

ARCEAK: An Automated Rule Checking Framework Enhanced with Architectural Knowledge

Junyong Chen*, Ling-I Wu*, Minyu Chen*, Xiaoying Qian[†], Haoze Zhu[‡], Qiongfang Zhang[‡], Guoqiang Li* 

*Shanghai Jiao Tong University, Shanghai 200240, China

{chen.jy, edithwuly, minkow, li.g}@sjtu.edu.cn

[†]Beijing University of Civil Engineering and Architecture, Beijing, China

202304050101@stu.bucea.edu.cn

[‡]East China Architectural Design & Research Institute, Shanghai, China

{haoze_zhu, qiongfang_zhang}@ecadi.com

Abstract—Automated Rule Checking (ARC) plays a crucial role in advancing the construction industry by addressing the laborious, inconsistent, and error-prone nature of traditional model review conducted by industry professionals. Manual assessment against intricate sets of rules often leads to significant project delays and expenses. In response to these challenges, ARC offers a promising solution to improve efficiency and compliance in design within the construction sector. However, the main challenge of ARC lies in translating regulatory text into a format suitable for computer processing. Current methods for rule interpretation require extensive manual labor, thereby limiting their practicality. To address this issue, our study introduces a novel approach that decomposes ARC into two distinct tasks: rule information extraction and verification code generation. Leveraging generative pre-trained transformers, our method aims to streamline the interpretation of regulatory texts and simplify the process of generating model compliance checking code. Through empirical evaluation and case studies, we showcase the effectiveness and potential of our approach in automating code compliance checking, enhancing the efficiency and reliability of construction projects.

Index Terms—Automated rule checking, Rule interpretation, Information extraction, Code generation, Generative pre-trained transformer, Large language model

I. INTRODUCTION

The Architecture, Engineering and Construction (AEC) industry has undergone significant digital transformation in recent years, transitioning from traditional 2D line drawings to data-centric construction processes. Ensuring compliance with established rules and regulations is crucial for delivering high-quality building design models. To replace or augment manual rule checking, the concept of *Automated Rule Checking (ARC)* has been introduced as a potential solution. ARC refers to a technology-driven approach automated building rule compliance checking by converting rules into machine-readable formats [1]. Traditional ARC approaches often rely on hard-coded or manual rule interpretation methods [2], [3], while modern researches benefit from advancements in natural language processing [4]–[6]. Recent ARC approaches leverage fine-tuned language models such as BERT [7] to convert natural language building rules into structured formats [8], [9], including knowledge graphs. Despite these advancements,

the substantial demand for domain-specific labeled data for fine-tuning continues to hinder the large-scale deployment of language models in the ARC domain.

Very recently, large language models (LLMs) such as GPT-3.5-Turbo have shown remarkable performance in various natural language processing (NLP) downstream tasks, including sentiment analysis [10], natural language inference [11] and misinformation detection [12]. Researchers [13], [14] have even assigned different characters to these models to specialize in various tasks and collaborate on developing software engineering systems. One of the most notable capabilities of LLMs is their zero-shot generalization across diverse tasks [15]. With well-designed prompts, these models can perform complex tasks at a human-level ability, addressing issues related to the lack of training data for fine-tuning. As for the ARC task, LLMs have already demonstrated its effectiveness in similar tasks such as natural language understanding [16], [17] and code generation [18], [19]. Thus, we also expect LLMs to address several challenges such as building rule interpretation and checking code generation.

To harness the powerful in-context learning capabilities of LLMs for the ARC task, we propose ARCEAK, an **A**utomated **R**ule **C**hecking **E**nabled with **A**rchitectural **K**nowledge. Our framework consists of two main stages. The first stage is LLM-based Rule Information Extraction, which focuses on extracting verification-related information from building rules. This stage is further divided into two steps: entity discovery (ED) and event extraction (EE). ED involves recognizing construction domain-specific entities, while EE identifies assignments related to these entities using construction domain knowledge augmented prompt engineering. The second one is LLM-based verification code generation, aiming to generate fine-grained, executable verification code by combining the extracted entities, events, and rule entry content with code generation prompts. This stage is also divided into two steps: code framework generation and rule checking code completion. Code framework generation involves creating the skeleton of the verification code, and rule checking code completion fills in the specific execution code. We conducted comprehensive experiments to evaluate the performance of our proposed framework. The results show

 Corresponding author

that, for the Rule Information Extraction stage, ARCEAK has improved the F1 score of ED by 60% and increased the precision of EE by 2.2%. For the Verification Code Generation stage, ARCEAK has achieved a compile pass rate of 63% with GPT-3.5-Turbo and a logic pass rate of 24% with GPT-4-Turbo. Meanwhile, the knowledge-augmented code generation has demonstrated significantly better performance than non-knowledge-augmented code generation.

The contributions of this paper are summarized as follows:

- We propose a novel LLM-based ARC framework enhanced with architectural knowledge, achieving an almost fully automated process for converting natural language building rules into executable verification code.
- We develop a robust construction domain-specific entity and event schema and established appropriate verification code generation granularity to balance problem complexity with the LLMs' code generation capabilities, ensuring the framework's scalability and transferability.
- We implement the ARCEAK framework and evaluated its performance using metrics designed with architecture experts, addressing both software engineering requirements and construction reliability.

Paper Organization: The structure of the remaining part of this paper is as follows: In Section II, we offer an extensive review of the background and related research. The design and implementation of the entire ARCEAK process are detailed in Section III. Section IV presents the performance of different stages and the final results achieved by ARCEAK. Section V presents multiple case studies and analyzes potential threats to validity. Finally, conclusions are outlined in Section VII.

II. BACKGROUND

In this section, we first introduce the background knowledge related to LLM and prompt engineering. Then, we explore key concepts in information extraction. Finally, we discuss code generation techniques, including both specialized and general LLMs for code generation.

A. Large Language Model

Large Language Models (LLMs) are generative models based on the pre-trained Transformer architecture [20]. Leveraging extensive multimodal data and employing pre-training and fine-tuning techniques, LLMs have significantly advanced the field of NLP by enhancing capabilities in multilingual translation [21]–[23], summarization generation [24], [25], and code simulation [26], [27]. LLM training generally involves three key stages: unsupervised learning on vast amounts of unlabeled text data without direct human annotations, supervised fine-tuning using labeled data tailored to specific tasks or domains, and reinforcement learning based on feedback from human evaluators or annotators.

Prompt Engineering offers a powerful method to extend capabilities of LLMs without the need for extensive model retraining or modification of parameters and has emerged as a transformative technique in the realm of LLMs [28], [29]. This approach harnesses prompts to tailor model behavior to

specific tasks or domains, thereby enhancing model efficacy and versatility. Prompt engineering can be categorized into two main types: *zero-shot prompting* [30] and *few-shot prompting* [15]. The key distinction between these two lies in whether examples related to the task are provided to the language model. Zero-shot prompting involves giving the LLM an instruction without any examples, while few-shot prompting includes providing a few relevant examples to guide the LLM's response. Zero-shot prompting and few-shot prompting can also be combined with in-context learning methods to further enhance the performance of LLMs in solving complex problems or unseen domain-specific tasks. One of the most widely-used in-context learning methods is Chain-of-Thought Prompting [29], which decomposes problems into intermediate steps and solves each one before arriving at the final answer.

