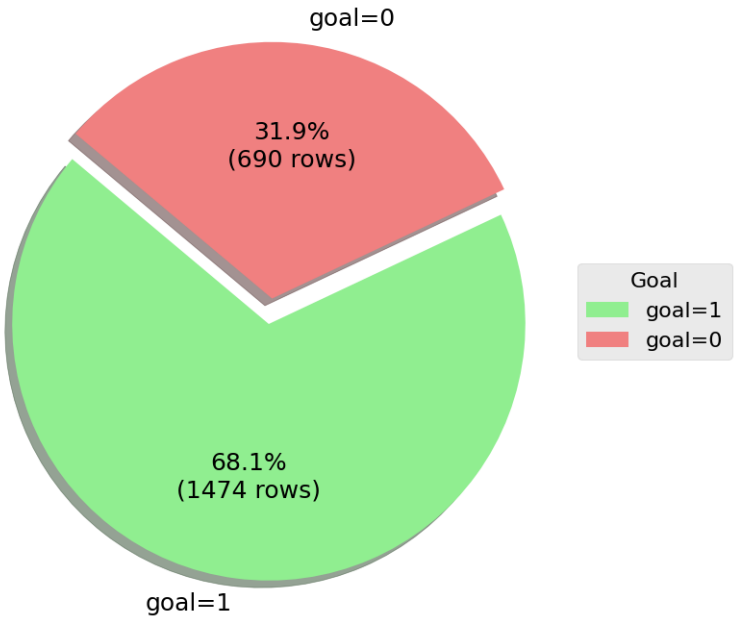

Figure 5.1. Requirements Distribution

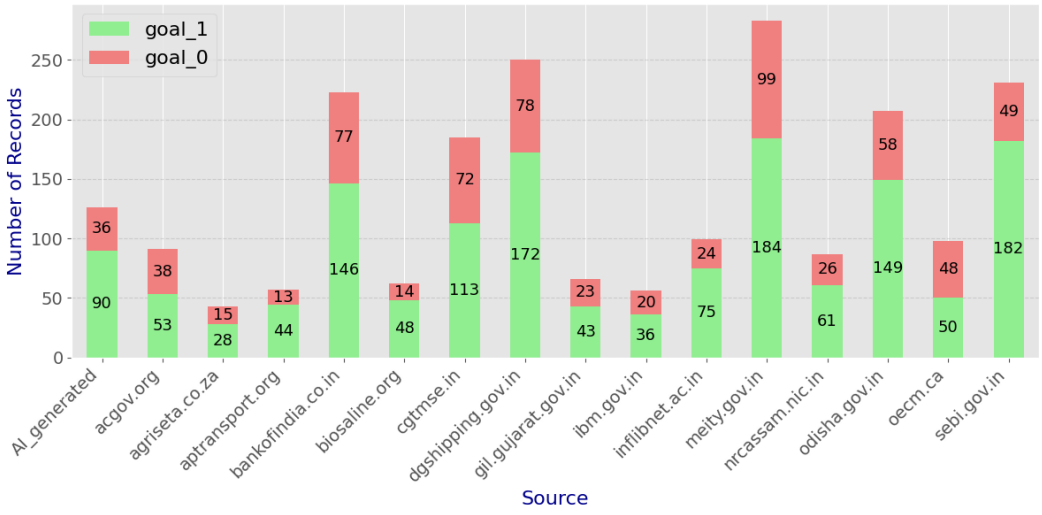

Figure 5.2. Distribution of Requirements By Sources

The donut chart presented in Figure 5.3 illustrates the geographical distribution of sentences extracted from the collected documents. India predominates, accounting for 78% of the total with 1,687 sentences. International organizations follow with a 9.7%, equivalent to 210 sentences. Sentences generated through artificial intelligence make up 5.8% of the sample, with a total of 126 sentences. Finally, Canada and South Africa complete with 4.5% (98 sentences) and 2% (43 sentences), respectively. The fact that most of the sentences come from Indian documents is not an obstacle, since the field of cloud computing is characterized by a very standardized language. Such lexical uniformity means that regional differences do not alter the homogeneity of the core content. In addition, the BERT model's ability to generalize ensures that our methodology is valid and applicable across the board.

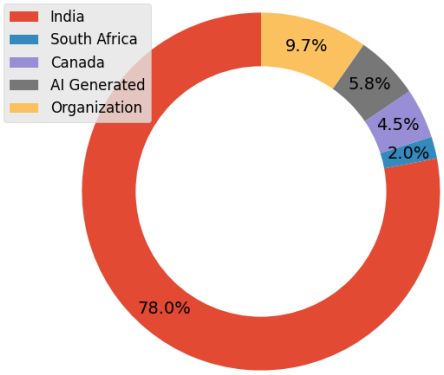

Figure 5.3. Geographical Distribution of Sentences

The horizontal bar plot in Figure 5.4 illustrates the frequency of the 6 cloud categories. While the frequency range shows some variation, it does not show extreme imbalance. "Management & monitoring" is the most frequent aspect with mentions in 553 sentences, followed by "security & compliance" with 364. "Cloud service essentials" is mentioned 341 times, while the "compute" aspect records 271 counts. "Network" and "data handling" respectively count 218 and 182 mentions.

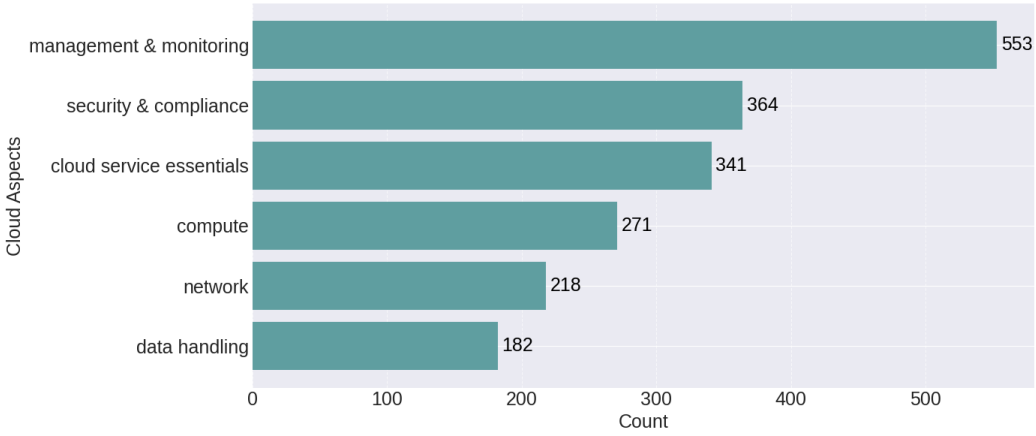

Figure 5.4. Distribution of Cloud Categories

Figure 5.5 displays a correlation matrix illustrating relations between the various categories. Although there are fairly low values, it can be seen that the categories corresponding to physical infrastructure requirements, "compute", "data handling" and "network" show some mutual correlation, suggesting a tendency of sentences to have these labels simultaneously.

