TITAN: <u>T</u>ask-oriented Prompt <u>I</u>mprovement with Script Gener<u>a</u>tio<u>n</u>

Chung-Yu Wang*
Alireza DaghighFarsoodeh
Hung Viet Pham
{cywang14,aliredaq,hvpham}@yorku.ca
York University, Canada

Abstract

Large language Models (LLMs) have demonstrated remarkable abilities across various tasks, leveraging advanced reasoning. Yet, they struggle with task-oriented prompts due to a lack of specific prior knowledge of the task answers. The current state-of-the-art approach, PAL, utilizes code generation to address this issue. However, PAL depends on manually crafted prompt templates and examples while still producing inaccurate results.

In this work, we present TITAN—a novel strategy designed to enhance LLMs' performance on task-oriented prompts. TITAN achieves this by generating scripts using a universal approach and zero-shot learning. Unlike existing methods, TITAN eliminates the need for detailed task-specific instructions and extensive manual efforts. TITAN enhances LLMs' performance on various tasks by utilizing their analytical and code-generation capabilities in a streamlined process. TITAN employs two key techniques: (1) step-back prompting to extract the task's input specifications and (2) chain-of-thought prompting to identify required procedural steps. This information is used to improve the LLMs' code-generation process. TITAN further refines the generated script through post-processing and the script is executed to retrieve the final answer.

Our comprehensive evaluation demonstrates TITAN's effectiveness in a diverse set of tasks. On average, TITAN outperforms the state-of-the-art zero-shot approach by 7.6% and 3.9% when paired with GPT-3.5 and GPT-4. Overall, without human annotation, TITAN achieves state-of-the-art performance in 8 out of 11 cases while only marginally losing to few-shot approaches (which needed human intervention) on three occasions by small margins. This work represents a significant advancement in addressing task-oriented prompts, offering a novel solution for effectively utilizing LLMs in everyday life tasks.

CCS Concepts

• **Computing methodologies** → **Machine learning**; **Semantic networks**; • **Applied computing**;

Keywords

Prompt Engineering, Code Generation, Large Language Models

1 Introduction

Large Language Models (LLMs) like GPT have significantly advanced the field of Natural Language Processing (NLP) through

---

*The work was done when Chung-Yu Wang was at York University

their proficiency in a wide range of tasks, including text generation [1, 30], translation [16, 42], summarization [19], and answering questions [21, 25]. These models are trained on vast datasets, enabling them to produce text that is both coherent and contextually appropriate [47]. However, when it comes to handling basic, task-oriented problems that involve numerical calculations or step executions, LLMs often fall short [24, 52]. This is an inherent weakness due to the way LLMs are designed and constructed. LLMs are trained on large text corpora and finely tuned for linguistic content creation and processing [3]. LLMs construct answers from text corpora and often face difficulties when the answers are not present directly in the training dataset but require precise numerical operations or step executions [15]. For instance, LLMs' limitation in counting tasks is evident in a simple test. If GPT-4 [2] using greedy decoding (0.0 temperature), is asked: "Ed had 22 more marbles than Doug. Doug lost 8 of his marbles at the playground. How many more marbles did Ed have than Doug then?", the answer would be: "Ed still had 22 more marbles than Doug". LLMs have a tendency to provide approximations or incorrect counts. This highlights the necessity for specialized prompt improvement to address these kinds of task-oriented challenges.

One approach is to utilize prompt engineering [27, 32, 50, 57, 60] to improve LLMs' performance on specific tasks. This process involves crafting well-defined, strategically structured prompts to guide models towards specific outcomes. For instance, Chain-of-Thought (CoT) [45], one of the prompting techniques, breaks down problems into intermediate, textual steps to facilitate problem understanding and reveal steps toward potential solutions. CoT could be applied to the above problems to help LLM derive steps to complete the task. However, CoT still struggles with task-oriented problems as it still inherits the LLMs' weakness of not having direct access to the numerical answers [15]. To address this, prior work proposes Program-Aided Language Models (PAL)[13]. PAL employs the generation of code as an intermediate step in the reasoning process to bridge the task execution gap.

Nevertheless, these prompt techniques perform optimally when used with few-shot prompting, in which hand-picked problem examples and answers are provided to help LLMs correctly comprehend and solve problems. However, crafting these examples for few-shot prompting is non-trivial and is an extra burden for the users without any related experience [10, 28, 35]. Using wrong examples [37] or wrongly ordering examples [31] for few-shot prompting will affect performance dramatically. To address the drawbacks of crafting few-shot prompts for task-oriented problems, we introduce TITAN, a novel prompting framework to address the challenges of task-oriented problems while requiring no user effort.

Distinct from PAL, which directly generates scripts, TITAN incorporates two additional intermediate reasoning stages: input extraction and step extraction. These additional reasoning stages facilitate the generation of accurate scripts without the need for labeled examples. Specifically, TITAN applies step-back prompting [57] to extract the inputs and their specifications for each task, making it the first framework to incorporate step-back prompting into code generation tasks. Input extraction helps LLMs overcome their tendency to perform poorly when given meaningless variables or when there is a misunderstanding of the problem's inputs. In parallel, TITAN extracts procedure steps to complete the task by utilizing CoT prompting, which has been shown to aid LLMs in accurately comprehending problems by breaking them down into steps [22, 45]. Finally, TITAN combines the information from these two additional reasoning stages to generate the most accurate scripts that are capable of producing precise answers.

To assess the effectiveness of TITAN, we evaluate it across seven distinct prior datasets [9, 13, 23, 33, 34, 39] containing mathematical and symbolic reasoning tasks and four additional task-oriented benchmarks we constructed in this work. The results demonstrate that TITAN, when paired with GPT-4, consistently outperforms PAL zero-shot variant by an average accuracy improvement of 3.9%. When compared to few-shot approaches, TITAN performs better or comparable in most cases (8 out of 11 datasets). Furthermore, TITAN remains effective even with less advanced LLMs, such as GPT-3.5. TITAN improves state-of-the-art zero-shot approaches on GPT-3.5 by an average of 7.6%. Overall, when using GPT-4, TITAN achieves state-of-the-art performance on 8 out of 11 evaluated datasets while only marginally losing to few-shot approaches (which needed human intervention) on three datasets by small margins. Our ablation study further indicates that the addition of input and step extraction stages enables TITAN to tackle task-oriented problems accurately without the need for hand-crafted examples and prompt templates.

This work's main contributions are as follows:

- A new TITAN framework that can improve LLMs' ability to solve task-oriented problems in a zero-shot manner by extracting inputs and procedure steps using step-back and chain-of-thought prompting respectively.
- Four task-oriented datasets consist of Finding, Counting, True/-False, and Generative problems that can be used to better assess the capacity of LLMs in addressing task-oriented challenges.
- An extensive evaluation of TITAN on eleven datasets employing two different popular versions of LLMs (GPT-3.5 Turbo and GPT-4).

