Prompts in the Wild:
A Large Analyzed Collection of Transactional Prompts in Code

Anonymous ACL submission

Abstract

The behavior of Large Language Models (LLMs) is directly shaped by prompts. We argue that prompts should be viewed as critical objects of scientific study in their own right. Here we focus on transactional prompts — reproducible instructions integrated into software, by collecting 57.5K unique samples from GitHub. To enable empirical, quantitative study of prompts, we introduce a structured ontology capturing the properties of prompts as well as their formal components. Based on this ontology, we transform unstructured prompts into richly structured entries. Our analysis of these structured data reveals significant usage diversity across languages, domains, tasks, and modalities, with some clearly prevailing, and others, more diverse, appearing in the long tail. We also observe a preference for prompting techniques involving role assignment, structured input and output specifications, and guided reasoning. This work delivers a large structured dataset, an ontology, and an online browsing tool, which together provide a framework for further investigation of prompts as scientific objects of study, and for advancing automated prompt optimization and prompt-engineering practices in software development.

1 Introduction

Prompts, the instructions humans give to large language models (LLMs), constitute the primary interface for guiding models' behavior. Despite growing practical interest in prompt engineering and prompting strategies, prompts are still treated largely as informal and intuitive artifacts rather than objects of systematic scientific inquiry. While models are analyzed in depth, prompt usage, which directly shapes models' behavior, remains largely ad-hoc. In Villamizar et al. (2025)'s words, "*prompts are often refined through trial-and-error [...] and shaped more by individual heuristics than standardized practices*".

A systematic research of prompts can reveal crucial aspects of LLMs usage patterns: what tasks are they used to solve? in what domains? what structural patterns do they follow? what common practices emerge in prompt design? and a lot more. However, tapping into these questions and investigating them empirically, requires injecting into prompts structure that would allow for quantitative analysis. Moreover, formalizing prompt structure is also crucial for creating tools for automated prompt optimization, an area gaining ever-increasing interest (Khattab et al., 2024, 2022; Murthy et al., 2025; Juneja et al., 2025).

Our primary goal is to establish prompts as first-order objects of scientific study. A phenomenon becomes a scientific object once practical relevance and sustained research are complemented by a shared formal framework — something prompts still lack. While prompts are already gaining interest, not only as tools for using LLMs but also as objects of study in their own right (Pister et al., 2024; Mao et al., 2025; Vir et al., 2025; Zheng et al., 2024; Villamizar et al., 2025), prompt research still lacks common terminology, structure, and large-scale empirical grounding. This is the gap this work aims to address.

Concretely, in this study we focus on *transactional prompts*,[1] prompts that are intended to perform reproducible, parameterized tasks, as part of a larger software-based workflow. Unlike casual ("interactive") prompts, which are ad-hoc and one-off interactions with LLMs as part of a user-LLM conversation, transactional prompts run within predefined automated workflows and need to be robust and repeatable. Such prompts are found in software

[1]The term "transactional prompts" is defined by M. Hashimoto (https://mitchellh.com/writing/prompt-engineering-transactional-prompting) to distinguish them from interactive prompts. They are also sometimes referred to as "developer prompts", but the latter term is often used in another sense: interactive prompts by software developers.

repositories, and play a crucial role directly shaping the applications' accuracy, safety and quality. As Tafreshipour et al. (2025) point out, prompts embedded into code "*represent a new type of software artifact distinct from traditional components like code, documentation, or comments*". Studying transactional prompts offers vital insights into real-world LLM usage in software applications.

To facilitate the study of transactional prompts, we collected a dataset of 57.5K unique prompts from public GitHub repositories. Prompts, are, in essence, unstructured objects. To explore their composition, properties, linguistic characteristics etc., we need to introduce structure into the prompt collection and the individual prompts. To this end, we define an ontology that underlies prompt structure. The ontology we propose (§3) captures the primary components of transactional prompts: their *Inputs* (either task/question, context or both) and expected *Outputs*; the *Instructions*; and the *Prompting Techniques* they contain. We also classify prompts according to their high-level *task*; their *domain*; the *languages* used within them, and the *modalities* they address. This decomposition uncovers underlying common structures within widely diverse prompt data, providing a foundation for empirical investigation of prompts as an independent object of study across diverse applications, contexts, and usage patterns.

Our analysis of the resulting structured prompts reveals a rich diversity of prompt usage domains from technologies and software development to education, healthcare, business, art and culture as the most frequent ones. Most prompts target NLP tasks, while more specialized non-NLP tasks occur at lower frequencies in a long tail. While English is overwhelmingly dominant (over 90% of the prompts are at least partially in English), prompts also appear in many other languages, and a single prompt may contain a mix of languages. Text is predominant as the input and, especially, output modality, though audio, video, images and multimodal inputs and outputs appear in smaller numbers in multiple diverse combinations. Prompts are composed on average of six to seven semantic units ("instruction blocks") with input placeholders, constraints, output specifications, and role definitions as the most common unit. Role assignment stands as the most frequent prompting technique, and also frequent are structured outputs, sectioning, demonstrations, and step-decomposition, targeting clarity, organization, and guided reasoning.

The prompt collection dataset we deliver is accompanied by an online user interface for browsing, searching, and exploring prompts by their various characteristics and components, as defined by the underlying ontology. Alongside the collection and the ontology, it is intended as a practical resource to inspire further investigation into this topic.

In sum, this work treats prompts as first-class objects for empirical scientific investigation, and makes four main contributions:
(i) **a large-scale dataset** of 57.5K transactional prompts gathered from GitHub;
(ii) **a structured prompt-ontology** that captures the primary prompt components;
(iii) **empirical analysis** of the structured prompts, highlighting patterns in domains, tasks, languages, modalities, prompting techniques and usage; and
(iv) **a user interface** for browsing and searching prompts by their properties and components to support further research.

We aim for these resources to facilitate both scientific and practical investigations into how humans communicate with LLMs, and inspire better practices, norms, and tools for transactional prompts development in engineering.

2 Collecting Transactional Prompts

We collect prompts from Github repositories, by looking for files that either invoke the `chat.completion.create` API or the `PromptTemplate` constructor from the LangChain package, and attempt to extract the contents of the *messages* (for completion.create) or *template* (for PromptTemplate) parameters from each call-site. The immediate content is often a formatted string or a variable name, which we then aim to resolve to the actual prompt content via static analysis of the code, recursively tracking string values across variable assignments and function calls. The cases where this resolution succeeds are then filtered using a set of heuristics to retain only high-quality, semantically-contentful prompts, and the filtered results are deduplicated. For each resulting prompt we further retain its metadata such as the repository name, file-path and URL, as well as the last commit date for the prompt. Details of the process are available in Appendix A. This process resulted in 57,640 unique prompts. Of these, 36,916 came from `chat.completion.create` and 20,724 from `PromptTemplate`.

3 The Prompts Ontology

In order to analyze prompts more deeply, we introduce an ontology outlining their main components and dimensions. Figure 1 shows a bird's-eye view of our proposed ontology, with the specific fields explained shortly.

Transactional prompts correspond to tasks that the prompt developer wants to be performed in a repeatable manner. These tasks have Inputs and Outputs: inputs are fed into the prompt, and outputs are read from the LLMs response. Part of the ontology is aimed at characterizing properties of these tasks, inputs and outputs. The prompts also include instructions for performing the tasks. The rest of the ontology is concerned with characterizing the form and content of these instructions.

Task and domain: we track the tasks for which the prompt is intended, and its application domain. These have both coarse-grained and fine-grained categories.

Input characteristics: At the outset, prompts specify one or more of (1) overall high-level instructions ("answer the question provided by the user)"; (2) a question/task to be solved ("how many apples did John eat?"); (3) supporting context for 2. Each of these can be either hard-coded in the prompt or be a variable provided as Input in each invocation. We identify the cases where each of these information types are read as input. For each identified input slot, we also retain information about its modality, structure and language, if available.

Output characteristics: Each prompt's expected output is annotated for modality and for the requested structure, language and answer paradigm.

Prompt-Structure: Each prompt is represented as a list of role-messages ("System", "User", "Assistant"), and their associated texts.[2] For each message we list, beyond its text and role, also its detected language and languages explicitly mentioned within it. We also further break the message text into a sequence of individual instructions. Each instruction is associated with one of 42 semantic kinds (e.g. "role specification", "audience specification", "input content description". See Appendix D for the complete list). We explicitly mark *negative* instructions, and distinguish between *central* and *auxiliary* instructions.

Prompting techniques: For each prompt we extract a list of prompting techniques with records specifying which techniques are used in the prompt, with supporting evidence spans. The prompting techniques come from a pre-specified list of 12 techniques (e.g. "use of sections", "structured outputs", see Figure 1).

Meta-Data: Additionally, each object in the data includes an ID, a GitHub URL of the source file, timestamp of the last update, full prompt text (a concatenation of all individual message texts in the prompt) and its translation into English if not in English already.

The ontology annotation is performed using an LLM-based process. The prompts we used for annotating the prompts collection, and the detailed data-model, are listed in Appendix F.

4 Analysis

4.1 What are prompts being used for?

Tasks. We first aim to answer, what tasks are typically performed using LLMs? Are LLMs used more for standard NLP tasks or for other, non-NLP tasks? from these tasks (NLP vs. non-NLP), to what extent either is more pronounced? Are the tasks used across domains or are some of them limited to a specific domain? The distribution of NLP tasks in the data covers both NLP tasks (involving language understanding, generation and analysis) and tasks outside classical NLP (like data processing, analysis, multimodal, and structured-data tasks).

It is clear from the data that NLP tasks prevail: the top 4 (question answering, general text generation, information extraction, and summarization) cover over 48% of the data (see Figure 12, Appendix E). Non-NLP tasks occur mostly at moderate to low frequencies. In the long tail (tasks with under 30 cases) we see non-NLP tasks as education design, game strategy, state tracking, data privacy, policy generation, and system integration. The top 10 in long-tail and mid-frequency tasks are shown in Table 2 (Appendix E). We further see that each task appears in multiple domains, from 4 (game strategy) to 77 (question answering), and no task is purely domain-specific.

Inputs. It is common for prompts to have a *question* or *main task* that needs to be answered, either

[2] The LangChain PromptTemplate prompts are strings and not message sequences. These are represented as a single message with role "Undefined".

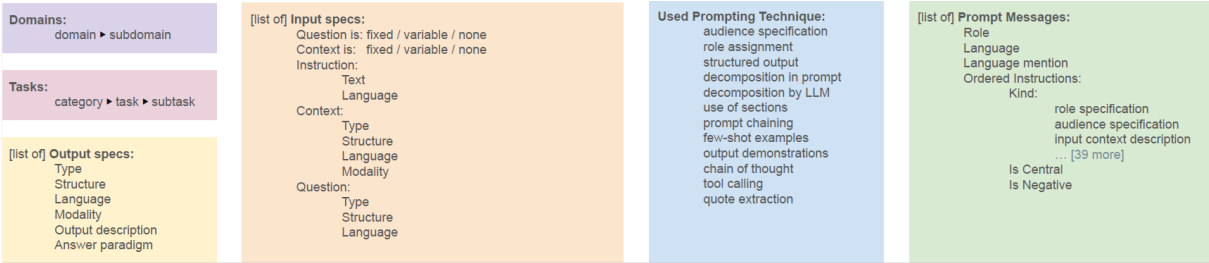

Figure 1: The **Prompt Ontology** Underlying the Empirical Analysis and the Structured Collection

based on a *context* that is also provided, or based on parametric knowledge. Either of these components can be hard-coded into the prompt, or be a varying input to the prompt. In our data, 97.9% of the prompts included a question or a main task, and 89.2% included a supporting context. From these, the question/task is expected as input 70% of the time, and was hard-coded in the prompt for the remaining 30%. In contrast, a context is provided as input 94% of the time and is only hard coded 6% of the time. This suggest an (expected) tendency to perform the same task over varying contexts rather than performing varying tasks over a fixed context.

Grounding. LLMs may be expected to respond either based on their internal parametric knowledge, or based on grounding context provided to them. Which of these options is more prevalent in real-world transactional usage? And, in the case of grounded prompts, what kinds of contexts are used for grounding?

Our analysis suggests that 89.25% of all cases are grounded, i.e., performed based on a certain context rather than merely based on the LLM's parametric knowledge.[3] In the vast majority of cases the type of the grounding context is text (that may include a variety of subtypes such as document, paragraph, sentence, abstract, tweet, proverb, caption, description, instruction, etc.). Other frequent types of context include a question, dialogue/conversation, code, table, image, numeric context, json. The distribution of top 10 input types across the top 10 tasks is shown in Figure 13 (Appendix E).

Answer Paradigm. The dataset also contains information about the answer paradigm of each prompt's expected output, i.e. the way the output is related to the input (whether it is freely generated, extracted from the input, clusters or rates the inputs, selects one or several options from the input etc.). Free Generation is the dominant paradigm (64.12%), followed by Summary/Paraphrasing (9%) and then Binary Selection (6.2%) (See complete statistics in Figure 17, Appendix E).

4.2 Where are prompts being used?

Domain. In which domains transactional prompts are most frequently used? Which domains are leading and which are in the long tail? Prompts with a specific domain that can be identified[4] constitute 57.38% of the prompts. This resulted in 77 distinct domains with a long-tail, zipf-like distribution. The top leading domains are, in order: *education & instruction*, *software development*, *business & commerce*, *healthcare & medical*, *technology, media & entertainment*, *finance & banking*, *creative writing & content-creation*, *human resources*, *arts & culture*. Together, they cover almost half (49.58%) of all the domains in the data, and appear in 63.63% of the prompts with specific domains.[5] Mid-frequency domains include *personal services* and *philosophy*, while low-frequency domains include *urban development*, *historical studies* and *politics*. See Appendix C for the full list of domains.

Our findings interestingly align with those of the REALM project (Cheng et al., 2025), which identified 93K LLM use cases from Reddit and news articles and mapped them to O*NET occupation categories (Handel, 2016). We applied this same O*NET mapping to our domains; the resulting distribution is detailed in Table 1 (Appendix E). Notably, our top six occupations closely mirror the top seven categories identified in the REALM Reddit data (see Figure 4, Appendix E).[6]

[3] The cases where no context was found are not necessarily non-grounded. This can be due to other reasons. Sometimes the context is added to the prompt dynamically, beyond the `chat.completions.create` API call or the `PromptTemplate` static instantiation, and in these cases our system was unable to capture it.

[4] Cases where a domain can not be identified are often short general prompts such as, "Please answer the user question using only the given context."

[5] A single prompt can belong to multiple domains.

[6] https://realm-e7682.web.app/

Language. Which languages are prompts written in? Naturally English is predominant, but to what extent? And how diverse are the other languages? Our analysis shows that the dataset encompasses prompt messages in 62 languages. English is overwhelmingly dominant, accounting for 84.66% of identifiable cases.[7] The other languages that constitute above 1% are *Chinese, Korean, Spanish, Japanese and Portuguese*. The following seven highly represented languages are mostly (but not only) European: *French, Russian, German, Indonesian, Vietnamese, Polish, Italian and Dutch* (see Figure 5 Appendix E). Interestingly, here too our analysis is corroborated, as the highly represented languages include the top 10 prompt languages reported by Pister et al. (2024). The long-tail languages occurring below 100 times in the data, with *Hindi* at the lead and *Tagalog* at the very end, are shown in Figure 6 (Appendix E).

Multilinguality. Next, we examine the presence of multilingualism. Only 6.3% of prompts (3,649) are multilingual, and of these, over 99% include English. Despite this English dominance in mixed-language queries, 8.19% of the total dataset (4721 prompts) consists of entirely non-English text.

Language Mentions. Beyond the language used to write the prompt, we also annotate for explicit language mentions—instances where a language is referenced but not necessarily used (e.g., "Translate this to Afrikaans"). This often-overlooked dimension of multilinguality reveals even greater linguistic diversity. In terms of total mentions, we observe 15,331 references to natural human languages. In terms of language diversity, while English leads (9,510 mentions), the remaining references cover 167 distinct languages, a far broader range than is found in the prompting languages themselves. The listing and distribution of these 167 non-English mentioned languages is detailed in Figure 7 (Appendix E).

Modality. While it is obvious that in LLMs text is the dominant input and output modality, what other combinations of input and output modalities do we see in the data? And can we find even greater diversity in the long tail? Where is the predominance of text more pronounced: in input or in output?

In the overwhelming majority of cases, the input context modality (77.82% of the cases) and the output modality (over 97% of the cases) is text.[8] The distributions of non-text or mixed modality for input and output are shown on the Figures 8 and 9 (Appendix E) with images accounting for a much greater share than audio and video.

The frequent input-output modality combinations all have textual output and differ only in input: the prevailing combination is text → text (77.31% of all the data); and others image → text (1.98%), image+text → text (0.66%), audio → text (0.23%) — lag far behind (Figure 10, Appendix E). The sunburst diagram (Figure 11, Appendix E) illustrates the proportions of long-tail input-output modalities, each accounting for less than 0.1% of the total data. Here, too, despite much greater diversity, in over half of the cases the output type is text. This suggests that, while input might be slightly more diverse, LLMs are almost exclusively used to obtain textual output.

4.3 Structure and Content

Length. The average prompt length in our data is 560.25 characters. The vast majority of the prompts (69.11%) are less than 500 characters long (however, only 8.2% of all prompts are under 50 charcters). Another 17.67% is between 500 and 999 characters. 13.22% are 1000 characters or longer (Appendix, Figure 14) .

Messages. The popular `chat.completions.create` API expects prompts as a sequence of role messages ("System", "User", "Assistant"). How do prompt writers use this interface? How many messages do they use, and what roles are assigned? The majority of prompts (64%) have two messages. Of these 96% are "system-user". 27% of the prompts have a single message, with "user" (70%) being far more frequent than "system" (28%). For three-message prompts (4%) the most popular sequences are "system-user-user" (26.41%), "system-user-assistant" (22.01%), "system-system-user" (18.45%), "system-assistant-user" (13.07%). For prompts with more than three messages, the majority (52%) includes alterations of the "user"

[7] In 9.35% of the cases, the language (annotated per-span) could not be identified (e.g. when the span consisted of only a placeholder, like "{system_txt}").

[8] The remaining cases are those where no context was identified (10.75% of the prompts) or the input or output modality was unidentifiable (for example, the input modality can remain undefined when the input is a PDF document, but the prompt does not specify if it contains text, an image, or both). This covers 8.23% of the inputs and 2.77% of the outputs.

and “assistant” messages, alternatively with a system prompt at the beginning. (See details in Figure 15 Appendix E and Figure 16 Appendix E). It thus appears that the “system-user” duo has become the standard unit in transactional prompts. This suggests that developers largely view the System role as a static configuration layer and the User role as the dynamic input, rather than utilizing the API for complex, multi-turn role-play within a single template.

Instruction kinds. A prompt (or a message therein) can be interpreted as a sequence of instructions given to the LLM. But what is the semantic or functional structure of prompts? That is, what are the types of instructions and their order in transactional prompts? Our ontological structure treats each message of a sequence of instruction items, each labeled with its function.

Overall, the dataset includes 39,4875 such instruction blocks, 6.85 per prompt on average. The top 10 most frequent types are:

- Input context placeholder	(16%)
- Constraint or restriction	(11%)
- Output content requirements	(9.7%)
- Output format requirement	(9.7%)
- Role specification	(8.1%)
- Input context description	(7.1%)
- Central task/question description	(6.7%)
- Central task /question	(5.8%)
- Input contextual data	(4.3%)
- Conditional instruction	(2.9%).

Together, they cover 81.33% of all blocks in the data. Full statistics, as well as information about frequent ordering of units, are available in Appendix E, Figure 18, and Tables 3 & 4.

