



MEng EECS Capstone Report
University of California, Berkeley

Interactive Neural SQL Synthesis

Authors:

Anish Doshi

Harry Singh

Chia-En Chiang

Wesley A. Cheng

Advisors:

Dr. Xiaodong (Dawn) Song

Xinyun Chen



Executive Summary

Structured Query Language (SQL), is ubiquitous throughout the data science community. However, SQL is syntactically complex, difficult to remember, and does not always align with users' mental models. We aim to change this by introducing a methodology as well as a user interface to enable translation of natural language (NL) queries into formal SQL. We introduce a methodology for creating targeted contextual inputs to OpenAI Codex, and perform novel benchmarks showing that Codex is able to achieve ~54% accuracy on a challenging NL to SQL (NL2SQL) dataset. By sampling queries from Codex, as well as other state-of-the-art neural models such as RAT-SQL and T5, we provide a collection of multiple SQL query suggestions to surface to the user. We also introduce a novel neural ranker that is capable of ranking these suggestions based on the user's interactions with their database using our User Interface (UI). To encapsulate our approach, we develop an interactive application that allows users to type, speak, and select data as input to the neural models. We designed a user study to validate our approach, and found that our approach reduces the time taken to author query by ~5x. Qualitative feedback received from data scientists and engineers has been overwhelmingly positive, with 100% of users stating they would recommend our application to others.

Table of Contents

Business Motivation	3
Technical Report	5
Background & Existing Approaches	5
Synthesizing SQL from Natural Language	5
Providing an Interface for Users	6
Methodology	7
Neural Generation	8
Codex	9
RAT-SQL + GAP + NAT-SQL	11
T5 + PICARD	11
Post-processing and Neural Ranking	11
System Design	13
User Interface Design	15
Results	16
Model Benchmark	16
Sample Queries	20
User Studies	21
Conclusion	22
References	24
Appendix A: Gantt Chart	27
Appendix B: User Study Script	28
Appendix C: User Study Data	30

Business Motivation

The use of databases has become universal in the 21st century. From storing medical records, to processing streams of sensor data, to indexing education records, databases are used in every domain to store, process, index, and secure data for later use. Interacting with most modern databases requires a programming language called SQL (Structured Query Language). This language is used to set up and structure databases, make updates, inserts, additions, and deletions, and query the database for its records.

Given the importance of databases, it should not come as a surprise that SQL is one of the most important skills of data scientists as well as the workforce and of the 21st century based on industry reports from Coursera, Indeed, and KDnuggets [1, 2, 3]. Given that the data science profession is growing at an annual rate of 31% year-over-year, which is much faster than average according to the U.S. Bureau of Labor Statistics, SQL is a must-have skill for anyone to compete in the 21st century job market [4, 5]. The data science profession is also of paramount importance to companies around the world as evident from 2020 The Future of Jobs Report from the World Economic Forum, who surveyed employers across 15 industry sectors in 26 of the world's advanced and emerging economies to give insight on employment trends in the next 5 years. The job report concluded that the projected highest increase in demand for job roles are data scientists as COVID-19 has accelerated digital transformation of businesses as a necessary survival strategy instead previously held belief that it is merely an optional marginal improvement in operations (Figure 1) [6]. Given these massive tailwinds to upskill to data roles, it is necessary for many aspiring data scientists to learn SQL in the coming years.

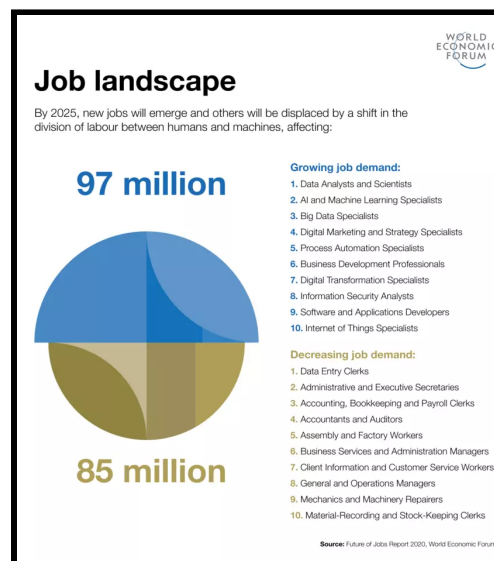


Figure 1: World Job Landscape 2020-2025

SQL is a specialized language designed in the 1970s at IBM to interact with relational databases. Although the recent revisions of the SQL standard have introduced non-relational database features like JavaScript Object Notation (JSON) in 2016, the increasing adoption of non-relational databases at twice the rate of relational databases is reducing the value of learning SQL as data becomes more JSON-like, which non-relational database excel at, rather than table-like, which relational databases excel at. This trend is likely due to the enormous influx of data from the Internet of Things sector hosted on emerging Cloud Computing platforms [7, 8, 9]. SQL as a language is very limited in its capacity to port to other domains compared to general purpose languages like Python that are used in many different settings like web development, machine learning, and many other fields. It would not be unreasonable to ask, “Given these headwinds on the declining value of SQL in the future job market, why even learn it at all?” We ask the same question and believe that it is not only a valid question to ask, but also regard SQL as a huge roadblock in transitioning our current workforce to the 21st century.

Given industry demand and recent breakthroughs in code generation or program synthesis, we see it fit to create a platform that allows data scientists to bypass SQL and use English instead in interacting with their databases. The difficulty with using English, however, is that it lacks *structure*. Programming languages can be deterministically and safely understood by computers. English, however, lacks this inherent structure - and without a way to safely understand the intent from English phrases, interacting with databases may seem impossible for the large population of the workforce without knowledge of SQL.

The advent of deep learning has offered another option. The last few decades have seen tremendous breakthroughs such as the Transformer model architecture, and new GPU architectures specifically designed to train them at an ever increasing scale. Extensive research using these breakthroughs have resulted in neural networks trained to translate Natural Language to SQL (NL2SQL) [10, 11]. One noteworthy example of this is the recent OpenAI model called Codex, a billion-parameter GPT-3 model fine-tuned on GitHub code, for the sole task of generating Python code from English that produced state-of-the-art results on code generation. Amazingly, although it was designed to generate Python code, it is able to generate other languages such as SQL as well due to the enormity of the pretrained corpus of the GPT-3 model [12].

However, we note the lack of *usable applications* that make use of this technology. Commercial database applications such as SAP Hana, Snowflake, and Microsoft SQL Server provide battle-tested interfaces to perform operations on databases and author SQL queries within editors, but as of yet have not taken advantage of the advances in NL2SQL. Of course, translation from NL to SQL is still not perfect, but we argue that

providing an interface for someone to *iteratively* use the outputs of such models still dramatically improves anyone's ability to interact with databases. For veterans of SQL, having suggestions for automated tasks they find themselves performing reduces the time taken to write new queries. For newcomers, offering SQL suggestions from English commands allows people to start interacting with databases in ways they could not before.

This platform not only makes data scientists more productive as they wrangle SQL less, but also breaks down the barrier of entry to the profession. In effect, this platform will not only alleviate the current shortage of data scientists around the world, but also democratize the data science profession to the world. Our team developed an interactive application that allows users to type, speak, and select data as input to the NL2SQL synthesizer. Initial qualitative feedback received from data scientists has been overwhelmingly positive, with 100% of users stating they would recommend our app to others, as they observe a 5x increase in their query writing speed.

Technical Report

Background & Existing Approaches

Our goal is to build an interface for a user to enter in a natural language query (e.g. "Fetch the latest 10 invoices"), and having the system display a *synthesized* SQL query that matches their intent, (e.g. "SELECT * from Invoices ORDER BY date DESC LIMIT 10"). Research into this task has mainly progressed from the academic community, with collaboration from corporate research teams at companies like Google, Facebook, and Salesforce.

Synthesizing SQL from Natural Language

SQL is amenable as a target for automated synthesis because it is a *formal language* - its composite tokens (keywords like "SELECT", delimiters like ",", and other characters) come from a fixed alphabet, and its structure is guaranteed to follow a definite grammar. In fact, many synthesis strategies for SQL focus on formally searching over this grammar. The challenge, however, is understanding how this search can be guided by ambiguously specified natural language (which does not strictly follow a grammar).