2 Approach

In this work, we introduce TITAN, a novel approach to improve LLMs responses to task-oriented prompts. This is achieved by generating code through the use of a universal prompt template coupled with zero-shot learning techniques. TITAN significantly differs from prior prompt-specific strategies [6, 7, 13, 29, 53, 55, 59] since it provides a general and effective way to boost the performance of LLMs in performing various tasks without the need for individual prompt adjustments. TITAN enhances LLMs in two key ways: firstly, by utilizing the inherent abilities of LLMs to dissect and
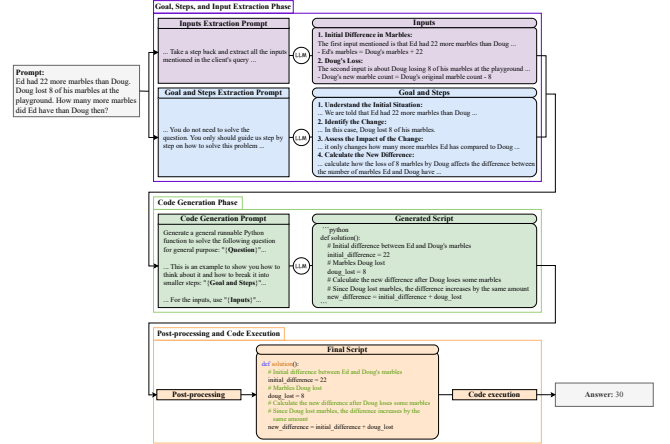

Figure 1: TITAN Overview

analyze complex queries; and secondly, by employing the code generation ability of LLMs to create executable scripts that produce the answers.

2.1 Challenges

As discussed in the Introduction, task-oriented prompts that involve numerical numbers (e.g., Ed had 22 more marbles than Doug. Doug lost 8 of his marbles at the playground. How many more marbles did Ed have than Doug then?) are not a direct fit for LLMs [15]. LLMs excel at analyzing and understanding complex queries by learning prior knowledge from a large corpus of text data [4, 44]. However, for LLMs to correctly respond to a query, the answer must be composed of such prior knowledge [14]. In the case of task-oriented prompts, the answer often is the result of an execution of some procedure described by the prompt. Hence in most cases, the result does not directly exist in the prior knowledge.

Conversely, LLMs have shown proficiency in generating code from natural language descriptions [11, 26, 38, 40, 51], as their training datasets contain numerous coding instances that the LLMs can utilize. By combining the analysis ability of LLMs with the task execution function of computer programs, we can create a system that is capable of precisely performing complex task-oriented prompts.

Prior approaches improve task-oriented prompts with code generation by relying heavily on task-specific prompts, requiring extensive manual effort and domain expertise. Such approaches [7, 13, 53, 55] employ uniquely crafted prompt templates tailored to individual tasks, along with few-shot learning techniques, to create scripts that can produce the desired response. Thus, for a specific task, the user would need to design a prompt template and provide some learning examples to get the most accurate responses from LLMs.

Figure 1 shows an example of a step-by-step execution of TITAN. Specifically, given the input prompt "Ed had 22 more marbles than Doug. Doug lost 8 of his marbles at the playground. How many more marbles did Ed have than Doug then?", which asks TITAN to perform a subtraction task of "Marbles Ed had" and "Marbles Doug had after he lost some of them at the playground". TITAN performs step-back analysis to extract the specification of the inputs (i.e., the

initial difference in marbles and the amount that Doug had after the accident). At the same time, TITAN performs chain-of-thought (CoT) analysis to extract the procedure (i.e., Understanding the situation and the changes after the accident) as well as the expected output (i.e., calculating the new difference). By combining these two pieces of information, TITAN was able to generate the solution script that correctly outputs the expected result.

2.2 Script generation with step-back and zero-shot chain-of-thought prompting

To reduce the reliance on human effort in utilizing LLMs for task-oriented prompts, TITAN leverages two recently proposed prompt engineering techniques: step-back prompting [57] and zero-shot chain-of-thought prompting [22]. Step-back prompting helps TITAN analyze the query to identify the relevant inputs and their requirements. Additionally, zero-shot chain-of-thought prompting analyzes the query to extract the relevant steps and procedures to perform the task while requiring no additional input from the user. By integrating these two processes, TITAN could produce a script that faithfully represents the original task, which in turn yields a precise response.

Extracting inputs and specification phase: TITAN employs step-back prompting to identify the input requirements from the initial prompt. This process involves querying the LLM to identify and outline the specific inputs needed for each task. By taking a step back, TITAN enables the model to focus on the essential inputs required for successful code execution in the later stage.

Step-back prompting consists of two stages: abstraction and reasoning. Instead of asking the question directly, the abstraction stage asks the LLM a step-back question about a related higher-level concept or principle [57]. In the reasoning stage, the LLM is asked to reason about the high-level concept or principle facts found after the step-back question. In this work, TITAN utilizes the abstraction stage to extract inputs and their specifications from the original prompt. It then performs the reasoning stage when generating code based on the extracted inputs.

As demonstrated in Figure 1, TITAN performs the abstraction stage by asking a step-back question "... Take a step back and extract all the inputs mentioned in the client's query ..." to extract the inputs from the original prompt. Specifically, by directing the LLM to focus on a higher concept (i.e., inputs and their specifications) without directly addressing the original prompt, TITAN can extract accurate inputs that help the LLM in the later code generation phase. In this example, TITAN extracts the two inputs (i.e., "22" and "8") and their specifications (i.e., "initial difference in marbles" and "Doug's loss"). These are then integrated into the generated code later as `initial_difference = 22` and `doug_lost = 8`.

Extracting the goal and procedure steps phase: To improve the precision of the code generation step, TITAN applies zero-shot chain-of-thought prompting to delve deeper into the goal of the task and the logical steps needed to achieve it while requiring no additional examples. TITAN prompts the LLM to articulate a step-by-step reasoning process, effectively mapping out the pathway from problem statement to solution. As shown in Figure 1, TITAN uses the prompt "... should guide us step by step on how to solve this problem ..." to extract the procedure steps needed to

address the original prompt. By encouraging the LLM to express explicitly the thought process, TITAN can extract a detailed goal and the methodological procedure required to complete the task. This process clarifies the objective to ensure the generated code aligns closely with the intended outcome. By utilizing the zero-shot variant of chain-of-thought, TITAN eliminates the need for additional examples that prior work such as PAL requires to perform optimally.

Figure 1 shows that by instructing the LLM to outline steps toward a goal without directly seeking the final answer, TITAN can clarify the main objective "Calculate the New Difference". Additionally, chain-of-thought prompting helps TITAN uncover logical thinking to solve the problem "It only changes how many more marbles Ed has compared to Doug". This phase guides the LLM to better understand the final goal and process during the code generation phase. This is demonstrated in the final generated code `new_difference = initial_difference + doug_lost` which matches the extracted steps.

Code generation phase: Combining the information extracted in the two previous phases, TITAN prompts the LLM to generate code based on clearly defined inputs, the articulated goal, and well-reasoned steps. This code generation phase combines the inputs of step-back prompting and the steps from chain-of-though prompting to synthesize a coherent and functional code output. Specifically, TITAN employs the prompt "Generate a general Python function to solve the following question for general purpose: {question}" to ask LLMs to generate a Python function for the question. Followed up by the prompts "This is an example to show you how to think about it and how to break it into smaller steps: "{The Output from Goal and Steps Extraction}"" and " For the inputs, use "{The Output from Inputs Extraction}"" to aggregate the outputs from previous phase into code generation phase. In this way, TITAN is able to guide the LLM to produce code that is logically aligned with the task's objectives. For example, TITAN obtained the generated script (in Figure 1) given the original prompt "Ed had 22 more marbles than Doug. Doug lost 8 of his marbles at the playground. How many more marbles did Ed have than Doug then?" This example demonstrates the effectiveness of step-back and chain-of-thought prompting in helping LLM generate code with the correct inputs `initial_difference = 22` and `doug_lost = 8`, and precise representation of the task `new_difference = initial_difference + doug_lost`.

