Measuring an LLM's Proficiency at using APIs: A Query Generation Strategy

Ying Sheng[*][†]
Google Research
Mountain View, CA, USA
yingsheng@google.com

Sudeep Gandhe[*]
Google Research
Mountain View, CA, USA
srgandhe@google.com

Bhargav Kanagal
Google Research
Mountain View, CA, USA
bhargav@google.com

Nick Edmonds
Google Research
Mountain View, CA, USA
nge@google.com

Zachary Fisher
Google Research
Mountain View, CA, USA
zachfisher@google.com

Sandeep Tata
Google Research
Mountain View, CA, USA
tata@google.com

Aarush Selvan
Google
Mountain View, CA, USA
aarushs@google.com

Abstract

Connecting Large Language Models (LLMs) with the ability to leverage APIs (Web Search, Charting, Calculators, Calendar, Flight Search, Hotel Search, Data Lookup, etc.) is likely to allow us to solve a variety of new hard problems. Several research efforts have made this observation and suggested recipes for LLMs to emit API calls, and proposed mechanisms by which they can generate additional text conditioned on the output for the API call. However, in practice, the focus has been on relatively simple slot-filling tasks that make an API call rather unlocking novel capabilities by combining different tools, reasoning over the response from a tool, making multiple invocations, or complex planning. In this paper, we pose the following question: what does it mean to say that an LLM is proficient at using a set of APIs? We answer this question in the context of structured APIs by defining seven capabilities for API-use. We provide an approach for generating synthetic tasks that exercise each of these capabilities given only the description of an API[1]. We argue that this provides practitioners with a principled way to construct a dataset to evaluate an LLM's ability to use a given set of APIs. Through human evaluations, we show that our approach produces high-quality tasks for each of the seven capabilities. We also describe how we used this approach to on-board new API and create principled evaluation sets for multiple LLM-based products.

[*]Equal Contribution.
[†]Corresponding author.
[1]Code available at http://goo.gle/qgen_for_tool_use

CCS Concepts

• **Human-centered computing → Natural language interfaces**; • **Computing methodologies → Natural language generation**; **Learning from demonstrations**.

Keywords

Tool-Use, LLMs, Synthetic Data Generation, Benchmarking

ACM Reference Format:
Ying Sheng, Sudeep Gandhe, Bhargav Kanagal, Nick Edmonds, Zachary Fisher, Sandeep Tata, and Aarush Selvan. 2024. Measuring an LLM's Proficiency at using APIs: A Query Generation Strategy. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '24), August 25–29, 2024, Barcelona, Spain.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3637528.3671592

1 Introduction

Large Language Models (LLMs) have demonstrated impressive capabilities [1, 5] on a variety of natural language tasks including creative writing, summarization, code generation, and reading comprehension. However, they do not possess certain capabilities that are present in other tools – arithmetic, unit-conversion, charting and visualization, database lookup for world events, weather, currency exchange rates, bus schedules, etc. Several researchers have pointed out that endowing LLMs with the ability to make API calls and incorporating the result to produce a final response could unlock several new applications.

Recent papers have proposed multiple ways to connect LLMs to tools to accomplish complex tasks. The ToolFormer [18] paper argued that we can go beyond tool-use limited to task specific settings like open-book QA [13]. The key insight was to use a small number of human-annotations to teach the LLM to generate a tool call when appropriate. Such an LLM can be paired with a mechanism to detect when a tool call is emitted, and replace the span with the response from the tool, and continue the process of conditional decoding. This results in outputs like the following:

```
TEXT: Jack had 8 pens and Mary had 5 pens. Mary gave
```

```
3 pens to Jack. How many does Jack have now?
ANSWER: Jack has [Calculator(8 + 3) --> 11] pens.
```

In the above example, the LLM emits a call to the Calculator API within the square brackets. Conditional decoding continues after the result from the API replaces the text of the invocation. Several efforts have built on this idea, exploring the design space around how to represent the set of available tools, how to generate fine-tuning data for each tool, whether the LLMs can plan and execute complex tasks, etc. See Section 5 for a detailed discussion.

Given the exciting potential of connecting LLMs with APIs to accomplish more complex tasks, it is not ideal that much of the evaluation in this space has reused existing benchmarks designed for mathematical reasoning [14] and question-answering [8, 16]. In this paper we pose the following question – what is a principled way to evaluate the ability of an LLM to use a given set of tools/APIs to accomplish complex tasks? To make this tractable, we make the following choices: first, we limit ourselves to structured APIs. This includes tools with simple input-output specifications like calculator, calendar, weather, factual QA, translation service, etc.[2] Second, we focus on asking what new capabilities the LLM can unlock beyond providing a natural language interface to invoke the appropriate tool (commonly referred to as single-shot slot-filling).

Here, we identify seven tool-use capabilities that indicate an LLM's proficiency with a tool or set of tools. We start with single-shot-slot-filling ("can the model make the correct API call?") to reasoning-over-outputs ("can the model perform simple reasoning over the outputs of the API call to solve the given task?") to more complex tool-chaining ("can the model combine two or more tools to solve a task?"). Figure 1 shows an aspirational example of how an LLM that is proficient across these capabilities and connected to public APIs (viz., web search, real-estate search, maps search etc.) and perhaps private enterprise APIs (viz., a rental estimator, a home pricing tool) can solve a complex query[3] like finding an investment property for a user given a set of constraints.

In a practical setting, when building a product one wants to evaluate the performance of the LLM on a specific set of tools that are going to be available in production. Just because an LLM has demonstrated the ability to use a calculator, does not automatically mean it is going to be proficient with a flight-search tool. A reasonable evaluation set at the very least would highlight if the LLM knew how to take advantage of all the parameters supported in the API. A good evaluation set would tell us if connecting the LLM to the API unlocked any new capabilities. A task similar to the one described in Figure 1 will make an ideal candidate for a good evaluation set.

One approach to create such an evaluation set would be to request expert human annotators to author tasks that exercise tool use. But this is costly and time-consuming. Another approach is to collect example tasks from actual usage of the product by real users. But this may result in a set of tasks that is skewed towards easier tasks than the complex ones. Also in practice, we may want to evaluate for tools that are not yet launched in production. We need a scalable and controllable approach that can create an evaluation set.

