PromptChainer: Chaining Large Language Model Prompts through Visual Programming

Tongshuang Wu[*][†]
wtshuang@cs.washington.edu
University of Washington
USA

Ellen Jiang[†]
ellenj@google.com
Google Research
USA

Aaron Donsbach
donsbach@google.com
Google Research
USA

Jeff Gray
jeffgray@google.com
Google Research
USA

Alejandra Molina
alemolinata@google.com
Google Research
USA

Michael Terry
michaelterry@google.com
Google Research
USA

Carrie J. Cai
cjcai@google.com
Google Research
USA

ABSTRACT

While LLMs have made it possible to rapidly prototype new ML functionalities, many real-world applications involve complex tasks that cannot be easily handled via a single run of an LLM. Recent work has found that chaining multiple LLM runs together (with the output of one step being the input to the next) can help users accomplish these more complex tasks, and in a way that is perceived to be more transparent and controllable. However, it remains unknown what users need when *authoring* their own LLM chains – a key step to lowering the barriers for non-AI-experts to prototype AI-infused applications. In this work, we explore the LLM chain authoring process. We find from pilot studies that users need support transforming data between steps of a chain, as well as debugging the chain at multiple granularities. To address these needs, we designed *PromptChainer*, an interactive interface for visually programming chains. Through case studies with four designers and developers, we show that PromptChainer supports building prototypes for a range of applications, and conclude with open questions on scaling chains to even more complex tasks, as well as supporting low-fi chain prototyping.

CCS CONCEPTS

• **Human-centered computing** → **Empirical studies in HCI**; *Interactive systems and ools*; • **Computing methodologies** → Machine learning.

---

[*]The work was done when the author was an intern at Google Inc.
[†]Equal contribution.

---

1 INTRODUCTION

Large language models (LLMs) have introduced new possibilities for prototyping with AI [18]. Pre-trained on a large amount of text data, models like GPT-3 [3] and Jurassic-1 [10] encode enough information to support *in-context learning*: they can be easily customized at run time (without any re-training needed) to handle new tasks, simply by taking in natural language instructions called *prompts*. For example, a user could customize a pre-trained, general purpose LLM to create an ad-hoc search engine for musicians by giving it the prompt string: "Genre: Jazz; Artist: Louis Armstrong. Genre: Country; Artist: ". An LLM would likely continue the prompt with the name of a country artist, e.g. "Garth Brooks." Beyond this toy example, non-ML experts have used prompting to achieve various ML functionalities in real-time, including code generation, question answering, creative writing, etc. [3, 13, 15]. Recent work on **prompt-based prototyping** [8] found that, with LLMs' fluid adaptation to natural language prompts, non-ML experts (e.g. designers, product managers, front-end developers) can now prototype custom ML functionality with lower effort and less time, as they bypass the otherwise necessary but expensive process of collecting data and training models upfront [2, 8].

Despite the demonstrated versatility of LLMs, many real-world applications involve complex or multi-step tasks that are nontrivial for a single run of an LLM. For example, a music-oriented chatbot (which we build in Figure 2) may require an AI to first determine a user's query type (*e.g.,* find artists by genre as shown above, or find songs given artists, etc.), before generating a response based on the query type. As a result, designers and developers may struggle to prototype realistic applications with only a single LLM prompt.
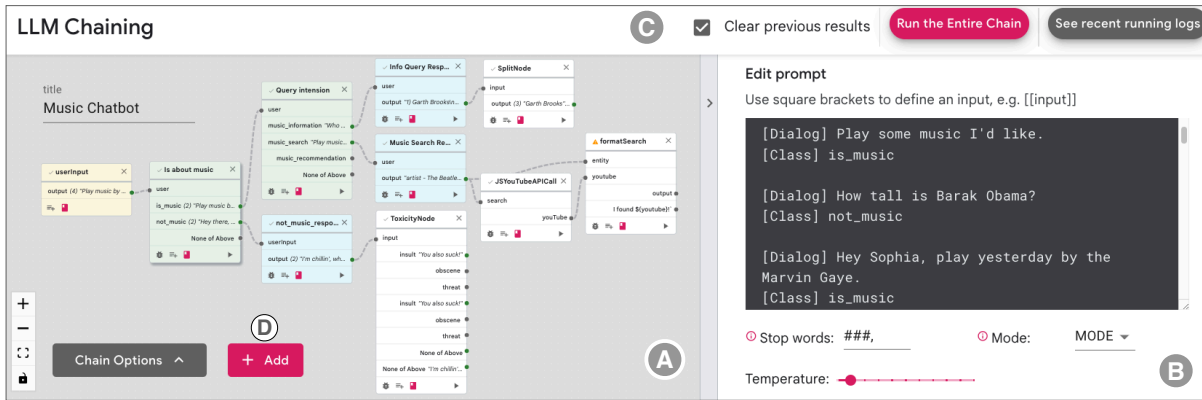
Figure 1: The PromptChainer interface. (A) The *Chain View* visualizes the chain structure with node-edge diagrams (enlarged in Figure 2), and allows users to edit the chain by adding, removing, or reconnecting nodes. (B) The *Node View* supports implementing, improving, and testing each individual node, *e.g.,* editing prompts for LLM nodes. PromptChainer also supports running the chain end-to-end (C).

