ReportGPT: Human-in-the-loop Verifiable Table-to-Text Generation

Lucas Cecchi
J.P. Morgan AI Research
New York, NY
lucas.cecchi@jpmchase.com

Petr Babkin
J.P. Morgan AI Research
New York, NY
petr.babkin@jpmchase.com

Abstract

Recent developments in the quality and accessibility of large language models have precipitated a surge in user-facing tools for content generation. Motivated by a necessity for human quality control of these systems, we introduce ReportGPT: a pipeline framework for verifiable human-in-the-loop table-to-text generation. ReportGPT is based on a domain specific language, which acts as a proof mechanism for generating verifiable commentary. This allows users to quickly check the relevancy and factuality of model outputs. User selections then become few-shot examples for improving the performance of the pipeline. We configure 3 approaches to our pipeline, and find that usage of language models in ReportGPT's components trade off precision for more insightful downstream commentary. Furthermore, ReportGPT learns from human feedback in real-time, needing only a few samples to improve performance.

1 Introduction

Data-to-text generation has been a longstanding problem in natural language processing (Gatt and Krahmer, 2018; Sharma et al., 2022). Recent advancements in deep learning gave rise to transformer-based models that have achieved state of the art performance (Gatt and Krahmer, 2018; Sharma et al., 2022; OpenAI, 2022; Manyika, 2023). Within any real-world context, these systems must grapple with hallucinations and omissions caused by the underlying language model. Neglecting to address this may lead to the dissemination of misleading or false content. With this motivation we introduce ReportGPT, a framework for human-in-the-loop table-to-text generation that consists of a domain specific language and a set of modules that use it as a representation for generating verifiable commentary. Human verification of Data-to-Text Generation, while clearly vital, can be difficult and time consuming especially when it involves checking numerical calculations, as the human must perform the calculations in order to verify the output. The ReportGPT DSL acts as a proof mechanism, allowing users to effectively verify outputs for relevancy and correctness. Our approaches learn efficiently from human feedback using a Bayesian updating mechanism and few-shot prompting with language models. Our experiments show that usage of language models in our pipeline trade precision for insightfulness. As a result, while there are fewer factual outputs, those that are accurate tend to be more insightful for the end user.

2 Related Works

Many approaches to data-to-text generation utilize pre-trained, instruction tuned large language models (Manyika, 2023; OpenAI, 2022; Sanh et al., 2022; Ouyang et al., 2022). Various recent works have proposed improving the factuality and relevancy of these models by grounding the outputs with logical representations (Saha et al., 2022; Liu et al., 2022; Gao et al., 2023). Saha et al. (2022) utilize a logical form to represent reasoning paths, which are then ranked and converted to natural language via a surface realization. The ranking uses a BERT-base model trained to classify reasoning path and table tuples as correct or incorrect. During generation, a best-first search is conducted using the correct class probability as a saliency score. Liu et al. (2022) pre-train transformers on table to logical form objectives and then fine-tune on table to text objectives. Gao et al. (2023) utilize python programs as intermediate reasoning steps in the chain of thought of language models and demonstrate their effectiveness across 13 mathematical, symbolic and algorithmic reasoning tasks. OpenAI Code Interpreter (Lu, 2023), utilizes python programs in the chain of thought to perform a wide variety of tasks related to data processing and analysis. Hennigen et al. (2023) propose symbolically grounded generation, where LLMs are prompted to interleave their output with references to spans of input text. These references are used to reduce the effort of manual verification. ReportGPT defines a domain specific language that serves both as an intermediate representation to logically ground downstream text and as a verification mechanism. The DSL contains direct references to the input table, allowing users to quickly verify corresponding textual outputs. Going one step further, our pipeline utilizes human feedback as an online learning signal to improve precision.

