GenEdit: Compounding Operators and Continuous Improvement to Tackle Text-to-SQL in the Enterprise

Karime Maamari
Distyl AI
karime@distyl.ai

Connor Landy
Distyl AI
connor@distyl.ai

Amine Mhedhbi
Polytechnique Montreal
amine.mhedhbi@polymtl.ca

ABSTRACT

Recent advancements in Text-to-SQL, driven by large language models, are democratizing data access. Despite these advancements, enterprise deployments remain challenging due to the need to capture business-specific knowledge, handle complex queries, and meet expectations of continuous improvements. To address these issues, we designed and implemented GenEdit: our Text-to-SQL generation system that improves with user feedback. GenEdit builds and maintains a company-specific knowledge set, employs a pipeline of operators decomposing SQL generation, and uses feedback to update its knowledge set to improve future SQL generations.

We describe GenEdit's architecture made of two core modules: (i) decomposed SQL generation; and (ii) knowledge set edits based on user feedback. For generation, GenEdit leverages compounding operators to improve knowledge retrieval and to create a plan as chain-of-thought steps that guides generation. GenEdit first retrieves relevant examples in an initial retrieval stage where original SQL queries are decomposed into sub-statements, clauses or subqueries. It then also retrieves instructions and schema elements. Using the retrieved contextual information, GenEdit then generates step-by-step plan in natural language on how to produce the query. Finally, GenEdit uses the plan to generate SQL, minimizing the need for model reasoning, which enhances complex SQL generation. If necessary, GenEdit regenerates the query based on syntactic and semantic errors. The knowledge set edits are recommended through an interactive copilot, allowing users to iterate on their feedback and to regenerate SQL queries as needed. Each generation uses staged edits which update the generation prompt. Once the feedback is submitted, it gets merged after passing regression testing and obtaining an approval, improving future generations.

1 INTRODUCTION

Recent advancements in Text-to-SQL have broadened access to data for a wider range of users while enabling faster query authoring using database management systems (DBMSs). At the core of these advancements are large language models (LLMs), which achieve unprecedented accuracy through in-context learning (ICL) [19, 32] and fine-tuning [8, 13, 15]. Nevertheless many of these approaches are not readily deployable in the enterprise setting. From our experience with customer deployments, a Text-to-SQL solution ought to: i) understand *external knowledge*, *e.g.*, a company's specific terminology and processes; ii) handle *large query complexity* due to the schemas of data warehouses and the inherent complexity of the queries themselves; and iii) improve *systematically over time*.

Consider the following query, denoted as $Q_{fin-perf}$, which will serve as a running example. This query originates from a data analyst working at a holding company with ownership in multiple sports organizations.[1] The query is defined as follows:

> Identify our 5 sports organisations with the best and worst QoQFP in Canada for Q2 2023.

The query asks for QoQFP, *i.e.*, quarter-over-quarter financial performance. The QoQFP acronym within this company has a very specific meaning and associated calculations; the query cannot be answered without understanding such specifics. An LLM might in fact have a different interpretation of this acronym due to its pre-training. Furthermore, the query's equivalent SQL has very high complexity. The equivalent SQL to $Q_{fin-perf}$ is shown in Appendix §A with appropriate domain and data masking. Such highly complex queries have been reported as a core Text-to-SQL challenge [9] and differ significantly to the common queries found in current popular public benchmarks [14, 31]. Finally, data analysts expect that even if the initial generated SQL fails, the query generation should improve over time.

GenEdit is our purpose-built system to address the challenges of enterprise scenarios similar to the prior example. First, GenEdit captures the specific context of a company by building and maintaining a knowledge set. The set is a *view* containing pairs of: i) natural language; and ii) SQL examples, natural language instructions (or hints) for generation, and database schemas. Second, GenEdit handles the complexity challenge by relying on a multi-operator pipeline using LLM calls to decompose the problem and generates a step-by-step plan thereby reducing the need for LLM reasoning. Finally, GenEdit improves over time as users provide free text feedback leading to editing suggestions to the knowledge set and prompts. This in turn improves future generations. GenEdit contains an edits recommendation module that supports subject matter experts in improving the system without understanding its internals.

Previous work on Text-to-SQL shares similarities with our approach. For example, GenEdit pipelines are decomposed into operators and rely on few-shot examples for generation [19, 21, 24]. Our approach however deviates in important ways. GenEdit operators do not retrieve or produce separate context fed to the LLM for generation; instead, they compound, *e.g.*, the choice of relevant examples informs the choice of instructions to retrieve. GenEdit also imposes a very different intermediate representation fed to the model to generate SQL. Instead of using full Text-to-SQL pairs as examples, GenEdit decomposes examples into smaller clauses,

 15th Annual Conference on Innovative Data Systems Research (CIDR '25). January 19-22, Amsterdam, The Netherlands

[1] While $Q_{fin-perf}$ represents the actual structure of a query in production, we mask the domain and any relevant customer or user details.

which are used for the chain-of-thought (CoT) optimization [28, 32]. GenEdit first generates a CoT plan in which one or more steps describe a Common Table Expression (CTE) expected to be part of the output candidate query. GenEdit then constructs its final output by combining these CTEs as described in the plan with an added SELECT-FROM-WHERE instruction.

In the rest of the paper, we overview GenEdit's architecture (§2). We primarily cover our insights designing GenEdit, which are:

- **Compounding Operators** (§3.1):

1. *Context Expansion* (§3.1.1): we decompose SQL generation into multiple operators where the output of the prior operator is used in the subsequent one. For example, consider a list of operators retrieving relevant contextual information for the input query from the knowledge set. Let an initial operator select relevant examples of the task, *i.e.*, (natural language query, SQL) pairs. The selection of these examples informs that of relevant instructions, which are natural language descriptions in the form of hints of what to do or definition of certain terms; the selection of both then informs the choice of relevant schema elements and so on. Such context expansion is akin to *query expansion* and improves the performance of subsequent retrieval operators by adding more relevant context to the query and hence finding more relevant elements in the knowledge set.
2. *Planning for Generation* (§3.1.2): in order to generate queries of high complexity, we minimize the need for LLM reasoning. We do so by splitting the task of SQL generation into two operators. First, we generate an initial *CoT plan*, *i.e.*, a step-by-step natural language description of how to generate the output SQL query. Each step describes a SQL sub-statement, a clause, or subquery in natural language and SQL. Then, we use this plan and the retrieved knowledge as the context necessary to make a prediction.