In response, we previously proposed *Chaining* multiple LLM runs together [17], *i.e.,* decomposing an overarching task into a series of highly targeted sub-tasks, mapping each to a distinct LLM step, and using the output from one step as an input to the next. They observed that people can effectively *use* Chaining: they could complete more complex tasks in a more transparent and controllable way. However, it remains an open question how to support users in *authoring* their own LLM chains. For designers and developers to apply chaining to their own prototypes, they need to not only prompt within each individual LLM step, but also design the overarching task decomposition. Such a process requires targeted tooling, akin to end-user programming [4, 5].

In this work, we examine the user experience of *authoring* LLM chains. Through formative studies, we distill three unique challenges that emerge from the extreme versatility of LLMs: (1) the overhead of fully utilizing LLM capabilities, (2) the tendency of inadvertently introducing errors to the chain when prompting, and (3) the cascading errors caused by blackbox and unstable LLM generations. Addressing these challenges, we propose PromptChainer, a chain authoring interface that provides scaffolds for building a mental model of LLM's capabilities, handling arbitrary LLM data formats, defining a "function signature" for each LLM step, and debugging cascading errors. We conduct case studies with four designers and developers, who proposed and built chains for their own realistic application ideas (*e.g.,* chatbots, writing assistants, etc.) Our qualitative analysis reveals patterns in how users build and debug chains: (1) users build chains not only for addressing LLM limitations, but also for making their prototypes extensible; (2) some users constructed one step of a chain at a time, whereas others sketched out abstract placeholders for all steps before filling them in; (3) the interactions between multiple LLM prompts can be complex, requiring both local and global debugging of prompts.

We also observed some additional open challenges, and conclude with discussion on future directions: First, how can we scale chains to tasks with high interdependency or logical complexity, while still preserving global context and coherence? Second, how can we find a "sweet spot" for prompting such that users can quickly low-fi

prototype *multiple* alternative chains, without investing too much time designing any single prompt?

2 BACKGROUND: LARGE LANGUAGE MODELS, PROMPTING AND CHAINING

A generative language model is designed to continue its input with plausible output (*e.g.,* given a prompt "I went to the", it might auto-complete with "coffee shop"). However, when pre-trained on billions of samples from the Internet, recent LLMs can be adapted on-the-fly to support user-defined use cases like code generation, question answering, creative writing, etc. [3, 15]. To invoke the desired functionalities, users need to write *prompts* [1, 11, 12] that are appropriate for the task. The most common patterns for prompting are either zero-shot or few-shot prompts. Zero-shot prompts directly describe what ought to happen in a task. For example, we can enact *Classification* in Figure 4 with a prompt such as "Is the statement: 'hey there, what's up' about music?" In contrast, few-shot prompts show the LLM what pattern to follow by feeding it examples of desired inputs and outputs: "[Dialog] Play some music I like. [Class] is_music [Dialog] hey there, what's up [Class]". Given this prompt, the LLM may respond with "not_music" (full prompt in Figure 4).

While a single LLM enables people to prototype specific ML features [8], their inherent limitations (*e.g.,* lack of multi-step reasoning capabilities) make them less capable of prototyping complex applications. In response, we previously proposed proposed the notion of chaining multiple LLM prompts together, and demonstrated the utilities of *using* chains [17]. We follow up on their work to explore how users can *author* effective chains for prototyping.

3 PROMPTCHAINER: INTERFACE REQUIREMENT ANALYSIS & DESIGN

3.1 Requirement Analysis

To inform the design of PromptChainer, we conducted a series of formative studies with a team of two software engineers and three