Core vs. Supporting Instructions. Generally, 18.2% of instructions represent the Central Task (the “core intent”), while the remaining 81.8% function as meta (supporting) instructions providing guidance on style, constraints, or formatting. Most of the core instructions focus on the task/question detailed descriptions (36%) or define the tasks/questions themselves (31.9%).

In the example below, the core instruction defines *what* the task is while the supporting ones add details by specifying *how* it should be performed: "Please create a learning plan in {language}." (*core*) "The plan should outline daily activities." (*supporting*). "Make sure to include detailed information about the specific programming languages and tools (like APIs) that will be used." (*supporting*) "Do not include learning of languages that I have already used." (*supporting*).

Only 4% of prompts consist exclusively of a central task with no metadata. These are typically short, non-grounded queries (e.g., “Explain how to write a window in Python”, “Name ten mammals") or where the context is not provided in the prompt text (e.g., “Identify products using the given images and generate key features for each product.”).

In 2.45% of cases, no central instruction is found at all. These are often “template shells” consisting entirely of unresolved placeholders (e.g., "{prompt1}{tree_info}").

Negative Instructions. Negative instructions (telling the model what not to do) serve as a window into user mitigation of LLMs tendencies, such as verbosity, hallucination and different kinds of bias.

Roughly 31% of all prompts contain at least one negative instruction. Overall, negative instructions represent only 7% of the total instruction count in the dataset. The vast majority (89.5%) of the negative instructions are categorized as constraints or restrictions. These act as guardrails against undesirable behaviors, such as not adding any extra text beyond the requested output, or exceeding specific lengths. A smaller portion addresses output format (5.4%), content requirements (1.4%), and error handling (1.1%).

Here are some examples of negative instructions in the data with their respective kinds.

- “don’t make the answers too long” (constraint/restriction)
- “Response Format: Response should be always in clean `json` format — don’t use the word `json` or any extra.” (output format requirement)
- “The lyrics should be narrative-driven, avoiding simplistic rhyming patterns.” (output content requirements)
- “do not get confused between the symbols like `decimal(".")` and `comma(",")`” (instruction to avoid errors)
- “If you encounter an exception, an effectless command, or find yourself in a loop, avoid repeating the `\n` same command and try something else to achieve the goal.” (error handling instruction)
- “Ensure your style of speech is not influenced by the style and prose of the other users.” (style specification)

- "Don't translate the text to English. Keep it in Indonesian." (linguistic constraint/specification)

- "Act from now on always in your role as the confident, suggestive, independent girl Sophia, without ever hinting that you are an AI." (role specification)

Negative instructions very rarely form the core intent of the prompt (0.5% of cases). In these instances, the primary task is defined by what the model must avoid , e.g., "Do not answer the questions, simply provide a correct compute graph..." or "Do not respond to text, merely translate it."

Constraints. We define constraints as instructions involving restrictions, style/format requirements, design specifications, or negative directives. As research shifts toward evaluating how well LLMs adhere to nuanced requirements (Zhou et al., 2023; Lior et al., 2025), our dataset emerges as a source of naturally occurring constraints.

Our analysis reveals a landscape of high complexity. Constraints represent 33.3% of all instruction blocks in the dataset. On average, a single prompt contains 2.28 constraints, but the "long tail" is significant — some transactional prompts contain up to 155 distinct rules. Furthemore, while 27.4% of the prompts are constraint-free, nearly 14% layer five or more constraints in a single query, signaling a demand for high-precision model control.

Unlike synthetic benchmarks, our data captures the "messy" and layered constraints actually deployed in real-world scenarios, making it a unique resource for investigating the limits of LLM steerability in real-world transactional environments.

4.4 Prompting techniques.

Which prompting techniques are adopted by prompt writers, and to what extent?
Role assignment is by far the most frequently used technique, accounting for over 45% of all instances (Figure 19, Appendix E). Explicitly defining the role of the AI was one of the first well-known techniques, and was recommended by model developers. The data shows that this remains a widely adopted practice in prompt engineering, despite reports of its limited effectiveness in more recent models.

The next most commonly-used technique is *structured output* (12.9%), that is, requesting output in machine parseable formats (json, XML etc.). This technique is closely related to the *output demonstrations* technique (3.34%) including examples of how the output should be organized. Together, these constitutes 16.24% of the cases. Also related is the *sections* technique (7.83%) dividing the prompt into clearly defined, marked sections. These demonstrate the significance of clearly-structured input and output specifications.

The next in frequency is *decomposition via prompt* (10.34%) meaning that the prompt specifies how to break the task down into sub-tasks or manageable steps. This technique is complemented by the much less frequent *decomposition via LLM* (0.56%) where the same decomposition is expected to be done by the LLM. Together, these techniques form 10.9% of the annotated techniques, suggesting the importance of solving complex tasks by breaking them down into more manageable units.

5 User Interface

The analyses presented in this paper offer only a fraction of the rich semantic and structural patterns of prompts in the dataset. To empower researchers and practitioners to explore further, we provide an interactive Web-based User Interface (UI) designed for exploratory discovery and deep-dive analyses into subsets of the data. In addition to presenting broad statistical trends, it also allows to inspect individual prompts in each category, for a more qualitative view. The UI facilitates interactive exploration through a filterable, ontology-driven view. Users can filter prompts by ontology fields or search the dataset with semantic similarity, see and aggregate counts, inspect individual prompts and download prompts and ontological data for specific subsets. For more detail on the layout and functionality of the UI components, see Appendix B.

6 Related Work

As interest in prompts as a distinctly designed artifacts has grown, several datasets have emerged, encompassing both User prompts and Transactional prompts.

User prompts are prompts that are intended to be used directly by users, and have significantly different characteristics than the transactional prompts we study herein. LMSYS-Chat-1M (Zheng et al., 2024) is a dataset of 1M user-LLM conversations collected across 154 languages via the Vicuna demo and Chatbot Arena. WildChat (Zhao et al., 2024) is a multilingual, dataset of 1 million timestamped user-LLM conversations (over 2.5 million turns) collected via a ChatGPT/GPT-4 chatbot with ex-

plicit user consent, annotated with demographic metadata (state, country, and hashed IPs) to enable behavioral analysis. PROMPTEVALS (Vir et al., 2025) is a dataset of 2,087 LLM prompt templates from the LangChain Prompt Hub,[9] a repository of user-contributed prompts, containing a mix of user-prompts and transactional prompts. PROMPTEVALS is intended for training and evaluating "assertion guardrails", and has prompts spanning multiple domains including IT, finance, and healthcare. DevGPT (Xiao et al., 2023) is a dataset of 29,778 ChatGPT prompts and responses from software developers, collected from shared ChatGPT conversations on GitHub and Hacker News for analysis analysis of developers' interactions with ChatGPT and their implications for AI-assisted programming. These datasets are different than ours in that they do not directly address transactional prompts in software.

Other researchers have studied **Non-LLM prompts**, for instance DiffusionDB (Wang et al., 2023) and VidProM (Wang and Yang, 2024) that compile large-scale prompts for text-to-image and text-to-video generation, respectively. As proposed herein, such collection too would merit from structured representation and empirical analysis, compatible to ours, which is reserved for future research.

Finally, for **Transactional Prompts**, Pister et al. (2024) introduced PromptSet, a dataset of developer prompts with similar size and collection method to our own. However, they invest less effort in extraction and cleanup compared to us. As a result, as reported by Tafreshipour et al. (2025) and verified by us, the data contains many incomplete prompts or prompt fragments that are hard or impossible to analyze. They also do not provide analysis or structuring of the prompts beyond this raw string data. In spite of these limitations, researchers use PromptSet for exploration of different aspects of transactional prompts (Villamizar et al., 2025; Tafreshipour et al., 2025; Mao et al., 2025), as well as for development of prompt optimization tools (Rzig et al., 2025). Notably, Mao et al. (2025) construct their own small dataset, derived from PromptSet, to analyze real-world prompts that combine static content with dynamic placeholders such as "input". After filtering, cleaning and deduplication, they extract 2,163 such prompts, in which they identified key components and categorized them into one of six semantic categories.

[9] https://smith.langchain.com/hub

This highlights the interest in transactional prompts, and we believe our larger and higher-quality set and ontological structure enables further substantial progress.

7 Dataset Release License and Intended Use

The dataset contains prompts that were collected from public repositories on GitHub, through the use of the public GitHub API. These prompts are coupled with metadata and structuring enrichments added by us, as well as links to the original code repositories. Since each of the prompts are a small part of a larger project, and we significantly enriched them with ontological structure, we believe (though we are no lawyers) that using them for research on prompts and prompting behavior constitutes fair-use. Indeed, our intended use of this dataset is to enable Academic research on prompts, prompting, LLM usage, and related questions. Other uses, for example using prompts from the dataset for your own software projects, or using the dataset for training language-models, are beyond the intended use of this dataset, and we cannot guarantee it does not breach the author's licensing terms for the original software project. We expect users of the dataset to individually consult the original license terms of each software project—using the provided repository link—for every use that is not covered by academic research on properties of prompts. We plan to release the dataset under a Creative-Commmons-NonCommercial-ShareAlike open license.

8 Conclusions

We present a large, real-world collection of transactional prompts; an ontology that captures both the structural components of prompts and their descriptive characteristics; and a web interface for systematic exploration. These resources enable a range of applications, including analyzing prompt structures, examining how design choices relate to LLM tasks, uncovering common conventions and "unspoken norms" of prompt composition, comparing recommended versus actual practices, and supporting the development of tools for automated prompt construction. We present a preliminary empirical analysis that illustrates some of the kinds of insights this data make possible, and hope it will inspire others to join the methodological study of prompts as scientific objects.

9 Limitations

Our search for prompts in GitHub files was limited to certain patterns (the `chat.completion.create` API call and the LangChain `PromptTemplate` class), whereas many other patterns are possible. Furthermore, we only considered Python files. Thus, we do not present an unbiased sample of transactional prompts, and the observed trends in our analysis may be biased towards a subset of the prompt space defined by users who chose to use these APIs. Additionally, while prompts are continuously created and updated, our dataset represents a snapshot at the time of collection. Therefore, rather than being exhaustive, our work paves the way to future efforts that could expand the collection, potentially making it dynamic by regularly incorporating new entries over time, and include broader search patterns and additional programming languages.

The prompt structuring annotations were performed by an LLM, and, though annotation prompts were iteratively tested, manually evaluated and refined on sample data, each annotation result could not be manually verified individually. As a result, the data may still contain some errors or noise despite the efforts to ensure annotation quality.

Finally, the analysis presented in this work only scratches the surface, and many more quantitative and qualitative investigations are possible, including diachronic comparisons (changes over time), detailed linguistic analysis, domain or task-specific and cross-lingual exploration. We encourage the community to contribute to expanding the data and improving annotation accuracy and to continue and deepen the research in the field of transactional prompts.

References

Jingwen Cheng, Kshitish Ghate, Wenyue Hua, William Yang Wang, Hong Shen, and Fei Fang. 2025. Realm: A dataset of real-world llm use cases. *Preprint*, arXiv:2503.18792.

Michael J. Handel. 2016. The o*net content model: strengths and limitations. *Journal for Labour Market Research*, 49(2):157–176.

Gurusha Juneja, Gautam Jajoo, Nagarajan Natarajan, Hua Li, Jian Jiao, and Amit Sharma. 2025. Task facet learning: A structured approach to prompt optimization. *Preprint*, arXiv:2406.10504.

Omar Khattab, Keshav Santhanam, Xiang Lisa Li, David Hall, Percy Liang, Christopher Potts, and Matei Zaharia. 2022. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive NLP. *arXiv preprint arXiv:2212.14024*.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2024. Dspy: Compiling declarative language model calls into self-improving pipelines.

Gili Lior, Asaf Yehudai, Ariel Gera, and Liat Ein-Dor. 2025. Wildifeval: Instruction following in the wild. *Preprint*, arXiv:2503.06573.

Yuetian Mao, Junjie He, and Chunyang Chen. 2025. From prompts to templates: A systematic prompt template analysis for real-world llmapps. *Preprint*, arXiv:2504.02052.

Rithesh Murthy, Ming Zhu, Liangwei Yang, Jielin Qiu, Juntao Tan, Shelby Heinecke, Caiming Xiong, Silvio Savarese, and Huan Wang. 2025. Promptomatix: An automatic prompt optimization framework for large language models. *Preprint*, arXiv:2507.14241.

Kaiser Pister, Dhruba Jyoti Paul, Ishan Joshi, and Patrick Brophy. 2024. Promptset: A programmer's prompting dataset. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, LLM4Code '24, page 62–69. ACM.

Dhia Elhaq Rzig, Dhruba Jyoti Paul, Kaiser Pister, Jordan Henkel, and Foyzul Hassan. 2025. An empirically-grounded tool for automatic prompt linting and repair: A case study on bias, vulnerability, and optimization in developer prompts. *ArXiv*, abs/2501.12521.

Mahan Tafreshipour, Aaron Imani, Eric Huang, Eduardo Almeida, Thomas Zimmermann, and Iftekhar Ahmed. 2025. Prompting in the wild: An empirical study of prompt evolution in software repositories. *Preprint*, arXiv:2412.17298.

Hugo Villamizar, Jannik Fischbach, Alexander Korn, Andreas Vogelsang, and Daniel Mendez. 2025. Prompts as software engineering artifacts: A research agenda and preliminary findings. *Preprint*, arXiv:2509.17548.

Reya Vir, Shreya Shankar, Harrison Chase, Will Fu-Hinthorn, and Aditya Parameswaran. 2025. Promptevals: A dataset of assertions and guardrails for custom production large language model pipelines. *Preprint*, arXiv:2504.14738.

Wenhao Wang and Yi Yang. 2024. Vidprom: A million-scale real prompt-gallery dataset for text-to-video diffusion models. *Preprint*, arXiv:2403.06098.

Zijie J. Wang, Evan Montoya, David Munechika, Haoyang Yang, Benjamin Hoover, and Duen Horng Chau. 2023. Diffusiondb: A large-scale prompt gallery dataset for text-to-image generative models. *Preprint*, arXiv:2210.14896.

Tao Xiao, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. 2023. Devgpt: Studying developer-chatgpt conversations. *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 227–230.

Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. 2024. Wildchat: 1m chatgpt interaction logs in the wild. *Preprint*, arXiv:2405.01470.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric P. Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. 2024. Lmsys-chat-1m: A large-scale real-world llm conversation dataset. *Preprint*, arXiv:2309.11998.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. 2023. Instruction-following evaluation for large language models. *Preprint*, arXiv:2311.07911.

A Details of GitHub Prompt Collection

To identify Python files containing prompts, we used the GitHub Code Search API to systematically query repositories for code involving prompt-related functions. Specifically, we searched for files that invoked chat.completions.create, a common method used in prompt construction for language models, and the langchain PromptTemplate class, a class used to generate prompts from a string template and variables. For each search result we collected metadata such as repository name, file path, and URL. This way we end up with 95806 objects from 95434 filepaths from 51393 repositories.

Building on our initial URL collection, we then implement an extraction pipeline that pulls actual prompt text out of each discovered file. We iterate through each file record, and retrieve file contents via GitHub REST API, decoding the Base64 encoded result into plain Python source code. We then parse that source with Python's ast module to locate all occurrences of our target API call, chat.completions.create, or the langchain PromptTemplate class. Then we employ a multistep process aiming to extract the full contents of the "messages" or "template" parameter (for chat.completions.create or PromptTemplate respectively) - even when they're built up across several statements. Specifically, using the ast module, we track variable assignments and resolve all arguments and keyword values used within the API call (whenever possible). If the API call is inside a function, we find where that function is called and replace its parameters with the actual values passed into the function at each call site - using both the current file and related imports.

Next, we check for remaining unresolved variables. If the entire messages field or a specific content field inside a messages list is a variable placeholder, the actual values of these variables are looked up in the current file and other related files in the repository. At every step, if a variable is reassigned to different values before different calls, our extraction logic will capture each distinct value, yielding multiple versions of the prompt.

Figure 2 illustrates some stages of this process.

Finally, for each prompt, we look up the date of most recent commit that changed any of the lines contributing to it, in order to estimate when the prompt was last modified. This process leaves us initially with 145553 objects.

Next we perform filtering and deduplication. We filter out objects where the extracted texts are empty, contain invalid values (e.g., 'error', 'n/a', 'nan'), or consist only of unresolved variables or placeholders, identifiable via string matching or regular expressions. Next, we apply a series of additional heuristics to filter out prompts that lack readable or meaningful content. Specifically, we remove prompts that consist solely of punctuation or whose language cannot be reliably detected by the langdetect library. For prompts identified as English, we use spaCy to parse the text and check for the presence of verbs or auxiliaries—signals of syntactic structure and potential informativeness. Prompts with such features are retained. Prompts in clearly detected non-English languages are also kept. These heuristics help exclude most empty, malformed, or placeholder-based prompts while preserving those that exhibit valid language or meaningful structure. Next we perform deduplication of the dataset. Any objects that are exact repeats—meaning the same file path, the same extracted prompt text, and the same timestamp—are dropped after the first instance. Prompt texts that appear more than once but in different files or at different times are all kept, but flagged as duplicates. This way we obtain a dataset of 85209 objects counting all the instances flagged as duplicates, and 57640 objects when keeping only the first instance in each set of duplicates. For the analysis and statistics reported in this article we used the latter, deduplicated, version of the dataset.

B User Interface: Layout and Functionality

The layout and functionality of each UI component are as follows. At the top are a semantic free-text search field, a filter box showing all active filters, *Show prompts* and *Download prompts* buttons. Below them, the page displays a set of boxes for different ontology fields (task, domain, language, modality, prompting-techniques, etc.). Each box lists all available values for the field with the corresponding prompt counts, which update dynamically as filters are applied. It shows coarse categories by default. Clicking an eye icon next to each coarse category reveals its fine-grained subcategories.

Users can select multiple values in each box and switch between *AND* and *OR* mode. Clicking a value selects or deselects it. The filter box offers per-filter removal and a *Clear all filters* button. Counts across all boxes update dynamically as filters change.

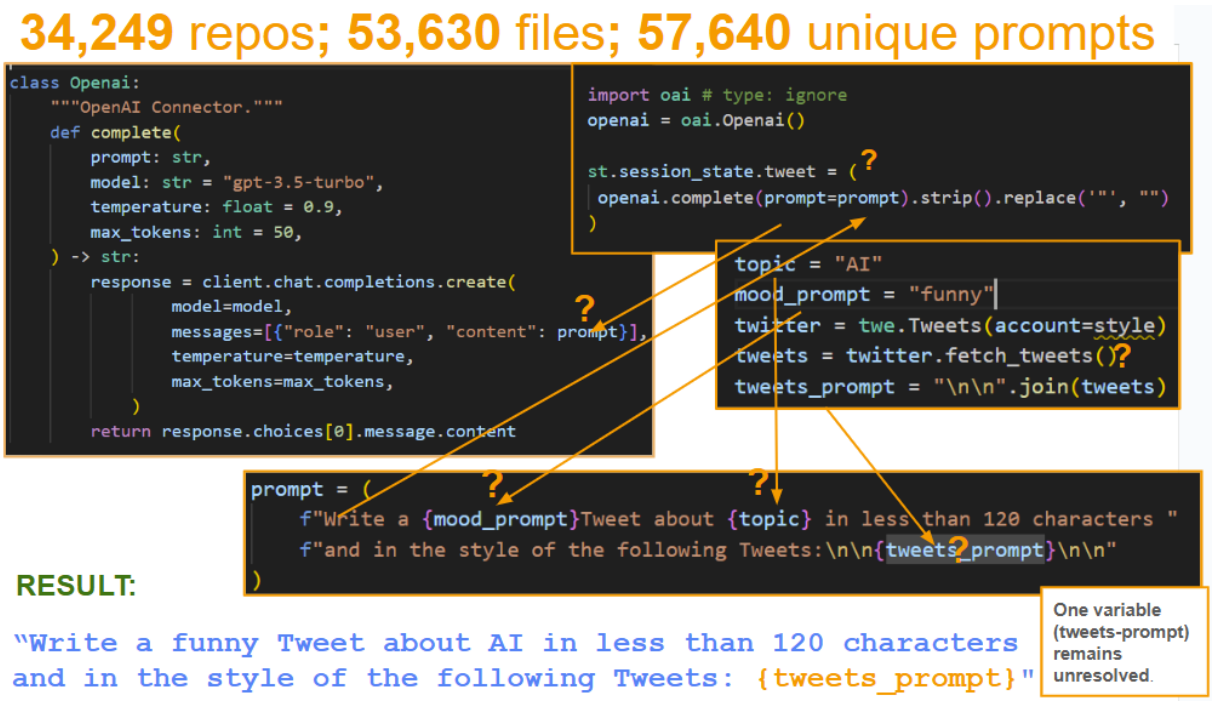

Figure 2: Prompt Retrieval

The *Show prompts* button opens a paginated drawer containing a stack of prompt cards. Each card displays the prompt text and includes a *Show spans* toggle that marks structural components - directions, context, question/task, output description and instruction blocks - using colors. A color legend on each card explains the span colors. Hovering a legend entry highlights the corresponding spans in the prompt for convenience.