In recent years, a multi-step deep learning approach has proven to be the most effective way to address this. The natural language query is fed into a neural network encoder, which often makes use of neural attention to generate a representation that captures the semantic intent of the query. Next, the *schema* of the database, i.e. table names, column names, and optionally types + foreign key constraints, is also encoded, either in the same sequence as the query or with another model. TaPas uses a pre-trained language model, BERT, to jointly encode both schema and NL query [13]; another approach is to use a graphical attention model, as done by RAT-SQL [14]. Finally, a neural network outputs the SQL query token by token, with some approaches instead of using "slot-filling" or an intermediate language representation [15].

Recently, OpenAI's Codex model, trained on GitHub, provided an alternative approach to SQL specific models: by training a large language model on a *huge* corpus of code sourced from GitHub, OpenAI showed that numerous code synthesis tasks can be performed simply by inputting a natural language description of the task into the model [16, 17].

These methods are typically trained and evaluated on benchmark datasets containing thousands of manually curated natural language query/SQL query pairs. The two most notable such datasets are Spider and CoSQL [18, 19]; with the latter benchmark more oriented towards *conversational* natural language to SQL methods.

Providing an Interface for Users

Research on the user interface/experience (UI/UX) aspect of SQL synthesis is surprisingly sparse, and mainly originates from the corporate research community. Salesforce's Photon, released in 2020, provides a conversational interface that mimics a "text chat" with an interactive SQL synthesizer [20]. Microsoft's DIY, released in 2021, includes a tool to let users post process neurally synthesized SQL queries [21]. However, it remains to be seen whether such tools provide effective and natural experiences for users. Instead, by developing our tool with human-computer interaction in mind, as Sarah Chasins encourages in the class "PL and HCI: Better Together", we aim to provide flexibility and intuitiveness for users [22]. To ensure our approach is solving real problems, we will also continuously conduct *user studies*, to improve our application, which is a crucial component of building a usable application.

Methodology

Overall Architecture

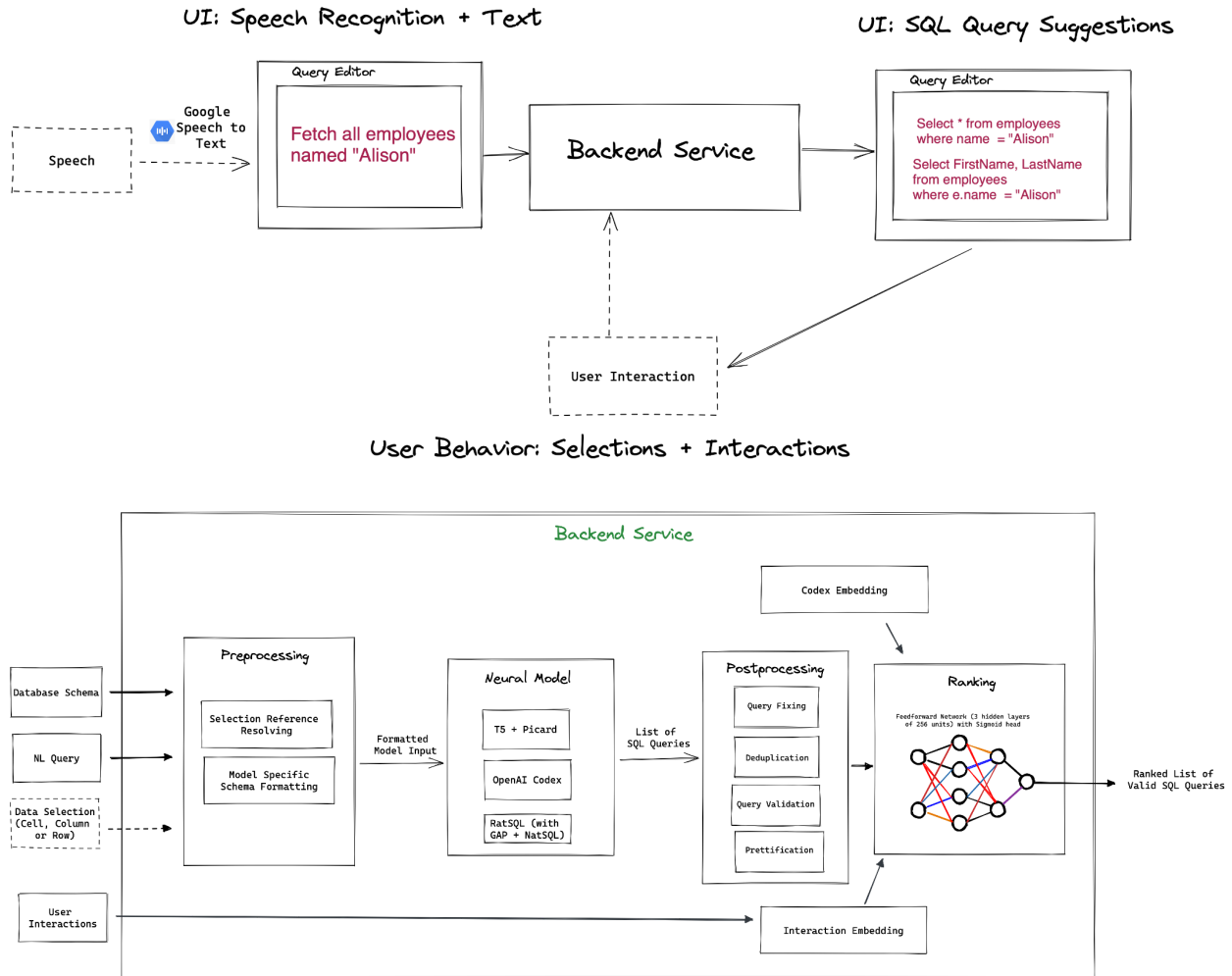


Figure 2: Overview of our overall system and backend. Our algorithm consists of pre-processing multiple types of input specifications, calling multiple neural SQL generation models, post-processing outputs to be valid queries, and ranking with another neural model.

In this section, we detail our approach to building an interactive SQL synthesizer from natural language. We first discuss our algorithm we used to pre-process and convert natural language queries into SQL using an ensemble of neural models, the ranking and post-processing steps used to filter and score each SQL query by likelihood, and finally discuss the actual system implementation and UI design. In order to properly track our

progress on each of these stages, our team employed the use of a Gantt chart (Appendix A) and meticulously brought up issues.

Specifications and Pre-processing

The main inputs to our algorithm are the **natural language command** (having been typed out into the editor, or spoken and recognized with our speech recognition API), the **database schema**, and **data selection info**: our user interface design allows the user to interact with the database in a natural manner. As such, it is natural for a user to select columns, rows, or cells whilst navigating and searching for information.

Pre-processing essentially consists of translating the natural language query into one with *references to selected data* filled in. For example, if the user enters the command "Show me all employees with this job", with a cell in column `job` selected with value "Sales Representative", then the pre-processed command would be "Show me all employees with Job having value 'Sales Representative'". Our experiments show that this sort of format for the substituted reference has good results when sent through our neural models.

Neural Generation

In the first stage of our algorithm, we take an input natural language query, as well as a database schema, and feed inputs to several neural models to generate SQL queries. Our goal is to sample multiple models in order to maximize the likelihood that a returned query will match the user's intent.

All models chosen are transformer-based architectures, as discussed above. The inputs to each model are sequences of tokens representing the natural language. Each trained model applies multiple transformer layers, which make heavy use of multi-attention in both the encoding and decoding steps. Critical to the performance of each model is the inclusion of the *database schema*, which allows the model to make use of table and column relationships when outputting SQL. For our models, we consider the schema to be a mapping between tables in the database and column names - for simplicity, we do not expose primary key/foreign key relationships or data types to the model. We choose per-model sampling parameters to, such as search method, temperature, and top-k, to encourage a diversity of SQL candidates without sacrificing quality.