- **Decomposing Examples** (§3.2): due to our novel CoT plan, instead of using few-shot examples as pairs of natural language and SQL queries, we decompose SQL examples repeatedly and have them at different granularity such as subqueries, clauses, or sub-statements. The user's needs are best captured through the sub-statements obtained from such decomposition. Many of the sub-statements end up being repeated across the space of expected SQL queries. As such, each step of the CoT plan describes generating a sub-statement similar to the examples.

- **Recommending Edits** (§4.1): we found a large set of SQL generation issues to stem from either: i) not understanding a particular context in the natural query (*e.g.*, use of 'our' in the running example); ii) a mistake in the decomposed examples (*e.g.*, how to do a specific calculation); or iii) retrieval accuracy in the pipeline such as missing relevant schema elements or examples. Hence, the primary approach to improve the system is to edit the knowledge set. An edit can update decomposed examples or instructions, insert new ones, or delete existing ones. It can alternatively add instructions to the retrieval and reranking operations within the pipeline.

- **Interactive Collaboration** (§4.2): Text-to-SQL is not a standalone product and instead ships in our case within an analytics engine. After a SQL generation, an SME can submit feedback and iterate on it through a user interface. Submitting feedback generates recommendations of edits to the knowledge set. The user chooses a subset if the edits to stage and regenerates the query. Users can keep iterating on their feedback until they are satisfied with the regenerated query based on their staged edits. The user then publishes the staged edits and can even make further direct edits themselves. Once staged, the edits to the knowledge set are tested for regression. Currently, these staged edits require human approval after passing regression testing. All edits due to user feedback are logged into a history that can be audited and can be used to revert back to any prior checkpoint.

2 GENEDIT ARCHITECTURE

GenEdit has two core modules: a SQL generation module and an edits recommendation module. These are shown in the system architectural diagram in Fig. 1. The SQL generation components are highlighted by the *pre-processing* and *inference* phase boxes, while the edits recommendation module contains the *feedback mechanism*, *automatic evaluation*, and *human review*. We give an overview of the SQL generation module and describe our ICL technique that has been deployed in production. The details of SQL generation and edit recommendation are in sections §3 and §4, respectively.

2.1 SQL Generation Overview

We rely on pre-processing to construct the knowledge set. At inference, we use a subset of the relevant knowledge for generation:

- ***Pre-processing*:** this phase takes as input: i) SQL queries from logs of prior executions; and ii) documents that contain domain-specific terminology and practices. As output, it produces an external knowledge set as a materialized view that contains examples, instructions, and schema elements, grouped by *user intents*. A user intent describes a particular need or request, *e.g.*, *'financial performance'* or *'TV viewership numbers'* for $Q_{fin-perf}$. These intents are mined and verified by SMEs. An important aspect of maintenance is keeping track of provenance in the view to update it as documents change. GenEdit's specific representation for examples and instructions is novel (detailed in §3.2) and emerges in part from the decomposition approach at inference. We will explain examples and instructions momentarily. The schema is obtained directly from the database or documentation from data catalogs. The schema is augmented with possible attribute values. Specifically, we add the top-5 most frequent values per attribute to the schema information.

- ***Inference*:** given a natural language input query, a pipeline made of multiple operators generates an output SQL query as follows. First, GenEdit reformulates the query using a *canonical format* (operator 1 of inference in Fig. 1) and classifies the query intents (operator 2 of inference in Fig. 1). One example of changes to the query to conform to the canonical format is to always begin with "Show me ...". Second, it retrieves the relevant knowledge from the view using the classified intents. It then relies on further retrieval operations of each component to retrieve the next as shown in operators 3, 4, and 5 in the architectural diagram. Finally, GenEdit generates a candidate query. Unlike prior approaches, we decompose generation to minimize the need for LLM reasoning by performing two model calls. The first call generates a step-by-step plan in natural language describing how to write the query and a second call uses the plan for the actual SQL generation. In case of an error, GenEdit retries generation by adding the perceived errors as context and using self-correction similar to prior work [25].