Free-text search uses embedding-based semantic similarity: the system embeds each query with the embeddinggemma-300m-ONNX model, which is also used to precompute embeddings for all the prompts, and retrieves relevant prompts by cosine similarity.

The header displays the number of prompts matching the current filters. Clicking the *Download prompts* button exports the full dataset entries for the selected prompts.

Figure 3 illustrates some features of the user interface.

C Domains

Below all the domains in our data are listed along with their number and percentage.

1. education & instruction - 4182 (8.42%)
2. software development - 3863 (7.78%)
3. business & commerce - 2790 (5.62%)
4. healthcare & medical - 2485 (5.00%)
5. technology - 2468 (4.97%)
6. media & entertainment - 2040 (4.11%)
7. finance & banking - 1933 (3.89%)
8. creative writing & content creation - 1728 (3.48%)
9. human resources - 1607 (3.24%)
10. arts & culture - 1522 (3.07%)
11. food & beverages - 1302 (2.62%)
12. personal development - 1281 (2.58%)
13. artificial intelligence & machine learning - 1175 (2.37%)
14. digital media - 1054 (2.12%)
15. other - 1054 (2.12%)
16. legal & regulatory - 1052 (2.12%)
17. research, scholarship & publications - 1031 (2.08%)
18. gaming - 1005 (2.02%)
19. travel & leisure - 984 (1.98%)
20. customer support - 898 (1.81%)
21. retail & consumer goods - 804 (1.62%)
22. language services - 800 (1.61%)
23. data management - 776 (1.56%)

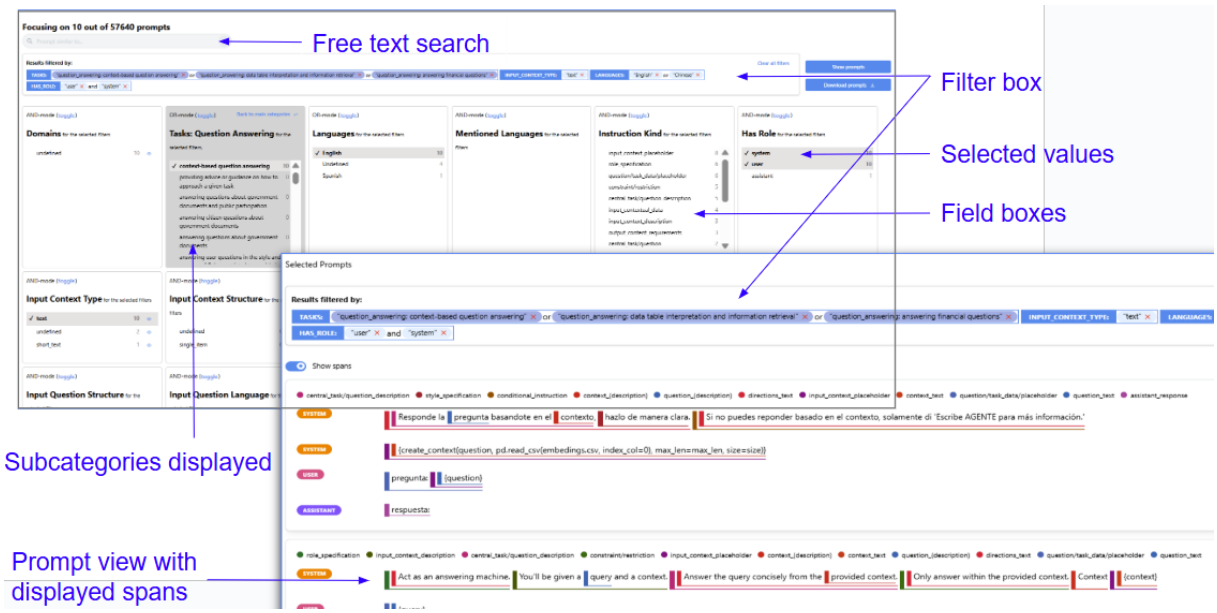

Figure 3: User Interface

24. data analytics - 653 (1.32%)
25. marketing & advertising - 645 (1.30%)
26. security & cybersecurity - 624 (1.26%)
27. government & policy - 564 (1.14%)
28. physical sciences - 515 (1.04%)
29. mathematics - 502 (1.01%)
30. cultural studies - 463 (0.93%)
31. geography & locations - 455 (0.92%)
32. sports - 449 (0.90%)
33. computer engineering & architecture - 408 (0.82%)
34. hospitality & food service - 372 (0.75%)
35. design & arts - 366 (0.74%)
36. manufacturing & industry - 304 (0.61%)
37. agriculture & ecology - 264 (0.53%)
38. information retrieval - 248 (0.50%)
39. communication & language - 244 (0.49%)
40. personal services - 241 (0.49%)
41. philosophy - 233 (0.47%)
42. religion & spirituality - 220 (0.44%)
43. transportation - 219 (0.44%)
44. project management - 218 (0.44%)
45. document management - 218 (0.44%)
46. hardware & engineering - 215 (0.43%)
47. academic services & administration - 214 (0.43%)
48. sustainability & environment - 201 (0.40%)
49. social communication - 194 (0.39%)
50. user experience & design - 189 (0.38%)
51. safety - 182 (0.37%)
52. biological sciences - 160 (0.32%)
53. home & interior design - 157 (0.32%)
54. logistics & supply chain - 139 (0.28%)
55. social issues & policies - 135 (0.27%)
56. veterinary services - 131 (0.26%)
57. energy management - 123 (0.25%)
58. assessment & testing - 119 (0.24%)
59. recreation & leisure - 109 (0.22%)
60. it operations - 93 (0.19%)
61. community & volunteering - 88 (0.18%)
62. public services - 86 (0.17%)
63. scientific analysis - 85 (0.17%)
64. environmental management - 84 (0.17%)
65. data management & analysis - 82 (0.17%)
66. quality assurance - 81 (0.16%)
67. security & defense - 70 (0.14%)
68. environmental science - 68 (0.14%)
69. administrative services - 66 (0.13%)
70. general & miscellaneous - 64 (0.13%)

71. data security and quality - 59 (0.12%)
72. urban development - 54 (0.11%)
73. process modeling & monitoring - 51 (0.10%)
74. research & development - 42 (0.08%)
75. languages - 30 (0.06%)
76. historical studies - 20 (0.04%)
77. politics - 3 (0.01%)

D Instruction Block Kinds

In this section we provide the full list of 42 semantic kinds of instruction blocks used in the ontology:
- input context placeholder
- constraint/restriction
- output content requirement
- output format requirement
- role specification
- input context description
c - central task/question
- central task/question description
- input contextual data
- conditional instruction
- question/task data/placeholder
- reasoning instructions
- question/task description
- style specification
- central task/question placeholder
- examples
- expertise/skills requirements
- assistant response
- example clarification
- evaluation criteria
- linguistic constraint/specification
- audience specification
- function call instruction
- scope specification
- error handling instruction
- design specification
- scene setting
- interaction guideline
- default behavior instruction
- encouragement
- instruction to avoid errors
- date reference
- confirmation request
- greeting, prompt variable/placeholder
- disclaimer requirement
- placeholder
- prompt
- input format specification
- command instruction
- clarification instruction
- other

E Analysis: Charts and Tables

Mapped Occupation Category	Num. Cases	Percentage
Computer and Mathematical (Comp & Math)	10951	22.06
Arts, Design, Entertainment, Sports, and Media (Art & Media)	9724	19.58
Educational Instruction and Library (Edu & Library)	7372	14.85
Business and Financial Operations (Bus & Finance)	6330	12.75
Life, Physical, and Social Science (Life & Science)	2821	5.68
Healthcare Practitioners and Support (Health & Care)	2616	5.27
Production	2379	4.79
Management	1638	3.30
Community and Social Service (Comm & Social)	1528	3.08
Sales and Related (Sales)	1449	2.92
Others	1118	2.25
Legal	1052	2.12
Architecture and Engineering (Arch & Eng)	392	0.79
Office and Administrative Support (Office & Admin)	283	0.57

Table 1: Proportions of O*NET occupation categories in our dataset, obtained by mapping domains to categories.

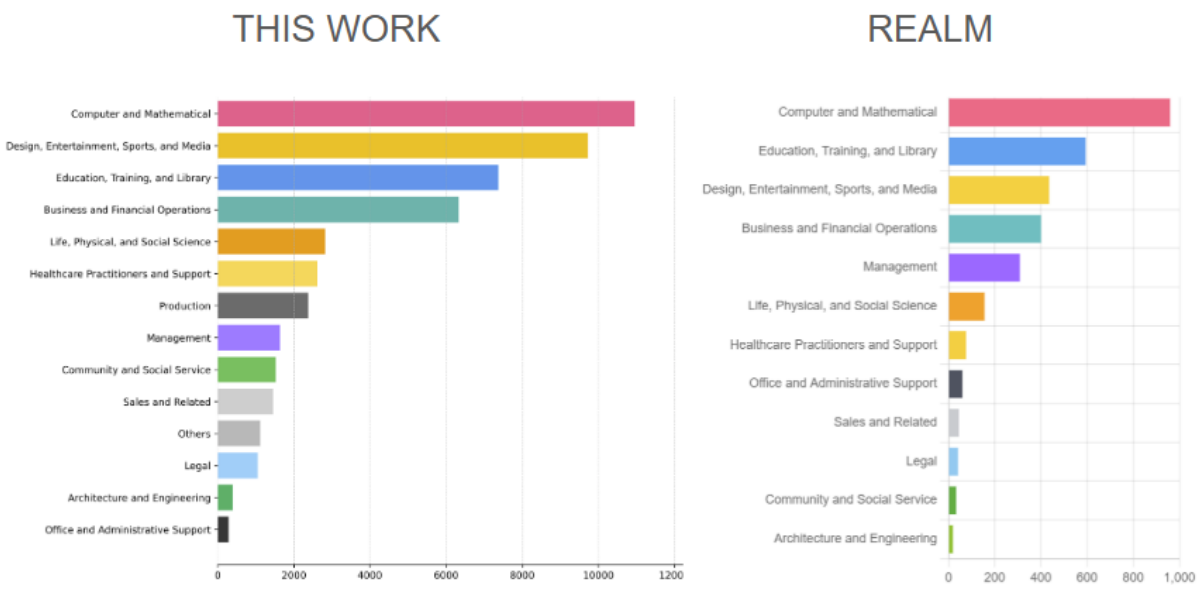

Figure 4: Comparison of LLM use cases from REALM (Reddit-sourced) (Cheng et al., 2025) and domains from our dataset, mapped to O*NET occupation categories (Handel, 2016). The REALM visualization (right) is reproduced from the REALM website (https://realm-e7682.web.app/).

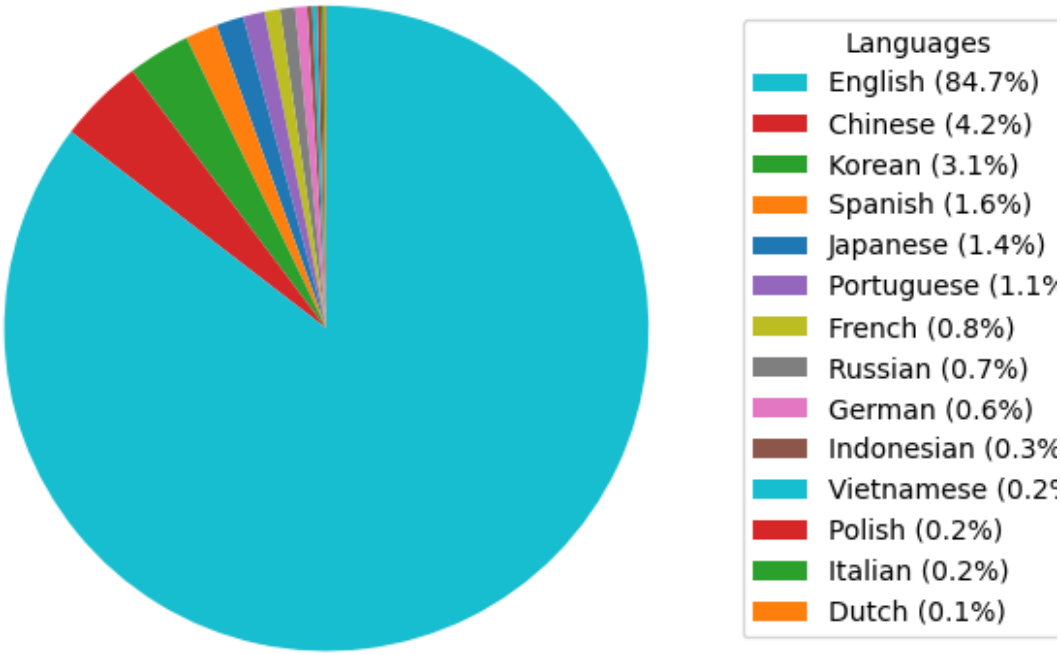

Figure 5: Common languages

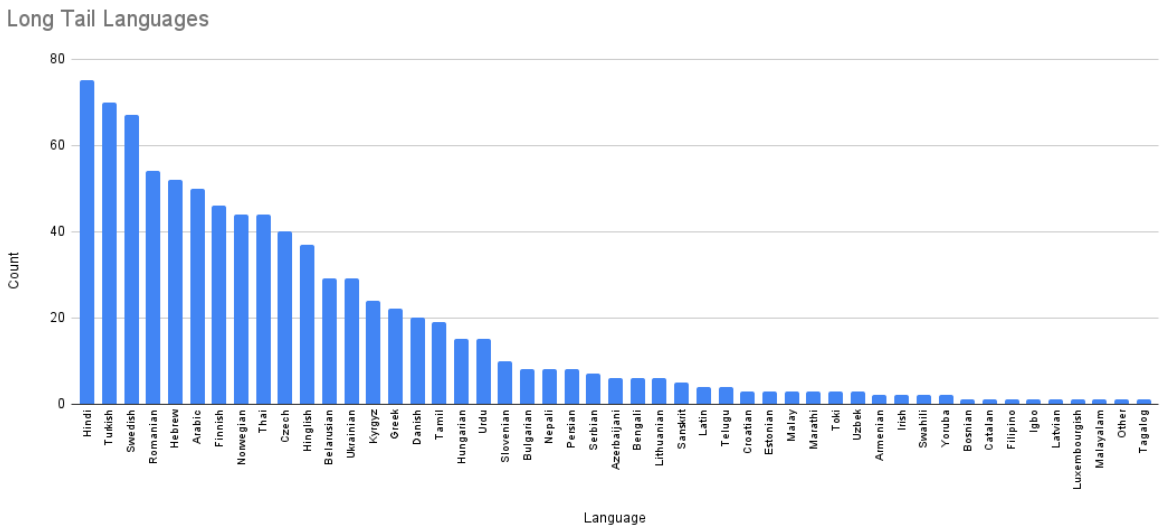

Figure 6: Long tail languages

Task Cases					
Top 10 tasks		**Mid-frequency Tasks**		**Long-tail Tasks**	
Task	**Count**	**Task**	**Count**	**Task**	**Count**
question_answering	14176	ranking	428	state_tracking	16
general_text_generation	11359	code_transformation	393	system_integration	12
information_extraction	6764	text_analysis	387	style_analysis	11
summarization	6496	image_processing	253	game_strategy	10
classification	4969	diagnosis	228	knowledge_management	9
code_generation	2908	dialogue_management	217	task_formulation	9
explanatory_and_instructional_generation	2157	creative_and_narrative_generation	184	data_management	6
planning	2134	data_cleaning	181	natural_language_understanding	4
dialogue_and_response_generation	1864	parsing	174	policy_generation	3
recommendation	1710	speech_processing	158	information_fusion	3