Post-processing and code execution: Since LLMs return free-form responses, TITAN employ a rigorous extraction and validation process. Specifically, TITAN utilizes regular expressions to extract the generated code from the responses. The regular expressions target consistent formatting markers (i.e., "'Python'" for code generated by LLMs to ensure consistent extraction accuracy. Once the code is extracted, additional post-processing is required such as importing required packages and fixing indentation errors. The code is then executed automatically and the output is extracted as the final response to the user prompt. To get the result, we first set up rules to identify math questions from the code's output. If the output matches these rules, we take and return this part. If not, we just clean up the output and use that as the final answer. This makes sure we always have a neat and relevant result. For example, The

Table 1: Datasets overview

Dataset	N	Input	Output
GSM8K [9]	1319	Question	Number
GSMHard [13]	1319	Question	Number
SVAMP [34]	1000	Question	Number
ASDIV [33]	2096	Question	Number
AddSub [23]	395	Question	Number
MultiArith [23]	600	Question	Number
Penguins [39]	149	Table + Text + Question	Number + Text
Finding	660	Question	Text
Counting	1100	Question	Number
True/False	500	Question	Binary
Generative	1100	Question	Text + List

Final Script in Figure 1 is the final generated code that is executed and the output is extracted as 30 (i.e., the value of the variable `new_difference`).

3 Experimental Setup

3.1 Task-oriented datasets

Drawing from the prior study [15], when faced with a more challenging question on prime numbers that were sufficiently common to have representation on the Internet, the system performed adequately at that time. Hence, we create four task-oriented datasets from scratch to thoroughly evaluate TITAN performance. These include simple task-oriented prompts which we found to be particularly difficult for LLMs to directly address. This new dataset will benefit future research, given the current trend in prompt construction towards decomposition techniques, such as Least-to-Most Prompting [60] and Decomposed Prompting [20]. Our dataset includes decomposition tasks such as finding, counting, true/false questions, and generative tasks.

Table 1 shows the summary of these datasets. Each dataset includes multiple task templates that can be used to generate the prompts and the expected responses. Table 2 shows the complete template sets for the task-oriented datasets.

Finding Dataset: Given that LLMs have been shown to be strong in natural language tasks, it was a surprise to see LLMs (including GPT-4), perform poorly on simple Finding tasks where the prompt asks the models to identify specific patterns and letters within some given text. This Finding dataset includes 1100 queries that ask for a pattern or a word as a response, we demonstrate prompt templates for generating the Finding dataset in Table 2. These types of templates evaluate LLMs' capacity to discern patterns amidst varying textual contexts.

Counting Dataset: Counting tasks is not a natural fit for LLMs. As demonstrated in the Introduction, GPT models can make mistakes when performing counting tasks. In this dataset, we include 1100 queries that require numerical responses. We form the dataset with various prompt templates stated in Table 2. This dataset is designed to test LLMs' ability to handle counting-related challenges such as enumerating numbers, unique letters, and words.

True/False Dataset: True/False dataset includes tasks requiring LLMs to discern the veracity of statements related to natural language processing. The dataset includes queries that describe the natural language processing tasks on sentences or words. The models are required to respond with a binary digit. The dataset consists of 500 samples with auto-generated answers from five prompt templates listed in Table 2. These templates encompass five distinct

tasks, focusing on identifying aspects such as spacing, capitalization, repetition of words, and spelling differences within natural language sentences or words. Utilizing binary responses reduces the complexity of response composition and focuses the evaluation on the models' ability to analyze and perform natural language tasks.

Generative Dataset: Performing generative tasks is generally considered a strong point for LLMs which are generative in nature. LLMs are known to generate surprisingly well-structured responses for various creative prompts. However, procedural generative tasks that require strict procedure steps to complete are different from the typical free-form creative tasks that LLMs are known for. For example, in a procedural generative task, the LLM could be tasked with creating a word from the first letter of each word in a sentence. We construct a generative dataset that aims to assess the capability of models to produce novel responses by following a given procedure and inputs. This dataset comprises 1100 queries, formatted as prompts which ask the models to generate a new word as a response in various ways. There are five distinct categories of prompts listed in Table 2 that challenge various aspects of textual manipulation, including word swapping, capitalization adjustments, and modifications to the letters at the end of words. This dataset seeks to thoroughly explore the LLM's generative potential and its ability to understand and manipulate language constructs in novel ways.

3.2 Mathematical and symbolic reasoning datasets

Following prior work [13, 55], we evaluate TITAN on mathematical and symbolic reasoning datasets such as GSM8K, GSMHard [9], SVAMP [34], MAWPS [23], PENGUINS [13, 39], and ASDIv[33].

GSM8K and GSMHard are expansive collections of high-quality, linguistically varied grade school mathematics word problems, meticulously curated by adept problem composers. Each problem within the dataset necessitates a solution process that spans between two to eight steps, predominantly involving a series of fundamental arithmetic operations (i.e., addition, subtraction, multiplication, and division) to deduce the conclusive answer.

SVAMP (i.e., Simple Variations on Arithmetic Math Word Problems) dataset presents itself as a challenge set specifically designed for elementary-level Math Word Problems (MWP). An MWP is defined by a concise narrative in Natural Language, delineating a scenario or state of the world, which culminates in posing a query regarding one or more unknown quantities.

MAWPS is an online compendium dedicated to Math Word Problems (MWP) and serves as a comprehensive dataset for the evaluation of diverse algorithms.

PENGUINS delineates a task framework that integrates a tabular dataset of penguins, augmented with supplementary descriptors in natural language. The primary objective within this framework is to deduce answers to queries concerning the attributes of the penguins based on the provided dataset and descriptions.

ASDIv corpus (i.e., Academia Sinica Diverse MWP Dataset) is another MWP dataset which consists diverse language patterns and problem types, constituting an English MWP collection. This