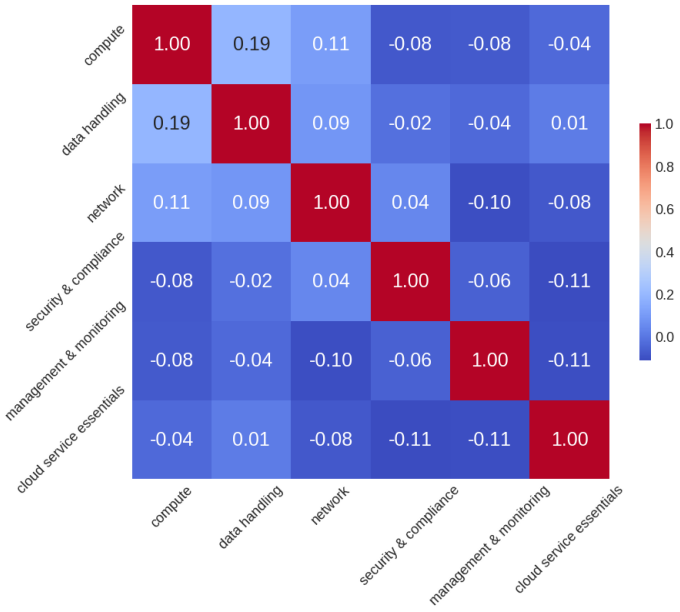

Figure 5.5. Correlation Matrix of Cloud Categories

In Figure 5.6, is shown the distribution of sentence lengths in the dataset. A balanced distribution, e.g., following a bell shape, is preferable as it suggests a fair variety in the length, without imbalances. The analysis shows that most of the sentences are in a length range of 0-50 words, with a maximum concentration around 15 to 20 words. The plot does not show the presence of any particular outliers. The average sentence length is 22 words, while the shortest sentence recorded is "24x7 technical support" which is only 3 words long. The maximum one, consisting of 174 words, is a bulleted list turned into a single sentence.

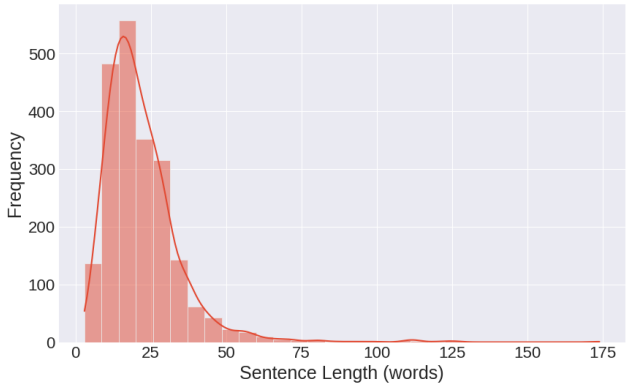

Figure 5.6. Distribution of Sentence Lengths

Chapter 6

Fine-Tuning BERT for Classifying Sentences

This chapter details the process for developing a classifier that assigns sentences to the main categories of the ontology. Considering the possibility for a single sentence to belong to several categories simultaneously, a multi-label classification approach was chosen. Here we discuss the technical basis of BERT, the experimental setup adopted, the classifier training process and, finally, the performance evaluation.

6.1 Technical Background

As is illustrated in Figure 6.1, the BERT model is trained in two stages and then evaluated by comparing its predictions with our ground truth. This section reviews the first two steps.

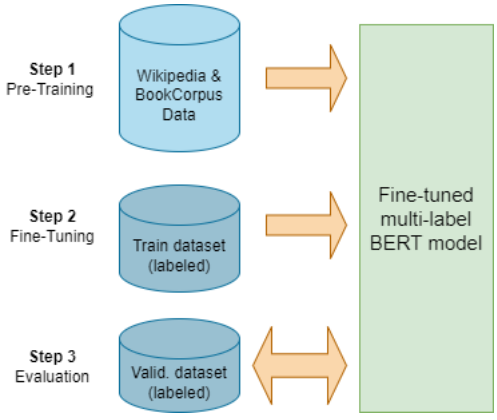

Figure 6.1. Training Process

Step 1: Pre-training. It refers to the process of initializing a model with pre-existing knowledge, using a large set of raw texts including the BookCorpus and English-language Wikipedia. There are many pre-trained ready-to-use models such as, roBERTa, alBERT and distilBERT. For this study was chosen BERT-Base

because it represents an optimal balance between size and performance. It is characterized by 12 transform hidden layers; each one has 768 units, which determine the size of the feature vector, and 12 attention heads, which enable it to collect information from different text positions simultaneously. Altogether, the chosen model reaches 110 million parameters.

During pretraining, the model learnt to understand the structure of natural language through two main techniques:

- **Masked Language Model (MLM)**: some words in a sentence are replaced with the special token [MASK] and the model's task is to predict the original words based on the context provided by the other words in the sentence. This approach allows BERT to learn a bidirectional context, that is, the context on both the left and right of each "masked" word.
- **Next Sentence Prediction (NSP)**: BERT is trained to predict whether a sentence B is a logical sequel to a sentence A by teaching the model to understand the relationship between the sentences.

An analysis, illustrated in Figure 6.2, was conducted to decide whether to opt for the "cased" or "uncased" configuration. Acronyms such as "VM", "DC", "DR", "VPN" and "RAM" were identified, but in other cases it might be ineffective to distinguish words such as "Service", "DATA", "Security", etc. Therefore, we opted for the "uncased" configuration, which treats text by converting it to lower case before processing, thus making it case-independent. This choice was guided by the belief that a case-insensitive model would better suit the needs of the project, allowing us to focus more on the semantic understanding of sentences than on their spelling.

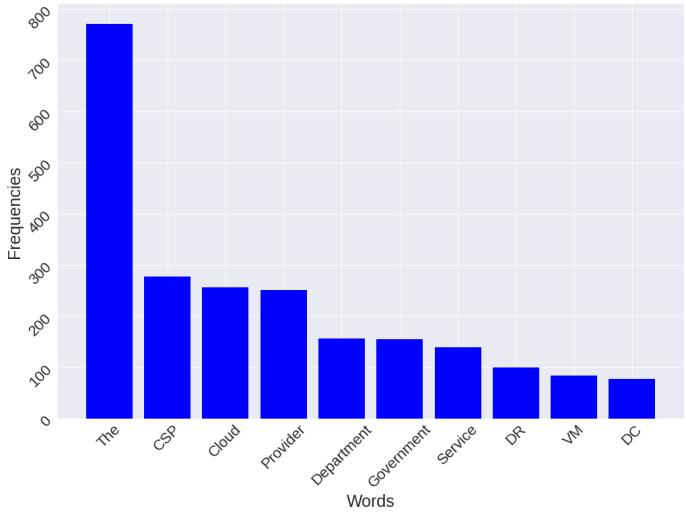

Figure 6.2. Uppercase Word Analysis

Step 2: Fine-Tuning. BERT can be fine-tuned on specific NLP tasks, which in our case is text classification, through the use of a labeled dataset. At this stage, the parameters of the pre-trained model are adjusted to better fit the specific task. In

our scenario, the model should be trained to predict the presence or absence of each label for each instance. This process requires advanced hardware to perform the intensive computations efficiently. I experimented with using my local NVIDIA GPU, but without achieving the hoped-for performance. As a result, it was decided to turn to Google Colab, which provides its Tesla T4 GPUs with CUDA, a graphics card that significantly speeds up model training. It is important to note, however, that these computational resources are subject to limitations in terms of usage duration and memory capacity.

