Engineering, Backend, Data / ML, Uber AI

QueryGPT – Natural Language to SQL Using Generative AI

September 19, 2024 / Global


Natural Language
to
SQL

Introduction

SQL is a vital tool used daily by engineers, operations managers, and data scientists at Uber to access and manipulate terabytes of data. Crafting these queries not only requires a solid understanding of SQL syntax, but also deep knowledge of how our internal data models represent business concepts. QueryGPT aims to bridge this gap, enabling users to generate SQL queries through natural language prompts, thereby significantly enhancing productivity.

QueryGPT uses large language models (LLM), vector databases, and similarity search to generate complex queries from English questions that are provided by the user as input.

This article chronicles our development journey over the past year and where we are today with this vision.

Motivation

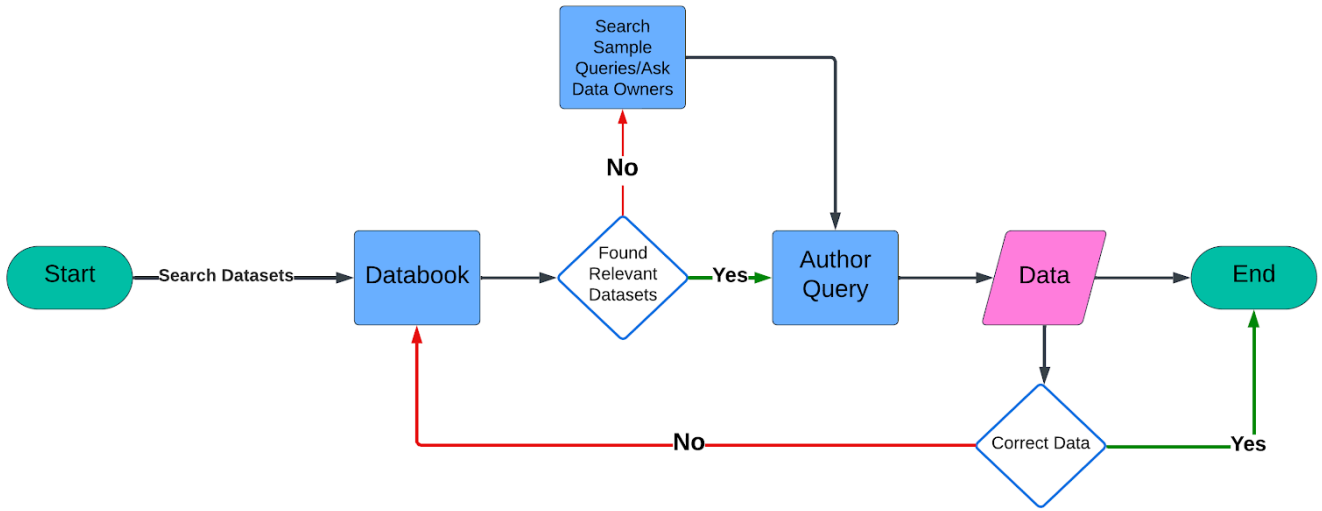

Figure 1: Query Authoring Process.

At Uber, our data platform handles approximately 1.2 million interactive queries each month. The Operations organization, one of the largest user cohorts, contributes to about 36% of these queries. Authoring these queries generally requires a fair amount of time between searching for relevant datasets in our data dictionary and then authoring the query inside our editor. Given that each query can take around 10 minutes to author, the introduction of QueryGPT, which can automate this process and generate reliable queries in just about 3 minutes, represents a major productivity gain.

If we make a conservative estimate that each query takes about 10 minutes to author, QueryGPT can automate this process and provide sufficiently reliable queries in about 3 minutes. This would result in a major productivity gain for Uber.

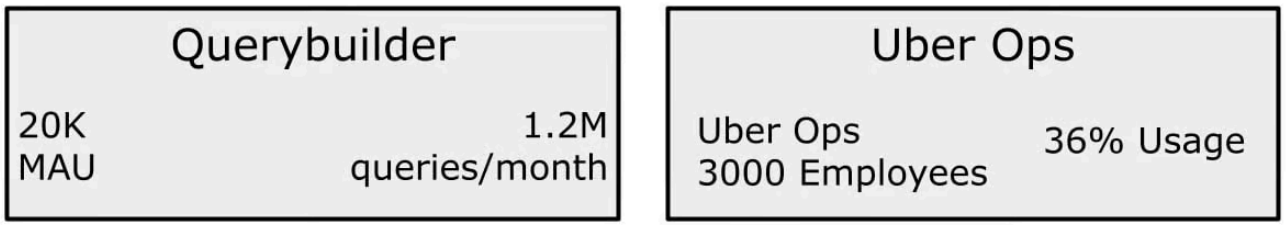

Figure 2: Querybuilder Usage.