Table 2: The templates to create the four task-oriented datasets

Dataset	Prompt Template	Response
Finding	Choose the word from the three options provided that does not have "**{word}**" within it. The single word given is: "['**{word}**', '**{word}**', '**{word}**']".	Word
	Taking into account that "**{letter}**" is identical to "**{letter}**", seek out the word among these three that has the most unique letter count. The words are "['**{word}**', '**{word}**', '**{word}**']".	
	Assuming "**{word}**" has precisely one "**{word}**", identify from the list below the word(s) that also contain exactly one "**{word}**". The list includes: "['**{word}**', '**{word}**', '**{word}**']".	
	Among the three words listed, select the one that initiates with "**{letter}**". The words for consideration are "**{words}**".	
Counting	Excluding words that have fewer than four letters, how many words, spaced apart by 'space', exist in this sentence? The input is: {**sentence**}.	Number
	How many numeric characters are found in "**{word}**"?	
	What is the count of "**{letter}**" in "**{word}**" when ignoring uppercase letters?	
	What is the total number of distinct letters in "**{word}**", disregarding case?	
	How many vowels can be found in the "**{word}**"	
True/False	If there is a space in "**{word}**", is there any space in "**{word}**"? If there is a space return "1", otherwise return "0".	0 / 1
	Is there a capitalization difference between "**{word}**" and "**{word}**"? If there is a difference return "1", otherwise return "0".	
	Does this sentence has more than 3 spaces? "**{sentence}**" If there are more than 3 spaces return "1", otherwise return "0".	
	Is there any repeated word in the following sentence? "**{sentence}**" If there are repeated words return "1", otherwise return "0".	
	If we assume the letter "**{letter}**" is equal to the letter "**{letter}**", is there any spelling difference between "**{word}**" and "**{word}**"? If there is a difference return "1", otherwise return "0".	
Generative	Take the first letter of each word within the specified sentence, join these letters to construct and return a new word. Words are spaced apart. The input is: "**{sentence}**"	Word
	Switch the initial two letters of the word provided and return the word thus generated. The input is: "**{word}**"	
	Replace the final letter of the given word with an 's' and return the newly formed word. The input is: "**{word}**"	
	Capitalize the first character of the given word and return the word with the adjustment. The input is: "**{word}**"	
	Replace the first letters of the words with each other and return the adjusted versions as the response. The words are: "['**{word}**', '**{word}**', '**{word}**', ...]"	List

corpus is designed for the assessment of the proficiency of various MWP solvers.

3.3 Baselines

PAL [13] is the state-of-the-art approach that focuses on task-oriented prompts. PAL employs a code interpreter for problem reasoning with few-shot prompting while TITAN utilizes zero-shot learning. To gain a deeper understanding of how our approach compares to the state-of-the-art we also include PAL's zero-shot prompting technique (PAL ZS) as our second baseline. PAL ZS utilizes the same prompt template as PAL but without the use of examples.

Recently, a few variants of PAL have been proposed such as Model Selection [55] and X-of-Thoughts [29]. Distinct from TITAN, which employs a two-step intermediate process for script creation, the Model Selection approach alternates between using CoT or PAL based on which yields more accurate outcomes as determined by language model evaluations. Meanwhile, X-of-Thought adopts the framework consisting of planning and verification phases, selecting the optimal method from among CoT, Program-of-Thought (PoT), and Equation-of-Thought (EoT) for problem-solving. Since these variants are also using code generation, we adopt their result as our baseline.

Our comparison does not include another variant, Code-based Self-Verification (CSV) [59], due to its dependence on the GPT-4 Code Interpreter and its evaluation solely on the MATH dataset. The dataset does not align with the types of problems TITAN aims to address.

Follow prior work, we also adopt the most recent GPT-4 model (i.e., gpt-4-0125-preview) and a less powerful GPT-3.5 model (i.e., gpt-3.5-turbo-0125) as our backend language model.

3.4 Experiment Details

We replicate PAL's result by executing PAL code from their GitHub using the same version of GPT-3.5 and GPT-4 as stated in the original paper. To run PAL on our task-oriented datasets, we form the PAL few-shot prompt with four examples by randomly selecting one example from each task-oriented dataset, otherwise, we keep all PAL's templates the same.

The zero-shot version of PAL uses the same templates but without any examples. Since PAL didn't provide a zero-shot prompt, we developed the zero-shot version of PAL inspired by another related study, PoT [7]. We adopt their prompt format by first posing

Table 3: Comparison of TITAN accuracy (%) to PAL Zero-shot, evaluated across eleven benchmarks using two GPT models. The best accuracy scores for each dataset and model are bold.

LLM	GPT-3.5			GPT-4		
Approach	PAL ZS	**TITAN**		PAL ZS	**TITAN**	
Metric	*Acc*	*Acc*	Δ	*Acc*	*Acc*	Δ
GSM8K	76.6	**84.2**	↑7.6	93.6	**95.3**	↑1.7
GSMHard	61.8	**69.6**	↑7.8	74.1	**78.2**	↑4.1
ASDIV	85.3	**91.4**	↑6.1	92.7	**97.2**	↑4.5
SVAMP	82.8	**84.3**	↑1.5	94.0	**94.8**	↑0.8
AddSub	**93.1**	89.8	↓3.3	95.7	**97.7**	↑2.0
Multiarith	**97.3**	96.8	↓0.5	96.8	**98.7**	↑1.9
Penguin	59.1	**94.3**	↑35.2	87.2	**97.5**	↑10.3
Finding	93.8	**98.4**	↑4.6	95.5	**99.8**	↑4.3
Counting	**89.1**	87.8	↓1.3	87.5	**89.8**	↑2.3
True/False	59.2	**76.7**	↑17.5	84.4	**93.8**	↑9.4
Generative	84.9	**94.1**	↑9.2	98.7	**99.9**	↑1.2
Average	80.3	**87.9**	↑7.6	90.9	**94.8**	↑3.9

the problem, followed by a request for the LLMs to complete a Python function named "solution()" without adding any examples. The function name and the return type are the same as the PAL few-shot version. For a fair comparison, we utilize the same metric used by PAL, which adopts exact match scores for evaluation. Unless specified differently, all experiments by default employ greedy decoding, adjusting the temperature of the language models to 0.0.

4 Result and Discussion

In this section, we compare TITAN against state-of-the-art code generation with zero-shot (RQ1) or few-shot (RQ2). In RQ3, we evaluate if self-consistency could help improve TITAN at a significant cost. Finally, we perform an ablation study (RQ4) to evaluate the contribution of each component in TITAN (i.e., input extraction and step extraction).

4.1 RQ1: How does TITAN compare to the state-of-the-art zero-shot prompting approaches with code generation?

Similar to prior work, TITAN employs script generation to solve mathematical and task-oriented problems. Hence, we first assess the effectiveness of TITAN when comparing state-of-the-art approaches with code generation in the zero-shot scenario. Existing code generation methods such as PAL, Model Selection (MS), and X-of-Thought (XoT), all utilize few-shot prompting. To evaluate how TITAN is compared to the prior work in the zero-shot scenario, we create a baseline PAL ZS (PAL zero-shot version) by incorporating a zero-shot prompts template [7] into PAL to be our zero-shot baseline.

Table 3 compares TITAN's accuracy to other state-of-the-art zero-shot approaches (Row Approach) with code generation across 11 datasets (Columns *Acc*). The table is divided into two sections each corresponding to a version of GPT (e.g., GPT-3.5 and GPT-4). It includes the Δ column to show the relative performance differences between TITAN and PAL ZS. For instance, TITAN paired with GPT-4 outperforms the state-of-the-art zero-shot approach PAL ZS in all 11 datasets with a margin as big as 10.3% on the Penguin dataset. When a weaker LLM model such as GPT-3.5 is used, TITAN still

Table 4: Comparison of TITAN accuracy (%) to other Few-shot code generation approaches, evaluated across eleven benchmarks using two GPT models. The * symbol indicates the accuracy is from original papers. The best accuracy scores for each dataset and model are bold.