Given the prohibitive cost of acquiring enough user data to train deep models, we use models that were evaluated on the Spider dataset, which includes 10,000+ queries over 200 databases. The goal of the Spider challenge is to develop natural language interfaces

to cross-domain databases. We use this dataset to run benchmarks on our methodologies. The complexity and realistic nature of Spider makes it a useful representation of the space of potential natural language/schema/query combinations.

Motivated by this, we also choose models which have been proven to perform extremely well on this dataset: RAT-SQL and T5, as evidenced by the Yale semantic parsing leaderboard [18]. In our system's default settings, we sample 4 queries from Codex, 2 queries from RAT-SQL, and 2 queries from T5. We cover each model in turn.

Codex

Our main tool for generation is OpenAI's Codex is a large scale transformer architecture matching GPT-3. OpenAI's Codex is a general-purpose programming model, meaning that it can be applied to essentially any programming task (though results may vary). Codex is currently used to power Github Copilot, a powerful code completion tool for Visual Studio Code.

In contrast to other models, Codex has not been pre-trained on only SQL, but on a very large dataset from Github that includes primarily Python code. However, we found that Codex generalized very effectively to the natural language to SQL domain, with minimal hyperparameter tuning required.

In order for Codex to properly respond and generate valid SQL data points, appropriate context must be provided. The context, in this case, consists of example natural language queries as well as the corresponding SQL commands to provide Codex an idea of what the response should look like. Thus, extensive benchmarking was required to properly determine the context.

In order to properly provide context, we use the Spider dataset to sample queries. Spider is a large-scale complex and cross-domain semantic parsing and text-to-SQL dataset. It consists of 10,181 questions and 5,693 unique complex SQL queries on 200 databases with multiple tables covering many different domains. Thus, we are able to generate database schema for tables from the different domains as well as natural language/SQL query pairs that are valid for that particular database.

Sample input to codex:	
Database Schema {	<pre># Table singer, columns = ['Singer_ID', 'Name', 'Country', 'Song_Name', 'Song_release_year', 'Age'] # Table concert, columns = ['concert_ID', 'concert_Name', 'Theme', 'Stadium_ID', 'Year'] # Table singer_in_concert, columns = ['concert_ID', 'Singer_ID']</pre>
Sample NL/SQL 1 {	<pre># What is the total number of singers?" # SELECT COUNT(*) FROM singer"</pre>
Sample NL/SQL 2 {	<pre># Show name, country, age for all singers ordered by age from the oldest to the youngest. # SELECT Name, Country, Age FROM singer ORDER BY Age DESC</pre>
Given Query {	<pre># What are the names, countries, and ages for every singer in descending order of age? SELECT</pre>

Figure 3: Sample Context provided to Codex

Figure 3 shows an example of a sample context provided to Codex. This consists of the database schema we are working with, as well as two sample natural language queries (NL) with their corresponding correct SQL generation. This is followed by the question we are attempting to answer via Codex.

Three types of contexts were fed into Codex as follows:

- 1) **Predicted Context:** Predictions made by Codex are added in as context on subsequent turns. This results in a best accuracy of ~47%, however, if an incorrect prediction is made, it cascades into generating more incorrect predictions down the road as codex infers the incorrect prediction as a positive context sample.
- 2) **Oracle Context:** Correct (or “Oracle”) predictions pulled from the Spider dataset are passed in as context to Codex. This results in a high accuracy of ~76%, however, this cannot be translated into the real world as such oracle predictions would not be available in the real world.
- 3) **Targeted Context:** This is a combination of the above approaches. Based on the user’s database schema, we search for the most closely related oracle queries. This results in a lower accuracy of ~55%, however, this does not have the issue with cascading incorrect predictions and can be translated into the real world.

The above three contexts were tested on questions with difficulty levels ranging from easy to hard based on complexity and the results are detailed in the Results section below. Based on practicality, the targeted context methodology is used going forward.

RAT-SQL + GAP + NAT-SQL

RAT-SQL, applies *relation aware* schema encoding in order to synthesize queries that take into account table relations and linking. Although we are not using schema relations as input, we still found that RAT-SQL generalizes very effectively to unseen schemas. Additionally, we use a model that has been pre-trained with GAP, a data generation method. Finally, we use NAT-SQL as an intermediate language target that has been shown to improve the quality of arbitrary NL to SQL models. We use *beam search* and sample the top two beams from RAT-SQL+GAP+NAT-SQL as candidates.

T5 + PICARD

Our implementation also samples multiple queries from T5, a general transformer language model. We choose a model that was initially trained on the WikiSQL dataset, but also augment it with Picard, a constrained decoding scheme that makes sure that decoded sequences satisfy the lexical and syntactic constraints of SQL. This approach has been proven to work extremely well on the Spider dataset. Critically, we also noticed that T5 + Picard is able to synthesize syntactically correct SQL subqueries, which Codex struggled with. We use *nucleus sampling* to generate potential sequences, with a top p parameter of 90%.

Post-processing and Neural Ranking

The 8 generated output queries from the above models are collected in a list and fed to the pre-processing and ranking steps. The goal of this stage is to improve the quality of the results and maximize the likelihood that we can suggest the most relevant query to the user. Given an input set of generated SQL queries, post-processing can be broken down into four steps: deduplication, column+table name rewriting, removal of invalid queries, and prettifying.

We first deduplicate queries, that is, removing queries that appear identically multiple times in our set. Next, we apply *column/table name rewriting*. We noticed that the outputs of our generation models tended to sometimes be *almost* correct, but would often involve incorrect column names/table names. For example, in a schema with table name "employees", we noticed that many queries would look like "SELECT * FROM employee". To fix such misspellings, we apply the following procedure. First, we parse the query into an AST, which we were able to do with most queries (we noticed this was possible with 1024/1041 predicted queries from the Spider dataset). Next, we rewrite any incorrectly referenced *table* names to their closest equivalent in the schema, using the

string matching library `fuzzywuzzy` (which uses edit distance based string similarity algorithms). Next, we collect the referenced table names into a set. Finally, we rewrite incorrectly referenced *column* names to their closest equivalent, in all columns present in the referenced table names. The procedure guarantees that a SQL query will at least reference valid table names + column names, which we qualitatively noticed was the main source of errors in generated queries. We apply this procedure to each query in the list, and therefore create a list of fixed queries.

However, to be certain that we are only suggesting queries that would actually run on the input data, we also *test* queries on the database connection. While not possible in general, because we consider only "`SELECT`" queries, and simple SQLite databases for our user studies, we are able to quickly check that a query is valid by actually running it. To do this, we send the current db connection info to the backend, update each query to end on a "`LIMIT 2`" (max limit of 2 queries) parameter, and execute it over the database connection with the `sqlalchemy` library. Queries that error out are removed from the set. Finally, we prettify the results: convert queries to uppercase, apply indentation, and standardize delimiters. The output of our post-processing is a set of unique fixed, validated, and prettified SQL queries that are ready to be ranked.

Post-processed queries are then fed into a separate neural model, the *ranker*, to be scored and sorted. To collect training examples for this ranker, we log instances of suggestion generation and executed queries from the application. For a given requested NL query, db schema, and editor interaction state, we can label its corresponding candidate SQL query as *positive* or *negative* based on if it was what the user wanted. By monitoring this throughout our user studies and other demos, we were able to collect a set of around 100 examples.

Our neural ranker uses a lightweight feedforward architecture with sigmoid head to predict the likelihood of the combination of a given NL, schema and SQL query. The network consumes embeddings created from the 3 generative models: in Codex's case, we call a dedicated endpoint for this; for the other two models, we take the average of the hidden states outputted by the decoder models. These embeddings are concatenated together, and concatenated with an *interaction* embedding that encodes features from recent interaction the user has had with the app. This final embedding is fed through a feedforward network consisting of 3 dense layers of 256 hidden units each, and an output layer of one dimension with sigmoid activation. We minimize the binary cross-entropy between this output and the actual predicted value. Our model was optimized with Adam, with a learning rate of 0.01 over 20 epochs. On our limited dataset of 100 examples, our model achieves around 75% validation accuracy in predicting whether a suggestion would be positive: we note that there was indeed a lot of noise present in the dataset, given that

most of it was collected while our users were completing pre-determined tasks. We hope that further data, and refinement of our positive/negative example conditions, can improve our model's performance and generalizability.

