

From Use Cases to Sequence Diagrams: Schema-Constrained Generation with Large Language Models

Chunhao Huang¹, Yuan Yao¹, Taolue Chen², Xiaoxing Ma¹

¹State Key Laboratory of Novel Software Technology, Nanjing University, China

²Birkbeck, University of London, UK

hch@smail.nju.edu.cn, {y.yao, xxm}@nju.edu.cn, t.chen@bbk.ac.uk

Abstract—Sequence diagrams are widely used to model interactions among system components and external actors by explicitly describing how operations are executed over time. However, constructing sequence diagrams manually is tedious and time-consuming in practice, and their automated generation remains largely unresolved. In this work, we study LLM-based sequence diagram generation from use cases. We first construct a high-quality dataset of industrial-scale use cases paired with corresponding sequence diagrams, supporting reproducible and extensible future research. We then investigate structured generation under schema constraints using both workflow and chain-of-thought (CoT) prompting strategies, enabling the generation of complex sequence diagrams with nested combined fragments. In addition, we propose an evaluation framework that measures not only the correctness of sequential messages but also the structural validity of nested control structures. Experimental results demonstrate that schema constraints together with CoT/workflow guidance substantially improve generation quality over a basic LLM-only approach.

Index Terms—Sequence Diagram Generation, Large Language Models, Schema-Constrained Generation, Prompt Engineering,

I. INTRODUCTION

Sequence diagram is a fundamental behavioral modeling tool in software engineering, widely used to capture and visualize interactions among system components and external actors [1]–[4]. There are many elements in a sequence diagram [5], among which the most important ones include *participants*, *messages*, and *combined fragments*. Participants are depicted as lifelines in a sequence diagram with their heads containing the participant names. These participants usually include system components and outside actors, which can be human (e.g., lecturer and student) or non-human (e.g., server, printer, and executable software). A message (or an interaction) consists of a sender, a receiver, and the triggered action. Combined fragments (or operators) [6] are used to model various control structures. For example, `alt` and `opt` can be used to model branches, `loop` operator for modeling loops, `break` operator for modeling exceptions, `seq` and `strict` for specifying order, `par` and `critical` for specifying concurrent or atomic interactions. Combined fragments can be arbitrarily nested.

Automated generation of sequence diagrams from requirements is an appealing yet challenging problem. Early studies focused mainly on identifying participants and basic messages,

using NLP techniques [7], [8] to recognize the grammar structure of sentences and producing interactions in a rule-based manner. However, they have primarily targeted simple diagrams and failed to capture advanced constructs such as branches and loops. Recent studies have conducted preliminary experiments and verified the ability of large language models (LLMs) in generating sequence diagrams from use cases [9]–[12]. However, as indicated by these studies, LLMs are still unreliable to generate a correct and complete diagram, and tend to produce both syntactic errors and semantic errors due to their hallucination limitation. This problem becomes more severe when the use case [13], [14] becomes more complex.

To this end, we systematically investigate LLM-based sequence diagram generation, including real-world benchmark datasets, reproducible generation methods, and structural evaluation criteria.

Dataset construction. We construct a high-quality dataset of industrial-scale use cases paired with manually annotated sequence diagrams. The dataset is derived from the real-world *SuperFrog Scheduler* project. The requirements documentation contains 26 user-goal level use cases that involve multiple participants and non-trivial control-flow logic. Unlike many existing datasets that focus on simple interactions, most use cases in our dataset include extensions/conditional branches, or iterative behaviors, which naturally translate into nested combined fragments in sequence diagrams. The sequence diagrams are annotated using a custom-designed JSON schema, which not only restricts the generation task to conform to the syntax of sequence diagrams, but also supports the automatic generation of PlantUML diagrams for visualization and manual inspection.

Diagram generation approaches. We investigate multiple strategies for generating sequence diagrams from natural-language use cases using large language models. In particular, we systematically compare direct generation, schema-constrained generation, chain-of-thought (CoT) prompting, and a multi-stage workflow-based generation approach. The latter two are also schema-constrained and use nearly identical prompts. The difference is that CoT lets LLMs reason step by step in a single generation call, while the workflow-based approach decomposes the generation task into smaller subtasks

and imitates the reasoning process by passing intermediate results. Through controlled experiments on DeepSeek-V3.2 and GPT-5.1, we find that syntax errors are eliminated after introducing the schema constraint, and the CoT and workflow-based approaches further improve the semantic correctness of generated diagrams by providing more detailed guidance and structured reasoning.

Evaluation criteria. We propose an automated evaluation framework for generated sequence diagrams. Existing work typically relies on manual inspection or coarse-grained similarity measures to check the quality of generated sequence diagrams, which are insufficient for evaluating complex behavioral models. In contrast, our framework evaluates sequence diagrams at multiple structural levels, including participants, messages, and combined fragments. To support automated evaluation, we design an intermediate representation and develop analysis procedures that extract structural information from generated diagrams. This enables precise and automated measurement of different diagram components, providing a fine-grained view of model performance.

II. RELATED WORK

Automated diagram generation has been an active research topic in software engineering for decades. Existing approaches to UML diagram generation can generally be divided into two categories: rule-based methods and data-driven approaches.

Rule-based methods. Earlier work on diagram generation focused on transforming natural-language requirements into UML models using rule-based and linguistic analysis techniques [15]–[20]. These approaches typically apply predefined syntactic rules to extract modeling elements such as classes, actors, and interactions. For example, Karaa et al. [21] defined more than 1000 patterns for the extraction of the class diagram concepts. Ramzan et al. [22] combined NLP-based concept extraction with machine learning methods to improve the accuracy of UML class diagram generation, particularly for identifying relationships between classes. Alashqar [23] proposed a scenario-based method that generates UML diagrams from structured requirement templates, reducing ambiguity in natural language specifications. Jahan et al. [9] presented an automated approach towards generating UML sequence diagrams from textual use cases written in natural language. The approach uses different NLP techniques combined with some rule-based decision approaches to identify problem level objects and interactions.

Although rule-based methods provide interpretable pipelines and deterministic outputs, they typically require carefully designed linguistic rules and struggle with ambiguous or complex requirements.

Data-driven methods. Recent advances in large language models [24]–[27] have enabled a new generation of diagram synthesis approaches. Unlike traditional NLP pipelines, LLMs can capture contextual semantics [28], [29] and generate structured diagram representations directly from textual requirements [10], [11]. For example, Chen et al. [30] proposed a novel abstraction-concretization framework that enhances

the consistency and quality of generated graph models by considering multiple outputs from an LLM. Ferrari et al. [11] investigated the use of ChatGPT for generating UML sequence diagrams from natural-language requirements and observed that the generated diagrams generally follow UML conventions but may contain structural inaccuracies and omissions. Xiao et al. [12] conducted a multi-model evaluation of LLM-based UML sequence diagram generation and showed that modern language models can produce diagrams with reasonable clarity and correctness across multiple domains, although complex interaction scenarios remain challenging.

The above work has shown that LLM-based methods can significantly outperform traditional rule-based pipelines in UML diagram generation tasks, but often lack explicit structural guarantees and may produce inconsistent diagram elements. This work mitigates this gap by focusing on the structured extraction and analysis of diagram elements in the context of sequence diagram generation.

III. MOTIVATION

Our experiments show that when GPT 5.1 is prompted to directly generate PlantUML code according to use case descriptions, 26.9% of the generated sequence diagrams have syntactic errors leading to compilation failure. Here, we present an example in Table I. This use case (named UC-17) has passed the compilation, and it involves non-trivial control-flow logic consisting of conditional and iterative control structures, which translates into nested combined fragments (loops and branches) in the corresponding sequence diagrams. It has three main branches: create, edit, and delete events. The latter two branches are omitted in the use case description and corresponding sequence diagrams for brevity.

