Metadata of the chapter that will be visualized in SpringerLink

Book Title	End-User Development	
Series Title		
Chapter Title	End-User Customization of Trigger-Action Rules Through Fine-Tuned LLMs	
Copyright Year	2025	
Copyright HolderName	The Author(s), under exclusive license to Springer Nature Switzerland AG	

Corresponding Author	Family Name	**Cimino**
	Particle	
	Given Name	**Gaetano**
	Prefix	
	Suffix	
	Role	
	Division	
	Organization	University of Salerno
	Address	Fisciano, Italy
	Division	
	Organization	University of British Columbia
	Address	Vancouver, Canada
	Email	gcimino@unisa.it
		gaetano.cimino@ubc.ca
	ORCID	http://orcid.org/0000-0001-8061-7104
Author	Family Name	**Deufemia**
	Particle	
	Given Name	**Vincenzo**
	Prefix	
	Suffix	
	Role	
	Division	
	Organization	University of Salerno
	Address	Fisciano, Italy
	Email	deufemia@unisa.it
	ORCID	http://orcid.org/0000-0002-6711-3590

Abstract

While Trigger-Action Platforms (TAPs) provide an effective solution for end-users to automate interactions between smart devices and online services through customizable rules, their flexibility is often limited by restricted access to source code, confining users to predefined templates and basic configurations, or the need for programming expertise when modifications are allowed. To address this challenge, we propose a methodology using fine-tuned autoregressive Large Language Models (LLMs) that translates natural language descriptions into executable rule code. This approach removes the need for manual coding, reducing the technical barrier and enabling users to express their automation intents more intuitively. To evaluate our approach, we used a dataset of trigger-action rules from the If-This-Then-That (IFTTT) platform along with their natural language descriptions. These descriptions underwent refinement through an LLM-based preprocessing step and were then used to evaluate the performance of eight open-source LLMs in generating syntactically correct and functionally equivalent rule implementations. Among

these models, Codestral achieved the best results, obtaining a ROUGE-L score of 63% and a METEOR score of 54%.

Keywords (separated by '-')	Trigger-Action Platforms - GenAI - No-code Programming - Rule Customization - User Intent Modeling

End-User Customization of Trigger-Action Rules Through Fine-Tuned LLMs

Gaetano Cimino[1,2(✉)] and Vincenzo Deufemia[1]

[1] University of Salerno, Fisciano, Italy
{gcimino,deufemia}@unisa.it
[2] University of British Columbia, Vancouver, Canada
gaetano.cimino@ubc.ca

Abstract. While Trigger-Action Platforms (TAPs) provide an effective solution for end-users to automate interactions between smart devices and online services through customizable rules, their flexibility is often limited by restricted access to source code, confining users to predefined templates and basic configurations, or the need for programming expertise when modifications are allowed. To address this challenge, we propose a methodology using fine-tuned autoregressive Large Language Models (LLMs) that translates natural language descriptions into executable rule code. This approach removes the need for manual coding, reducing the technical barrier and enabling users to express their automation intents more intuitively. To evaluate our approach, we used a dataset of trigger-action rules from the If-This-Then-That (IFTTT) platform along with their natural language descriptions. These descriptions underwent refinement through an LLM-based preprocessing step and were then used to evaluate the performance of eight open-source LLMs in generating syntactically correct and functionally equivalent rule implementations. Among these models, Codestral achieved the best results, obtaining a ROUGE-L score of 63% and a METEOR score of 54%.

Keywords: Trigger-Action Platforms · GenAI · No-code Programming · Rule Customization · User Intent Modeling

1 Introduction

Within the domain of End-User Development (EUD) [2,6,36], Trigger-Action Platforms (TAPs) enable the automation of interactions between smart devices and online services, allowing users—regardless of their programming expertise—to define rules that associate events (triggers) with corresponding actions [16, 24]. These platforms operate based on *trigger-action programming* [23,42], a paradigm structured around an IF-THEN format. In this framework, a *trigger* is defined as an event or condition that occurs within a system, and this event serves to activate the rule. The *action*, in turn, represents the behavior or response that

C. Santoro et al. (Eds.): IS-EUD 2025, LNCS 15713, pp. 1–19, 2025.
https://doi.org/10.1007/978-3-031-95452-8_2

is executed once the trigger is activated. To structure these rules, TAPs rely on an abstract representation of service providers, commonly referred to as *channels* [7,9]. A channel acts as an interface to various services, enabling TAPs to detect triggers and perform the associated actions. Channels are integral in ensuring the interaction between the user's defined rules and the corresponding external services or devices.

A typical IF-THEN rule in a TAP, as illustrated in Fig. 1, could be as follows: **IF** "*the user receives an email*" (with the trigger channel being *Office365*), **THEN** "*post a message to a specific Slack channel*" (with the action channel being *Slack*). This rule automates the process of notifying a designated Slack channel whenever the user receives an email in their Office365 inbox, thus streamlining the communication and task management process. Furthermore, both the trigger and action components may incorporate specific channel-related details, often referred to as *fields* [47], such as the subject of the email for Office365 and the target channel for Slack. Through the use of such trigger-action rules, TAPs provide a flexible and efficient means of automating workflows, allowing users to create meaningful interactions between different devices and services without requiring advanced technical expertise.

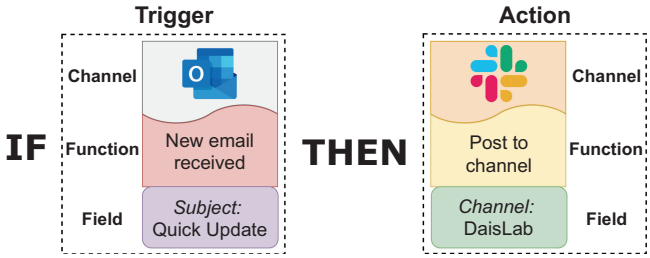

Fig. 1. Example of a trigger-action rule.