6.2 Experimental Setup for Fine-Tuning

This section examines the preparation and optimization process for fine-tuning the BERT-Base uncased model for multi-label classification. The tokenizer, the libraries used, the hyperparameters and the partitioning of the dataset are described in detail.

Tokenizer. Tokenization is a crucial step in preparing the raw text so that it can be effectively interpreted and used by the neural network. BERT uses the *WordPiece* tokenizer, which fragments the text into smaller elements called tokens. This set also includes special tokens: the [CLS] classification token that is inserted at the beginning of each input, the [SEP] separator token that is used to signal its conclusion, and the [PAD] token for padding. BERT provides a system for handling terms not in its vocabulary, referred to as Out-Of-Vocabulary (OOV). This system allows for the decomposition of unknown words into smaller segments for which the model has already developed contextual representations, or embeddings. To give a practical example, also shown in Listing 6.1, the model does not know the word "Scalability". BERT divides it into elementary tokens that it knows, such as "scala" and "##bility". This approach allows the model to recognize and interpret various forms of root words, expanding its ability to understand language beyond the limits of the original vocabulary. Eventually, the encoding process is performed, where each identified token is encoded by mapping to a unique numeric ID, corresponding to an index in the model's vocabulary. This vocabulary, which has a size of 30,522 tokens, represents the list of all tokens recognized by the model.

```
sentence = "Scalability is key as well"

tokens = tokenizer.tokenize(sentence)

encoded_input = tokenizer.encode_plus(
    sentence,
    add_special_tokens=True)

print("Tokens:", tokens)
print("Token IDs:", encoded_input['input_ids'])

# Outputs
# Tokens: [CLS,'scala', '##bility', 'is', 'key', 'as', 'well', SEP]
# Token IDs: [101, 26743, 8553, 2003, 3145, 2004, 2092, 102]
```

Listing 6.1. Tokenization of a Sentence

Libraries. The fine-tuning was implemented in a Google Colab notebook, an interactive development environment that allows Python code to be written and executed in the browser. The libraries used are the following:

- **Transformers** [36]: A library developed by Hugging Face which is used to access *bert-base-uncased* pretrained model and its tokenizer.
- **PyTorch** [37]: Machine learning library that facilitates the training and definition of neural networks using GPU support. Used to create and train the neural network model, manage data through DataLoader and compute the gradient during the backpropagation phase.
- **NumPy** [38]: Essential for scientific computation in Python. It is used to handle multidimensional arrays and matrices, as well as provide mathematical functions for complex operations.
- **Pandas** [39]: Provides high-performance, easy-to-use data structures and data manipulation tools.
- **Scikit-learn** [40]: Provides tools for data analysis and machine learning algorithms. It is used for dividing the dataset into training set and validation set, as well as for computing evaluation metrics.

Hyperparameters. These are parameters set before training begins that regulate the training process of the model, influencing its ability to learn from the training data. Several experiments were conducted in order to find the optimal configuration:

- **Batch Size = 16**. The number of training examples processed together in a single training iteration. The chosen value provides a good balance between the model's generalization capability and computational efficiency.
- **Learning Rate = 3e-05**. The rate at which the model updates its weights during training. This value ensures that the model fits the training data effectively, minimizing the loss function without incurring too abrupt adjustments that could lead to instability in the learning process.
- **Epochs = 5**. The number of times the entire training dataset is passed through the model. This amount was sufficient to allow the model to adequately learn from the data without suffering from overfitting.
- **Max Sequence Length = 300 tokens**. The maximum token length of the input sentence. If the tokenized sentence exceeds this value, it will be truncated. BERT-Base can handle sentences up to 512 tokens, but 300 was chosen to save memory and because sentences in the dataset do not exceed it.
- **Weight Decay = 1e-6**. A regularization term added to prevent overfitting by reducing the size of the model weights during training.

Train and Validation Sets. The dataset was split into two distinct sets: one dedicated to training (training set) and the other to validation (validation set). This split sees 85% of the data assigned to the former, corresponding to 1839 records, and 15% to the latter, corresponding to 325 records. Before, it was implemented a shuffle of the records to avoid any potential bias related to the order in which the documents were presented, thus ensuring that the model does not learn specific sequences. The number 42 was selected as the random split seed to guarantee reproducibility, adhering to a common scientific convention and its cultural significance as "the answer to life, the universe, and everything" from Douglas Adams' "The Hitchhiker's Guide to the Galaxy". During the split, care was taken to represent the six target labels in both sets, though achieving a perfectly uniform distribution proved challenging.

6.3 Training Implementation

Before proceeding with training, the architecture of the neural network is defined. The corresponding code fragment is shown in listing 6.2.

```
1  class BERTClass(torch.nn.Module):
2      def __init__(self):
3          super(BERTClass, self).__init__()
4          self.bert_model = BertModel.from_pretrained('bert-base-
       uncased', return_dict=True)
5          self.dropout = torch.nn.Dropout(0.3)
6          self.linear = torch.nn.Linear(768, 6)
7
8      def forward(self, input_ids, attn_mask, token_type_ids):
9          output = self.bert_model(
10             input_ids,
11             attention_mask=attn_mask,
12             token_type_ids=token_type_ids
13         )
14         output_dropout = self.dropout(output.pooler_output)
15         output = self.linear(output_dropout)
16         return output
```

Listing 6.2. Neural Network Architecture

Initially, the pre-trained model is loaded, to which two layers are added:

- **Dropout Layer**: is a regularization technique used to prevent overfitting by randomly disabling some units (neurons) in the network during each training iteration. There is a 30% probability that the output of each node in this layer will be set to zero. The model must then find ways to adapt and continue to perform well even when some of its input is missing.
- **Linear Layer**: performs a linear transformation by mapping a 768-dimensional feature vector to a 6-dimensional one, corresponding to the 6 labels, using the formula $y = xA^T + b$. Here, x is the input, A is the matrix of weights with size 768×6, b is the bias vector of size 6, and y is the transformed output.

In the forward method, is defined how the model processes the input and produces the output. The output goes through the dropout level to reduce the risk of overfitting and then, in the linear layer, is transformed in a classification vector of dimension 6.