Figure 1 shows the result of UC-17 generated by GPT 5.1, which exhibits various problems. (1) **Imagined participants:** two extra participants are “designed” by GPT 5.1 and are not mentioned in the use case. (2) **Verbose messages:** the generated messages are more verbose and less focused on the core interactions between the Spirit Director and the system. For example, a simple action of validation is split into 6 interactions among three participants, which makes the diagram more complex and less comprehensible. Other needless messages exist as well because of unnecessary structural elements. (3) **Incorrect control flow:** As stated explicitly in UC-17, there should be a loop between step 4 and step 7. Figure 1 fails to capture this loop and instead generates an `alt` fragment with a “return to a previous step” message. However, this is not legal in a sequence diagram. The absence of this loop causes inevitable logic errors. Besides that, unnecessary operands such as “cancel creation” in the `alt` fragment and “done managing” in the outer loop are making the diagram more tedious.

From this example, we can see that it is challenging for LLMs to directly generate a valid sequence diagram when faced with a complex use case like UC-17. To unleash the ability of LLMs in this specific task, we combined several tactics, including setting up pre-defined output format constraints,

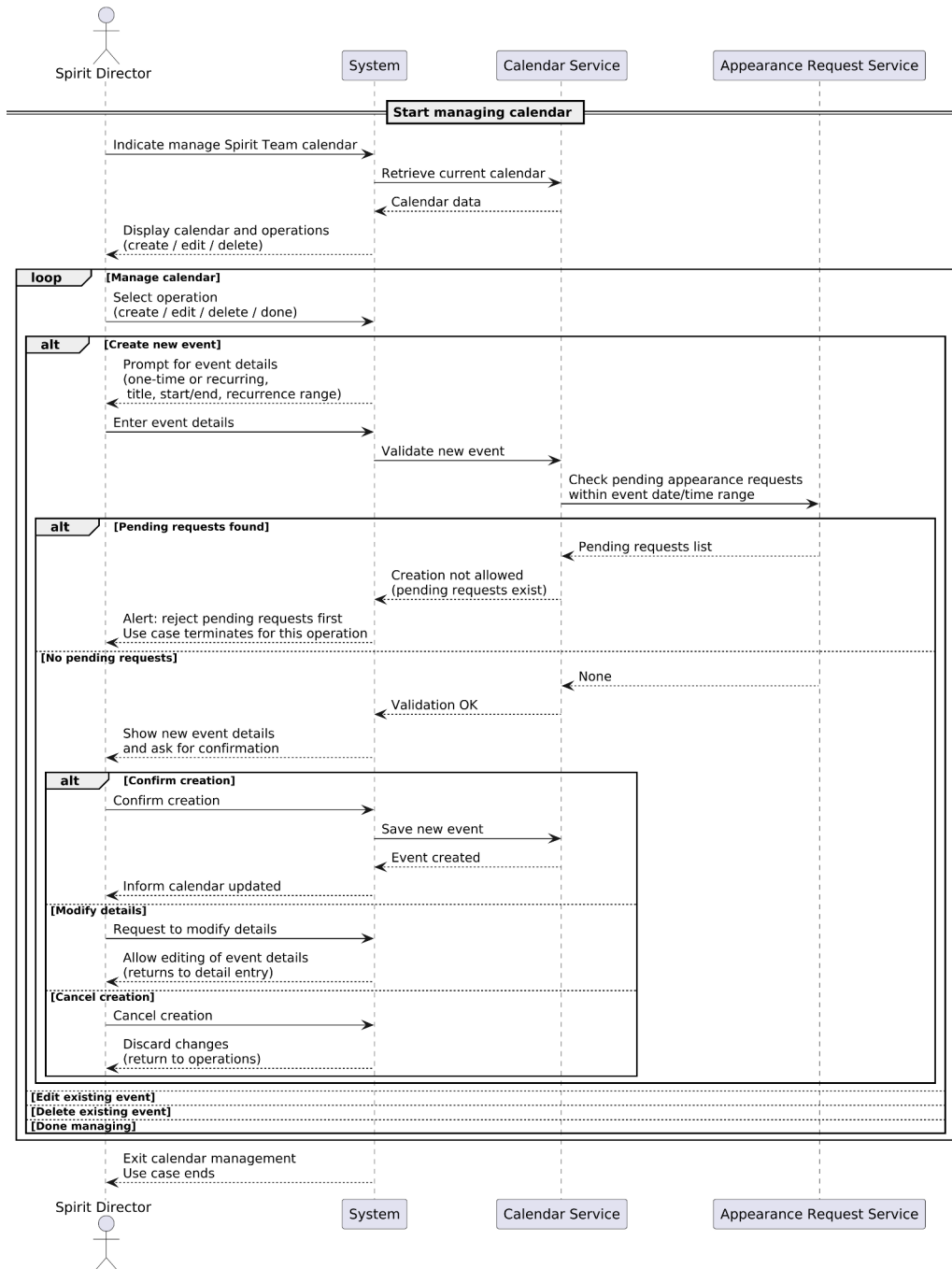


Fig. 1: The sequence diagram of UC-17 generated by GPT 5.1 with basic prompting.

TABLE I: Abbreviated Use Case 17 of SuperFrog Scheduler. The complete use cases can be found in our project website (see the ‘data and code availability’ section).

UC ID and Name	UC-17: Manage the Spirit Team calendar
Primary Actor	Spirit Director
Secondary Actors	None
Trigger	The Spirit Director indicates to manage the Spirit Team calendar.
Description	The Spirit Director creates events on the Spirit Team calendar to block out time for dead (study) days, finals week, or major holidays so that customers cannot request SuperFrog appearances during those periods. The Spirit Director can also edit and delete calendar events to maintain accurate scheduling information.
Main Success Scenario	<ol style="list-style-type: none"> 1. The Spirit Director indicates to manage the Spirit Team calendar. 2. The system displays the calendar and asks the Spirit Director to select the operations: create (step 4–8), edit (step 9–13), or delete (step 14–17) events. 3. The Spirit Director selects one of the three operations. <p>Based on the Spirit Director’s selection in step 3, the flow goes through either step 4–8, or step 9–13, or step 14–17.</p> <ol style="list-style-type: none"> 4. The Spirit Director selects to create either a one-time event (e.g., finals week) or a recurring event. For each event, the Spirit Director provides: Event title (e.g., finals week) , Start date and time and end date and time, Recurrence start date and end date if applicable 5. The system validates the creation. 6. The System displays the details of the new event and asks the Spirit Director to confirm the creation. 7. The Spirit Director either confirms the creation (continues the normal flow) or chooses to modify the details (return to step 4). 8. The System saves the creation and informs the Spirit Director that the calendar has been updated. <p>(step 9–17 omitted)</p> <ol style="list-style-type: none"> 18. The Spirit Director either indicates that she is done (goes to step 19) or continues to manage the calendar (returns to step 2). 19. Use case ends.
Extensions	<ol style="list-style-type: none"> 5a. Pending appearance requests exist in the selected time range. <ol style="list-style-type: none"> 5a1. The system alerts the Spirit Director to reject pending requests. 5a2. Use case terminates. <p>(extensions in step 9–17 omitted)</p>

well-structured and detailed prompts [31], task decomposition, etc. As will be later shown in Figure 4, the generation result of UC-17 with schema constraint and CoT prompting is significantly improved. The participants and messages are more accurate and concise, and the control flow is easier to follow. The diagram is more comprehensible and informative to readers. We will present more details about our approach and evaluation results in the following sections.