B. Information Extraction

Information Extraction (IE) is a fundamental domain in NLP to convert plain text into structured knowledge format [31], [32]. The IE tasks cover:

- **Entity Discovery (ED)** [33] encompasses both entity recognition [34] and entity typing [35]. The former is concerned with identifying spans of entities (e.g., ‘Steve’), and the latter focuses on assigning types to these identified entities (e.g., ‘PERSON’).
- **Event Extraction (EE)** [36] generally involves two stages. The first stage, event detection [37], focuses on identifying trigger words that signify the occurrence of specific events. The second stage, event argument extraction [38], seeks to extract the arguments associated with these events from given text. The set of target arguments varies depending on the event’s definition.

Consequently, recent generative IE methods that leverage LLMs to generate structural information have gained more attention than merely extracting it from plain text. Generative IE methods have demonstrated greater flexibility compared to traditional IE approaches. However, due to the limited presence of domain-specific data in LLM training, prompt engineering can be employed to enable LLMs to learn input-output mappings for specific downstream tasks without the need for fine-tuning. Inspired by previous works [39], [40], our work builds upon a generative IE method to extract information from building rules.

C. Code Generation

Code Generation involves creating programs that adhere to the constraints set by the underlying task [41]. These constraints typically come in diverse forms, such as input/output pairs, examples, problem descriptions, partial programs, and assertions. The remarkable success of transformers in natural language modeling has sparked significant interest among researchers in leveraging transformer models for code generation.

On one hand, there has been a proliferation of specialized LLMs tailored for code generation. Notably, OpenAI has introduced Codex [42], a GPT-3 model fine-tuned on publicly

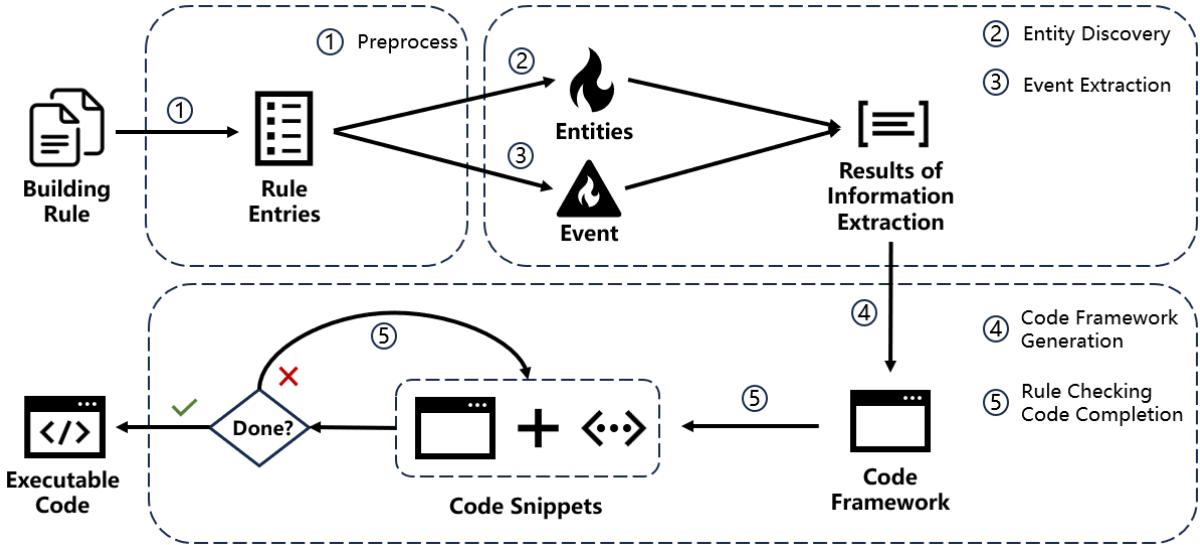


Fig. 1. The overall architecture of proposed ARCEAK

available code from GitHub, boasting a maximum parameter count of 12 billion. Microsoft unveiled GPT-C [43], a variant of GPT-2 retrained on a vast unsupervised multilingual source code dataset, followed by the release of PyMT5 [44] and CodeXGLUE [45]. On the other hand, rapid engineering techniques are employed to adapt general LLMs for code generation tasks. General LLMs like Llama3 [46] and GPT-4 [47] have also demonstrated impressive performance in various code-related tasks. Additionally, prompt engineering techniques have been proposed to enhance the performance of general LLMs on specific tasks by carefully designing the input prompts.

III. METHODOLOGY

In this section, we introduce ARCEAK, a novel ARC framework, which enhances verification code generation through the integration of architectural knowledge. We begin with an overview of ARCEAK, followed by a detailed explanation of its components in the subsequent subsections.

A. Overview

The implementation of ARC can be delineated into two primary stages: *rule information extraction* and *verification code generation*. The former is dedicated to analyzing the textual rules within the architectural domain and extracting pertinent information essential for verification purposes. The latter stage focuses on the generation of execution code aimed at assessing whether a given architectural blueprint complies with the building rules and requirements.

To mitigate or eliminate the necessity for manual annotation and model training during the rule information extraction and verification code generation stages, we introduce the ARCEAK framework, leveraging LLMs. The overall architecture of ARCEAK is illustrated in Fig. 1. Initially, we undertake a preprocessing phase for building rules, employing a prompt

designed for rule splitting and error correction. Subsequently, we proceed to extract pertinent information, such as entities and events, from the refined rule content. Finally, our attention shifts to generating high-level code informed by knowledge-enhanced building rules. The subsequent sub-sections offer a comprehensive elucidation of the distinct phases within ARCEAK.

B. Preprocess

Since the national code files are consistently published in PDF format, which is not suitable for information extraction, we need to preprocess the original file and convert it into a structured format. An illustration of the preprocessing results is presented in Fig. 2.

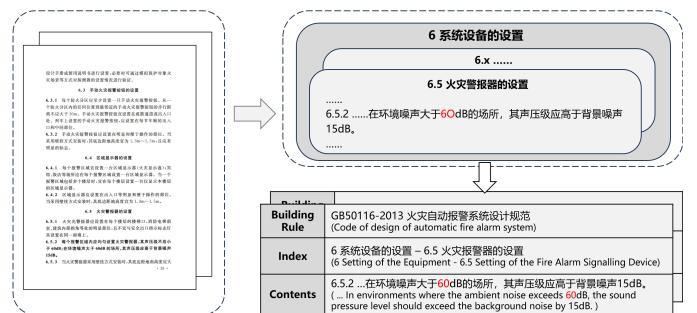


Fig. 2. An example of preprocessing result

First, we convert the PDF file into a TXT file with UTF-8 encoding using Adobe OCR tools. However, the TXT format is not compatible with tables and pictures, which are crucial for understanding the rules and cannot be ignored. After verifying that the number of tables and pictures is manageable, we manually convert the tables into CSV files and the pictures into PNG format.

Second, we continue preprocessing the text portion of the building rules into a structured format. Typically, building rules are organized into chapters, which are filled with sections, and sections with individual entries. Each rule entry is our main target for information extraction and code generation. Therefore, we utilize an LLM to split the entire chapter into individual rule entries using a few-shot prompting approach. After the initial split, we index each entry with its corresponding chapter and section numbers to serve as a navigational guide.

Finally, since similar characters like "0" and "o" or "1" and "I" may be misinterpreted by Adobe OCR tools, leading to semantic errors in subsequent information extraction and code generation stages, OCR mistake detection and correction is required. Additionally, we aim to integrate the content of tables and pictures into the textual content to maintain the alignment of rule entries. Therefore, we leverage the multimodal contextual understanding ability of LLMs by providing the textual content along with the linked tables and pictures to the LLMs, enabling a comprehensive and mistake-free understanding of each rule entry. A manual check is also conducted to ensure the accuracy of the conversion.