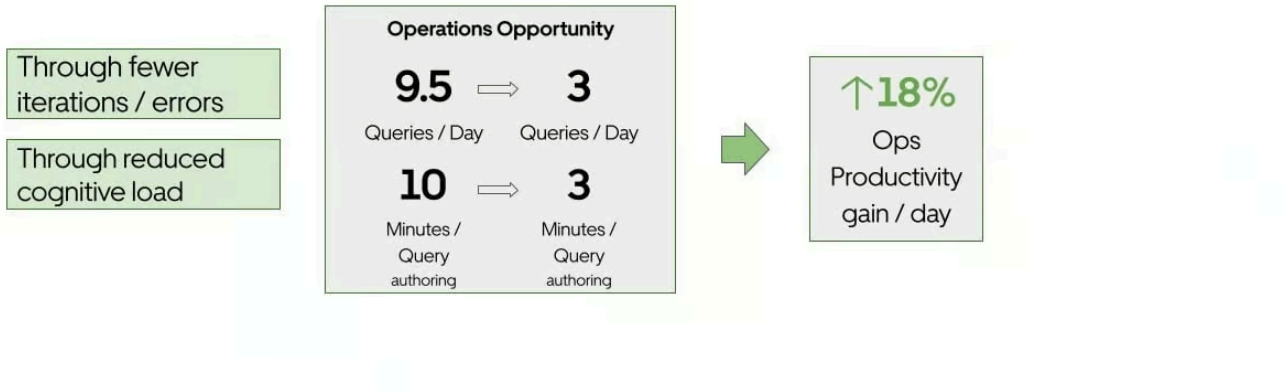

Figure 3: QueryGPT Impact.

Architecture

QueryGPT originated as a proposal during Uber's Generative AI Hackdays in May 2023. Since then, we have iteratively refined the core algorithm behind QueryGPT, transitioning it from concept to a production-ready service. Below, we detail the evolution of QueryGPT and its current architecture, highlighting key enhancements..

We've described below our current version of QueryGPT. Please bear in mind that there were about 20+ iterations of the algorithm between the 2 detailed below and if we were to list and describe each, the length of this blog article would put Ayn Rand's *Atlas Shrugged* to shame.

Hackdayz (version 1)

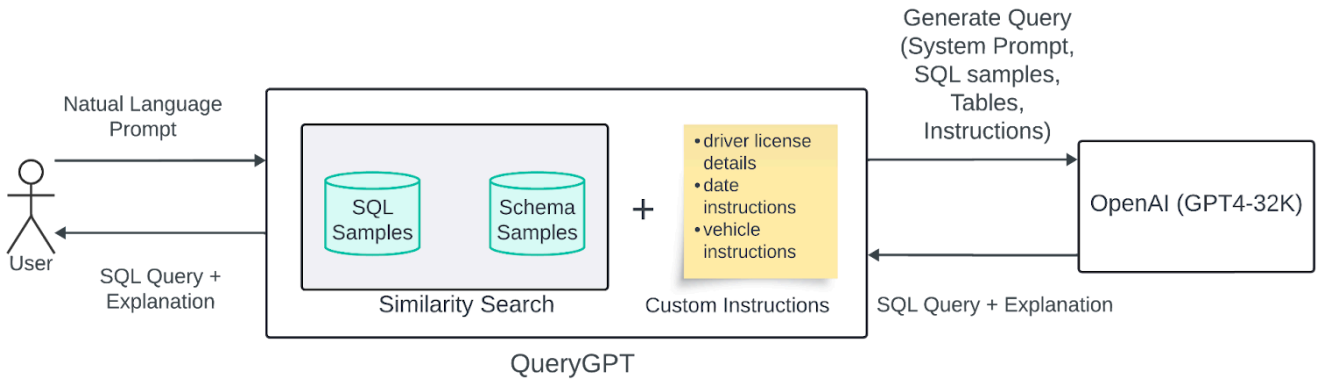

Figure 4: QueryGPT Hackdayz version

The first version of QueryGPT relied on a fairly simple RAG to fetch the relevant samples we needed to include in our query generation call to the LLM (Few Shot Prompting). We would take the user's natural language prompt, vectorize it and do a similarity search (using k-nearest neighbor search) on the SQL samples and schemas to fetch 3 relevant tables and 7 relevant SQL samples.