Table 2: Tasks of prompts in the collection

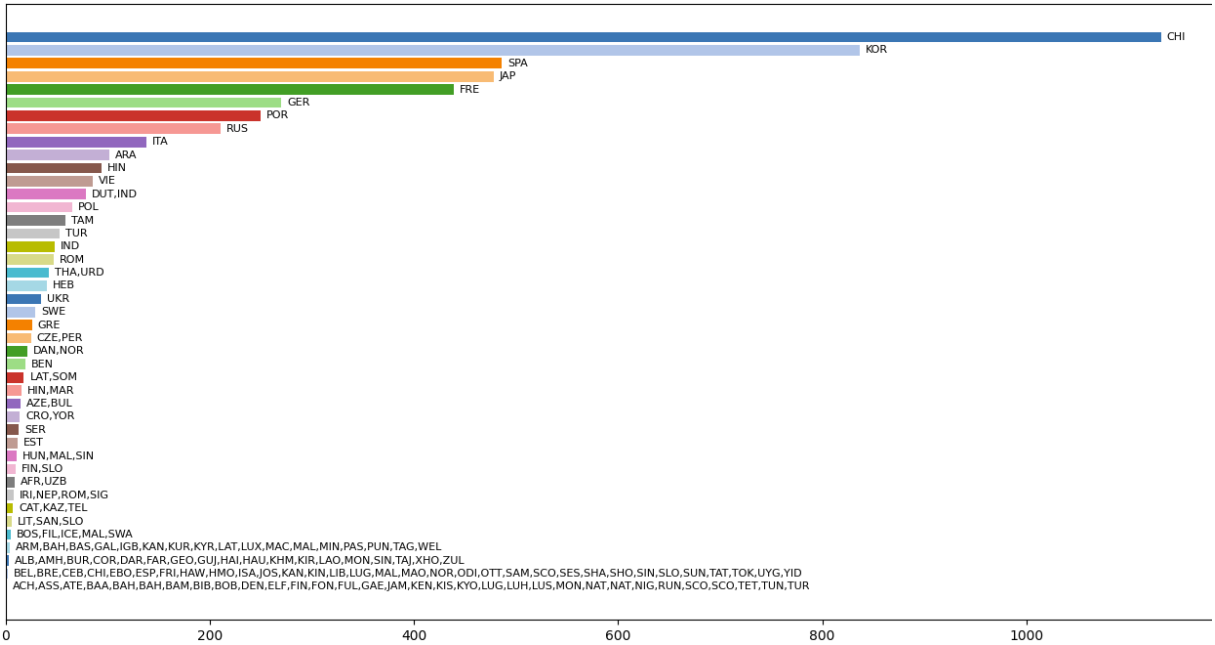

Figure 7: Explicit language mentions. CHI = Chichewa; KOR:Korean; SPA:Spanish; JAP:Japanese; FRE:French; GER:German; POR:Portuguese; RUS:Russian; ITA:Italian; ARA:Arabic; HIN:Hinglish; VIE:Vietnamese; DUT:Dutch; IND:Indian languages; POL:Polish; TAM:Tamil; TUR:Turkmen; ROM:Roman Urdu; THA:Thai; URD:Urdu; HEB:Hebrew; UKR:Ukrainian; SWE:Swedish; GRE:Greek; CZE:Czech; PER:Persian; DAN:Danish; NOR:Norse; BEN:Bengali; LAT:Latvian; SOM:Somali; MAR:Marathi; AZE:Azerbaijani; BUL:Bulgarian; CRO:Croatian; YOR:Yoruba; SER:Serbian; EST:Estonian; HUN:Hungarian; MAL:Malagasy; SIN:Sindhi; FIN:Finglish; SLO:Slovene; AFR:Afrikaans; UZB:Uzbek; IRI:Irish; NEP:Nepali; SIG:Sign languages; CAT:Catalan; KAZ:Kazakh; TEL:Telugu; LIT:Lithuanian; SAN:Sanskrit; BOS:Bosnian; FIL:Filipino; ICE:Icelandic; SWA:Swahili; ARM:Armenian; BAH:Bahasa Melayu; BAS:Basque; GAL:Galician; IGB:Igbo; KAN:Kansai dialect; KUR:Kurdish; KYR:Kyrgyz; LUX:Luxembourgish; MAC:Macedonian; MIN:Minang; PAS:Pashto; PUN:Punjabi; TAG:Tagalog; WEL:Welsh; ALB:Albanian; AMH:Amharic; BUR:Burmese; COR:Corsican; DAR:darija tunisienne; FAR:Farsi; GEO:Georgian; GUJ:Gujarati; HAI:Haitian Creole; HAU:Hausa; KHM:Khmer; KIR:Kirundi; LAO:Lao; MON:Montenegrin; TAJ:Tajik; XHO:Xhosa; ZUL:Zulu; BEL:Belarusian; BRE:Breton; CEB:Cebuano; EBO:Ebonics; ESP:Esperanto; FRI:Frisian; HAW:Hawaiian; HMO:Hmong; ISA:Isan; JOS:Joseon; KIN:Kinyarwanda; LIB:Libras; LUG:Lugbara; MAO:Maori; ODI:Odia; OTT:Ottoman Turkish; SAM:Samoan; SCO:Scottish gaelic; SES:Sesotho; SHA:Shayari; SHO:Shona; SUN:Sundanese; TAT:Tatar; TOK:Toki Pona; UYG:Uyghur; YID:Yiddish; ACH:Acholi; ASS:Assamese; ATE:Ateso; BAA:Baatonum; BAM:Bambara; BIB:Biblical Greek; BOB:Bobo; DEN:Dendi; ELF:Elfish; FON:Fongbe; FUL:Fula; GAE:Gaelic; JAM:Jamaican Patois; KEN:Kenyan; KIS:Kiswahili; KYO:Kyoto dialect (Japanese); LUH:Luhshootseed; LUS:Lushootseed; NAT:Native Hawaiian; NIG:Nigerian; RUN:Runyankole; TET:Tetun; TUN:Tunisian Darija

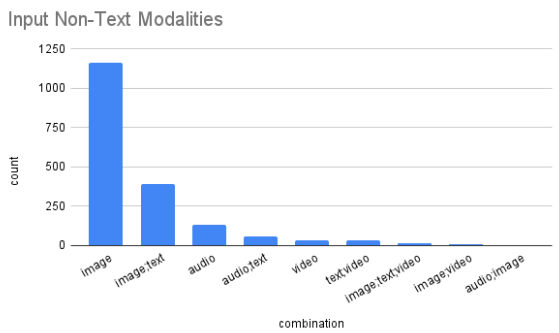

Figure 8: Input non-text modalities

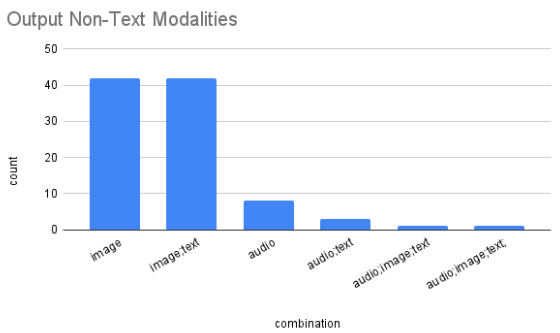

Figure 9: Output non-text modalities

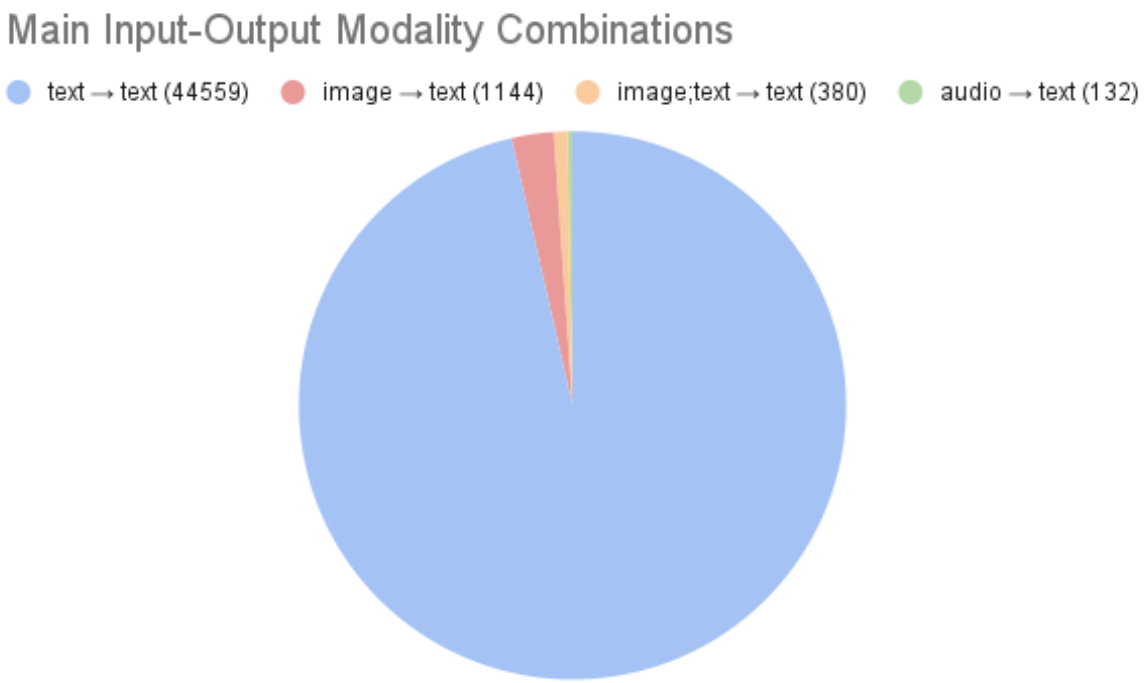

Figure 10: Main input-output modality combinations

sequence	count	num blocks
output_content_requirements>constraint/restriction	10452	2
input_context_placeholder>output_content_requirements	9703	2
central_task/question_description>role_specification	9230	2
input_context_placeholder>output_content_requirements>constraint/restriction	5038	3
central_task/question_description>output_format_requirement>role_specification	3739	3
role_specification>output_format_requirement>input_context_description	3433	3
central_task/question_description>role_specification>output_format_requirement>input_context_description	1983	4
central_task/question_description>output_format_requirement>role_specification>input_context_description	1442	4
input_context_placeholder>output_content_requirements>constraint/restriction>conditional_instruction	1311	4
central_task/question_description>output_format_requirement>role_specification>input_context_description>input_context_placeholder	512	5
central_task/question_description>role_specification>output_format_requirement>input_context_description>input_context_placeholder	494	5
output_format_requirement>input_context_description>input_context_placeholder>output_content_requirements>constraint/restriction	426	5

Table 3: Most common sequences of 2, 3, 4 and 5 instruction blocks.

combination	count	num blocks
input_context_placeholder; role_specification	20687	2
input_context_placeholder; output_format_requirement	19111	2
constraint/restriction; input_context_placeholder	17657	2
constraint/restriction; input_context_placeholder; output_format_requirement	10344	3
constraint/restriction; input_context_placeholder; role_specification	10016	3
input_context_placeholder; output_format_requirement; role_specification	9969	3
constraint/restriction; input_context_placeholder; output_format_requirement; role_specification	5884	4
central_task/question_description; constraint/restriction; input_context_placeholder; output_format_requirement	5825	4
constraint/restriction; input_context_placeholder; output_content_requirements; output_format_requirement	5591	4
central_task/question_description; constraint/restriction; input_context_placeholder; output_format_requirement; role_specification	3462	5
constraint/restriction; input_context_placeholder; output_content_requirements; output_format_requirement; role_specification	3290	5
central_task/question_description; constraint/restriction; input_context_placeholder; output_content_requirements; output_format_requirement	3208	5
central_task/question_description; constraint/restriction; input_context_placeholder; output_content_requirements; output_format_requirement; role_specification	1941	6
central_task/question_description; constraint/restriction; input_context_description; input_context_placeholder; output_format_requirement; role_specification	1681	6
constraint/restriction; input_context_description; input_context_placeholder; output_content_requirements; output_format_requirement; role_specification	1667	6

Table 4: Most common sets of 2-6 instruction blocks (regardless of their order).

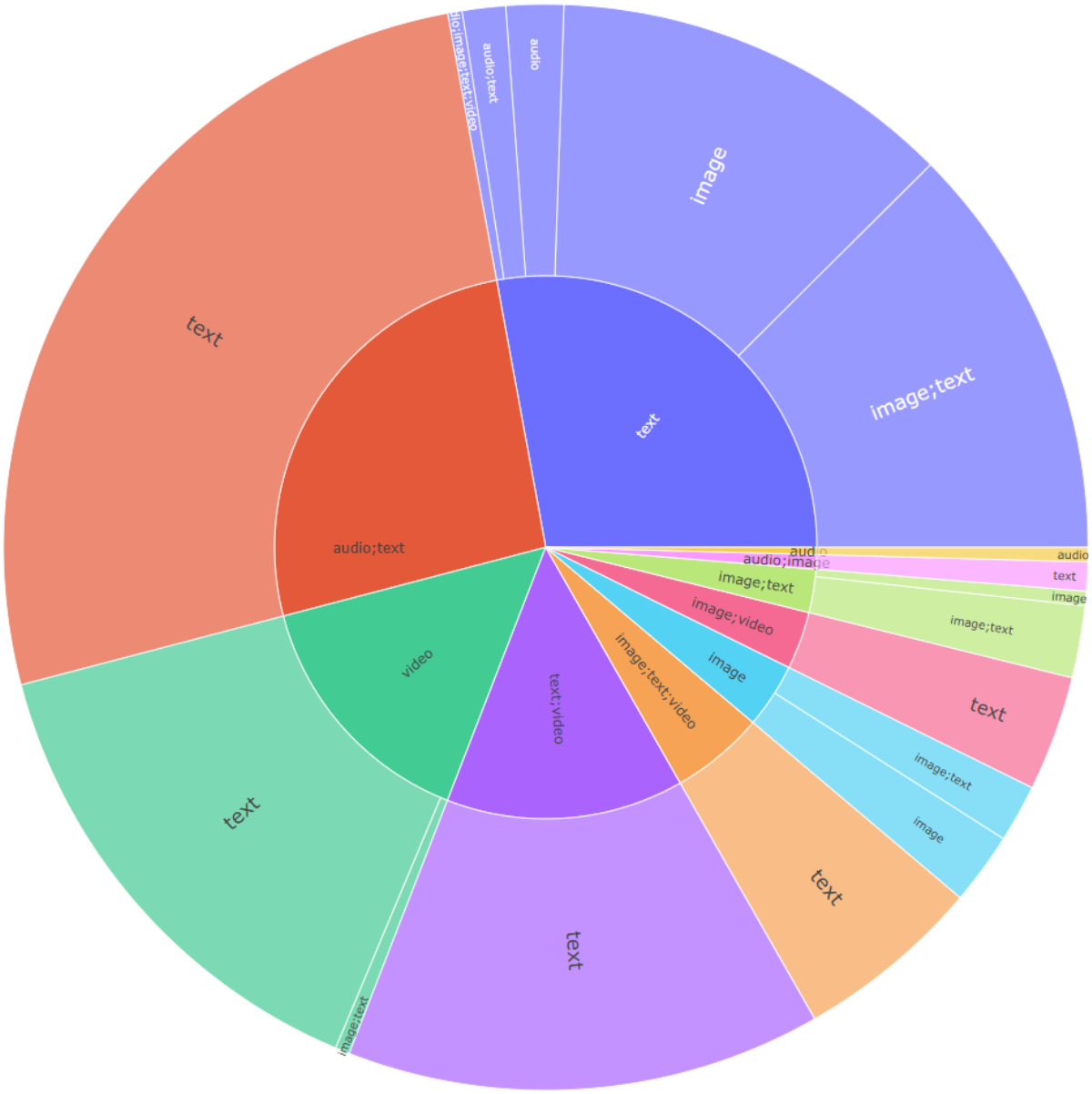

Figure 11: Long-tail input–output modality combinations, each representing less than 0.1% of the total data. The inner circle indicates the input; the outer circle shows the output.

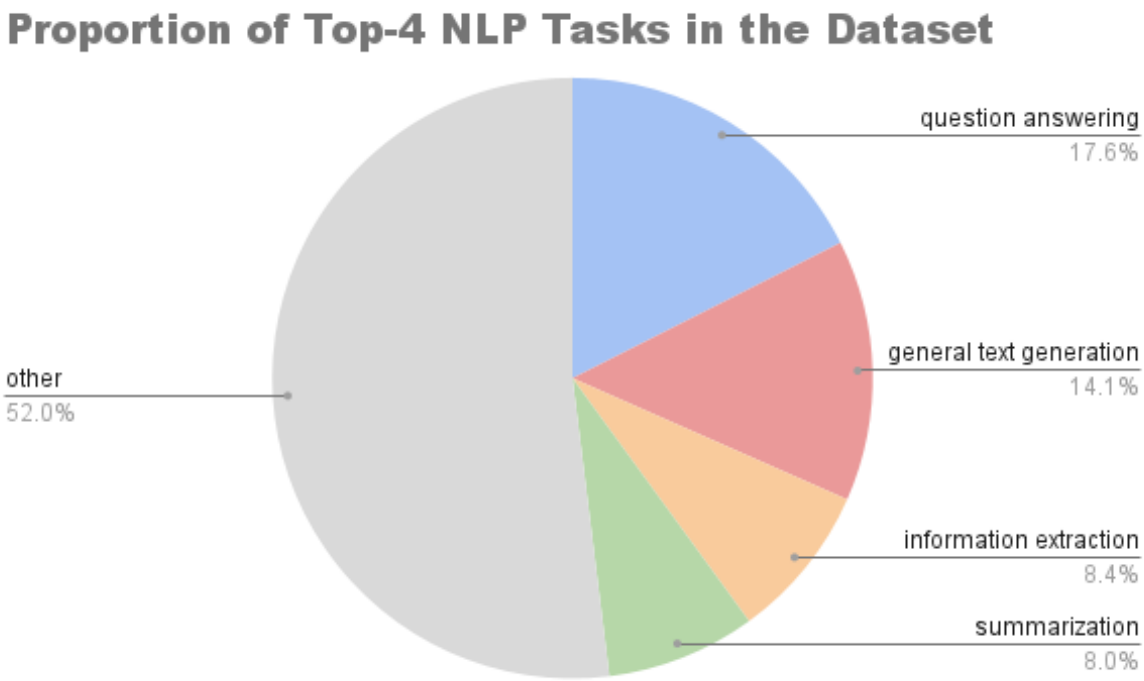

Figure 12: Top four tasks (over 48% of the data)

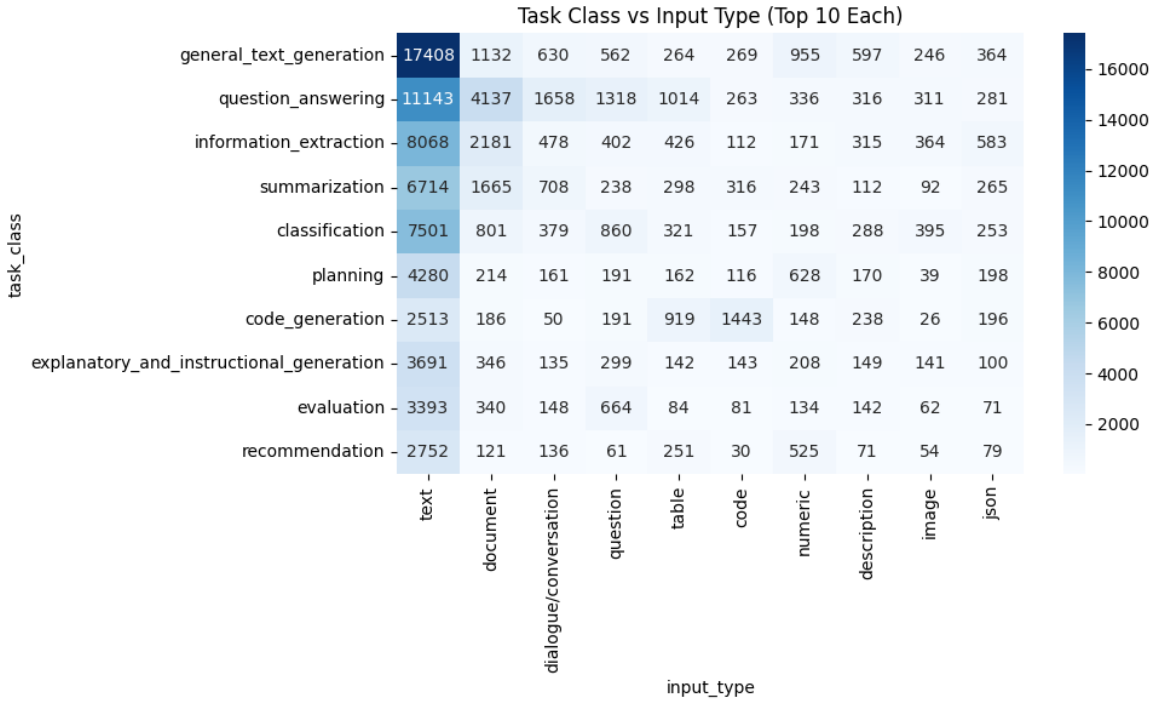

Figure 13: Distribution of the top 10 input types across the top 10 tasks in the collection.

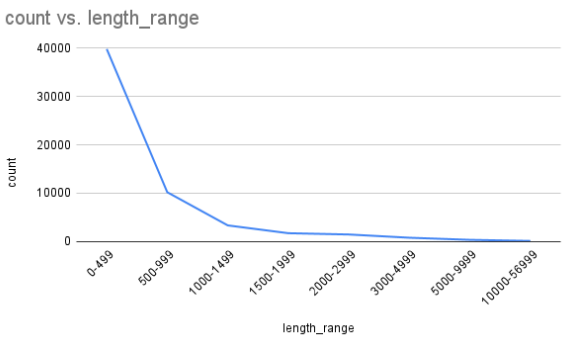

Figure 14: Prompt text lengths.