LLM	GPT-3.5								GPT-4					
Approach Zero-shot	PAL ✗		MS* ✗		XoT* ✗		**TITAN** ✓		PAL ✗		MS* ✗		**TITAN** ✓	
Metric	*Acc*	*TΔ*	*Acc*	*TΔ*	*Acc*	*TΔ*	*Acc*	*AΔ*	*Acc*	*TΔ*	*Acc*	*TΔ*	*Acc*	*AΔ*
GSM8K	81.0	↓3.2	82.6	↓1.6	83.3	↓0.9	**84.2**	↑0.9	94.8	↓0.5	**95.6**	↑0.3	95.3	↓0.3
GSMHard	64.1	↓5.5	-		63.4	↓6.2	**69.6**	↑5.5	70.9	↓7.3	-		**78.2**	↑4.1
ASDIV	86.4	↓5.0	89.4	↓2.0	-		**91.4**	↑2.0	92.1	↓5.1	93.5	↓3.7	**97.2**	↑3.7
SVAMP	**84.6**	↑0.3	84.3	-0.0	83.6	↓0.7	84.3	↓0.3	94.6	↓0.2	93.7	↓1.1	**94.8**	↑0.2
AddSub	**92.7**	↑2.9	90.6	↑0.8	90.5	↑0.7	89.8	↓3.3	97.2	↓0.5	95.7	↓2.0	**97.7**	↑0.5
Multiarith	**98.7**	↑1.9	98.7	↑1.9	97.3	↑0.5	96.8	↓1.9	98.8	↑0.1	**99.0**	↑0.3	98.7	↓0.3
Penguin	**96.6**	↑2.3	-		-		94.3	↓2.3	96.6	↓0.9	-		**97.5**	↑0.9
Finding	97.7	↓0.7	-		-		**98.4**	↑0.7	**99.8**	-0.0	-		**99.8**	-0.0
Counting	84.6	↓3.2	-		-		**87.8**	↑3.2	88.5	↓1.3	-		**89.8**	↑1.3
True/False	76.0	↓0.7	-		-		**76.7**	↑0.7	**95.6**	↑1.8	-		93.8	↓1.8
Generative	87.7	↓6.4	-		-		**94.1**	↑6.4	99.4	↓0.5	-		**99.9**	↑0.5
Average	86.4	↓1.5	-		-		**87.9**	↑1.5	93.5	↓1.3	-		**94.8**	↑1.3

outshines PAL ZS on 8 over 11 datasets with margins as big as 35.2% in the case of the Penguin dataset. The reason weaker LLM models fall short on some datasets is that the less advanced GPT model is unable to leverage the additional stages that TITAN offers. In scenarios involving simple datasets, inundating simpler models with excessive information can lead to incorrect responses. [8]

The average row specifies the average performance of each approach across 11 datasets. On average, TITAN achieved state-of-the-art zero-shot performance with significant margins of 7.6% and 3.9% when used on GPT-3.5 and GPT-4 respectively. Overall, the weaker LLM models such as GPT-3.5 benefit more from the additional stages that TITAN utilizes. In contrast, the stronger models such as GPT-4 have better reasoning capability build-in and hence benefit less from step-back and chain-of-thought prompting. This result highlights TITAN's ability to significantly boost the performance of weaker language models.

When looking at the GPT-4 result, TITAN improvement on PAL ZS is relatively small (around 2%) on simpler datasets with small numbers and easy questions such as AddSub, Multiarith, and Counting. On the other hand, on more challenging datasets involving large numbers, tables, and complicated questions such as GSMHard, True/False, and Penguin datasets, TITAN significantly outperforms PAL-ZS (by 4.1%, 9.4%, and 10.3% respectively) This suggests that the TITAN is particularly suitable for complex task-oriented prompts.

> **Finding 1:** TITAN outperforms the state-of-the-art zero-shot approach by 7.6% and 3.9% when paired with GPT-3.5 and GPT-4. In some cases, the performance gain can be as big as 35.2% and 10.3% with GPT-3.5 and GPT-4 respectively.

4.2 RQ2: How does TITAN compare to the state-of-the-art few-shot prompting approaches with code generation?

In this RQ, we assess the broader effectiveness of TITAN when comparing state-of-the-art approaches with code generation that also utilizes few-shot prompting such as PAL few-shot version, Model Selection (MS), and X-of-Thought (XoT). The details of each

baseline are discussed in Sec 3.3. We replicate PAL on the original and our task-oriented datasets, we employ the code provided in the original study.

Table 4 compares TITAN's accuracy to other state-of-the-art Few-shot approaches with code generation. The asterisk "*" symbol indicates approaches that are not possible to replicate (i.e., due to missing executable source code) and for which the accuracy values are taken from the original papers. This is why the Table does not show results for MS and XoT on our proposed task-oriented datasets which is indicated by the dash ("-").

In Table 4, for each combination of LLM and dataset, the best accuracy value is highlighted in bold, indicating state-of-the-art performance. The $A\Delta$ columns illustrate the improvement that TITAN makes when compared to all baselines. For example, TITAN achieves state-of-the-art performance on 8 over 11 datasets when paired with GPT-4 LLM while only marginally losing to few-shot approaches (which needed additional non-trivial examples) on three datasets GSM8K, Multiarith, and True/False (by 0.3%, 0.3% and 1.8% respectively as indicated by the $A\Delta$ column).

Overall, when paired with the most advanced LLM (i.e., GPT-4), TITAN provides the most enhancement among all approaches across most datasets. This can be seen by looking at the $T\Delta$ columns which indicate the difference between each baseline and TITAN. Only in four cases where PAL and MS (which requires non-trivial examples) outperform TITAN with marginal gaps (i.e., 0.1%, 0.3%, 0.3%, and 1.8%).

On average, TITAN with GPT-4 achieves an average performance boost of 1.3% compared to PAL (which unlike TITAN requires few-shot examples). When evaluated on GPT-3.5, TITAN performs better on average than alternative few-shot approaches, achieving an average accuracy enhancement of 1.5% over PAL. These results indicate that TITAN achieves state-of-the-art performance in most cases even against few-shot approaches that require non-trivial examples.

When moving from GPT-3.5 to GPT-4, the average performance gain TITAN provides over PAL is only 0.2%(between 1.5% and 1.3%) which is significantly smaller than the 3.7% (between 7.6% and 3.9%) gain TITAN provides over PAL ZS. This might be because the more advanced language models (i.g., GPT-4) benefit more from few-shot prompting and hence are harder to enhance with step-back and chain-of-thought prompting that TITAN employs. Nonetheless, TITAN enhances the performance of these models across the board.

> **Finding 2:** Overall, TITAN achieves state-of-the-art performance in most (8 out of 11) cases when paired with GPT-4 while only marginally losing to few-shot approaches (which needed additional non-trivial examples) on three datasets by small margins. Overall, TITAN outperforms PAL by 1.3%.

To better demonstrate how TITAN was able to effectively generate correct responses and illustrate TITAN's limitations we include two examples where: **1:** TITAN is correct while PAL fails and **2:** both PAL and TITAN fail.

Case 1: Figure 2 shows an example from the ASDIV dataset where TITAN can combine correct reasoning from the input and step extraction phase to generate the correct code. Specifically, TITAN is

Figure 2: An example prompt where TITAN successes while PAL fails.