To generate the interaction embedding relative to a SQL query, we consider several features that associate the query with logged interaction information. First, the `"tables_present_in_query"` feature counts the number of table names in the SQL query that the user has also clicked in the sidebar in the last 5 minutes. `"columns_present_in_query"` is defined in the same way for column names. The `"table_currently_viewed"` feature is set to 1 if any table referenced in the SQL query is currently present in the output table view, and 0 otherwise. We also incorporate selection references: `"column_currently_selected"` is 1 if there is a column currently selected by the user that is also referenced in the SQL query, and 0 otherwise. Finally, `"time_taken_to_write"` is a continuous feature defined as how much time the user took to write the query, which we approximate as the number of seconds the JS editor was in an active state before the *Generate Suggestions* button was pressed.

We argue that recent actions the user has had with the tool, not just the direct NL typed, are important in clarifying intent and improving suggestion quality. In the future, we hope to investigate *broader* representations of user interactions: for example, treating the entire interaction process as a *sequence* of states, and using a recurrent or transformer model to process them into an interaction embedding, could improve our recommendations further.

The output of this model is a given probability for each possible SQL query in our set, which is used to rank them before being sent back to the tool. An important aspect of our model is that it is also *re-trained* given batches of new examples logged from the tool. For personalization, we define a *positive training example* as a tuple of input specifications, interactions, and a query that was both executed on the data and then either copied, or was not replaced with a different suggested query. We define a *negative training example* as a tuple where the SQL query did not satisfy these criteria. By logging this information we are able to continuously update the model to best understand the user's interaction behavior with the app.

System Design

Our system is divided into two main components: backend neural models services, and frontend user interfaces. The backend component takes in natural language input and outputs top formal SQL queries. Based on the functionality of backend services, it's

further divided into pre-processing, neural networks models, and post-processing components. The pre-processing component trims natural languages with the information of table schema into columns and databases of interest. The neural network models receive parsed natural language from the pre-processing component and outputs potential formal SQL queries. The post-processing component gives the ranking for multiple SQL queries from the neural network model and sends back the result to the user interface.

The system runs as a desktop application, with backend components running on several servers. We build our frontend on top of `sqlectron-gui`, an open source desktop application written in ElectronJS. We choose to implement our functionality within `SQLectron` to robust SQL desktop application. The frontend components have two main functions. One is converting natural language into lists of formal SQL queries. The other is executing queries based on users' choices. The injected neural network-based model on the user interface can convert natural language to a selection of top formal SQL queries. The other SQL GUI-based function on the user interface that enables users to execute the target query and shows the query result on the user interface.

Our backend is written in Python, with calls handled over HTTP through a FastAPI (`gunicorn`) server. The main entry point into this server is a POST endpoint that takes in a natural language query and table schema as a dictionary, and returns a list of processed, ranked queries. In order to ensure the stability and efficiency of our model, the non-Codex neural models for generation and ranking run on a GPU cluster, which is accessed over SSH. Our code uses the Python library `Fabric` to quickly access this cluster, query the API running there, and collect the results. Our post-processing makes use of several Python libraries: `sqlparse` to parse the SQL into an Abstract Syntax Tree (AST) that we can manipulate for the table/column name fixing, and `sqlalchemy` to execute (limited) queries against the db connection to validate them against the database. Throughout the backend processing, we also make heavy use of the library `fuzzywuzzy` to perform approximate string matching.

Communication between the user interface and backend services includes passing the information from the user interface to the backend, receiving the information from the backend to the user interface, and also logging user behavior for optimizing ranking services into a JSON db store. Table schema and natural language input are sent from the user interface to the backend component. After all the phases of backend service, including pre-processing, neural models and post-processing, the list of the top-ranking SQL queries are sent from the backend to the front-end interface. In addition, users' clicks are logged to serve as additional information for data augmentation in our custom ranking model.

User Interface Design

In the User Interface (UI) design, we built our UI components based on *SQLectron* GUI. The additional functionalities are modified to show the top-ranked formal SQL queries and to interact with users' behavior. *SQLectron* is a GUI tool that connects to SQL databases, takes in SQL query, and returns the query results to the users. A new dialog box showing the lists of top-ranking SQL queries is created to show the result from our backend components. In addition, a copy function is added for users to choose the formal SQL query and copy it to the edit box for further editing. The UI also tracks user statistics in order to provide contextual information to our neural ranker.

The screenshot displays the application's user interface. At the top left, a code editor shows a generated SQL query: `-- Generated from "Fetch all invoices billed in Germany"` and `SELECT * FROM invoices WHERE BillingCountry = 'Germany';`. Below the editor are buttons for "Copy to Clipboard" and "Record Command". To the right, a dialog box prompts the user to "Click any suggestion to copy it to the editor." and shows the same SQL query. Below the dialog is a "Generate new suggestions" button. At the bottom, a table displays the results of the query, with columns for InvoiceId, CustomerId, InvoiceDate, BillingAddress, BillingCity, BillingState, BillingCountry, BillingPostalCode, and Total. The table contains 14 rows of data.

InvoiceId	CustomerId	InvoiceDate	BillingAddress	BillingCity	BillingState	BillingCountry	BillingPostalC...	Total
1	2	2009-01-01 00:...	Theodor-Heuss-...	Stuttgart	NULL	Germany	70174	1.98
6	37	2009-01-19 00:...	Berger Straße 10	Frankfurt	NULL	Germany	60316	0.99
7	38	2009-02-01 00:...	Barbarossastraß...	Berlin	NULL	Germany	10779	1.98
12	2	2009-02-11 00:...	Theodor-Heuss-...	Stuttgart	NULL	Germany	70174	13.86
29	36	2009-05-05 00:...	Tauentzienstraß...	Berlin	NULL	Germany	10789	1.98
30	38	2009-05-06 00:...	Barbarossastraß...	Berlin	NULL	Germany	10779	3.96
40	36	2009-06-15 00:...	Tauentzienstraß...	Berlin	NULL	Germany	10789	13.86
52	38	2009-08-08 00:...	Barbarossastraß...	Berlin	NULL	Germany	10779	5.94
67	2	2009-10-12 00:...	Theodor-Heuss-...	Stuttgart	NULL	Germany	70174	8.91
95	36	2010-02-13 00:...	Tauentzienstraß...	Berlin	NULL	Germany	10789	8.91
104	38	2010-03-29 00:...	Barbarossastraß...	Berlin	NULL	Germany	10779	0.99
127	37	2010-07-13 00:...	Berger Straße 10	Frankfurt	NULL	Germany	60316	1.98

Figure 4: Screenshot of the application. Given the command "Fetch all invoices billed in Germany", we generate a SQL suggestion, which the user has copied to their editor and executed to see records.

Speech Recognition

We additionally implement a voice control mechanism for users to record commands instead of typing. When this button is pressed, we capture audio and send it to Google's Speech-to-Text API to convert the command audio into text. The resulting command then

gets pasted into the editor, so that it can be viewed by the user and any modifications can be directly made before generating SQL suggestions.