We describe an approach to generating such synthetic evaluation set that allows anyone building a tool-enhanced LLM product to generate an evaluation set that tests all these capabilities along with diversity metrics that guarantee coverage within each capability. This approach simply uses the documentation provided for an API call to generate queries, and does not require any other human inputs.

We expect our approach will be useful for the scenario where an LLM is connected to a modest numbers of tools (say ten to hundred) and is expected to be proficient with them. This is in contrast to focusing on the retrieval problem, where there is a larger pool of tools [15] and overlapping services across tools (e.g. multiple hotel-search tools with a preference policy) and the focus is on picking the one right tool and setting the parameters appropriately. This is also in contrast to general planning-focused benchmarks that try to understand if a model can complete a goal like shopping for a specified object on a website [21]. Our work is not aimed at any specific task like open-book QA or web-navigation. Instead we're interested in arriving at a principled way to understand the proficiency of an LLM when it has access to structured APIs and the kinds of capabilities that are unlocked. We would like to point out that we are *not* proposing algorithms to better pre-train or fine-tune LLMs to leverage a set of APIs. We're also not recommending any algorithms for automatically generating fine-tuning examples given a set of tools, while that may be a natural extension from some of this work. Our focus in this paper is to propose a principled mechanism to generate API-specific datasets to understand an LLM's proficiency with a given set of tools.

We make the following contributions in this paper:

- We argue that it is important to have a principled way to understand if an LLM is proficient at using a given set of APIs. We define seven capability dimensions along which to measure an LLM's proficiency.
- We describe a technique to generate synthetic tasks for each of these capability dimensions given only the description

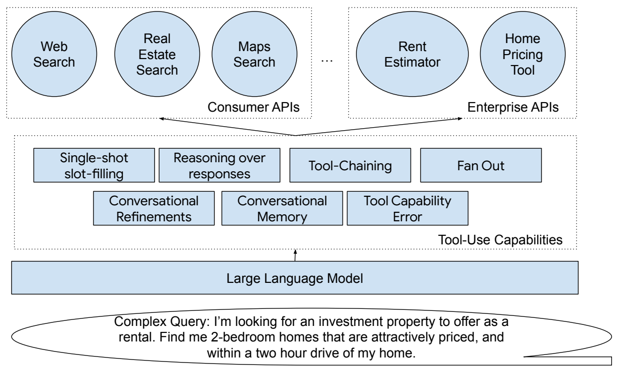


Figure 1: Connecting LLMs to APIs can unlock abilities to solve complex tasks. "looking for investment properties" requires combining calls to maps search, real-estate search, a home pricing tool, and a rent estimation tool.

[2]This excludes from our consideration complex tools like the Python runtime, a database engine, etc. Understanding the ability of an LLM to generate python code or SQL statements is a rich research area that goes well beyond understanding relatively simple structured APIs.
[3]We use tasks and queries interchangeably.

Capability	Input	Output	Code	Dialog
Slot-filling	✓			
Reasoning over responses	✓	✓	✓	
Tool-chaining	✓	✓	✓	
Fan-out	✓		✓	
Conversational refinement	✓			✓
Conversational memory	✓			✓
Tool capability error	✓	✓		

Table 1: Each tool-use capability relies on one or more fundamental abilities of the underlying LLM: understanding the input schema for an API, understanding the output schema, ability to generate code, and the ability to understand dialog across multiple turns.

of an API. We describe design decisions around leveraging chain-of-thought, few-shot-prompted critique models, and rejection sampling. We describe ways to measure the diversity/coverage of the data generated along multiple axes.

- Present an evaluation demonstrating that this technique can be used to generate high-quality tasks. We also describe case studies where this technique was used to generate synthetic evaluation data to onboard tools for beta products that connect an LLM with structured APIs.

2 Capability Dimensions

We describe the seven capabilities we identified in this section along with examples. Each of these capabilities requires a different combination of basic skills in the LLM across: (a) understanding the input schema of the API, (b) understanding the output schema of the API, (c) ability to generate code, and (d) ability to understand dialog. Table 1 shows a summary of these seven capabilities. Within each capability bucket, we describe how to measure the diversity of generated data.

(1) Single-shot slot-filling: This is the most basic capability, where the user simply provides all the information required to invoke the API in a single turn. For a flight-search API, this may be something like "find me a one-way flight from SFO to NYC leaving on 23rd and returning on the 25th of November". Many papers, as well as some commercial features like Function Calling [7] limit themselves to this capability. A diverse set of tasks here would exercise all the parameters in the API. A full-featured commercial flight search API can have 20-30 optional parameters. Ideally the tasks would exercise several different parameter combinations as well. Measuring the fraction of parameters that the generated tasks exercise as well as the parameter combinations should give us a sense for the space covered by a set of queries for this capability.

(2) Reasoning over outputs: This capability dimension requires the LLM to reason over the responses from a tool in order to answer the question. Continuing with the example of the flight-search API, Consider the following example: "Find me a flight to NYC for this Thursday that lands before 5pm." Typical flight-search APIs often do have a parameter to express restrictions by departure time, but no parameters to limit flights by arrival time. Solving this task would require making a call to find all flights leaving on Thursday, and then filter the results by those that arrive at one of the destination NYC airports before 5pm. A diverse dataset here would require reasoning over various attributes present in the returned objects from the API.

(3) Conversational refinements: This is a dialog-centric capability, where the user asks for something that requires changing the API invocation. Consider the following example: "Prompt1: Find me flights from SFO to NYC leaving on 23rd and returning on the 25th", "Prompt2: What if I came back on the 27th?" Prompt1 by itself might require an API call, however, prompt2 by itself does not have all the information required to issue a call to the flights API, but given the context of Prompt1, it should be able to infer the right values for the parameters like departure date (23rd) and return date (27th). A diverse dataset here would cover refinements that correspond to changing the value for each of the parameters in the API.

(4) Tool-chaining: This capability allows an LLM to call tool A, and use some element from that result in order to call tool B. Consider the following query: "I would like to visit the Grand Canyon. Find me flights for a 5-day trip leaving on the 15th". This might require first calling a "Web Search" tool to find the airport that is closest to the Grand Canyon, and then calling the flight-search API with that airport as the destination airport. A diverse dataset here would of course include the target tool both at the start and end of the tool-chain. Ideally, the dataset would exercise different combinations of tools to produce a tool-chain.