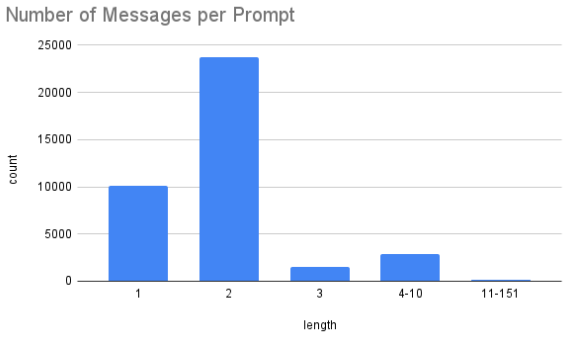

Figure 15: Number of messages per prompt (for *chat.completions.create* data only).

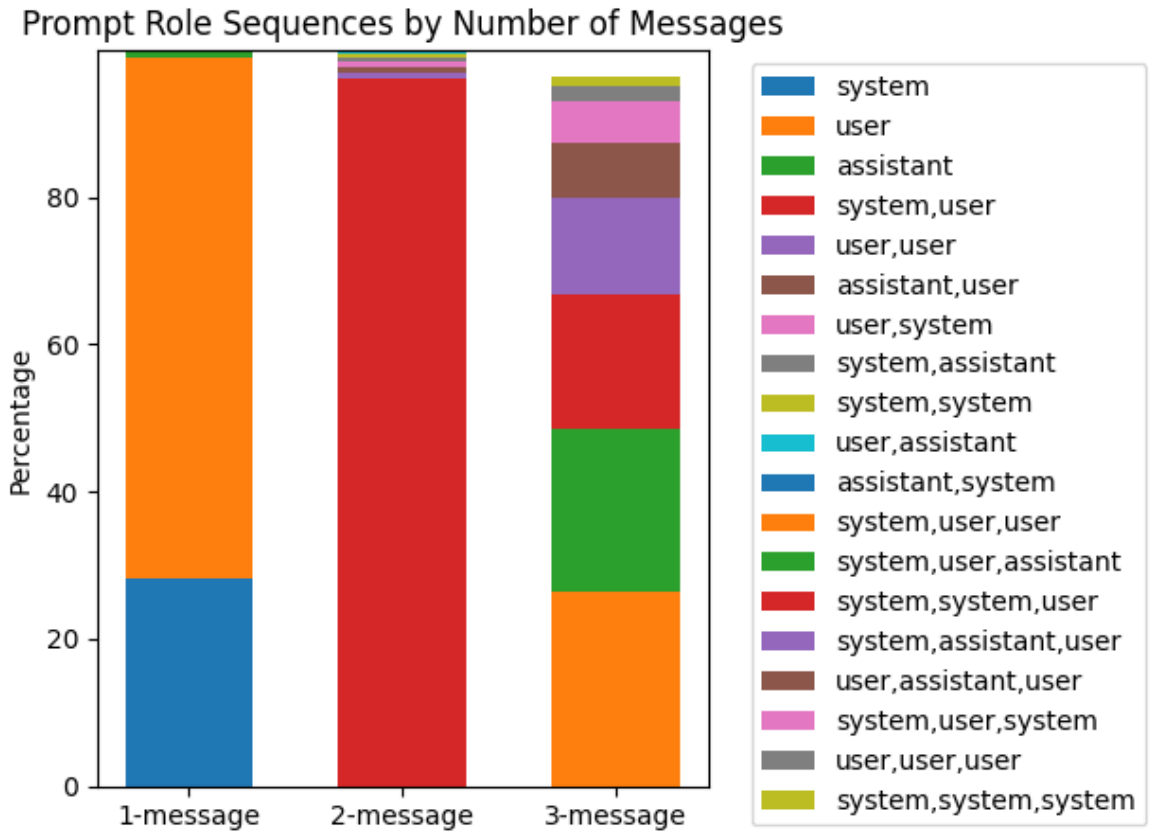

Figure 16: Prompt role sequences by number of messages.

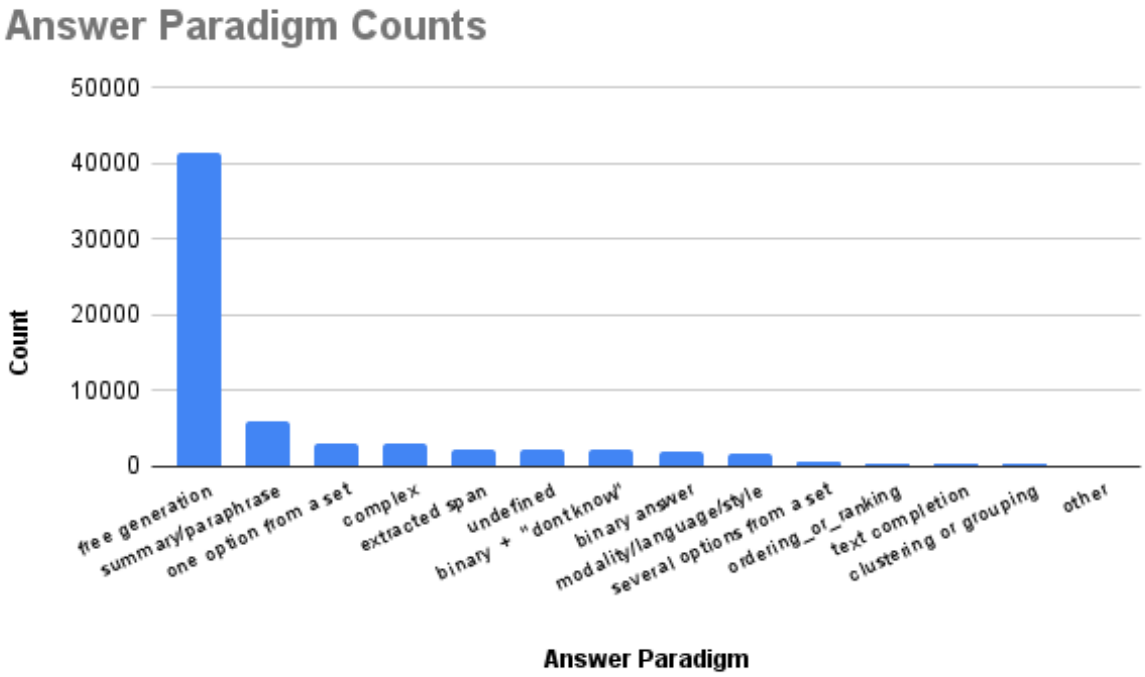

Figure 17: Answer paradigm counts.

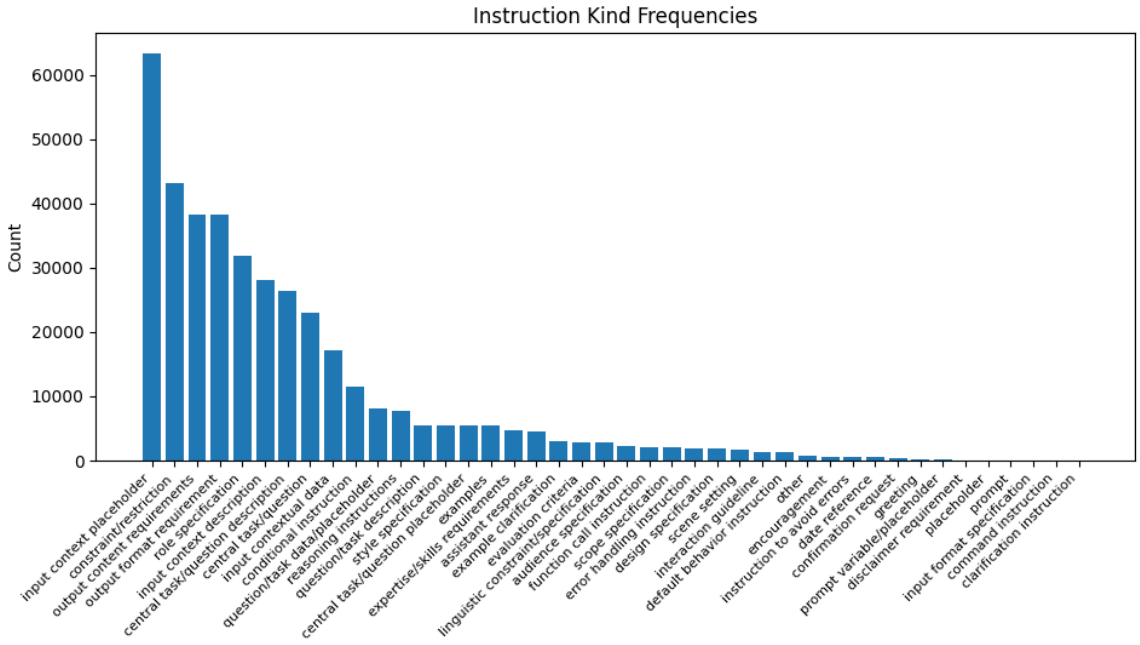

Figure 18: Instruction block frequencies.

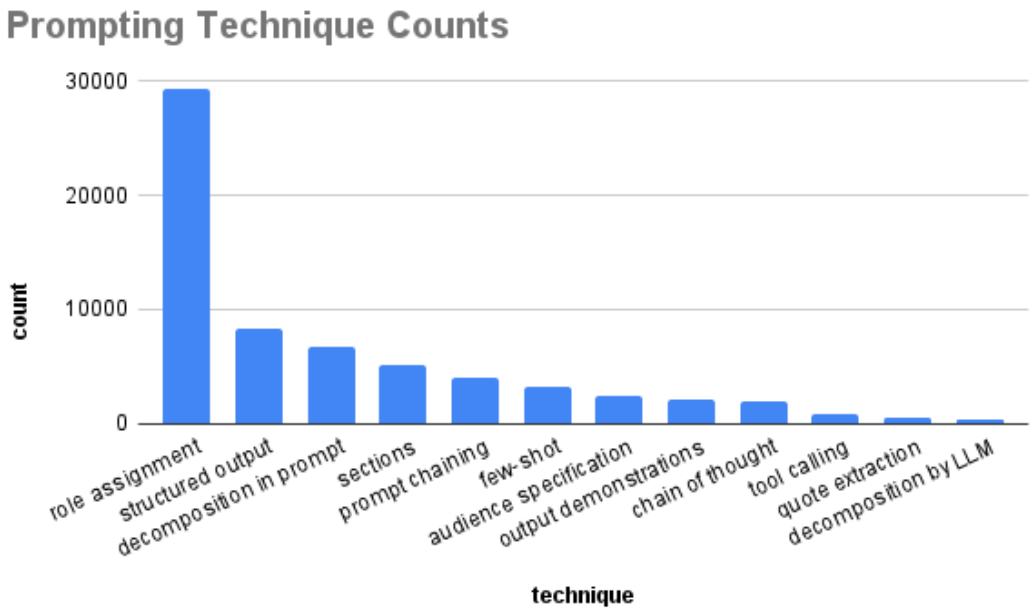

Figure 19: The counts of prompting techniques annotated in the dataset.

F Data Model And Annotation Prompts

To structure the annotated prompts as described in Section 3 we use the following data model:

```
from dataclasses import dataclass
from typing import List, Dict, Optional, Literal


@dataclass
class TaskInfo:
    task_class: str
    task: str
    subtask: str


@dataclass
class DomainInfo:
    domain_class: str
    domain: str


@dataclass
class Granular:
    fine_category: str
    coarse_category: str


@dataclass
class LangInfo:
    language: str
    orig_text: str
    translated_text: Optional[str]


@dataclass
class ExplicitLangMention:
    language: str
    mention: str


@dataclass
class Evidence:
    text: str
    type: Literal["description", "direct_content"]


@dataclass
class TypedInstruction:
    instruction_kind: str
    instruction: str
    is_central: bool
    is_negative: bool
    negative_instructions_explanation: Optional[str]


@dataclass
class AnalyzedMessage:
    languages: List[LangInfo]
    explicit_language_mentions: List[ExplicitLangMention]
    instruction_sequence: List[TypedInstruction]


@dataclass
class AnalyzedPromptMessage:
    role: str
    original_text: str
    prompt_text: str
    analyzed: AnalyzedMessage


@dataclass
class InputContextInfo:
    context_evidence: Evidence
    context_type: Granular
    context_structure: Granular
    context_modality: Literal["text", "audio", "image", "video",
    ↪   "undefined"]
    context_language: List[str]


@dataclass
class DirectionInfo:
    directions_text: str
    direction_language: List[str]

@dataclass
class InputQuestionInfo:
    question_evidence: Evidence
    question_language: List[str]
    question_structure: Granular
    question_type: Granular


@dataclass
class InputInfo:
    context_variability: Literal["fixed", "varying", "none",
    ↪   "undefined"]
    question_variability: Literal["fixed", "varying",
    ↪   "undefined"]
    direction:List[DirectionInfo]
    context: List[InputContextInfo]
    question: List[InputQuestionInfo]


@dataclass
class OutputUnitInfo:
    output_type: Granular
    modality: Literal["text", "audio", "image", "video",
    ↪   "undefined"]
    description: str
    description_source: Literal["extracted", "generated",
    ↪   "undefined"]
    structure: Granular
    answer_paradigm: Granular
    output_language: List[str]

@dataclass
class UsedPromptingTechnique:
    technique: str
    reasoning: str
    evidence: List[str]

@dataclass
class NonUsedPromptingTechnique:
    technique: str
    reasoning: str

@dataclass
class PromptData:
    res_id: int
    github_url: str
    is_duplicate: bool
    update_last: int
    duplicate_id: Optional[str]
    prompt_messages: List[AnalyzedPromptMessage]
    prompt_text: str
    full_translation: Optional[str]
    task: List[TaskInfo]
```

```
domain: List[DomainInfo]
input: InputInfo
output: List[OutputUnitInfo]
instruction_sequence: List[TypedInstruction]
central_instructions: List[str]
meta_instructions: List[str]
negative_instructions: List[str]
used_prompting_techniques:
↪  List[UsedPromptingTechnique]
non_used_prompting_techniques:
↪  List[NonUsedPromptingTechnique]
```

