Large Language Models Based Automatic Synthesis of Software Specifications

SHANTANU MANDAL, Texas A&M University, USA
ADHRIK CHETHAN, Texas A&M University, USA
VAHID JANFAZA, Texas A&M University, USA
S M FARABI MAHMUD, Texas A&M University, USA
TODD A ANDERSON, Intel Labs, USA
JAVIER TUREK, Intel Labs, USA
JESMIN JAHAN TITHI, Intel Labs, USA
ABDULLAH MUZAHID, Texas A&M University, USA

Software configurations play a crucial role in determining the behavior of software systems. In order to ensure safe and error-free operation, it is necessary to identify the correct configuration, along with their valid bounds and rules, which are commonly referred to as software specifications. As software systems grow in complexity and scale, the number of configurations and associated specifications required to ensure the correct operation can become large and prohibitively difficult to manipulate manually. Due to the fast pace of software development, it is often the case that correct software specifications are not thoroughly checked or validated within the software itself. Rather, they are frequently discussed and documented in a variety of external sources, including software manuals, code comments, and online discussion forums. Therefore, it is hard for the system administrator to know the correct specifications of configurations due to the lack of clarity, organization, and a centralized unified source to look at. To address this challenge, we propose SpecSyn a framework that leverages a state-of-the-art large language model to automatically synthesize software specifications from natural language sources. Our approach formulates software specification synthesis as a sequence-to-sequence learning problem and investigates the extraction of specifications from large contextual texts. This is the **first** work that uses a large language model for end-to-end specification synthesis from natural language texts. Empirical results demonstrate that our system outperforms prior state-of-the-art specification synthesis tool by 21% in terms of F1 score and can find specifications from single as well as multiple sentences.

Additional Key Words and Phrases: software specification synthesis, natural language processing, deep learning

1 INTRODUCTION

Software configurations represent an essential component of software systems. System failures induced by software misconfigurations (i.e., not setting various configuration parameters according certain specifications) have become increasingly common [17, 42, 61]. Such misconfigurations give rise to a range of issues, including application outages, security vulnerabilities, and inaccuracies in program execution [14, 57, 60]. The adverse impact of software misconfigurations is demonstrated in several high-profile cases [9, 10, 16, 33]. For instance, a configuration error caused by an internet backbone company, resulted in a nationwide network outage in 2017 [10]. Similarly, in 2019, millions of Facebook users were affected by a server misconfiguration outage [9]. Furthermore, a system configuration change led to a five-hour outage of AT&T's 911 service, preventing numerous callers from accessing the emergency line [16]. Therefore, the development of effective tools to help prevent software misconfigurations is of utmost importance.

The research community has recognized the criticality of software misconfiguration and proposed numerous efforts to address it by developing techniques to check, troubleshoot and diagnose configuration errors [1, 5, 60]. However, the majority of the approaches can generally only be applied after an error has occurred. Primarily, the root cause of software misconfiguration is attributed to human errors. Thus, a more effective strategy for avoiding such misconfigurations would be to guide or enforce the correct usage of configuration practices, rather than identifying and correcting them after a failure has already occurred. Typically, software configurations are set by software administrators. To aid these administrators, software vendors often release user manuals that describe different configuration specifications. These manuals, typically available in PDF or HTML format, provide guidance on the correct and recommended setup of configurations to system administrators, containing detailed textual descriptions of configuration parameters, their descriptions, usage, and constraints. Nevertheless, as these manuals are exceedingly voluminous, many administrators tend not to read them in detail and instead rely on intuition to configure software [34, 59], frequently leading to misconfiguration and subsequent software failures. Hence, it is crucial to develop an automatic tool that extracts configuration specifications from these sources, to provide guidance to administrators, or integrate automated tools that suggest best practices. Thus, this paper investigates the feasibility of building an automatic tool capable of extracting specifications from unstructured sources of configuration descriptions, which are predominantly written in a natural language such as English.

Specifications refer to a set of legitimate rules or guidelines that dictate the configuration of software. Failure to adhere to these specifications by configuring the software with invalid parameters may result in software malfunction. Extensive research has been conducted to extract software specifications from unstructured specification sources [32, 44, 56]. For instance, in PracExtractor [56], Xiang et al. utilize the Universal Dependency algorithm [11] to synthesize specifications from software manuals. This technique establishes a syntactic mapping between various parts of speech

Authors' addresses: Shantanu Mandal, Texas A&M University, USA, shanto@tamu.edu; Adhrik Chethan, Texas A&M University, USA; Vahid Janfaza, Texas A&M University, USA; S M Farabi Mahmud, Texas A&M University, USA; Todd A Anderson, Intel Labs, USA; Javier Turek, Intel Labs, USA; Jesmin Jahan Tithi, Intel Labs, USA; Abdullah Muzahid, Texas A&M University, USA.

within a sentence. To construct the set of syntactic relationship trees, they initially collected samples from software manuals that contain valid specifications. They then attempted to match other sentences' syntactic relation with the collected syntactic relation tree. If a match was found, it implied that the samples also contained specifications and it was extracted based on the relation. The authors of ConfigV [44] have employed a rule-based approach to synthesize specifications from configuration files. This approach involves initially parsing a training set of configuration files, which may be partially correct, to create a well-structured and probabilistically-typed intermediate representation. A learner that utilizes an association rule algorithm is then employed to translate this intermediate representation into a set of rules. These rules are subsequently refined and ranked through rule graph analysis to synthesize specifications. Researchers have also tried to synthesize specifications from programming source code by employing static analysis [32]. However, specifications collected through this approach need more analysis and expert knowledge for refinement by humans.

The majority of prior methods for synthesizing specifications from unstructured sources have relied on rule-based approaches. However, such approaches are known to have limited generalizability and may require human intervention during certain steps of the synthesis process. Alternatively, learning-based approaches are better suited for discovering relationships in unstructured data, particularly those utilizing deep learning techniques that have shown significant success in various unstructured data domains, including natural languages, images, and videos. As the main objective of this paper is to synthesize specifications from a natural language source, we explore the potential of deep learning-based approaches to address this problem.

To this end, we have framed the specification synthesis problem as an **end-to-end sequence-to-sequence** learning problem. The resulting system is referred to as **SpecSyn** , which takes texts as an input and produces specifications as an output. The synthesis process involves two steps of prediction. First, it checks whether the input texts contain any specification. If they do, secondly, SpecSyn performs end-to-end synthesis of the specifications. Given that the input for specification synthesis is in the form of a natural language text, we have incorporated a large pre-trained language model to enhance the model's understanding of the context. One such widely used model is BERT [12], a deep learning-based model that pre-trains on a large corpus of English texts to learn the latent representations (i.e., a vector) of words and sentences within their respective contexts. We have fine-tuned [48] the BERT model for our specification synthesis task using a custom decoder (a decoder is a part of the language model that produces outputs based on the latent representations). The incorporation of this large language model has enabled us to synthesize not only simple single-sentence specifications but also complex sets of specifications from texts consisting of multiple sentences and parameters. Figure 1 shows the high level workflow of SpecSyn .

1.1 Contributions

We make the following technical contributions in this paper.