3 ReportGPT

The ReportGPT pipeline consists of a domain specific language for computations over tabular data, as well as a set of modules that interface with it. The modules are: *Program Generation*, *Program Execution*, *Surface Realization*, and *Human Feedback*. The pipeline flow is as follows: *Program Generation* takes a table as input, outputs a distribution over programs and samples a batch from this distribution. These programs are then executed by the *Program Execution* module. The batch of programs and their results are fed into the *Surface Realization* module, which outputs a single sentence natural language description corresponding to each program and result tuple. These sentences, as well as their underlying program and result tuple, are shown to the user. The user verifies the correctness and relevancy of each individual sentence by checking its alignment with the underlying program. The *Human Feedback* module stores the user selections, both positive and negative, and uses them to update the *Program Generation* and *Surface Realization* modules. This loop is iterated until the user is satisfied with their set of verified sentences. Figure 1 provides an example of the full execution of the pipeline, showing the functionality of each module and intermediate steps. Sections 3.1 and 3.2 describe how tables and programs are represented within ReportGPT, respectively. Sections 3.3-3.6 discuss each of the above-mentioned modules in detail.

3.1 Table Linearization

Many data-to-text generation systems represent tables as linear sequences of attribute value pairs (Zhang et al., 2020; Radford et al., 2018; Raffel et al., 2020; Kasner and Dusek, 2022). This format gives poor scaling of the number of tokens required to encode the table, $\mathcal{O}(\text{Rows} * \text{Columns})$, which can be problematic for usage with language models due to their finite context size and high cost per token. If we limit our DSL, described in section 3.2, to only require header information we can limit the linearization to the title, row headers, and column headers. This format, shown below, brings our token count down to $\mathcal{O}(\text{Rows} + \text{Columns})$.

```
Title: <TITLE>
Rows: <ROW HEADERS>
Columns: <COL HEADERS>
```

3.2 ReportGPT Domain Specific Language

Now that we have defined a suitable table representation, we define a DSL that performs calculations over this table. Programs are generated by the Program Generation module (section 3.3), executed by the Program Execution module (section 3.4), and reasoned over by the Surface Realization module (section 3.5). End-to-end models for data-to-text generation, notably T5 (Raffel et al., 2020), struggle to generate good summaries when numerical calculations are involved (Sharma et al., 2022). Intermediate program representations remedy this by allowing the model to first generate programs, which are automatically executed, and then reason about their results. This assists models that struggle with numerical calculation while excelling at program generation and program reasoning tasks, and is a common approach to neural data-to-text generation (Saha et al., 2022; Chen et al., 2020; Liu et al., 2022; Gao et al., 2023; Cheng et al., 2022). Several of these works (Saha et al., 2022; Chen et al., 2020; Cheng et al., 2022) opt for a minimal language instead of Microsoft Excel or Python. This makes the programs simpler and lighter weight while maintaining high expressiveness. Within the ReportGPT framework we require that our language supports useful operations on tabular data and that its alignment with a corresponding natural language sentence can be human verified. To be verifiable, it should be human readable and easily linked back to the input table. We propose a domain specific language that trades expressiveness for simplicity and readability. The ReportGPT domain specific language contains the 12 operations shown below.

```
get, sum, avg, max, min,
argmax, argmin, std,
eq, less_than, diff, proportion
```

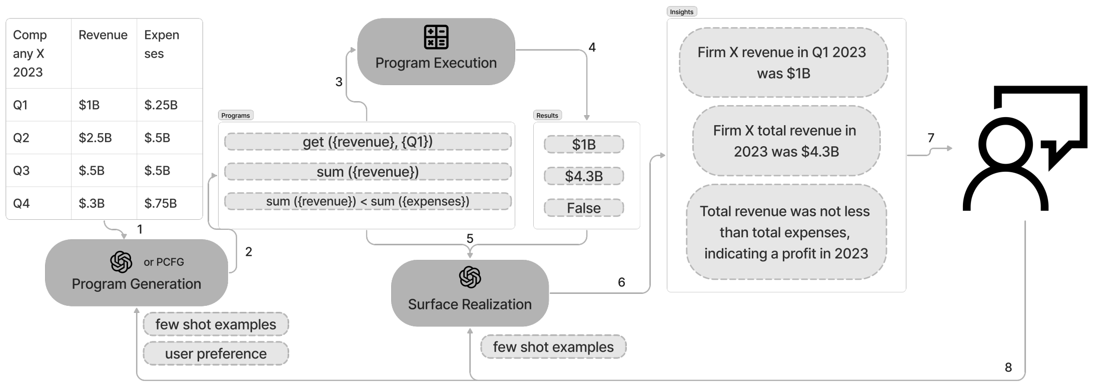

Figure 1: The ReportGPT Architecture. Tables are fed into the program generation module consisting of a language model call, several chained language model calls, or a PCFG. Programs are executed and fed into the surface realization module, which consists of a single language model call. This module outputs the final commentary, which is shown to the end user. User selections are used as few shot examples for the Program Generation and Surface Realization Modules.