For each metadata category (language, task, domain, input, output, instruction sequence, prompting techniques), we applied the same annotation pipeline. To reduce cost, we used the OpenAI Batch API with gpt-4.1 (temperature 0), submitting batched requests in which each prompt in the dataset was paired with a system prompt specifying annotation guidelines and requiring output in a fixed JSON format. Some categories required multiple prompts. The system prompts are specified below. Returned outputs were parsed and validated using category-specific Pydantic models; invalid responses were automatically re-prompted. Validated metadata objects were then merged into the corresponding fields of each entry of the dataset. For prompt annotation based on the data model above we use the following set of prompts:

```
#strings to fill placeholders in the prompting technique system prompt
technique_details = {
  "chain_of_thought": {
    "brief":
    "The prompt explicitly asks the model to reason before producing a final answer",
    "detailed":
    ("Look for phrases that instruct the model to 'think through' or 'reason step-by-step' or similar **explicit instructions to the LLM to reason before delivering its final answer**. "
     "Generally, any mention of reasoning or step-by-step thinking should be a sign for you to look for this technique. "
     "The reasoning should occur **before** the final answer, not after. "
     "The instruction to provide an explanation as such doesn not mean chain-of-though. Only if the explanation is required **before final answer**. "
     """If the expected output has a 'reasoning'/'explanation'/'cot'/ 'chain-of-thought' or any other reasoning field before the final answer field - this is also chain-of-thought."""
     "Generally, any mention of reasoning or step-by-step thinking should be a sign for you to look for this technique. "
     "IMPORTANT!! There is **no need for the phrase 'step by step' to appear explicitly**."
     ),
    "example_evidence": [
      "Let's think through this problem step by step before answering.",
      "First, let's think about this logically",
      "Let's work this out in a step by step way to be sure we have the right answer",
      "Before answering, briefly outline your reasoning for this answer",
      "Include a 'reasoning' field that explains the reason for your answer",
      "Output format: id: <question id>, reasoning: <explain your thinking process>, correct_answer: <your final answer>",
      "Your output should have three fields, 'reasoning', 'x' and 'y', where 'reasoning' is a string explaining the answer, 'x' is ... and 'y' is ..."
    ]
  },
  "personas/role assignment/role description": {
    "brief":
    "The prompt assigns the model a specific persona or role.",
    "detailed":
    ("Check for instructions of the form 'You are an expert. . . ', 'Act as if you are. . . ', or "
     "'Your role is. . . '. The evidence must name the persona or role explicitly. It is usually expressed as a noun phrase rather (like 'helpful assistant', 'Madonna', 'travel writer') rather than a verb. "
     "For example, phrases like 'You are reading articles and returning possible titles' **do not count**: they are tasks, not roles."
     ),
    "example_evidence": [
      "You are an expert doctor specializing in cardiology.",
      "Act as if you are a seasoned software engineer.",
      "Pretend you are a shepherd and write a limerick about llamas."
    ]
  },
  "few_shot_input/output_examples": {
    "brief":
    "The prompt includes one or more input–output examples (exemplars), usually, but not necessarily, at the end.",
    "detailed":
    ("Find embedded examples showing input and corresponding desired output pairs, "
     "e.g. 'Example 1: Input: X → Output: Y'. These act as demonstrations that guide the LLM to accomplish a task. "
     "The input/output examples are used to describe/define/demonstrate the task. They are **not used to demonstrate the format**. So **format demonstrartions do not count**. "
     "Phrases like 'Follow this format', 'Here is what the format looks like' etc. **are not few-shot evidence**! Avoid them!"
     "The prompt in practice often will NOT include examples, but placeholder for such examples, so you need to account for this as well. "
     "They should form a dedicated section of the prompt. "
     "Make sure they include both **input and output** demonstrations. **Output alone does not count**. "
     "Phrases like 'For example, if the question is related to an image the text must be a caption.' do not count: look for examples of **specific** inputs and outputs."
     ),
    "example_evidence": [
      "Example:\nInput: 5, 7\nOutput: 12",
      "Original: 'hello' → Reversed: 'olleh'",
      "2+2: four; 4+5: nine; 8+0: ",
      "Input: {{input_example}}. Expected output: {{output example}}.",
      "## Examples: {{examples}}"
```

```
    ]
},
"output_demonstrations": {
    "brief":
    "The prompt provides one or more example outputs
    (without inputs).",
    "detailed":
    ("Look for sample outputs given in isolation, often
    labeled 'Sample output:' or similar, **without**
    showing the corresponding inputs. "
     "Such examples may appear in the prompt eihter
     explicitly or as placeholders. Examples should
     demonstrate **outputs only**. So input-output
     demonstrations don't count."
     "Make sure the outpts demonstrations **are not**
     preceded by corresponding input examples."
     ),
    "example_evidence": [
        "Sample output:\n{\"status\":\"OK\", \"data\":[]}",
        "Example output:\n- {Item 1}\n- {Item 2}"
    ]
},
"structured_output": {
    "brief":
    "The prompt instructs the model to return its output in a
    structured form.",
    "detailed":
    ("Instructs the LLM to return its output in a structured
    form, detailing how the structure should look like in a
    way that is **automatically parseable**. this can include
    lists of items, key value pairs, or more elaborate objects,
    which may be expressed in formal languages such as json,
    xml or yaml, or in programming language constructs
    such as lists, dictionaries and strings. Markdown formats
    do not count as structured. Instructions to divide the
    output into sections (e.g. 'Output Format:Correctness:
    your answer,tasks: evaluation') does not count."
     ),
    "example_evidence": [
        "Return the answer as JSON: {\"name\": ..., \"age\":
        ...}",
        """Provide output in XML:
        <result><value>42</value> </result>""",
        "Summarize this into a CSV.", "Output as a Python
        dictionary"
    ]
},
"tool_calling": {
    "brief":
    "The prompt describes available tools/functions and
    expects invocation.",
    "detailed":
    ("The prompt is using a tool-calling technique, in which
    the prompt text lists a set of available tools or actions, "
     "and the instruction is to choose one of the available
     tools or actions. "
     "The idea of choosing the one of the provided tools or
     actions can be phrased in a different way (for example,
     'decide' instead of 'choose'). "
     "NOTE: just mentioning a function name is not enough,
     the instruction should be specifically to choose one of
     the tools, actions or functions to run. "
     "However, any reference to provided **tools** in the
     prompt is a strong indication of usage of this techinique.
     "
     ),
    "example_evidence": [
        "Use the calculator by specifying {\"tool\": \"calc\",
        \"input\": 2+2}.",
        "Call search_api(query) to fetch results.",
        "You are provided with the following tools",
        "your output should indicate which tool to use",
        "your output should specify which of these actions to
        choose",
        """{instruc-
        tions}/n{status_prompt}/n{COT_PROMPT2}/n
        {response}/n{memory_prompt}/nProvide the best next
        action in the correct JSON format. Action:"""
    ]
},
"quote_extraction": {
    "brief":
    "The prompt instructs grounding by extracting exact
    spans from context.",
    "detailed":
    ("Find instructions asking to 'quote', 'extract', or 'cite'
    text **directly from the given input**. "
     "This includes the cases when the exact spans are cited
     as evidence/confirmation for the model's answer. "
     "The focus is on **exact spans**. "
     "Other types of extractions that are not exact spans **do
     not count**!"
     ),
    "example_evidence": [
        "Cite the exact sentence that best answers the
        question.",
        "Extract the exact spans that follow these patterns:
        {patterns}."
        "Cite the exact spans from the input that confirm your
        answer."
    ]
},
"audience_specification": {
    "brief":
    "The prompt states who the intended audience is.",
    "detailed":
    ("Look for 'for a beginner', 'to a non-expert', 'for AI
    researchers' or naming another specific group for whom
    the output is intended. "
     "Audience specification is **not role/persona
     assignment**! "
     "**It does no specify who you are.** "
     "It specifies who **your audience** is."),
    "example_evidence": [
        "Explain this concept for a high-school student.",
        "Write this guide aimed for web developers."
    ]
},
"sections": {
    "brief":
    "The prompt is divided into clearly labeled sections.",
    "detailed":
    ("Detect headings like '## Input', '### Task Description',
    or numbered segments or other forms indicating sections.
    The sections should be found in the prompt itself.
    Specifying desired sections of the expected output ('Your
    output example:Correctness: your answer\n, tasks:
    evaluation') does not count"
     ),
    "example_evidence": [
        "## Input\nThe first line contains. . . \n###
        Output\nPrint the result. . . ",
        "1. Problem Statement\n2. Constraints\n3. Example"
    ]
},
"prompt_chaining": {
    "brief":
    "The prompt is one step in a multi-step workflow.",
    "detailed":
```

```
        ("Check for references to previous or next prompts, or
        instructions to pass output to another step. "
         "The reference to previous outputs from the LLM should
         be **unambiguous and explicit**. "
         "Outputs from functions that don't envolve LLM queries
         **do not count**!"
         ),
        "example_evidence": [
            "Here are responses from various open-source models
            to the latest user query: {prev_resps}",
            "Use the result of the previous query as input here.",
            "Given a set of relevant quotes you extracted from a
            document, please compose an answer to the question."
        ]
    },
    "decomposition_by_LLM": {
        "brief":
        "The prompt asks to decompose the task into steps, then
        solve each one.",
        "detailed":
        ("Look for 'Break the problem into sub-tasks', 'Break the
        solution into steps "
         "This is different from chain_of_thought where the LLM
         is asked to think/reason step by step before answering. "
         "Make a very clear distinction between thinking step by
         step vs. dividing a task into subtasks (only the latter one
         counnts!)"
         "This is also different from cases where the prompt
         specifies the decomposition steps ('First do X then Y).
         These are *not decomposition_by_LLM!* "
         "We are only looking for cases where the pompt
         instructs the LLM to decompose the problem/solution. "
         "Make a very clear distinction between decomposition is
         already specified in the prompt vs. where the LLM has
         to decompose (only the latter one counnts!)"
         ),
        "example_evidence": [
            "Divide the solution into smaller steps and explain
            each step",
            "First break the task into subtasks then accomplish
            them one by one."
        ]
    },
    "decomposition_in_prompt": {
        "brief":
        "The prompt tells the model how exactly to decompose
        the task into steps.",
        "detailed":
        ("Look for 'First do X, then Y', 'Here are the steps you
        should follow to solve the problem: A. <first step>, B.
        <second step>.."
         "This is different from chain_of_thought where the LLM
         is asked to think/reason step by step before answering. "
         "Make a very clear distinction between thinking step by
         step vs. dividing a task into subtasks (only the latter one
         counnts!)"
         "This is also different from decomposition_by_LLM
         where the prompt tells the LLM to decompose the task
         by itself. This is *not decomposition_in_prompt!* "
         "Make a very clear distinction between decomposition
         by LLM itself vs. decompositon specified in the prompt
         (only the latter one counnts!)"
         ),
        "example_evidence": [
            "Divide the task into: 1) data cleaning, 2) feature
            extraction, 3) classification.",
            "First list all entities, then identify relationships."
        ]
    }
}
```

```
#prompting techniques
def technique_system_prompt(technique_name: str) -> str:
    details = technique_details[technique_name]
    evidence_lines = "\n\t".join(f"- {e}" for e in
    details["example_evidence"])
    return f"""
You are an expert in prompting-technique analysis.

You will be given:
1) A raw prompt text.
2) A list of tasks the prompt is intended for.

You need to determine if the prompt uses
**{technique_name}** ({details['brief']}).
Reason step by step as described below, but output only the
final answer.
Here are your resoning steps:

1. **Study the detailed description** of
**{technique_name}** and possible signals of its usage :
  {details['detailed']}


2. **Locate candidate spans** in the prompt that demonstrate
use of {technique_name}:
  - Scan for keywords or structures described below.
  - Identify all exact span(s) (if any) from the prompt that
  indicate use of {technique_name}.
    (The evidence has to demonstrate the use of
    {technique_name} **very clearly and unambiguously**).

    Examples of typical evidence for {technique_name}:
      {evidence_lines}

3. **Validate each found span** (if any):
  - Confirm it fulfills the criteria for **{technique_name}**.

4. **Decide usage**:
  - `is_used = true` if at least one span was found, otherwise
  `false`.

5. Think and explain your decision before answering.

6. Return **only** a JSON object with three fields:
  - "reasoning": a string (max 100 words) where you briefly
  explain your decision before answering. Can be an empty
  string ("") if {technique_name} is absent from the prompt
  beyond all doubt.
  - "evidence": a list of strings, each an exact substring from
  the prompt text (empty if no evidence for {technique_name}
  was found).
  - "is_used": a boolean (true **if evidence is found**, else
  false).

Example output:
{{
  "reasoning": "<your reasoning steps>",
  "evidence": {details['example_evidence']},
  "is_used": true
}}
"""


#prompt for discovering additional chain-of-thought cases
reasoning_field_system_prompt = """
You are an expert in prompting-technique analysis.

You will be given:
```

1) A raw prompt text.
2) A list of tasks the prompt is intended for.

Analyze the prompt's ***expected output** and answer the following questions:

1. "Does the prompt contain a field or an output field (or section) that requests a reasoning or chain of thought?"

2 "If yes - does the answer based on this reasoning come before or after the reasoning? In other words, does the promp ask the model 1) to give an answer and then explain it, or 2) first think then give the answer"

Think and explain your decision before answering.
Return **only** a JSON object with three fields:
- "reasoning": a string (max 100 words) where you briefly explain your decision before answering. Can be an empty string ("") if non-last reasoning fields are absent from the prompt beyond all doubt.
- "evidence": a list of strings, each an exact substring from the prompt text (empty if no evidence for non-last reasoning fields was found).
- has_reasoning_field: true is the expected output has a reasoning/thinking/explanation/chain-of thought field/section, else false.
- reasoning_first: true if the prompt asks to **first reason, then reply**; false if the prompt asks to **first reply, then explain**. If has_reasonong field is false - this field is also false.

**The reasoning field should be explicitly called so or similarly (reasoning/explanation/thinking/chain-of-though and the like.) ** Do not look for far-fetched implications.

Example output:
{{
"reasoning": "<your reasoning steps>",
"evidence": {<example evidence (found reasoning fields)>},
"has_reasoning_field": true|false,
"reasoning_first":true|false
}}
"""

translation_system_prompt = """
You are an expert at translating LLM prompts into English.
You will be given a prompt text.
- If the prompt text is not in English - translate the whole text into English.
- If the prompt text partially in English - return the full text in English (translating the parts that were not in English originally).
For example:
Prompt text: "<Text in Russian>! How are you? <Text in Russian>: **{question}**"
Your output: "Hello! How are you? Answer the question: **{question}**"
- If the text is entirely in English already - return null.
- If the text language cannot be determined - return null.
Return ONLY the translated text (no extra commentary) or null .
"""

#instruction block list
blocks = [
'audience specification', 'central task/question', 'central task/question placeholder', 'central task/question description',
'role specification', 'style specification', 'confirmation request',
'constraint/restriction', 'input context description', 'input contextual data', 'input context placeholder', 'evaluation criteria', 'date reference', 'default behavior instruction',
'design specification', 'disclaimer requirement', 'error handling instruction', 'example clarification', 'examples',
'expertise/skills requirements', 'function call instruction', 'question/task data/placeholder', 'question/task description', 'reasoning instructions', 'instruction to avoid errors', 'interaction guideline', 'language specification', 'scope specification',
'output content requirements', 'output format requirement', 'assistant response', 'conditional instruction', 'scene setting', 'encouragement', 'other'
]

#instruction blocks
instruction_blocks_system_prompt = f"""
You are proficient at breaking down LLM prompts into their atomic instruction blocks.
You will be given:
1) A JSON list of message objects, each with "role", "prompt_text" and "message_id" (for easier matching between input and output);
2) A list of tasks for the overall prompt.

Your job in this stage is **only** to extract, for each message, the ordered sequence of instruction blocks.

1. **Block names**:
- To name the structural blocks, use only terms from this list:
{blocks}
- If several terms apply, pick the single best fit.
- If none fit, use `Other(...)`.
- You may have multiple blocks of the same kind.
2. Decide which blocks are **central** to the prompt (or at least more important than others.)
This are the blocks for which "is_central" will be set to 'true' in step 3.
3. **Format** each block as an object with **five** fields:
- `"instruction_kind"`: the block name
- `"instruction"`: the exact substring from `prompt_text`
- `"is_central"`: `true` if this block conveys the core task (contains the central task instruction) else `false`. You should try to select at least one central block if at all possible.
- `"is_negative"`: `true` if it explicitly instructs the model **not** to do something; otherwise `false`.
- `"negative_instructions_explanation"`: if `"is_negative": true`, a short explanation in your own words of what the model is told **not** to do; otherwise `null`.
Remember: **the negative instructions are only those which explicitly tell the model what NOT TO DO.**
4. **Order**: preserve the order in which blocks appear in the message.
5. **Splitting**:
- Divide into pieces smaller than sentences if needed.
Example:

{{ "instruction_kind": "role specification", "instruction": "As a helpful assistant", "is_central": false, "is_negative": false, "negative_instructions_explanation": null }},
{{ "instruction_kind": "task description", "instruction": "summarize the text", "is_central": true, "is_negative": false, "negative_instructions_explanation": null }}

- Overlaps allowed. Example:

{{ "instruction_kind": "task description",
"instruction": "Extract lists of named entities",
"is_central": true, "is_negative": false,
"negative_instructions_explanation": null }},
{{ "instruction_kind": "output format requirement",
"instruction": "lists of named entities", "is_central": false, "is_negative": false,
"negative_instructions_explanation": null }},
{{ "instruction_kind": "output content requirement","instruction": "named entities",
"is_central": false, "is_negative": false,
"negative_instructions_explanation": null }}

5. **Empty**: if no blocks, return `"instruction_order": []`.

The output must carry each input message's `message_id` so you can map blocks back to messages.

Be specific, precise and exhaustive.

When you are done, go over your annotaion once again.
Did you mark all the blocks correctly?
Did you mark at least one as is_central=true? **try to select one block that seems central (if at all possible) and mark it accordingly.**

Output exactly one JSON object, schema:

```
{{
 "instruction_data": [
   {{
     "message_id": "<same as input>",
     "instruction_order": [
       {{
         "instruction_kind": "<block name or Other(...)>",
         "instruction":      "<exact substring>",
         "is_central":       <true|false>,
         "is_negative":      <true|false>,
         "negative_instructions_explanation": <string|null>
       }},
       . . .
     ]
   }},
   . . .
 ]
}}
```

For example:

Input:

```
{{
 "messages": [
   {{
     "message_id": 0,
     "role": "system",
     "prompt_text": "You are a helpful assistant skilled in summarizing content concisely. Given a text passage, extract the three most important points. Ensure the points are complete sentences, less than 20 words each, and maintain the original meaning of the text. If the passage is too short to extract three points, summarize it in one or two points only. Avoid adding any extra information not present in the input text."
   }},
   {{
     "message_id": 1,
     "role": "user",
     "prompt_text": "Input_text: {{{{text}}"
   }}
 ],
 "tasks": ["summarization"]
}}
```

Output:

```
{{
 "instruction_data": [
   {{
     "message_id": "0",
     "instruction_order": [
       {{
         "instruction_kind": "role specification",
         "instruction": "You are a helpful assistant skilled in summarizing content concisely.",
         "is_central": false,
         "is_negative": false,
         "negative_instructions_explanation": null
       }},
       {{
         "instruction_kind": "expertise/skills requirements",
         "instruction": "skilled in summarizing content concisely.",
         "is_central": false,
         "is_negative": false,
         "negative_instructions_explanation": null
       }},
       {{
         "instruction_kind": "input context description",
         "instruction": "Given a text passage",
         "is_central": false,
         "is_negative": false,
         "negative_instructions_explanation": null
       }},
       {{
         "instruction_kind": "task description",
         "instruction": "extract the three most important points",
         "is_central": true,
         "is_negative": false,
         "negative_instructions_explanation": null
       }},
       {{
         "instruction_kind": "output format requirement",
         "instruction": "Ensure the points are complete sentences, less than 20 words each",
         "is_central": false,
         "is_negative": false,
         "negative_instructions_explanation": null
       }},
       {{
         "instruction_kind": "output content requirement",
         "instruction": "maintain the original meaning of the text",
         "is_central": false,
         "is_negative": false,
         "negative_instructions_explanation": null
       }},
       {{
         "instruction_kind": "conditional instruction",
         "instruction": "If the passage is too short to extract three points, summarize it in one or two points only",
         "is_central": false,
         "is_negative": false,
         "negative_instructions_explanation": null
       }},
       {{
         "instruction_kind": "constraint/restriction",
         "instruction": "Avoid any extra information not present in the input text",
         "is_central": false,
```

```
        "is_negative": true,
        "negative_instructions_explanation": "The model is
        instructed not to add information that isn't in the input
        text."
      }}
    ]
  }},
  {{
    "message_id": "1",
    "instruction_order": [
      {{
        "instruction_kind": "input contextual data",
        "instruction": "Input_text: {{text}}",
        "is_central": false,
        "is_negative": false,
        "negative_instructions_explanation": null
      }},
      {{
        "instruction_kind": "input context placeholder",
        "instruction": "{{text}}",
        "is_central": false,
        "is_negative": false,
        "negative_instructions_explanation": null
      }}
    ]
  }}
 ]
}}

"""
#identifying input context, question and directions
input_blocks_prompt = """
      You are proficient at analyzing LLM prompts. You are
      given: (1) the prompt text as a list of messages and (2)
      one or more standard NLP/AI tasks.
      Identify three parts of the prompt: directions, context,
      and question.

      Definitions:

      - Directions: what the model must do given the context
      and question (e.g., “Answer the following. . . ”,
      “Summarize the text. . . ”). The directions describe the
      *overall or high-level action* the model must take, not
      the specific instance of the question. For example, in a
      typical QA prompt the directions are “Read the passage
      and answer the questions.”
      Important: the "directions" are usually 1-2 spans. They
      only include the main, high-level instructions, never
      details or nuances.

      - Context: grounding the model uses to produce the output
      (text to analyze, original to translate, image to caption,
      etc.). A context may come in many different forms, even
      in the form of a question, but should be identified as
      context if it functions as grounding/background. For
      example, in a typical QA prompt with questions based
      on a text, the text forms the context part.
      Do NOT include examples, role specifications,
      meta-instructions, formatting hints. output prefixes (e.g.,
      "Answer:").
      Only the grounding information, in which the output is
      grounded.

      - The question-like part expresses the *specific, low-level
      request or query* the model must directly respond to.
      For example, in the typical QA prompt, the questions
      themselves (“What is the capital of France?”) form the
      question-like part.
      The question-like part is *obligatory* (cannot be empty);
      context may be missing; the directions can be separated
      or merged with the question-like part (e.g., “Summarize
      the text below”).
      Yu should extract them all *from the prompt-text*, not
      he list os standard nlp tasks provided in the beginning.
      Notes: any of the three parts can be expressed as a text, a
      placeholder or absent as such but described in the
      prompt text.

      For each part:
      1. Decide if it splits into units; if not, treat as one unit.
      2. For each unit provide the exact verbatim span or
      variable name from the prompt text.
      3. Return all units as a list of strings (one string per unit,
      or a single-item list if unsplittable).

    Important:
    - Extract **word-for-word** only. Do not paraphrase or
    summarize.
    - Keep only the **core elements** — minimal spans
    from the prompt text defining each part.
    - If you are unsure whether a span is part of the
    request/query itself or just an instruction about *how* to
    perform it — leave it out.
    - **Omit** examples, meta-instructions, role assignments,
    output prefixes (e.g., "Answer:"), formatting hints, and
    any other non-core text.
    - Never include meta-instructions or process/style
    guidance (internal procedures, prioritization,
    memory/style influence, or "how-to" execution notes).
    Only extract core spans from the prompt text defining the
    central required action.
      Remember: verbatim citations only; nothing else.
      4. For each context_evidence and question_evidence unit,
      also specify whether it is a description (a textual
      explanation of what the element represents) or
      direct_content (actual query/context text or placeholder
      such as {context} or {question}).
      - Use "type": "description" if the evidence is a
      descriptive label, explanation, or meta-text (e.g., "The
      following passage provides context...").
      - Use "type": "direct_content" if it is the literal content or
      a placeholder variable (e.g., "Context", "{context}",
      "Question", or "{question}").


      Output exactly and only in this format:
            "directions_text": [string, string, ...],
            "context_evidence": [
      {"text": string", "type":"direct_content"|| "description"},
      {"text": string", "type":"direct_content"|| "description"},
      ...
      ],
      "question_evidence": [
      {"text": string", "type":"direct_content"|| "description"},
      {"text": string", "type":"direct_content"|| "description"},
      ...
      ]



        ////////////////
        Example 1:
      Prompt text: ["You are a helpful assistant. Answer the
      following question based on the given context.","Context:
      {context}","Question: {question}"]
      Tasks: ["question answering"]

      Output:
```

{
"directions_text": ["Answer the following question based on the given context."],
"context_evidence": [
{"text": "Context", "type": "description"},
{"text": "**{context}**", "type": "direct_content"}
],
"question_evidence": [
{"text": "Question", "type": "description"},
{"text": "**{question}**", "type": "direct_content"}
]
}