TAPs differ in the level of access they provide for rule definition, directly impacting how users can interact with and modify automation processes. Certain platforms, such as Alexa, restrict access to rule source code, making tasks like debugging or verification more challenging. In contrast, others allow users to adjust rule behavior by directly modifying the code. For instance, the If-This-Then-That (IFTTT[1]) platform enables expert users to refine the behavior of a rule through the use of code, commonly referred to as **filter code** [15]. This JavaScript code snippet processes trigger fields to customize action fields or bypass an action event based on specific conditions. For example, if users wish to send a Slack message only when a new email with a subject containing the keyword "IFTTT" is received, they must implement the filter code shown in Fig. 2 [15]. The variable *Office365Mail.newEmail* holds the trigger fields, such as the subject, while *Slack.postToChannel* provides various functions, such as

[1] https://www.ifttt.com/.

setMessage(), to assign values to different action fields. The *skip()* function is used to bypass an action. Writing filter code can be challenging for non-expert users due to its technical complexity and the required programming skills. Consequently, non-expert users are often limited to predefined rule templates or basic configuration options, which constrains their ability to create complex and customized automation behaviors.

```
let str = Office365Mail.newEmail.Subject
if (str.indexOf("IFTTT") === -1) {
    Slack.postToChannel.skip()
} else {
    Slack.postToChannel.setMessage("Email " +
        Office365Mail.newEmail.Subject + " just received!")
}
```

Fig. 2. An example of filter code for the rule shown in Fig. 1.

Research efforts have aimed to simplify the rule definition process by developing methods that convert user intent, expressed in natural language, into structured rule components [22] or by recommending new automation rules based on those already defined by the user [44]. These approaches have proven effective in streamlining rule creation and enhancing user experience. However, their functionality is primarily limited to suggesting predefined values for channels, functionalities, and fields. They do not address the challenge of personalizing trigger-action rule behaviors through code customization.

To overcome the limitations of existing approaches, this paper introduces a novel no-code strategy that leverages Large Language Models (LLMs) to bridge the gap between natural language descriptions and executable rule code. Our goal is to empower end-users, particularly those without programming expertise, to define and customize trigger-action rules without manually writing code. This effort aligns with our main research question:

RQ. How effectively can LLMs generate source code for trigger-action rules based on natural language descriptions?

To answer the research question, we curated a dataset of trigger-action rules from IFTTT, accompanied by their corresponding natural language descriptions. These descriptions were refined using an LLM-based preprocessing step to enhance clarity and consistency before being used to fine-tune eight open-source LLMs. Our methodology employs a transformer-based architecture that incrementally generates source code tokens in a left-to-right sequence, progressively constructing a valid rule implementation based on user-provided descriptions. Model performance was assessed by evaluating the ability to generate syntactically correct and functionally equivalent rule implementations. Among the tested

models, `Codestral` achieved the best results, with a METEOR score of 54% and a ROUGE-L score of 63%.

The remainder of this paper is organized as follows. Section 2 reviews related work on natural language processing for automation rule generation and code synthesis. Section 3 formalizes the problem, outlining the challenges of translating natural language descriptions into executable trigger-action rules. Section 4 presents our proposed methodology, detailing the design of our LLM-based approach for rule code generation. Section 5 describes the evaluation framework, including dataset construction, model selection, and assessment metrics. Section 6 reports and discusses the experimental results, comparing the performance of different LLMs in generating rule-based code. Finally, Sect. 7 concludes the paper and outlines directions for future research.

2 Related Work

Several studies have examined methods to enhance the user-friendliness of TAPs. For instance, research in end-user programming [31,32] and visual rule-based systems [17] has focused on simplifying the creation of automation rules. However, these approaches still require manual configuration of rule components, which restricts flexibility and scalability. Recent advancements have explored the application of natural language processing techniques for automation [22,46], aiming to generate rule components from natural language descriptions. While these tools facilitate rule configuration, defining complex behaviors often necessitates writing source code, which remains inaccessible to non-expert users. Furthermore, other studies have investigated methods for recommending modifications to users to debug rule behaviors [18,48,49], with a particular focus on preventing unintended situations. However, these approaches primarily address specific issues in rule configurations rather than enabling users to fully utilize the platform's capabilities. In contrast, our solution facilitates customization by generating specific code, allowing even non-expert users to tailor rule behaviors while ensuring compliance with platform requirements.

The task of code generation from natural language descriptions has been extensively studied in recent years, particularly with the advent of large-scale neural models [30]. Early approaches to code generation relied on template-based methods [27] and rule-based systems [3], which required significant manual effort to define syntactic and semantic rules for mapping natural language to code. With the rise of deep learning, sequence-to-sequence models, initially designed for machine translation, were adapted to generate code from natural language inputs [34,45]. More recently, transformer-based models such as OpenAI Codex [14] and CodeT5 [43] have demonstrated significant improvements in code generation accuracy by leveraging pretraining on large-scale code repositories. Despite these advancements, existing work on code generation has primarily focused on general-purpose programming tasks such as function synthesis, SQL query generation, and API call prediction [14,37,50]. However, no prior research has explicitly addressed the problem of generating code for trigger-action rules.

The present work fills this gap by proposing a methodology that employs fine-tuned LLMs to automatically generate source code for trigger-action rules from natural language descriptions.

3 Problem Formulation

TAPs facilitate task automation by linking trigger events to corresponding action responses. Formally, let $\mathcal{C}$ be the set of available channels, where each channel $c \in \mathcal{C}$ corresponds to an entity responsible for managing services (e.g., Slack for messaging automation). Each channel $c \in \mathcal{C}$ is associated with two distinct sets of functionalities:

- $TF(c)$, denoting the set of **trigger** functionalities provided by the channel,
- $AF(c)$, defining the set of **action** functionalities supported by the channel.

These functionalities determine the conditions a channel can monitor or the operations it can execute, forming the basis for specifying trigger-action rule behaviors. Users configure automation rules by selecting channels and their respective functionalities from a catalog, which lists the available services.

A trigger-action rule can be formally defined as a pair comprising trigger and action components:

$$R = \big(\underbrace{\{(c_i, t_{c_i}, F(t_{c_i}))\}_{i=1}^{m}}_{trigger\ \ component}, \underbrace{\{(c_j, a_{c_j}, F(a_{c_j}))\}_{j=1}^{n}}_{action\ \ component} \big)$$

The *trigger component* consists of a set of m triggers, where each trigger is specified by selecting a channel $c_i \in C$ and a functionality $t_{c_i} \in TF(c_i)$, which determines the condition that activates the rule. Additional configuration may be required through a set of fields $F(t_{c_i}) = \{f_{t^1_{c_i}}, \ldots, f_{t^l_{c_i}}\}$, where each $f_{t^k_{c_i}}$ represents a parameter for defining the trigger's behavior. For instance, in the rule depicted in Fig. 1, the trigger channel is *Office365*, and the functionality is "*New email received*". Further configuration is required to define the trigger's behavior, such as specifying the *subject*, *sender*, or *time* fields, which establish the conditions under which the rule is activated.