Problem Formulation: We formulate the task of software specification extraction from natural language texts as an end-to-end sequence-to-sequence learning problem. This approach involves mapping an input sequence, consisting of natural language texts, to an output sequence that comprises of single or a set of relevant software specifications.

Contextual Model Integration: In order to achieve accurate and effective extraction of software specifications from natural language texts, we propose to use the state-of-the-art BERT [12] model. By leveraging the acquired knowledge and advanced language processing capabilities of a pre-trained BERT model, we aim to extract relevant specifications from the text in a manner that is both accurate and efficient. Through empirical analysis, we demonstrate the effectiveness of SpecSyn in the context of software specification extraction from natural language texts. To the best of our knowledge, this is the **first** paper that uses a large language model for end-to-end synthesis of configurations specifications from natural language texts.

Complex Dependency Modeling: The current investigation presents a model that is able to process text consisting of multiple sentences. Specifically, our proposed model is designed to effectively capture complex specification relations within longer text. Notably, the model is capable of discerning the relationships between multiple specifications contained within a single sentence, as well as extracting individual specifications that are connected in a meaningful manner within a text. The efficacy of our model is demonstrated through empirical analyses, which provide evidence of its ability to accurately identify and extract relevant information from longer text data.

Generality: The framework, SpecSyn , is capable of processing any natural language text, thereby rendering it independent of the source of textual data. Consequently, it does not solely rely on software manuals for the extraction of software specifications. Rather, it can be utilized to extract relevant specifications from a wide range of sources including software codebase comments, as well as online resources such as StackOverflow and discussion forums. This flexibility allows for the construction of a more comprehensive and robust set of specifications, thereby enhancing the overall effectiveness of the framework.

1.2 Outline

The remainder of this paper is organized as follows. Section 2 describes the details of SpecSyn framework. First, the section lays out the specification definition, followed by discussion on specification extraction types and specification categories. Then, different specification sources and dataset construction approaches are described. After that, we describe the model development and integration of a large language model with our framework. Section 3 describes the experimental setup and experimental results. Section 4 introduces the background and related work. Section 5 discusses potential future work. Finally, Section 6 concludes this paper.

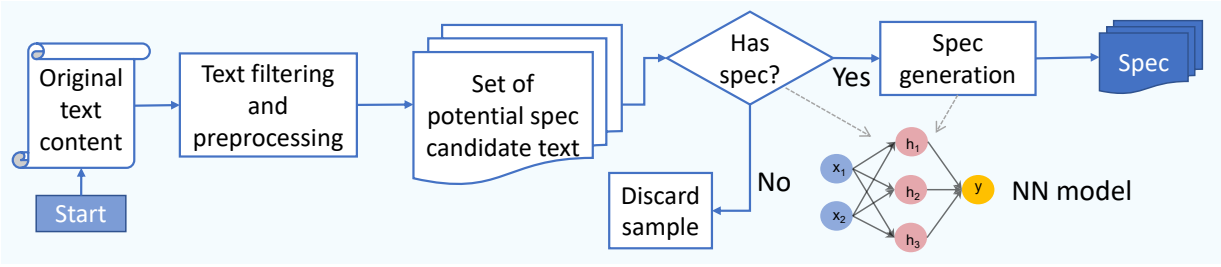

Fig. 1. Overview of SpecSyn

2 SPECSYN : SPECIFICATION SYNTHESIS FRAMEWORK

2.1 Software Specification

Software specification is a set of rules that consists of relation between various keywords, numbers or pre-formatted string. Here, keywords represent different configuration parameters. Specification defines how a configuration should be presented by outlining the specific rules and requirements for configuration format and structure. Formally, a specification can be defined as Definition 1.

Definition 1: A specification S is defined as $S = \{R_i\}$, where R_i is a rule represented by a tuple, $R_i = \langle K_i, V_i, L_i \rangle$. Here,
$K_i \in Keyword$
$V_i \in \{\mathbb{R}, Keyword, S\}$ *where* $\mathbb{R} = Set\ of\ Real\ Numbers, S = String$
$L_i \in \{\emptyset, =, \neq, >, <, AND, OR, Interval, Set, Use, With, String\ Format\}$

The goal of SpecSyn is to analyze the natural language texts from various sources and produce a specification, if there is any in the texts. Next we are going to classify types of specification extraction as well as various categories of specifications.

2.1.1 Specification Extraction Type: Based on the presence of specification in the text, we categorize specification extractions into two major types: *Simple* and *Complex*. By distinguishing between these two types, we are able to better evaluate the performance of our extraction process. Previous research has focused exclusively on the *Simple* extraction category, while the *Complex* category demands more sophisticated extraction process and modeling techniques. Table 1 shows examples of text for different specification extraction types.

Type	Example
Simple	It is necessary to use a number greater than 1500 for *user_port*
$\text{Complex}_{\text{Single}}$	The default pointer size in bytes is used when *max_rows* option is specified. This variable should be between 2 and 7.
$\text{Complex}_{\text{Multi}}$	*have_ssl* and *have_open_ssl* need to be set True to enable secured connection.

Table 1. Examples of specification extraction types

Simple Extraction: *Simple* extraction refers to specifications extraction process where the specification is located within a single sentence containing only one specification. The majority of specification extractions fall within this category. The task of modeling *Simple* extractions is comparatively less intricate due to the concise nature of general sentences. The specifications are also defined in a more straightforward manner. We observed that fewer training examples are required to train a model for extracting *Simple* categories as opposed to *Complex* categories.

Complex Extraction: The second type, known as *Complex* Extraction, consists of cases where multiple sentences are required to extract specifications from the text or when there are multiple specifications intertwined in a text that need to be extracted. *Complex* extraction has been further subdivided into two distinct subcategories, namely $Complex_{Single}$ and $Complex_{Multi}$. The $Complex_{Single}$ category is applicable when only one specification is present in the text, but the extraction process requires examining multiple sentences. Typically, in such cases, the first sentence identifies the parameter keyword, while the subsequent sentences describe the specification. The sentences usually connected by some pronoun. In order to extract the specification and identify the keyword to which it refers, all sentences are necessary to look at. On the other hand, $Complex_{Multi}$ refers to situations where multiple specifications are defined within a text, and multiple parameter keywords are used to refer to a single specification. Basically in this category multiple specifications of multiple keywords are described in a compact intertwined fashion in a natural language.

2.1.2 Specification Categories: We categorize the specification into five major categories based on the underlying specification definition and property. Among them, *Quantitative* represents specifications that are quantifiable, such as numerical or boolean comparisons. *Utilization* categorizes any usage of keyword for any particular task, *Interrelation* categorizes correlation between keywords, *Attribute* categorizes different attribute such as path, domain etc., and *Generic* categorizes general suggestions without any specific criteria. The categories and their examples are described in table 2.