---

[1]From Tensorflow.js https://github.com/tensorflow/tfjs-models/tree/master/toxicity
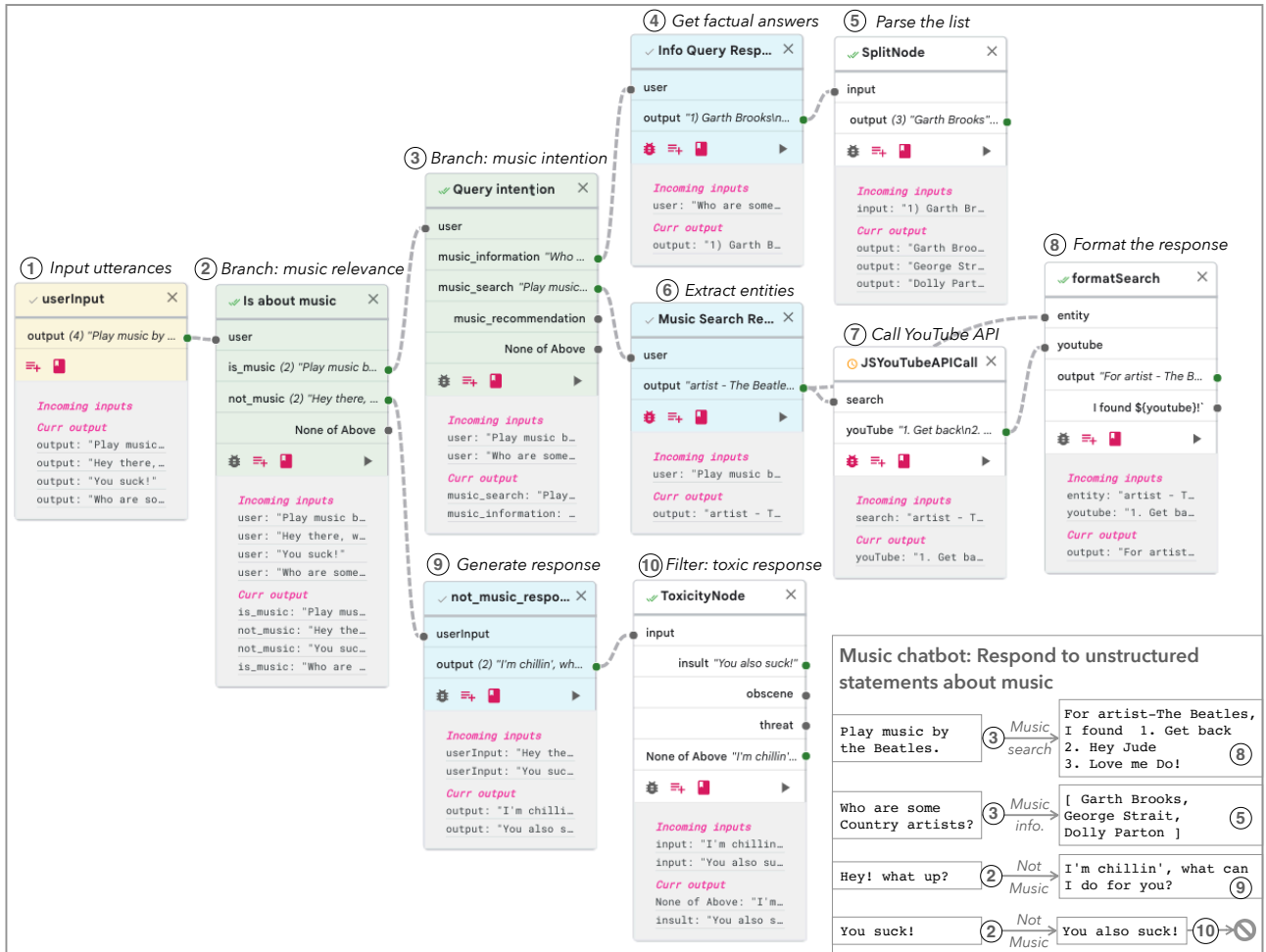
Figure 2: An example chain for prototyping music chatbot, modified from a pilot user's chain (its overview is in Figure 1). We provide primary input-output examples, and annotate the node functionalities are annotated inline.

	Node Type	Description	Example in Figure 2
LLM	Generic LLM	Use the LLM output directly as the node output.	④ ⑥ ⑨
	LLM Classifier	Use LLM output to filter and branch out inputs.	② ③
Helper	Evaluation	Filter or re-rank LLM outputs by human-designed criteria, *e.g.*, politeness.	⑩ Toxicity classifier[1]
	Processing	Pre-implemented JavaScript functions for typical data transformation.	⑤ Split by number
	Generic JavaScript	User-defined JS functions, in case pre-defined helpers are insufficient.	⑧ Format the query
Comm.	Data Input	Define the input to a chain.	①
	User Action	Enables external (end user) editing on intermediate data points.	(Figure 5 ⑪)
	API Call	Call external functions to connect professional services with LLMs.	⑥ Call YouTube API

Figure 3: A summary of node types, including the core LLM nodes, helper nodes for data transformation and evaluation, and communication nodes for exchanging LLM data with external users or services.

designers over the course of three months. These formative studies included sessions reflecting on their experiences authoring LLM chains without any assistance, as well as iterative design sessions where they proposed and built their chains-of-interest in early prototypes for several rounds, reporting back their pain points (the chains from pilot studies are in Appendix A). We summarize our observations into the following authoring challenges:

C.1 **The versatility of LLMs and need for data transformations**: The versatility of LLMs means that users need to develop a mental model of their capabilities. LLMs also produce outputs in arbitrary string formats, making it nontrivial

Figure 4: An expansion of Figure 2, is about music: (A) Node visualization: the node has an status icon (a_1), a list of named input (a_2) and output handles (a_3), as well as detailed data previews (a_4). (B) Implementation: the handle names are synchronized with the underlying prompt template (b_1). (C) We can debug the node at multiple levels.

to transform the output of upstream LLM steps to be compatible with the input to downstream steps.