To define the loss function, which calculates the error between model predictions and true labels during training, BCEWithLogitsLoss is used. It is a common choice for multi-label classification problems, where each label is treated independently as a binary problem. It consists of an efficient combination in a single class of a sigmoid function, used to compress the model output into an interval between 0 and 1, and the Binary Cross Entropy (BCE). The optimizer, Adam, is used to update the neural network weights and improve performance.

```
def train_model(n_epochs, training_loader, validation_loader, model,
    optimizer, best_model_path, device):
    valid_loss_min = np.Inf  # Tracker for validation loss

    for epoch in range(1, n_epochs+1):
        model.train()  # Training
        for batch_idx, data in enumerate(training_loader):
            ...
            # Forward pass
            outputs = model(ids, mask, token_type_ids)
            loss = loss_fn(outputs, targets)
            # Backward pass and optimization
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            train_loss += loss.item()
        train_loss /= len(training_loader)

        model.eval() # Validation
        with torch.no_grad():
            for batch_idx, data in enumerate(validation_loader):
                ...
                outputs = model(ids, mask, token_type_ids)
                loss = loss_fn(outputs, targets)
                valid_loss += loss.item()
        valid_loss /= len(validation_loader)

        # Save model if loss diminished
        if valid_loss <= valid_loss_min:
            torch.save(model.state_dict(), best_model_path)
            valid_loss_min = valid_loss
        elif valid_loss > valid_loss_min:
            print("Early stopping")
            break
    return model
```

Listing 6.3. Portion of the Custom Training Cycle Code

Listing 6.3 illustrates the code cell implemented to perform the training, which consists of a custom loop. For each epoch, the model goes through a training phase followed by a validation phase. During training, is performed a forward step to obtain the predictions and to compute the loss between the predictions and the ground truth, and then perform updating of the model weights by back-propagation and optimization. Then, in the validation step, the model is evaluated on the validation set to check its ability to generalize, without updating its weights. The calculated validation loss provides a measure of the model's effectiveness on data not

seen during training. Tracking training and validation loss for each epoch allows the model's progress to be monitored. Figure 6.3 shows the plot of losses over the epochs. If the validation loss does not decrease further, the early stopping mechanism is triggered to prevent overfitting, stopping training and ensuring that the model is well generalizable and without overfitting.

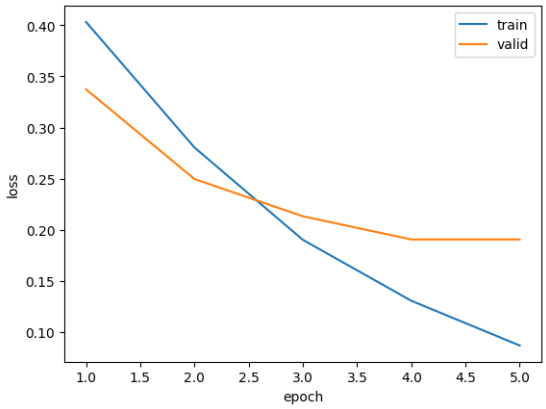

Figure 6.3. Variation of Training Loss and Validation Loss

6.4 Model Evaluation

For each analyzed sentence, the output of the model can be viewed as a probability distribution among six independent labels. To determine which categories to assign to a specific sentence, a threshold value of *0.20* is adopted. Categories above this threshold are considered relevant. This particular value was selected because it allows the model to identify relevant labels without being overly restrictive.

In the evaluation phase with the validation set, the focus is on a twofold empirical assessment of the model's performance. It begins by testing the efficiency of the model in separating requirements from non-requirements, defining as a requirement any sentence that the model classifies under at least one category. Next, it is of interest to evaluate the model's ability to assign the correct categories to the identified requirements. For this purpose, standard evaluation metrics for classification are applied, including:

- **Accuracy**: is the ratio of correct predictions to the total number of predictions. The formula is:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}$$

- **Precision**: is the ratio of correct positive predictions to the total number of positive predictions. The formula is:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

- **Recall (also known as Sensitivity)**: quantifies the number of correct positive predictions among all possible positive predictions. The formula is:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

- **F1-score**: is the harmonic mean of precision and recall. The formula is:

$$\text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

According to Berry [41], automated solutions for RE tasks should ideally have a 100% recall rate; We therefore try to maximize this parameter to minimize the risk of missed requirements. At the same time, we also seek acceptable precision to ensure that we are not overwhelmed by false positives.

Evaluation Binary Classification. In the binary classification of sentences between non-requirements and requirements, the model demonstrated high performance, achieving 88% accuracy. Table 6.1 shows the details of the results, including precision, recall and F1-score. The model's high competence in identifying requirements is evident, evidenced by a recall of 96%. On the other hand, the ability to recognize non-requirements exhibits a lower recall, a phenomenon attributable to the model's tendency to wrongly assign additional categories to sentences. However, this behavior is considered an acceptable trade-off.

Table 6.1. Binary Classification Report

Class	Precision	Recall	F1-Score
Non-Requirement	0.90	0.71	0.79
Requirement	0.88	0.96	0.92

Figure 6.4 exposes the confusion matrix, which provides a graphic illustration of the model's efficiency. This matrix highlights the count of True Negatives (TN), False Positives (FP), False Negatives (FN) and True Positives (TP), which are essential for determining key metrics such as precision and recall. In particular, False Negatives, which is a critical aspect, appears to be significantly minimized: just 8 requirements were not identified.

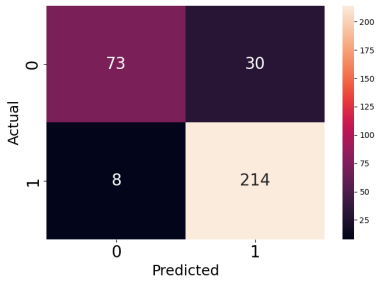

Figure 6.4. Confusion Matrix For Binary Classification

Evaluation Multi-Label Classification. Table 6.2 shows details of the model's performance on specific metrics such as precision, recall, and F1 score for each category analyzed. The performance recorded is high overall, underscoring the model's ability to recognize relevant instances while maintaining acceptable levels of precision in predictions. Through a macro average score of the results, which gives equal weight to each class regardless of its frequency, we show an average precision of 70%, a recall of 86% and an F1 score of 76%. For a more detailed view, the confusion matrices for each category are available in Figure 6.5.

Table 6.2. Multi-Label Classification Report

Category	Precision	Recall	F1-Score
Compute	0.60	0.86	0.70
Data Handling	0.64	0.90	0.75
Network	0.63	0.84	0.72
Security & Compliance	0.79	0.89	0.84
Management & Monitoring	0.86	0.79	0.82
Cloud Service Essentials	0.65	0.87	0.74
Macro Avg	**0.70**	**0.86**	**0.76**