(5) Fan-out: This capability checks that the LLM can invoke the API multiple times when necessary, and summarize the results across the invocations. "I'm thinking of a weekend trip leaving on Friday and returning on Sunday. I'm considering NYC, Chicago, or Boston. Which is the cheapest option?" This would require that the LLM invoke three API calls, one to each of the cities the user is considering, and reason over the returned responses to figure out which choice might be the cheapest.

(6) Multi-turn memory: This capability exercises the model's ability to truly be a conversational tool and retain memory from previous turns of conversation. This is a more general version of the "Conversational refinements" capability, but in this case, we relax the restriction that the new task ought to exercise the same tool. "Prompt 1: what islands should I visit my first time in hawaii ? Prompt 2: which should I visit and for how long if I'm staying 5 days ? Prompt 3: how do I get from oahu to maui Prompt 4: find flights for this itinerary."‘ Prompt 4 would require retaining the information from Prompts 1 - 3 in order to make the right set of calls to the flights API.

(7) Error Handling: This capability tests the model's ability to understand what is possible and what is not possible to solve with the given API. This capability requires that the model understands when a request exceeds the API supports and responds appropriately. Consider the example "Find flights to NYC with vegetarian food options." Ideally, the model should

respond saying something along the lines of "There is no way to search for flights with constraints on food options."

Note that these capabilities are not mutually exclusive. Consider a prompt like "how far is each of these hotels from the nearest subway station". This combines fan-out, tool-chaining, and likely multi-turn memory since the user is referring a list of hotels that was the result of a previous invocation of a tool.

3 Generating Synthetic Tasks

A key design consideration is to generate tasks given just the description of the tool. We assume that we are given the OpenAPI specification for a target tool – see Figure 2 for an example. The specification contains the input and output schemas as well as a natural language description of each field. Given this API specification, we can generate synthetic tasks corresponding to each of the capabilities described in Section 2 as follows.

We take advantage of three key ideas: (1) We want to avoid any manual work to be done for each tool, so the approach ought to be able to produce reasonable tasks based on the description of the tool alone. We start with a few-shot prompted large model, where each example consists of an API spec and sample queries written by an engineer. We then simply append the API for the target tool to this prompt, and ask the LLM to generate queries. (2) We leverage chain-of-thought style prompting to guide the LLM

```
openapi: "3.0.0"
info:
  title: flights_tool
  description: Flights tool to search and get booking links.
paths:
  /search:
    get:
      operationId: search
      parameters:
        - name: origin
          in: query
          description: The location where the trip starts.
          schema:
            type: string
          required: false
        - name: destination
          in: query
          description: The final destination of the trip.
          schema:
            type: string
          required: false
        - name: earliest_departure_date
          in: query
          description: Filter for the earliest (or only)
                       departure date in YYYY-MM-DD format.
          schema:
            type: string
          required: false
        - name: earliest_return_date
          in: query
          description: Filter for the earliest (or only)
                       return date in YYYY-MM-DD format.
          schema:
            type: string
          required: false
      responses:
          '200':
            description: The result of flights search.
            content:
              application/json:
                schema:
                  $ref: "#/components/schemas/SearchResult"
components:
  schemas:
    SearchResult:
      ...
      ...
```

Figure 2: Example partial OpenAPI specification for a flights tool. Schema for the response is specified in a SearchResult object (not shown here for brevity).

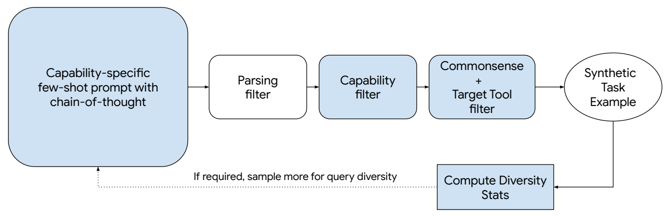
```
You are given an OpenAPI specification of a tool in YAML. Your task is to come up with questions in
natural language form that can be answered by invoking the tool.For each question, list the parameters
in the API that it populates, along with corresponding values.

OpenAPI spec:
{{{flights_yaml}}}
Assume that today is August 1, 2023.

Write 3 questions.  After completing, end with '-----' on a new line.
# Required parameters: 'destination'. So, destination must be present in every
question.
#1: [origin=SF; destination=NYC] book flights from SF to NYC for a 3 day business trip.
#2: [origin=DC; destination=london; earliest_departure_date=2023-08-22;
earliest_return_date=2023-09-03; seating_classes=premium economy; one_way=False]
round trip-ticket from DC to london in premium economy leaving on Aug 22 and
returning on Sep 3rd.
#3: [origin=LAX; destination=Portland; earliest_departure_date=2023-09-07;
earliest_return_date=2023-09-10; num_adult_passengers=2; num_infant_in_lap_passengers=1;
include_airlines=united] united flight for 2 adults and 1 lap infant from LAX for a
3 day trip to Portland starting Sep 7th.
-----

OpenAPI spec:
{{{hotels_yaml}}}
Assume that today is August 1, 2023.

Write 3 questions, one per line. After completing, end with '-----' on a new line.
# Required parameters: 'query'. So, query must be present in every question.
#1: [query=hotels in seattle; check_in_date=2023-09-01; check_out_date=2023-09-05] find me
hotels in seattle from sep 1-5.
#2: [query=hotels in new york near times square; check_in_date=2023-09-06;
length_of_stay=3] book a hotel in new york near times square for 3 nights checking in
tomorrow evening.
#3: [query=hotels in london; check_in_date=2023-08-22; check_out_date=2023-09-03] hotels in
london checking in Aug 22 and until Sep 3rd.
-----

OpenAPI spec:
{{{tool_yaml}}}
Assume that today is {{{date}}}.
Write 10 questions, one per line. After completing, end with '-----' on a new line.
# Required parameters:
```

Figure 3: Example generator prompt for single-shot-slot-filling with placeholders for the YAML spec for the flights and hotels tools.

to generate tasks can be reasonably fulfilled by the API and avoid generating irrelevant tasks such as asking for a car rental when presented with a flight-search tool. We force the chain-of-thought to have a carefully prescribed structure, so that we can discard generated tasks that may not belong to the capability (explained below with examples). (3) Finally, we build a set of critic models that can evaluate these generated tasks and discard ones that do not satisfy constraints.