Figure 3: An example prompt where both PAL and TITAN fail.

able to identify the inputs `initial_difference` and `doug_lost` as well as the importance of reasoning: `#Since Doug lost marbles, the difference increases by the same amount` which is realized by the important extracted step: `new_difference = initial_difference + doug_lost`. This is further illustrated by Figure 1 where the extraction of goals and steps yields the correct logical steps: "Calculate the New Difference" On the other hand, without either of these reasoning phases, PAL fails to generate the correct responses. Specifically, PAL identifies the input name wrong `more_marbles_ed` which hinders the integration reasoning into the solution. This example showcases the distinct differences between PAL and TITAN and highlights TITAN's profound grasp of the problem context due to the reasoning phases.

Case 2: Figure 3 shows an example where both PAL and TITAN fail to generate the correct answer. Both methods correctly reason about the composition of the answer by counting the email invitations, the email invitations' friends, the phone invitations, and their spouses. However, both fail to account for one additional seat for Alex herself. This example might expose TITAN's inherent weakness from the underlying GPT model (i.e., GPT-4) where the GPT model does not link the number of seats needed to the number of attendants and instead links to the number of invitees mentioned in the prompt. We suspect that this can be due to GPT's autoregressive mechanism (i.e., tokens are generated sequentially) that makes each token's production reliant heavily on the outcomes generated before it [48].

Table 5: TITAN self-consistency integration on GSM8K, Multiarith, and True/False datasets utilizing GPT-4 with majority voting from three samples.

	TITAN	TITAN + SC@3
GSM8K	95.3	95.6 ↑0.3
Multiarith	98.7	99.5 ↑0.8
True/False	93.8	95.4 ↑1.6

This highlights unique challenges and flaws in the way GPT-4 learns and encodes these processes [15].

4.3 RQ3: Can self-consistency help improve TITAN?

Self-consistency [43] is an agnostic strategy that can be applied to CoT prompting approaches to improve the underlying approach performance. One downsize of self-consistency is that it requires many duplicated queries to be sent to the LLM which can have diminishing returns [18]. Since TITAN incorporates the CoT we hypothesize that incorporating self-consistency could further improve TITAN's performance. However, due to the elevated cost, we do not incorporate self-consistency in TITAN by default and instead use the temperature of 0.

In this RQ3, we want to test if self-consistency can help improve TITAN further and if the benefit warrants the cost. Specifically, we apply self-consistency on TITAN for three datasets (GSM8K, Multiarith, and True/False) to see if self-consistency can help TITAN achieve state-of-the-art performance. As indicated by the original paper [43], we set the LLM's temperature to a nonzero value to retrieve distinct response samples. In this case, we set the GPT-4 temperature to 0.7 before running TITAN three times. The final response is subsequently chosen by majority voting.

In Table 5, TITAN + SC3 (self-consistency with three samples) indicates the accuracy (%) of TITAN when self-consistency is integrated with three responses. When comparing the vanilla TITAN, self-consistency helps enhance TITAN by 0.3% on GSM8K, 0.8% on Multiarith, and 1.6% on True/False at the cost of triple the number of queries. If the cost of self-consistency is not a factor, TITAN + SC3 would achieve state-of-the-art performance on GSM8K and Multiarith while getting within 0.2% of the best approach on True/False. Overall, the result suggests the adoption of self-consistency not only bolsters TITAN's robustness but also achieves superior performance relative to greedy decoding (i.e., setting the LLM temperature to 0.0 which is TITAN's default setting) at a significant cost.

> **Finding 3:** Overall, Self-consistency can improve TITAN performance with significant cost. Specifically, with triple the cost, TITAN + SC@3 is shown to improve TITAN by achieving state-of-the-art performance on GSM8K and Multiarith.

4.4 RQ4: How each component contribute to TITAN's overall performance?

Input and step extraction are two important phases that help TITAN generate precise scripts that can be used to generate correct

Table 6: TITAN Ablation Study

Dataset	W/o Input Extraction	W/o Step Extraction	TITAN
GSM8K	95.6 ↑0.3	93.4 ↓1.9	95.3
GSMHard	77.5 ↓0.7	76.1 ↓2.1	78.2
ASDIV	95.8 ↓1.4	95.3 ↓1.9	97.2
SVAMP	94.5 ↓0.3	92.9 ↓1.9	94.8
AddSub	97.7 -0.0	97.7 -0.0	97.7
Multiarith	98.5 ↓0.2	98.3 ↓0.5	98.7
Penguins	95.3 ↓2.2	94.6 ↓2.9	97.5
Finding	98.5 ↓1.5	98.3 ↓1.3	99.8
Counting	93.6 ↑3.8	88.6 ↓1.2	89.8
True/False	94.5 ↑0.7	95.1 ↑1.3	93.8
Generative	99.0 ↓0.9	99.1 ↓0.8	99.9
Average	94.5 ↓0.3	93.5 ↓1.4	94.8

Figure 4: An example where both reasoning phases together help TITAN generate the correct response.

responses. In this RQ, we perform an ablation study with GPT-4 to see how each component contributes to TITAN's overall performance.

Table 6 shows how turning off one of the main reasoning components (Column W/o Input Extraction and W/o Step Extraction) can reduce the effectiveness of TITAN. The ↓ and ↑ indicate that removing the component reduces or increases the effectiveness of TITAN respectively. For example, removing the step extraction phase reduces TITAN accuracy on the Penguins dataset by 2.9%. On all datasets, removing the step extraction phase reduces TITAN's performance significantly with an average reduction in performance of 1.5%. The result suggests that step extraction significantly contributes to TITAN's performance.

On the other hand, removing the input extraction phase hinders TITAN's performance in a few cases. For example, on the Counting dataset, TITAN gains 3.8% without the input extraction phase. This could be because, for the counting tasks, there are fewer inputs and including additional information about the input complicates the script generation prompt and affects the final script. This result suggests that input extraction might have less contribution to the overall success of TITAN in correctly crafting the answers. However,

in a broader term, input extraction is still a necessary component of TITAN as removing it reduces TITAN performance by an average of 0.3%.

On AddSub, removing either component does not trigger significant accuracy changes (i.e., changes larger than 0.1%). This suggests that each reasoning phase already provides enough information to enhance the code generation. This is evident in Table 4, where TITAN outperforms PAL (without such reasonings) by 0.5% on the AddSub dataset. Overall, both components are essential to the overall performance of TITAN as removing either reduces the overall performance of TITAN.

This is further illustrated by the example in Figure 4, where without either input and step extraction, TITAN failed to utilize the correct input or follow the precise procedure. Specifically, without input extraction, the framework fails to distinguish between the `original_salary` and the `new_monthly_salary`, leading to incorrect bonus calculations. On the other hand, without step extraction, the model fails to recognize the correct steps to compute the `raise_amount` (i.e., should be across 12 months) and bonus (should be on raised salary). When utilizing both reasoning phases, TITAN was able to correctly extract precise inputs and procedural steps to complete the task in a clear and precise manner.

> **Finding 4:** Step extraction phase plays a significant role in enhancing TITAN's performance while in some cases input extraction can hinder the performance of the framework. However, both components are essential to the overall performance of TITAN as removing either reduces the overall performance of TITAN.

5 Related Work

Prompt engineering with code generation: There has been prior work that utilizes code generation to address the gap in LLMs' execution ability when it comes to task-oriented prompts. PAL[13] introduces an innovative method for tackling mathematical problems through script generation combined with few-shot prompting. Further improvement is made by a Model Selection technique (MS) [55] which employs both PAL and CoT (Chain-of-Thought) in tandem by selecting the best response between them.