Results

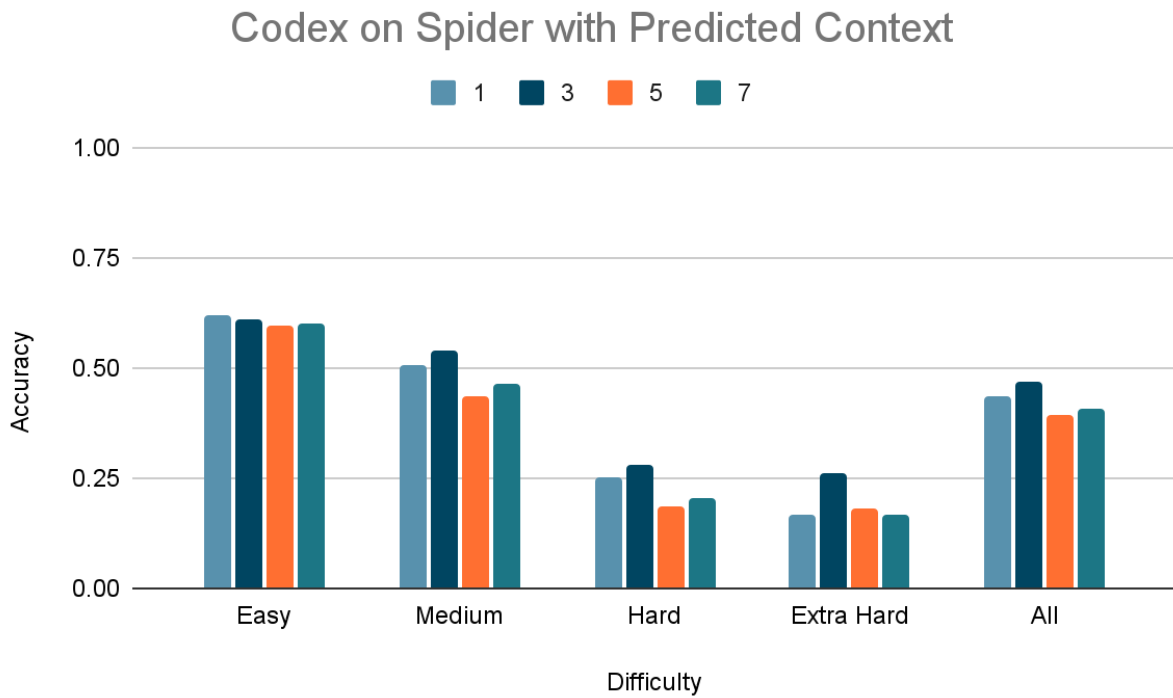
We break the evaluation of our approach into two components - a benchmarking study of Codex on the Spider dataset, and a qualitative user study to measure the effectiveness of our application as a whole.

Model Benchmark

As we can see from the results of benchmarking OpenAI Codex on the Spider dataset in 3 different settings, the best performing is with 5 oracle contexts while the worst performing is with 7 targeted contexts. Overall, we see that OpenAI Codex performs best with oracle context compared to predicted context and targeted context. From these results, we can conclude that to get the best performance on the Spider dataset, we should use oracle context as a hyperparameter to be tuned. However, this does not translate well into the real world and in order to avoid the model from suffering from cascading failures due to earlier errors in code generation, we conclude that the best approach is using targeted context in most real-world settings.

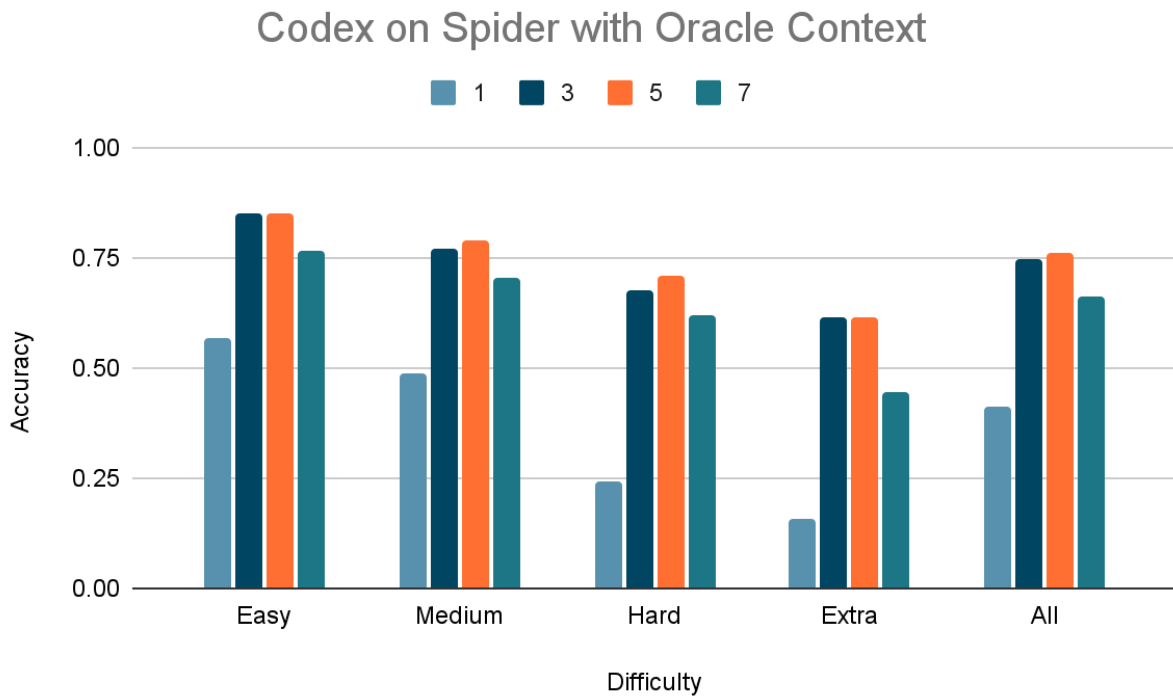
Difficulty	Easy	Medium	Hard	Extra	All
Count	248	446	174	166	1034

Figure 5: Number of queries used for accuracy generation



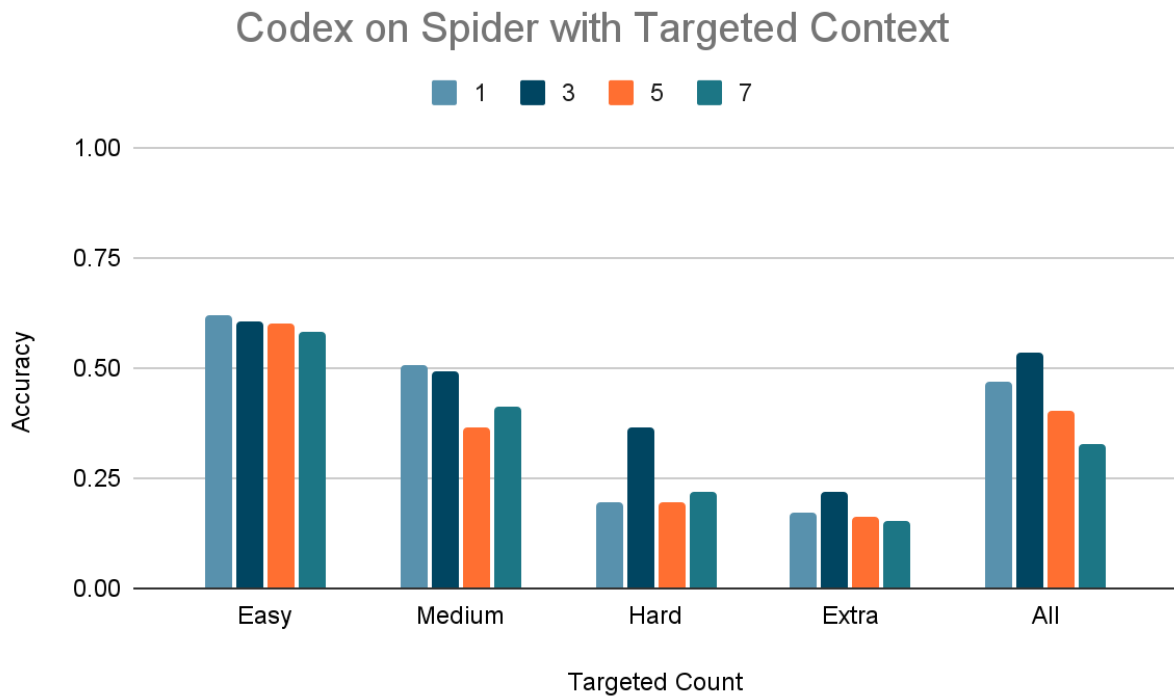
	Difficulty				
Predictions	Easy	Medium	Hard	Extra	All
1	0.621	0.507	0.253	0.169	0.437
3	0.609	0.540	0.282	0.259	0.468
5	0.597	0.437	0.184	0.181	0.392
7	0.601	0.466	0.207	0.169	0.407