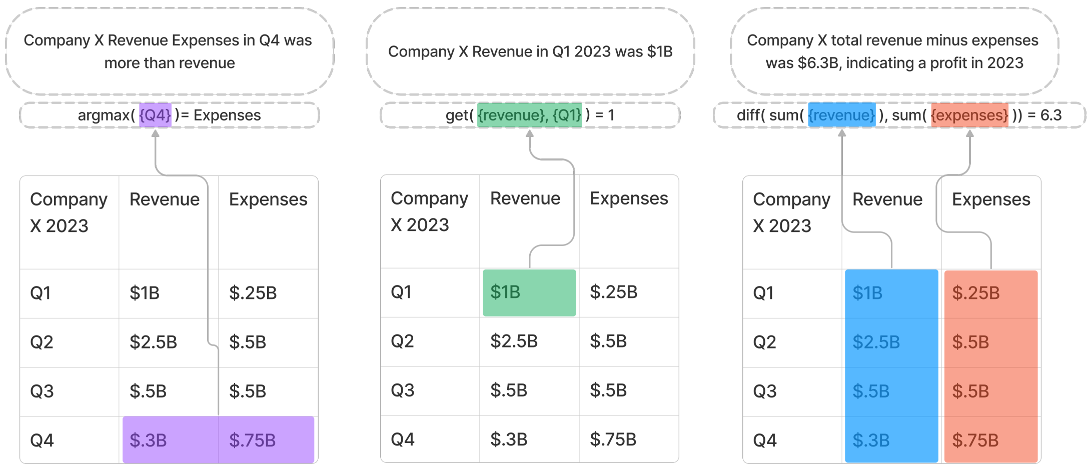
Company X Revenue Expenses in Q4 was more than revenue
argmax({Q4})= Expenses
Company X Revenue in Q1 2023 was $1B
get({revenue}, {Q1}) = 1
Company X total revenue minus expenses was $6.3B, indicating a profit in 2023
diff(sum({revenue}), sum({expenses})) = 6.3
Company X 2023
Revenue
Expenses
Q1
$1B
$.25B
Q2
$2.5B
$.5B
Q3
$.5B
$.5B
Q4
$.3B
$.75B

Figure 2: Illustration of the verification process. Commentary is linked to the table via corresponding programs. Programs contain row and column headers that are linked to sections of the table via color coded highlighting

In a Lisp-like syntax, each operator is matched to its operands in parenthesis. Valid operands consist exclusively of row and column headers, which are referenced in curly braces, as well as results from other operations. Refer to Figures 2 and 1 for examples of programs along with their target tables and corresponding results.

The following sections describe the four main ReportGPT modules illustrated in Figure 1.

3.3 Program Generation

Given an input table, the Program Generation module iteratively produces programs that are executable over the table. Formally, the module outputs a distribution of progams conditioned on this table. At inference time, we sample a batch from this distribution without replacement, using a temperature hyper-parameter to control the randomness of sampling. Next, we pass the sampled batch to the next module in the pipeline: Program Execution. Downstream, these programs and their execution results are realized into commentary sentences. The Human Feedback module, described in section 3.6, matches natural language sentences with corresponding programs generated by the Program Generation module to an accept or reject decision. Once these selections are available, the module should update to produce programs more likely to be accepted by the user. Programs that correspond to accepted sentences should not be generated again, as they have already been reviewed by the user. With these requirements in mind, we define three approaches to Program Generation.

3.3.1 PCFG-based

In our first approach, we define a probabilistic context-free grammar and devise a simple mechanism for generating programs that can be efficiently updated with user preferences. A Context-Free Grammar consists of a set of non-terminal strings, $\{\alpha_1, \alpha_2, ...\alpha_n\}$, and a set of production rules that can be applied to each non-terminal string $\alpha_i \to \beta_0, ...\alpha_i \to \beta_m$. A Probabilistic Context-Free Grammar consists of a Context-Free Grammar and set of probabilities for each production rule given a non-terminal. We write these probabilities as $P(\alpha_i \to \cdot|\alpha_i)$ with $\sum_{j=1}^{m} P(\alpha_i \to \beta_j|\alpha_i) = 1$.