Figure 3 shows a pared-down version of our generation prompt for the single-shot-slot-filling category. Note that before generating a task, the LLM is instructed to decode a chain-of-thought-like string within square-brackets. For the single-shot slot filling category, this is simply the parameters that it is going to build the query around. For this category, we parse the k=v formatted chain-of-thought and verify that each of the parameters is indeed present in the tool specification using simple python code. This helps us discard any generations that contain hallucinated parameters. Requiring similarly structured chain-of-thought in other capabilities allows us to discard low-quality generation.

We write a critic prompt for each of the capabilities to identify and discard other low quality generations. These are built by having a human inspect a set of generated queries, and use the bad ones as negative examples in the critic prompt. Figure 5 shows an example. Most of the errors we noticed were inconsistent values between the chain-of-thought and the actual generated query. We use two additional critics: a commonsense critic that rejects any query that is deemed "unrealistic", and a target-tool filter that checks to make



Capability-specific few-shot prompt with chain-of-thought
Parsing filter
Capability filter
Commonsense + Target Tool filter
Synthetic Task Example
Compute Diversity Stats
If required, sample more for query diversity

Figure 4: Sketch of the synthetic data generation process: first, we sample from the LLM using a capability-specific prompt that takes the tool API as an input. The generated samples are sent through three filters: first, a parsing filter that checks that the sample matches the expected format; second, a few-shot prompted capability-filter that verifies if a generated sample requires the use of the target capability; and third, a commonsense filter that checks to see if the target tool is required to answer the given query.

```
Your task is to judge whether a query can be answered by calling a tool once, and whether
the parameters are properly populated. Here are a few examples.

input: [origin=SF; destination=NYC] book flights from SF to NYC for a 3 day business trip.
answer: yes, this query will require one flight search call. The parameters specified are
origin and destination, and they are assigned correct values from the query.

input: [query=hotels in new york near times square for 3 night; check_in_date=2023-09-06;
length_of_stay=3] book a hotel in new york near times square for 3 nights checking in
tomorrow evening.
answer: no, 3 night is already specified in length_of_stay, should be removed from query.

input: [origin=hong kong; destination=sydney; max_stops=1] search for flights from hong kong
to sydney nonstop.
answer: no, max_stops should be 0 for 'nonstop', but the parameter is set to 1.

Here's a new input:
input: {{{query}}}
answer:
```

Figure 5: Example critic prompt for single-shot-slot-filling.

sure that the query requires the use of the tool in question to produce a good answer. Figure 4 sketches the overall flow.

3.1 Measuring Diversity of Generated Data

We expect a synthetic data generation approach like ours to be used in two ways:

(1) *Fine-tuning:* Generate training examples to fine-tune an LLM to help onboard new tools. In particular, on queries where the model is performing poorly.
(2) *Evaluation:* Generate an evaluation dataset to faithfully evaluate the LLMs tool-use proficiency. This is especially useful when there is no traffic from real users.

These examples are of the form of (query, code) pairs, i.e., given a query, the model needs to produce code that makes tool calls to fulfill the query [4]. For effectively addressing both applications, understanding if the generated dataset is diverse (in terms of both query and code) is particularly important. Since our objective is tool use via code, we claim that we do not need to measure diversity at the level of parameter values. Instead we only need to measure diversity at the level of parameter names. Also note that we are not interested in metrics that capture language diversity (e.g., paraphrasing) since LLMs today are quite proficient at language comprehension. We design the following metrics:

Parameter Coverage. The most straightforward measure for diversity is *parameter coverage*, i.e., the fraction of API parameters that are covered by a set of generated queries. To do so, we need to predict the parameters that are specified by a given query. For certain capabilities like single-shot-slot-filling and fan-out, these can be directly read off the chain-of-thought that was used to generate the query. For other capabilities, we devise a *parameter extraction* few-shot prompt that takes in an input query, the API info, and returns all the parameter names and values referenced by the query. By aggregating these across the generated queries, we can estimate parameter coverage.

Parameter Combination Coverage. We also extend the same idea to measure diversity across parameter combinations. For example, a flights API call with (origin, destination, departure_time) captures the user intent on leaving at the specified departure time, whereas a flights API call with (origin, destination, arrival_time) fields captures the intent where the user wishes to be at the destination at the specified arrival time. Across all the generated queries, we compute the number of unique parameter combinations as another measure of diversity.

Capability-Specific Metrics. We also want to exam if different kinds of code may be required to solve the task within a capability. The challenge is to be able to do this without looking at the code itself. We augment the chain-of-thought to add *capability decorators* to parameters on which the capability is applied. For instance, for the real estate query *show me single family homes in Los Gatos and Saratoga*, we are fanning out over the location attribute, so we add `location(fanout)` to the CoT. Similarly, for the query *show me directions from the FB headquarters to the cheapest open house in Menlo Park*, which chains a real-estate search tool and a map-search tool, we add `chaining(street_address, location)`. These decorator tags will allow us to capture a proxy for code complexity in a straightforward manner. Within each capability, we compute the number of unique decorators across all the generated queries. See Table 7 for other capability-specific decorators. In Section 4 we report a few of the above diversity metrics on our generated datasets.

4 Evaluation

The basic way to understand if this approach is effective is to generate queries in each of these buckets for unseen APIs and perform human evaluations. In this section, we first measure query precision using queries generated for 3 APIs: a real estate search API, a recipe search API and a business search API. We pick one tool with the most parameters to evaluate the diversity of the queries. We then measure each component's contribution in ablation study section. Finally, we do a naive baseline to confirm the following hypothesis: without careful capability-specific prompts, merely prompting the LLM to generate queries results in the output being dominated by single-shot slot-filling queries.

In all the experiments below, we use instruction tuned PaLM 2-L [1] for both query generation and query criticism. For query

[4]The focus of the current paper is on generating queries; the code is assumed to be authored by humans.

generation, the LLM is called with temperature = 1.0 to encourage variety of queries. For query evaluation, temperature = 0.0 is used to get the most confident result.