Similarly, the *action component* consists of a set of n actions, where each action is defined by choosing a channel $c_j \in C$ and a functionality $a_{c_j} \in AF(c_j)$, specifying the system's response when the rule is activated. The action functionality may also require configuration through a set of fields $F(a_{c_j}) = \{f_{a^1_{c_j}}, \ldots, f_{a^u_{c_j}}\}$, where each $f_{a^k_{c_j}}$ is a parameter necessary to define the action's behavior. In the rule depicted in Fig. 1, the action channel is *Slack*, and the functionality is "*Post to Channel*", which triggers the system to post a message to a Slack channel. The fields for this action include *message*, *title*, and *channel*, which define the content of the message, its title, and the target channel in Slack where the message will be posted.

The task addressed in this study involves automatically translating a natural language description d of a trigger-action rule R into a corresponding code snippet $\hat{y}_R$. For example, a possible description for generating the code depicted in

Fig. 2, aimed at personalizing the behavior of the rule shown in Fig. 1, is as follows: "*I would like to send a Slack message only when a new email contains the keyword IFTTT in the subject*". This translation is carried out by a code generation model $\mathcal{M}$, which leverages the intent expressed in d to produce syntactically and semantically meaningful rule implementations.

4 Methodology

The proposed methodology comprises two main steps: (*i*) Data Collection and Preprocessing and (*ii*) Fine-Tuning of the Rule Generation Model, as illustrated in Fig. 3. The first step involves gathering and refining pairs of natural language descriptions of trigger-action rule behaviors along with their corresponding code. The second step involves fine-tuning an autoregressive LLM by optimizing the conditional likelihood of the target code sequence. This process ensures that the model effectively learns to map natural language descriptions to their corresponding executable representations.

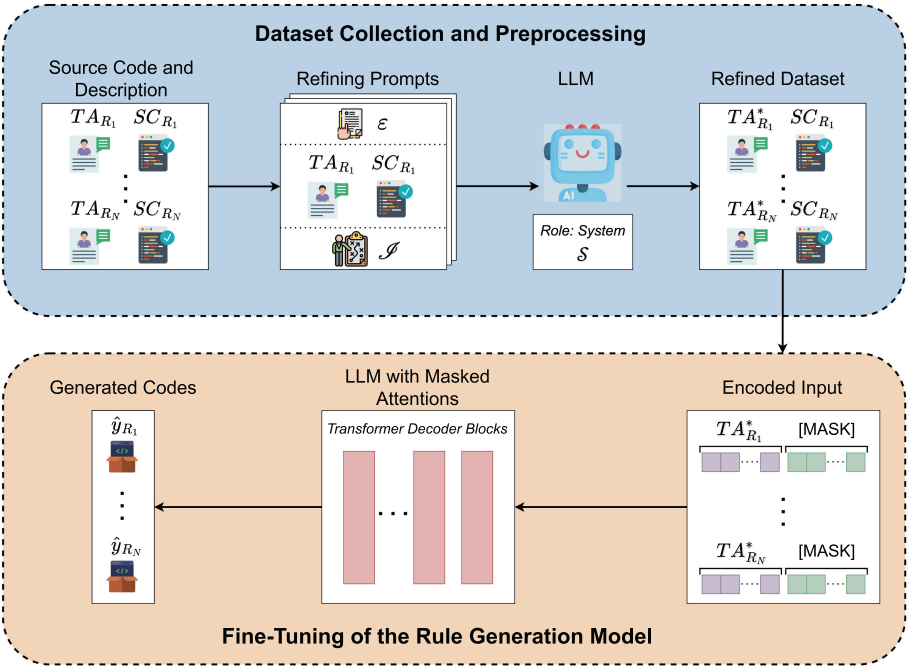

Fig. 3. Overview of the proposed methodology, which includes dataset collection and preprocessing, followed by fine-tuning an LLM for rule code generation.

4.1 Dataset Collection and Preprocessing

For the construction of our dataset, we utilized the IFTTT platform, which offers a key advantage: when defining a rule, users can provide a natural language description of its behavior. This feature serves multiple purposes, enabling users to recall the functionality of their defined rules and allowing others to understand the behavior of a rule created by someone else. Once a rule is established, users have the option to share it within a public catalog, facilitating the activation of predefined rules without requiring manual reconstruction. The catalog can be searched using a description of the desired rule, which is then matched against existing rule descriptions to filter the results. Consequently, these descriptions can be considered as user-expressed intents in natural language, making them particularly valuable for our code generation task.

We collected 615 pairs of natural language descriptions and their corresponding filter codes from IFTTT through the crawler used in [15]. However, as noted in previous studies [8,10,11], user-provided descriptions may be unclear or inconsistent, failing to accurately reflect the defined rule. Since our objective is to obtain a precise source code representation that fully captures the rule's components and fields, we assume that users have complete awareness of the rule they intend to create. In general, the problem of verifying the consistency between user-defined descriptions and actual rule components has been previously investigated [10,11].

To ensure consistency in the dataset, we introduce a refinement process that enhances user-provided descriptions based on their corresponding source code. As illustrated in Fig. 3, this process utilizes an LLM that applies a structured *refining prompt* for each rule.

Formally, given a pair $\langle TA_{R_i}, SC_{R_i} \rangle$, where TA_{R_i} represents the natural language description of rule R_i and SC_{R_i} denotes its corresponding source code, we employ a structured prompting approach to refine the description. The goal is to generate $TA^*_{R_i}$, a version of TA_{R_i} that better captures the semantics of the rule while preserving the original source code.

The refining prompt consists of two structured components:

- A contextual explanation $\mathcal{E}$ detailing the structure and purpose of trigger-action rules.
- A directive $\mathcal{I}$ instructing the model to either enhance the description by adding necessary details from SC_{R_i} or leave it unchanged if it is already sufficiently informative.

Additionally, the model can receive guidance through the *System* role, which specifies domain-specific aspects to emphasize, particularly the refinement of trigger-action rule descriptions. This structured prompting ensures that the generated description is both interpretable and complete.

The exact prompt used for the refinement process is as follows:

Refining Prompt

Context:
A trigger-action rule defines an automation where a specific condition (trigger) leads to an action being executed. These rules can be represented in code, specifying the conditions under which different actions should be performed or skipped.