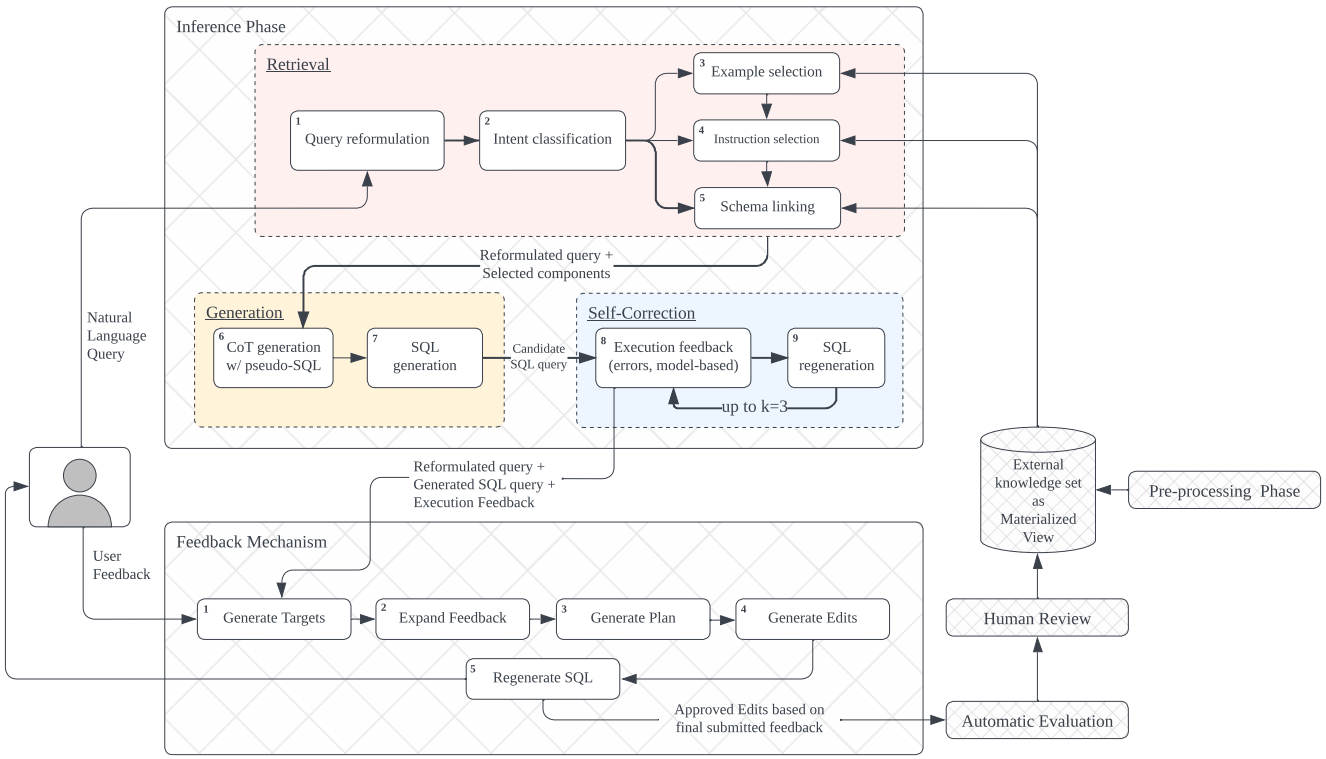


Figure 1: Overview of GenEdit architecture showcasing its pipeline operators for SQL generation and edits recommendation modules.

3 SQL GENERATION

In order to generate a candidate SQL query, GenEdit follows the inference phase in Fig. 1. Given an input query, the compounding retrieval operators and the generation planning step help in constructing a prompt similar in structure to the one shown in Fig. 2. This prompt is then used to produce one or more candidate SQL queries. If more than one candidate query is generated, GenEdit picks the *'best'* one. GenEdit then might regenerate the query up to k times based on syntactic and semantic errors.

We cover the details of the retrieval operators with context expansion and the CoT planning prior to the SQL generation. We then cover the representation used for the examples and instructions. We note that the schema representation is similar to prior approaches and therefore is omitted [16].

3.1 Compounding Operators

At inference, the pipeline classifies the intents of the input query and retrieves relevant context for generation. Each context retrieval operator relies on the output of prior ones specifically for re-ranking. We refer to this as context expansion which enables better re-ranking for the retrieval operators. In this section, we also cover how GenEdit decomposes SQL generation through planning, which in turn minimizes the need for LLM reasoning and allows us to tackle queries of much higher complexity.

3.1.1 Context Expansion. Before retrieval, the query reformulation operator (#1 in Fig. 1) reformulates the query into a chosen canonical form. Then, the intent classification operator (#2 in Fig. 1) identifies the user intents. Recall that user intents are mined in a pre-processing phase and that each intent is associated with examples, instructions, and a set of schema elements, *i.e.*, tables and columns considered relevant to the intent.

As such, the example selection operator (#3 in Fig. 1) uses the user intents to retrieve their associated examples from the view. It then retrieves further relevant examples based on the query. The operator then re-ranks all examples based on a cosine similarity score with the reformulated query. Next, the instruction selection operator (#4 in Fig. 1) retrieves instructions in a similar fashion to the example selection operator; however, it re-ranks using not just the query but also the selected examples. Finally, the schema linking operator (#5 in Fig. 1) uses LLM calls to identify relevant elements as done in prior work [22] and adds a re-ranker for filtering to manage the LLM context of the model generating SQL [15].

3.1.2 Planning for Generation. First, we construct a CoT reasoning plan using an LLM call with the reformulated input query and retrieved external knowledge [10, 26, 27]. The plan contains natural language steps of how to write the query. Each step in the plan is augmented with '*pseudo-SQL*' inspired by selected examples. Pseudo-SQL refers to a SQL sub-statement associated with a step. The substatement is written surrounded by dots (. . .) as a prefix and a suffix indicating that it is part of a larger query.