The majority of the specifications identified in our study can be categorized as *Quantitative*. *Quantitative* specification can have multiple definitions. The *Generic* category lacks a specific definition and is primarily characterized by suggestions or recommendations. The remaining three categories are infrequent and each have only

Category	Example
Quantitative	It is recommended to raise the *ulimit* to 10,000, but more likely 10,240 because the value is usually expressed in multiples of 1024.
Utilization	Mount option *sync* is strongly recommended since it can minimize or avoid reordered writes, which results in more predictable throughput.
Interrelation	If you are having problems with the service, it is suggested you follow the instructions below to try starting *httpd.exe* from a console window, and work out the errors before struggling to start it as a service again.
Attribute	To avoid the ambiguity, users can specify the plugin option as *–pluginsql-mode*. Use of the *–plugin* prefix for plugin options is recommended to avoid any question of ambiguity.
Generic	It is recommended but not required that *–ssl-ca* also be specified so that the public certificate provided by the server can be verified.

Table 2. Different categories of specifications with examples

Category	Definition	Pattern
Quantitative	p == v, p < v | p > v, p ∈ [v, v'], p ∈ {v, v'},	$v_{<value>}$, $less_{syn}$ | $more_{syn}$ than v_{value}, $between_{syn}$ $v'_{<value>}$ to $v2_{<value>}$, $v_{<value>}$ or $v'_{<value>}$
Utilization	use(p)	$used_{syn}$ | $useful_{syn}$
Interrelation	with(p, p'), prefer(p, p')	$along_{syn}$ with p'_{para} $prefer_{syn}p'_{para}$
Attribute	format(p,f)	$f_{<format>}$
Generic	{recommended, prioritize, ...}	

Table 3. Specification definition and patterns

one specification definition. Table 3 describes the definition and pattern of each categories.

The significance of specification categorization lies in its potential to facilitate more effective utilization of detected specifications during later stages. Even though in our work the initial detection of specifications is a binary prediction process independent of their categories, the ability to categorize specifications can provide valuable insights for optimizing their deployment in later stages such as suggesting them to system administrators or incorporating them to other tools. Therefore, in our work, we also demonstrate the capability of our framework to accurately predict specification categories with different decoders. Although these categories are intuitive and general to notice, we are inspired about them from prior work [56]. However, previous research has not explored the detection capabilities of various categories of specification as we have done in our framework.

2.2 Data Collection

Data is the lifeline of any deep learning-based solutions, and collecting data to train and test any deep learning-based system is one of the most significant and challenging tasks. For specification synthesis, it becomes even more difficult since it requires highly specific and domain-dependent data. Since there is no standard dataset available for specification synthesis, we have to collect and create our own datasets, which is a time-consuming and resource-intensive task. However, despite the challenges, there are several sources where software specifications can be found. The main source of specifications is the software manual. However, it can also be derived from comments embedded in the software code base, particularly when the source code is publicly available. It may be obtained from various other sources also such as Stack Overflow, online discussion forums, and other community-driven platforms. In the following section, we will describe different sources of software specifications and the specification collection process.

Name	Software type	Format	Pages	Keyword
MySQL	Database	PDF	6644	Yes
PostgreSQL	Database	PDF	3055	Yes
HDFS	Distributed Storage	HTML	2331	Yes
HBase	Distributed Storage	HTML	787	Yes
Cassandra	Distributed Storage	HTML	50	No
Spark	Distributed Computing	PDF	66	Yes
HTTPD	Web Server	HTML	1009	Yes
NGINX	Proxy	HTML	900	No
Squid	Proxy	HTML	330	Yes
Flink	Stream Processing	HTML	1434	Yes

Table 4. Description of software manuals

2.2.1 Data Source: We mainly collect specifications from three major sources. The most important source of the specification is software manuals. The major portion of specifications is collected from manuals. We also collected specifications from source code comments and other online sources.

Software Manuals The main source of software specification is the software manual. These manuals are typically provided by the corresponding software vendor and contain detailed information about how to use the software, including its features, functions, and limitations. Software manuals are often available in different formats such as PDF, HTML, or in online. Table 4 details the summary of software manuals that we used to collect specifications. In order to construct our dataset, we gathered specifications from 10 diverse software manuals, spanning a range of software domains. Typically, software manuals are extensive in nature, containing a considerable amount of information. For instance, MySQL's manuals consist of a

total of around 6500 pages. Due to the sheer volume of information contained in these manuals, it is impractical to read them line by line. Therefore, a keyword-based filtering method is employed to extract only the relevant sentences for closer examination. These keywords typically correspond to configuration parameters, which can be located within the manuals. The majority of software packages typically have their keywords listed either in the manuals or on their configuration page. However, in our study, we were unable to find such a listing for NGINX and Cassandra. In the case of these two software packages, we solicited human assistance to aid us in extracting the specifications. While simple specifications can often be derived from individual sentences, more complex specifications may require analysis of neighboring sentences. As complex specifications may be embedded within the text, the focus is on the section of the text where the keyword is present.

Other Sources In addition to the manual-based specification, we also collected specifications from two other additional sources: software code base comments and online discussion forums. As our method operates with natural language, it is independent to the source of the data, provided that it is composed in a natural language. The inclusion of these alternative sources is primarily intended to demonstrate the generalizability of our framework across a broad range of text-based specification descriptions.

To demonstrate the source code base specification generalizability, we develop and integrate a parser into our framework and apply the parsing to MySQL source code only. However, the same methodology can be applied to parsing other software as well. For the purpose of parsing comments from MySQL source code, we use a Python-based parser. It is essential to be cautious when parsing the source code in this manner, as the codebase also contains commented-out codes. But we are only interested in text-based comments. Therefore, commented out code needs to be discarded for a better-refined dataset. In addition, a crawler is also developed to extract software keyword-specific posts from StackOverflow to augment the specification-related dataset.

Tag	Pattern
<bool>	"enable" | "on" | "true" | "disable" | "false" | "off" | ...
<num>	$\forall$ w $\in \mathbb{R}$
<unit>	“byte” | “MB” | “ms” | “%” | ...
<keyword>	$\forall$ w $\in$ Configuration Parameter Name
<format>	“email address” | “absolute path” | “domain name” | ...

Table 5. Description of pattern for data composition

2.2.2 Data Composition: Prior to generating the candidate specification texts from the original contents, our system refines the dataset through a series of preprocessing steps. First, the texts are divided into smaller candidate texts based on the extraction type. Then it verifies the presence of the keyword in the candidate texts. If the keyword is absent, the candidate texts are discarded and not considered as a potential sample for further processing. Both the extraction types (i.e., *Simple* and *Complex*) require the presence of relevant keywords in the candidate texts. The keywords can easily be found for each of the software in different places (e.g., manual, web). Upon identifying potential candidates, the system proceeds to search for predefined patterns within the candidate text as specified in Table 5. These patterns are then replaced with corresponding tags. In instances where identical patterns are present multiple times within the text, tags are differentiated through the use of different identifier. This process enhances the generality of the potential candidates, thereby enabling easier detection and synthesis of specifications by the model. Thus, each candidate comprises of tuple $\langle C, T \rangle$, where C represents the candidate text and T represents the associated pattern tag set. After detecting and synthesizing specification based on C, the system can reconstruct the original representation of the specification by replacing the tag in synthesized specification of C with the corresponding pattern in T.