IV. OUR APPROACHES

A. Problem Definition

1) *Use case scenario*: A use case scenario is a structured description of interactions among involved participants that realize a specific functional goal. As the example in Table I shows, a typical use case usually consists of use case name, actor list, trigger, description, pre-conditions, post-conditions, main success scenario, extensions, associated information, etc.

2) *Sequence diagram*: A sequence diagram is represented as a structured model consisting of a set of participants and an ordered sequence of diagram elements. Diagram elements include messages, combined fragments, and references.

Messages represent directed interactions between participants and contain a sender, a receiver, a message label, and an optional time annotation.

Combined fragments describe structured control flow in the interaction. Each fragment has an operator (e.g., `alt`, `opt`, or `loop`) and one or more operands. An operand contains an optional guard condition and a sequence of diagram elements. Since operands may contain other fragments, the structure is inherently recursive.

References represent references to external interaction descriptions or reused interaction patterns. A reference element encapsulates a sequence of diagram elements, allowing complex interactions to be modularized and reused.

We concentrate on the generation of the content and core interaction logic of sequence diagrams. The following elements are deliberately excluded from the scope of generation:

- Lifeline activation/deactivation;
- Message types (synchronous/asynchronous, return messages, etc.);
- Seldom-used combined fragments (e.g., `neg`, `assert`);
- Stylistic elements (notes, comments, etc.);
- Other components (e.g., state invariants, continuation markers, etc.).

3) *Generation task*: We define the task as the generation of sequence diagrams from use case scenarios. The generated diagram is expected to satisfy three requirements:

- **Schema validity**: the diagram conforms to the pre-defined schema;

- **Interaction grounding:** diagram elements correspond to interactions described in the use case;
- **Ordering preservation:** the relative order of interactions follows the use case flow.

The problem can therefore be characterized as: automatically translating natural-language interaction descriptions into schema-valid hierarchical interaction models while preserving actor semantics and control-flow structure.

B. Dataset Construction

To evaluate the performance of our approach in generating sequence diagrams, we utilize a real-world software project *SuperFrog Scheduler*, which is a medium-sized web-based request management system designed for Texas Christian University (TCU). The system manages the scheduling of the university's mascot "SuperFrog" for various events. The team documented 26 user-goal level use cases [32], covering intricate scheduling logic typical of enterprise-grade software.

This project was selected due to its representative complexity and multi-stakeholder nature. Most use cases involve interactions among at least three participants. Only two of the 26 use cases are of pure linear sequence, and all the rest use cases include extensions besides main success scenarios, or branches in main success scenarios, which can lead to branches or loops in the corresponding sequence diagram.

We perform a rigorous manual annotation of the 26 use cases to establish a high-quality ground truth for our experiments. For each use case, we authored two primary representations of the corresponding sequence diagram:

- 1) **Structured JSON:** To facilitate automated evaluation and ensure adherence to structured specification, we designed a custom JSON schema, which limits the task of sequence diagram generation to focusing on the generation of participants, messages and the selected combined fragments, neglecting styles, comments, notes, etc. Each use case was manually mapped to this schema.
- 2) **PlantUML [33], [34]:** This is a standard domain-specific language (DSL) for visualizing sequence diagrams, used for manual evaluation. We implement a tool to automatically generate the DSL from the JSON files.

Our manual annotation adhere to the following principles:

- The sequence diagrams should represent the original use case correctly and completely as much as possible;
- The sequence diagrams should be as simple as possible, without redundant participants or messages, and with a clear structure that reflects the logic of the use case;
- When there are multiple valid representations, the one that best conforms to the structure of the original use case is preferred.

The final dataset consists of 26 requirement-diagram pairs, providing a robust benchmark for evaluating the alignment between natural language requirements and behavioral models.

C. Schema Constraint

We design a structural specification for sequence diagrams using JSON schema as the meta-modeling language, which

serves as the target output format for our generation approach. This specification serves as an abstract syntax that precisely describes the required components and their relationships. As shown in Figure 2, the scope of our sequence diagram generation is limited to the generation of participants/actors, messages and the selected combined fragments. A valid sequence diagram is a JSON object with a fixed type field ('sd'), a title, a non-empty list of actors (each identified by a unique integer id and a name), and a top-level sequence array of diagram elements. The elements are defined polymorphically via a 'oneOf' construct, supporting atomic messages and combined fragments such as alt, opt, loop, par and others.

Crucially, the specification is recursive: combined fragments (e.g., alt, opt, loop) may contain nested diagram elements, which may further include combined fragments. This recursion faithfully captures the hierarchical nature of sequence diagrams, enabling arbitrarily deep nesting of control structures.

D. Methodology

The generation approaches are built upon LLMs. In addition to direct generation, we propose schema-constrained generation, and two structured generation approaches: CoT prompting and workflow-based generation.

1) *Schema-constrained generation:* This approach prompts LLMs to generate sequence diagrams following the JSON schema we defined. We utilize the LLMs' json mode to enforce syntactically valid JSON output.

2) *CoT prompting:* The structure of the CoT prompting is shown in Figure 3. The prompt begins with an overall task description that specifies the role of the model as an expert in requirements engineering and UML modeling. The task definition constrains the model to generate sequence diagrams strictly according to a predefined JSON schema. In addition, explicit chain-of-thought instructions are provided to guide the reasoning process before producing the final result.

The reasoning process is divided into five major steps. In the *Extract Information* step, the model transforms the original use case text into a structured intermediate representation containing the title, description, actors, main success scenario, and extensions. This step reduces noise in the use cases and ensures that interaction-related information is preserved.

The *Determine Actors* step refines the set of participants involved in the interactions. Since the actors explicitly specified in the use case may not be the corresponding participants in the sequence diagram, the model is instructed to remove redundant actors and include missing ones so that all actors involved in the interactions are represented in the final diagram.

The *Identify Loops* step focuses on detecting iterative behaviors described in the use cases. We observed in preliminary experiments that an unidentified loop largely affects the quality of the generated diagrams. Loops are identified through phrases such as "return to step n " or "continue from step n ". The prompt explicitly requires that all detected loops must be represented in the generated sequence diagram. Additional