Figure 6: Codex on Spider with Predicted Context



	Difficulty				
Oracle	Easy	Medium	Hard	Extra	All
1	0.569	0.487	0.241	0.157	0.412
3	0.851	0.769	0.678	0.614	0.749
5	0.851	0.791	0.707	0.614	0.763
7	0.766	0.704	0.621	0.446	0.663

Figure 7: Codex on Spider with Oracle Context

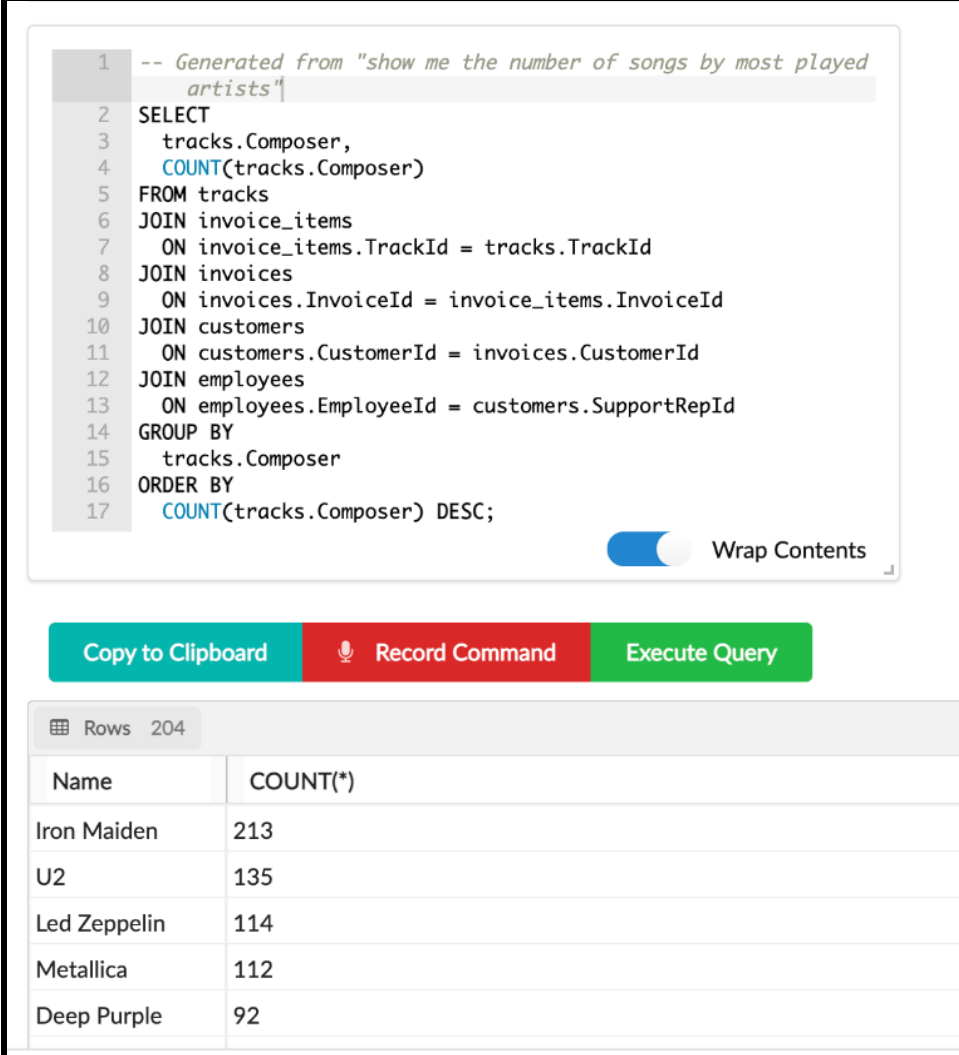


	Difficulty				
Targeted	Easy	Medium	Hard	Extra	All
1	0.621	0.507	0.194	0.171	0.467
3	0.605	0.491	0.364	0.217	0.536
5	0.601	0.367	0.197	0.163	0.402
7	0.581	0.411	0.217	0.153	0.327

Figure 8: Codex on Spider with Targeted Context

Sample Queries

We conducted extensive testing regarding complex queries to test the capabilities of the model. Our model was able to successfully perform multiple inner joins for a query to respond properly. Figure 9 illustrates a query requiring four inner joins on three distinct tables to execute properly. Such a query would require skillful mastery of SQL to properly construct, however, our model was able to do so just via natural language.



The screenshot displays a SQL query editor interface. At the top, a comment indicates the query was generated from the natural language prompt: "show me the number of songs by most played artists". The query itself is a SELECT statement that joins the tracks, invoice_items, invoices, and customers tables, and also includes an employees table. It groups the results by the composer's name and orders them by the count of songs in descending order. Below the query editor, there are three buttons: "Copy to Clipboard", "Record Command", and "Execute Query". The "Execute Query" button has been clicked, resulting in a table with 204 rows. The table has two columns: "Name" and "COUNT(*)". The top five rows of the table are shown, listing the composers Iron Maiden (213), U2 (135), Led Zeppelin (114), Metallica (112), and Deep Purple (92).

```

1  -- Generated from "show me the number of songs by most played
   artists"
2  SELECT
3      tracks.Composer,
4      COUNT(tracks.Composer)
5  FROM tracks
6  JOIN invoice_items
7      ON invoice_items.TrackId = tracks.TrackId
8  JOIN invoices
9      ON invoices.InvoiceId = invoice_items.InvoiceId
10 JOIN customers
11     ON customers.CustomerId = invoices.CustomerId
12 JOIN employees
13     ON employees.EmployeeId = customers.SupportRepId
14 GROUP BY
15     tracks.Composer
16 ORDER BY
17     COUNT(tracks.Composer) DESC;

```

Copy to Clipboard Record Command Execute Query

Rows 204

Name	COUNT(*)
Iron Maiden	213
U2	135
Led Zeppelin	114
Metallica	112
Deep Purple	92

Figure 9: Generated SQL code for NL query
 "Show me the number of songs by the most played artists"

User Studies

We conducted user studies to evaluate whether this technology is useful in the day-to-day life of data scientists and engineers. Our experimental setup is using Zoom as a platform for participants to access our technology through its remote-control feature for remote interviews and as a recording instrument to measure the participants' usage behavior for both remote and in-person interviews. Appendix B depicts the script used to conduct all the user studies.

Our user study methodology consists of two parts: behavioral interview and task completion. We use the net promoter score as the success criteria of the technology's usefulness. We use our behavioral interview to surface pros and cons of our technology to incorporate as future improvements. At the end of each interview, we will provide a survey where the main takeaway is a net promoter score that asks each participant if they would recommend this technology to their colleagues. The main tasks from our user study is to solve two SQL questions, one easy and one hard difficulty, where user study participants are given the database schema and a question to be solved in both raw SQL and English (natural language).

We interviewed 14 participants with varying SQL experience from 0 to 3 years, with an average of 1.67 years SQL experience. Their academic backgrounds are computer science, data science, industrial engineering, and mechanical engineering, while their professional backgrounds are software engineering, data science, project management, and mechanical engineering.

Based on time taken for the task completion (Appendix C), we see that the average speed of using natural language instead of SQL on the easy task is 3.20x, while that of the hard task is 4.94x. In addition, we asked them how likely they would recommend this to their friend. We got an enormous amount of positive feedback from our users with 100% positive feedback: 42.9% (8) extremely likely and 57.1% (6) likely (Figure 10).