The first version used 7 tier 1 tables and 20 SQL queries as our sample data set. The SQL queries were supposed to provide the LLM guidance on how to use the table schemas provided and the table schemas provided the LLM information about the columns that existed on those tables.

For example, for a tier 1 table, *uber.trips_data* (not a real table), this is what the schema would look like:

```
CREATE TABLE uber.trips_data
    trip_id INT NOT NULL,
    vehicle_type INT NOT NULL,
    trip_start_location STRING NOT NULL,
    trip_end_location STRING NOT NULL
)
```

Figure 5: uber.trips_data table.

To help the LLM understand internal Uber lingo and work with Uber datasets, we also included some custom instructions in the LLM call. Shown below is a snippet of how we wanted the LLM to work with dates:

```
## DATATYPE FUNCTIONS AND COMPARISONS ##

It's important to consider the data type of columns in a given dataset because certain functions
can only be applied to some column types.
For example date functions like DATE_TRUNC(), DATE_ADD can only be used with TIMESTAMPs.

SUBSTR() and CONCAT() can only be used on STRINGs.
```

Figure 6: Custom Instructions for handling dates in Uber datasets.

We would wrap all the relevant schema samples, SQL samples, user's natural language prompt, and Uber Business instructions around a system prompt and send the request to the LLM.

The response would include an "SQL Query" and an "Explanation" of how the LLM generated the query:

```
{
    "explanation": "To answer the question, we need to join the 'uber.trips' table with
    the 'uber.cities' table to filter trips by the city name 'Seattle'. We will use the
    'city_id' column to join these tables. Since 'uber.trips' is partitioned by
    'date_val', we must include a filter for this partition column. We will filter for
    trips that were completed yesterday using the 'is_trip_completed' column and the
    'date_val' column. The 'date_val' column will be cast to a DATE type to compare
    with the date function that calculates yesterday's date. We will use a COUNT()
    function to get the number of trips and GROUP BY to group the results by
    'city_name'.",
    "sql_query": "SELECT COUNT(*) AS number_of_trips FROM uber.trips ft JOIN uber.city
    dc ON ft.city_id = dc.city_id WHERE ft.is_trip_completed = TRUE AND dc.name_of_city
    = 'Seattle' AND DATE(ft.date_val) = DATE_ADD('day', -1, CURRENT_DATE) AND ft.
    date_val IS NOT NULL"
}
```

Figure 7: Generated SQL with Explanation.

While this version of the algorithm worked well for a small set of schemas and SQL samples, as we started to onboard more tables and their associated SQL samples into the service, we started seeing declining accuracy in the generated queries.

Better RAG

Doing a simple similarity search for a user's natural language prompt ("Find the number of trips completed yesterday in Seattle") on schema samples (*CREATE TABLE...*) and SQL queries (*SELECT a, b, c FROM uber.foo ...*) doesn't return relevant results.

Understanding User's Intent

Another issue we found was that it's incredibly challenging to go from a user's natural language prompt to finding the relevant schemas. What we needed was an intermediate step, which classifies the user's prompt into an "intent" that maps to relevant schemas and SQL samples.

Handling Large Schemas

We have some really large schemas on some Tier 1 tables at Uber, with some spanning over 200 columns. These large tables could use up as much as 40-60K tokens in the request object. Having 3 or more of these tables would break the LLM call since the largest available model (at the time) only supported 32K tokens.

Current Design

The diagram below shows the current design of QueryGPT that we're running in production. The current version includes many iterative changes from the first version.