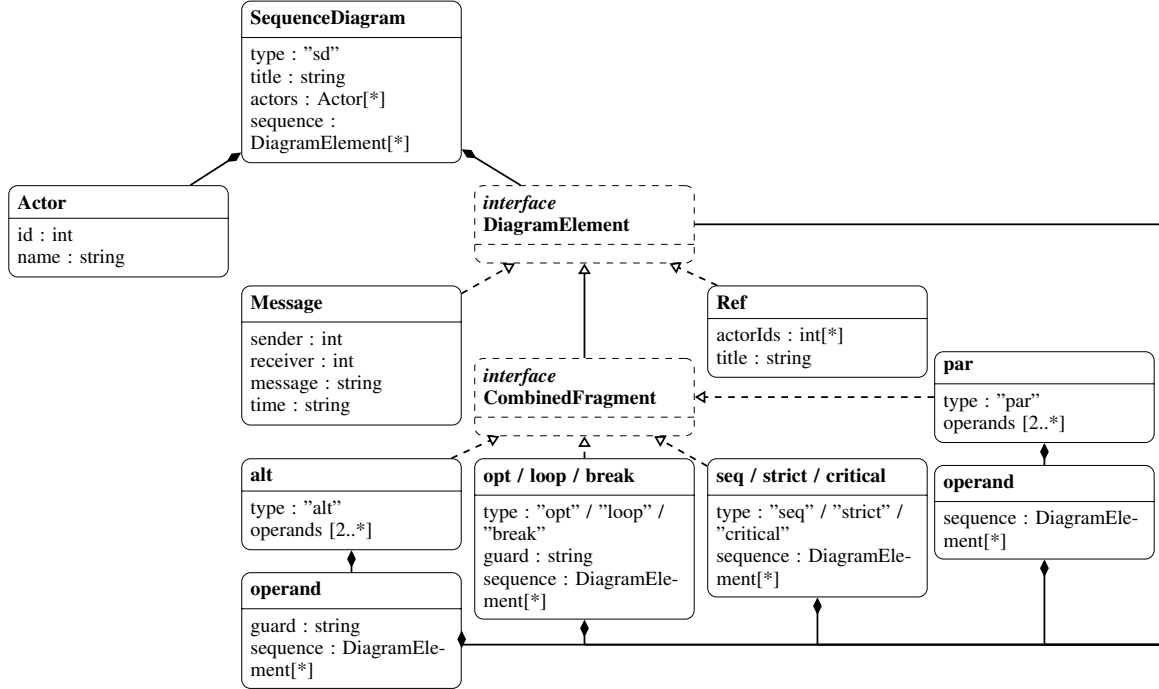


Fig. 2: Meta-model representation of the JSON schema for LLM-generated sequence diagrams. The model defines diagram structure, actor entities, and polymorphic diagram elements including message interactions and combined fragments.

constraints are imposed to prevent ambiguous overlapping loops while allowing nested loops.

In the *Construct Diagram* step, we instruct the model to generate the sequence diagram in JSON format according to the schema specification. We provide domain-specific knowledge, detailed instructions and short examples curated for generating messages and combined fragments correctly.

The *Check and Revise* step requires the model to verify the logical consistency and completeness of the generated diagram before producing the final result. This step helps reduce semantic inconsistencies between use cases and the generated diagrams.

After each reasoning step, the model outputs the reasoning process, and in the end the final JSON object. The JSON schema and few-shot examples are appended to the prompt to provide output constraints and reference solutions. The examples are composed manually and their content is completely irrelevant to the use cases in our dataset.

3) *Workflow-based generation*: The workflow-based generation approach uses similar prompts but decomposes the generation task into multiple model calls. Instead of producing the entire diagram in a single inference as in CoT prompting, the workflow approach invokes the model sequentially for subtasks such as information extraction, actor identification, and diagram construction. The intermediate outputs are processed and reused as inputs for subsequent steps. This decomposition reduces the cognitive load of each generation step and improves stability for complex use cases. In preliminary

experiments, we employed a workflow with check and revise loop after the generation step, but we found that the check and revise loop does not improve the generation quality much, and sometimes it makes the generation result worse and more complex. We therefore remove this loop in the final design of workflow-based generation, but retain this step in the CoT prompting design.

Compared with direct prompting, the structured prompting strategies provide clearer task decomposition, stronger output constraints, and improved controllability, which are essential for generating complex sequence diagrams from industrial requirement documents.

V. EVALUATION

We conducted experiments on four configurations: (1) **basic** (direct generation), which directly prompts LLMs to generate a sequence diagram in the form of plantUml code; (2) **schema-only**, which prompts LLMs to generate a sequence diagram according to the predefined JSON schema; (3) **CoT prompting**, which provides LLMs with instructions specified in Figure 3 to perform step-by-step reasoning and generation in a single call; (4) **workflow-based generation**, which adopts similar prompts as in CoT prompting, but decomposes the generation task into multiple steps. The experiments are conducted on two LLMs: DeepSeek-V3.2 and GPT 5.1. We use the chat model of DeepSeek-V3.2 with reasoning mode off. The temperature is set to 0 for both models.

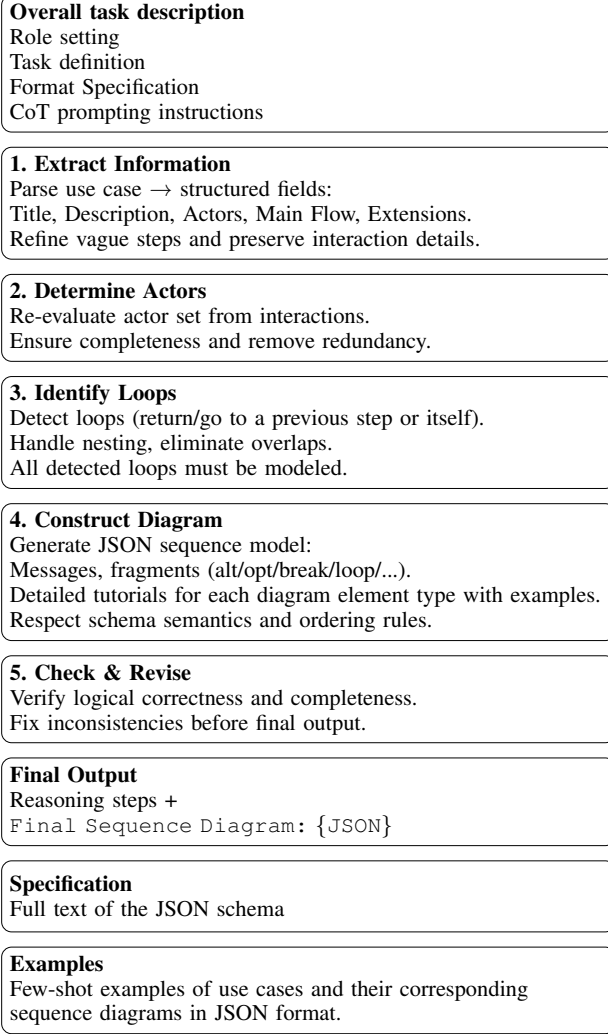


Fig. 3: Structure of the chain-of-thought prompt

A. Evaluation Framework

To rigorously assess the quality of the generated sequence diagrams, we evaluate the alignment between the model-generated diagrams and the ground truth across three key dimensions: participants, messages, and jump scenarios.

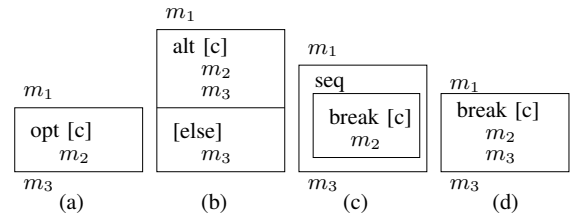
- **Participants:** This dimension evaluates the model’s ability to correctly identify the participants involved in the use case.
- **Messages:** This dimension evaluates the correctness of interaction content in the generated sequence diagrams, including both basic messages and references to other diagrams (ref).
- **Jump Scenarios:** We propose this dimension to evaluate the structural correctness of sequence diagrams. A jump scenario is defined as “ $msg_before \xrightarrow{condition} msg_after$ ”, where the conditions come from the combined fragments with guards (i.e., alt/opt/loop/break).

Participants and Messages are evaluated to ensure the content correctness of the generated diagrams, while Jump Scenarios are proposed to evaluate the structural and logical correctness of the generated diagrams. The execution of each sequence diagram can be modeled into a control-flow graph (CFG), where the nodes represent messages or combined fragments and the edges represent jumps with or without conditions. Since UML is a flexible modeling language, the same logic can be represented in different ways, which may lead to CFGs with significant distinctions. As a result, it is difficult and not appropriate to evaluate sequence diagrams by graph isomorphism or graph matching. This is why we propose to evaluate the jump scenarios, which are more fine-grained and easier to compare. Previous work has observed that LLMs make nearly no mistakes when faced with pure linear sequences. Therefore, under the assumption that LLMs can generate sequence snippets in correct order in each nested layer, we focus on verifying possible interactions right before and after every condition judgment, which can reflect local logic correctness, the overall logic correctness is also guaranteed to a large extent. This approach meets the necessity of the correctness of the diagram structure.