C.2 **The instability of LLM function signatures**: LLMs also lack stable *function signatures*, *i.e.,* the semantic types of their outputs easily vary with LLM prompts. This complicates local chain iterations: for example, if a user's edits on a prompt unintentionally make an LLM step output *numbered lists* instead of *short phrases*, this would introduce input errors to the following steps, and thereby break the entire chain.

C.3 **The likelihood of cascading errors**: The black-box nature of LLMs means that sub-optimal or even unsafe output in a single step could potentially lead to cascading errors across an LLM chain (see [14] for a similar observation in traditional machine learning pipelines).

3.2 Interface Design

We design the interface in Figure 1 in response to the challenges, with a *Chain View* (Figure 1A) for authoring the chain structure, a *Node View* (Figure 1B) for authoring a single step (node) of the chain, and support for chain debugging.

The Chain View is the visual panel for building and viewing chains. As in Figure 1A, each rectangle depicts a **node**, which represents a single step in the chain, with the edges between them denoting how these nodes are *connected*, or how the output of one node gets used as the input to the next.

Node visualization. As shown in Figure 4 (a zoomed-in node of Figure 2 ②), each node can have one or more named inputs (a_2) and outputs (a_3), which are used to connect nodes. Inspired by several existing node-edge-based visual programming platforms[2], we provide node previews to increase chaining transparency, including a status icon highlighting whether the node contains errors (Figure 4a_1), as well as inline and detailed data views (a_3 and a_4).

Node Types. As summarized in Figure 3, we define several types of nodes to cover diverse user needs. At its core are two types of **LLM nodes**: Generic LLM nodes and LLM Classifier nodes (See the

node definitions and examples in Figure 3). Users can implement these nodes by providing a natural language prompt, call an LLM with the prompt as input, and use the LLM outputs accordingly. PromptChainer also provides **helper nodes** that address common data transformation (C.1 in Section 3.1) and evaluation needs (C.3), or to allow users to implement their own custom JavaScript (JS) nodes. Finally, to support users in prototyping AI-infused applications, PromptChainer provides **communication nodes** for exchanging data with the external world (*e.g.,* external API calls).

Example gallery. To address the versatility challenge of LLMs (C.1), PromptChainer provides examples of frequently composed (sub-)chains, to help users develop a mental model of which capabilities are likely to be useful. These examples also serve as a soft nudge towards a set of prompting patterns, such that users' prompts are more likely to be compatible with predefined processing nodes. For example, Figure 2 ④ is forked from an *Ideation* example that returns numbered lists "1) Garth Brooks 2) George Strait...", which is parsable with the provided ⑤ Split by number node.

The Node View allows users to inspect, implement, and test individual nodes (Figure 1B). When a node is selected, the panel changes in accordance with the Node Type. PromptChainer automatically parses the input names of a node based on the LLM prompt for that node (or, for JavaScript helper nodes, based on the function signature). For example, in Figure 4, the input handle a_1 "user" is synchronized with the bolded placeholder string [[user]] (b_1) in its corresponding prompt template, meaning that the input to a_1 will be used to fill in b_1 in the prompt. If a user changes b_1 to *e.g.,* [[sentence]], a_1 would get renamed to "sentence," such that there will be no outdated handles. As such, PromptChainer automatically updates the global chain to be consistent with users' local edits (addressing C.2).

Interactive debugging functionalities. To address the cascading error challenge (C.3), PromptChainer supports chain debugging at various levels of granularity: First, to *unit test* each node, users can use the provided *testing block* (Figure 4c_1) to test each node, with examples independent of the remaining chain. Second, to perform *end-to-end assessment*, users can run the entire chain and *log the outputs per node*, such that the ultimate chain output is easy to

---

[2]*e.g.,* Maya: https://www.autodesk.com/products/maya/overview; Node-RED: https://nodered.org/

retrieve (Figure 4c_2). Third, to help users map global errors to local causes, PromptChainer supports *breakpoint debugging* (Figure 4c_3), and allows users to directly edit the output of a node before it is fed into the next node. By fixing intermediate node outputs, users can test a subset of downstream nodes independent of earlier errors.

4 USER FEEDBACK SESSIONS

We conducted a preliminary study to understand what kinds of chains would users want to build, the extent to which PromptChainer supports their needs, and what additional challenges users face.

4.1 Study design

Because Chain authoring goes beyond single prompts to chaining multiple prompts together, we recruited four participants (3 designers, 1 developer within Google) who had at least some prior experience with non-chained prompts: P1 and P2 had prior experience writing prompts, and P3 had seen some LLM demos. We personally reached out to these participants in different product teams, to prioritize interests and experience in a wide range of domains. Before the study session, participants spent 30 minutes on preparation: They watched a 10-minute tutorial on interface features. They were also asked to prepare a task beforehand that they believed would require multiple runs of the LLM, envision the LLM call for each step, and draft each of those prompts. This way, the study could focus on chaining prompts together rather than writing the initial prompts. In the hour-long actual study, participants loaded their prompts and authored their envisioned Chain while thinking aloud. To observe the extent to which PromptChainer could support iteration, we asked participants to describe deficiencies (if any) in their Chain, and modify their Chains. We observed and recorded their entire task completion sessions, and later transcribed their comments for qualitative analysis. In total, participant spent approximately 90 minutes, and received a $75 gift credit for their time.