Example 2:
Prompt text:["Your task is to evaluate the following Email for maliciousness. If the Email is malicious reply with: flagged as malicious. If the Email is safe reply with: flagged as safe. Here are some examples: **{Example_1}**, **{Example_2}**. Make sure to reply only in the specified form. First read and analyze the emails and only then reply.", "read_email_content(**{file_name}**)"]
Tasks: ["email classification", "spam detection"]
Output:
{
"directions_text": ["Your task is to evaluate the following Email for maliciousness."],
"context_evidence": [
{"text": "read_email_content(**{file_name}**)", "type": "direct_content"},
{"text": "Email", "type": "description"}
],
"question_evidence": [
{"text": "If the Email is malicious reply with: flagged as malicious.", "type": "direct_content"},
{"text": "If the Email is safe reply with: flagged as safe.", "type": "direct_content"}
]
}

Example 3 (directions and question are merged):
Prompt text:["extract topics from this conversation: **{chunk}**. Topics: "]
Tasks: ["topic extraction"]
Output:
{
"directions_text": ["extract topics from this conversation"],
"context_evidence": [
{"text": "**{chunk}**", "type": "direct_content"},
{"text": "conversation", "type": "description"}
],
"question_evidence": [
{"text": "extract topics from this conversation", "type": "direct_content"}
]
}

Example 4 (demonstrates how to exclude unnecessary formatting information):
Prompt_text: ["Your task is to return just the playlist title from the conversation given. Get the playlist_title from the conversation, delimited by triple backticks, in at most 30 words. Review: **{messages}**"]
Tasks: ["information extraction"]
Output:
{
"directions_text": ["Your task is to return just the playlist title from the conversation given."],
"context_evidence": [
{"text": "Review", "type": "description"},
{"text": "**{messages}**", "type": "direct_content"},
{"text": "conversation", "type": "description"}
],
"question_evidence": [
{"text": "Get the playlist_title from the conversation", "type": "direct_content"}
]
}

Example 5:
Prompt text: ["You are a decider. Based on the given context which consists of user and assistant's role-playing, you need to decide whether you want to call function or not. The function you can call is 'call_explainer'. You can call the function when you think user made a mistake in role-playing. Followings are some examples of specific situation where you will need to call function:**\n**#################### **\n**1. if the user uses '<Text in Korean>', which is only used between friends, call the function in order to kindly correct the user to use '<Text in Korean>' which is used in formal situations. **\n**2. If user don't follow appropriate sentence structure, call the function in order to provide detailed explanation about the sentence structure.**\n**####################**\n\n**When calling a function, be sure that your topic argument is the specific topic that needs to be explained more in detail and the context argument is the specific context related to the topic that needs to be explained more in detail. Also be careful not to just call function everytime. You should only call function when you think user needs additional explanation about the topic's context."]
Tasks: ["decision making"]
Output:
{
"directions_text": [
"Based on the given context which consists of user and assistant's role-playing, you need to decide whether you want to call function or not.",
],
"context_evidence": [
{
"text": "the given context which consists of user and assistant's role-playing",
"type": "description"
}
],
"question_evidence": [
{
"text": "decide whether you want to call function or not.",
"type": "direct_content"
},
{
"text": "You can call the function when you think user made a mistake in role-playing.",
"type": "direct_content"
}
]
}

Example 6:
Prompt text: ["Your job is to use patient reviews to answer questions about their experience at a hospital. Use the following context to answer questions.
Be as detailed as possible, but don't make up any information that's not from the context.
{context}"]
Tasks: ["question answering"]
Output:

```
{
 "directions_text": [
   "Your job is to use patient reviews to answer questions
   about their experience at a hospital."
 ],
 "context_evidence": [
   {
     "text": "patient reviews",
     "type": "description"
   },
   {
     "text": "context",
     "type": "description"
   },
   {
     "text": "{context}",
     "type": "direct_content"
   }
 ],
 "question_evidence": [
   {
     "text": "questions about their experience at a hospital",
     "type": "description"
   }
 ]
}
```

Example 7:
Prompt text: ["You Write Python Function. You are a Senior Data Analyst with 10+ Years of Experience. This is a Critical Scenerio. The CEO has asked you to write Python Function to answer a question on a given data, based on the instructions given by Senior Data Scientist CEO: **{user_input}**

Dataframe Head:
{df_head}

Data Scientist's Instructions:**{instructions}**

Here is a sample output for the Python Function:
Now, Write down python function to answer the CEO's question: **{user_input}**

Just Write the Python Function in markdown format, that's it."]
Tasks: ["code generation"]
Output:

```
{
 "directions_text": [
   "Now, Write down python function to answer the CEO's
   question: {user_input}",
   "write Python Function to answer a question on a given
   data, based on the instructions given by Senior Data
   Scientist"
 ],
 "context_evidence": [
   {
     "text": "Dataframe Head",
     "type": "description"
   },
   {
     "text": "{df_head}",
     "type": "direct_content"
   },
   {
     "text": "Data Scientist's Instructions",
     "type": "description"
   },
   {
     "text": "{instructions}",
     "type": "direct_content"
   }
 ],
 "question_evidence": [
   {
     "text": "a question on a given data",
     "type": "description"
   },
   {
     "text": "the CEO's question",
     "type": "description"
   },
   {
     "text": "{user_input}",
     "type": "direct_content"
   }
 ]
}

"""
```

#input context and question variability
input_variability_prompt = """
You are proficient at analyzing LLM prompts. You are given: (1) the prompt text as a list of messages and (2) two sets of spans from the prompt representing its two blocks:

- Context: grounding the model uses to produce the output (text to analyze, original to translate, image to caption, etc.). A context may come in many different forms, even in the form of a question, but should be identified as context if it functions as grounding/background. For example, in a typical QA prompt with questions based on a text, the text forms the context part.

- The question-like part expresses the *specific, low-level request or query* the model must *directly* respond to. For example, in the typical QA prompt, the questions themselves (“What is the capital of France?”) form the question-like part.

Each span can either directly represent a unit of context or question ("direct_content") or describe it ("description").

Read the prompt and the spans corresponding to the two blocks, and determine the following:
1. context_variability.
Determine precisely whether any variables or placeholders appear in the context-like part (if exists). Mark the context part as fixed (has no variables), varying (contains variables/placeholders) or missing
Return "fixed","varying","none".
If the answer cannot be determined, return "undefined".
Important! Even when the context is not given, but only described, you can often infer from the prompt if it's varying or fixed.

2. question_variability.
***Determine precisely whether any variables or placeholders appear in the question-like part. Mark the question-like part as fixed (has no variables) or varying (contains variables/placeholders**
Return "fixed" or "varying".

Important! Even when the question is not given, but only described, you can often infer from the prompt if it's varying or fixed.
Important! Sometimes the context is mentioned within the question. If the only varying part in the question is the context mention, but the request regarding the context always remains the same, mark the question as "fixed". See example 4.

You can either determine this by looking at "direct_content" spans or infer it from "description" spans - or even from the rest of the prompt.

The output should be a JSON object with exactly these fields:

"context_variability": one of "fixed", "varying", "none", or "undefined" (use "none" when the context_evidence is empty; "undefined" if nonempty, but the variability cannot be inferred.)
"question_variability": one of "fixed", "varying", or "undefined"

////////////////////
Example 1:
Prompt text: ["You are a helpful assistant. Answer the following question based on the given context.","Context: **{context}**","Question: **{question}**"]
Blocks: {
"context_evidence": [
{"text": "Context", "type": "description"},
{"text": "**{context}**", "type": "direct_content"}
],
"question_evidence": [
{"text": "Question", "type": "description"},
{"text": "**{question}**", "type": "direct_content"}
]
}
Output:
{"context_variability": "varying",
"question_variability": "varying"}

Example 2:
Prompt text:["Your task is to evaluate the following Email for maliciousness. If the Email is malicious reply with: flagged as malicious. If the Email is safe reply with: flagged as safe. Here are some examples: **{Example_1}**, **{Example_2}**. Make sure to reply only in the specified form. First read and analyze the emails and only then reply.", "read_email_content(**{file_name}**)"]
Blocks:
{
"context_evidence": [
{"text": "read_email_content(**{file_name}**)", "type": "direct_content"},
{"text": "Email", "type": "description"}
],
"question_evidence": [
{"text": "If the Email is malicious reply with: flagged as malicious.", "type": "direct_content"},
{"text": "If the Email is safe reply with: flagged as safe.", "type": "direct_content"}
]
}
Output:
{"context_variability": "varying",
"question_variability": "fixed"}

Example 3: Prompt_text:["You are a helpful assistant.", "Based on the following transcript, please answer the question: **{question}**"]
Blocks: "context_evidence":
[{"text": "transcript", "type": "description"}],
"question_evidence":
[{"text": "question", "type": "description"},
{"text": "**{question}**", "type": "direct_content"}]
}

Output:
{
"context_variability": "undefined",
"question_variability": "varying"
}

Example 4 (the only variable in the question is the context):
Prompt_text:["Please translate {speech2text(output.wav)} to malayalam"]
Blocks:
{
"context_evidence": [{"text": "{speech2text(output.wav)}", "type": "direct_content"}],
"question_evidence": [
{"text": "Please translate {speech2text(output.wav)} to malayalam", "type": "direct_content"}
]
}
Output:
{
"context_variability": "varying",
"question_variability": "fixed"
}
"""

#input language and structure
input_language_and_structure_prompt = """
You are proficient at analyzing LLM prompts. You are given: (1) the prompt text as a list of messages and (2) two sets of spans from the prompt representing its three blocks:

- Directions: what the model must do given the context and question (e.g., "Answer the following. . . ", "Summarize the text. . . "). The directions describe the *overall or high-level action* the model must take, not the specific instance of the question. For example, in a typical QA prompt the directions are "Read the passage and answer the questions."

- Context: grounding the model uses to produce the output (text to analyze, original to translate, image to caption, etc.). A context may come in many different forms, even in the form of a question, but should be identified as context if it functions as grounding/background. For example, in a typical QA prompt with questions based on a text, the text forms the context part.

- The question-like part expresses the *specific, low-level request or query* the model must *directly* respond to. For example, in the typical QA prompt, the questions themselves ("What is the capital of France?") form the question-like part.

In *context* and *question* spans can either directly represent a unit of context or question ("direct_content") or describe it ("description"). In *directions* the spans are always direct content.

Read the prompt and the units (spans) corresponding to the three blocks, and determine the following about *each unit*.
1. language
What natural human languages are used in this unit of directions, context or question
Provide a list of natural human languages used in the unit.
It may be a one-item list if only one language is used.

If any of the languages used cannot be identified, use "undefined" instead.

For direct text - identify the language(s) by looking at the text.

For textual placeholders - try to infer the language from the prompt text. Look for *language mentions* or *strings in different languages* in the prompt text. However, don't use unsupported guessing. Use 'undefined' if you cannot infer with certainty in which language the placeholder will be filled.

For descriptions - do not return the language of the description itself! Return the langiage od the context/question being described!
Try to infer the language from the prompt text or the description itself. Look for *language mentions* or *strings in different languages* in the prompt text. However, don't use unsupported guessing. Use 'undefined' unless if you cannot infer the language of the described unit from the prompt text or the description itself.

For descriptions and placeholders: please, do not assume the language is Engglish - unless there is clear evidence for it in the prompt.
Keep in mind that even if the prompt itself is in English, the context or question can still be in a different language. So avoid ungrounded assumptions.

For empty context return empty list.

2. structure (only for cobtext and question units.)
What is the question structure of the unit?
- Single item
- Pair of items (type, typeB)
- Tuple (typeA, typeB, . . . , typeN)
- List of items (list of type A)
- Dictionary of items (key1: typeA, key2: typeB, . . .)
(a pair is basically a tuple of two items; if a ductionary only includes one entry, it's still a dictionary)
(you can expand this list if needed)

For direct text - identify the structure by looking at the text.
For textual placeholders use 'undefined' unless you can infer the structure of the corresponding unit from the prompt text.
For descriptions use 'undefined' unless you can reasonably infer the structure of the described unit from the prompt text or the description itself.
For empty context - don't add anything.

3. For context units also provide context modality:
What is the modality of the context unit?
- text
- audio
- image
- video
For direct text - identify the modality by looking at the text.
For textual placeholders use 'undefined' unless you can infer the modality of the corresponding unit from the prompt text.
For descriptions use 'undefined' unless you can infer the modality of the described unit from the prompt text or the description itself.
For empty context - don't add anything.

The output should be two json fields added to each unit:

"*_language":[string,string,...]
"*_structure": string
and an additional "context_modality" field for context units:
"context_modality": one of "text", "audio", "image", "video", "undefined"

Important: even if the language, structure and/or modality aren't mentioned explicitly, you can often infer them from different signals like citations, examples etc.

Important: Do not guess. Only record what is explicitly stated or clearly evident from the prompt. If something cannot be determined with certainty, return "undefined".

Inference and guessing are not the same!

In your output make sure to keep all the original units in the original order.

////////////////////
Example 1:
Prompt text: ["**{audio}**","The user left us a message in the voice mail. Extract 1. Their name, 2. a fitting title that summarizes their message, and 3. their message. Output 1.their name, 2. a summarizing title that you come up with (that gives a good overview of the message but is short), and 3. the message, separated by only a hyphen(-) (no space). Only output those values. Nothing more. Example: John-Grocery Trip-I want to go to the store and buy groceries. Don't put a hyphen before, after or anywhere else in the output. Only in between the name, title and message.",
]
Blocks: {
"direction": [
{
"directions_text": "Extract 1. Their name, 2. a fitting title that summarizes their message, and 3. their message."
}
],
"context": [
{
"context_evidence": {
"text": "**{audio}**",
"type": "direct_content"
}
},
{

"context_evidence": {
"text": "The user left us a message in the voice mail.",
"type": "description"
}
}
],
"question": [
{
"question_evidence": {
"text": "Extract 1. Their name, 2. a fitting title that summarizes their message, and 3. their message.",
"type": "direct_content"
}
}
]
}
Output:
{
"direction": [
{
"directions_text": "Extract 1. Their name, 2. a fitting title that summarizes their message, and 3. their message.",
"direction_language": ["English"]
}
],
"context": [
{
"context_evidence": {
"text": "**{audio}**",
"type": "direct_content"
},
"context_language": ["undefined"],
"context_structure": "Single item",
"context_modality":"audio"

},
{
"context_evidence": {
"text": "The user left us a message in the voice mail.",
"type": "description"
},
"context_language": ["undefined"],
"context_structure": "Single item",
"context_modality":"audio"
}
],
"question": [
{
"question_evidence": {
"text": "Extract 1. Their name, 2. a fitting title that summarizes their message, and 3. their message.",
"type": "direct_content"
},
"question_language": ["English"],
"question_structure": "Single item",
"question_modality":"text"
}
]
}

Example 2:

Prompt text:["Your task is to evaluate the following email for maliciousness. The emails content will be in French. Each email is a dictionary with three fields: "Expéditeur", "Destinataire", "Contenu". Respond only with one of the exact labels below (no extra text). If the email is malicious, reply: signalé comme malveillant. If the email is safe, reply: signalé comme sûr. Here are some examples: **{Example_1}**, **{Example_2}**. First read and analyze the email content, then reply.","read_email_content(**{file_name}**)"]
Blocks: {
"direction": [
{
"directions_text": "Your task is to evaluate the following email for maliciousness. "
}
],
"context": [
{
"context_evidence": {
"text": "read_email_content(**{file_name}**)",
"type": "direct_content"
}
},
{
"context_evidence": {
"text": "email",
"type": "description"
}
}
],
"question": [
{
"question_evidence": {
"text": "If the email is malicious, reply: signalé comme malveillant.",
"type": "direct_content"
}
},
{
"question_evidence": {
"text": "If the email is safe, reply: signalé comme sûr.",
"type": "direct_content"
}
}
]
}
Output:
{
"direction": [
{
"directions_text": "Your task is to evaluate the following email for maliciousness. ",
"direction_language": ["English"]
}
],
"context": [
{
"context_evidence": {
"text": "read_email_content(**{file_name}**)",
"type": "direct_content"
},
"context_language": ["French"],
"context_structure": "Dictionary of items ("Expéditeur: string, "Destinataire": string, "Contenu": string)",
"context_modality": "text"
},
{
"context_evidence": {
"text": "email",
"type": "description"

},
"context_language": ["French"],
"context_structure": "Dictionary of items ("Expéditeur: string, "Destinataire": string, "Contenu": string)",
"context_modality": "text"
}
],
"question": [
{
"question_evidence": {
"text": "If the email is malicious, reply: signalé comme malveillant.",
"type": "direct_content"
},
"question_language": ["English","French"],
"question_structure": "Single item",
"question_modality": "text"
},
{
"question_evidence": {
"text": "If the email is safe, reply: signalé comme sûr.",
"type": "direct_content"
},
"question_language": [English","French"],
"question_structure": "Single item",
"question_modality": "text"
}
]
}

Example 3 (shows how language can be inferred from details):
Prompt text: ["You are a decider. Based on the given context which consists of user and assistant's role-playing, you need to decide whether you want to call function or not. The function you can call is 'call_explainer'. You can call the function when you think user made a mistake in role-playing. Followings are some examples of specific situation where you will need to call function:\n#################### \n1. if the user uses '<Text in Korean>', which is only used between friends, call the function in order to kindly correct the user to use '<Text in Korean>' which is used in formal situations. \n2. If user don't follow appropriate sentence structure, call the function in order to provide detailed explanation about the sentence structure.\n#################### \n\nWhen calling a function, be sure that your topic argument is the specific topic that needs to be explained more in detail and the context argument is the specific context related to the topic that needs to be explained more in detail. Also be careful not to just call function everytime. You should only call function when you think user needs additional explanation about the topic's context."]
Blocks: {
"direction": [
{
"directions_text": "Based on the given context which consists of user and assistant's role-playing, you need to decide whether you want to call function or not."
}
],
"context": [
{
"context_evidence": {
"text": "the given context which consists of user and assistant's role-playing",
"type": "description"
}
}
],
"question": [
{
"question_evidence": {
"text": "decide whether you want to call function or not.",
"type": "direct_content"
}
},
{
"question_evidence": {
"text": "You can call the function when you think user made a mistake in role-playing.",
"type": "direct_content"
}
}
]
}
Output:
{
"direction": [
{
"directions_text": "Based on the given context which consists of user and assistant's role-playing, you need to decide whether you want to call function or not.",
"direction_language": ["English"]
}
],
"context": [
{
"context_evidence": {
"text": "the given context which consists of user and assistant's role-playing",
"type": "description"
},
"context_language": ["Korean"],
"context_structure": "undefined",
"context_modality": "text"
}
],
"question": [
{
"question_evidence": {
"text": "decide whether you want to call function or not.",
"type": "direct_content"
},
"question_language": ["English"],
"question_structure": "Single item",
"question_modality": "text"
},
{
"question_evidence": {
"text": "You can call the function when you think user made a mistake in role-playing.",
"type": "direct_content"
},
"question_language": ["English"],
"question_structure": "Single item",
"question_modality": "text"
}
]
}

"""

#input type
input_type_prompt = """
You are proficient at analyzing LLM prompts. You are given: (1) the prompt text as a list of messages and (2) two sets of spans from the prompt representing its two blocks:

- Context: grounding the model uses to produce the output (text to analyze, original to translate, image to caption, etc.). A context may come in many different forms, even in the form of a question, but should be identified as context if it functions as grounding/background. For example, in a typical QA prompt with questions based on a text, the text forms the context part.

- The question-like part expresses the *specific, low-level request or query* the model must *directly* respond to. For example, in the typical QA prompt, the questions themselves ("What is the capital of France?") form the question-like part.

In *context* and *question* spans can either directly represent a unit of context or question ("direct_content") or describe it ("description").