We implement a lightweight parser that transforms a PlantUML sequence diagram into a structured representation and then derive Participants, Messages, and Jump Scenarios from it. The core procedure is summarized in Algorithm 1. The functions GETPRECEDINGMESSAGES and GETSUBSEQUENTMESSAGES are designed to retrieve messages (references are regarded as normal messages as well) that may occur right before and after a node, respectively.

Now we discuss the equivalence of Jump Scenarios produced by the same branch situation in different ways of expression.

(1) **one alternative path.** Possible expressions include:



All these expressions yield the same jump scenarios $m_1 \xrightarrow{c} m_2$ and $m_1 \xrightarrow{c} m_3$. In (c), m_1 can be placed inside `seq` as well, which does not change the logic. Note that we use a single message for simplicity; each message can be extended to a sequence of diagram elements. For example, in (a), if m_1 is a diagram element, we can replace it with the result of GETPRECEDINGMESSAGES (OPT). This also applies in the following part.

(2) **multiple alternative paths.** Possible expressions include (a) and (b) of the following diagram. Both expressions yield the same jump scenarios $m_1 \xrightarrow{c_1} m_2$, $m_1 \xrightarrow{c_2} m_3$, and $m_1 \xrightarrow{c_1 \wedge c_2} m_4$. In (a), m_4 can be placed inside `alt` with an extra `else` operand as well.

(3) **nested alternative paths.** Possible expressions include (c)

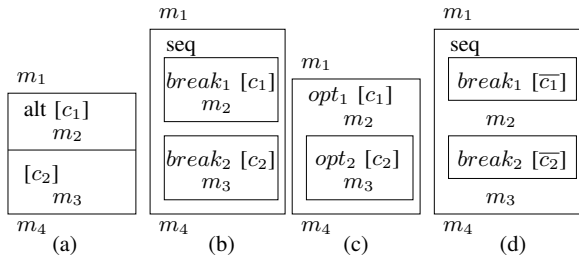
Algorithm 1 Sequence Diagram Parser

```

▷ Input:
F := PlantUML file
▷ Output:
P := Participants; M := Messages; J := Jump Scenarios

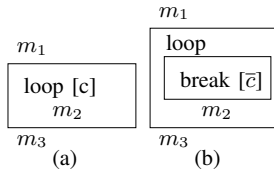
procedure EXTRACTFROMPUML(F)
  P, M, Parsed Items I ← PARSEPUML(F)
  Abstract syntax tree (AST) T ← BUILDAST(I)
  Control-flow graph (CFG) G ← BUILDCFG(T)
  E ← edges of G
  J ← ∅
  ▷ branch nodes: alt/opt/loop/break
  for all branch nodes b ∈ G do
    Eb ← {e ∈ E | e.start = b}
    if |Eb| = 1 and Eb[0].condition = ∅ then
      continue
    Ep ← GETPRECEDINGMESSAGES(b)
    for all eb ∈ Eb do
      cond ← eb.condition
      if eb.end is a Message or eb.end = G.end or eb.end
      is ref node then
        Es ← {eb.end}
      else
        Es ← GETSUBSEQUENTMESSAGES(eb.end)
      for all ep ∈ Ep do
        for all es ∈ Es do
          J ← J ∪ {(ep, cond, es)}
    output P, M, J
  
```

and (d) of the following diagram. Both expressions yield the same jump scenarios $m_1 \xrightarrow{c_1} m_2$, $m_1 \xrightarrow{\bar{c}_1} m_4$, $m_2 \xrightarrow{c_2} m_3$, and $m_2 \xrightarrow{\bar{c}_2} m_4$. We choose *opt* here for simplicity; it can be easily proved that this applies to *alt* as well.



(4) **loop**. Besides the guard condition of a *loop*, *break* is the only way to jump out of a loop. So we consider the following three situations:

(4.1) **exit at the top**. Possible expressions include

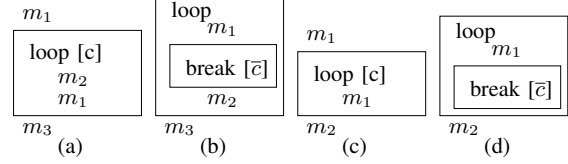


Both expressions yield the same jump scenarios $m_1 \xrightarrow{c} m_2$, $m_1 \xrightarrow{\bar{c}} m_3$, $m_2 \xrightarrow{c} m_2$ and $m_2 \xrightarrow{\bar{c}} m_3$.

(4.2) **exit in the middle**: Possible expressions include (a) and (b) of the following diagram. Both expressions yield the same jump scenarios $m_1 \xrightarrow{c} m_2$ and $m_1 \xrightarrow{\bar{c}} m_3$.

(4.3) **exit at the bottom**: Possible expressions include (c) and

(d) of the following diagram. Both expressions yield the same jump scenarios $m_1 \xrightarrow{c} m_1$ and $m_1 \xrightarrow{\bar{c}} m_2$.



B. Evaluation Metrics

Given that LLMs may use synonyms for the same software entities (e.g., “sys” or “system”) and generate different sentences for the same message or condition (e.g., “input validation passed” or “validation success”), we employ GPT 5.1 as an LLM-as-a-Judge to perform semantic matching between the generated diagrams and the baseline. For each dimension, we extract the elements from both the baseline and the evaluated PlantUML files. We then prompt GPT 5.1 to perform a 1-to-1 mapping on participants. Then we divide messages into groups such that messages in the same group have the same sender and receiver, and perform 1-to-1 mapping within each group. We do the same for jump scenarios, which are grouped by the same preceding message and subsequent message. The mapping results are then used to calculate **Precision**, **Recall** and **F1-score** for each dimension.¹ Additionally, we report the **Error Rate**, defined as the proportion of use cases where the generated code failed to compile. Diagrams that fail to compile are excluded from the calculation of average Precision/Recall/F1 scores.

Human Validation of LLM-as-a-Judge. Since the above evaluation metric relies on GPT 5.1 for semantic matching, we additionally conduct a manual validation of the LLM-as-a-Judge semantic matching process. A subset of 10 use cases was randomly selected, covering a range of structural complexity including linear flows, alternative paths, loops, and nested combined fragments. Two expert evaluators independently verified the correctness of the 1-to-1 mappings produced by GPT 5.1 for participants, messages, and jump scenarios. When the expert evaluators disagree, they reach a final consensus through discussion. For each dimension, we computed the matching accuracy (i.e., whether the matching results of GPT 5.1 align with those of expert evaluators) over the 10 use cases, and the results are summarized in Table III. Overall, the manual validation confirmed that the LLM-as-a-Judge is highly reliable for participant and message mappings, with some discrepancies observed in the jump scenario mappings. Since jump scenarios in the same group share the same preceding and subsequent messages, we provide GPT 5.1 with only the conditional statements to do the mapping. GPT 5.1 may make mistakes when faced with complex conditional statements such

¹Note that we conduct 1-to-1 mapping in each dimension to cover the measurement of redundancy. For example, if the baseline contains two messages but the generated diagram contains four messages (suppose the extra two messages can be mapped to messages of the baseline as well), the maximum number of matches is two, which leads to a precision of 0.5 and a recall of 1.0.

TABLE II: Evaluation results (P/R/F1) for different sequence diagram generation methods. Significant improvements are observed when schema constraints and CoT/workflow reasoning are included.