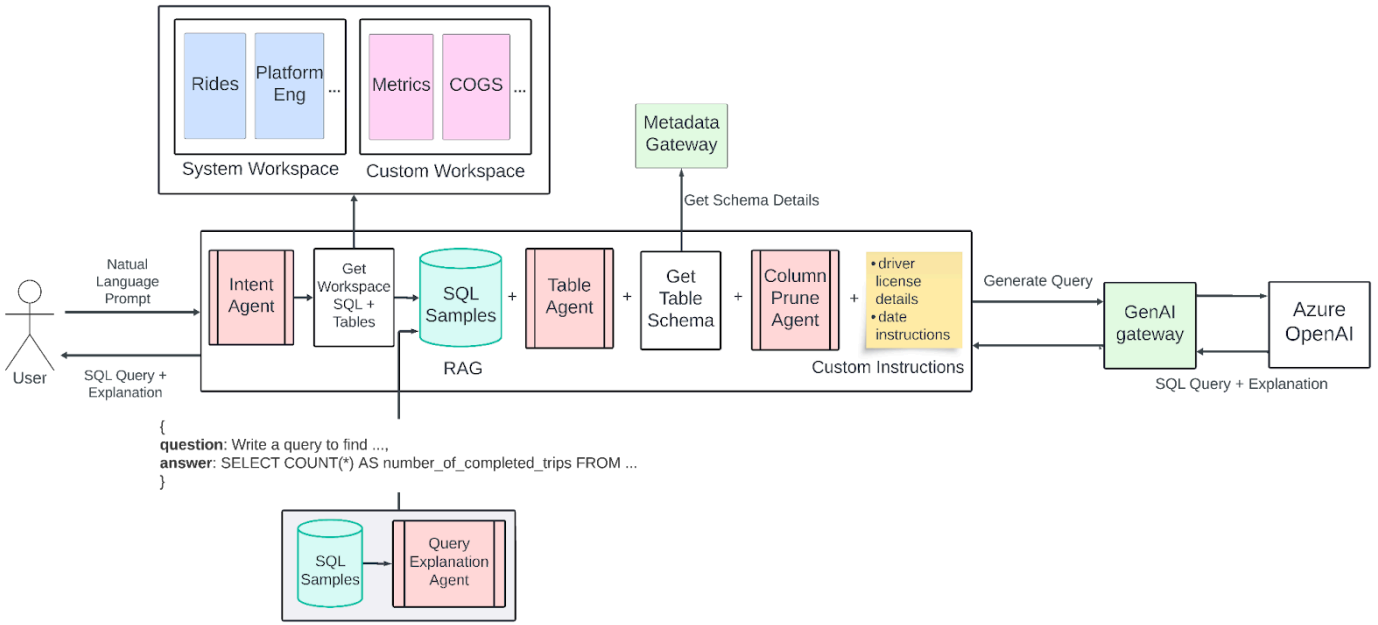

Figure 8: QueryGPT (current).

Workspaces

In our current design, we introduced "workspaces," which are curated collections of SQL samples and tables tailored to specific business domains such as Ads, Mobility, and Core Services. These workspaces help narrow the focus for the LLM, improving the relevance and accuracy of generated queries.

We've identified some of the more common business domains inside Uber and created those in the backend as "System Workspaces." Mobility is one of these system workspaces that we identified as foundational domains that could be used for query generation.

Mobility: Queries that include trips, driver, document details, etc.

Write **a** query to find **the** number **of** trips that were completed **by** Teslas **in** Seattle yesterday

Along with these, we also shipped 11 other system workspaces, including “Core Services,” “Platform Engineering,” “IT,” “Ads,” etc.

We also included a feature that allows users to create “Custom Workspaces” if none of the existing system workspaces fit their requirement and use those for query generation.

Intent Agent

Every incoming prompt from the user now first runs through an “intent” agent. The purpose of this intent agent is to map the user’s question to one or more business domains/workspaces (and by extension a set of SQL samples and tables mapped to the domain). We use an LLM call to infer the intent from the user question and map these to “system” workspaces or “custom” workspaces.

Picking a business domain allowed us to drastically narrow the search radius for RAG.

Table Agent

Allowing users to select the tables used in the query generation came up as feedback from some users who saw that the tables that were eventually picked by QueryGPT were not correct.

To address this feedback, we added another LLM agent (Table Agent) that would pick the right tables and send those out to the user to either “ACK” or edit the given list and set the right tables. A screenshot of what the user would see is shown below:

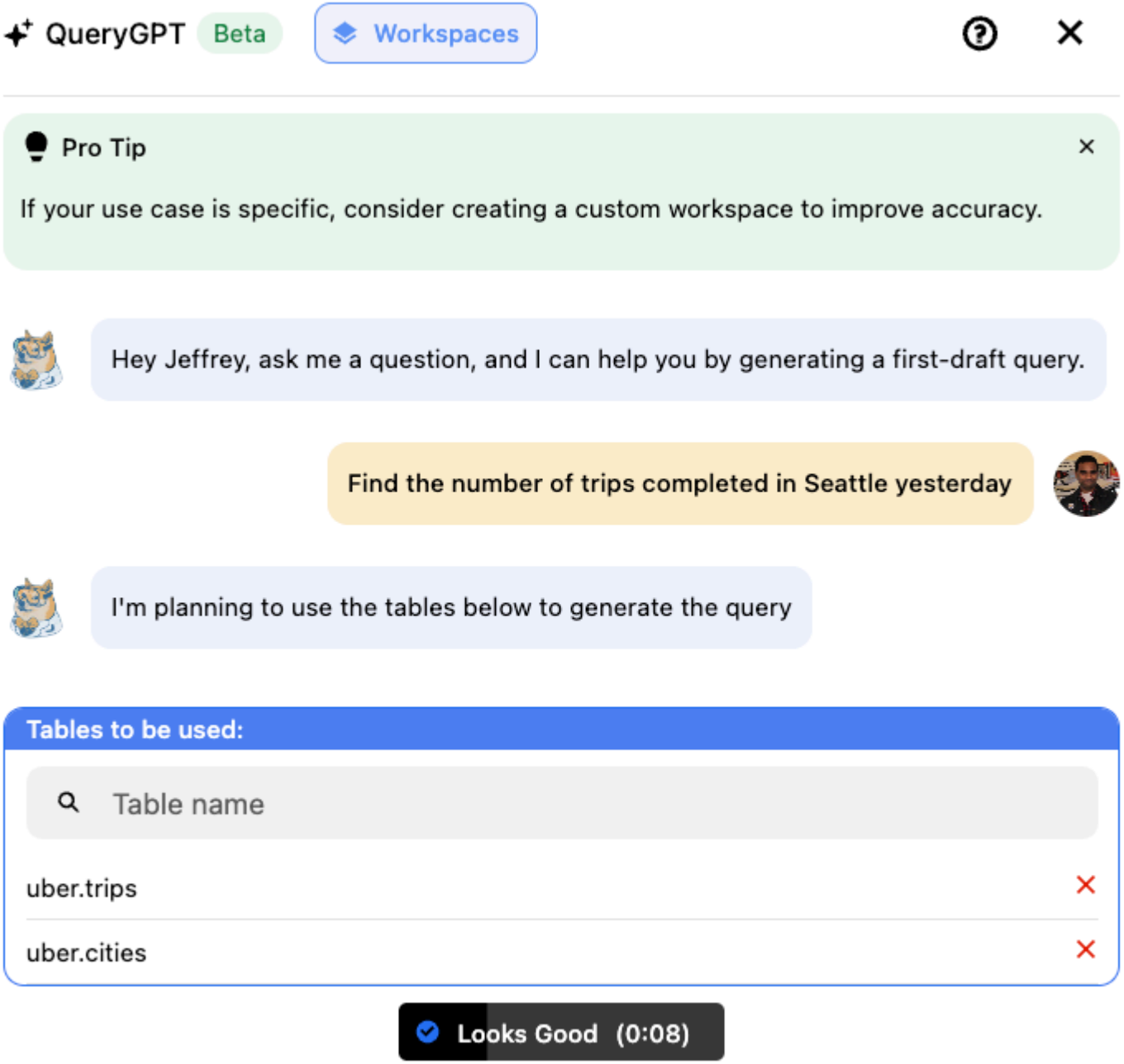

Figure 9: Table Agent.

The user would either select the “Looks Good” button or edit the existing list and modify the list of tables to be used by the LLM for query generation.

Column Prune Agent

Another interesting issue we ran into after rolling QueryGPT out to a larger set of users was the “intermittent” token size issue during query generation for some requests. We were using the OpenAI GPT-4 Turbo model with 128K token limit (1106), but were still seeing token limit issues because some requests included one or more tables that each consumed a large amount of tokens.

To address this issue, we implemented a “Column Prune” agent, wherein we use an LLM call to prune the irrelevant columns from the schemas we provided to the LLM.