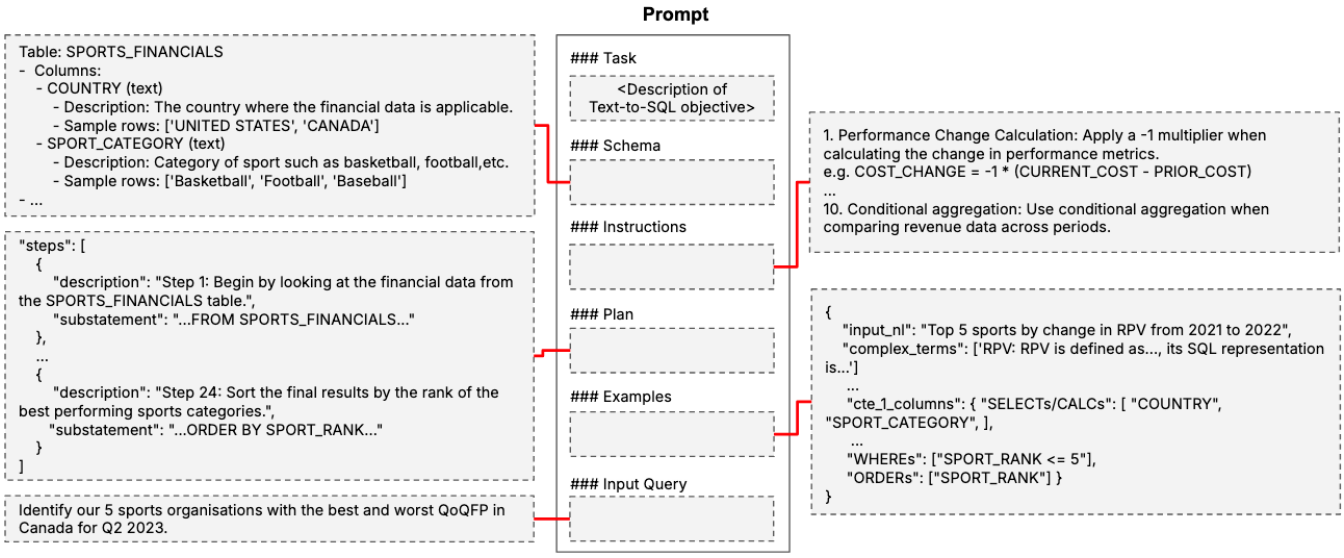


Figure 2: Example of retrieved knowledge and plan generated for $Q_{fin-perf}$, which is then used for SQL prediction.

In the prompt, the CoT plan is represented as a JSON object containing an ordered list of steps where each element is a pair of step description in natural language and pseudo-SQL. Consider the plan for $Q_{fin-perf}$ in Fig. 2, its first step is *"Begin by looking at the financial data from the SPORTS_FINANCIALS table."* with the pseudo-SQL: "`...FROM SPORTS_FINANCIALS...`". The plan in the example contains 24 steps in total.

We rely on another LLM call to generate one or more candidate SQL queries using the reformulated query, retrieved external knowledge, and CoT plan.

3.2 Decomposing Examples

In pre-processing, GenEdit creates a materialized view of examples and instructions using query logs and domain-specific documents. Below, we describe the actual representation of each component. Fig. 2 shows the structure of the various representation within a prompt for the running example query $Q_{fin-perf}$.

3.2.1 Example Representation. Our examples are SQL sub-statements and each has an associated equivalent natural language description. This is in contrast with traditional examples which are full SQL queries represented in their raw format.

The sub-statements are obtained as follows. We take as input the full SQL queries from historical logs or directly from domain experts. We first rewrite the queries to use CTEs (WITH clause with subqueries). Then, each rewritten query is decomposed into sub-queries based on its subqueries in the WITH clauses, and finally into sub-statements based on inner clauses. As a result, we represent examples in the decomposed form as shown in Fig. 2. The prompt shows various examples such as a CTE columns projection, a WHERE clause and an ORDER BY clause. We augment these sub-statements with natural language descriptions of the complex operations, tables, and sub-queries involved. For instance, in the example in Fig. 2, we have the RPV (revenue per viewer) term defined and its SQL statement, where the RPV example is associated with an intent. Our use of CTEs in the output and associated decomposition is a novel approach to task decomposition, though it has inspirations from the prior sketch-based slot-filling techniques [29].

3.2.2 Instruction Representation. Instructions for SQL generations are natural language guidelines provided to the model detailing how to interpret the input and formulate it into a correct SQL query. We obtain the instructions from example queries and documents containing domain-specific terminology and practices. We represent them in natural language with expected SQL sub-expressions when relevant. For instance, Fig. 2 shows two instructions. The first explaining how to calculate performance change *('Apply a -1 multiplier when calculating the change in performance metrics')* and the second explaining the need to use conditional aggregations *('when comparing revenue data across periods')*.

3.3 Evaluation

3.3.1 Dataset. We evaluate our framework on the BIRD benchmark [14]. BIRD is widely accepted as the most challenging Text-to-SQL benchmark available. It contains queries from 95 databases spanning multiple domains. The difficulty of BIRD is in the imprecision of its data, queries, and external knowledge, making the benchmark the most representative of the real-world Text-to-SQL problem to date. To simplify and minimize the evaluation cost, we use the dev set by sampling 10% of each database, as proposed in the evaluation of prior work [22].

3.3.2 Evaluation Metric. The BIRD benchmark relies on the the Execution Accuracy (EX) as an accuracy evaluation metrics. EX is the proportion of queries for which the output of the predicted SQL query is identical to that of the ground truth SQL query.

3.3.3 Model Selection. We use `GPT-4o` across all operators, except for schema linking, where we instead employ `GPT-4o-mini` to reduce primarily cost and then latency.

Methods	BIRD-Dev			
	Simple	Moderate	Challenging	All
CHESS [22]	65.43	64.81	58.33	64.62
MAC-SQL [24]	65.73	52.69	40.28	59.39
TA-SQL [20]	63.14	48.60	36.11	56.19
DAIL-SQL [7]	62.5	43.2	37.5	54.3
C3-SQL [5]	58.9	38.5	31.9	50.2
GenEdit	69.89	39.29	36.36	60.61

Table 1: Performance of GenEdit on the BIRD benchmark, compared to prior solutions.

Method	Sim.	Mod.	Chall.	Total
GenEdit	69.89	39.29	36.36	60.61
w/o Schema Linking	67.74	42.86	18.18	58.33 (↓ 2.28)
w/o Instructions	58.06	28.57	36.36	50.00 (↓ 10.61)
w/o Examples	69.89	35.71	9.09	59.09 (↓ 1.52)
w/o Pseudo-SQL	62.37	25.00	18.18	50.76 (↓ 9.85)
w/o Decomposition	66.67	46.43	18.18	58.33 (↓ 2.28)

Table 2: An ablation study looking at the EX change without (w/o) certain operators.