Given the following code representation of a trigger-action rule:
[Rule Code]
And a description of its functionality:
[Rule Description]

Instruction:
Rewrite the description to fully reflect the logic in the given code, ensuring that it explicitly incorporates the evaluated values of relevant variables in the rule. Do not include the code or variable names in the rewritten description. If the provided description is already complete, return it unchanged.

Instead, the system role was set to emphasize expertise in refining and enhancing descriptions for clarity. The specific instruction given to the model was *"You are an expert in refining and enhancing descriptions of trigger-action rules to ensure clarity"*.

To illustrate the refinement process, consider the following rule description:

Original Description

"This rule will send advisories only during specified times on weekdays."

and the corresponding source code:

```
var Hour = Meta.currentUserTime.hour()
var Day = Meta.currentUserTime.day()

if (Day == 6 || Day == 7) {
  IfNotifications.sendNotification.skip()
}
else if (Hour < 7 || (Hour > 9 && Hour < 17) || Hour > 19) {
  IfNotifications.sendNotification.skip()
}
```

The use of the refining prompt enables the generation of a more detailed and specific version, as demonstrated in the refined description:

Refined Description

"This rule will send an alert for advisories between 7-9 a.m. and 5-7 p.m. on weekdays."

In this case, the model adds specific time frames (7–9 a.m. and 5–7 p.m.) to clarify the conditions under which the rule triggers an alert, enhancing the semantic clarity of the rule description.

4.2 Fine-Tuning of the Rule Generation Model

Fine-tuning an autoregressive LLM for trigger-action rule code generation consists of adapting a pretrained model to a domain-specific task, wherein a natural language description is mapped to an executable rule representation.

Given a dataset of description-code pairs $\mathcal{D} = \{\langle TA^*_{R_i}, SC_{R_i}\rangle\}_{i=1}^{N}$, where $TA^*_{R_i}$ is a refined natural language description of a trigger-action rule and SC_{R_i} is the corresponding source code representing the rule's behavior, the model applies a masking strategy to conceal the source code components SC_{R_i}, as illustrated in Fig. 3. Formally, a masking function Φ is applied to each pair, producing masked training examples:

$$\Phi(TA^*_{R_i}, SC_{R_i}) = (TA^*_{R_i}, \texttt{[MASK]})$$

where `[MASK]` represents the hidden source code that the model must predict. This ensures that the model learns to generate the code solely based on the natural language description.

We employ a transformer-based autoregressive model $\mathcal{M}_\theta$, initialized with pretrained parameters θ_0. Fine-tuning is performed by optimizing the conditional likelihood of the correct code sequence given the natural language description:

$$\theta^* = \arg\max_\theta \sum_{i=1}^{N} \log P_{\mathcal{M}_\theta}(\hat{y}_{R_i} \mid TA^*_{R_i})$$

where $P_{\mathcal{M}_\theta}(\hat{y}_{R_i} \mid TA^*_{R_i})$ is the probability assigned by the model to the sequence $\hat{y}_{R_i}$ given $TA^*_{R_i}$.

The model generates source code token-by-token using a left-to-right decoding process. Given a description $TA^*_{R_i}$, the generation follows:

$$P_{\mathcal{M}_\theta}(\hat{y}_{R_i} \mid TA^*_{R_i}) = \prod_{x=1}^{X} P_{\mathcal{M}_\theta}(\hat{y}_{R_{i,x}} \mid \hat{y}_{R_{i,<x}}, TA^*_{R_i})$$

where X is the length of the target code sequence and $\hat{y}_{R_{i,<x}} = (\hat{y}_{R_{i,1}}, \ldots, \hat{y}_{R_{i,x-1}})$ represents the previously generated tokens. Decoding can be performed via greedy search, beam search, or nucleus sampling.

Fine-tuning is conducted using a cross-entropy loss over the token sequences:

$$\mathcal{L}(\theta) = -\sum_{i=1}^{N}\sum_{x=1}^{X_i} \log P_{\mathcal{M}_\theta}(\hat{y}_{R_{i,x}} \mid \hat{y}_{R_{i,<x}}, TA^*_{R_i})$$

Each generated code $\hat{y}_{R_i}$ is compared against the reference implementation $y^*_{R_i}$ to assess the effectiveness of the model.

5 Evaluation Framework

This section outlines the evaluation framework of the proposed methodology. Specifically, it begins by defining the research questions that guide our investigation, followed by a detailed description of the evaluation setup, including model configurations and fine-tuning procedures. Finally, we present the metrics used to assess the performance of the generated code in terms of syntactic correctness and semantic fidelity.

5.1 Assessing Code Quality

To comprehensively answer the main research question, we define two subquestions that guide our investigation:

RQ1. How effectively can fine-tuned LLMs generate code snippets conform to the expected syntactic of the reference rule implementations?

RQ2. How effectively fine-tuned LLMs generate code snippets that capture the intended semantics of natural language descriptions?

By addressing these questions, we aim to evaluate whether the generated rule implementations not only adhere to the required syntax but also correctly execute the intended automation logic. This dual perspective is crucial, as syntactic correctness alone does not guarantee that the resulting code aligns with the expected behavior described by users.

5.2 Evaluation Setup

Although proprietary LLMs, such as GPT-4 [1] and Claude 3.7[2], offer advanced capabilities for code generation, their usage is often associated with considerable costs. This poses a potential barrier for certain users, particularly in contexts where TAPs allow the free definition of rules. In such cases, dependence on proprietary models may result in additional expenses that not all users may

[2] https://www.anthropic.com/news/claude-3-7-sonnet.

be willing to pay. To address this limitation, our methodology primarily leverages lightweight, open-source models that are freely available. This choice facilitates broader accessibility and enables deployment even on resource-constrained devices, such as mini PCs equipped with GPUs. Specifically, we fine-tuned four open-source LLMs: `Deepseek-LLM-7B` [20], `Mistral-7B-v0.2` [29], `Qwen2.5-7B` [38], and `Llama-2-7B` [41]. Furthermore, in light of the growing number of LLMs fine-tuned specifically for code generation [30], we also incorporated their corresponding fine-tuned code-generation variants: `Deepseek-Coder-6.7B` [25], `Codestral-7B-v0.1`[3], `Qwen2.5-Coder-7B` [28], and `CodeLlama-7B` [40]. The fine-tuning for all models was conducted with specific configurations to optimize performance while maintaining computational efficiency. During inference, we employed greedy decoding, generating a single code for each natural language description.