One may argue that passing the keyword set for finding potential specification candidate text may introduce more redundancy. Rather, the system should identify the keyword itself. However, knowing the keyword set for a particular software is necessary. Multiple software can have same keyword with different specification rule. Therefore, if it is not known for which software the specifications are being synthesized, then the system can synthesize a syntactically correct but potentially wrong specification for a software. This is especially true if SpecSyn synthesizes specification from other sources. Therefore, knowing for which software the specifications are being synthesized and corresponding keyword set is quite important. Also, Keyword-based filtering enables us to accomplish two goals. First, it eliminates a significant portion of the samples that are irrelevant in nature. Since a model can easily recognize these and accurately classify them as non-specification, the model's performance would be heavily skewed towards making accurate predictions of true negatives. Therefore, considering sentences that has keywords or solely considering neighboring sentences with keywords will allow for a fairer performance comparison. Second, keyword based filtering also discards a major portion of false positives. For example text like “See page 157 for details of MySQL 11.7.8” does not hold any specification, but this can be detected as specification if the model is not trained with larger number of samples. In our study, we discovered that a model can also be trained for all cases even without implementing keyword-based filtering. However, such approach requires a larger number of samples to be used for model training. Also, identifying syntactically valid but semantically incorrect specifications requires additional failure checks through software execution. Therefore, due to resource and time constraint, in this project, we pursue a keyword-based filtering process and keep the other ideas as potential future work.

2.3 Model Development

2.3.1 Contextual Model Integration: BERT. Contextual model integration refers to the process of combining language models to improve the performance of natural language processing tasks. The idea is to leverage the strengths of different models and combine them in a way that captures the complex relationships between words and their contexts in a given text. One of the approaches of contextual model integration is to use a hierarchical model, where

one model is used to capture the overall context of the input sequence and another model is used to make more specific predictions based on the context. In our case, we use BERT (Bidirectional Encoder Representations from Transformers), a pre-trained large language model [12], that is trained on a large dataset of natural language texts and we fine-tune it with our task-specific custom decoder.

BERT [12] is a powerful neural network model that has revolutionized the field of natural language processing (NLP) in recent years. It was first introduced by Google in 2018 and has since become one of the most widely used and effective language models in NLP. The core idea behind BERT is to leverage the power of Transformer-based architectures [51] to create a deep bidirectional language model that can capture contextual information from both directions of the input sequence. Unlike traditional language models, which are trained in a left-to-right or right-to-left fashion, BERT is trained using a masked language model (MLM) objective that randomly masks certain tokens in the input sequence and requires the model to predict the missing word based on the surrounding context. One of the key innovations of BERT is the use of multiple layers of self-attention to capture complex relationships between words in the input sequence. Each layer of the model contains a self-attention [51] mechanism that allows the model to attend to different parts of the input sequence and capture dependencies between words that are far apart in the input. Additionally, BERT uses a combination of word embeddings, positional embeddings, and segment embeddings to capture both the meaning and position of words in the input sequence. All of these embeddings are self-learned through back-propagation while training on a larger dataset. BERT has proven to be highly effective for a wide range of NLP tasks, including text classification, question-answering, language translation, etc . [38]. In order to adapt BERT to specific tasks, researchers typically fine-tune [48] the model by adding a task-specific output layer and training the model on a task-specific dataset. The fine-tuning process allows the model to learn task-specific features and improve its performance on the target task.

2.3.2 Specification Detection and Generation. The specification synthesis process is a two-step procedure that involves specification detection and generation. In the first step, the text is examined to ascertain the presence or absence of a specification. If a specification is detected, in the second step the specification is synthesized. Figure 2 shows the neural network model for SpecSyn . Specification detection is a binary classification problem, where a BERT encoder is utilized. An encoder is a sequence of layers to convert the input texts in a hidden vector h_c. A special token $[CLS]$ is added at the beginning of the text to generate h_c of the entire sequence for classification. This hidden state is subsequently passed to the decoder. A decoder is a sequence of layers to produce the final output from h_c. The softmax layer in the decoder produces the binary prediction of the presence or absence of a specification in the input text. Basically, the specification classification process can be described as Equation 1-3, where $X = \{x_{CLS}, x_1, x_2, ..., x_n\}$ is the tokenize set of input text, x_{CLS} is the special token, W is the weight matrix for the custom decoder and c is the expected output prediction. We fine-tune W to maximize the log probability of the correct class.

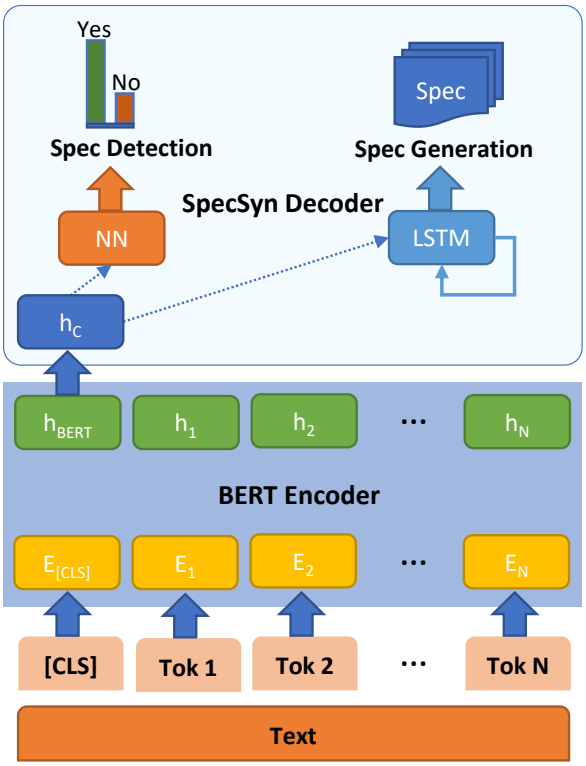

Fig. 2. SpecSyn model architecture

For specification classification tasks, BERT takes the final hidden state h_{BERT} of the first token $[CLS]$ as the representation of the whole sequence. A simple softmax layer-based classifier is added as the custom decoder on the top of BERT encoder to predict the probability of label c.

$$h_{\text{BERT}} = f_{\text{BERT}}(X) \tag{1}$$

$$\boldsymbol{h_c} = f_1(W_1 h_{\text{BERT}}) \tag{2}$$

$$p(c|\boldsymbol{h_c}) = \text{softmax}(W_2 \boldsymbol{h_c}) \tag{3}$$

For the specification synthesizing process, we use the same hidden state h_c and pass it to a different sequence generation decoder. This decoder synthesizes the specification according to the pattern defined in Table 3. The synthesizing decoder is an LSTM-based [21] recurrent neural network defined as Equation 5. It takes h_c as the hidden state and a start token to start the synthesis process and construct the specification as it goes.

$$h_0 = \boldsymbol{h_c} \tag{4}$$

$$o_i = \text{LSTM}(h_{i-1}, o_{i-1}) \tag{5}$$

$$o_{\text{Specification}} = \{o_1, ..., o_m\} \tag{6}$$

A potential argument in favor of utilizing a single decoder for both the specification classification and generation tasks may be put forth. However, given that the majority of texts does not contain specifications and the generation of specifications is a complex task in comparison to binary classification, we found in our study that the use of two separate decoders yield better results in classifying and generating specifications within a text.