4.1 Query Quality

As described in section 3 , we first generate query set N for the given tool APIs and capability buckets. We then use the query critics to filter low quality queries and get a query set N_{filtered}. Lastly, we use an LLM to retrieve the parameters used by these queries to compute the diversity metrics.

We generate 10 queries per capability buckets for the 3 APIs and manually assessed all the 210 queries. The queries were assessed by engineers familiar with Python and the use of these tools. The criteria for a good query is twofold:

- The query is well-formed: it is a fluent sentence that makes sense and uses the target tool.
- The query belongs to the target capability bucket.

See Table 2 for the per-bucket precision. Precision is defined as

$$\text{precision} = \frac{\text{queries with positive human rating}}{\text{total number of queries generated}}$$

Among these 7 buckets, some are easier than others. For example, single-shot slot-filling is at 90% precision, while 'reasoning over outputs' is only at 67% indicating that it is harder to generate high-quality queries here. See Table 6 for some examples of queries rated positive and negative queries for each of the buckets. Here are 3 error types we observe:

- The LLM does not fully understand the API definition. The negative queries for 'within tool refinement' bucket in Table 6 asked for homes with 2 parking spots, but number of parking spots is not available in the API's response schema. Another example is the 'fan-out' negative queries. The 'propertyType' parameter in the real estate search API can take a list of values, so the search for single family houses and town house can be full-filled in a single API call instead of a fan-out pattern.
- The LLM misunderstands the capability definition. The 'multi-turn memory' bucket requires queries refer back to previous dialog turns. The negative query in Table 6 for 'multi-turn memory' explicitly says 'Austin' instead of words like 'there'.
- Query is too broad to be realistic. The negative query for 'tool-chaining' falls in this category. There are a lot of rental properties in Seattle downtown, asking for transit available for each of them is unrealistic.

4.2 Synthetic Query Diversity

Query diversity is measured by examining coverage over the parameters in the APIs. This coverage need not be uniform. In fact, it probably should not be. It is natural that some parameters occur more frequently than others. For example, the `query` field is usually present in the real-estate API; it is the main parameter. Fields such as `hoaMax` are used less frequently. Defining the target distribution depends on the task. In some cases, generating synthetic data that matches real user queries may be optimal. In others, like generated queries to turn into training examples, favoring more complex queries may yield better model performance. The combination of our query generation and critic models allow us to adjust the query diversity as desired. In Appendix C we show that our approach can match query-diversity of human-written queries.

Capability	Precision
Single-shot slot-filling	0.90
Reasoning over outputs	0.67
Within tool refinements	1.00
Tool-chaining	0.73
Fan-out	0.87
Multi-turn memory	0.67
Error handling	0.83

Table 2: Query quality: we manually evaluated 210 post-filtering queries evenly distributed across 3 APIs and 7 buckets and report the per-bucket precision

Another measurement for diversity is the number of unique parameter combinations. We experiment with the real estate API with 20 parameters. With 53 human queries, there are 41 unique parameter combinations. With synthetic approach, we can generate many more queries. In our experiment, we generated 187 queries with 125 unique parameter combinations. We can grow the unique combinations quite easily by generating more queries. Generating large synthetic sets like this allow us to test model performance on infrequently used parameters in the API.

4.3 Ablation Study

To compare the effectiveness of each component in the query generation prompt, we evaluated two additional experimental setups in addition to the full pipeline Name (full):

- QGen-CoT: remove the per-query chain-of-thought for all the few-shot examples.
- QGen-CoT-Few Shot: remove the few-shot examples, only keep the per-category instructions.

Instead of evaluating the final precision of queries filtered by query critics, we use yield rate to measure the effectiveness of the query generation. It is defined as:

$$\text{yield rate} = \frac{\text{queries pass query critic}}{\text{number of queries generated}}$$

The reason is 1) the only way to evaluate filtered queries is by human evaluation, which is too expensive to do for all the experiments; 2) good query critics can mask the low quality in query generation, the precision of raw query set is a more direct indication of the query generation prompt's quality.

But the yield rate metric only works well when we have high quality query critics. So we first evaluate the query critic quality. As mentioned in section 4.1, we consider two aspects when evaluating queries: first, whether the query is well-formed, and second, whether the query belongs to the target capability bucket. To understand the quality of the two query critics, we generate 84 queries covering 3 APIs and 7 capability buckets with critic results. We then have humans evaluate these queries. The precision and recall of the query critcs are reported in (Table 3). The 'well-formed' critic has above 0.95 precision and recall. The 'belong to capability' critic have 0.85 precision and 0.83 recall. Both are reasonably reliable.

In (Table 4), we report the yield rate for all capabilities under the 3 experiment setups. We can conclude that adding few-shot exemplars along with chain-of-thought is usually helpful. But the effectiveness of adding these two components are different across the capability buckets. For easy buckets like 'single-shot slot-filling', the LLM can do a decent job in a zero-shot setup. This is consistent with our observations in Naive Approach. The LLM can also do a decent job for 'reasoning over outputs' and 'within tool refinements' in the zero-shot setup. In contrast, for the 'fan-out' category, CoT is extremely helpful. This is likely due to the clear structure of CoT, e.g. the chain-of-thought for query "Prompt 1: Find the cheapest business class flight from SF to Hawaii for each weekend in August." is `[earliest_departure_date=fanout:each weekend in August]`. The model can clearly understand a few parameters are chosen to fan-out over, i.e. have multiple values. For tool-chaining, the zero-shot setup (QGen-CoT-FS) is better than the few-shot without chain-of-thought setup (QGen-CoT). Our hypothesis is the example queries without chain-of-thoughts are not obvious to the LLM about why they are tool-chaining. Adding them becomes harmful for tool-chaining query generation.

	Precision	Recall	F1
Well formed?	0.96	0.97	0.96
Belongs to capability?	0.85	0.83	0.84

Table 3: Precision and recall of the two critics using human evaluation over 84 queries.

Capability	**Full**	**No-CoT**	**No-CoT-No-FS**
Single-shot slot-filling	**0.83**	0.81	0.79
Reasoning over outputs	**0.96**	0.95	0.77
Within tool refinements	0.88	**0.90**	0.86
Tool-chaining	**0.48**	0.12	0.37
Fan-out	**1.0**	0.27	0.18
Multi-turn memory	0.43	**0.45**	0.11
Error handling	**0.48**	0.04	0.07

Table 4: Ablation study comparing the full solution with two setups: dropping CoT; dropping both CoT and few-shot examples. We report the yield rate from query critics for each of the capability buckets.