Method	Participants (P/R/F1)	Messages (P/R/F1)	Jump Scenarios (P/R/F1)	Compile Error Rate
deepseek-chat-basic	0.844 / 0.992 / 0.904	0.596 / 0.801 / 0.673	0.593 / 0.291 / 0.346	1/26 (3.8%)
deepseek-chat-schema-only	0.979 / 0.992 / 0.984	0.837 / 0.772 / 0.797	0.517 / 0.423 / 0.411	0/26 (0.0%)
deepseek-chat-workflow	1.000 / 0.985 / 0.991	0.870 / 0.857 / 0.861	0.482 / 0.434 / 0.401	0/26 (0.0%)
deepseek-chat-CoT	0.979 / 0.985 / 0.978	0.922 / 0.868 / 0.891	0.346 / 0.480 / 0.358	0/26 (0.0%)
gpt5.1-basic	0.598 / 1.000 / 0.719	0.508 / 0.729 / 0.585	0.395 / 0.383 / 0.334	7/26 (26.9%)
gpt5.1-schema-only	0.840 / 0.992 / 0.891	0.623 / 0.839 / 0.705	0.382 / 0.478 / 0.386	0/26 (0.0%)
gpt5.1-workflow	0.992 / 0.985 / 0.987	0.728 / 0.881 / 0.792	0.336 / 0.532 / 0.353	0/26 (0.0%)
gpt5.1-CoT	1.000 / 1.000 / 1.000	0.813 / 0.906 / 0.852	0.356 / 0.598 / 0.409	0/26 (0.0%)

TABLE III: Manual validation results of LLM-as-a-Judge semantic matching.

Dimension	Avg Accuracy	Min	Max
Actor	100%	100%	100%
Message	100%	100%	100%
JumpScenario	92.1%	72.4%	100%

as “ $\text{not}(c_1 \wedge c_2)$ ”. However, the average accuracy of 92.1% for jump scenarios indicates that the majority of mappings are correct.

C. Experimental Results and Analysis

Table II summarizes the Precision (P), Recall (R), and F1-score for each method, alongside the syntax error rate.

1) *Overall Impact of Schema Constraints*: A notable difference emerges in syntax robustness under basic prompting. GPT 5.1 exhibits a high compile error rate (26.9%) when generating diagrams directly, whereas DeepSeek-V3.2 shows only 3.8%, because GPT 5.1 tends to generate more comments, notes or keywords (e.g. “continue”, “stop”) that are not compliant with the syntax of PlantUml. However, syntax errors are eliminated after schema constraints are applied in both models, demonstrating the effectiveness of schema-based prompting in ensuring syntactic correctness. Besides that, across both DeepSeek-V3.2 and GPT 5.1, introducing the JSON schema consistently improves overall performance compared to direct generation (basic): Participants, Messages and Jump Scenarios all achieve markedly higher F1-scores. These results indicate that schema constraint effectively guides the model toward producing syntactically valid and semantically aligned outputs.

2) *Extraction of Participants and Messages*: Although participant identification is generally the easiest task, our approach still yields steadily improved F1-scores across both models. DeepSeek-V3.2 achieves high F1-scores across all settings (above 0.90), while GPT 5.1 reaches perfect precision and recall (1.000/1.000/1.000) under the CoT configuration. This suggests that schema constraints and structured reasoning can further enhance the model’s ability to accurately identify participants, even when the task is relatively straightforward. Message performance shows significant improvement as well. For DeepSeek-V3.2, the CoT configuration yields the highest Message F1-score (0.891), followed closely by the workflow

configuration (0.861). For GPT 5.1, CoT again achieves the best Message F1-score (0.852), outperforming both schema-only and workflow settings.

3) *Jump Scenario Detection*: Jump Scenarios represent the most challenging dimension, as they require the correct modeling of conditional control flow. Both models exhibit lower F1-scores in this category compared to Participants and Messages, which is not only due to the inherent complexity but also the rigorous mapping algorithm as stated in Section V-B. DeepSeek-V3.2 achieves its best Jump Scenario F1-score under the schema-only setting (0.411), while GPT 5.1 performs best under CoT (0.409). However, CoT prompting and workflow improve recall significantly compared with schema-only. It indicates that when these two methods are trying to model the branching and looping logic correctly, they refactored the diagram structure, which leads to more structure redundancy.

4) *Workflow vs. CoT*: Both workflow and CoT configurations incorporate schema guidance, but differ in invocation strategy: CoT performs reasoning in a single LLM call, whereas workflow decomposes the task into multiple calls. For DeepSeek-V3.2, workflow provides the strongest performance overall with only a small decline in Message F1-score compared to CoT. For GPT 5.1, CoT consistently achieves the highest metric scores across all 3 dimensions. GPT 5.1 is a reasoning model by default, while in this experiment, we use DeepSeek-V3.2 in non-reasoning mode, i.e. DeepSeek-chat. These findings suggest that single-call reasoning may be more appropriate for reasoning models, because no information is lost in the context, while multi-step workflow orchestration may be more suitable for non-reasoning models, because it purifies the context for each step and reduces the risk of error propagation.

5) *Expert Assessment of Generated Sequence Diagrams*: To further assess the quality of generated sequence diagrams, we conducted an expert assessment of two GPT 5.1 configurations: **CoT prompting** and **basic prompting**. Two evaluators with experience in UML sequence diagrams independently reviewed all successfully compiled diagrams from both configurations. The evaluators were blind to the generation strategy during the assessment.

Each diagram was rated along three dimensions corresponding to the automated evaluation metrics: (1) **Participant**

TABLE IV: Expert assessment scores of the generated sequence diagrams.

Dimension	GPT 5.1-basic	GPT 5.1-CoT
Participants	3.69 ± 0.06	4.96 ± 0.00
Messages	3.29 ± 0.29	4.85 ± 0.08
Structure	2.92 ± 0.08	4.50 ± 0.19

correctness; (2) **Message correctness;** (3) **Structural correctness**, assessing whether branching, looping, and nesting logic correctly reflected the intended behavior. Ratings were given on a 5-point Likert scale (1 = poor, 5 = near-perfect).

Table IV summarizes the expert assessment results. Across all three dimensions, GPT 5.1-CoT substantially outperforms GPT 5.1-basic. Participant correctness receives near-perfect scores under CoT prompting (4.96 ± 0.00), indicating that participants are consistently and accurately identified. Message correctness and structural correctness also show considerable improvements over basic prompting, suggesting that our approach helps the model better capture both interaction content and control-flow logic.

6) *Summary:* The experimental results demonstrate that schema-constrained generation substantially improves both syntactic validity and semantic alignment of generated sequence diagrams. Across both models, structured prompting strategies further improve generation quality.

There are also several implications of the results. First, the strong improvement brought by schema constraints suggests that structured intermediate representations are essential for reliable diagram generation with LLMs. This observation may generalize to the generation of other software models. Second, the results indicate that modern LLMs are already capable of supporting analysts in transforming textual requirements into reasonably accurate sequence diagrams. They can substantially reduce manual modeling effort by providing an initial draft for refinement. Finally, while current LLMs can capture interaction content relatively well, reasoning over nested control-flow semantics remains difficult. This suggests that future work should focus less on surface-level extraction and more on behavioral reasoning, constraint checking, and formal verification mechanisms integrated with LLM generation.

D. Case Study

We revisit the case study of UC-17 to illustrate the improvements achieved by our approach. Figure 4 presents the generated diagram under CoT configurations.