X-of-Thoughts (XoT) [29] represents another code generation strategy, focusing on resolving mathematical and algebraic equations by dynamically switching among different prompting methods. Another distinct method, CSV [59], approaches the resolution of mathematical challenges through coding, which heavily relies on the GPT-4 Code Interpreter. Its efficacy is evaluated exclusively on the MATH dataset [17], indicating a specialized focus on coding solutions for mathematical issues.

In contrast, TITAN diverges significantly from these methods by adopting a zero-shot learning technique, which enhances its capacity for generalization across diverse problem sets. TITAN aims to solve reasoning questions without relying on any hand-crafted data, few-shot learning techniques, or human annotators.

Traditional prompt engineering without code generation: The exploration of concepts such as Chain-of-Thought [45] and Step-Back [57] has revealed the capacity of LLMs for zero-shot learning [22] primarily through their ability to process information in a step-by-step manner. Prior work such as PHP [56], Self-Contrast [53], and Boosting-of-Thought (BoT) [6] has been developed to uncover the reasoning paths through post-hoc strategies of prompt engineering, aiming to enhance the models' problem-solving capabilities by mimicking human-like [16] reasoning processes. These techniques use self-verification methods. For example, BoT iteratively generates, assesses, and refines thoughts using the model's self-evaluation. Self-Contrast exploits diverse solutions to enrich reasoning, while PHP refines reasoning paths with LLM outputs toward the correct answer.

Because the mechanisms and effectiveness of self-correction in LLMs are not well-understood [18], we do not include such iterative mechanism in TITAN. Recent research indicates that LLMs are not yet capable of self-correction [18]. Distinct from such interactive methods which require human annotations or many more query rounds, TITAN does not require manual effort to incorporate evaluation information into the reasoning process.

Code generation with LLMs: Recently, research in using LLMs for code generation [5, 12, 36, 49] has made significant advancements. These advancements often come from training these models with more specific examples of code, which makes them better at specific coding tasks [5]. Another line of research is guided code generation [54, 58] where LLMs are utilized to efficiently create code. Specifically, Willard et al. [46] introduce a system that guides LLMs to produce text using rules and structures from programming languages, reducing unnecessary steps in creating code sequences. Another recent work, SynCode [41], is a framework that improves how LLMs understand and generate code by focusing on the rules of programming languages. This approach helps create more accurate code by filtering out mistakes and focusing on the correct coding syntax. TITAN differs from these code generation research in the general nature of TITAN's input (i.e., prompt) where TITAN is designed to improve the overall question/answer capability of LLMs by utilizing script generation and not to solve general software engineering problems related to code generation.

6 Conclusion

In this work, we introduced TITAN, a novel approach for natural language reasoning that leverages script generation through the extraction of inputs and steps by utilizing Step-Back and Chain-of-Thought prompting respectively. We evaluate TITAN on 11 datasets with a comprehensive comparison with prior state-of-the-art. Unlike preceding approaches that predominantly rely on few-shot prompting techniques, TITAN employs a zero-shot prompting strategy, thereby eliminating the requirement for hand-crafted data. Our findings demonstrate that TITAN exhibits superior performance in a diverse set of tasks. Furthermore, the integration of TITAN with self-consistency further enhances its efficacy (with some additional cost), thereby underscoring the potential of TITAN as a robust solution for advanced natural language reasoning challenges.

7 Data Availability

We release our code and data through the following link: https://github.com/Smart-System-testing-lab/TITAN-Task-oriented-Prompt-Improvement-with-Script-Generation.

References

[1] Yelaman Abdullin, Diego Molla-Aliod, Bahadorreza Ofoghi, John Yearwood, and Qingyang Li. 2024. Synthetic Dialogue Dataset Generation using LLM Agents. arXiv preprint arXiv:2401.17461 (2024).

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. arXiv preprint arXiv:2303.08774 (2023).

[3] Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. 2024. Large language models for mathematical reasoning: Progresses and challenges. arXiv preprint arXiv:2402.00157 (2024).

[4] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2023. A survey on evaluation of large language models. ACM Transactions on Intelligent Systems and Technology (2023).

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021).

[6] Sijia Chen, Baochun Li, and Di Niu. 2024. Boosting of Thoughts: Trial-and-Error Problem Solving with Large Language Models. arXiv:2402.11140 [cs.CL]

[7] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks. arXiv:2211.12588 [cs.CL]

[8] Cheng-Han Chiang and Hung-yi Lee. 2024. Over-reasoning and redundant calculation of large language models. arXiv preprint arXiv:2401.11467 (2024).

[9] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. arXiv:2110.14168 [cs.LG]

[10] Hai Dang, Lukas Mecke, Florian Lehmann, Sven Goller, and Daniel Buschek. 2022. How to Prompt? Opportunities and Challenges of Zero- and Few-Shot Learning for Human-AI Interaction in Creative Applications of Generative Models. arXiv:2209.01390 [cs.HC]

[11] Enrique Dehaerne, Bappaditya Dey, Sandip Halder, Stefan De Gendt, and Wannes Meert. 2022. Code generation using machine learning: A systematic review. Ieee Access 10 (2022), 82434–82455.

[12] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. arXiv preprint arXiv:2204.05999 (2022).

[13] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-aided Language Models. arXiv:2211.10435 [cs.CL]

[14] Yingqiang Ge, Wenyue Hua, Kai Mei, Juntao Tan, Shuyuan Xu, Zelong Li, Yongfeng Zhang, et al. 2024. Openagi: When llm meets domain experts. Advances in Neural Information Processing Systems 36 (2024).

[15] Ben Goertzel. 2023. Generative AI vs. AGI: The Cognitive Strengths and Weaknesses of Modern LLMs. arXiv preprint arXiv:2309.10371 (2023).

[16] Zhiwei He, Tian Liang, Wenxiang Jiao, Zhuosheng Zhang, Yujiu Yang, Rui Wang, Zhaopeng Tu, Shuming Shi, and Xing Wang. 2024. Exploring human-like translation strategy with large language models. Transactions of the Association for Computational Linguistics 12 (2024), 229–246.

[17] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring Mathematical Problem Solving With the MATH Dataset. arXiv:2103.03874 [cs.LG]

[18] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2023. Large language models cannot self-correct reasoning yet. arXiv preprint arXiv:2310.01798 (2023).

[19] Hanlei Jin, Yang Zhang, Dan Meng, Jun Wang, and Jinghua Tan. 2024. A Comprehensive Survey on Process-Oriented Automatic Text Summarization with Exploration of LLM-Based Methods. arXiv preprint arXiv:2403.02901 (2024).

[20] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2022. Decomposed prompting: A modular approach for solving complex tasks. arXiv preprint arXiv:2210.02406 (2022).

[21] Jaehyung Kim, Jaehyun Nam, Sangwoo Mo, Jongjin Park, Sang-Woo Lee, Minjoon Seo, Jung-Woo Ha, and Jinwoo Shin. 2023. SuRe: Improving Open-domain Question Answering of LLMs via Summarized Retrieval. In The Twelfth International Conference on Learning Representations.

[22] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. Advances in neural information processing systems 35 (2022), 22199–22213.