C. Rule Information Extraction

Information Extraction (IE) refers to the process of automatically extracting structured information from unstructured or semi-structured data, such as text. Extracting information from building rules, which encompasses implicit structural entities and intricate constraint conditions, poses several challenges, including complex context understanding and recognition of architectural terminology.

To overcome these issues and extract rule information more accurately and comprehensively, we design an information extraction mechanism based on knowledge-enriched zero-shot prompting with Classification Annotation (CA), which requires minimal manual involvement, to extract entities and events from preprocessed rule entries. IE stage mainly contains two tasks: entity discovery and event extraction.

TABLE I
ADOPTED ENTITY TYPES FOR ENTITY CLASSIFICATION

Type name	Examples
Building System Component Zone	building, dwelling building fire extinguishing system, fire automatic alarm system detector, button fire compartment, smoke-proof compartment

Entity Discovery(ED) encompasses both entity recognition and entity classification. To enhance LLMs' ability to accurately discover domain-specific entities in rule entries, we recommend an adaptive prompt optimization approach in this section. Initially, we allocate instructions for recognizing entities within the given rule entry and concurrently classifying the identified entities into possible types. To address the specificity of entity types, we elaborate on the target

entity classification categories to LLMs through entity type classification annotations within the context part of the prompt for the ED task. The CA for ED task may encompass any entity types as long as there are few examples of these types and they perform well with clear and unambiguous heuristic type classification. Here, we employ the four entity types and corresponding examples shown in Table I.

Event Extraction(EE) involves identifying and classifying events described in text. An event, serving as a fine-grained semantic unit to describe the state of entities and their actions, is typically defined as a textual span comprising a predicate and its arguments. Complex conditions pose one of the primary challenges in interpreting building rules. Rule entries in building rules can be abstracted into two categories: attribute assignment with conditions and attribute assignment without conditions.

In our study, we define an attribute assignment as a type of event, where entities identified in the ED phase serve as "trigger words", with the goal of extracting arguments related to "assignment" events. A complete argument entry consists of the following components:

- **Entity of Attribute:** The entity to which the attribute belongs.
- **Attribute Name:** The name of the extracted attribute.
- **Conditions or Constraints:** The constraints on attribute assignment.
- **Comparator:** The comparator used for the attribute value.
- **Attribute Value:** The value assigned to the attribute.

Since general LLMs lack domain-specific knowledge of construction, the arguments describing the core information of 'assignment' events present considerable complexity, making it difficult for LLMs to identify and extract them directly without additional explanation. To enhance the LLMs' ability to recognize assignments in the construction domain, we compiled common assignment expressions for six types of assignments. The examples of common assignment expressions are shown in Table II.

TABLE II
EXAMPLES OF COMMON EXPRESSIONS OF ASSIGNMENTS

Type name	Examples
Direct attribute constraint	length, width, height
Quantity constraint	quantity, piece, unit
Distance constraint	distance, spacing, separation
Classification constraint	type, category
Spatial constraint	top, above, centered
Other indirect constraint	components, composed of

After the IE stage, the entities and events in the construction domain are extracted from the rule entry content, and the unstructured rule entry content is formatted as structured JSON lists.

D. Verification Code Generation

Code generation refers to the process of automatically generating source code by a computer program. Manual code writing and verification is expensive, and generating code from structured rules is effective for explicit rules but often incomplete due to the presence of implicit rules. To address this, we propose a knowledge-augmented code generation workflow that leverages the code generation and completion capabilities of LLMs. Given the complexity of verification code generation, we follow the approach of Plan-and-Solve Prompting [48], dividing the entire process into two primary steps: code framework generation and rule-checking code completion.

Code Framework Generation aims to construct a fundamental code skeleton, encompassing vital sections for variable initialization, function definitions, and control structures necessary for conditional evaluations. To enable LLMs to grasp the domain knowledge of building rules and generate precise code representing the details of rule entries, as well as to control the granularity of the function code and align the code structure more closely with the specific requirements of the building rules, we integrate entities and arguments related to the "assignment" event extracted during the IE stage with code framework generation prompt instructions. To address semantic dependencies between rules, we use a parser to determine whether the current rule entry depends on other rule entries; if such dependencies exist, both the current rule entry and its dependencies are provided to the LLM. Furthermore, to enhance the coverage rate, we aim to guide LLMs to annotate the generated verification code with the target rule entry index, ensuring that each rule component presented in the rule entries is comprehensively checked and represented in the code framework. Our code generation prompt is shown in Table III.

TABLE III
CODE GENERATION PROMPT

Code Generation Prompts
Instruction: You are a Revit secondary development engineer who is skilled in writing Revit compliance check code. Please convert the building specifications surrounded by " " into Revit check code in C# language. Please refer to the entities and properties extracted from the specifications and write compliance check codes.
Input Data: <entity list>, <event list>, "<rule content>"
Output Format: Please first generate the code framework. The framework should include basic structures for variable initialization, function definitions, and conditional statements. For unimplemented functions, please add <unimplemented> before the function definition. Ensure that each specification check item appears in the code.

Rule-Checking Code Completion begins with integrating detailed logic and specific code snippets into the basic code skeleton formed during code framework generation. This crucial phase involves populating the established framework with specific details and logic tailored to the building rules. Placeholders previously defined in the skeleton are filled with actual code sections generated to check compliance with the rules.

Furthermore, to pinpoint the specific rule that an architectural blueprint violates, we instruct LLMs to add detailed assert statements, including the specific rule entry according to the indexed comments in the code. Our code completion prompt is shown in Table IV.

TABLE IV
CODE COMPLETION PROMPT

Code Completion Prompts
Instruction: You are a Revit secondary development engineer who is skilled in writing Revit compliance check code. Please convert the building specifications surrounded by " " into Revit check code in C# language. Please refer to the entities and properties extracted from the specifications and write compliance check codes. We will provide a code framework. Please implement the unimplemented functions in the code framework enclosed by " " "
Input Data: <code framework>, <unimplemented function>, <entity list>, <event list>, "<rule content>"
Output Format: Please provide the complete code.

Since new unimplemented or undefined functions may be added during the code completion process, iterations are necessary. To determine if the code completion is complete, we design a parser to extract all variables and functions used in the generated verification code, then verify whether each variable and function is defined and implemented. Additionally, because it is uncertain whether the code generated by LLMs is runnable and may contain compilation errors, we employ a code self-refinement process to enhance the runnability of the generated verification code. In this process, we provide the LLMs with the completed verification code along with reported errors, allowing them to refine the code they generated.

After the knowledge-augmented verification code generation stage, we facilitate a smoother transition from natural language descriptions of building rules to executable verification code. This code can be used to verify whether architectural blueprints comply with regulations by invoking the API of the selected models. An example of code generation result is shown in Fig. 3.