A PCFG can be sampled to produce a string by beginning with the starting non-terminal S, and iteratively applying production rules to non-terminal strings until left with exclusively terminal strings. Production rules are selected for each expansion of a non-terminal α_i by sampling $\alpha_i \to \beta_j$ from $P(\alpha_i \to \cdot|\alpha_i)$. We define a Context-Free Grammar for ReportGPT DSL, shown below.

```
S -> Z | Y Z Z
Z -> get {R} {C} | X {R} | X {C}
X -> sum | avg | max | min
    | argmax | argmin | std
    | eq | less_than | diff | proportion
Y -> eq | less_than | diff | proportion
R -> <ROW HEADERS>
C -> <COLUMN HEADERS>
```

Generating a program using the PCFG also generates a parse tree: the set of production rules selected during generation. Given a set of production rules containing $\alpha_1 \to \beta_1, \alpha_2 \to \beta_2, \ldots, \alpha_n \to \beta_n$, we define COUNT as the following:

$$\text{COUNT}(\alpha \to \beta) = \sum_{i=1}^{n} [\alpha_i \to \beta_i = \alpha \to \beta]$$

Downstream, the user makes selections based on natural language sentences with underlying programs. This produces two sets of transitions, one corresponding to accepted programs and another to rejected programs. We count the number of times that a certain transition appears in accepted programs $\text{COUNT}_{\text{acc}}(\alpha \to \beta)$, and rejected programs $\text{COUNT}_{\text{rej}}(\alpha \to \beta)$. We then compute the acceptance rate for this transition and apply a soft-max with a temperature parameter θ to obtain a probability.

$$R_{\alpha \to \beta} = \frac{\text{COUNT}_{\text{acc}}(\alpha \to \beta) + 1}{\text{COUNT}_{\text{acc}}(\alpha \to \beta) + \text{COUNT}_{\text{rej}}(\alpha \to \beta) + 1}$$

$$P(\alpha \to \beta|\alpha) = \frac{\exp(\frac{R_{\alpha \to \beta}}{\theta})}{\sum_{\beta} \exp(\frac{R_{\alpha \to \beta}}{\theta})}$$

3.3.2 LLM-based

Large language models, specifically GPT-4, have shown remarkable performance in program generation conditioned on human intent (Bubeck et al., 2023). As an alternative to a PCFG, we utilize GPT-4 for Program Generation. We implement this as a single API call with a prompt that includes the task description, the linearized table, a short description of ReportGPT DSL, and few-shot examples. These few-shot examples consist of programs corresponding to commentary that has been accepted by the user. We ask the language model not to generate these programs again. Note that we can apply a temperature parameter to our API calls to increase or decrease randomness.

3.3.3 Chaining LLMs

Given recent advances in chain of thought prompting and chaining language models (Wu et al., 2022b,a; Wei et al., 2022), we hypothesized that allowing the model to first generate relevant questions about the table, and then answer them, would lead to more insightful downstream commentary. To achieve this, we chain two GPT-4 API calls: one that ingests the table metadata and generates questions, and another that generates programs to answer these questions. The first API call prompt consists of a task description, the linearized table, a description of ReportGPT DSL, and few-shot examples. The second API call prompt is similar, but with a different task description and with the output of the previous call concatenated at the end. Few-shot examples consist of questions in call 1, and (question, program) tuples in call 2. These examples are taken from corresponding user accepted commentary.

3.4 Program Execution

The Program Execution module evaluates a program on an input table and outputs the result. We implement it as an interpreter for ReportGPT DSL written in Python. First, the table is represented as a Python object. The interpreter then scans the program from left to right, matching operators to a operands in parenthesis.

3.5 Surface Realization

The Surface Realization module turns program representations and their execution results into natural language sentences. These sentences should be a faithful descriptions of their corresponding programs, without incorporating outside knowledge or omitting any facts. The Human Feedback module is directly responsible for filtering these errors in the process described in section 3.6. Surface Realization is a common task in the NLP literature for which Language Models have exhibited strong performance (Farahnak et al., 2020; Saha et al., 2022). Thus, we implement surface realization as a GPT-4 API call. The API call prompt consists of a task description, the linearized table a description of ReportGPT DSL, few-shot example tuples of (program, result, sentence), and the input (programs, results).