[23] Rik Koncel-Kedziorski, Subhro Roy, Aida Amini, Nate Kushman, and Hannaneh Hajishirzi. 2016. MAWPS: A math word problem repository. In Proceedings of the 2016 conference of the north american chapter of the association for computational linguistics: human language technologies. 1152–1157.

[24] Yuan Li, Yixuan Zhang, and Lichao Sun. 2023. Metaagents: Simulating interactions of human behaviors for llm-based task-oriented coordination via collaborative generative agents. arXiv preprint arXiv:2310.06500 (2023).

[25] Zhenyu Li, Sunqi Fan, Yu Gu, Xiuxing Li, Zhichao Duan, Bowen Dong, Ning Liu, and Jianyong Wang. 2024. FlexKBQA: A Flexible LLM-Powered Framework for Few-Shot Knowledge Base Question Answering. arXiv:2308.12060 [cs.CL]

[26] Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. 2023. Improving chatgpt prompt for code generation. arXiv preprint arXiv:2305.08360 (2023).

[27] Jiacheng Liu, Alisa Liu, Ximing Lu, Sean Welleck, Peter West, Ronan Le Bras, Yejin Choi, and Hannaneh Hajishirzi. 2022. Generated Knowledge Prompting for Commonsense Reasoning. arXiv:2110.08387 [cs.CL]

[28] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2021. What Makes Good In-Context Examples for GPT-3? arXiv preprint arXiv:2101.06804 (2021).

[29] Tengxiao Liu, Qipeng Guo, Yuqing Yang, Xiangkun Hu, Yue Zhang, Xipeng Qiu, and Zheng Zhang. 2023. Plan, Verify and Switch: Integrated Reasoning with Diverse X-of-Thoughts. arXiv:2310.14628 [cs.CL]

[30] Albert Lu, Hongxin Zhang, Yanzhe Zhang, Xuezhi Wang, and Diyi Yang. 2023. Bounding the capabilities of large language models in open text generation with prompt constraints. arXiv preprint arXiv:2302.09185 (2023).

[31] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2021. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. arXiv preprint arXiv:2104.08786 (2021).

[32] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. Advances in Neural Information Processing Systems 36 (2024).

[33] Shen-Yun Miao, Chao-Chun Liang, and Keh-Yih Su. 2021. A diverse corpus for evaluating and developing English math word problem solvers. arXiv preprint arXiv:2106.15772 (2021).

[34] Arkil Patel, Satwik Bhattamishra, and Navin Goyal. 2021. Are NLP Models really able to Solve Simple Math Word Problems? arXiv:2103.07191 [cs.CL]

[35] Ethan Perez, Douwe Kiela, and Kyunghyun Cho. 2021. True few-shot learning with language models. Advances in neural information processing systems 34 (2021), 11054–11070.

[36] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950 (2023).

[37] Ohad Rubin, Jonathan Herzig, and Jonathan Berant. 2021. Learning to retrieve prompts for in-context learning. arXiv preprint arXiv:2112.08633 (2021).

[38] Jiho Shin and Jaechang Nam. 2021. A survey of automatic code generation from natural language. Journal of Information Processing Systems 17, 3 (2021), 537–555.

[39] Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, et al. 2022. Challenging big-bench tasks and whether chain-of-thought can solve them. arXiv preprint arXiv:2210.09261 (2022).

[40] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering. 1433–1443.

[41] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2024. Improving LLM Code Generation with Grammar Augmentation. arXiv preprint arXiv:2403.01632 (2024).

[42] Longyue Wang, Chenyang Lyu, Tianbo Ji, Zhirui Zhang, Dian Yu, Shuming Shi, and Zhaopeng Tu. 2023. Document-level machine translation with large language models. arXiv preprint arXiv:2304.02210 (2023).

[43] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. arXiv:2203.11171 [cs.CL]

[44] Zezhong Wang, Fangkai Yang, Pu Zhao, Lu Wang, Jue Zhang, Mohit Garg, Qingwei Lin, and Dongmei Zhang. 2023. Empower large language model to perform better on industrial domain-specific question answering. arXiv preprint arXiv:2305.11541 (2023).

[45] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems 35 (2022), 24824–24837.

[46] Brandon T Willard and Rémi Louf. 2023. Efficient Guided Generation for Large Language Models. arXiv e-prints (2023), arXiv–2307.

[47] Junchao Wu, Shu Yang, Runzhe Zhan, Yulin Yuan, Derek F Wong, and Lidia S Chao. 2023. A survey on llm-gernerated text detection: Necessity, methods, and future directions. arXiv preprint arXiv:2310.14724 (2023).

[48] Zhaozhuo Xu, Zirui Liu, Beidi Chen, Yuxin Tang, Jue Wang, Kaixiong Zhou, Xia Hu, and Anshumali Shrivastava. 2023. Compress, then prompt: Improving accuracy-efficiency trade-off of llm inference with transferable prompt. arXiv preprint arXiv:2305.11186 (2023).

[49] Zezhou Yang, Sirong Chen, Cuiyun Gao, Zhenhao Li, Ge Li, and Rongcong Lv. 2023. Deep learning based code generation methods: A literature review. arXiv preprint arXiv:2303.01056 (2023).

[50] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. Advances in Neural Information Processing Systems 36 (2024).

[51] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. 2023. Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. arXiv preprint arXiv:2304.10778 (2023).

[52] Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. 2023. MetaMath: Bootstrap Your Own Mathematical Questions for Large Language Models. arXiv preprint arXiv:2309.12284 (2023).

[53] Wenqi Zhang, Yongliang Shen, Linjuan Wu, Qiuying Peng, Jun Wang, Yueting Zhuang, and Weiming Lu. 2024. Self-contrast: Better reflection through inconsistent solving perspectives. arXiv preprint arXiv:2401.02009 (2024).

[54] Junchen Zhao, Yurun Song, Junlin Wang, and Ian G Harris. 2022. GAP-Gen: Guided Automatic Python Code Generation. arXiv preprint arXiv:2201.08810 (2022).

[55] Xu Zhao, Yuxi Xie, Kenji Kawaguchi, Junxian He, and Qizhe Xie. 2023. Automatic Model Selection with Large Language Models for Reasoning. arXiv preprint arXiv:2305.14333 (2023).

[56] Chuanyang Zheng, Zhengying Liu, Enze Xie, Zhenguo Li, and Yu Li. 2023. Progressive-hint prompting improves reasoning in large language models. arXiv preprint arXiv:2304.09797 (2023).

[57] Huaixiu Steven Zheng, Swaroop Mishra, Xinyun Chen, Heng-Tze Cheng, Ed H Chi, Quoc V Le, and Denny Zhou. 2023. Take a step back: evoking reasoning via abstraction in large language models. arXiv preprint arXiv:2310.06117 (2023).

[58] Wenqing Zheng, SP Sharan, Ajay Kumar Jaiswal, Kevin Wang, Yihan Xi, Dejia Xu, and Zhangyang Wang. 2023. Outline, then details: Syntactically guided coarse-to-fine code generation. In International Conference on Machine Learning. PMLR, 42403–42419.

[59] Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, and Hongsheng Li. 2023. Solving Challenging Math Word Problems Using GPT-4 Code Interpreter with Code-based Self-Verification. arXiv:2308.07921 [cs.CL]

[60] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. 2023. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. arXiv:2205.10625 [cs.AI]