To save memory and boost efficiency, we used 4-bit quantization with a normalized floating-point format tailored for transformers. This ensures fast, stable computation with minimal precision loss. We also avoided introducing additional quantization overhead that could destabilize training. To optimize fine-tuning, we used LoRA [26], a memory-efficient method that updates a small parameter subset. We set rank to 16, used a scaling factor of 32, and applied 5% dropout to avoid overfitting. LoRA was adapted for causal language modeling.

For fine-tuning, we configured the training process to span 5 epochs, ensuring sufficient learning while preventing overfitting. The batch size per device was set to 2 to accommodate hardware constraints, and gradient accumulation was employed over 4 steps to simulate a larger batch size without excessive memory usage. A cosine learning rate scheduler was applied, starting with an initial learning rate of $1e-4$ that gradually decayed over time. To prevent excessive weight updates, weight decay regularization was introduced at a rate of 0.01, and gradient clipping was employed with a maximum norm of 1 to maintain stable training dynamics. Mixed precision training was enabled using *bfloat16* to enhance computational efficiency without compromising numerical accuracy.

We refined the natural language descriptions using `DeepSeek-R1` [19], which aided in enhancing the clarity and completeness of the descriptions.

The finalized dataset underwent manual validation and was subsequently partitioned into training, validation, and test sets to ensure robust evaluation. Initially, 80% of the data was allocated to training and 20% to testing. From the training set, an additional 10% was reserved for validation, enabling hyperparameter selection based on performance on unseen samples.

5.3 Evaluation Metrics

To evaluate the effectiveness of the proposed methodology for code generation, we employ a range of widely recognized metrics used in performance assessment:

– **BLEU** [35]: Assesses the precision of n-grams in the generated text by comparing them with reference texts, rewarding greater overlap.

[3] https://mistral.ai/en/news/codestral.

- **ROUGE-L** [33]: Measures the longest common subsequence between the generated and reference text, capturing structural coherence.
- **ROUGE-1** [33]: Calculates unigram recall, quantifying the overlap of individual tokens between the generated and reference code.
- **ROUGE-2** [33]: Computes bigram recall, assessing the overlap of two-token sequences to evaluate structural accuracy.
- **METEOR** [4]: Evaluates the similarity between generated and reference texts by considering factors such as synonymy, stemming, and word order, offering a more flexible approach to handle variations in the output.
- **CodeBERTScore** [51]: Leverages contextual embeddings from CodeBERT [21] to compute the similarity between generated and reference code, capturing semantic equivalence even when exact token matches are absent.
- **CodeBLEU** [39]: Extends BLEU with syntax-aware and data-flow matching techniques to better assess the quality of generated code, ensuring alignment with reference implementations beyond mere token overlap.

6 Results

In the following sections, we compare the strengths and limitations of various LLM models in generating rule implementations. This includes assessing how well the generated code matches the expected structure and syntax of the reference implementations (RQ1) and how effectively it captures the user-intended semantics as described in the natural language input (RQ2). The results from fine-tuned LLMs, based on both syntactic and semantic evaluation metrics, are presented in Table 1. This comparison helps to highlight the most effective models for automating the generation of trigger-action rules and provides insights into the trade-offs between syntactic correctness and semantic fidelity.

6.1 Answering RQ1: Syntactic Quality of Generated Code

To evaluate how well the generated code conforms to the expected syntax, we consider BLEU, ROUGE-L, ROUGE-1, and ROUGE-2. These metrics assess

Table 1. Performance of fine-tuned LLMs for trigger-action rule code generation.

LLM	Metrics						
	BLEU	ROUGE-L	ROUGE-1	ROUGE-2	METEOR	CodeBERTScore	CodeBLEU
Deepseek-LLM-7B	0.29	0.56	0.60	0.48	0.46	0.87	0.31
Deepseek-Coder-6.7B	0.27	0.53	0.57	0.44	0.41	0.86	0.30
Mistral-7B-v0.2	0.36	0.61	0.65	0.55	0.52	0.89	0.35
Codestral-7B-v0.1	**0.38**	**0.63**	**0.67**	**0.56**	**0.54**	**0.90**	**0.36**
Qwen2.5-7B	0.12	0.26	0.29	0.21	0.36	0.83	0.32
Qwen2.5-Coder-7B	0.07	0.22	0.25	0.18	0.34	0.81	0.28
Llama-2-7B	0.28	0.55	0.58	0.45	0.43	0.87	0.29
CodeLlama-7B	0.25	0.52	0.56	0.44	0.43	0.86	0.32

token-level similarity, measuring how closely the generated code matches reference implementations in terms of structure and sequence accuracy.

Table 1 shows that `Codestral-7B-v0.1` achieves the highest syntactic performance, with BLEU (0.38), ROUGE-L (0.63), ROUGE-1 (0.67), and ROUGE-2 (0.56), demonstrating strong adherence to reference syntax. `Mistral-7B-v0.2` follows closely, attaining a BLEU score of 0.36 and a ROUGE-L score of 0.61, indicating high syntactic fidelity. These models correctly generate rule structures, including conditional checks and function calls, with minimal errors.

`Deepseek-LLM-7B` and `Llama-2-7B` exhibit comparable performance, with BLEU scores of 0.29 and 0.28, respectively, and ROUGE-1 scores of 0.60 and 0.58. While these models effectively capture syntactic patterns, they occasionally produce minor syntax errors. For example, they might generate incorrect assignment operators (`if(Min = 0)` instead of `if(Min == 0)`) or omit necessary closing brackets – e.g., "`var depositAmountString = depositAmount.toString(`".

`Deepseek-Coder-6.7B` performs slightly below these models, with a BLEU score of 0.27 and a ROUGE-1 score of 0.57. This model tends to generate misplaced brackets in rule definitions and omit function calls. `CodeLlama-7B` and `Qwen2.5-7B` display weaker syntactic alignment, with BLEU scores of 0.25 and 0.12, respectively. `CodeLlama-7B` maintains moderate syntactic correctness with a ROUGE-L score of 0.52, but it often generates incorrectly formatted function calls, such as `PhoneCall.callMyPhone.setArgs(‘‘Remaining ’’ + 10 - Min + ‘‘ Minutes’’)`, which causes unintended string concatenation errors. `Qwen2.5-7B` struggles significantly, achieving a ROUGE-L score of only 0.26, indicating poor token alignment and structural coherence. Common errors from this model include misplaced parentheses, unterminated string literals, and invalid numeric operations, such as missing operators between numbers and variables, leading to syntax errors. Lastly, `Qwen2.5-Coder-7B` exhibits the lowest syntactic performance among the evaluated models, with a BLEU score of 0.07 and a ROUGE-L score of 0.22, highlighting its limited syntactic fidelity. This model frequently produces syntactically incorrect outputs, such as improper function calls, e.g., `Weather.currentWeatherAtTime_sunriseAt` instead of the correct `Weather.currentWeather[0].SunsetAt`.