3.3.4 Results. Table 1 showcases the performance of GenEdit against prior approaches. Comparing accuracy on 10% of the BIRD benchmark dev set, GenEdit ranks second compared to open-source submissions, *i.e.*, submissions with available code and tenth compared to all submissions at 60.91%.[2] In Table 2, we present an ablation of various operators within our pipeline to showcase the benefit of each. We see instructions are providing the most benefit while examples the least. This does not minimize the usefulness of examples as they are what we use to add pseudo-SQL to the CoT plan, which provides the second most benefit.

When comparing GenEdit with another approach we developed [15], we find that our newer approach outperforms GenEdit on the BIRD dev set with an EX of 67.21%. The other approach has simpler operators then the one we describe here. It uses fine-tuned `GPT-4o` and maximizes the schema contextual information at generation. Most surprisingly however, we use GenEdit within our enterprise deployments as the other approach can't handle the same query complexity even though it outperforms GenEdit on BIRD. Note that newer benchmarks such as Beaver [3] and Spider 2.0 [12] introduce more complexity to overcome such a mismatch between real deployments and public benchmarks.

4 CONTINUOUS IMPROVEMENT

In our early deployments, SMEs would give feedback directly to engineers who would then update the knowledge set manually. This is an unsustainable approach that cannot scale across deployments nor as use cases are added. In this section, we overview our edit recommendations module, which primarily aims to shorten the feedback integration loop. GenEdit acts as a crowdsourcing system [4] and as an interactive machine learning system [1].

Recommended edits to the knowledge set are generated using the provided feedback in a fully-automated manner, following feedback operators #1 to #4 in Fig. 1. These generated edits are reviewed by SMEs and staged within a UI. SMEs can regenerate and iterate on their feedback until satisfied. They can then submit their edits which are tested for regressions and upon approval published and merged. We explain the operators associated with generating recommended edits to the knowledge set and overview the module's UI.

4.1 Recommending Edits

As shown in Fig. 1, GenEdit takes a generated query and associated user feedback as input and generates edit recommendations that then lead to a query regeneration as output.

Recommending edits relies on 4 operators:

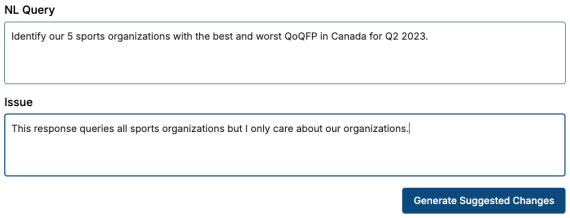
i. **Generate Targets**: operator #1 determines which of the instructions and examples retrieved for generation are relevant to the user feedback. It also generates a brief explanation of why the feedback is indeed relevant.
ii. **Expand Feedback**: operator #2 expands on the prior explanation for why the feedback is relevant to the chosen elements (instructions and examples).
iii. **Planning of Edits**: operator #3 takes the expanded feedback and creates a step-by-step CoT plan for what changes are required and how to apply them.
iv. **Generate Edits**: operator #4 generates a full revised output in the relevant form (*e.g.*, decomposed examples) based on the plan.

4.2 Interactive Collaboration

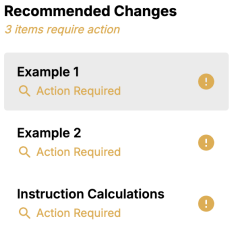
GenEdit includes a web-based UI designed to create, manage, and process feedback automatically, leading to edits to examples and instructions in the knowledge set. Two key components of the interface are the feedback solver and the knowledge set library.

4.2.1 Feedback Solver. Once the user submits a natural language query, the interface generates the equivalent SQL with a natural language summary. The user can then run the query and inspect the tabular output as well as a set of automatically generated plots. If there are any issues with the query, the user can provide natural language feedback as shown in Fig. 3a for $Q_{fin-perf}$ where the feedback is that *'This response queries all sports organizations but I only care about our organizations.'*. Once the feedback is submitted, recommended edits (obtained as described above in §4.1) are shown in a side panel. In our example, the system recommends three edits, two for examples and one for an instruction as shown in Fig. 3c. The user can then review each of the edits, make further changes if necessary, and finally stage some or all of them. Staging in this case means accepting the edit and taking it to an environment that mimics the deployed system for testing. The user staged edits are highlighted in the UI as shown in Fig. 3e. Once ready, the user can regenerate the query and continue iterating on their feedback and staging edits. The full view containing the various elements after regeneration is shown in Fig. 3e. Once done, the edits are submitted and then go through regression testing. If they pass, they are pending for approval.

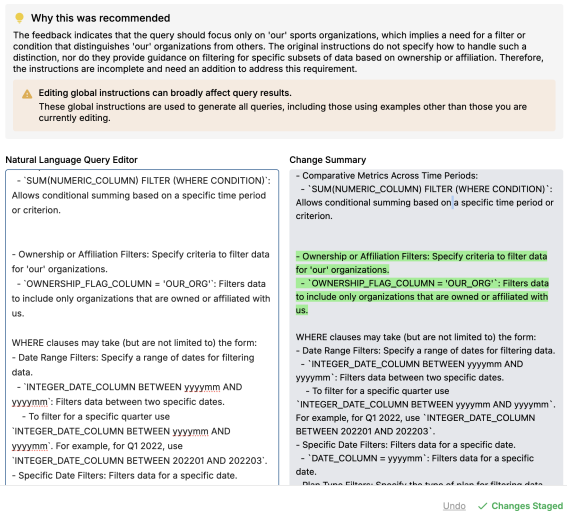
[2] evaluation numbers from Aug. 2024



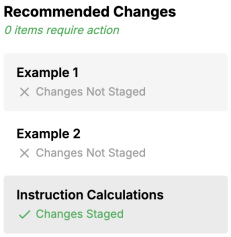
(a) Interface for Submitting Feedback.