IV. EVALUATION

To comprehensively evaluate the performance of the proposed framework ARCEAK, we conduct a large-scale study to seek to address the following research questions (RQ):

- **RQ1: How effective is ARCEAK in extracting information from a building rule?** As described above, we employ a knowledge-enriched zero-shot prompting strategy to enhance LLM performance in the IE stage. This RQ aims to verify whether the incorporation of CA in ARCEAK improves the LLM's ability to perform IE more effectively.
- **RQ2: How comprehensive and accurate is the code generated by ARCEAK in enforcing building rules?** We employed a two-step code generation process, enhanced by knowledge extracted during the IE stage. This

```

1  using System;
2  // [...]
3
4  namespace DetectorComplianceCheck {
5      // [...]
6      public Result Execute(...) {
7          // [...]
8          // Check the horizontal distance to the walls and beam edges
9          CheckHorizontalDistance(instance);
10         // [...]
11     }
12
13     // Index 6 Setting of the equipment
14     // - 6.2 Setting of the fire alarm signalling device
15     // - 6.2.5 Setting of the fire detector
16     private void CheckHorizontalDistance(FamilyInstance instance) {
17
18         throw new NotImplementedException();
19     }
20 }

```

```

1  namespace DetectorComplianceCheck {
2      // [...]
3
4      private void CheckHorizontalDistance(FamilyInstance instance) {
5          LocationPoint location = instance.Location as LocationPoint;
6          List<Element> wallsAndBeams = // [...]
7          foreach (var wallOrBeam in wallsAndBeams) {
8
9              Options geomOptions = new Options();
10             geomOptions.ComputeReferences = true;
11             GeometryElement geomElem = wallOrBeam.get_Geometry(geomOptions);
12
13             foreach (GeometryObject geomObj in geomElem) {
14                 if (!(geomObj is Solid)) continue;
15                 Solid geomSolid = geomObj as Solid;
16                 // [...]
17             }
18         }
19     }
20 }

```

Fig. 3. An example of code generation result

RQ seeks to determine whether the knowledge augmentation and two-step generation approach in ARCEAK positively impact domain-specific code generation.

- **RQ3: How does the IE stage in ARCEAK enhance the accuracy and efficiency of verification code generation stage?** ARC is a domain-specific and complex task, and merely providing evaluation metrics may not be sufficient to fully convey the effectiveness of ARCEAK. It is valuable to analyze concrete cases to better understand the impact of ARCEAK's IE stage on generating building compliance checking code.

RQ1:How effective is ARCEAK in extracting information from a building rule?

Setup. In our study, the Chinese building code *GB50116-2013* (Code for Design of Automatic Fire Alarm System) is selected to validate the IE stage in ARCEAK. For the IE stage of ARCEAK, we implement our method using GPT-3.5¹, and compare it to Chain-of-Thought(CoT) Prompting [29] without CA, which we call naive CoT. This comparison is conducted across both ED and EE tasks. In the naive baseline, the model is given the natural language instruction and is asked to directly discover the entity and extract events related to the discovered entity.

Metrics. To compare the performance of the ED phase of IE stage, we employed the following three distinct metrics. Prior to introducing these metrics, it is essential to revisit three fundamental concepts: True Positives(TP), False Positives(FP) and False Negatives(FN). TP denotes correct predictions made by the classifier for different entity types. Conversely, FP and FN denote instances from different type of entities that have been misclassified.

- *Precision:* Precision is the ratio of the number of entities extracted from rule with correct types to the number of entities extracted from rule , which is calculated with,

$$Precision = \frac{TP}{TP + FP} \quad (1)$$

- *Recall:* Recall is the ratio of the number of entities extracted from rule with correct types to the number

of entities which should be extracted from rule(ground truth)d , which is calculated with,

$$Recall = \frac{TP}{TP + FN} \quad (2)$$

- *F1:* F1 score is a weighted average of the framework precision and recall, which ranges from 0 to 1. A higher F1 score indicates better comprehensive performance of the framework. F1 score is defined as the harmonic mean of the precision and recall, which can be calculated with,

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \quad (3)$$

To comprehensively evaluate the performance of the EE phase of IE stage, we employed four distinct metrics,

- *Tri-R:* Tri-R is the ratio of the number of intersection of events extracted from rule and events which should be extracted from rule with the same "trigger word" entity to the number of events which should be extracted from rule(ground truth), which represents the ratio of how many real events are extracted.
- *Arg-P_a:* Arg-P_a is the ratio of the number of intersection of events extracted from rule and ground truth of event with the same entity and attribute to the number of events extracted from rule, which represents the ratio of how many extracted events extract the true attribute.
- *Arg-P_A:* Arg-P_A represents the ratio of extracted events that contain all the required components.
- *Arg-P_O:* Arg-P_O represents the ratio of extracted events that contain all the required components and are returned in the correct order as specified by the prompt.

TABLE V
EVALUATION ON CLASSIFICATION ANNOTATION IN ENTITY DISCOVERY STAGE

	Naive CoT	ARCEAK
Number of Extracted Entities	407	774
Precision	0.042	0.631
Recall	0.020	0.713
F1	0.027	0.669

¹<https://openai.com/>

Result and Analysis. The primary goal of RQ1 is to assess the effectiveness of CA in ARCEAK for extracting information with minimal manual involvement. During the ED phase of the IE stage, the LLM extracts and classifies potential architectural entities. In the EE phase, the LLM detects “assignment” events based on “trigger word” entities and extracts their potential arguments. These experiments use the entire set of rule entries from GB50116-2013. To ensure accurate evaluation, a two-layer assessment structure is implemented. The first layer involves five junior evaluators cross-evaluating the IE results, with each result reviewed by two evaluators. In the second layer, a domain expert equipped with empirical knowledge further assesses the results.

Table V shows ARCEAK’s performance in the ED phase of the IE stage. ARCEAK extract 90% more entities than naive CoT prompting, while achieving approximately 64% higher F1 score. The data indicates that CA significantly enhances the LLM’s ability to extract and classify architectural entities. The poor performance of the naive CoT approach, lacking CA, highlights the limitations of LLMs in extracting domain-specific information. In contrast, the improved performance with CA underscores its effectiveness in the ED phase.

TABLE VI
EVALUATION ON CLASSIFICATION ANNOTATION IN EVENT EXTRACTION STAGE

	Naive CoT	ARCEAK
Number of Extracted Event	659	693
Tri-R	0.839	0.844
Arg-P _a	0.553	0.535
Arg-P _A	0.378	0.365
Arg-P _O	0.205	0.227

Table VI shows the comprehensive performance of ARCEAK in the EE phase of the IE stage, which demonstrates that prompting with CA can improve the event detection and argument extraction capabilities of the LLM. In the EE phase, CA resulted in 1.8% and 1.3% lower performance in Arg-P_a and Arg-P_A, respectively. However, it achieved a 2.2% higher performance in Arg-P_O. This indicates that while the LLM prompted by naive CoT tends to retain more information in the extracted events without fully understanding the relevance of the retained information, CA in the EE phase enhances the precision of extracting specific types of arguments.

Answer to RQ1: ARCEAK significantly improves the precision, recall, and F1 score in the ED phase. In the EE phase, ARCEAK enhances the LLM’s ability to detect events and extract arguments with higher precision for specific argument types. Overall, ARCEAK’s CA method effectively enhances the LLM’s performance in IE stage.

RQ2: How comprehensive and accurate is the code generated by ARCEAK in enforcing building rules?

Setup. For verification code generation stage of ARCEAK, rule entries from GB50116-2013 are selected and twenty room-level BIM models containing components that both comply with and deviate from the selected rule are constructed to evaluate the performance of prompt with CoT. We implement our method with GPT-3.5 and GPT-4, and compare to code generation without CoT prompt. The code generated by LLM is running on Revit 2020 to test for the accurate performance.