Content correctness: The generated two participants *Spirit Director* and *System* are both extracted from the use case description and are exactly the same as the baseline, while the basic prompting result contains two extra participants, which are not mentioned in the use case. The precision of messages is 0.83 compared to 0.33 for basic prompting. As can be observed in Figure 1, basic prompting produces many redundant messages, while the messages produced by CoT prompting are more concise and compliant with the use case.

Content completeness: All baseline participants are identified in both the CoT and basic prompting results. As for messages, CoT prompting achieves a recall of 1.0 on this use case, compared to 0.76 for basic prompting. CoT prompting helps capture all the key interactions.

Structure validity: The structural problems of Figure 1 have been discussed in Section III. The diagram generated by CoT prompting is more comprehensible and informative to readers. All loops (steps 4–7 and steps 2–18) are correctly captured. The conditional guards and nested combined fragments are basically all reasonable and compliant with the structure of the original use case. A logical error also exists though. As the use case states, the execution should terminate after an alert of rejecting pending requests. But the “break” fragment used in the diagram only breaks the inner loop, whereas the execution continues to “confirm creation” after the break, which causes a logic error. To make it correct, extra break fragments are needed to break the outer fragments to terminate the execution. But overall, this error does not hamper the comprehension of the use case too much, and the diagram makes it concise and human-readable while keeping the capture of the core logic and the majority of the contents from the use case.

VI. DISCUSSIONS

This section discusses threats to validity and limitations of our work.

First, the current results are based on 26 use cases from one project, and incorporating more projects would help to further validate the effectiveness of the proposed approach. However, The construction of high-quality annotated data remains challenging, because constructing reliable ground truth sequence diagrams requires substantial manual effort. Future work should expand the evaluation to broader datasets covering more domains and projects.

Second, automatically evaluating the quality of sequence diagrams is an extremely challenging task. UML sequence diagrams are primarily designed for human understanding and communication, and there is often no single canonical representation for the same underlying behavior. In this work, we provide an initial automated evaluation framework. In particular, the jump scenarios extracted by our method are invariant to several common alternative expressions of equivalent logic. However, it still cannot fully characterize complete behavioral equivalence between diagrams. More sophisticated behavioral analysis methods remain future work.

Third, the sequence diagrams generated by our approach may have consistency issues in terms of participant names and message parameters. We have qualitatively inspected this issue and observed that schema constraints generally improve consistency by enforcing normalized participant identifiers and structured message representations. However, different diagrams still may use semantically similar but lexically different expressions for messages or conditions, and message parameters remain informal text descriptions rather than rigorously typed specifications.

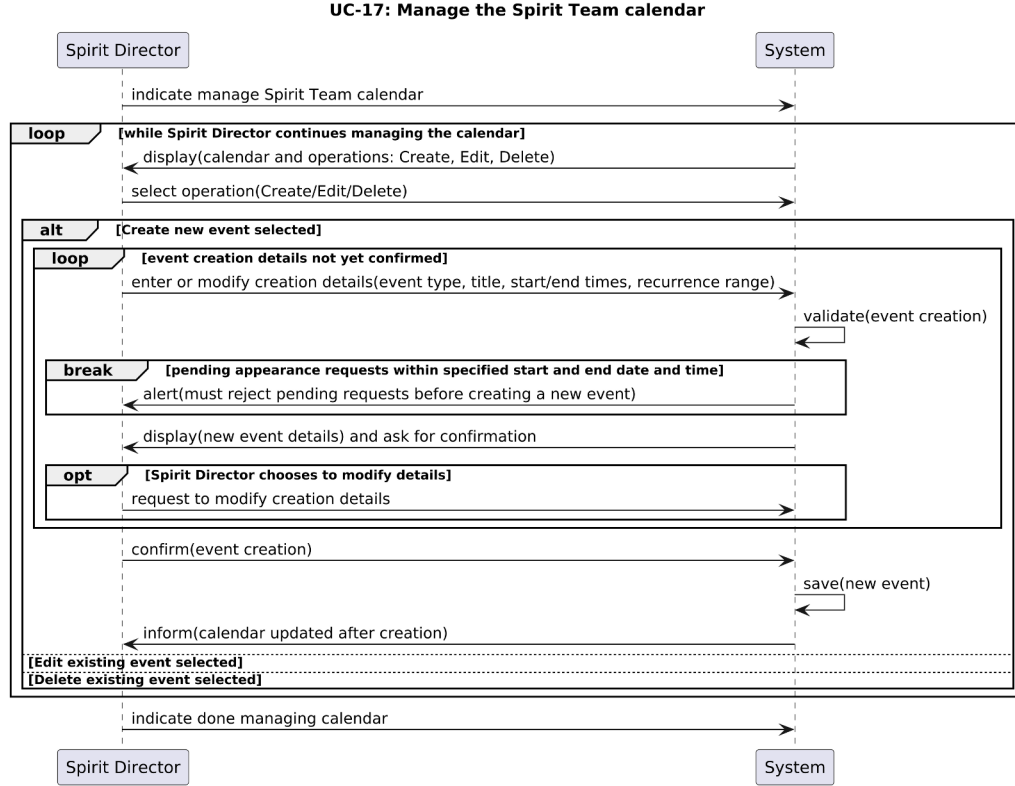


Fig. 4: The sequence diagram of UC-17 generated by GPT 5.1 with schema constraint and CoT prompting.

VII. CONCLUSIONS

This paper studied the problem of generating UML sequence diagrams from natural-language use case scenarios using large language models. While LLMs show promising capabilities, direct generation often produces diagrams with syntax errors, missing elements, and incorrect control-flow structures. We then propose structured generation approaches based on chain-of-thought prompting and workflow-based decomposition, guided by a formally defined JSON schema. A dataset of 26 industrial-level use cases was constructed and manually annotated to support systematic evaluation. We also introduced an evaluation framework that measures participants, messages, and jump scenarios to automatically assess both the content and structure correctness. Both empirical results and human expert assessments show that structured prompting with schema constraint significantly improves the quality of generated sequence diagrams compared with direct prompting.

DATA AND CODE AVAILABILITY

All of the data, code, experiment results are available at <https://github.com/BotiLee/UMLgpt>.

ACKNOWLEDGMENT

This work is supported by the Frontier Technologies R&D Program of Jiangsu (BF2024059), the National Natural Sci-

ence Foundation of China (No. 92582201, 62572229), and the “111 Center” (No. B26023).

REFERENCES

- [1] I. Jacobson and J. R. G. Booch, *The unified modeling language reference manual*. Reading, Mass.: Addison-Wesley Longman, 2021.
- [2] T. Yue, L. C. Briand, and Y. Labiche, “atoucan: An automated framework to derive uml analysis models from use case models,” *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 3, May 2015. [Online]. Available: <https://doi.org/10.1145/2699697>
- [3] R. Jolak, M. Savary-Leblanc, M. Dalibor, A. Wortmann, R. Hebig, J. Vincur, I. Polasek, X. Le Pallec, S. Gérard, and M. R. V. Chaudron, “Software engineering whispers: The effect of textual vs. graphical software design descriptions on software design communication,” *Empirical Softw. Engg.*, vol. 25, no. 6, p. 4427–4471, Nov. 2020. [Online]. Available: <https://doi.org/10.1007/s10664-020-09835-6>
- [4] M. Ozkaya, “Are the uml modelling tools powerful enough for practitioners? a literature review,” *IET Software*, vol. 13, no. 5, p. 338–354, Oct. 2019. [Online]. Available: <https://doi.org/10.1049/iet-sen.2018.5409>
- [5] *ISO/IEC 19505-1:2012, Information technology — Object Management Group Unified Modeling Language (OMG UML) — Part 1: Infrastructure*, International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) Std., 2012. [Online]. Available: <https://www.iso.org/standard/32624.html>
- [6] M. Seidl, M. Scholz, C. Huemer, and G. Kappel, *UML @ Classroom: An Introduction to Object-Oriented Modeling*, ser. Undergraduate Topics in Computer Science. Cham: Springer International Publishing, 2015. [Online]. Available: <https://link.springer.com/10.1007/978-3-319-12742-2>