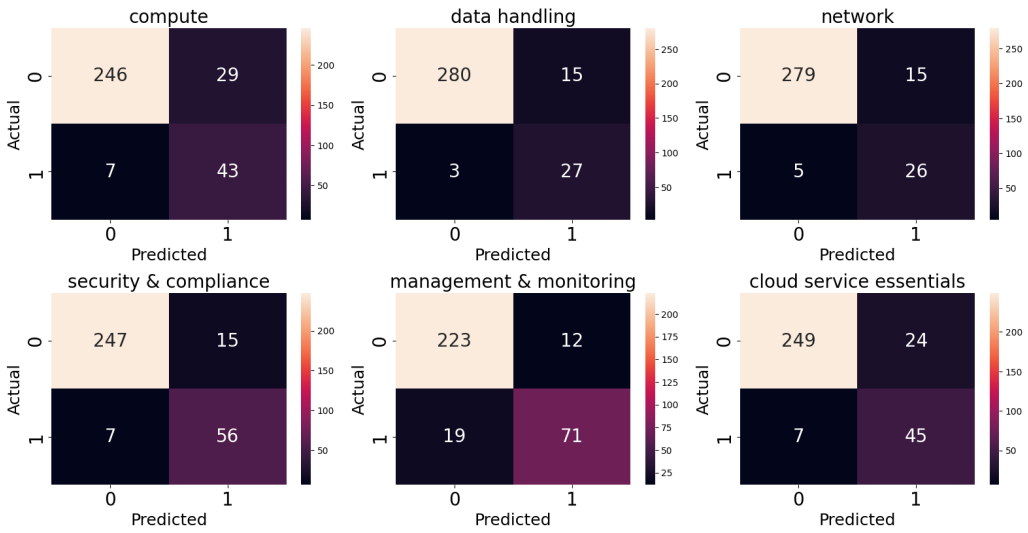

Figure 6.5. Confusion Matrix For Each Cloud Category

Receiver Operating Characteristic (ROC) curve analysis was applied to each label, with the corresponding plots presented in Figure 6.6, which also includes the overall average. The ROC curve represents an effective evaluation method that reflects the performance of a classification model at various levels of decision thresholds. In that plot, the True Positive Rate is placed on the vertical axis while the False Positive Rate is placed on the horizontal axis. Within the plot, the dashed diagonal line symbolizes the result of a classifier operating randomly, comparable to a coin flip. The goal is for the model curve to lie significantly above this line, coming as close as possible to the upper left corner, thus indicating high model

efficiency. The Area Under the Curve (AUC) is an indicator that summarizes in a single value the ability of the model to classify over all possible threshold levels. An average AUC value of 0.96 highlights an outstanding ability of the model to correctly discriminate positive instances, minimizing the frequency of errors.

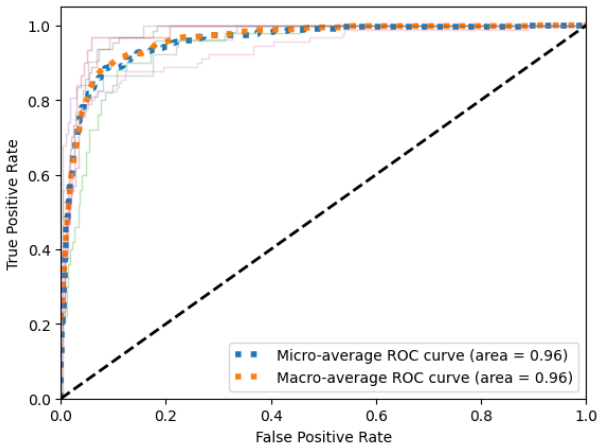

Figure 6.6. ROC Curve

Chapter 7

Methodology Validation

This chapter explains in detail the methodology adopted, tracing the different steps involved, and applies it to a concrete case study. A public handbook, intentionally excluded from the dataset, is used for this test. This document was found to be particularly suitable for our scenario, as it is a useful guide and support for public sector authorities in purchasing cloud services, providing examples of requirements.

7.1 Parsing the User-Supplied Document

In the first phase, the goal is to identify and isolate the sentences in the user-provided document, which consists of free text and with paragraphs, if any.

The proposed approach begins with full-text extraction, a process that transforms the content into a long string of text that includes all sentences. Next, the focus shifts to breaking down this text body into individual sentences, an operation known in the NLP field as Sentence Segmentation.

The core of this phase is Sentence Boundary Detection (SBD), which is the identification of the correct boundaries of each sentence. A first idea might be to apply a manual strategy, trying to divide chunks of text by looking for punctuation marks. However, relying exclusively on such rule-based methods often leads to unsuccessful results, given the frequent presence of exceptions and ambiguities, such as dots used in abbreviations or numerical expressions.

One possible alternative is to use pre-trained models. In the project we make use of SpaCy [42], a Python library specializing in NLP that is widely recognized in industry. SpaCy offers pre-trained models that include an advanced sentence segmenter. This model uses machine learning to predict tokens that serve as sentence boundaries, based on large-scale annotated text corpora in which sentence boundaries have been explicitly flagged. Using this approach, the model is able to identify typical patterns that indicate sentence boundaries, such as the presence of periods, question marks, and exclamation points, and to analyze the surrounding context.

In the test document, cloud requirements are presented in tabular form, sometimes accompanied by comments. A total of 173 sentences were extracted and grouped to conduct the analysis. Some examples are shown in Table 7.1.

Table 7.1. Sample sentences extracted from the test document

Sentence
Provide network firewalls and web application firewall capabilities to create private networks, and control access to instances
The CISP must be certified ISO 27001
It is essential that a minimum of certifications are met
Demonstrate the ability to encrypt data in transit
The customer must be able to build or run applications in compliance with GDPR
The CISP must offer tools to alert the customer whenever a cost threshold is surpassed
Demonstrate the auto-recovery of an instance / set of instances following a failed health check
Demonstrate the automatic scaling capabilities of an application deployed behind a load balancer in a virtual compute environment
Demonstrate the ability to scale both capacity and throughput of block storage without interruption to the workload
Demonstrate a multi-region deployment of a web application including a globally replicated database
Demonstrate the ability to host containerbased workloads
Demonstrate your ability to scale from 10 to 100,000 concurrent users of the web application through dynamic auto-scaling
The CISP must be adherent to Climate Neutral Data Centre Pact
Provide the capability to implement a defence in depth strategy and thwart DDoS attacks
Demonstrate your ability to block malicious attempts to exploit the application through SQL Injection, XSS Scripting attack and other attacks
The CISP needs to be transparent as to how the application is operated and managed

7.2 Sentences Classification

In the second stage of the process, the goal is to feed the extracted sentences to the deep learning model, described earlier in section 6, with the goal of assigning the macro categories of the ontology treated in each one.

Before proceeding, it is essential to ensure that each sentence has been prepared correctly by following a cleaning procedure, similar to that used for the sentences in the dataset. This preliminary step is automated and includes carefully harmonizing the terminology used and correcting any imperfections in the text, thus ensuring that each sentence is error-free.

Next, the model is applied to each sentence through an iterative cycle. The specific applied function, illustrated in Listing 7.1, requires a few inputs: the current (cleaned) sentence to be processed, the fine-tuned BERT model, the BERT tokenizer, the maximum tokens length, and the device on which the computation is performed.