Here’s what the output from the agent looks like:

```
{
    "pruned_schema": {
        "explanation": "The user's question is about the number of trips completed yesterday in
        Seattle. The table provided is a dimension table for cities. To answer the question, we need
        to identify the city 'Seattle', so we need the 'name_of_city' column. Other columns such as
        'is_operational', 'lang_code', 'ctry_uuid', 'min_score', 'shape_val', 'mg_region', 'market',
        'ctry_code', 'tz', 'city_id', 'terr_uuid', 'timestamp', 'dist_unit', 'currency', 'longit',
        'country', 'ctry_iso2', 'operational', 'sb_region', 'latit', 'upd_epoch', 'ctry_iso3',
        'plan_unit', 'upd_timestamp', 'gfence_uuid', 'display_name', 'ctry_id', 'require_mob_ver',
        'lnch_date' are not directly relevant to the question asked. However, without more context on
        how the trips are linked to the cities, it is conservative to keep the 'city_id' as it might
        be used as a foreign key in another table that stores trip data.",
        "schema": "{\"tableName\":\"uber.cities\",\"tableDescription\":\"This dimension table stores
        one row per city that Uber is operating in or has operated previously providing any of uber
        products eg. Transport, Food, etc. This table excludes cities commissioned for product
        testing.\",\"columns\":[{\"name\":\"city_id\",\"type\":\"BIGINT\",\"isPartitionKey\":false},
        {\"name\":\"name_of_city\",\"type\":\"STRING\",\"isPartitionKey\":false}],
        \"tier\":\"TIER_ONE\",\"usagePercentile\":99.9}"
    }
}
```


Figure 10: Column Prune Agent

The output would include a skinnier version of each schema that we needed for query generation. This change massively improved not just the token size and by extension the cost of each LLM call, but also reduced the latency since the input size was much smaller.

Output

No changes were made to the output structure with the current design. The response would include a SQL Query and an explanation from the LLM about how the query was generated similar to what's shown in Figure 7.

Evaluation

To track incremental improvements in QueryGPT's performance, we needed a standardized evaluation procedure. This enabled us to differentiate between repeated vs. anomalous shortcomings of the service and ensure algorithm changes were incrementally improving performance in aggregate.

Evaluation Set

Curating a set of golden question-to-SQL answer mappings for evaluation required manual upfront investment. We identified a set of real questions from the QueryGPT logs, and manually verified the correct intent, schemas required to answer the question, and the golden SQL. The question set covers a variety of datasets and business domains.

Evaluation Procedure

We developed a flexible procedure that can capture signals throughout the query generation process in production and staging environments using different product flows:

Product Flow	Purpose	Procedure

Vanilla	Measures QueryGPT's baseline performance.	Input a question.QueryGPT infers the intent and datasets needed to answer the question.Generate the SQL using inferred datasets and intent.Evaluate intent, datasets, and SQL.
Decoupled	Measures QueryGPT performance with the human-in-the-loop experience. Enables component-level evaluation by removing dependencies on performance on earlier outcomes.	Input a question, intent, and datasets needed to answer the question.QueryGPT infers the intent and datasets.Generate the SQL using the actual (not inferred) intent and datasets. Evaluate intent, datasets, and SQL.

For each question in the evaluation, we capture the following signals:

Intent: Is the intent assigned to the question as accurate?

Table Overlap: Are the tables identified via Search + Table Agent correct? This is represented as a score between 0 and 1. For example, if the query needed to answer the questions "How many trips were canceled by drivers last week in Los Angeles?" required the use of [*fact_trip_state* , *dim_city*], and QueryGPT identified [*dim_city, fact_eats_trip*], the Search Overlap Score for this output would be 0.5, because one of the two tables required to answer the question was selected.

Successful Run: Does the generated query run successfully?

Run Has Output: Does the query execution return > 0 records. (Sometimes, QueryGPT hallucinates filters like WHERE status = “Finished” when, the filter should have been WHERE status = “Completed” resulting in a successful run with no output).

Qualitative Query Similarity: How similar is the generated query relative to the golden SQL? We use an LLM to assign a similarity score between 0 and 1. This allows us to quickly see if a generated query that is failing for a syntactic reason is on the right track in terms of columns used, joins, functions applied, etc.

We visualize progress over time to identify regressions and patterns revealing areas for improvement.

The figure below is an example of question-level run results enabling us to see repeated shortcomings at individual question level.

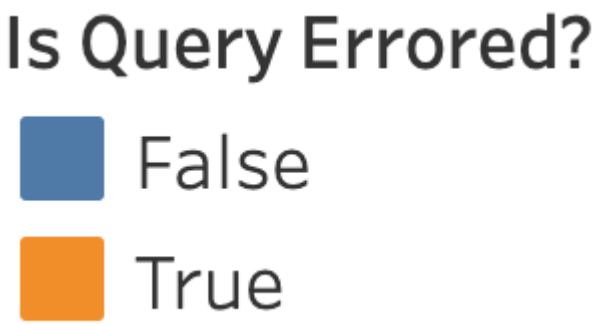

Figure 11A: SQL Query Evaluation.