In summary, the results achieved allow to answer our first RQ as follow:

> `Codestral-7B-v0.1` achieves the highest syntactic performance among the evaluated models, demonstrating strong alignment with the reference syntax. `Mistral-7B-v0.2` follows closely behind, while `Deepseek-LLM-7B` and `Llama-2-7B` effectively capture syntactic patterns, though they do not outperform `Codestral-7B-v0.1` or `Mistral-7B-v0.2`. In contrast, models such as `CodeLlama-7B` and `Qwen2.5-7B` exhibit weaker syntactic alignment, with `Qwen2.5-Coder-7B` displaying the lowest syntactic fidelity across all models assessed.

6.2 Answering RQ2: Semantic Quality of Generated Code

Beyond syntactic accuracy, it is essential to evaluate whether the generated code correctly implements the intended functionality described in natural language. To assess semantic fidelity, we employ METEOR, CodeBERTScore, and CodeBLEU, which capture both lexical and embedding-based similarity as well as functional correctness.

As presented in Table 1, `Codestral-7B-v0.1` achieves the highest semantic accuracy, with a METEOR score of 0.54, a CodeBERTScore score of 0.90, and a CodeBLEU score of 0.36, indicating its superior ability to capture the intended rule logic. `Mistral-7B-v0.2` follows closely, with a METEOR score of 0.52, a CodeBERTScore score of 0.89, and a CodeBLEU score of 0.35, further confirming its strong semantic alignment.

`Deepseek-LLM-7B` also demonstrates competitive performance, attaining a METEOR score of 0.46, a CodeBERTScore score of 0.87, and a CodeBLEU score of 0.31. This model occasionally generates rules with flawed conditions, such as incorrect time ranges, misplaced conditions, or logical inversions that lead to actions executing at unintended times. For example, one generated rule intended to skip notifications between 22:00 and 08:00 but instead allowed notifications to be sent during this period. In another instance, it generated a rule that misinterpreted the weekday condition, performing an action on weekends instead of weekdays.

Meanwhile, `Llama-2-7B` and `Deepseek-Coder-6.7B` achieve METEOR scores of 0.43 and 0.41, respectively, with corresponding CodeBERTScore scores of 0.87 and 0.86 and CodeBLEU scores of 0.29 and 0.30, reflecting moderate semantic correctness. These models have been seen to introduce errors such as omitting critical conditions or failing to initialize key variables, causing unintended behavior in rule execution. For instance, one rule generated by `Llama-2-7B` attempted to post a message to Slack without specifying the target channel, rendering the action incomplete.

`CodeLlama-7B`, despite exhibiting weaker syntactic performance, attains a CodeBLEU score of 0.32, suggesting that its generated code retains a degree of functional consistency. Interestingly, while `Qwen2.5-7B` also struggles with syntax, its CodeBLEU score of 0.32 indicates that, despite lower token alignment, its outputs still partially preserve the intended functionality. However, its METEOR score of 0.36 and CodeBERTScore score of 0.83 reflect comparatively lower semantic accuracy than most models. Nevertheless, it surpasses `Qwen2.5-Coder-7B`, which exhibits the lowest semantic accuracy. This model has been observed to generate rules with redundant or contradictory conditions, such as skipping an action both during and outside a designated time range, which contradicts the desired behavior. For example, one rule intended to skip light activation at night ended up skipping it both during the night and day. Furthermore, `Qwen2.5-Coder-7B` occasionally generates logic that attempted to modify undefined variables, leading to runtime failures; for example, one rule attempted to adjust a temperature setting using a non-existent variable reference.

In summary, the results achieved allow to answer our second RQ as follow:

> `Codestral-7B-v0.1` leads in semantic accuracy, showing strong functionality alignment. `Mistral-7B-v0.2` closely follows with robust alignment. `Deepseek-LLM-7B` performs well, while `Llama-2-7B` and `Deepseek-Coder-6.7B` show moderate correctness but don't surpass the top models. `CodeLlama-7B`, despite weaker syntax, retains functional consistency. `Qwen2.5-7B`, struggling with syntax, still shows some functionality, though with lower accuracy. `Qwen2.5-Coder-7B` has the weakest semantic performance, struggling to align with the intended rule logic.

6.3 Findings

Our experimental evaluation highlights several key insights regarding the efficacy of LLMs in generating trigger-action rule code from natural language descriptions. The following extended findings provide a deeper analysis of the observed trends and model performance characteristics:

1. **Consistency Across Metrics:** `Codestral-7B-v0.1` consistently outperforms other models across both syntactic and semantic evaluation metrics, establishing itself as the most effective solution for supporting non-expert users in configuring automation rules. Notably, its strong BLEU and ROUGE scores confirm its ability to replicate the structural coherence of reference implementations, while its METEOR, CodeBERTScore, and CodeBLEU results demonstrate superior alignment with user intentions.
2. **Trade-offs Between Syntax and Semantics:** While `Deepseek-LLM-7B` exhibits notable syntactic accuracy, its lower METEOR, CodeBERTScore, and CodeBLEU values indicate occasional misalignment with intended functionality. This suggests that models excelling in token alignment may sacrifice some semantic fidelity. Conversely, `CodeLlama-7B`, despite weaker token-level performance, achieves a comparable CodeBLEU score, underscoring its ability to preserve functional behavior despite reduced syntactic precision.
3. **Performance Variation by Complexity:** An analysis of rule complexity reveals that `Codestral-7B-v0.1` maintains stable performance across simple and complex rules alike. In contrast, models like `Deepseek-Coder-6.7B` and `Qwen2.5-Coder-7B` exhibit pronounced performance degradation for rules featuring multiple conditions, nested logic, or extended parameter configurations. This indicates that robust context modeling and improved sequence dependencies are key strengths of `Codestral-7B-v0.1`.
4. **Performance Discrepancy in Code-Specific Models:** Models fine-tuned specifically for code generation often exhibit lower performance compared to those trained on more diverse datasets. This trend is evident in models such as `Deepseek-Coder-6.7B`, which demonstrates lower semantic fidelity than its general-purpose counterpart, `Deepseek-LLM-7B`. A similar pattern emerges in the comparisons between `CodeLlama-7B` and `Llama-2-7B`, as well