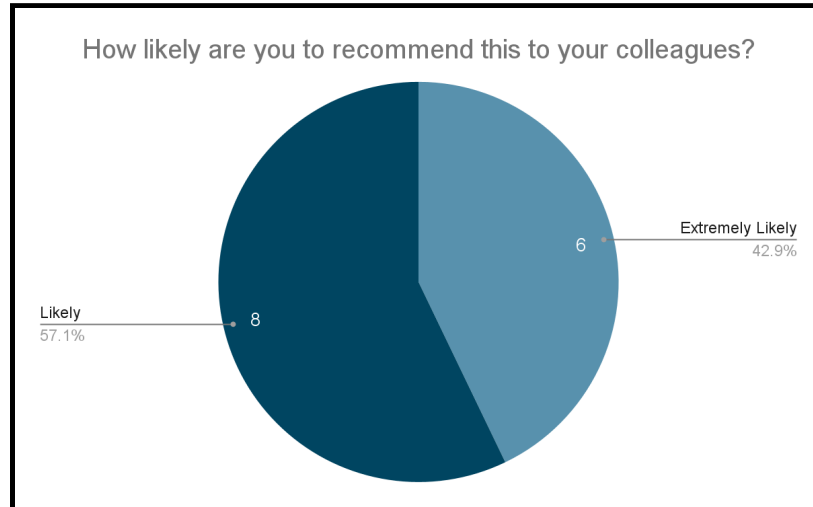


Figure 10: Net Promoter Score

The main pros that have surfaced during behavioral interviews are that our technology was useful during times when they have to construct complicated SQL queries, that it saves time to write SQL as it is a good starting reference to build upon, and that it helps learn SQL.

The main cons that have surfaced during behavioral interviews are that our technology should have been in web interface instead of a desktop interface as it is easier to access, and that they wouldn't trust the generated SQL queries as they have no way to verify its validity other than executing it, which makes it hard to debug.

Conclusion

As SQL's ubiquity grows with the rise in the number of data science professionals, SQL's unintuitiveness still poses a major barrier to entry. We introduce a methodology, as well as a user interface, using several state-of-the-art models to empower data scientists to focus on solving business problems by interacting with databases using natural language rather than wrangling with SQL queries. We have demonstrated the relevance and importance of this problem domain, garnered current research through literature review, and explained our system and user study methodologies.

We primarily use OpenAI's Codex, RAT-SQL, and T5 to generate formalized SQL queries from natural language. However, these models require appropriate context to be provided and thus, we introduce targeted context generation based on the user's database schema and benchmark it upon the Spider dataset. We also introduce a novel neural ranking

model that ranks multiple SQL queries that originated from different state-of-the-art neural models. This involves tracking user interactions through our user interface and augmenting the data to improve our baseline ranking model.

Both these novel methodologies allow the user to generate useful and personalized SQL code for their natural language query. The feedback garnered from the user studies has been positive, with the majority stating they would highly recommend our tool to other data scientists. Going forward, we aim to create new neural models that would be designed specifically for this particular use case. This would require a large amount of compute power to be available alongside new datasets to train upon.

We also aim to improve upon our user interface to allow more capacity to capture more user interaction data. This data would serve as data augmentation which would be fed directly into our neural ranker model in order to generate results that are more personalized. With these new capabilities, we expect to increase our overall accuracy as well as our net promoter score.

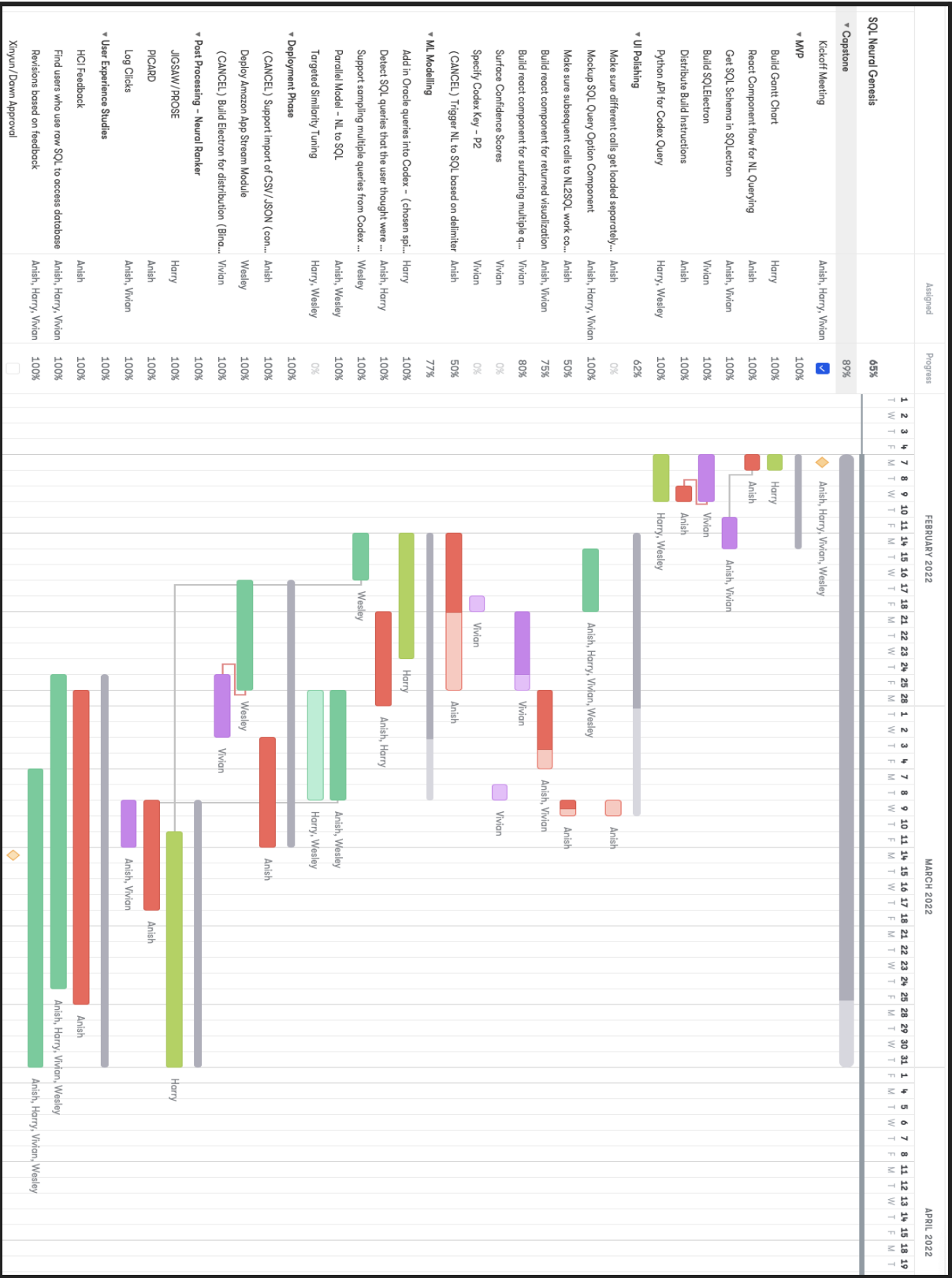
References

1. Coursera. "What is a Data Scientist? Salary, Skills, and How to Become One". Coursera, 02 May 2022.
<https://www.coursera.org/articles/what-is-a-data-scientist>
2. Indeed Career Guide. "Top 12 Skills You Need to Become a Data Scientist". Indeed, 30 Dec 2021.
<https://www.indeed.com/career-advice/resumes-cover-letters/skills-for-a-data-scientist>
3. Hale, Jeff. "The Most in Demand Skills for Data Scientists". KDnuggets, Nov. 2018.
<https://www.kdnuggets.com/2018/11/most-demand-skills-data-scientists.html>
4. U.S. Bureau of Labor Statistics. "Occupational Employment and Wages, May 2021; 15-2051 Data Scientists". U.S. Bureau of Labor Statistics, May 2021.
<https://www.bls.gov/oes/current/oes152051.htm>
5. U.S. Bureau of Labor Statistics. "Computer and mathematical occupations". U.S. Bureau of Labor Statistics, May 2020.
<https://www.bls.gov/ooh/about/data-for-occupations-not-covered-in-detail.htm#Computer%20and%20mathematical%20occupations>
6. World Economic Forum. "The Future of Jobs Report 2020". World Economic Forum, 20 Oct 2020.
<https://www.weforum.org/reports/the-future-of-jobs-report-2020>
7. Wikipedia. "SQL". Wikipedia, 2022. <https://en.wikipedia.org/wiki/SQL>
8. ISO. "ISO/IEC 9075-1:2016 Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework)". ISO, Dec 2016.
<https://www.iso.org/standard/63555.html>
9. Gartner Research. "Market Share: Database Management Systems, Worldwide, 2020". Gartner, 06 May 2021. <https://www.gartner.com/en/documents/4001330>
10. Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin. "Attention is All You Need". Neural Information Processing Systems (NIPS), 06 Dec 2017.
<https://arxiv.org/abs/1706.03762>
11. Salvator, Dave. "H100 Transformer Engine Supercharges AI Training, Delivering Up to 6x Higher Performance Without Losing Accuracy". NVIDIA, 22 March 2022.
<https://blogs.nvidia.com/blog/2022/03/22/h100-transformer-engine/>
12. OpenAI. "OpenAI Codex". OpenAI, 10 Aug 2021.
<https://openai.com/blog/openai-codex/>
13. Herzig, Jonathan, Pawel Krzysztof Nowak, Thomas Muller, Francesco Piccinno, Julian Martin Eisenschlos. "TaPas: Weakly Supervised Table Parsing via