In our study, we categorize specifications into different categories. We can predict different classes of these categories with the same method. Upon detection of a specification, a decoder with a softmax layer having the desired number of classes can be used to classify different categories. A detailed analysis of the result is presented in the results section for this.

2.3.3 Loss Function. In the context of specification detection, a binary classifier is utilized to determine whether a given text contains any specifications or not. For this, we use weighted cross entropy loss function [3] defined by Equation 7 where M is the number of classes, log is natural log, y is binary indicator (0 or 1) if class label c is the correct classification for observation o, and p is the predicted probability observation o is of class c.

$$\mathscr{L} = -\sum_{c=1}^{M} w_c y_{o,c} \log(p_o, c) \tag{7}$$

Weighted cross entropy loss is a commonly used loss function in classification problems when dealing with imbalanced datasets [20, 27]. This method assigns a higher weight to the minority class samples to improve the performance of the model. The weight is determined based on the distribution of the minority class samples in the dataset. By assigning a higher weight to this class samples, the model is encouraged to focus more on correctly classifying these. Weighted cross entropy loss has been shown to be effective in improving the performance of models on imbalanced datasets, particularly in text classification tasks. Its usage can lead to a more accurate and reliable classification of the minority class. In our specific dataset, it has been observed that the number of texts containing specifications is significantly lower than those that do not contain any specifications, leading to an imbalanced dataset. In order to address this issue, we have utilized weighted cross entropy loss function. The weight is determined based on the distribution of specification-containing texts in the training set.

3 RESULTS

In total, we collected 300 specifications from various sources, including 10 software manuals and other resources. The majority of the specifications were extracted from two sizable software manuals, due to their extensive page count. We categorize the software manuals into 3 groups according to their software types. From Table 4, first 2 softwares are categorized as Database software, next 4 as Distributed System and last 4 are categorized as Proxy.

The quantity of available specifications are limited, as evidenced by prior research [56] too . Given that the dataset used in the previous study is not publicly accessible, we collected same number of specification as them. A deep learning based network requires a moderate number of training examples in order to train. However, given the lower number of available specification, it becomes challenging to create training and testing sets out of them. Therefore we generate synthetic training examples by sampling 50 real specifications from our collected set and create the training set. Since we know the specification of these random samples, we put random text inside them to create synthetic samples. That way we create our training set that consists of 3000 samples in total. Thus, we were able to create a good amount of samples for the training and we use rest 250 samples as the testing set to evaluate.

In our model, we used a pretrained BERT model available online at https://huggingface.co/. Although, the BERT model is large in nature, since this is pretrained, we do not need to train them from scratch. We design our own decoder neural network for our own task. We use a feed forward neural network consisting of 2 hidden layer of 50 neurons each with a softmax function on top for specification detection. Weighted cross entropy loss is used for this decoder. To generate specifications, we use an LSTM-based recurrent neural network with 20 hidden units. Given the limited number of training examples, our designed neural network demonstrates sufficient capacity to achieve better prediction performance. We train our model for 100 epochs until the training converges.

3.1 Demonstration of Synthesis Ability

We compare our work with state-of-the-art specification synthesis tool named PracExtractor [56]. PracExtractor uses Universal Dependency algorithm. This tool only work for *Simple* extraction type specification. On average a *Simple* extracted type specification is 3 tokens long. Therefore, given the short length of specifications, our model is capable of synthesizing them accurately once they are detected. PracExtractor does not have any detection mechanism. Therefore, when they are able to synthesize a specification they are counted as successfully detected.

Table 6 shows the specification detection performance comparison between PracExtractor and SpecSyn . It showed result with three different software groups - Database, Distributed System and Proxy. PracExtractor performs poorly compared to SpecSyn in all three cases. In case of Database software, SpecSyn has higher Precision 0.95 compared to 0.84 Precision of PracExtractor. The Recall score is 0.90 which is significantly better than Recall for PracExtractor (0.58). Consequently F1 score for SpecSyn (0.92) is significantly better than that of PracExtractor (0.68) For Distributed System software type, SpecSyn has better Precision, Recall and f1 score compared to that of PracExtractor. In this case, Precision for SpecSyn is 0.90 vs Precision of PracExtractor is0.79, Recall for SpecSyn is 0.75 compared to 0.50 Recall value for PracExtractor. In case of F1 Score, SpecSyn has higher value 0.82 compared to the state of the art PracExtractor model (0.61) In case of Proxy applications, we see similar trends in the result. Precision for SpecSyn is 0.92 and Precision for PracExtractor is 0.81 only. Recall is also much better for SpecSyn (0.79) compared to Recall value for PracExtractor (0.52). So F1 score for Proxy type software is higher in SpecSyn (0.86) compared to F1 score of PracExtractor (0.65).

For combined results of all three types of software, we can see that SpecSyn has higher Precision (0.92) compared to PracExtractor

Software Type	PracExtractor			SpecSyn		
	Precision	Recall	F1 Score	Precision	Recall	F1 Score
Database	0.84	0.58	0.68	0.95	0.90	0.92
Distributed System	0.79	0.50	0.61	0.90	0.75	0.82
Proxy	0.81	0.52	0.64	0.92	0.79	0.85
Total	0.81	0.54	0.65	0.92	0.81	0.86

Table 6. SpecSyn 's specification synthesis ability compare to PracExtractor

that has (0.81) Precision only. This indicates that 92% of the predicted relevant results were accurate for SpecSyn while only 81% of the predicated relevant results were accurate for PracExtractor. SpecSyn has Recall value of 0.81 compared to a mere 0.54 Recall value for PracExtractor. Which signifies that SpecSyn could correctly identify 81% of the relevant cases where PracExtractor could only correctly identify 54% showing a huge improvement (1.5×) in detecting relevant cases. Overall, SpecSyn has a F1 score of 0.86 compared to F1 score of 0.65 by PracExtractor which clearly explains that SpecSyn performed much better than the existing state of the art model.

	True Positive	False Positive	False Negative
PracExtractor	82%	18%	73%
SpecSyn	94%	6%	21%

Table 7. Confusion matrix of specification detection capabilities

Table 7 shows confusion matrix for specification detection between SpecSyn and PracExtractor. It shows SpecSyn provides better true positive and false positive compare to PracExtractor. In terms of true positive it is 12% better in prediction. PracExtractor gives much high false positive (3x) prediction compare to SpecSyn . In terms of false negative detection, SpecSyn gives 21% compare to 73% of other tool.