4.4 Naive Approach

The simplest solution for this problem is to ask a language model to generate queries that require tool use without giving all the capability bucket definitions. If LLM can naturally generate queries spreading across different buckets, then we don't need the capability definitions to enforce the variety in synthetic queries. To confirm that the LLM cannot generate queries across all capability buckets, we prompt the instruction-tuned PaLM 2-L [1] and the next-generation LLM, Gemini XL [20], to generate 100 tool-use queries. We then filter the queries that fail critic model validation and assign capability buckets to the remainder. See Table 5 for the numbers of the queries in each of the 7 capability buckets.

Capability	**PaLM 2-L**	**Gemini XL**
Single-shot slot-filling	70	53
Reasoning over outputs	0	4
Within tool refinements	0	0
Tool-chaining	0	6
Fan-out	0	1
Multi-turn memory	0	0
Error handling	3	19

Table 5: Query distribution when prompting the LLM without capability definitions. We use PaLM 2-L and Gemini XL – both predominantly produce single-shot slot-filling queries.

For PaLM 2-L, the distribution is very skewed. Most of the queries generated are simple ones. Gemini-XL can generate a more diverse set of queries, but some capabilities like 'within tool refinements' and 'multi-turn memory' are empty. We present this as evidence that naively prompting a model may generate reasonable queries after filtering, but does not address all the capabilities we care about.

5 Related Work

Augmenting an LLM's knowledge with tools allows models to generalize beyond data present in the training set and potentially provide more up-to-date, higher quality results for queries well suited to being answered via existing APIs. Multiple approaches have been explored to extend LLM capabilities in this fashion. ReAct [21] builds on Chain-of-Thought using Thought-Action style generation then uses the trajectories generated by a large mode to fine-tune smaller modes. Extending reasoning to tool selection and treating tools as actions allows arbitrary APIs to be leveraged by an LLM. ToolFormer [18] fine-tunes an LLM with examples of text generated interleaved API calls. The decoder is modified so that after an API call, the remaining tokens are generated conditioned on the output of the API call. Models are carefully prompted to generate API calls and only a useful subset of these invocations are used for downstream fine-tuning. ToolLLM [17] uses a separate few-shot prompts for request generation and translating requests to a thought-and-action sequence. ToolBench is an instruction-tuning dataset generated using this approach. Gorilla [15] produces a similar dataset called APIBench. Instructions for calling APIs are generated using a recipe in SelfInstruct and the calls themselves are generated using a few shot model. Gorilla largely focuses on calling TorchHub, TensorHub, and HuggingFace APIs for classificaiton and seq→seq tasks. Each of these papers focuses on techniques for connecting LLMs to APIs like search, calculator, QA models, so that existing benchmarks can be re-used to measure progress. Calling more structured APIs is not discussed, and to our knowledge none of these papers really focuses on the question of how do we think about the LLM's proficiency with a specific set of tools in a principled way.

Some efforts like APIBank [10] have made progress on helping researchers understand tool-use proficiency across models. It provides a few hundred tool-use dialogues annotated with the appropriate tool call with a vocabulary of 73 commonly used APIs.

It tries to measure the LLM's ability to use the tools along three dimensions: Retrieve (does the LLM know to retrieve the correct tool that should be used for the given task?), Call (does the LLM know how to call this tool with the correct parameter values?), and Plan (for more complex tasks, does the LLM know how to plan a sequence of calls to APIs?). The paper divides up the space along two dimensions – the number of tools ("few" vs. "many") and the number of API invocations ("one" vs. "multiple").

Chameleon [11] proposes a program-synthesis approach to tool-use relying on few shot demonstrations and a library of ~9 tools. However, in contrast to our approach, they do not generate a tool-specific set of tasks to measure the LLM's ability. Instead, they rely on two existing benchmarks: ScienceQA and TabMWP which happen to be good tasks to measure if the LLM can use a reasonably combination of the 9 tools they chose. In that sense, our approach complements this paper in providing a way for researchers and practitioners to generate an evaluation set that exercises various capabilities across any set of tools specified.

API Bank [10] is a tool-use benchmark containing 53 commonly used API tools, and 264 annotated dialogues that encompass a total of 568 API calls. However, it does not measure complex capabilities over the variety of dimensions discussed here or propose quantitative measures of query diversity. It also cannot be customized for a specific set of tools. DS-1000 [9] is a benchmark for evaluating LLM code-generation capability in the context of data-science. The approach does not solve the problem of evaluating proficiency on a target set of tools. Furthermore, the capabilities required to solve general data science problems are different from the seven tool-use capabilities we describe in this work.

6 Case Studies

In this section we describe two products where we used our approach to generating synthetic data: (1) onboarding new tools to an AI-powered chatbot and (2) generating synthetic evaluation datasets for a generative AI powered Search experience.

6.1 Connecting Tools to an AI Chatbot

LLM-powered chatbots [2, 6] have rapidly evolved in the last year starting with conversational responses for creative tasks and simplified explanations to using tools [3] like maps, hotel search, etc. to accomplish complex tasks. Bard is currently focused on enabling high-proficiency use for dozens of tools helping users accomplish more complex tasks with these tools.

A key step to connecting a new tool to an LLM-powered conversational agent like Bard [2] is to provide several demonstrations of how to use that tool to include in the fine-tuning mixture. While there are several research efforts to automatically onboard a tool using techniques like self-instruction [15] and a multi-agent version of self-instruction [10] – we decided to focus on using the relatively proven practice of sourcing high-quality examples and including them in the mixture for fine-tuning.

In order to onboard a new tool (such as the flight search API), we first generated 10 queries in each of the 7 capability buckets subject to getting a diverse set of queries in each bucket. We then asked engineers to write demonstrations corresponding to each of these examples. The demonstrations were included in the supervised fine-tuning (SFT) mixture for training [12]. The resulting model was evaluated for its ability to use tools using a proprietary evaluation dataset constructed through user studies. In this engagement, we did not have the synthetic task generation approach ready by the time the evaluation set was being determined. Our approach has been used to generate hundreds of tasks in the more complex capability buckets, and tool-use demonstrations have been written for each of these tasks, resulting in significant improvements in the the model's performance on the tool-use evaluation set for several tools.