- Pre-training". Association for Computational Linguistics (ACL), 21 Apr 2020. <https://arxiv.org/abs/2004.02349>
14. Wang, Bailin, Richard Shin, Xiaodong Liu, Oleksandr Polozov, Matthew Richardson. "RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers". Association of Computational Linguistics (ACL), 21 Aug 2021. <https://arxiv.org/abs/1911.04942>
 15. Gan, Yujian, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John Drake, Qiaofu Zhang. "Natural SQL: Making SQL Easier to Infer from Natural Language Specifications". Empirical Methods in Natural Language Processing (EMNLP), 11 Sep 2021. <https://arxiv.org/abs/2109.05153>
 16. Chen, Mark, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, Wojciech Zaremba. "Evaluating Large Language Models Trained on Code". arXiv:2107.03374 [cs.LG], 14 Jul 2021. <https://arxiv.org/abs/2107.03374>
 17. Brown, Tom B., 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, Dario Amodei. "Language Models are Few-Shot Learners". arXiv:2005.14165, 22 Jul 2020. <https://arxiv.org/abs/2005.14165>
 18. Yu, Tao, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, Dragomir Radev. "Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task". Empirical Methods in Natural Language Processing, 02 Feb 2019. <https://arxiv.org/abs/1809.08887>
 19. Yu, Tao, Rui Zhang, He Yang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, Vincent

- Zhang, Caiming Xiong, Richard Socher, Walter S Lasecki, Dragomir Radev. "CoSQL: A Conversational Text-to-SQL Challenge Towards Cross-Domain Natural Language Interfaces to Databases". Empirical Methods in Natural Language Processing (EMNLP), 11 Sep 2019. <https://arxiv.org/abs/1909.05378>
20. Zeng, Jichuan, Xi Victoria Lin, Caiming Xiong, Richard Socher, Michael R. Lyu, Irwin King, Steven C.H. Hoi. "Photon: A Robust Cross-Domain Text-to-SQL System". Association of Computational Linguistics (ACL), 03 Aug 2020. <https://arxiv.org/abs/2007.15280>
21. Narechania, Arpit, Adam Fourney, Gonzalo, Ramos, Bongshin Lee. "DIY: Assessing the Correctness of Natural Language to SQL Systems". Association for Computing Machinery (ACM): Intelligent User Interfaces, 14 Apr 2021. <https://dl.acm.org/doi/10.1145/3397481.3450667>
22. Chasins, Sarah E., Elena L. Glassman, Joshua Sunshine. "PL and HCI: Better Together". Association of Computing Machinery (ACM): Communications of the ACM, Aug 2021. <https://cacm.acm.org/magazines/2021/8/254314-pl-and-hci/fulltext>

Appendix A: Gantt Chart



Appendix B: User Study Script

A. Project Introduction

- 1) **Thank you.** Thank you for agreeing to participate. My name is ____ and I'll be walking you through the session today.
- 2) **Have you ever done a user test before?** [If yes, have them describe it in a few sentences]
- 3) **Think out loud.** We ask that you think aloud as you go through the tasks so that we can understand your thoughts as well.
- 4) **Evaluating concepts, not you.** To be clear, we are evaluating the concepts and not you. There are no right or wrong answers.
- 5) **Encourage honesty.** Please be honest about these concepts since this is a great time in our project to make changes based on user feedback.
- 6) **What will happen?** In this session, we will show you a couple concepts, ask you to complete a few tasks and then ask you to compare and contrast them.

B. Pre-Experimental Questions

- 1) What is your familiarity with SQL?
 - a) How many years have you **written in SQL**?
 - b) What work environment have you used SQL? Academic or Industry?
- 2) How much time do you spend **writing SQL queries** on a daily basis? Or on a weekly basis?
- 3) What resources/tools/platforms/environments do you use to **write SQL queries**?
 - a) What would you first do to figure out what to write?
 - b) What or who would you ask?
 - c) Do you write SQL in raw form or use language specific libraries?
 - i) If raw form,
 - (1) What database do you use?
 - ii) If language-specific libraries,
 - (1) What programming language to access databases?
 - (2) What specific libraries with the programming language do you use?
- 4) What tools/platforms/environments do you use to *execute queries*?

C. Experiment

- 1) Easy SQL Question
 - a) Task 1: Do in **Raw SQL**
 - b) Task 2: Do in **Natural Language (English)**
- 2) Hard SQL Question
 - a) Task 1: Do in **Raw SQL**
 - b) Task 2: Do in **Natural Language (English)**

D. Post-Interview Questions

- 1) Tasks Oriented
 - a) (Recall the experiment) Could you elaborate on how you got to the result from using **Raw SQL queries**?
 - b) What do you think about using a **Raw SQL query** to get the result?
 - c) (Recall the experiment) Could you elaborate on how you got to the result from using **Natural Language (English)**?
 - d) What do you think about using **Natural Language (English)** to get the result instead of writing **Raw SQL** directly?
- 2) Comparison
 - a) **Compare and contrast** the two ways you did the tasks? How do you feel about using these two approaches? **Pros and Cons**?
 - b) If they preferred the NL2SQL:
 - i) Can you think of any tasks for which you wouldn't want to use the NL2SQL
- 3) System Optimization
 - a) User Flow (Complex or Mental effort of the process)
 - i) Is there any part in the workflow that you feel like we could **improve upon**?
 - ii) Is there any part in the workflow that you feel like we could **remove**?
 - b) User Interface
 - i) Is there any layout or button that you feel like we could **improve upon**?
 - ii) Is there any layout or button that you feel like we could **remove**?

Appendix C: User Study Data

Easy SQL Time (HH:MM:SS)	Easy NL Time (HH:MM:SS)	Easy SQL / Easy NL
00:15:35	00:03:00	5.19
00:04:05	00:01:30	2.72
00:06:20	00:01:05	5.85
00:10:00	00:01:48	5.56
00:06:41	00:01:50	3.65
00:03:21	00:01:43	1.95
00:05:12	00:02:34	2.03
00:03:56	00:01:56	2.03
00:03:27	00:01:40	2.07
00:01:40	00:01:54	0.88
00:02:48	00:01:04	2.63
00:04:54	00:00:57	5.16
00:04:19	00:01:47	2.42
00:04:02	00:01:32	2.63

Hard SQL Time (HH:MM:SS)	Hard NL Time (HH:MM:SS)	Hard SQL / Hard NL
00:18:20	00:05:00	3.67
00:03:05	00:00:20	9.25
00:02:00	00:00:50	2.40
00:03:00	00:00:23	7.83
00:02:50	00:00:30	5.67
00:02:39	00:00:59	2.69
00:07:57	00:01:57	4.08

00:02:29	00:01:12	2.07
00:01:35	00:00:20	4.75
00:05:13	00:01:23	3.77
00:00:59	00:00:46	1.28
00:07:05	00:00:22	19.32
00:02:05	00:01:43	1.21
00:01:53	00:01:34	1.20