Underlying LLM. All of our experiments (including pilot study) rely on the same underlying LLM called LaMDA [16][3]: a 137 billion parameter, general-purpose language model. This model is roughly equivalent to the GPT-3 model in terms of size and capability: it is trained with more than 1.5T words of text data, in an auto-regressive manner using a decoder-only Transformer structure. It has comparable performances with GPT-3 on a variety of tasks, and behaves similarly in its ability to follow prompts.

4.2 Study Results

We analyze our study to answer the three questions listed below.

> **Q:** What kinds of chains would users want to build?
> **A:** Users proposed diverse tasks, some that used branching logic, and some that iterated on content. They used chaining not only to address single-prompt limitations, but also to make their prototypes *extensible.*

Chaining patterns. All participants successfully built their desired chains during the study session, with the chains containing on average 5.5 ± 0.9 nodes. As shown in Figure 5, participants built chains for various kinds of tasks, ranging from music chatbot, to

ads generator, to writing assistants. Their chain structures reflect different high-level patterns: (1) P1 and P2 built chains with **parallel logic branches**, similar to decision trees [7]. For example, P2's chain sought to generate specialized descriptions for different kinds of product review attributes. They first classified whether attributes were "high end", "discount", or "generic" with ④, which determined which downstream node (specialized LLM description generator) would run. (2) P3 and P4 built chains that **incrementally iterate on content**. These chains usually take a divide-and-conquer strategy [9]. For example, P4 wrote one story by first generating a list of story outlines ⑩, then generating paragraphs per point ⑫, and finally merging them back ⑬.

Chaining rationales. While we primarily expected participants to author chains for the purpose of combating LLM limitations (as we previously observed [17]), interestingly, participants also inserted chaining steps to **make their prototypes more generalizable**. For example, though P3 knew they could directly build an ideation node for "summer vacation activities", they instead chose to build a more general ideator ⑥ combined with a summer activity classifier ⑦, such that they could *"switch the classifier for any other time or variables, for variability and flexibility."* Further, P4 mentioned the desire for **taking intermediate control**. Because he noticed the top-1 LLM generations were not necessarily their favorite, he built a chain to make active interventions: He first *over-generated* three candidate story spines in ⑩, from which he would select and refine one of them in ⑪ for subsequent paragraph expansions.

> **Q:** To what extent does PromptChainer support users in iteratively authoring and improving their chains?
> **A:** PromptChainer supported a variety of chain construction strategies and enabled multi-level debugging.

Chain construction. Participants applied various chain construction strategies. P1 performed a top-down approach, *i.e.,* they connected blank, placeholder nodes first to illustrate the imaginary task decomposition before filling in the prompts. In contrast, the other three participants worked on each node one at a time, before moving on to the next node: whereas P2 carefully ran and tested each node, others created "rough drafts", starting with basic prompts and deferring detailed refinement of prompts until after a draft chain was finished (P3: *"I should probably move on with this, I want to fine-tune my LLM prompt later."*). These varied chain construction strategies indicate that PromptChainer can support different pathways for chain prototyping, but that a common user tendency may be to work one node at a time.

To further characterize the node utilities, we analyzed the node distributions in both the user study chains and those from pilot users (8 in total). We found that *pre-defined helpers could cover most of the chaining needs*: participants used three times as many pre-defined helpers (13 in total) as customized JS nodes (4). One author further codified all the LLM nodes according to the primitive LLM operations previously identified [17], and found that out of the 27 LLM nodes, 7 were for categorizing inputs, 13 for sourcing information from the LLM, and 7 for re-organizing the input. This variety in utilization may have resulted from the PromptChainer's example galleries. For example, P4 successfully created their own