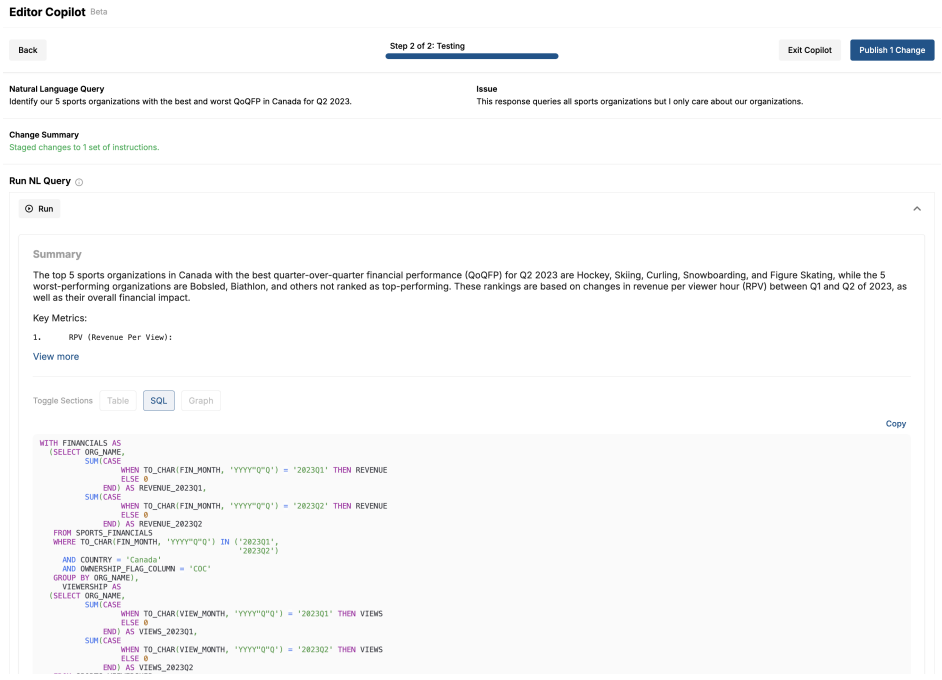
(b) Recommended Edits.



(c) Interface for Reviewing Recommended Edits.



(d) Staged Edits.



(e) Regeneration Interface including the original natural language query, the feedback, the regenerated query, and summary of the staged edits in the regeneration.

Figure 3: Example of GenEdit UI interfaces for the .

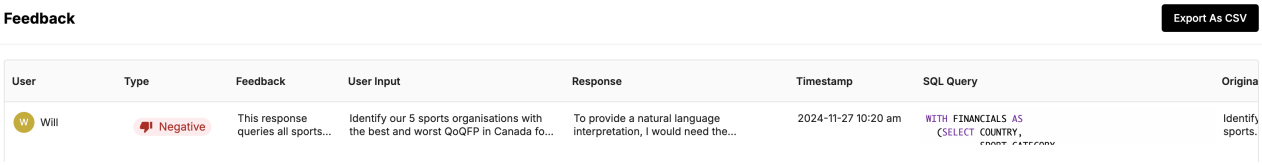


Figure 4: Example of a feedback displayed within the knowledge set library of GenEdit.

4.2.2 Knowledge Set Library. This interface allows an expert user to interact with the knowledge set. The library showcases the components of the knowledge set and their provenance, offering full visibility for reversion, comparison, and systematic learning from prior feedback. Fig. 4 shows an example of how old feedback is shown in a list ordered by timestamp. Experts may also directly edit the knowledge set within the library outside of the context of a query and its feedback.

4.2.3 Evaluation. We evaluate GenEdit's edits recommendation module in production based on two metrics: i) *how many suggested edits are automatically accepted as is*; ii) *how many edits are accepted after re-using the solver or doing manual knowledge set edits?*

5 RELATED WORK

Text-to-SQL. Early approaches struggled to incorporate external knowledge in a flexible manner due to contextual limitations [33]. Relevant schema elements were identified with semantic parsers [30], queries were generated through sketches [29], and self-correction was conducted through execution-guided decoding [25]. Conversely, modern LLM-based approaches benefit from their ability to seamlessly integrate relevant knowledge at the query generation stage, leading to improved accuracy and reliability across various stages, *e.g.*, schema linking, query generation, and self-correction [7, 11].

Crowdsourcing Systems. GenEdit is a crowdsourcing system (CS) from a problem-solving standpoint as it enlists a crowd of humans to help solve a problem defined by the system owners (in this case improving Text-to-SQL generation) [4]. Within GenEdit, users implicitly collaborate as we piggyback on an existing analytics engine. We attract feedback through *instant gratification*, by immediately showing a user their regenerated query to show that their contribution makes a difference. CSes were very popular around a decade ago (~2010s) with many systems proposed [6, 17]. These are similar in that the systems have operations requiring crowds. They also differ in that our approach is meant for improving system primitives instead of supporting an actual computation. Recently, a proposal for *prompt engineering* via declarative crowdsourcing has been proposed [18]. The proposal focuses on providing feedback on LLM-based data transformations to improve cost. GenEdit differs in that it is a generation pipeline; however, our work can be extended in similar ways by getting feedback on latency or specifying a dollar cost and parametrizing GenEdit pipelines differently.

Interactive Machine Learning Systems. GenEdit follows the setup of interactive ML systems and has similar goals in removing the ML experts from the feedback loop [1]. While our UI design is driven by feedback from users of the system, there has been a lot of prior work exploring the best approach to ML system interactivity. Such work highlights the impact of interactions on user behaviour and vice-versa and aims to design interactions based on understood behaviour [2]. We are only aware of one interactive system for Text-to-SQL that allows users to directly edit a step-by-step explanation of the query generation to fix errors [23]. Unlike GenEdit, this approach fixes a single instance of a SQL query after generation without continuous improvement, *i.e.*, there is no deep integration of the feedback. As such, the same query can fail again later. GenEditedits the knowledge set and hence future prompts to ensure lasting improvements.