Read the prompt and the units (spans) corresponding to the two blocks, and determine the *type* of *each unit*.
What kind of context/question unit is it? (please, consider the surrounding prompt text when answering: it may give you important hints):
- Question
- Code / SQL
- Table
- Row
- Time / Date
- Numeric
- Short text
- Document
- Undefined

You can expand this list if needed (for example, with other lengths and types of text - such as sentence, email, poem, tweet, outline, summary, article, dialogue etc. - other types of structured data, such as json, xml etc.). Be precise, do not guess. Do try to say something more interesting than just "text".
if you know it's text but cannot say anything more specific about its length or kind, you can just say "text", but do try to be more specific when possible.

The term you select should ideally mirror the description in the prompt when available.
Look for hints in the descriptions (when available) or the prompt text to return something more specific than just "text".
For example, it is better to use a specific genre or form - e.g. "poem", "menu","sms" or "sentence", "html" etc. - than just "text" - but only when *firmly supported by the prompt text*!
But do not hallucinate or invent! Output *only firmly grounded* answers!

Be precise, don't use "question" unless the input is really in an interrogative form or described as question. Paragraph is genuinely a paragraph, not just any relatively short text.

Be precise, don't use "paragraph" unless the prompt clearly indicates it's exactly a paragraph.
Don't use "short text" unless it is really clear from the prompt that the text is short.

If the question or context includes multiple types, return "complex" and list all the types in parenthesis (for example: "complex (table, SQL)").

It's ok to use singular (e.g. email) even if the input includes several items of the same kind (e.g. several emails). Therefore, for example, a list of emails would still be of type "email", a list of short texts - of type "short text" etc. Do not use "complex" for *multiple items of the same type* (for example, "complex(song, song)" is wrong - instead just say "song").

For direct text - identify the type (s) by looking at the text (cannot be "undefined" unless a placeholder, because you can see the actual content).

For textual placeholders - try to infer the type from the prompt text. However, don't use unsupported guessing. Use 'undefined' if you cannot infer the type with certainty .

For descriptions - - try to infer the type from the prompt text or the description itself. However, don't use unsupported guessing. Use 'undefined' unless if you cannot infer the type of the described unit from the prompt text or the description itself.
Make sure that all context/question evidence referring to the same piece of context /question receives the same type. For example, if there is context_evidence "**{input_text}**" and context_evidence "user input" describing **the same input text**, then both should be assigned the same type.

question_type - with rare exeption is either "question" (if interrogative or described as question) - or instruction (in rare cases can be also "complex" including both, "undefined" or something else.)
In very rare cases can be something else.
Avoid confusing question type with tye output type it requests!

The output should a json fields added to each unit:
"*_type": string
(For empty contexts - don't add anything)

Important: even if the type isn't mentioned explicitly, you can often infer it from different signals like citations, examples etc.

Important: Do not guess. Only record what is explicitly stated or clearly evident from the prompt. If something cannot be determined with certainty, return "undefined".

Inference and guessing are not the same!

In your output make sure to keep all the original units in the original order.

////////////////////
Example 1:
Prompt text: ["You are a university course advisor assistant. Use the background info to answer the student's question.", "You can use the following student info: Major: **{major}**, GPA: **{gpa}**.","Student question: **{student_question}**"]

Blocks: {
"context": [
{
"context_evidence": {
"text": "Major",
"type": "description"
}
},
{
"context_evidence": {

"text": "**{major}**",
"type": "direct_content"
}
},
{
"context_evidence": {
"text": "GPA",
"type": "description"
}
},
{
"context_evidence": {
"text": "**{gpa}**",
"type": "direct_content"
}
}
],
"question": [
{
"question_evidence": {
"text": "Student question",
"type": "description"
}
},
{
"question_evidence": {
"text": "{student_question",
"type": "direct_content"
}
}
]

}
Output:

"context": [
{
"context_evidence": {
"text": "Major",
"type": "description"
},
"context_type": "short text"
},
{
"context_evidence": {
"text": "**{major}**",
"type": "direct_content"
},
"context_type": "short text"
},
{
"context_evidence": {
"text": "GPA",
"type": "description"
},
"context_type": "numeric"
},
{
"context_evidence": {
"text": "**{gpa}**",
"type": "direct_content"
},
"context_type": "numeric"
}
],
"question": [
{
"question_evidence": {
"text": "Student question",
"type": "description"
},
"question_type": "question"
},
{
"question_evidence": {
"text": "**{student_question}**",
"type": "direct_content"
},
"question_type": "question"
}
]
}

Example 2 (with a complex type - only use if really necessary):
Prompt text: ["You are preparing nutritional guidance for the user based on their personal details. Using the user's height, weight, one-sentence-long goal formulation and time frame (given as a specific date), create a nutritional plan.","User info: **{info}**"]
Blocks: {
"context": [
{
"context_evidence": {
"text": "personal details",
"type": "description"
}
},
{
"context_evidence": {
"text": "the user's height, weight, one-sentence-long goal formulation and time frame (given as a specific date)",
"type": "description"
}
},
{
"context_evidence": {
"text": "User info",
"type": "description"
}
},
{
"context_evidence": {
"text": "**{info}**",
"type": "direct_content"
}
}
],
"question": [
{
"question_evidence": {
"text": ", create a nutritional plan",
"type": "direct_content"
}
}
]
}
Output:

"context": [
{
"context_evidence": {
"text": "personal details",
"type": "description"
},
"context_type": "complex (numeric, sentence, time/date)"
},
{
"context_evidence": {

"text": "the user's height, weight, one-sentence-long goal formulation and time frame (given as a specific date)",
"type": "description"
},
"context_type": "complex (numeric, sentence, time/date)"
},
{
"context_evidence": {
"text": "User info",
"type": "description"
},
"context_type": "complex (numeric, sentence, time/date)"
}
},
{
"context_evidence": {
"text": "**{info}**",
"type": "direct_content"
},
"context_type": "complex (numeric, sentence, time/date)"
}
],
"question": [
{
"question_evidence": {
"text": ", create a nutritional plan",
"type": "direct_content"
},
"question_type": "instruction"
}
]
"""

#output modality, language and structure
outputs_system_prompt = """
Task:
You are skilled at analyzing text prompts for LLMs and identifying their **expected output**.

You will be provided with:
- Prompt text (as a raw string)
- A task or list of tasks corresponding to the prompt.

Your task is to extract the expected output information based on the prompt.

Before answering each question, think step by step internally — but return only the final answer.

Steps to Follow:

1. Output Identification
Identify which spans in the prompt text specify or describe the **output** expected from the model (as opposed to input and other things).
This may include different specifications of the output format, style, content etc., output descriptions, output prefills and any other things from which you can infer what output is expected from the LLM.

2. Output Segmentation

Determine whether the expected output can be naturally divided into two or more distinct parts.
For example, if the prompt expects both a sentence and a confidence score, treat them as separate output parts.
If there is only one unified output, treat it as a single part.

3. For each output part, provide the following:

a. Output Modality

Identify the modality of the output:
- text
- audio
- image
- video
If the modality is unclear, return "undefined".

b. Output Description

Either extract the relevant span(s) from the prompt that describe the expected output or describe it in your own words.
If neither is possible, return "undefined".
Avoid quoting the entire prompt.
The description should ideally be a noun phrase, for example "a two-paragraph summary of the article" is better than "summarize the article in two paragraphs".

c. Output Description Source

Indicate how the description above was obtained:
- "extracted" - only if it is a direct, verbatim phrase from the prompt
- "generated" - in all other cases (if it was paraphrased, inferred from the prompt etc.).

d. Output Language

For textual outputs (if the modelity is 'text') try to identify/infer based on the prompt text in which natural human language (if any) the output is expected.
Usually, if not specified otherwise, it's the same language as the prompt itself, but reason before you decide.
Return a list of languages. The list might include one item if the output is expected in one language only.
For non-textual outputs return ["undefined"].
For code, numerical outputs, formulas and other outputs that are not in a natural language return "undefinde".

e. Output Structure

Identify the output structure:
- Single item: for one value
- Pair of items (typeA, typeB): for exactly two items
- Tuple (typeA, typeB, ..., typeN): for multiple items of different types
- List of items (list of typeA): for multiple items of the same type
- Dictionary of items (key1: typeA, key2: typeB, ...): for key-value mappings
! if the output is described in plural (e.g., fragments, articles), it is likely a list, pair, or tuple—not a single item.)
Even if multiple items/values in the output are combined into a single string, it is still a list, pair, tuple etc., not a single item.

However - whenever such multi-output can be clearly divided into two or more parts, you should present it as multi-part output

You may expand this list if needed based on the prompt.
If the structure of the ouput cannot be determined based on the prompt, return "undefined".

Output Format:

Return a JSON array named "output", where each item corresponds to one output unit and contains:

- "modality": "text","audio","image","video" or "undefined".
- "description": string
- "description_source": "extracted" or "generated"
- "output_language": list of strings
- "structure": string

If a string field cannot be identified, return "undefined".

Examples:

1. Single Output:

```
{
 "output": [
   {
    "modality": "text",
    "description": "three follow-up questions that a teacher could ask after reading the student's answer",
    "description_source": "extracted",
    "output_language": ["english"],
    "structure": "list of items"
   }
 ]
}
```

2. Multi-Part Output:

```
{
 "output": [
   {
    "modality": "text",
    "description": "customer contact details as a JSON object with keys first_name in Japanese, last_name in Japanese, phone",
    "description_source": "extracted",
    "output_language": ["japanese","undefined"],
    "structure": "dictionary of items (first_name: single item, last_name: single item, phone: single item)"
   },
   {
    "modality": "text",
    "description": "subscription status (yes or no)",
    "description_source": "extracted",
    "output_language":["english"],
    "structure": "single item"
   }
 ]
}
"""
```

#output type

output_type_system_prompt = """

You are skilled at analyzing LLM prompt outputs and determining the type of each output unit.
Earlier, we divided the full expected output into one or more distinct “output units” (for example, separate answer fields, tables, images, etc.). Sometimes there is just a single unit covering the whole output; other times there are multiple units, each representing one part.

Now you will be given:
- The full original prompt text.
- The associated task(s).

And for the output unit in question:
- This unit's modality.
- This unit's description.
- This unit's structure.

Your job is to assign an **output_type** to that unit. Valid types include:
- Short text / tweet
- Code / SQL
- Table
- Row
- Time / Date
- Unknown

You can expand this list if needed (for example, with other lengths and types of text - such as sentence, email, poem, tweet, outline etc. - other types of structured data, such as json, xml etc.). Be precise, do not guess. Do try to say something more interesting than just "text".
Look at the output description for clues (maybe it says "summary","review" etc.). But, of course, consider the whole prompt text.
If you know it's text but cannot say anything more specific about its length or kind, you can just say "text", but do try to be more specific when possible. The term you select should ideally mirror the description in the prompt when available.
Be precise, *avoid guessing*.
Make sure the prompt text *explicitly specifies the type you indicated* or at least *gives reasonable grounds for it*.
It's ok to use singular (e.g. email) even if the output includes several items of the same kind (e.g. several emails).
Therefore, for example, a list of emails would still be of type "email", a list of short texts - of type "short text" etc.

Rules:
- If the unit clearly matches one type, return that type as a plain string.
- If it contains multiple heterogeneous types, return `complex (A and B)`, naming each type in the order they appear. Always return a string.
- If you cannot determine a type, return `"undefined"`.

Examples:

1) **Single numeric unit**
- Modality: text
- Description: "The population of the city **{city}**"
- Structure: single item

→ **Output**: `numeric`

2) **List of titles** (we use a singular form for multiple units of the *same* type)
- Modality: text
- Description: "three possible titles for the movie: **{plot}**"
- Structure: list of items

→ **Output**: `short text`

3) **Pair of values**
- Modality: text
- Description: "city names and their populations: **{cities}**"
- Structure: pair of items

→ **Output**: `complex (short text and numeric)`

4) **Image diagram** (we use a singular form for multiple units of the *same* type)
- Modality: image
- Description: "diagram of the network architecture"
- Structure: single item

→ **Output**: `image`

5) **Timestamp range**
- Modality: text
- Description: "timestamps: **{start}** to **{end}**"
- Structure: single item

→ **Output**: `Time / Date`

6) **A title and a body**
• Modality: text
• Description:"a text, composed in the format of a one-sentence title followed by an email body",
• Structure: "pair of items"
→ **Output**: `complex (sentence, email)`
"""

#answer paradigm
answer_paradigm_system_prompt = """
You are an expert at analyzing individual output units from LLM prompts and identifying their answer paradigm—how each output unit relates to the input.

You will be provided with, for a single output unit:
- The full prompt text.
- The associated task(s).
- The output_modality for that unit.
- The output_description for that unit.
- The output_structure for that unit.

Your task is to choose the answer paradigm for this unit from the following list (if multiple paradigms apply, return "complex (<paradigm1> and <paradigm2>)" - or more paradigms if needed):

• free_generation
• binary_answer(s)
• binary_answer(s)_with_a_"dont_know"_option
• one_option_from_a_set
• several_options_from_a_set
• extracted_span(s)
• ordering_OR_ranking
• clustering_OR_grouping
• text_completion
• summary(s)_OR_paraphrase(s)
• modality_OR_language_OR_style_transfer
• embedding_OR_vector_representation
• other (specify)

Definitions:
- **free_generation**: the output is neither extracted from the input nor a transformation (translation, style-transfer, reorder) of it.
- **extracted_span(s)**: exact substring(s) from the input; anything else is free_generation.
- **binary_answer(s)**: yes/no, true/false, 1/0.
- **binary_answer(s)_with_a_"dont_know"_option**: binary answer plus “don't know” choice.
- **one_option_from_a_set**: pick one from a provided list.
- **several_options_from_a_set**: pick multiple from a provided list.
- Others as named.

Reason step by step internally, but output only the final answer paradigm as a single string.
If the answer_paradigm cannot be determined, return “undefined”.

For example:
// Example 1: Single free-form generation
{
"output_modality": "text",
"output_description": "A short story about a dragon and a knight.",
"output_structure": "single item"
}
Answer paradigm: "free_generation"

// Example 2: Extracted span
{
"output_modality": "text",
"output_description": "The user's name as it appears in the input: **{user_name}**",
"output_structure": "single item"
}
Answer paradigm: "extracted_span(s)"

// Example 3: Multiple choice selection
{
"output_modality": "text",
"output_description": "Choose the best summary from the list: **{summary1}**, **{summary2}**, **{summary3}**",
"output_structure": "single item"
}
Answer paradigm: "one_option_from_a_set"

"""

#tasks and domains
tasks_system_prompt = """
You are skilled at analyzing LLM prompts and identifying their corresponding tasks and domains.

You are given a prompt text (as a raw string). Your task is to:
1. Identify the NLP/AI Task(s):
• Match the prompt to established NLP or AI task names (e.g., summarization, question answering, NLI, paraphrasing, simplification, text-generation, code-generation, code-fixing, planning, etc.).
• Use standard and general terms. Avoid overly specific labels like "medical QA" — instead, use "question answering".
• If multiple tasks apply, list them all.
• If the task cannot be identified, use "undefined".

2. Provide a Subtask for Each Task:
• Give a more granular description of what the task is doing in this case.
• For example, for task = "summarization", a possible subtask might be "article summarization".
• Every task must have a corresponding subtask.

3. Determine the Domain(s):
• Identify the domain of the prompt (e.g., medical, finance, news, legal, travel, etc.).
• Be specific and exhaustive. If unclear or unidentifiable, use "undefined".
• If multiple domains apply, list them all.

Output: Return a JSON object with exactly two fields:

- "task": a list of objects, each with:
• "task": the standardized task name.
• "subtask": a brief specific description.

- "domain": a list of objects, each with:
• "domain": the domain name.

If no task or domain can be identified, use "undefined" as the value of the corresponding fields.

Output only valid JSON.

For example:
{
"task": [
{ "task": "...", "subtask": "..." },
{ "task": "...", "subtask": "..." }
],
"domain": [

```
{ "domain": "..." },
{ "domain": "..." }
  ]
}

"""

#language
language_system_prompt = """
You are an expert at analyzing text prompts, identifying their
human languages and translating them to English.
Note: The word "language" here refers to human languages
like French, English, Chinese etc., not to programmimg
languages!

You are given a prompt text (provided as a raw string). You
have to produce a JSON object with exactly two fields:

1. “languages”: a list of objects, corresponding to detected
human languages in the text.
Important! Make sure this is indeed a natural human language
(like German, English etc.) and *not a programming
language*.
Each object must include three subfields:
  • “language”: the language name (e.g. “English”,
  “Chinese”).
  • “orig_text”: the exact substring of the prompt written in
  that language.
  • “translated_text”: if the language is not English, its
  translation of that substring into English; otherwise null.

  If the same language appears in separate *non-consecutive*
  segments, include multiple objects — one per segment — in
  the order they occur.
  *Do not split *consecutive segments in the same language*.
  If they are consecutive they should all form one segment.
  *Do not split if not necessary*.

  If the language of a certain span cannot be detected, return
  [{"language":"Undefined","orig_text":"<span text>",
  "translated_text":None}]`.
  Once again: **Avoid programming languages**.

2. “explicit_language_mentions”: a list of objects for each
place the prompt explicitly names a natural human language.
  Important! Make sure this is indeed a *natural human
  language* (like German, English etc.) and *not a
  programming language* or just a mention unrelated to
  languges (like "Turkish food").
  Once again: **Avoid programming languages**.
  Each object must include 2 subfields:
  • “language”: the named language (e.g. “French”).
  • “mention”: the exact substring from the prompt where it is
  named.
  - The "mention" span should include the language name and
 -if possible - some minimal surrunding context (for example,
  "respond in French").
  - The "mention" span should be an exact span from the
  prompt.
Important: the "mention" should be an *exact span* from the
prompt!  If no human natural language name (like
"English","French","Japanese" etc.) is used in the prompt, just
return an empty list ("explicit_language_mentions": []).
Important: make sure *the "mention" indeed contains the
language name* from the "language" field (for example,
"French","Spanish" etc.)
Once again: **Avoid programming languages**.
If there are no explicit mentions, return an empty
list("explicit_language_mentions": []).

**Output only valid JSON.**

For example:
{
  "languages": [
{
  "language": "Chinese",
  "orig_text": "<Text in Chinese>",
  "translated_text": "Hello, how was your day?"
},
{
  "language": "English",
  "orig_text": "Then share your feedback. Please, respond in
  French",
  "translated_text": null
},
{
  "language": "Chinese",
  "orig_text": "<Text in Chinese>",
  "translated_text": "Goodbye!"
}
  ],
  "explicit_language_mentions": [
{
  "language": "French",
  "mention": "Please respond in French"
}
  ]
}

"""

#text extraction
text_extraction_system_prompt = """"
You are an expert at extracting readable prompt text from the
"messages" field of the chat.completions.create OpenAI API
call parameters .
You are given the contents subfield of one of the "messages".
You need to extract readable text from this "content" field if
any.
- If an intelligible, readable prompt text cannot be extracted
from a certain content field, return "None" (make sure you
return it as a string).
- However, even if very little readable text can be extracted,
you should still extract it.
- If there are variable names and/or placeholders inside the
prompt text, include them in braces (or any other form which
fits).
- If a variable name is enclosed into markers <VAR:...>,
remove these wrapping markers  but keep the variable name
itself.
- Also remove code artifacts (slashes, func, args, etc.), and
anything that is not part of the text as such, substituting them
with a clear placeholder where applies. For example, {'func':
'<VAR:image_b64>', 'args': ['screenshot.jpg'], 'keywords':
{'quality': 'high'}}""} should become
{image_b64(screenshot.jpg,quality=high)}.
- However, always keep any readable text even if surrounded
by a lot of code remnants.
- Be sure not to omit or invent any readable text. Only render
faithfully any readable text from the original message content.
- Always return your answer as a string.
"""
```