The function initially tokenizes the sentence, generating 'IDS' (unique identifiers for each token), 'MASK' (an attention mask to identify real tokens with respect to padding), and 'TOKEN_TYPE_IDS' (indications for models that handle multiple sequences, in our case all zeros). These elements prepare the input for the model and are transferred to the specified computing device. The model, set in evaluation mode, processes the input without computing gradients. The output is then transformed into probabilities using a sigmoid function and returned as a list of six values.

```
def test_model(sentence, model, tokenizer, max_len, device):
    # Tokenize the input sentence
    inputs = tokenizer.encode_plus(
        sentence, None, add_special_tokens=True, max_length=max_len,
        padding='max_length', return_token_type_ids=True, truncation=
    True, return_attention_mask=True, return_tensors='pt')

    ids = inputs['input_ids'].to(device, dtype=torch.long)
    mask = inputs['attention_mask'].to(device, dtype=torch.long)
    token_type_ids = inputs['token_type_ids'].to(device, dtype=torch.
    long)

    # Evaluate the model
    model.eval()
    with torch.no_grad():
        outputs = model(ids, mask, token_type_ids)
        predictions = torch.sigmoid(outputs).cpu().detach().numpy()
    return predictions.tolist()
```

Listing 7.1. Function to apply the model to a given sentence

Tables 7.2 and 7.3 provide two examples of sentences to which the function was applied; the corresponding output probabilities are shown for each. Then, by processing these values, only the categories registering a probability greater than 0.20, which is the predetermined threshold, are selected. Results highlights the effectiveness of the model in categorizing sentences, demonstrating its excellent ability to interpret natural language.

"Data encryption capabilities available in storage and database services"

Category	Probability
compute	0.0413
data handling	**0.6828**
network	0.0458
security & compliance	**0.8484**
management & monitoring	0.0701
cloud service essentials	0.0781

Table 7.2. Probabilities in the multi-label classification setting

The model correctly identified that the sentence deals with both security and data management, as they specifically deal with the concepts of encryption and the use of services such as storage and databases (will be identified in the next step).

"Demonstrate a multi-region deployment of a web application including a globally replicated database"

Category	Probability
compute	**0.2454**
data handling	**0.9050**
network	0.0298
security & compliance	0.0273
management & monitoring	0.0920
cloud service essentials	**0.4200**

Table 7.3. Probabilities in the multi-label classification setting

The sentence shows a strong correlation with the topic of data handling, given the reference to a database service. In parallel, the cloud service essentials are touched upon through the concepts of multi-region and replication. Finally, compute category is assigned, confirmed by the request for a web application.

Some sentences may not have received any category assignment; these are identified as non-requirements and are therefore excluded from further analysis as being irrelevant.

Figure 7.1 illustrates the composition of the sentences analyzed in the test document, showing that most of them were classified by the model as cloud requirements, that is, with at least one category.

In Figure 7.2 is presented a plot showing the presence of the labels assigned to these requirements for a more detailed analysis.

Both charts are included in the final report, this facilitates immediate understanding of the content and gives stakeholders a clear indication of the focus areas within the cloud context analyzed.

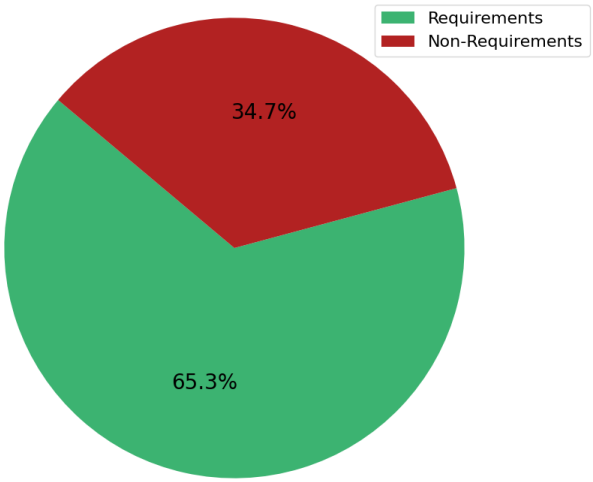

Figure 7.1. Distribution of Sentences in the Test Document

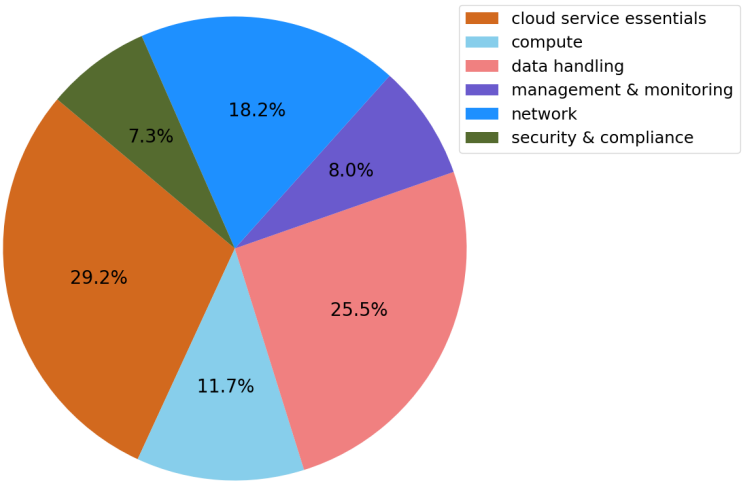

Figure 7.2. Requirements Distribution in the Test Document

7.3 Keyword Matching: Cloud Features Extraction

In the previous phase, the deep learning model associated each requirement with a group of main categories. Now the goal is to analyze these requirements more deeply to extract the specific cloud features defined in the ontology. For each requirement, are checked only the cloud features contained in the associated macro categories.

It is used an NLP technique called Keyword Searching, which is based on identifying matches with a predetermined set of key terms, by using regular expressions, for example. The presence of a keyword within a requirement indicates correlation with the corresponding cloud feature.

This approach was chosen based on the assumption that each cloud feature can be expressed in various and limited ways, and is reinforced by the previous step in which the model contextualized requirements by interpreting the meaning of words.

It was necessary to identify and link a set of keywords, including those composed of multiple terms, to each cloud feature. For the collection of keywords, manual Term Extraction was performed by analyzing the terminology present in the sentences of the dataset, which consists of 2164 records. This process made it possible to refine the definition of new keywords progressively.

Both the analyzed requirements and keywords have been normalized; this process is essential to ensure that the keyword search step performs well:

- The first normalization operation is "lowercasing", which standardizes the shape of the text by converting everything to lowercase, thus ensuring that keyword searching is not affected by differences between upper and lowercase letters.

- Disturbing characters such as "-", and "/", which can distort the analysis, were removed. As an example, a word such as "auto-scaling" is split into two words, "auto" and "scaling". Punctuation has also been eliminated, since signs such as commas and semicolons are no longer essential.

- Finally, stemming was applied, reducing words to their root form, or "stem", through the use of the popular PorterStemmer algorithm from the NLTK library [43]. This step is crucial to ensure that words in different inflected forms are recognized as the same entity. It is important to note that stemming can generate "stems" that do not correspond to real words, as in the case of "database" reduced to "databas", or "management" to "manag". However, this heuristic approach, although aggressive, is balanced by the previous contextualization performed by the model, which restricts the semantic range that a stem can have.

Table 7.4 shows the keywords associated with some of the cloud features. This assignment was made for each cloud feature defined in the ontology, of which there are 71 in total. It should be recalled that each feature has only one main category and subcategory to which it belongs, which are not mentioned due to space issues.