6.2 Generative AI with Search

Several web search providers [4, 19] have recently integrated LLMs with search to provide a generative AI powered response that augments classic search results. Search itself is an incredibly useful tool, and being able to use it proficiently in combination with structured tools like flights search, hotel search etc. can unlock powerful new capabilities. This might help us better answer complex questions like "Find me one way flights from NYC to Delhi where I don't need a transit visa for the layovers. I have a South African passport." To power the Search + LLMs + Tools product, we generated synthetic evaluation datasets for each tool across the seven capabilities. We continued sampling until diversity threshold for each of the capabilities was met and we had 100+ tasks for each tool in each capability. We then sent these tasks to be reviewed by a human, and included all the tasks that passed review into the synthetic evaluation set. Details of the specific evaluation set are proprietary, but the experiments in Section 4 should give the reader a sense for the kinds of queries and the computational effort required to generate them. We then re-sampled queries for all tools and categories as a source for engineers to write demonstrations to include to the fine-tuning mixture like we did for Bard above.

7 Conclusions

Language models have great potential for automation of complex tasks. However, we need a principled way to evaluate the performance of LLMs on tasks that involve tool use. In this paper, we introduced a taxonomy of capabilities to enable researchers and engineers to understand tool use in the context of LLMs.

We have also provided a method for synthesizing queries across each capability using a structured description of the tool's API as the only input. We demonstrated a novel tool-centric method for evaluating LLM capabilities using these synthesized queries. These synthetic queries can also be used for improving existing tool use models by generating candidate trajectories, obtaining quality ratings from humans (or other models), and using the rater data for supervised fine-tuning or reinforcement learning. Our methods ensure that a diverse range of data is generated. This process can also be used to onboard new tools, by leveraging LLM's capabilities for in-context learning and generalization.

In future work, we will explore the quantitative effects of our stratified synthetic data on end-to-end metrics for tool use capabilities. We will also explore the efficacy of using LLMs to automatically evaluate tool use capabilities. A simple self-improvement pipeline can be established: first, generate trajectories for difficult queries;

then, generate sample responses; use a model to identify good generations; and finally use this feedback to improve the generator. Such a system would close the loop and allow for completely automatic, high quality synthesis of tool use trajectories.

Acknowledgements

We would like to acknowledge the contribution of several engineers and product leads in Bard, Search, and the Google Partner Engineering organization that helped guide the early design choices we made in this project: Adam Coimbra, ZJ Yan, Luyan Chi, Dmitry Minaev, Vamsi Bedapudi, Yi Sun, Jinjing Zhou, Kashyap Kolipaka, and Dhruv Nayar.

References

[1] Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, Eric Chu, Jonathan H. Clark, Laurent El Shafey, Yanping Huang, Kathy Meier-Hellstern, Gaurav Mishra, Erica Moreira, Mark Omernick, Kevin Robinson, Sebastian Ruder, Yi Tay, Kefan Xiao, Yuanzhong Xu, Yujing Zhang, Gustavo Hernandez Abrego, Junwhan Ahn, Jacob Austin, Paul Barham, Jan Botha, James Bradbury, Siddhartha Brahma, Kevin Brooks, Michele Catasta, Yong Cheng, Colin Cherry, Christopher A. Choquette-Choo, Aakanksha Chowdhery, Clément Crepy, Shachi Dave, Mostafa Dehghani, Sunipa Dev, Jacob Devlin, Mark Díaz, Nan Du, Ethan Dyer, Vlad Feinberg, Fangxiaoyu Feng, Vlad Fienber, Markus Freitag, Xavier Garcia, Sebastian Gehrmann, Lucas Gonzalez, Guy Gur-Ari, Steven Hand, Hadi Hashemi, Le Hou, Joshua Howland, Andrea Hu, Jeffrey Hui, Jeremy Hurwitz, Michael Isard, Abe Ittycheriah, Matthew Jagielski, Wenhao Jia, Kathleen Kenealy, Maxim Krikun, Sneha Kudugunta, Chang Lan, Katherine Lee, Benjamin Lee, Eric Li, Music Li, Wei Li, YaGuang Li, Jian Li, Hyeontaek Lim, Hanzhao Lin, Zhongtao Liu, Frederick Liu, Marcello Maggioni, Aroma Mahendru, Joshua Maynez, Vedant Misra, Maysam Moussalem, Zachary Nado, John Nham, Eric Ni, Andrew Nystrom, Alicia Parrish, Marie Pellat, Martin Polacek, Alex Polozov, Reiner Pope, Siyuan Qiao, Emily Reif, Bryan Richter, Parker Riley, Alex Castro Ros, Aurko Roy, Brennan Saeta, Rajkumar Samuel, Renee Shelby, Ambrose Slone, Daniel Smilkov, David R. So, Daniel Sohn, Simon Tokumine, Dasha Valter, Vijay Vasudevan, Kiran Vodrahalli, Xuezhi Wang, Pidong Wang, Zirui Wang, Tao Wang, John Wieting, Yuhuai Wu, Kelvin Xu, Yunhan Xu, Linting Xue, Pengcheng Yin, Jiahui Yu, Qiao Zhang, Steven Zheng, Ce Zheng, Weikang Zhou, Denny Zhou, Slav Petrov, and Yonghui Wu. 2023. PaLM 2 Technical Report. arXiv:cs.CL/2305.10403
[2] Bard [n.d.]. Google Bard. http://bard.google.com. Accessed: 2024-01-19.
[3] bard-ext [n.d.]. Bard can now connect to your Google apps and services. https://blog.google/products/bard/google-bard-new-features-update-sept-2023/. Accessed: 2024-01-19.
[4] Bing [n.d.]. Reinventing search with a new AI-powered Microsoft Bing and Edge, your copilot for the web. https://blogs.microsoft.com/blog/2023/02/07/reinventing-search-with-a-new-ai-powered-microsoft-bing-and-edge-your-copilot-for-the-web/. Accessed: 2024-01-22.
[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:cs.CL/2005.14165
[6] ChatGPT [n.d.]. ChatGPT. http://chat.openai.com. Accessed: 2024-01-19.
[7] functioncalling [n.d.]. Function Calling. https://platform.openai.com/docs/guides/function-calling. Accessed: 2024-01-29.
[8] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. 2019. Natural Questions: A Benchmark for Question Answering Research. *Transactions of the Association for Computational Linguistics* 7 (2019), 452–466. https://doi.org/10.1162/tacl_a_00276
[9] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. arXiv:cs.SE/2211.11501
[10] Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs. arXiv:cs.CL/2304.08244
[11] Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-Play Compositional Reasoning with Large Language Models. arXiv:cs.CL/2304.09842
[12] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. arXiv:cs.CL/2203.02155
[13] Aaron Parisi, Yao Zhao, and Noah Fiedel. 2022. TALM: Tool Augmented Language Models. arXiv:cs.CL/2205.12255
[14] Arkil Patel, Satwik Bhattamishra, and Navin Goyal. 2021. Are NLP Models really able to Solve Simple Math Word Problems?. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Online, 2080–2094. https://doi.org/10.18653/v1/2021.naacl-main.168
[15] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large Language Model Connected with Massive APIs. arXiv:cs.CL/2305.15334
[16] Fabio Petroni, Tim Rocktäschel, Sebastian Riedel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, and Alexander Miller. 2019. Language Models as Knowledge Bases?. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, Hong Kong, China, 2463–2473. https://doi.org/10.18653/v1/D19-1250
[17] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. arXiv:cs.AI/2307.16789
[18] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. arXiv:cs.CL/2302.04761
[19] SGE [n.d.]. Supercharging Search with generative AI. https://blog.google/products/search/generative-ai-search/. Accessed: 2024-01-22.
[20] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models.
[21] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:cs.CL/2210.03629