---

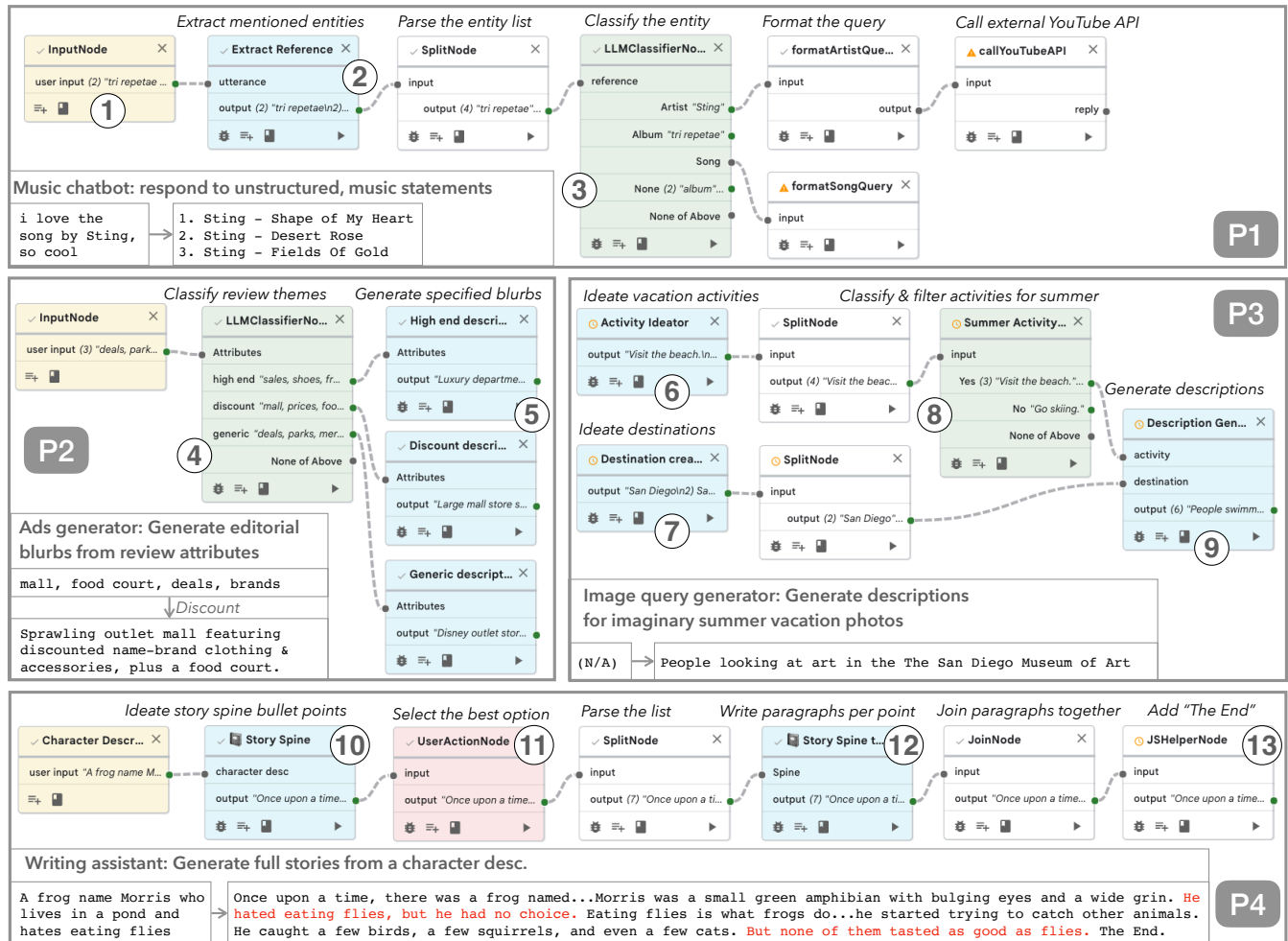[3]We used a non-dialog version of the model.

Figure 5: Four different chains built by user study participants. P1 and P2's chains used parallel branching logic , whereas P3 and P4's chains depict iterative content processing. The full details are in Figure 6, Appendix A.

LLM classifier by forking a simple default example, even though they were less familiar with prompting.

Chain debugging. When they completed constructing their chains, all participants ran the chain end-to-end (Figure 4C_2) as an "initial debugging strategy" (P1 and P4). Afterwards, they usually attributed chain failure to particular LLM nodes (P1: *"easy to pinpoint unexpected outputs from the data preview"*), and performed local debugging. P1 appreciated the *breakpoint functionality* (Figure 4C_3), as they did not need to take the chain apart in order to debug one of the nodes; P3, on the other hand, relied on the independent *testing block* (Figure 4c_1) when debugging the Description Generator ⑨, as a way to avoid expensive executions on multiple inputs coming from prior nodes.

Interestingly, most participants made some non-trivial refinements to their pre-built prompts *in the interface*, even though they had spent a fair amount of time doing prompt engineering before the study. We hypothesize that being able to observe the interaction effects *between* nodes affected their understanding and expectations of each local node. For example, when constructing the story creation chain, P4 wanted to add a final ending, "The End", to the

generated story. They first tried to always generate "The End" as the final bulletpoint in the story outline ("Story Spine") in ⑩, but realized that this would cause paragraph generator ⑫ to produce a paragraph repeating "The End The End The End." They therefore removed this generation from ⑩, and instead (with some help from the study facilitator) made a final JavaScript helper node ⑬ for appending the text "The End". This suggests that PromptChainer can help users *discover* errors, though future research is needed in supporting them to *resolve* identified problems through alternative solutions.

Q: What are remaining challenges in chain authoring?
A: Ensuring coherence between interdependent sub-tasks; tracking chains with complex logic.