Cloud Feature	Keywords
Load Balancing	slb, load balancing, traffic distribution, load distribution, load management, load balancer, distributing traffic, lb, traffic management
NoSQL Database	key-value, graph database, non-relational database, nosql database, nosql
Firewall	firewall, packet filtering, proxy firewall, prevent access, packet inspection, traffic management, access control list, acl, inspect traffic
Encryption	encryption, cipher text, data ciphering, encrypted, encryption protocols, digital signature, key management, encryption keys, key lifecycle management, cryptographic key, kms
Dashboard	dashboard, display, management console, configuration management, self-service portal, portal, selfservice portal, console, web portal, graphical user interface, gui, panel, administration interface, management interface, single interface, user interface, web Interface, online interface, webui
Multi-Region	multi-region, regional availability, cross-region, location diversity, multi-regional, availability zones, multi-site, multi-cluster, multiple locations, geographical locations

Table 7.4. Keywords mined for some cloud features

After extracting cloud features, it is essential to illustrate the results to the user through precise requirements refinement, which consists of organizing and specifying the information gathered to make it clear and easily understood. This refinement process, which is automated to ensure efficiency and accuracy, is illustrated in the figure 7.3, which shows a portion of the requirements refinement performed automatically for the test document, where the extracted concepts are summarized and presented to the user.

1) The following requirements were extracted for main category "Compute":
- 1.1) Cloud features for subcategory "Resource types" are listed below:
  - 1.1.1) Virtual Machine (7)
  - 1.1.2) Containerization (3)
  - 1.1.3) Serverless (2)
- 1.2) Cloud features for subcategory "Elasticity" are listed below:
  - 1.2.1) Scaling (8)
  - 1.2.2) Load Balancing (4)
- 1.3) Cloud features for subcategory "Host provisioning" are listed below:
  - 1.3.1) Auto-Provisioning (1)

2) The following requirements were extracted for main category "Data handling":
- 2.1) Cloud features for subcategory "Data storage" are listed below:
  - 2.1.1) Data Storage (5)
- 2.2) Cloud features for subcategory "Database systems" are listed below:
  - 2.2.1) Relational Database (4)
  - 2.2.2) NoSQL Database (2)

3) The following requirements were extracted for main category "Network":
- 3.1) Cloud features for subcategory "Infrastructure" are listed below:
  - 3.1.1) Local Network (6)
- 3.2) Cloud features for subcategory "Services" are listed below:
  - 3.2.1) Firewall (3)
- 3.3) Cloud features for subcategory "Metrics" are listed below:
  - 3.3.1) Latency (1)

Figure 7.3. Portion of Refinement of Cloud Requirements

As shown in Figure 7.3, at this stage, a clear, comprehensive, and well-structured requirements refinement was generated entirely through an automated process.

In accordance with Requirement Engineering standards, where effective organization of requirements is crucial, we adopted a structured "forest"-like representation, organized in a modular manner.

The specification is divided into a numbered list: each top-level element corresponds with the 6 main categories of the ontology, representing core concepts. For the sake of space, only the first 3 macro categories are shown. Within these categories, there are sublists detailing the various subcategories that were established in the second layer of the ontology. For each of these, cloud features identified through keyword matching are listed, including frequencies of occurrence for each feature.

This modular structure, made possible thanks to the definition of the 3-level conceptual model described in section 4, ensures that the refinement is comprehensive, covering all possible concepts of cloud computing, while being unambiguous.

7.4 Mapping to Cloud Products

In the final stage of the project, the key objective is to associate each cloud feature, previously identified, with the specific products offered by leading cloud service providers, such as Amazon AWS, Google Cloud, and Microsoft Azure. For each cloud feature, a detailed analysis was conducted on the respective markets to identify the products that best represent it. There are some cloud features that have a "one-to-one" match and other cases where a cloud feature is represented by a possible group of products. Regarding the test document, the table 7.4 lists the products of the three vendors that match the cloud features previously identified. For space issues only mappings within the main categories "compute", "data handling", and "security and compliance" are shown.

Category: Compute

Feature: Virtual Machine
- AWS: EC2
- Google: Compute Engine
- Azure: Virtual Machines

Feature: Containerization
- AWS: ECS, EKS
- Google: Kubernetes Engine
- Azure: Kubernetes Service (AKS)

Feature: Serverless
- AWS: Lambda
- Google: Cloud Functions
- Azure: Functions

Feature: Scaling
- AWS: EC2 Auto Scaling, EC2
- Google: Cloud AutoScaling, Compute Engine
- Azure: Virtual Machine Scale Sets, Virtual Machines

Feature: Load Balancing
- AWS: ELB, EC2
- Google: Cloud Load Balancing, Compute Engine
- Azure: Load Balancer, Virtual Machines

Feature: Auto-Provisioning
- AWS: CloudFormation
- Google: Deployment Manager
- Azure: Resource Manager

Category: Data handling

Feature: Data Storage
- AWS: S3 (Simple Storage Service)
- Google: Cloud Storage
- Azure: Blob Storage

Feature: Relational Database
- AWS: RDS, Aurora
- Google: Cloud SQL
- Azure: SQL Database

Feature: NoSQL Database
- AWS: DynamoDB
- Google: Firestore
- Azure: Cosmos DB

Category: Security & compliance

Feature: Core IAM
- AWS: IAM
- Google: Cloud IAM
- Azure: Azure Active Directory (AD)

Feature: Data Security
- AWS: Macie
- Google: Sensitive Data Protection
- Azure: Azure Information Protection

Feature: Encryption
- AWS: Key Management Service
- Google: Cloud Key Management Service
- Azure: Azure Key Vault

Feature: Confidentiality
- AWS: Secrets Manager
- Google: Secret Manager
- Azure: Azure Key Vault (Secrets)

Feature: Vulnerability Assessment
- AWS: Inspector
- Google: Security Command Center
- Azure: Azure Security Center

Feature: Threat Detection
- AWS: GuardDuty
- Google: Security Command Center
- Azure: Azure Security Center

Feature: Cyber-Attack Protection
- AWS: Shield
- Google: Cloud Armor
- Azure: Azure DDoS Protection

Figure 7.4. Mappings in the test document of some cloud features to vendor products

Chapter 8

Conclusion and Future Work

The proposed approach, integrating advanced Deep Learning (DL) and natural language processing (NLP) techniques, has shown promise in simplifying the cloud services specification process. This innovative methodology automates the transformation of user needs, expressed in natural language, into detailed and structured technical specifications, thus facilitating the connection between user needs and the wide range of available cloud products of the different providers.

The results obtained demonstrate the effectiveness of the methodology. The BERT model showed high performance in accurately identifying and categorizing sentences, achieving excellent results in evaluation with metrics. The keyword searching phase further refined this process by extracting specific cloud features from the categorized requirements. An innovative method of collecting sentences to build the dataset was also proposed, exploiting generative AI technologies, which significantly expanded the quality and variety of training data available.

However, there are several areas where future work could improve and expand its capabilities. First, refinement of the ontology and expansion of keyword sets could improve the granularity of cloud feature extraction. In addition, the initial stage where sentences are extracted through document parsing could be customized by training a tailored model.