as `Qwen2.5-Coder-7B` and `Qwen2.5-7B`. This discrepancy may arise from the distinctive nature of trigger-action rule code, which integrates domain-specific logic with unconventional syntax patterns. Models trained on a broader range of linguistic structures appear to generalize more effectively to these hybrid patterns. However, the Mistral family presents an exception to this trend. `Codestral-7B-v0.1`, a code-specialized variant, outperforms `Mistral-7B-v0.2` across all metrics. This suggests that while general-purpose models benefit from exposure to diverse linguistic structures, a sufficiently well-trained code-focused model can achieve superior results if it effectively captures both syntactic correctness and the nuanced reasoning required for rule logic. The advantage of `Codestral-7B-v0.1` may stem from its ability to preserve the strong language modeling capabilities of Mistral while incorporating a refined understanding of code structure, thereby facilitating a more effective alignment between textual descriptions and executable rules.

7 Conclusion and Future Work

This paper introduces a methodology that leverages LLMs to enable the no-code configuration of trigger-action rules on TAPs. By enabling end-users to generate rule implementations directly from natural language descriptions, our approach improves accessibility and reduces reliance on programming expertise. Through an extensive evaluation, we demonstrated that LLMs, particularly `Codestral`, achieve high performance in both syntactic correctness and semantic fidelity when generating executable rule code. These findings highlight the potential of LLMs in simplifying EUD and making automation more intuitive and adaptable for non-expert users.

Despite the demonstrated effectiveness of our approach, several limitations remain that warrant further investigation. Although the generated code snippets are structurally coherent and syntactically correct, they do not always fully capture the intended behavior specified by users—particularly in complex scenarios involving intricate conditional logic, interdependent triggers, or multi-step workflows. Additionally, while our refinement process enhances the clarity of rule descriptions, inconsistencies between user-provided inputs and generated outputs may still occur. Future research will explore several directions to address these limitations. First, integrating user feedback mechanisms into the code generation pipeline could improve alignment with user intent, allowing for iterative refinement of generated codes [5]. Second, enhancing the interpretability of generated codes through explainability techniques will be crucial for fostering trust and usability [9,12]. Third, expanding the evaluation to real-world user studies will provide deeper insights into the practical applicability and effectiveness of our approach in diverse automation contexts, such as in understanding and controlling security and privacy threats [13]. Finally, broadening the dataset used to fine-tune the models will be essential for improving generalization and handling a wider variety of user-defined automation scenarios.

References

1. Achiam, J., et al.: GPT-4 technical report. arXiv preprint arXiv:2303.08774 (2023)
2. Alhirabi, N., Rana, O., Perera, C.: Security and privacy requirements for the internet of things: a survey. ACM Trans. Internet Things **2**(1), 1–37 (2021)
3. Bajwa, I.S., Siddique, M.I., Choudhary, M.A.: Rule based production systems for automatic code generation in Java. In: Proceedings of 1st International Conference on Digital Information Management, pp. 300–305 (2006)
4. Banerjee, S., Lavie, A.: Meteor: an automatic metric for MT evaluation with improved correlation with human judgments. In: Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization, pp. 65–72 (2005)
5. Bang, Y., et al.: A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. arXiv preprint arXiv:2302.04023 (2023)
6. Barricelli, B.R., Cassano, F., Fogli, D., Piccinno, A.: End-user development, end-user programming and end-user software engineering: a systematic mapping study. J. Syst. Softw. **149**, 101–137 (2019)
7. Breve, B., Cimino, G., Desolda, G., Deufemia, V., Elefante, A.: On the user perception of security risks of TAP rules: a user study. In: Proceedings of International Symposium on End User Development (ISEUD 2023). Lecture Notes in Computer Science, vol. 13917, pp. 162–179. Springer, Cham (2023)
8. Breve, B., Cimino, G., Deufemia, V.: Identifying security and privacy violation rules in trigger-action IoT platforms with NLP models. IEEE Internet Things J. **10**(6), 5607–5622 (2022)
9. Breve, B., Cimino, G., Deufemia, V.: Hybrid prompt learning for generating justifications of security risks in automation rules. ACM Trans. Intell. Syst. Technol. **15**(5), 1–26 (2024)
10. Breve, B., Cimino, G., Deufemia, V., Elefante, A.: Unleashing the power of NLP models for semantic consistency checking of automation rules. J. Vis. Lang. Comput. **2023**(2), 1–14 (2023)
11. Breve, B., Cimino, G., Deufemia, V., Elefante, A., et al.: A BERT-based model for semantic consistency checking of automation rules. In: Proceedings of the 29th International Distributed Multimedia Systems Conference on Visualization and Visual Languages, pp. 87–93 (2023)
12. Breve, B., Cimino, G., Deufemia, V., et al.: Towards explainable security for ECA rules. In: Proceedings of the 3rd International Workshop on Empowering People in Dealing with Internet of Things Ecosystems (EMPATHY 20222) co-located with International Conference on Advanced Visual Interfaces (AVI). CEUR Workshop Proceedings, vol. 3172, pp. 26–30 (2022)
13. Breve, B., Desolda, G., Deufemia, V., Greco, F., Matera, M.: An end-user development approach to secure smart environments. In: Fogli, D., Tetteroo, D., Barricelli, B.R., Borsci, S., Markopoulos, P., Papadopoulos, G.A. (eds.) IS-EUD 2021. LNCS, vol. 12724, pp. 36–52. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79840-6_3
14. Chen, M., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)
15. Chen, Y., Alhanahnah, M., Sabelfeld, A., Chatterjee, R., Fernandes, E.: Practical data access minimization in trigger-action platforms. In: Proceedings of 31st USENIX Security Symposium, pp. 2929–2945 (2022)