- [7] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, K. Bontcheva and J. Zhu, Eds. Baltimore, Maryland: Association for Computational Linguistics, Jun. 2014, pp. 55–60. [Online]. Available: <https://aclanthology.org/P14-5010/>
- [8] L. Zhao, W. Alhoshan, A. Ferrari, K. J. Letsholo, M. A. Ajagbe, E.-V. Chioasca, and R. T. Batista-Navarro, "Natural language processing (nlp) for requirements engineering: A systematic mapping study," 2020. [Online]. Available: <https://arxiv.org/abs/2004.01099>
- [9] M. Jahan, Z. S. H. Abad, and B. Far, "Generating Sequence Diagram from Natural Language Requirements," in *2021 IEEE 29th International Requirements Engineering Conference Workshops (REW)*, Sep. 2021, pp. 39–48. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/9582297>
- [10] M. Jahan, M. M. Hassan, R. Golpayegani, G. Ranjbaran, C. Roy, B. Roy, and K. Schneider, "Automated Derivation of UML Sequence Diagrams from User Stories: Unleashing the Power of Generative AI vs. a Rule-Based Approach," in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 138–148. [Online]. Available: <https://dl.acm.org/doi/10.1145/3640310.3674081>
- [11] A. Ferrari, S. Abualhaija, and C. Arora, "Model Generation with LLMs: From Requirements to UML Sequence Diagrams," in *2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW)*, Jun. 2024, pp. 291–300. [Online]. Available: <https://ieeexplore.ieee.org/document/10628665>
- [12] C. Xiao, D. Ståhl, and J. Bosch, "UML Sequence Diagram Generation: A Multi-Model, Multi-Domain Evaluation," in *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Apr. 2025, pp. 272–283. [Online]. Available: <https://ieeexplore.ieee.org/document/11121690>
- [13] F. Dalpiaz, P. Gieske, and A. Sturm, "On deriving conceptual models from user requirements: An empirical study," *Information and Software Technology*, vol. 131, p. 106484, 03 2021.
- [14] P. G. Daryl Kulak, *Use Cases: Requirements in Context*. Addison-Wesley Longman Publishing Co., Inc., 75 Arlington Street, Suite 300 Boston, MA, United States, 2003.
- [15] O. Dawood and A.-E.-K. Sahraoui, "From requirements engineering to uml using natural language processing – survey study," *European Journal of Engineering Research and Science*, vol. 2, p. 44, 01 2017.
- [16] S. Gulia and T. Choudhury, "An efficient automated design to generate uml diagram from natural language specifications," in *2016 6th International Conference - Cloud System and Big Data Engineering (Confluence)*, 2016, pp. 641–648.
- [17] L. M. Segundo, R. R. Herrera, and K. Y. P. Herrera, "Uml sequence diagram generator system from use case description using natural language," in *Electronics, Robotics and Automotive Mechanics Conference (CERMA 2007)*, 2007, pp. 360–363.
- [18] L. Kof, "Scenarios: Identifying missing objects and actions by means of computational linguistics," in *15th IEEE International Requirements Engineering Conference (RE 2007)*, 2007, pp. 121–130.
- [19] A. Gupta, G. Poels, and P. Bera, "Creation of multiple conceptual models from user stories – a natural language processing approach," in *Advances in Conceptual Modeling*, G. Guizzardi, F. Gailly, and R. Suzana Pitangueira Maciel, Eds. Cham: Springer International Publishing, 2019, pp. 47–57.
- [20] T. Yue, L. C. Briand, and Y. Labiche, "An automated approach to transform use cases into activity diagrams," in *Modelling Foundations and Applications*, T. Kühne, B. Selic, M.-P. Gervais, and F. Terrier, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 337–353.
- [21] W. Ben Abdesslem Karaa, Z. Ben Azzouz, A. Singh, N. Dey, A. S. Ashour, and H. Ben Ghazala, "Automatic builder of class diagram (ABCD): an application of UML generation from functional requirements," *Software: Practice and Experience*, vol. 46, no. 11, pp. 1443–1458, Nov. 2016. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2384>
- [22] M. Ramzan, G. S. Sadiqi, M. S. Bashir, S. Raza, and A. Batool, "A Rule-Based Approach for Automatic Generation of Class Diagram from Functional Requirements Using Natural Language Processing and Machine Learning," *Journal of Computing & Biomedical Informatics*, vol. 7, no. 02, Sep. 2024. [Online]. Available: <https://jcibi.org/index.php/Main/article/view/546>
- [23] A. M. Alashqar, "Automatic generation of uml diagrams from scenario-based user requirements," *Jordanian Journal of Computers and Information Technology (JJCIT)*, vol. 07, no. 02, pp. 180 – 191, 2021.
- [24] L. Belzner, T. Gabor, and M. Wirsing, "Large language model assisted software engineering: Prospects, challenges, and a case study," in *Bridging the Gap Between AI and Reality*, B. Steffen, Ed. Cham: Springer Nature Switzerland, 2024, pp. 355–374.
- [25] C. Arora, J. Grundy, and M. Abdelrazek, "Advancing requirements engineering through generative ai: Assessing the role of llms," 2023. [Online]. Available: <https://arxiv.org/abs/2310.13976>
- [26] K. Chen, Y. Yang, B. Chen, J. A. Hernández López, G. Mussbacher, and D. Varró, "Automated domain modeling with large language models: A comparative study," in *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2023, pp. 162–172.
- [27] K. Ronanki, C. Berger, and J. Horkoff, "Investigating chatgpt's potential to assist in requirements elicitation processes," 2023. [Online]. Available: <https://arxiv.org/abs/2307.07381>
- [28] C. Jain, P. Anish, A. Singh, and S. Ghaisas, "A transformer-based approach for abstractive summarization of requirements from obligations in software engineering contracts," 09 2023, pp. 169–179.
- [29] R. Djaber and H. Ismail, "Ai as a co-engineer: A case study of chatgpt in software lifecycle," *Research Square*, 2023, <https://www.researchsquare.com/article/rs-3809973/latest.pdf>. [Online]. Available: <https://app.dimensions.ai/details/publication/pub.1167397943>
- [30] B. Chen, O. Wei, B. Zheng, and G. Mussbacher, "Accurate and consistent graph model generation from text with large language models," in *28th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2025, Grand Rapids, MI, USA, October 5-10, 2025*. IEEE, 2025, pp. 130–141. [Online]. Available: <https://doi.org/10.1109/MODELS67397.2025.00018>
- [31] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with chatgpt," 2023. [Online]. Available: <https://arxiv.org/abs/2302.11382>
- [32] B. Wei, "Requirements are All You Need: From Requirements to Code with LLMs," in *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, Jun. 2024, pp. 416–422. [Online]. Available: <https://ieeexplore.ieee.org/document/10628487>
- [33] PlantUML Team, "Plantuml: Open-source tool that uses simple textual descriptions to draw uml diagrams," <https://plantuml.com/>, n.d., accessed: 2026-02-25.
- [34] B. Pando and J. Castillo, "Plantumlgen: A tool for teaching model driven development," in *2022 17th Iberian Conference on Information Systems and Technologies (CISTI)*, 2022, pp. 1–6.