Extraction Type	Detection			Generation
	Precision	Recall	F1 Score	
Simple	0.92	0.81	0.86	100%
$\text{Complex}_{\text{Single}}$	0.81	0.70	0.75	87%
$\text{Complex}_{\text{Multi}}$	0.73	0.64	0.68	83%

Table 8. SpecSyn 's Synthesis capability across different specification extraction type

Table 8 demonstrates the specification synthesis results across different specification extraction type. The F1 score for specification detection in *Simple* extraction type is 0.86 where the maximum F1 score is 0.75 for the *Complex* category. The Precision in *Simple* extraction type is 0.92 that implies it is very good at detecting specification. Among *Complex* type $Complex_{Single}$ can detect specification with Precision of 0.81 where $Complex_{Multi}$ can detect specification with a Precision score of 0.73. However, the later category has higher F1 score than that of other. In terms of specification synthesis, 100% specification can be synthesized in *Simple* category given they are correctly detected. Since, when the model is accurate at detecting specification the chances of synthesize a specification is higher. Moreover, in *Simple* cases average synthesized specification length is 3. Therefore, it is able to achieve 100% accuracy in terms of specification generation. On the other hand, $Complex_{Single}$ synthesize 4% more specification compare to $Complex_{Multi}$.

Category	**Precision**	**Recall**	**F1 score**
Quantitative	0.95	0.82	0.88
Utilization	0.8	0.89	0.84
Interrelation	0.8	0.89	0.84
Attribute	0.95	0.90	0.92
Generic	1.0	0.71	0.83

Table 9. SpecSyn Synthesis ability across different specification categories

3.2 Performance of Specification Categorization

Table 9 shows Precision, Recall and F1 score across different categories of specifications - namely for Quantitative, Utilization, Interrelation, Attribute, Generic categories. For Quantitative category, SpecSyn had a Precision of 0.95 and Recall of 0.82 which gave a F1 score of 0.88. For both Utilization and Interrelation categories, Precision value was 0.8 while Recall was 0.89 and F1 score was 0.84. For Attribute category, SpecSyn performed better than previous categories with very high Precision of 0.95 and Recall value of 0.9 which gave the highest F1 score of 0.92 across any categories. However, the highest Precision was achieved for Generic categories with 1.0 value that is all the predicted relevant results were accurate for SpecSyn . However, for Generic category Recall value was a bit lower than other categories with 71% of the relevant cases detected. So, F1 score was lower than other categories with a value of 0.83

3.3 Characterization of Models

We have used three different pretrained models and collected their Precision, Recall and F1 score. As we can see in Table 10, BERT [12] have the highest Precision among the models we have tested. BERT has Precision value of 0.92, compared to Precision value of 0.91 for both $\text{BERT}_{\text{Tiny}}$ and GPT [43] models. In case of identifying the fraction of relevant items that can be retrieved, i.e. the Recall score,

Language Model	Precision	Recall	F1 score
$BERT_{Tiny}$	0.91	0.82	0.86
GPT[43]	0.91	0.80	0.85
BERT	0.92	0.81	0.86

Table 10. Model characterization with pre-trained language models

$BERT_{Tiny}$ performs slightly higher 0.82 than the BERT model 0.81. GPT model has the lowest Recall score of 0.80 among these three models. Combining these two score, the harmonic mean of Precision and Recall, known as F1 score, shows that both BERT and $BERT_{Tiny}$ models perform better with F1 score of 0.86 compared to the GPT model which has a F1 score of 0.85. However, the performance of each of these three models is not significantly different from one another.

4 RELATED WORK

Software configuration and specification extraction with NLP. Numerous studies have addressed the challenge of effectively diagnosing and solving software configuration problems. One line of research focuses on using static analysis techniques to identify configuration errors before they result in system failures [15, 32, 41]. Other works, such as [22] and [4], aim to enhance system observability to detect configuration errors in situ. Some approaches propose proactive methods to detect and troubleshoot customer issues, such as [23] and [58], which introduce early detection systems to prevent configuration errors from causing significant damage. Online error detection systems based on context have been proposed in [63], while [65] proposes a misconfiguration detection system based on system environment and correlation information. Shared knowledge or event traces have been used to diagnose misconfigurations in [2] and [62], respectively. An automated troubleshooting approach based on dynamic information flow analysis is presented in [6], while [39] proposes a parallelized approach to information flow queries. Precomputing approaches to possible configuration error diagnoses have been proposed in [40]. [52] proposes a troubleshooting approach based on peer pressure, while [53] suggests a state-based approach to change and configuration management. On the other hand, [54] proposes a search-based approach to configuration debugging, and [66] introduces an automated diagnosis system for configuration errors. In addition to technical approaches, some papers emphasize the importance of not blaming users for configuration errors and instead focusing on developing better tools and processes to prevent them, such as in [59]. Probabilistic approaches have also been proposed to learn configuration file languages and identify and diagnose configuration problems, as demonstrated in [45]. Similarly, [44] presents an association rule learning-based approach to synthesize configuration file specifications from a set of example configurations. Also, [47] proposes a causality analysis-based technique to identify the root cause of configuration errors and automate the configuration management process.

One particularly influential work in this area is the PracExtractor [56], which employs natural language processing to analyze specific configurations and convert them into specifications to identify potential system admin flaws. However, PracExtractor has limitations, such as inflexibility and low generalization ability due to the specific format required for accurate extraction. Additionally, PracExtractor struggles with large paragraph settings, which our SpecSyn system seeks to improve upon. Other research efforts have also explored methods for inferring specifications from text [8, 13, 28, 35, 49, 50, 55, 64, 67, 68], but they too have limitations. Our work builds on these prior efforts by leveraging their insights to provide a more accurate and effective model for specification extraction.

Specification extraction using Knowledge Base. The use of a Knowledge Base (KB) has been explored in prior research for specification extraction, as seen in ConfSeer[36]. However, similar to PracExtractor, this approach has its limitations. ConfSeer relies on a KB to analyze potential configuration issues, which can make it feel more like a search engine. While this approach has its benefits, it can also limit flexibility. SpecSyn aims to address this limitation by analyzing unstructured data to provide a greater variety of specifications. Other systems have been proposed to assist with finding configurations, such as[52, 54], but they come with their own issues, such as high overheads and requirements for large datasets. Associated rule learning has been used in previous studies to address dataset issues[7, 19, 24, 29].

BERT based extraction and NLP In contrast to previous studies, we employed and fine-tuned the BiDirectional Encoder Representations and Transformers model, or BERT[12], to gain a better understanding and improve the training of our dataset. BERT is a new natural language processing model that offers a more in-depth and refined fine-tuning technique. However, there are some limitations associated with using BERT. As it was developed in 2018, it is still a relatively new model and may not be fully developed with regard to training sets. Additionally, it can be expensive to train and result in slower training times due to its many weights[37]. Despite these limitations, BERT does an excellent job of processing specific input and producing output with new specifications, providing great flexibility as it eliminates the need for a KB as used in ConfSeer and can expand upon the findings of the PracExtractor system.

In recent times, deep learning is heavily used to address diverse system-related issues, such as program synthesis [30, 31], partial program correction [18], bug fixing [26] etc. Several studies have employed natural language processing (NLP) to inspect and analyze software configurations, particularly in the context of security analysis[46]. This work focused on analyzing two security systems, CIS and Siemens, and trained two models, LDA and BERT, which is the model used in our study. However, this study had certain limitations, such as the narrow focus on only two types of security systems, which could limit the diversity of the training data. Moreover, the study found that the BERT model, according to their metric of measurement, was not accurate enough to detect misconfigurations. Nevertheless, this study provided a useful baseline for identifying configuration issues using NLP and made their dataset publicly available on Kaggle, facilitating further research in the field.