16. Cimino, G., Deufemia, V.: SIGFRID: unsupervised, platform-agnostic interference detection in IoT automation rules. ACM Trans. Internet Things (2025)
17. Corcella, L., Manca, M., Paternò, F., Santoro, C.: A visual tool for analysing IoT trigger/action programming. In: Proceedings of International Working Conference Human-Centered Software Engineering, pp. 189–206. Springer, Cham (2019)
18. Corno, F., De Russis, L., Monge Roffarello, A.: Empowering end users in debugging trigger-action rules. In: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, pp. 1–13 (2019)
19. DeepSeek-AI: DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. arXiv preprint arXiv:2501.12948 (2025)
20. DeepSeek-AI, Bi, X., et al.: DeepSeek LLM: scaling open-source language models with longtermism. arXiv preprint arXiv:2401.02954 (2024)
21. Feng, Z., et al.: CodeBERT: a pre-trained model for programming and natural languages. In: Cohn, T., He, Y., Liu, Y. (eds.) Findings of the Association for Computational Linguistics, pp. 1536–1547 (2020)
22. Gao, Y., Xiao, K., Li, F., Xu, W., Huang, J., Dong, W.: ChatIoT: zero-code generation of trigger-action based IoT programs. Proc. ACM Interact. Mob. Wearable Ubiquit. Technol. **8**(3), 1–29 (2024)
23. García-Herranz, M., Haya, P.A., Alamán, X.: Towards a ubiquitous end-user programming system for smart spaces. J. Univers. Comput. Sci. **16**(12), 1633–1649 (2010)
24. Ghiani, G., Manca, M., Paternò, F., Santoro, C.: Personalization of context-dependent applications through trigger-action rules. ACM Trans. Comput.-Hum. Interact. **24**(2), 1–33 (2017)
25. Guo, D., et al.: DeepSeek-coder: when the large language model meets programming–the rise of code intelligence. arXiv preprint arXiv:2401.14196 (2024)
26. Hu, E.J., et al.: LoRa: low-rank adaptation of large language models. arXiv preprint arXiv:2106.09685 (2021)
27. Hu, K., Duan, Z., Wang, J., Gao, L., Shang, L.: Template-based AADL automatic code generation. Front. Comput. Sci. **13**, 698–714 (2019)
28. Hui, B., et al.: Qwen2.5-coder technical report. arXiv preprint arXiv:2409.12186 (2024)
29. Jiang, A.Q., et al.: Mistral 7b. arXiv preprint arXiv:2310.06825 (2023)
30. Jiang, J., Wang, F., Shen, J., Kim, S., Kim, S.: A survey on large language models for code generation. arXiv preprint arXiv:2406.00515 (2024)
31. Krishna, A., Le Pallec, M., Mateescu, R., Salaün, G.: Design and deployment of expressive and correct web of things applications. ACM Trans. Internet Things **3**(1), 1–30 (2021)
32. Lieberman, H., Paternò, F., Klann, M., Wulf, V.: End-user development: an emerging paradigm. In: End User Development, pp. 1–8. Springer, Cham (2006)
33. Lin, C.Y.: ROUGE: a package for automatic evaluation of summaries. In: Text Summarization Branches Out, pp. 74–81. Association for Computational Linguistics (2004)
34. Ling, W., et al.: Latent predictor networks for code generation. In: Erk, K., Smith, N.A. (eds.) Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 599–609. Association for Computational Linguistics (2016)
35. Papineni, K., Roukos, S., Ward, T., Zhu, W.J.: BLEU: a method for automatic evaluation of machine translation. In: Proceedings of the 40th annual meeting of the Association for Computational Linguistics, pp. 311–318 (2002)

36. Paternò, F.: End user development: survey of an emerging field for empowering people. Int. Scholarly Res. Not. **2013**(1), 532659 (2013)
37. Phan, H., Sharma, A., Jannesari, A.: Generating context-aware API calls from natural language description using neural embeddings and machine translation. In: Proceedings of 36th IEEE/ACM International Conference on Automated Software Engineering Workshops, pp. 219–226. IEEE (2021)
38. Qwen, Yang, A., et al.: Qwen2.5 technical report. arXiv preprint arXiv:2412.15115 (2025)
39. Ren, S., et al.: CodeBLEU: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297 (2020)
40. Roziere, B., et al.: Code Llama: open foundation models for code. arXiv preprint arXiv:2308.12950 (2023)
41. Touvron, H., et al.: Llama 2: open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288 (2023)
42. Ur, B., McManus, E., Pak Yong Ho, M., Littman, M.L.: Practical trigger-action programming in the smart home. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 803–812 (2014)
43. Wang, Y., Wang, W., Joty, S., Hoi, S.C.: CodeT5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Moens, M.F., Huang, X., Specia, L., Yih, S.W. (eds.) Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pp. 8696–8708. Association for Computational Linguistics (2021)
44. Yao, Y., Kamani, M.M., Cheng, Z., Chen, L., Joe-Wong, C., Liu, T.: FedRule: federated rule recommendation system with graph neural networks. In: Proceedings of the 8th ACM/IEEE Conference on Internet of Things Design and Implementation, pp. 197–208 (2023)
45. Yin, P., Neubig, G.: A syntactic neural model for general-purpose code generation. arXiv preprint arXiv:1704.01696 (2017)
46. Yusuf, I.N.B., Jamal, D.B.A., Jiang, L., Lo, D.: RecipeGen++: an automated trigger action programs generator. In: Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1672–1676 (2022)
47. Yusuf, I.N.B., Jiang, L., Lo, D.: Accurate generation of trigger-action programs with domain-adapted sequence-to-sequence learning. In: Proceedings of the IEEE/ACM International Conference on Program Comprehension, pp. 99–110 (2022)
48. Zhang, L., He, W., Martinez, J., Brackenbury, N., Lu, S., Ur, B.: AutoTap: synthesizing and repairing trigger-action programs using LTL properties. In: Proceedings of IEEE/ACM 41st International Conference on Software Engineering, pp. 281–291. IEEE (2019)
49. Zhang, L., Zhou, C., Littman, M.L., Ur, B., Lu, S.: Helping users debug trigger-action programs. Proc. ACM Interact. Mob. Wearable Ubiquit. Technol. **6**(4), 1–32 (2023)
50. Zhong, V., Xiong, C., Socher, R.: Seq2SQL: generating structured queries from natural language using reinforcement learning. arXiv preprint arXiv:1709.00103 (2017)
51. Zhou, S., Alon, U., Agarwal, S., Neubig, G.: CodeBERTScore: evaluating code generation with pretrained models of code. In: Bouamor, H., Pino, J., Bali, K. (eds.) Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pp. 13921–13937 (2023)

Author Queries

Chapter 2

Query Refs.	Details Required	Author's response
AQ1	This is to inform you that corresponding author has been identified as per the information available in the Copyright form.	
AQ2	As Per Springer style, both city and country names must be present in the affiliations. Accordingly, we have inserted the city and country names in affiliations. Please check and confirm if the inserted city and country names is correct. If not, please provide us with the correct city and country names.	