3.6 Human Feedback

The automated portion of the ReportGPT pipeline ingests tables and produces natural language sentences with corresponding programs and their execution results. The human portion of ReportGPT, which we refer to as the Human Feedback module, is responsible for individually accepting or rejecting each output. The user is presented with tuples of (commentary sentence, program, result) linked back to the table through column and row highlighting, as depicted in Figure 2. The user then chooses to accept or reject each tuple depending on the following criteria. First, any tuple where the commentary contains omissions or hallucinations is rejected. Second, any tuple that contains uninteresting or trivial commentary, according to the individual user preference, is rejected. While the second may vary based on user preferences, the first criterion ensures that output commentaries are free of errors.

3.7 Batching

As seen in previous sections, we utilize GPT-4 APIs for our Surface Realization, and Program Generation modules. As a result, each forward pass of our pipeline may require as many as 4 API calls. These calls incur a high latency cost which translates to a low quality user experience. Cheng et al. (2023) propose batch prompting, a method that concatenates a batch of samples into a single prompt. Their experiments show no significant drops in performance while increasing throughput by a factor of the batch size. We adapt this approach to our pipeline and choose a batch size of 5, which we use for all of our calls.

4 Experiments

In this section, we describe our experimental design and results.

4.1 Dataset

In order to evaluate our proposed pipeline, we conduct a user study using tables from the HiTab dataset (Cheng et al., 2022). Hitab contains 3,700 complex tables sourced from over 30 domains. The tables contain noise in the form of missing cells. This approximates real-world data, and thus provides a suitable benchmark for how ReportGPT might perform in real-world use-cases.

4.2 Pipeline configurations

For all experiments shown in Tables 1 and 2, we report the results for 3 pipeline configurations and 4 settings. The configurations are: PCFG-based program generation, LLM-based program generation, and chained LLM-based program generation. The settings are zero-shot, 1-shot, 3-shot, and with manually written table titles. For zero-shot, we provide the HiTab tables to the pipeline as-is with no human feedback. We then construct 1-shot and 3-shot experiments using user labels from zero-shot. A notable source of errors in these experiments is that tables in HiTab's (Cheng et al., 2022) dataset are missing descriptive titles. In order to measure the effect this has on performance, we manually write titles for 20 tables and re-run our zero-shot experiment.

4.3 Experiment 1: Acceptance, Hallucination, and Error rates

For our first experiment, seen in Table 1, we run each pipeline configuration and setting on a set of tables. For each table in the experiment, the pipeline is used to generate 5 (program, result, commentary) tuples. For each tuple, the annotator is asked to choose 'accept', 'hallucination', 'program error'. Program error is chosen if the underlying program throws an error or is malformed, hallucination is chosen if the commentary is incorrect or not aligned with the underlying program. Accept is chosen if the commentary is correct and there is no program error. Additionally, the pipeline may fail to generate 5 samples for a table. In this case we report the missing tuples as 'dropped'. We utilize 4 researchers to annotate this task.

4.4 Experiment 2: Ranking and Number of Operations

For our second experiment, we compile accepted commentary from all configurations and settings in experiment 1. For each annotator, we sample 45 tables and 1 accepted commentary per table from

Configuration	Model	Acc. Rate	Halluc. Rate	Prog. Err. Rate	Gen.	Drop.
0-shot	Chained	31.81%	22.22%	43.18%	396	9
	LLM	61.0%	18.43%	19.11%	293	7
	PCFG	**84.0**%	18.33	0%	300	0
1-shot	Chained	54.0%	18.67%	9.33%	75	15
	LLM	73.1%	6.66%	12.22%	90	10
	PCFG	**87**%	13.0%	0%	100	0
3-shot	Chained	93.0%	0%	7.0%	100	0
	LLM	78%	4.0%	18.0%	100	0
	PCFG	**95.0**%	5.0%	0%	100	0
titles	Chained	62.0%	11.0%	27.0%	100	0
	LLM	89.0%	3.0%	8.0%	100	0
	PCFG	**98.0**%	2.0%	0%	100	0