A Sample Positive and Negative Generated Queries

Table 6 shows sample generated queries judged by human raters as positive or negative in each of the capabilities.

B Capability-Specific Diversity Metrics

Capability	Decorator
Reasoning Over Response	`response(field1, field2)`
Tool Chaining	`chaining(field1, field2)`
Fan-out	`fanout(field1, field2)`
Conversational Refinement	`refine(field1)`

Table 7: For several capabilities, we describe a capability-specific decorator to track if the query-generation process emits a diverse set of ways in which to combine

Table 7 presents the capability-specific decorators used to track query diversity. Recall that the distribution of parameters used and unique parameter combinations generated were discussed in Section 3.

C Distribution of Parameters in Generated Queries

Figure 6 shows the distributions of parameters across human-written and synthetically generated queries.

Capability	Positive queries	Negative queries
Single-shot slot-filling	Homes for sale in Seattle for less than 1M with at least 1500 sqft.	Homes in New York City under 2m that fall into the "all other home types" category.
Reasoning over outputs	Prompt 1: Which is the cheapest 2 bedroom rental available in Sunnyvale?, Prompt 2: What's their price?	Prompt 1: Find homes for rent in Mountain View with HOA less than $400, Prompt 2: Show me images of the top 3.
Within tool refinements	Prompt 1: what are the listings for 2 bedrooms 1 bath rentals in San Diego, Prompt 2: Only show listings under 2000 dollars	NA
Tool-chaining	Find the most expensive home in San Mateo and show me directions there from Berkeley.	What transit is available near rental property in downtown Seattle.
Fan-out	Prompt 1: What are some 2 bedrooms homes on the market in Sunnyvale and Mountain View that go for $600k, or less?	Prompt 1: What are the 3 bedrooms single family homes and townhouses on the market in Sunnyvale for at most $600k
Multi-turn memory	Prompt 1: Find homes in Miami Florida, Prompt 2: Make it 5 bedrooms and 3 bathrooms, Prompt 3: Size should be less than 3000 square feet, Prompt 4: Change it to rentals, Prompt 5: Set the price to $1500 per month.	Prompt 1: Find 3 bedrooms in Austin that are dog friendly, Prompt 2: Can you look up Austin's animal shelter?
Error handling	I'm looking for a home in the mountains with a view of the lake.	Show me all homes in the bay area with a walk score of 90 that are near a good elementary school.

Table 6: From human evaluation result, we sample one positive query and one negative query per capability. The precision for 'within tool refinements' is 100%, so there is no negative query.

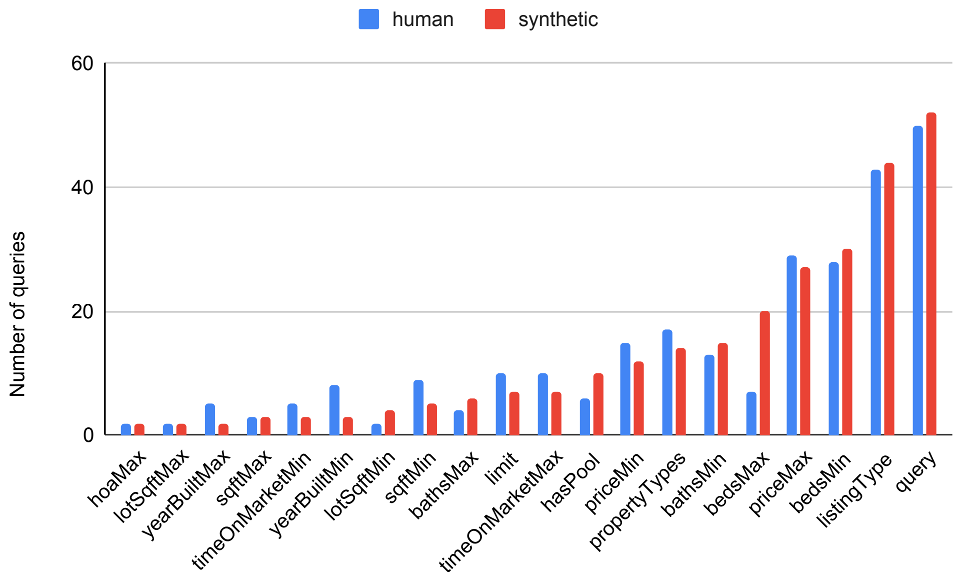


Figure 6: Comparison of the distribution of parameters between 53 human generated queries and and the same number of synthetic queries for the real estate search API.