Metrics. To comprehensively analysis the performance and cost of CoT prompt in code generation phase, we introduce the following metrics:

- *Code Integrity*: Code Integrity refers to the extent to which the generated code framework during the initial phase accurately represents and retains the requirements outlined in the building code.
- *Done@K*: Done@K is the ratio of the number of code completions that are successfully finalized before the K+1 term to the total number of code generations.
- *Compile Pass Rate*: Compile Pass Rate is the ratio of the number of the generated codes which raise no error while compiling to the number of the generated codes.
- *Logic Pass Rate*: Logic Pass Rate is the ratio of the number of the generated codes which contain no logic error(e.g., wrong comparison) to the number of the generated codes.
- *Pass Rate*: Pass Rate is the ratio of the number of the generated codes pass the test model to the number of the generated codes.

TABLE VII
COMPARISON BETWEEN GPT-3.5 AND GPT-4 ON CODE INTEGRITY AND DONE@K

	wo-CoT		w-CoT	
	GPT-3.5	GPT-4	GPT-3.5	GPT-4
Code Integrity	0.78	0.93	1.00	1.00
Done@1	0.36	0.30	0.20	0.24
Done@2	0.94	0.90	0.90	0.88

Result and Analysis. The RQ2 is mainly to evaluate whether the CoT prompt in ARCEAK for code generation is effective on generating codes that closely align with the specific requirements of rule entries. The CoT prompt is employed to enhance the LLM’s ability to interpret and implement detailed rule entries effectively. By using CoT, the LLM is prompted to reason through each step or requirement in a rule entry before generating the corresponding code.

Table VII represents the performance of ARCEAK in aligning code with rule entries accurately. With the introduction of CoT in LLM, the code integrity is improved to 100%, which means no part of the rule is overlooked and that the final output adheres closely to the specified requirements. Due to the instruction of a two-section generation strategy(generate the code framework first and then complete the unimplemented functions), Done@1 is decreased to approximately 20%, which

declines potentially escalates the computational and temporal costs associated with simpler rule entries. Nevertheless, Done@2 metric of code generation prompt with CoT achieves similar rates compared to the prompt without CoT, suggests that the model can effectively adjust and improve its initial outputs. This approach is particularly beneficial for complex coding tasks, where the initial framework helps ensure more precise completions in subsequent steps.

TABLE VIII
COMPARASION BETWEEN GPT-3.5 AND GPT-4 ON 0-SHOT AND 1-SHOT
CODE FRAMEWORK GENERATION

	0-shot		1-shot	
	GPT-3.5	GPT-4	GPT-3.5	GPT-4
Compile Pass	0.050	0.250	0.630	0.580
Logic Pass	0.020	0.130	0.110	0.240
Pass	0.0	0.080	0.030	0.100

Table VIII compares the performance of GPT-3.5 and GPT-4 in both 0-shot and 1-shot code framework generation scenarios. In the 1-shot scenario, GPT-3.5 exhibits a higher compile pass rate at 63% compared to GPT-4's 58%. This higher rate is attributed to GPT-3.5's tendency to use annotations or custom variables instead of direct component and property access operations, which reduces compilation errors. However, this approach leads to a significant drop in the logic pass rate and overall pass rate for GPT-3.5 compared to GPT-4. This is because the code generated by GPT-3.5 often fails to access model parameters correctly, affecting the functionality and logic accuracy of the code.

Answer to RQ2: The code generated by ARCEAK, particularly when using advanced models like GPT-4, is fairly comprehensive and accurate in enforcing building rules, despite some initial inefficiencies in the 1-shot scenario. The model's ability to adjust and refine its output ensures that it can meet detailed rule requirements effectively, making it suitable for complex tasks that demand high precision and adherence to specific standards.

RQ3: How does the IE stage in ARCEAK enhance the accuracy and efficiency of verification code generation stage?

Setup. To answer the RQ3, we implement our method with GPT-3.5 and GPT-4, and compare to code generation without the assistance of knowledge augmentation, which serves as a baseline to assess the effect of the IE stage.

Case Study. The RQ3 is proposed for analyzing the impact of knowledge augmentation on verification code generation and how extracted information from rule entries contributes to improved code accuracy and efficiency. The following case studies focus on specific instances where knowledge augmentation has markedly influenced the output of the code

generation process. By examining these instances, we aim to illustrate concrete examples of both success and challenges in integrating extracted information, providing a deeper understanding of the mechanisms and factors that influence the outcomes.

```

1 public void CheckCompliance() {
2     public void CheckFireDetectorCompliance(Room room, double area, double occupancy, double detectorProtectionArea) {
3         CheckMinimumDetectorsPerRoom(room);
4         CheckDetectorProtectionArea(room, detectorProtectionArea);
5         CheckDetectorSpacing(room, detectorProtectionArea);
6         CheckDetectorQuantity(room, area, occupancy, detectorProtectionArea);
7     }
8 }

1 public void CheckCompliance() {
2     public void CheckFireDetectorCompliance() {
3         // [.....]
4
5         RoomFilter roomFilter = new RoomFilter();
6         var rooms = new FilteredElementCollector(doc).OfCategory(BuiltInCategory.OST_Rooms).WherePasses(roomFilter);
7         foreach (Room room in rooms) {
8             // Search for point type fire detectors in the room
9             // [.....]
10        }
11    }
12 }
```

Fig. 4. An example on compatibility between the LLMs and selected model. The code above is generated without external information and the code below is generated with entity information extracted in Entity Discovery.

- **Enhancement on reliability:**

In this case study, we explore how the knowledge augmentation phase improves the compatibility between the LLM and a selected model. Without knowledge augmentation, LLMs might generate variables or implement data retrieval logic independently, leading to redundancy and inefficiency. However, with knowledge augmentation, the LLMs are better informed about the existing components and functions within the selected model, enabling them to utilize the model's APIs(in this case study, the model represents Revit) effectively.

As shown in Fig. 4, the code above is generated without external information, while the code below incorporates entity information extracted during the ED phase. The code generated by naive CoT prompting attempts to retrieve model elements by assuming method input parameters or arbitrarily defining variables or methods to represent the required parameters (e.g., "assuming the user has implemented a variable num_detector to represent the number of detectors in a room" or "assuming the existence of a function CheckDetectorPerRoom() to return the number of detectors"). This approach can sometimes result in compilation errors and a lower pass rate. In contrast, the knowledge-augmented code generation leverages existing Revit APIs and directly generates logic to retrieve model elements, resulting in more efficient and consistent code. This case study demonstrates that knowledge augmentation significantly enhances the compatibility between LLMs and the selected model, ensuring that the LLMs make optimal use of available resources.