Table 1: Acceptance, Hallucination, and Error Rates

Model	Avg. Rank	Avg. num ops
Chained	**1.71**	1.07
LLM	2.06	1.42
PCFG	2.23	1.26

Table 2: Ranking and Average Number of Operations

each of the 3 pipeline configurations. We present the table, as well as the 3 samples to the annotator and ask them to rank them in order of insightfulness. We define the insightfulness of commentary as its descriptiveness and usefulness to the reader. We take the average of these rankings and report them in Table 2. We utilize 2 researchers as annotators for this task, each giving rankings for 45 pairs of three samples with each pair of three taken from a distinct table. Additionally, we compute the average number of operations in the programs of accepted outputs for each configuration and report it in Table 2.

5 Results and Discussion

Table 1 shows that acceptance rates increase with few-shot examples, as well as with labelled tables. Table 1 also shows that our PCFG-based model attains the highest acceptance rates across the board, while scoring the lowest on our ranking experiment shown in Table 2. This highlights a trade-off between precision and insightfulness. Our PCFG based pipeline imposes the most rule-based constraints, and attains high precision with low insightfulness. It does so by generating programs from a pre-defined context-free grammar, which limits the output space compared to generating the programs from a language model. Our LLM based pipelines impose less constraints, and thus trade off increased insightfulness for decreased precision.

The results in Tables 1 and 2 demonstrate that downstream applications might benefit from different (or possibly hybrid) configurations. A PCFG-based approach increases the acceptance rate but does not necessarily produce commentary that is novel or insightful. In contrast, the Chained approach provides higher insightfulness and might be preferred in settings when multi-shot prompting is feasible.

Lastly, we examine whether the number of operations in the program is associated with the insightfulness of the commentary. This is based on the hypothesis that sophisticated calculations can lead to more novel or non-trivial outputs. Table 2 lists the number of operations against the ranking of each pipeline's outputs. As the table shows, a higher number of operations does not necessarily translate to more insightful commentary, demonstrating that insightfulness is a more semantically complex concept and automating it based on proxy metrics might not be useful to downstream applications.

6 Conclusion

Motivated by the necessity for human supervision in real world use cases, we introduce ReportGPT: a pipeline framework for verifiable human-in-the-loop table-to-text generation. ReportGPT consists of a domain specific language that enables verifiability, as well as a set of modules that generate and reason about it. We configure 3 approaches to our pipeline, and find a trade-off between precision and insightfulness.

7 Disclaimer

References

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *Preprint*, arXiv:2303.12712.

Wenhu Chen, Jianshu Chen, Yu Su, Zhiyu Chen, and William Yang Wang. 2020. Logical natural language generation from open-domain tables. *Preprint*, arXiv:2004.10404.

Zhoujun Cheng, Haoyu Dong, Zhiruo Wang, Ran Jia, Jiaqi Guo, Yan Gao, Shi Han, Jian-Guang Lou, and Dongmei Zhang. 2022. HiTab: A hierarchical table dataset for question answering and natural language generation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1094–1110, Dublin, Ireland. Association for Computational Linguistics.

Zhoujun Cheng, Jungo Kasai, and Tao Yu. 2023. Batch prompting: Efficient inference with large language model apis. *Preprint*, arXiv:2301.08721.

Farhood Farahnak, Laya Rafiee, Leila Kosseim, and Thomas Fevens. 2020. Surface realization using pretrained language models. In *Proceedings of the Third Workshop on Multilingual Surface Realisation*, pages 57–63, Barcelona, Spain (Online). Association for Computational Linguistics.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. *Preprint*, arXiv:2211.10435.

Albert Gatt and Emiel Krahmer. 2018. Survey of the state of the art in natural language generation: Core tasks, applications and evaluation. *J. Artif. Int. Res.*, 61(1):65–170.

Lucas Torroba Hennigen, Shannon Shen, Aniruddha Nrusimha, Bernhard Gapp, David Sontag, and Yoon Kim. 2023. Towards verifiable text generation with symbolic references. *Preprint*, arXiv:2311.09188.

Zdeněk Kasner and Ondrej Dusek. 2022. Neural pipeline for zero-shot data-to-text generation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3914–3932, Dublin, Ireland. Association for Computational Linguistics.