Recent studies have also investigated software configuration and misconfiguration using state analysis techniques[25]. In[25], the authors developed ConfDetect, a system that analyzes log files and

ranks them based on specific criteria, which can effectively predict misconfiguration errors. The system also utilizes NLP to extract logs. The authors reported substantial accuracy in diagnosing configuration errors. However, the system was only tested on three different systems, whereas our study has more variety in terms of data testing. Additionally, ConfDetect utilizes a knowledge base to detect configuration errors, which could limit its flexibility. Despite these limitations, the study provides valuable insights into finding proper software misconfiguration.

5 DISCUSSION AND FUTURE WORK

In this section, we aim to highlight some potential observations and discuss them in detail.

Why use LSTM base decoder for generating specifications while Transformer base decoder is considered as the state-of-the-art? The Transformer architecture offers advantages over the LSTM-based architecture in two scenarios: first, when dealing with a tremendously large dataset (i.e. 100GB) that requires parallel training without any previous recurrence dependency, and second, when handling very long sequences [51]. In our specific case, the dataset that we use is comparatively very small, and the sequence to be generated is also relatively short. As a result, the utilization of a Transformer-based decoder over an LSTM-based decoder will not bring any benefits.

Why applying data composition instead of passing original text? Data composition helps the model to generalize better. While it is feasible to train a model without data composition and achieve similar performance results, doing so would require a larger quantity of data and can be left for future exploration.

How long sequence has been used to synthesize specification? In this study, we limit ourselves to check one sentence for *Simple* extraction type and two sentences for *Complex* extraction type. Our findings indicate that two sentences adequately cover the majority of *Complex* extraction type specifications. Furthermore, our sample sentences have an average length of approximately 150 characters, which implies that we check a maximum of around 300 characters for *Complex* extraction. We also train our model with a maximum of two sentences long samples. As a result, the model's performance will be poor for longer sequences than it is trained on. Hence, it would require more data samples and a potentially larger parametric model architecture to overcome this issue. Therefore, we keep this as a future work. It would be valuable to investigate very long sequences and conduct a sequence length sensitivity analysis.

Furthermore, regarding comments written in software source code, it is possible to parse a greater number of software programs. Additionally, examining the neighboring source code can provide a better understanding of the context of the comments. For extracting specification from other online discussion forums, a more exhaustive search could be conducted to mine a larger set of specifications. However, this is beyond the scope and context of our current project.

6 CONCLUSION

In this paper we introduces a deep learning-based framework for automatic specification synthesis using a large language model. To the best of our knowledge, this is the **first** work to utilize a large language model to understand natural language context and synthesize specifications. We formulate specification synthesis task as a sequence learning problem and integrate BERT, a pre-trained large language model for this purpose. Our proposed framework SpecSyn outperforms prior state-of-the-art by 21% in terms of F1 score. This work opens up new direction to synthesize specification and can be extended for other similar system-related works as well.

REFERENCES

[1] 2007. *SOSP '07: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA). Association for Computing Machinery, New York, NY, USA.

[2] Bhavish Agarwal, Ranjita Bhagwan, Tathagata Das, Siddharth Eswaran, Venkata N Padmanabhan, and Geoffrey M Voelker. 2009. NetPrints: Diagnosing Home Network Misconfigurations Using Shared Knowledge.. In *NSDI*, Vol. 9. 349–364.

[3] R. Arumugam and R. Shanmugamani. 2018. *Hands-On Natural Language Processing with Python: A practical guide to applying deep learning architectures to your NLP applications.* Packt Publishing. https://books.google.com/books?id=ipplDwAAQBAJ

[4] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12).* 307–320.

[5] Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10).* USENIX Association, Vancouver, BC. https://www.usenix.org/conference/osdi10/automating-configuration-troubleshooting-dynamic-information-flow-analysis

[6] Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis.. In *OSDI*, Vol. 10. 1–14.

[7] Jay Ayres, Jason Flannick, Johannes Gehrke, and Tomi Yiu. 2002. Sequential pattern mining using a bitmap representation. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining.* 429–435.

[8] Xu Chen, Yun Mao, Z Morley Mao, and Jacobus Van der Merwe. 2010. Declarative configuration management for complex and dynamic networks. In *Proceedings of the 6th International COnference.* 1–12.

[9] Cloudflare. 2019. Facebook blames a server configuration change for yesterday's outage. (2019). https://blog.cloudflare.com/october-2021-facebook-outage/

[10] CNN. 2017. Here's why you may have had internet problems today. (2017). https://money.cnn.com/2017/11/06/technology/business/internet-outage-comcast-level-3/index.html

[11] Marie-Catherine de Marneffe, Timothy Dozat, Natalia Silveira, Katri Haverinen, Filip Ginter, Joakim Nivre, and Christopher D. Manning. 2014. Universal Stanford dependencies: A cross-linguistic typology. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14).* European Language Resources Association (ELRA), Reykjavik, Iceland, 4585–4592. http://www.lrec-conf.org/proceedings/lrec2014/pdf/1062_Paper.pdf

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. https://doi.org/10.48550/ARXIV.1810.04805

[13] William Enck, Thomas Moyer, Patrick McDaniel, Subhabrata Sen, Panagiotis Sebos, Sylke Spoerel, Albert Greenberg, Yu-Wei Eric Sung, Sanjay Rao, and William Aiello. 2009. Configuration management at massive scale: System design and experience. *IEEE Journal on Selected Areas in Communications* 27, 3 (2009), 323–335.

[14] Birhanu Eshete, Adolfo Villafiorita, and Komminist Weldemariam. 2011. Early detection of security misconfiguration vulnerabilities in web applications. In *2011 Sixth International Conference on Availability, Reliability and Security.* IEEE, 169–174.

[15] Nick Feamster and Hari Balakrishnan. 2005. Detecting BGP configuration faults with static analysis. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2.* 43–56.

[16] fiercewireless. 2017. AT&T failure was caused by a system configuration change. (2017). https://www.fiercewireless.com/wireless/at-t-s-911-outage-result-mistakes-made-by-at-t-fcc-s-pai-says

[17] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16).* Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/2987550.2987583

[18] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*, Vol. 31.

[19] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. 2007. Frequent pattern mining: current status and future directions. *Data mining and knowledge discovery* 15, 1 (2007), 55–86.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

[21] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (nov 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[22] Peng Huang, Chuanxiong Guo, Jacob R Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and enhancing in situ system observability for failure detection. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 1–16.

[23] Yu Jin, Nick Duffield, Alexandre Gerber, Patrick Haffner, Subhabrata Sen, and Zhi-Li Zhang. 2010. Nevermind, the problem is already fixed: proactively detecting and troubleshooting customer dsl problems. In *Proceedings of the 6th International COnference*. 1–12.

[24] Pat Langley and Herbert A Simon. 1995. Applications of machine learning and rule induction. *Commun. ACM* 38, 11 (1995), 54–64.