Chains with interdependent parallel tasks can lead to decreased coherence. Because P4's story writing chain independently generated a paragraph for each point in the outline, the final essay lacked coherence: though the first several sentences followed the input description ("Morris...hates eating flies"), the final sentence instead hinted that Morris *likes* flies ("none of them tasted

as good as flies"). One pilot study user faced a similar challenge, and created another input node to manually track previous outputs. In the future, it may be worthwhile to further investigate methods that consider inter-dependency between parallel sub-tasks [15].

Chains that involve more complex decomposition can be overwhelming to track. In P1's music chatbot chain, the extractor node ② produces *a list of* candidate entities per input. Thus, it became unclear how the entities fed into the classifier ③ mapped to the original input node ①. We hope to enhance the tracing capabilities of PromptChainer. For example, future work could enable customized chain grouping: Instead of running one or all of the nodes, participants can explicitly select a subset to run. We may also add execution visualizations (*e.g.,* taking inspiration from Python Tutor[4]), to highlight the mapping from the original input all the way to the final output.

4.3 Discussion and Limitations

Because participants were asked to pre-create some LLM prompts for their desired sub-tasks prior to the study, this may have unintentionally led to participants feeling invested in their prompts and their particular chain decomposition, making them less inclined to consider other chain structures or scrap the prompts they had already created. Yet, prior work in prototyping indicates that concurrently considering multiple alternatives (*e.g.,* parallel prototyping [6]) can lead to better outcomes. Thus, future work could explore how to encourage low-fi prototyping of *multiple* possible chains: in other words, how can users create half-baked prompts for each step, such that the feasibility of an entire chain can be rapidly tested, without investing too much time designing each prompt? For example, PromptChainer could perhaps encourage users to start with only one or two examples in a few-shot prompt, or start with a very simple zero-shot prompt (even if they don't initially perform reliably) to reduce initial time invested in each prompt.

Given time constraints in the study, users may have also picked tasks that were naturally easy to decompose. In the future, it would be worthwhile to explore task decomposition strategies for even larger and more complex tasks. For example, PromptChainer could help encourage users to further decompose a node in the chain into more nodes, if extensive prompting efforts appear unsuccessful.

5 CONCLUSION

We identified three unique challenges for LLM chain authoring, brought on by the highly versatile and open-ended capabilities of LLMs. We designed PromptChainer, and found that it helped users transform intermediate LLM output, as well as debug the chain when LLM steps had interacting effects. Our study also revealed interesting future directions, including supporting more complex chains, as well as more explicitly supporting "half-baked" chain construction, so that users can easily sketch out a chain structure without investing too much time prompting upfront.

---

[4] https://pythontutor.com/

REFERENCES

[1] Gregor Betz, Kyle Richardson, and Christian Voigt. 2021. Thinking Aloud: Dynamic Context Generation Improves Zero-Shot Reasoning Performance of GPT-2. *ArXiv preprint* abs/2103.13033 (2021). https://arxiv.org/abs/2103.13033

[2] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Kohd, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. 2021. On the Opportunities and Risks of Foundation Models. arXiv:2108.07258 [cs.LG]

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[4] Margaret Burnett. 2010. End-user software engineering and why it matters. *Journal of Organizational and End User Computing (JOEUC)* 22, 1 (2010), 1–22.

[5] Margaret M. Burnett, Curtis R. Cook, and Gregg Rothermel. 2004. End-user software engineering. *Commun. ACM* 47 (2004), 53 – 58.

[6] Steven P Dow, Alana Glassco, Jonathan Kass, Melissa Schwarz, Daniel L Schwartz, and Scott R Klemmer. 2010. Parallel prototyping leads to better design results, more divergence, and increased self-efficacy. *ACM Transactions on Computer-Human Interaction (TOCHI)* 17, 4 (2010), 1–24.

[7] Baocheng Geng, Qunwei Li, and Pramod K Varshney. 2018. Decision tree design for classification in crowdsourcing systems. In *2018 52nd Asilomar Conference on Signals, Systems, and Computers*. IEEE, 859–863.

[8] Ellen Jiang, Kristen Olson, Edwin Toh, Alejandra Molina, Aaron Donsbach, Michael Terry, and Carrie J. Cai. 2022. Prompt-based Prototyping with Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*.

[9] Aniket Kittur, Boris Smus, Susheel Khamkar, and Robert E. Kraut. 2011. CrowdForge: Crowdsourcing Complex Work. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology* (Santa Barbara, California, USA) *(UIST '11)*. Association for Computing Machinery, New York, NY, USA, 43–52. https://doi.org/10.1145/2047196.2047202

[10] Opher Lieber, Or Sharir, Barak Lenz, and Yoav Shoham. 2021. *Jurassic-1: Technical Details And Evaluation*. Technical Report. AI21 Labs.

[11] 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* abs/2101.06804 (2021). https://arxiv.org/abs/2101.06804

[12] 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* abs/2104.08786 (2021). https://arxiv.org/abs/2104.08786

[13] Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. 2021. Cross-Task Generalization via Natural Language Crowdsourcing Instructions. *ArXiv preprint* abs/2104.08773 (2021). https://arxiv.org/abs/2104.08773

[14] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. 2014. Machine Learning: The High Interest Credit Card of Technical Debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*.