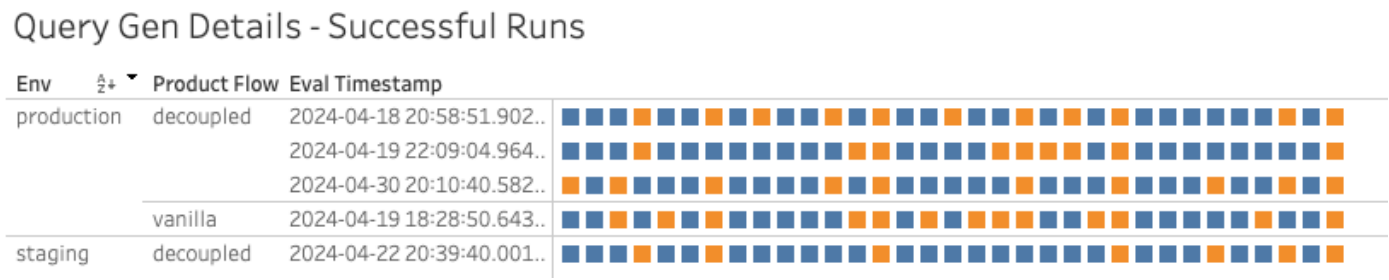

Figure 11B: SQL Query Evaluation across environments.

For each question, we can view the generated SQL, reason for the error, and related performance metrics. Below is a question whose generated query is regularly failing because it is not applying a partition filter in the where clause. However, according to the qualitative LLM-based evaluation, the generated SQL is otherwise similar to the golden SQL.

Eval Timestamp:	2024-04-19 18:28:50.643231
Env:	production
Is Query Errored?:	True
Product Flow:	vanilla
Question:	How many scheduled trips happened in the past 3 months grouped by the following attributes: city, reservation variant, is health, is central.
Error Classification:	missing_partition_filter
Has Output?:	False
LLM SQL Eval Score:	85.00%

Figure 12: SQL Evaluation Stats.

We also aggregate accuracy and latency metrics for each evaluation run to track performance over time.

Eval-Level Metrics

Env	Product Flow	Eval Timestamp	% Questions with Context Match	Avg. Dataset Overlap Score	Avg. LLM SQL Eval Score	% Queries with Successful Run	% Queries with > 0 Output
production	decoupled	2024-04-18 20:58:51.902..	75.76%	48.99%		66.67%	54.55%
		2024-04-19 22:09:04.964..	75.76%	50.51%	74.39%	72.73%	57.58%
		2024-04-30 20:10:40.582..	78.79%	47.47%	73.03%	69.70%	60.61%
	vanilla	2024-04-19 18:28:50.643..	81.82%	48.99%		60.61%	39.39%
staging	decoupled	2024-04-22 20:39:40.001..	78.79%	48.99%	71.36%	78.79%	63.64%

Figure 13: SQL Evaluation criterions.

Limitations

Due to the non-deterministic nature of LLMs, running the same evaluation with no changes to the underlying QueryGPT service can result in different outcomes. In general, we do not over-index decisions based on ~5% run-to-run changes in most metrics. Instead, we identify error patterns over longer time periods that can be addressed by specific feature improvements.

Uber has hundreds of thousands of datasets with varying levels of documentation. Thus, it is impossible for the set of evaluation questions to fully cover the universe of business questions that a user may ask. Instead, we curated a set of questions that represent the current usage of the product. As we improve accuracy and new bugs arise, the evaluation set will evolve to capture the direction of the product.

There is not always one correct answer. Often, the same question could be answered by querying different tables or writing queries in different styles. By visualizing the golden vs.

returned SQL and using the LLM-based evaluation score, we can understand if the generated query is written in a different style, but has a similar intent related to the golden SQL.

Learnings

Working with nascent technologies like GPTs and LLMs over the past year allowed us to experiment and learn a lot of different nuances of how agents and LLMs use data to generate responses to user questions. We've briefly described below some of the learnings from our journey:

LLMs are excellent classifiers

Our intermediate agents that we used in QueryGPT to decompose the user's natural language prompt into better signals for our RAG improved our accuracy a lot from the first version and a lot of it was due to the fact that the LLMs worked really well when given a small unit of specialized work to do.

The intent agent, table agent, and column prune agent each did an excellent job because they were asked to work on a single unit of work rather than a broad generalized task.

Hallucinations

This remains an area that we are constantly working on, but in a nutshell, we do see instances where the LLM would generate a query with tables that don't exist or with columns that don't exist on those tables.