[25] Ke Li, Yuan Xue, Yujie Shao, Bing Su, Yu-an Tan, and Jingjing Hu. 2021. Software Misconfiguration Troubleshooting Based on State Analysis. In *2021 IEEE Sixth International Conference on Data Science in Cyberspace (DSC)*. IEEE, 361–366.

[26] Yi Li, Shaohua Wang, and Tien N Nguyen. 2022. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering*. 511–523.

[27] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. 2017. Focal Loss for Dense Object Detection. *CoRR* abs/1708.02002 (2017). arXiv:1708.02002 http://arxiv.org/abs/1708.02002

[28] Boon Thau Loo, Joseph M Hellerstein, Ion Stoica, and Raghu Ramakrishnan. 2005. Declarative routing: extensible routing with declarative queries. *ACM SIGCOMM Computer Communication Review* 35, 4 (2005), 289–300.

[29] Nizar R Mabroukeh and Christie I Ezeife. 2010. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys (CSUR)* 43, 1 (2010), 1–41.

[30] Shantanu Mandal, Todd Anderson, Javier Turek, Justin Gottschlich, Shengtian Zhou, and Abdullah Muzahid. 2021. Learning Fitness Functions for Machine Programming. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 139–155. https://proceedings.mlsys.org/paper_files/paper/2021/file/32bb90e8976aab5298d5da10fe66f21d-Paper.pdf

[31] Shantanu Mandal, Todd A. Anderson, Javier Turek, Justin Gottschlich, and Abdullah Muzahid. 2022. Synthesizing Programs with Continuous Optimization. arXiv:2211.00828 [cs.AI]

[32] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 140–151. https://doi.org/10.1145/2568225.2568283

[33] Kiran Nagaraja, Fabio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. 2004. Understanding and Dealing with Operator Mistakes in Internet Services. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/osdi-04/understanding-and-dealing-operator-mistakes-internet-services

[34] David G. Novick and Karen Ward. 2006. Why Don't People Read the Manual?. In *Proceedings of the 24th Annual ACM International Conference on Design of Communication* (Myrtle Beach, SC, USA) *(SIGDOC '06)*. Association for Computing Machinery, New York, NY, USA, 11–18. https://doi.org/10.1145/1166324.1166329

[35] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring method specifications from natural language API descriptions. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 815–825.

[36] Rahul Potharaju, Joseph Chan, Luhui Hu, Cristina Nita-Rotaru, Mingshi Wang, Liyuan Zhang, and Navendu Jain. 2015. ConfSeer: leveraging customer support knowledge bases for automated misconfiguration detection. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1828–1839.

[37] ProjectPro. 2022. Bert NLP model explained for complete beginners. https://www.projectpro.io/article/bert-nlp-model-explained/558

[38] XiPeng Qiu, TianXiang Sun, YiGe Xu, YunFan Shao, Ning Dai, and XuanJing Huang. 2020. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences* 63, 10 (Sept. 2020), 1872–1897. https://doi.org/10.1007/s11431-020-1647-3

[39] Andrew Quinn, David Devecsery, Peter M Chen, and Jason Flinn. 2016. JetStream: Cluster-Scale Parallelization of Information Flow Queries.. In *OSDI*, Vol. 16. 451–466.

[40] Ariel Rabkin and Randy Katz. 2011. Precomputing possible configuration error diagnoses. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 193–202.

[41] Ariel Rabkin and Randy Katz. 2011. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering*. 131–140.

[42] Ariel Rabkin and Randy Howard Katz. 2013. How Hadoop Clusters Break. *IEEE Software* 30, 4 (2013), 88–94. https://doi.org/10.1109/MS.2012.73

[43] Alec Radford and Karthik Narasimhan. 2018. Improving Language Understanding by Generative Pre-Training.

[44] Mark Santolucito, Ennan Zhai, Rahul Dhodapkar, Aaron Shim, and Ruzica Piskac. 2017. Synthesizing Configuration File Specifications with Association Rule Learning. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 64 (oct 2017), 20 pages. https://doi.org/10.1145/3133888

[45] Mark Santolucito, Ennan Zhai, and Ruzica Piskac. 2016. Probabilistic automated language learning for configuration files. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*. Springer, 80–87.

[46] Patrick Stöckle, Theresa Wasserer, Bernd Grobauer, and Alexander Pretschner. 2022. Automated Identification of Security-Relevant Configuration Settings Using NLP. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[47] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. 2007. AutoBash: Improving configuration management with operating system causality analysis. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 237–250.

[48] Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. 2019. How to Fine-Tune BERT for Text Classification? *CoRR* abs/1905.05583 (2019). arXiv:1905.05583 http://arxiv.org/abs/1905.05583

[49] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or bad comments?*. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 145–158.

[50] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd international conference on software engineering*. 11–20.

[51] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762

[52] Helen J Wang, John C Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. 2004. Automatic Misconfiguration Troubleshooting with PeerPressure.. In *OSDI*, Vol. 4. 245–257.

[53] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J Wang, Chun Yuan, and Zheng Zhang. 2004. Strider: A black-box, state-based approach to change and configuration management and support. *Science of Computer Programming* 53, 2 (2004), 143–164.

[54] Andrew Whitaker, Richard S Cox, Steven D Gribble, et al. 2004. Configuration Debugging as Search: Finding the Needle in the Haystack.. In *OSDI*, Vol. 4. 6–6.

[55] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. 2015. Dase: Document-assisted symbolic execution for improving automated software testing. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 620–631.

[56] Chengcheng Xiang, Haochen Huang, Andrew Yoo, Yuanyuan Zhou, and Shankar Pasupathy. 2020. PracExtractor: Extracting Configuration Good Practices from Manuals to Detect Server Misconfigurations. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 265–280. https://www.usenix.org/conference/atc20/presentation/xiang

[57] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings (2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings)*. Association for Computing Machinery, Inc, 307–319. https://doi.org/10.1145/2786805.2786852 Publisher Copyright: © 2015 ACM.; 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 ; Conference date: 30-08-2015 Through 04-09-2015.

[58] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early Detection of Configuration Errors to Reduce Failure Damage.. In *OSDI*, Vol. 10. 3026877–3026925.

[59] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do Not Blame Users for Misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 244–259. https://doi.org/10.1145/2517349.2522727

[60] Tianyin Xu and Yuanyuan Zhou. 2015. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Comput. Surv.* 47, 4 (2015), 70:1–70:41. https://doi.org/10.1145/2791577

[61] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 159–172. https://doi.org/10.1145/2043556.2043572

[62] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. 2006. Automated known problem diagnosis with event traces. *ACM SIGOPS Operating Systems Review* 40, 4 (2006), 375–388.

[63] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. 2011. Context-based online configuration-error detection. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*. 28–28.

[64] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao, and Feng Qin. 2016. Automatic model generation from documentation for Java API functions. In *Proceedings of the 38th International Conference on Software Engineering*. 380–391.

[65] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. Encore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 687–700.

[66] Sai Zhang and Michael D Ernst. 2013. Automated diagnosis of software configuration errors. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 312–321.

[67] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. Inferring resource specifications from natural language API documentation. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 307–318.

[68] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 27–37.