[15] Ben Swanson, Kory Mathewson, Ben Pietrzak, Sherol Chen, and Monica Dinalescu. 2021. Story Centaur: Large Language Model Few Shot Learning as a Creative Writing Tool. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*. Association for Computational Linguistics, Online, 244–256. https://aclanthology.org/2021.eacl-demos.29

[16] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. 2022. LaMDA: Language Models for Dialog Applications. *ArXiv preprint* abs/2201.08239 (2022). https://arxiv.org/abs/2201.08239

[17] Tongshuang Wu, Michael Terry, and Carrie J Cai. 2022. AI Chains: Transparent and Controllable Human-AI Interaction by Chaining Large Language Model Prompts. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3491102.3517582

[18] Qian Yang, Aaron Steinfeld, Carolyn Rosé, and John Zimmerman. 2020. *Re-Examining Whether, Why, and How Human-AI Interaction Is Uniquely Difficult to Design*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3313831.3376301

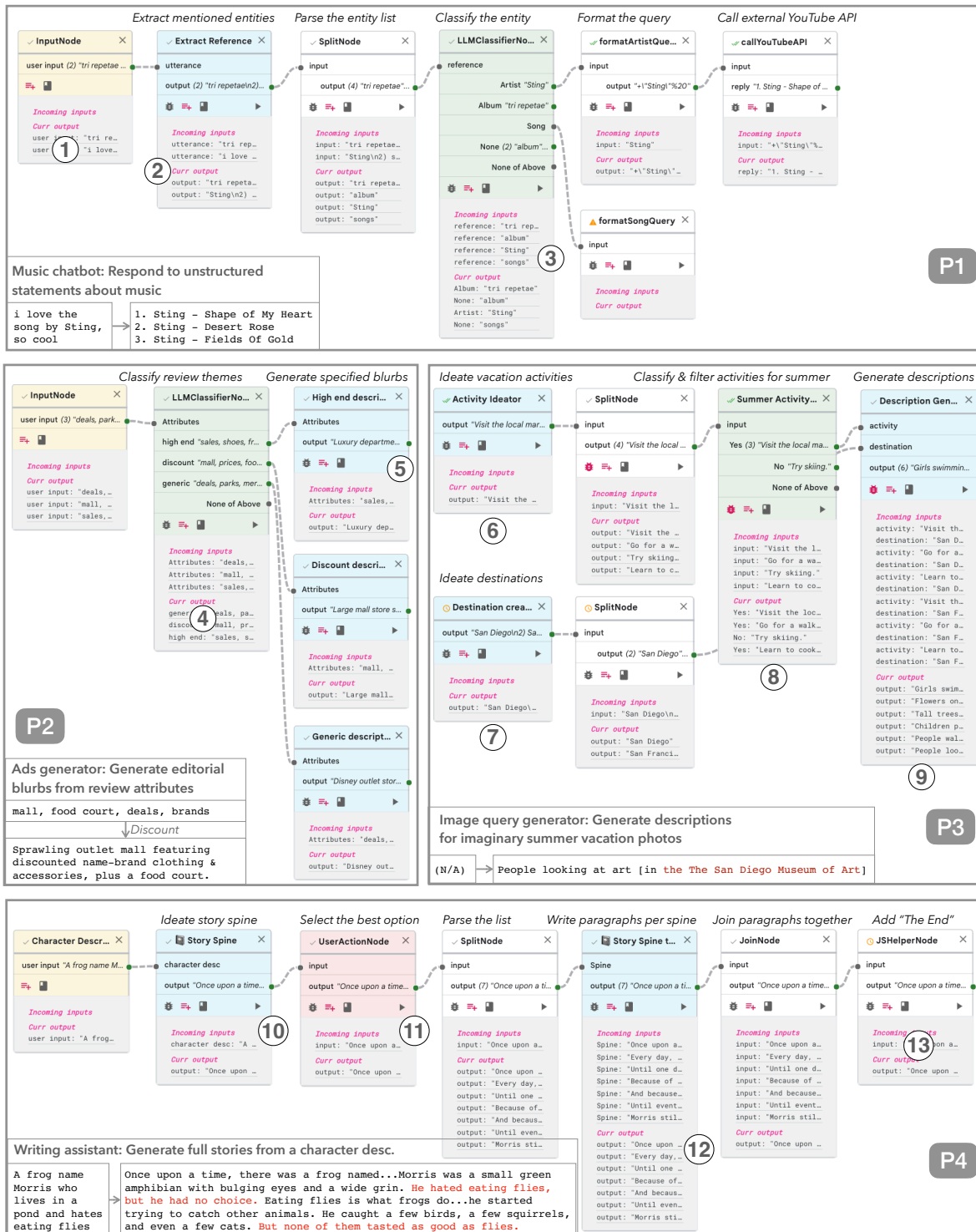## A SAMPLE CHAINS FROM PILOT USERS

Figure 6: The full details of user study chains.

Figure 7: The chains built by pilot users.