We've been experimenting with prompts to reduce hallucinations, introduced a chat style mode where users can iterate on the generated query and are also looking to include a "Validation" agent that recursively tries to fix hallucinations, but this remains an area that we haven't completely solved yet.

User prompts are not always "context"-rich

Questions entered by the users in QueryGPT ranged from very detailed with the right keywords to narrow the search radius to a 5 word question (with typos) to answer a broad question that would require joins across multiple tables.

Solely relying on user questions as "good" input to the LLM caused issues with accuracy and reliability. A "prompt enhancer" or "prompt expander" was needed to "massage" the user question into a more context-rich question before we sent those to the LLM.

High bar for SQL output generated by the LLM

While this version of QueryGPT is helpful for a broad set of personas, there is definitely an expectation from many that the queries produced will be highly accurate and "just work." The bar is high!

In our experience, we found that it was best to target and test with the right persona(s) as your initial user base when building a product like this.

Conclusion

The development of QueryGPT at Uber has been a transformative journey, significantly enhancing productivity and efficiency in generating SQL queries from natural language prompts. Leveraging advanced generative AI models, QueryGPT seamlessly integrates with Uber's extensive data ecosystem, reducing query authoring time and improving accuracy, addressing both the scale and complexity of our data needs.

While challenges such as handling large schemas and reducing hallucinations persist, our iterative approach and constant learning have enabled continuous improvements. QueryGPT not only simplifies data access but also democratizes it, making powerful data insights more accessible across various teams within Uber.

With our limited release to some teams in Operations and Support, we are averaging about 300 daily active users, with about 78% saying that the generated queries have reduced the amount of time they would've spent writing it from scratch.

As we look forward, the integration of more sophisticated AI techniques and user feedback will drive further enhancements, ensuring that QueryGPT remains a vital tool in our data platform.

Acknowledgements

QueryGPT was a cross-discipline effort across Uber, requiring expertise and domain knowledge from folks working in Engineering, Product Management and Operations. The product wouldn't exist today without the great contributions by Abhi Khune, Callie Busch, Jeffrey Johnson, Pradeep Chakka, Saketh Chintapalli, Adarsh Nagesh, Gaurav Paul and Ben Carroll.

Jeffrey Johnson

Jeffrey Johnson is a Staff Software Engineer on Uber's Data Platform team. Jeffrey has been primarily focussed on leveraging LLM's to improve productivity across different personas at Uber and leading security related initiatives for the Business Intelligence team.

Callie Busch

Callie Busch is a Software Engineer II on Uber's Data Platform team.

Abhi Khune

Abhi Khune is a Principal Engineer on Uber's Data Platform team. For the past 6 months, Abhi has been leading the technical strategy to modernize Uber's Data Platform to a Cloud-based architecture.

Pradeep Chakka

Pradeep Chakka is a Senior Software Engineer on Uber's Data Platform team, based in Dallas, TX.

Posted by Jeffrey Johnson, Callie Busch, Abhi Khune, Pradeep Chakka

Category: Engineering Backend Data / ML Uber AI

Related articles

Engineering, Backend, Data / ML, Uber AI

Enhancing Personalized CRM Communication with Contextual Bandit Strategies

March 27 / Global

Engineering, Backend

Automating Efficiency of Go programs with Profile-Guided Optimizations

March 13 / Global

Engineering, Backend

Adopting Arm at Scale: Transitioning to a Multi-Architecture Environment

February 27 / Global

Engineering, Backend

Adopting Arm at Scale: Bootstrapping Infrastructure

February 13 / Global

Engineering, Backend

MySQL At Uber

January 30 / Global

Most popular

Transit, Universities March 3 / Global
A beginner’s guide to Uber vouchers for riders

Engineering, Backend March 13 / Global
Automating Efficiency of Go programs with Profile-Guided Optimizations

Engineering, Backend, Data / ML, Uber AI March 27 / Global
Enhancing Personalized CRM Communication with Contextual Bandit Strategies

Transit, Universities April 2 / Global
How medical schools support the next generation of doctors with Uber

View more stories

Uber

Visit Help Center

Company's registered name - Uber Formosa Co. Ltd. Taxation registration number - 83118125

Company

About us

Our offerings

Newsroom

Investors

Blog

Careers

Uber AI

Gift cards

Products

Ride

Drive

Deliver

Eat

Uber for Business

Uber Freight

Global citizenship

Safety

Sustainability

Travel

Reserve

Airports

Cities

English

Dublin