- **Improvement on the control of granularity:**

This case study investigates how knowledge augmentation influences the granularity of functions generated by the LLMs. Granularity control is crucial for maintaining a balance between high-level abstractions and detailed implementations in code generation. Knowledge augmentation equips the LLMs with detailed information about the required level of abstraction, enabling them to generate functions with appropriate granularity.

```

1 public void CheckCompliance() {
2     private double GetProtectionArea(FamilyInstance detector) {
3         // [...]
4
5         if (detectorType == "探测器") {
6             // [...]
7         } else {
8             // [...]
9         }
10    } // [...]
11 }
12
13 }
14 }

1 public void CheckCompliance() {
2     private double GetProtectionArea(FamilyInstance detector) {
3         // [...]
4
5         if (detectorType == "感烟火灾探测器" || detectorType == "A型感温火灾探测器") {
6             // [...]
7         } else if (detectorType == "B型感温火灾探测器" || detectorType == "C型感温火灾探测器") {
8             // [...]
9         }
10    } // [...]
11 }
12
13 }
14 }

```

Fig. 5. An example of granularity in function generation. The code above is generated without knowledge augmentation and the code below is generated with knowledge augmentation

As illustrated in Fig. 5, the code above is generated without knowledge augmentation and the code below is generated with knowledge augmentation. For instance, for rule entry 6.2.2 in GB50116-2013, which involves different types of detectors (e.g., smoke fire detectors, A and B type temperature fire detectors, etc.), the code generated without the additional knowledge extracted from the IE stage of ARCEAK produces only a simple conditional check, ignoring the logic required to handle various detector types. In contrast, the code generated with knowledge augmentation includes more detailed and accurate handling logic. For complex rule entries like rule entry 6.2.2—which involves more than ten variables as decision criteria and exceeds 1,000 words in total length—LLMs tend to generate only abstract portions of the logic if the content is not explicitly emphasized in the prompts. However, when entity and event information extracted during the IE stage is included, LLMs are constrained to generate code that aligns with the extracted knowledge. The code generated without knowledge augmentation generates functions with better-controlled granularity, resulting in clearer, more maintainable code. This case study highlights the role of knowledge augmentation in guiding the LLMs to produce functions that align with project guidelines, thereby improving the overall quality of the generated code.

Answer to RQ3: By integrating extracted information, ARCEAK enhances the LLM's ability to generate accurate and efficient code. These case studies illustrate that knowledge augmentation not only improves compatibility with existing models but also ensures better control over the granularity of functions. Consequently, the overall quality of the generated code is significantly improved, demonstrating the value of the information extraction phase in the ARCEAK framework.

V. DISCUSSION

In this section, we summarize the correct and incorrect cases encountered during the experiments and evaluation process,

and analyze the potential causes behind them. We then discuss several threats that may impact the effectiveness of our work.

A. Case Study: Correct and Incorrect Results

We analyze several cases demonstrating ARCEAK's ability to guide LLMs in generating compliance checking code that better aligns with practical needs. However, there are still instances where our approach does not fully prevent the generation of erroneous results. Below, we provide a brief summary of these situations for further clarification.

1) Generating API or parameter: As shown in Fig.4, the code below generates the correct API for retrieving Room information: "FilteredElementCollector(doc).OfCategory(BuiltInCategory.OST_Rooms)". Using the Room API as an example, the API consists of four key components: two element retrieval APIs ("collector" and "filter") and two required parameters ("doc" and "class" enumeration). A correctly generated code for retrieving elements must ensure the accuracy of all four components. First, for the "doc" parameter and its retrieval, GPT-3.5 occasionally uses annotations or custom variables (e.g., "doc = _doc" or "Room room = getRoom() //implement your logic here to get room"), whereas GPT-4 is more likely to generate the correct "doc" parameter and element retrieval API. Next, with respect to the "class" enumeration, LLMs sometimes generate incorrect class enumeration values, particularly when dealing with complex entities. In such cases, LLMs might create custom enumeration values. Notably, GPT-4 demonstrates greater consistency and accuracy in generating the correct "class" enumeration values compared to GPT-3.5.

2) Generating verification code for complex rule entry: For verification code generation, positive pass rate examples are primarily concentrated on relatively simple rule entries. However, for rule entries involving complex operations (e.g., calculating the distance from a detector to the centerline of a wall), the pass rate begins to decline. GPT-3.5 often attempts to generate methods that are not implemented (without the `<unimplemented>` tag) to assume data, while GPT-4 tries to generate correct code but may sometimes miss special cases (e.g., transforming walls to avoid calculating the distance to the side of the wall, or excluding the floor where the detector is located when checking for obstructions within a certain range around the detector). For rule entries with complex logic (e.g., rule entry 6.2.2 in GB50116-2013, which contains over 1,000 words and includes multiple nested value intervals), even GPT-4 may overlook part of the decision logic.

B. Threats To Validity

In this section, We identify two main threats to the validity of our study:

1) Limited selection of models.: In this paper, we selected two LLMs for our experiments. However, it is important to acknowledge that other LLMs are available, including general models like Llama3 [46] and specialized models like CodeGen [49]. In future work, we plan to conduct experiments with a broader range of LLMs to more comprehensively explore the applicability of our framework.

2) *Limited dataset.*: In this paper, we used twenty room-level BIM models to verify the correctness of the code generated by ARCEAK, specifically testing the generation of C# code from text. It remains uncertain whether our experimental results and findings can be generalized to other languages (e.g., generating Python code for checking in Dynamo). In the future, we plan to build a larger building model dataset and make it publicly available to further support ARC research.

VI. RELATED WORK

Automated Rule Checking (ARC) is a technology-driven approach that automates the compliance verification process by converting building rules into computer-recognizable formats, such as decision tables [50] or query code [51]. As engineering projects become increasingly complex, manual compliance checking has grown both tedious and costly, while also raising the risk of human errors. In response, ARC offers a solution that can significantly reduce both time and expenses, all while improving the quality and accuracy of reviews. The rise of building modeling technologies has further bolstered this automation by making data more machine-readable, facilitating smoother integration into the compliance checking process. The ARC process consists of three key stages: 1) rule interpretation, which converts natural language rules into machine-readable formats, 2) building model preparation, which organizes the necessary information for rule checking, and 3) rule execution, where the prepared model is checked against the machine-readable rules [9]. Of these stages, rule interpretation and rule execution are particularly crucial and complex, warranting further research [52].

Current research on ARC for conditional rules still involves considerable manual effort, such as entity labeling or sentence reconstruction [53], [54]. Fang et al. [55] proposes a knowledge graph that fuses computer vision with ontologies to dynamically recognize construction hazards while adhering to evolving safety standards. Zhou et al. [56] proposes a smart method for diagnosing wind turbine faults using ontology-based FMECA knowledge and a JESS rule engine to speed up maintenance decision-making. Zhou et al. [8] utilized the pretrained language model BERT for automated semantic annotation to capture the semantic information of sentences and then generated code from labeled sentences. Zhen et al. [9] established an ontology to represent domain knowledge and then generate SPARQL-based queries based on a pattern matching algorithm. Even though ready-to-use NLP tools for knowledge extraction exist, there is still room for refinement in structured text processing tools to more closely cater to the unique characteristics of specific fields.

In addition to the challenge of standardizing architectural design rules, another critical factor affecting the efficiency of ARC is the availability and accessibility of data. Zhang et al. [57] proposes a method that enhances automated compliance checking in building designs by merging compliance information into the IFC schema using machine learning and natural language processing. Although the IFC standard format is often favored for ARC [58], Malsane et al. [59]

points out that Building Information Modeling (BIM) models typically lack the necessary detail level for such checks. Given the interconnected nature of BIM design and code checking, expecting BIM modelers to include all necessary review details is impractical. Many commercial ACC systems still require manual input of certain data. Although machine learning has proven to be useful in semantically enriching and reconciling IFC data exchange issues [60], converting the extensive information present in BIM models into a machine learning-friendly format continues to be an overwhelming task [61].