Ao Liu, Haoyu Dong, Naoaki Okazaki, Shi Han, and Dongmei Zhang. 2022. PLOG: Table-to-logic pretraining for logical table-to-text generation. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5531–5546, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Yiwen Lu. 2023. What to know about chatgpt's new code interpreter feature. *The New York Times*.

James Manyika. 2023. An overview of bard: an early experiment with generative ai.

OpenAI. 2022. Chat-gpt: Optimizing language models for dialogue.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, volume 35, pages 27730–27744. Curran Associates, Inc.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2018. Language models are unsupervised multitask learners.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Preprint*, arXiv:1910.10683.

Swarnadeep Saha, Xinyan Velocity Yu, Mohit Bansal, Ramakanth Pasunuru, and Asli Celikyilmaz. 2022. Murmur: Modular multi-step reasoning for semi-structured data-to-text generation. *Preprint*, arXiv:2212.08607.

Victor Sanh, Albert Webson, Colin Raffel, Stephen H. Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Fevry, Jason Alan Fries, Ryan Teehan, Tali Bers, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M. Rush. 2022. Multitask prompted training enables zero-shot task generalization. *Preprint*, arXiv:2110.08207.

Mandar Sharma, Ajay Gogineni, and Naren Ramakrishnan. 2022. Innovations in neural data-to-text generation.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*.

Tongshuang Wu, Ellen Jiang, Aaron Donsbach, Jeff Gray, Alejandra Molina, Michael Terry, and Carrie J Cai. 2022a. Promptchainer: Chaining large language model prompts through visual programming. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI EA '22, New York, NY, USA. Association for Computing Machinery.

Tongshuang Wu, Michael Terry, and Carrie J. Cai. 2022b. Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts. *Preprint*, arXiv:2110.01691.

Shuo Zhang, Zhuyun Dai, Krisztian Balog, and Jamie Callan. 2020. Summarizing and exploring tabular data in conversational search. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM.

A Program Generation Configurations

Figures 3, 4, and 5 illustrate the processes of program generation, chained program generation, and surface realization, respectively.

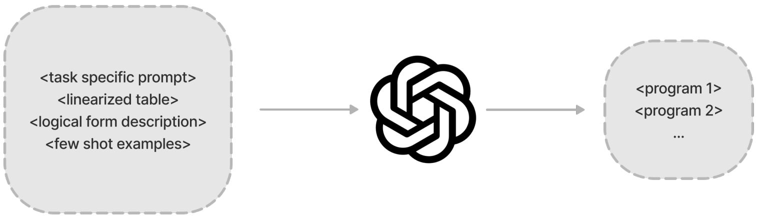

Figure 3: Prompt template and output format for LLM Program Generation module

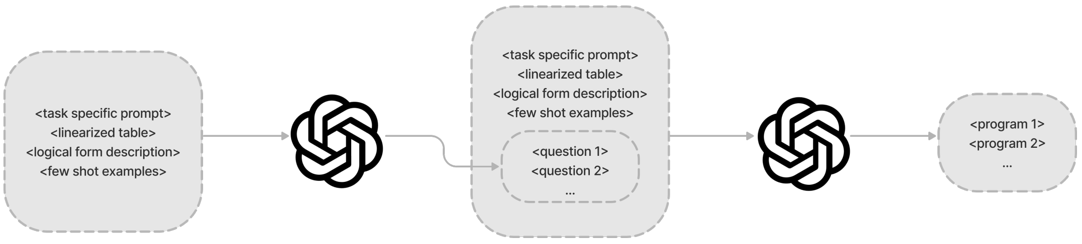

Figure 4: Prompt template and output format for Chained LLM Program Generation module

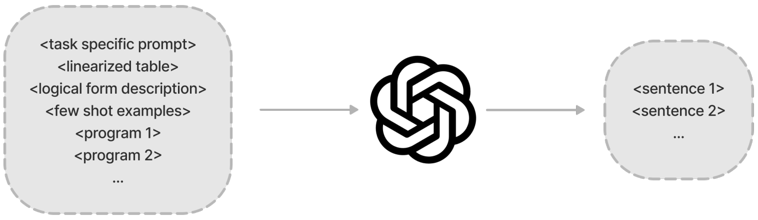

Figure 5: Prompt template and output format for LLM Surface Realization module