6 DEMO

We will demonstrate the effectiveness of GenEdit in generating, reviewing, and updating examples and instructions by running queries against a benchmark dataset of choice. First, we will take natural language queries and generate equivalent SQL. Second, we will identify issues with the generated output and provide feedback through the Feedback Solver interface (shown in Fig. 3). We will then update the feedback till the query is regenerated to satisfaction. Third, we submit the feedback, showcase an accuracy to check for regressions on few selected golden queries and move to human review where we can check the edits through the knowledge set interface (shown in Fig. 4). This is the interface that users can use to learn from past feedback or move between various knowledge set checkpoints and revert changes. Finally, we close the loop by accepting these changes and validating that the previously incorrect queries now return correct results.

7 CONCLUSION

We introduced GenEdit, a collaborative Text-to-SQL generation system designed to address enterprise-specific challenges. GenEdit can generate very complex SQL queries by leveraging a company-specific knowledge set and a pipeline of compounding operators. It improves retrieval of query specific knowledge and minimizes the need for LLM reasoning by decomposing text-to-SQL generation using a novel chain-of-thought plan. Finally, GenEdit's feedback interface enables continuous improvement. It allows users to iterate on their feedback regenerating the query until satisfaction. The submitted feedback updates the knowledge set which in turn is tested for regression and then merged. This turn improves future generations as it improves SQL generation prompts.

8 ACKNOWLEDGEMENTS

We would like to thank Harshini Jayaram, Will Morley, and the broader team at Distyl AI productionizing GenEdit for numerous useful discussions and help with technical issues.

REFERENCES

[1] S. Amershi, M. Cakmak, W. B. Knox, and T. Kulesza. Power to the people: The role of humans in interactive machine learning. *AI Mag.*, 2014.

[2] S. Amershi, M. Cakmak, W. B. Knox, T. Kulesza, and T. Lau. Iui workshop on interactive machine learning. *IUI*, 2013.

[3] P. B. Chen, F. Wenz, Y. Zhang, M. Kayali, N. Tatbul, M. J. Cafarella, Ç. Demiralp, and M. Stonebraker. BEAVER: an enterprise benchmark for text-to-sql. *CoRR*, abs/2409.02038, 2024.

[4] A. Doan, R. Ramakrishnan, and A. Y. Halevy. Crowdsourcing systems on the world-wide web. *CACM*, 2011.

[5] X. Dong, C. Zhang, Y. Ge, Y. Mao, Y. Gao, lu Chen, J. Lin, and D. Lou. C3: Zero-shot text-to-sql with chatgpt. *CoRR*, abs/2307.07306, 2023.

[6] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. *SIGMOD*, 2011.

[7] D. Gao, H. Wang, Y. Li, X. Sun, Y. Qian, B. Ding, and J. Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *CoRR*, abs/2308.15363, 2023.

[8] D. Gao, H. Wang, Y. Li, X. Sun, Y. Qian, B. Ding, and J. Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *PVLDB*, 2024.

[9] Z. Hong, Z. Yuan, Q. Zhang, H. Chen, J. Dong, F. Huang, and X. Huang. Next-generation database interfaces: A survey of llm-based text-to-sql. *CoRR*, abs/2406.08426, 2024.

[10] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa. Large language models are zero-shot reasoners. *CoRR*, abs/2205.11916, 2023.

[11] D. Lee, C. Park, J. Kim, and H. Park. Mcs-sql: Leveraging multiple prompts and multiple-choice selection for text-to-sql generation. *CoRR*, abs/2405.07467, 2024.

[12] F. Lei, J. Chen, Y. Ye, R. Cao, D. Shin, H. Su, Z. Suo, H. Gao, W. Hu, P. Yin, V. Zhong, C. Xiong, R. Sun, Q. Liu, S. Wang, and T. Yu. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *CoRR*, abs/2411.07763, 2024.

[13] H. Li, J. Zhang, H. Liu, J. Fan, X. Zhang, J. Zhu, R. Wei, H. Pan, C. Li, and H. Chen. Codes: Towards building open-source language models for text-to-sql. *SIGMOD*, 2024.

[14] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Cao, R. Geng, N. Huo, X. Zhou, C. Ma, G. Li, K. C. C. Chang, F. Huang, R. Cheng, and Y. Li. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *CoRR*, abs/2305.03111, 2023.

[15] K. Maamari, F. Abubaker, D. Jaroslawicz, and A. Mhedhbi. The death of schema linking? text-to-sql in the age of well-reasoned language models. *CoRR*, abs/2408.07702, 2024.

[16] K. Maamari and A. Mhedhbi. End-to-end text-to-sql generation within an analytics insight engine. *CoRR*, abs/2406.12104, 2024.

[17] A. G. Parameswaran and N. Polyzotis. Answering queries using humans, algorithms and databases. *CIDR*, 2011.

[18] A. G. Parameswaran, S. Shankar, P. Asawa, N. Jain, and Y. Wang. Revisiting prompt engineering via declarative crowdsourcing. *CIDR*, 2024.

[19] M. Pourreza and D. Rafiei. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *CoRR*, abs/2304.11015, 2023.

[20] G. Qu, J. Li, B. Li, B. Qin, N. Huo, C. Ma, and R. Cheng. Before generation, align it! a novel and effective strategy for mitigating hallucinations in text-to-sql generation. *CoRR*, abs/2405.15307, 2024.

[21] T. Ren, Y. Fan, Z. He, R. Huang, J. Dai, C. Huang, Y. Jing, K. Zhang, Y. Yang, and X. S. Wang. PURPLE: making an llm a better SQL writer. *ICDE*, 2024.

[22] S. Talaei, M. Pourreza, Y.-C. Chang, A. Mirhoseini, and A. Saberi. Chess: Contextual harnessing for efficient sql synthesis. *CoRR*, abs/2405.16755, 2024.

[23] Y. Tian, Z. Zhang, Z. Ning, T. J.-J. Li, J. K. Kummerfeld, and T. Zhang. Interactive text-to-sql generation via editable step-by-step explanations. *EMNLP*, 2023.