Despite significant advancements in ARC within the AEC industry, extracting requirements and compliance information from detailed text documents remains a challenge. Innovations in NLP and knowledge graphs are improving the efficiency and accuracy of these systems, but the full automation of BIM reviews continues to be an ongoing effort. This paper builds upon existing research and, leveraging LLMs, seeks to transform natural language building rules into executable verification code with minimal human intervention. This marks a step towards the intelligent evolution of compliance checking in construction.

VII. CONCLUSION

In this work, we focus on the field of Automated Rule Checking (ARC) with the aim of reducing the manual effort required to convert natural language building rules into verification code for selected models. To achieve this, we propose ARCEAK, a novel LLM-based Automated Rule Checking framework Enhanced with Architectural Knowledge. ARCEAK achieves an almost fully automated process for converting natural language building rules into executable verification code. By consulting construction domain experts, we developed a robust construction domain-specific entity and event schema and established appropriate verification and evaluation metrics. The evaluation results of ARCEAK demonstrate outstanding performance in rule information extraction and an acceptable compile pass rate in verification code generation.

In the future, we plan to continue improving our framework. This includes, but is not limited to, expanding our methodology to cover a wider range of building rules and providing LLMs with related API lists during the Rule Checking Code Completion stage to enhance compatibility between the LLMs and selected models, which is crucial for improving the logic pass rate. Additionally, we aim to evaluate our framework on real architectural blueprints.

REFERENCES

- [1] S. M. Ilal and H. M. Günaydin, “Computer representation of building codes for automated compliance checking,” *Automation in construction*, vol. 82, pp. 43–58, 2017.
- [2] C. Eastman, J.-m. Lee, Y.-s. Jeong, and J.-k. Lee, “Automatic rule-based checking of building designs,” *Automation in construction*, vol. 18, no. 8, pp. 1011–1033, 2009.
- [3] L. Jiang and R. M. Leicht, “Automated rule-based constructability checking: Case study of formwork,” *Journal of Management in Engineering*, vol. 31, no. 1, p. A4014004, 2015.

- [4] T. A. Lin and C. T. Fatt, "Building smart—a strategy for implementing bim solution in singapore," *Synthesis Journal. Singapore*, pp. 117–124, 2006.
- [5] Z. Ma and N. Mao, "An algorithm for automatic generation of construction quality inspection points based on bim," *Journal of Tongji University (Natural Science)*, vol. 44, no. 5, pp. 725–729, 2015.
- [6] X. Xing, B. Zhong, H. L. H. GC, and G. Chen, "Automatic code compliance checking for design drawings of architecture major and its key technologies based on bim," *J. Civ. Eng. Manag*, vol. 36, no. 05, pp. 129–136, 2019.
- [7] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, 2-7 June 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186.
- [8] Y. Zhou, Z. Zheng, J. Lin, and X. Lu, "Integrating NLP and context-free grammar for complex rule interpretation towards automated compliance checking," *Comput. Ind.*, vol. 142, p. 103746, 2022.
- [9] Z. Zheng, Y. Zhou, X. Lu, and J. Lin, "Knowledge-informed semantic alignment and rule interpretation for automated compliance checking," *Automation in Construction*, vol. 142, p. 104524, 2022.
- [10] Z. Wang, Q. Xie, Y. Feng, Z. Ding, Z. Yang, and R. Xia, "Is chatgpt a good sentiment analyzer? a preliminary study," *arXiv preprint arXiv:2304.04339*, 2024.
- [11] X. Li and et al., "Are chatgpt and GPT-4 general-purpose solvers for financial text analytics? A study on several typical tasks," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: EMNLP 2023, Singapore, 6-10 December 2023*, M. Wang and I. Zitouni, Eds. Association for Computational Linguistics, 2023, pp. 408–422.
- [12] S. Parikh, M. Tiwari, P. Tumbade, and Q. Vohra, "Exploring zero and few-shot techniques for intent classification," in *Proceedings of the the 61st Annual Meeting of the Association for Computational Linguistics: Industry Track, ACL 2023, Toronto, Canada, 9-14 July 2023*, S. Sitaram, B. B. Klebanov, and J. D. Williams, Eds. Association for Computational Linguistics, 2023, pp. 744–751.
- [13] J. He, C. Treude, and D. Lo, "Llm-based multi-agent systems for software engineering: Vision and the road ahead," *arXiv preprint arXiv:2404.04834*, 2024.
- [14] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair," *arXiv preprint arXiv:2403.17134*, 2024.
- [15] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *Proceedings of the 36th International Conference on Neural Information Processing Systems, NIPS 2022, New Orleans, LA, USA, November 28 - December 9 2022*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds. Curran Associates Inc., 2022, pp. 22 199 – 22 213.
- [16] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, 6-9 May 2019*. OpenReview.net, 2019.
- [17] A. Wang and et al., "Superglue: A stickier benchmark for general-purpose language understanding systems," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems, NIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 3261–3275.
- [18] S. Thakur and et al., "Verigen: A large language model for verilog code generation," *ACM Trans. Design Autom. Electr. Syst.*, vol. 29, no. 3, pp. 46:1–46:31, 2024.
- [19] X. Jiang, Y. Dong, L. Wang, F. Zheng, Q. Shang, G. Li, Z. Jin, and W. Jiao, "Self-planning code generation with large language models," *ACM Trans. Softw. Eng. Methodol.*, 2024.
- [20] A. Vaswani and et al., "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS 2017, 4-9 December 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.
- [21] A. Chronopoulou, D. Stojanovski, and A. M. Fraser, "Reusing a pre-trained language model on languages with limited corpora for unsupervised NMT," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, 16-20 November 2020*, B. Webber, T. Cohn, Y. He, and Y. Liu, Eds. Association for Computational Linguistics, 2020, pp. 2703–2711.
- [22] A. C. Stickland, X. Li, and M. Ghazvininejad, "Recipes for adapting pre-trained monolingual and multilingual models to machine translation," in *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume, EACL 2021, Online, 19 - 23 April 2021*, P. Merlo, J. Tiedemann, and R. Tsafatay, Eds. Association for Computational Linguistics, 2021, pp. 3440–3453.
- [23] J. Li, H. Zhou, S. Huang, S. Cheng, and J. Chen, "Eliciting the translation ability of large language models via multilingual finetuning with translation instructions," *Trans. Assoc. Comput. Linguistics*, vol. 12, pp. 576–592, 2024.
- [24] T. Zhang, F. Ladzhak, E. Durmus, P. Liang, K. R. McKeown, and T. B. Hashimoto, "Benchmarking large language models for news summarization," *Trans. Assoc. Comput. Linguistics*, vol. 12, pp. 39–57, 2024.
- [25] M. Ravaut, A. Sun, N. F. Chen, and S. Joty, "On context utilization in summarization with large language models," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, 11-16 August 2024*, L. Ku, A. Martins, and V. Srikanth, Eds. Association for Computational Linguistics, 2024, pp. 2764–2781.
- [26] M. Chen, G. Li, L.-I. Wu, R. Liu, Y. Su, X. Chang, and J. Xue, "Can language models pretend solvers? logic code simulation with llms," *arXiv preprint arXiv:2403.16097*, 2024.
- [27] E. La Malfa and et al., "Code simulation challenges for large language models," *arXiv preprint arXiv:2401.09074*, 2024.
- [28] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever et al., "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
- [29] J. Wei and et al., "Chain-of-thought prompting elicits reasoning in large language models," in *Proceedings of the 36th International Conference on Neural Information Processing Systems, NIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds. Red Hook, NY, USA: Curran Associates Inc., 2022, pp. 24 824 – 24 837.
- [30] T. B. Brown and et al., "Language models are few-shot learners," in *Proceedings of the 34th International Conference on Neural Information Processing Systems, NIPS 2020, Virtual, 6-12 December 2020*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020.
- [31] X. Wang and et al., "Instructuiue: Multi-task instruction tuning for unified information extraction," *arXiv preprint arXiv:2304.08085*, 2023.
- [32] Y. Lu, Q. Liu, D. Dai, X. Xiao, H. Lin, X. Han, L. Sun, and H. Wu, "Unified structure generation for universal information extraction," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 5755–5772.
- [33] H. Yan, T. Gui, J. Dai, Q. Guo, Z. Zhang, and X. Qiu, "A unified generative framework for various NER subtasks," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), ACL/IJCNLP 2021, Virtual Event, 1-6 August 2021*, C. Zong, F. Xia, W. Li, and R. Navigli, Eds. Association for Computational Linguistics, 2021, pp. 5808–5822.
- [34] L. Cui, Y. Wu, J. Liu, S. Yang, and Y. Zhang, "Template-based named entity recognition using BART," in *Findings of the Association for Computational Linguistics, ACL/IJCNLP 2021, Online Event, 1-6 August 2021*, ser. Findings of ACL, C. Zong, F. Xia, W. Li, and R. Navigli, Eds., vol. ACL/IJCNLP 2021. Association for Computational Linguistics, 2021, pp. 1835–1845.
- [35] S. Yuan and et al., "Generative entity typing with curriculum learning," in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, 7-11 December 2022*, Y. Goldberg, Z. Kozareva, and Y. Zhang, Eds. Association for Computational Linguistics, 2022, pp. 3061–3073.
- [36] J. Qi and et al., "Mastering the task of open information extraction with large language models and consistent reasoning environment," *arXiv preprint arXiv:2310.10590*, 2023.
- [37] A. P. B. Veysey, V. Lai, F. Dernoncourt, and T. H. Nguyen, "Unleash GPT-2 power for event detection," in *Proceedings of the 59th Annual*

- Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing(Volume 1: Long Papers), ACL/IJCNLP 2021, Virtual Event, 1-6 August 2021, C. Zong, F. Xia, W. Li, and R. Navigli, Eds. Association for Computational Linguistics, 2021, pp. 6271–6282.*
- [38] S. Li, H. Ji, and J. Han, “Document-level event argument extraction by conditional generation,” in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, 6-11 June 2021*, K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tür, I. Beltagy, S. Bethard, R. Cotterell, T. Chakraborty, and Y. Zhou, Eds. Association for Computational Linguistics, 2021, pp. 894–908.
- [39] O. Sainz, I. García-Ferrero, R. Agerri, O. L. de Lacalle, G. Rigau, and E. Agirre, “Gollie: Annotation guidelines improve zero-shot information-extraction,” in *12th International Conference on Learning Representations, ICLR 2024, Vienna, Austria, 7-11 May 2024*. OpenReview.net, 2024.
- [40] X. Wei and et al., “Zero-shot information extraction via chatting with chatgpt,” *arXiv preprint arXiv:2302.10205*, 2023.
- [41] E. Nijkamp and et al., “Codegen: An open large language model for code with multi-turn program synthesis,” in *11th International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, 1-5 May 2023*. OpenReview.net, 2023.
- [42] M. Chen and et al., “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.
- [43] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, “Intellicode compose: code generation using transformer,” in *Proceedings of the 2020 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020, Virtual Event, USA, 8-13 November 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 1433–1443.
- [44] C. B. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan, “Pyamt5: multi-mode translation of natural language and python code with transformers,” in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, 16-20 November 2020*, B. Webber, T. Cohn, Y. He, and Y. Liu, Eds. Association for Computational Linguistics, 2020, pp. 9052–9065.
- [45] S. Lu and et al., “Codexglue: A machine learning benchmark dataset for code understanding and generation,” in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, Virtual, December 2021*, J. Vanschoren and S. Yeung, Eds., 2021.
- [46] Meta, “meta-llama/llama3,” <https://github.com/meta-llama/llama3>, 2024, [Online; accessed 1-Oct-2024].
- [47] OpenAI, “OpenAI/GPT4,” <https://openai.com/>, 2024, [Online; accessed 1-Oct-2024].
- [48] L. Wang and et al., “Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models,” in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, 9-14 July 2023*, A. Rogers, J. L. Boyd-Graber, and N. Okazaki, Eds. Association for Computational Linguistics, 2023, pp. 2609–2634.
- [49] E. Nijkamp and et al., “Codegen: An open large language model for code with multi-turn program synthesis,” in *11th International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.
- [50] S. J. Fenves, E. H. Gaylord, and S. K. Goel, “Decision table formulation of the 1969 aisc specification,” *Civil Engineering Studies SRS-347*, 1969.
- [51] J. Peng and X. Liu, “Automated code compliance checking research based on bim and knowledge graph,” *Scientific Reports*, vol. 13, no. 1, p. 7065, 2023.
- [52] A. S. Ismail, K. N. Ali, and N. A. Iahad, “A review on bim-based automated code compliance checking system,” in *2017 International Conference on Research and Innovation in Information Systems*. IEEE, 2017, pp. 1–6.
- [53] B. Zhong, C. Gan, H. Luo, and X. Xing, “Ontology-based framework for building environmental monitoring and compliance checking under bim environment,” *Building and Environment*, vol. 141, pp. 127–142, 2018.
- [54] T. H. Beach, J.-L. Hippolyte, and Y. Rezgui, “Towards the adoption of automated regulatory compliance checking in the built environment,” *Automation in construction*, vol. 118, p. 103285, 2020.
- [55] W. Fang, L. Ma, P. E. Love, H. Luo, L. Ding, and A. Zhou, “Knowledge graph for identifying hazards on construction sites: Integrating computer vision with ontology,” *Automation in Construction*, vol. 119, p. 103310, 2020.
- [56] A. Zhou, D. Yu, and W. Zhang, “A research on intelligent fault diagnosis of wind turbines based on ontology and fmeca,” *Advanced Engineering Informatics*, vol. 29, no. 1, pp. 115–125, 2015.
- [57] J. Zhang and N. M. El-Gohary, “Extending building information models semiautomatically using semantic natural language processing techniques,” *Journal of Computing in Civil Engineering*, vol. 30, no. 5, p. C4016004, 2016.
- [58] J. Melzner, S. Zhang, J. Teizer, and H.-J. Bargstädt, “A case study on automated safety compliance checking to assist fall protection design and planning in building information models,” *Construction Management and Economics*, vol. 31, no. 6, pp. 661–674, 2013.
- [59] S. Malsane, J. Matthews, S. Lockley, P. E. Love, and D. Greenwood, “Development of an object model for automated compliance checking,” *Automation in construction*, vol. 49, pp. 51–58, 2015.
- [60] B. Koo, S. La, N.-W. Cho, and Y. Yu, “Using support vector machines to classify building elements for checking the semantic integrity of building information models,” *Automation in Construction*, vol. 98, pp. 183–194, 2019.
- [61] R. Sacks, T. Bloch, M. Katz, and R. Yosef, “Automating design review with artificial intelligence and bim: State of the art and research framework,” in *ASCE International Conference on Computing in Civil Engineering 2019*. American Society of Civil Engineers Reston, VA, 2019, pp. 353–360.