Another avenue for future improvements involves the analysis of requirements found but for which keyword searching has not found any cloud features to extract. One possible idea to handle them is to use vector embeddings, such as through a BERT family model, to compute the semantic similarity between the sentence and each possible keyword, selecting the highest similarities.

Finally, conducting experiments with real documents could provide valuable insights for further improvements. This would involve engaging potential end users, such as organizations that are planning to migrate to the cloud, to gather feedback and evaluate the practicality of the methodology in meeting their cloud service requirements.

Acknowledgments

I would like to express my sincere gratitude to those who supported and guided me along this path. Special thanks go to my supervisor, Prof. Emiliano Casalicchio, for allowing me to explore new stimulating fields of research and for his constant support and valuable guidance during the work. His renewed trust, in supporting me both in the path of the bachelor's thesis and now in the master's thesis, represents for me a precious sign of regard that I deeply appreciate. A special thanks is also extended to my family and friends, both near and far, for their unconditional support and for being a constant presence in my life. Finally, I would like to express my gratitude to my colleagues in the Department of Computer Science, who have always provided help and cooperation over the years, working together on numerous projects that have enriched my experience.

Bibliography

[1] *Zalazar, A., Ballejos, L., Rodriguez, S., "Analyzing Requirements Engineering for Cloud Computing", in "Requirements Engineering for Service and Cloud Computing", 2017*

[2] *Nawaz, F., Mohsin, A., Janjua, N.K., "Service description languages in cloud computing: state-of-the-art and research issues", SOCA, Vol. 13, pp. 109–125, 2019*

[3] *Liu, F., Tong, J., Mao, J., Bohn, R., Messina, J., Badger, M., and Leaf, D. (2011), "NIST Cloud Computing Reference Architecture," Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD*

[4] *Sommerville I., "Software Engineering", 9th edn. Pearson Education Inc., Saddle River, 2011*

[5] *Boehm B.W., "Software Engineering Economics", IEEE Transactions on Software Engineering, pp. 4–21, 1984*

[6] *IEEE, "Recommended Practice for Software Requirements Specifications", IEEE Standard 830-1998, IEEE Press, New York, 1998*

[7] *Feiler P.H., Gluch D.P., Hudak J.J., "The Architecture Analysis and Design Language (AADL): An Introduction", February 2006*

[8] *UML (Unified Modeling Language),* `https://www.uml.org/`

[9] *SysML (Systems Modeling Language),* `https://sysml.org/`

[10] *ArchiMate,* `https://www.archimatetool.com/`

[11] *Awan, Misbah Mehboob, et al., "Formal Requirements Specification: Z Notation Meta Model Facilitating Model to Model Transformation.", Proceedings of the 2020 9th International Conference on Software and Information Engineering (ICSIE), 2020*

[12] *Galhardo, P.; Silva, A.R.d., "Combining Rigorous Requirements Specifications with Low-Code Platforms to Rapid Development Software Business Applications", Appl. Sci., 2022*

[13] *Quidgest Genio,* `https://genio.quidgest.com`

[14] Sandobalin, J.; Insfran, E.; Abrahao, S., "Argon: A model-driven infrastructure provisioning tool", in ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion, IEEE, 2019

[15] Osaba, Eneko; Díaz-de-Arcaya, Josu; Orue-Echevarria Arrieta, Leire; Alonso, Juncal; López Lobo, Jesús; Benguria, Gorka; Etxaniz, Iñaki, "PIACERE project: description and prototype for optimizing infrastructure as code deployment configurations", 2022

[16] Martinez, Rex. "Artificial intelligence: Distinguishing between types and definitions." Nevada Law Journal, vol. 19, no. 3, p. 9, 2019

[17] Zhao, Liping; Alhoshan, Waad; Ferrari, Alessio; Letsholo, Keletso J.; Ajagbe, Muideen A.; Chioasca, Erol-Valeriu; Batista-Navarro, Riza T., "Natural Language Processing (NLP) for Requirements Engineering: A Systematic Mapping Study", 2020

[18] Liddy, E. D. "Natural Language Processing" In Encyclopedia of Library and Information Science, 2nd ed., New York: Marcel Decker, Inc., 2001

[19] Cleland-Huang, J.; Settimi, R.; Zou, X.; Solc, P., "Automated classification of non-functional requirements", Requirements Engineering, vol. 12, no. 2, pp. 103–120, 2007

[20] Binkhonain, M., Zhao, L. "A review of machine learning algorithms for identification and classification of non-functional requirements" Expert Systems with Applications: X, vol. 1, 2019

[21] Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; Polosukhin, I., "Attention Is All You Need", 2017

[22] Devlin J., Chang M.W., Lee K., Toutanova K., "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", 2018

[23] V. Ivanov, A. Sadovykh, A. Naumchev, A. Bagnato, K. Yakovlev, "Extracting Software Requirements from Unstructured Documents", 2022

[24] S. Bashir, M. Abbas, M. Saadatmand, E. P. Enoiu, M. Bohlin, and P. Lindberg, "Requirement or not, that is the question: A case from the railway industry", 2023

[25] A. Sainani, P. R. Anish, V. Joshi, S. Ghaisas, "Extracting and Classifying Requirements from Software Engineering Contracts", 2020

[26] Dae-Kyoo Kim, "Using ChatGPT to Develop Software Systems: Alert to Software Engineers?", Aprile 2023

[27] Endut N., et al., "A Systematic Literature Review on Multilabel Classification Based on Machine Learning Algorithms", TEMJournal, vol. 11, no. 2, pp. 658-666, 2022

[28] Staab, S.; Studer, R., "Handbook on Ontologies", Springer Science & Business Media, Karlsruhe, Germany, 2013

[29] Agbaegbu, J.; Arogundade, O.T.; Misra, S.; Damaševičius, R., "Ontologies in Cloud Computing—Review and Future Directions", Future Internet, 2021

[30] Amazon AWS Marketplace, https://aws.amazon.com/marketplace

[31] Google Cloud Marketplace, https://cloud.google.com/marketplace

[32] Microsoft Azure Marketplace, https://azuremarketplace.microsoft.com/

[33] Sayyad Shirabad, J. and Menzies, T.J., "The PROMISE Repository of Software Engineering Databases", School of Information Technology and Engineering, University of Ottawa, Canada, 2005

[34] Knauss, E., Houmb, S., Schneider, K., Islam, S., Jürjens, J., "Supporting Requirements Engineers in Recognising Security Issues", 2011

[35] A. Ferrari, G. O. Spagnolo and S. Gnesi, "PURE: A Dataset of Public Requirements Documents", 2017

[36] Transformers, https://huggingface.co/docs/transformers/index

[37] PyTorch, https://pytorch.org/

[38] NumPy, https://numpy.org/

[39] Pandas, https://pandas.pydata.org/

[40] Scikit-Learn, https://scikit-learn.org/stable/

[41] Berry, D.M., "Empirical evaluation of tools for hairy requirements engineering tasks", Empirical Software Engineering 26(6), 2021

[42] SpaCy, https://spacy.io/

[43] NLTK, https://www.nltk.org/