[24] B. Wang, C. Ren, J. Yang, X. Liang, J. Bai, L. Chai, Z. Yan, Q.-W. Zhang, D. Yin, X. Sun, and Z. Li. Mac-sql: A multi-agent collaborative framework for text-to-sql. *CoRR*, abs/2312.11242, 2024.

[25] C. Wang, K. Tatwawadi, M. Brockschmidt, P.-S. Huang, Y. Mao, O. Polozov, and R. Singh. Robust text-to-sql generation with execution-guided decoding, 2018.

[26] L. Wang, W. Xu, Y. Lan, Z. Hu, Y. Lan, R. K. Lee, and E. Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *CoRR*, abs/2305.04091, 2023.

[27] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. H. Chi, Q. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models. *CoRR*, abs/2201.11903, 2023.

[28] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS*, 2022.

[29] X. Xu, C. Liu, and D. Song. Sqlnet: Generating structured queries from natural language without reinforcement learning. *CoRR*, abs/1711.04436, 2017.

[30] T. Yu, M. Yasunaga, K. Yang, R. Zhang, D. Wang, Z. Li, and D. R. Radev. Syntaxsqlnet: Syntax tree networks for complex and cross-domaintext-to-sql task. *CoRR*, abs/1810.05237, 2018.

[31] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. R. Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *CoRR*, abs/1809.08887, 2019.

[32] H. Zhang, R. Cao, L. Chen, H. Xu, and K. Yu. ACT-SQL: in-context learning for text-to-sql with automatically-generated chain-of-thought. *CoRR*, abs/2310.17342, 2023.

[33] V. Zhong, C. Xiong, and R. Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017.

A OUTPUT - SQL GENERATED BY LANGUAGE MODEL

```
WITH
FINANCIALS AS (
   SELECT ORG_NAME,
          SUM(CASE WHEN TO_CHAR(FIN_MONTH, 'YYYY"Q"Q') = '2023Q1' THEN REVENUE ELSE 0 END) AS REVENUE_2023Q1,
          SUM(CASE WHEN TO_CHAR(FIN_MONTH, 'YYYY"Q"Q') = '2023Q2' THEN REVENUE ELSE 0 END) AS REVENUE_2023Q2
     FROM SPORTS_FINANCIALS
    WHERE TO_CHAR(FIN_MONTH, 'YYYY"Q"Q') IN ('2023Q1', '2023Q2')
      AND COUNTRY = 'Canada'
      AND OWNERSHIP_FLAG_COLUMN = 'COC'
    GROUP BY ORG_NAME
),
VIEWERSHIP AS (
   SELECT ORG_NAME,
          SUM(CASE WHEN TO_CHAR(VIEW_MONTH, 'YYYY"Q"Q') = '2023Q1' THEN VIEWS ELSE 0 END) AS VIEWS_2023Q1,
          SUM(CASE WHEN TO_CHAR(VIEW_MONTH, 'YYYY"Q"Q') = '2023Q2' THEN VIEWS ELSE 0 END) AS VIEWS_2023Q2
     FROM SPORTS_VIEWERSHIP
    WHERE TO_CHAR(VIEW_MONTH, 'YYYY"Q"Q') IN ('2023Q1', '2023Q2')
      AND COUNTRY = 'Canada'
      AND OWNERSHIP_FLAG_COLUMN = 'COC'
    GROUP BY ORG_NAME
),
CHANGE_IN_REVENUE AS (
  SELECT
    f.ORG_NAME,
    CAST(f.REVENUE_2023Q2 AS FLOAT) / NULLIF(v.VIEWS_2023Q2, 0) AS RPV,
    CAST(f.REVENUE_2023Q1 AS FLOAT) / NULLIF(v.VIEWS_2023Q1, 0) AS PRIOR_QTR_RPV,
    -1 * (
      (CAST(f.REVENUE_2023Q2 AS FLOAT) / NULLIF(v.VIEWS_2023Q2, 0)) -
      (CAST(f.REVENUE_2023Q1 AS FLOAT) / NULLIF(v.VIEWS_2023Q1, 0))
    ) AS RPV_CHANGE,
    ((CAST(f.REVENUE_2023Q2 AS FLOAT) / NULLIF(v.VIEWS_2023Q2, 0)) -
     (CAST(f.REVENUE_2023Q1 AS FLOAT) / NULLIF(v.VIEWS_2023Q1, 0)))
    ) * NULLIF(v.VIEWS_2023Q2, 0) as IMPACT,
    ROW_NUMBER() OVER (PARTITION BY f.COUNTRY ORDER BY (-1 * (
      (CAST(f.REVENUE_2023Q2 AS FLOAT) / NULLIF(v.VIEWS_2023Q2, 0)) -
      (CAST(f.REVENUE_2023Q1 AS FLOAT) / NULLIF(v.VIEWS_2023Q1, 0)))
    ) DESC) AS SPORT_RANK,
    ROW_NUMBER() OVER (PARTITION BY f.COUNTRY ORDER BY (-1 * (
      (CAST(f.REVENUE_2023Q2 AS FLOAT) / NULLIF(v.VIEWS_2023Q2, 0)) -
      (CAST(f.REVENUE_2023Q1 AS FLOAT) / NULLIF(v.VIEWS_2023Q1, 0)))
    ) ASC) AS WORST_SPORT_RANK
  FROM FINANCIALS f
   JOIN VIEWERSHIP v ON f.ORG_NAME = v.ORG_NAME
)

SELECT
  SPORT_RANK, ORG_NAME, RPV, PRIOR_QTR_RPV, RPV_CHANGE, IMPACT
FROM
  CHANGE_IN_REVENUE
WHERE
  SPORT_RANK <= 5 OR WORST_SPORT_RANK <= 5
ORDER BY
  SPORT_RANK;
```