Continual Learning of Domain Knowledge from Human Feedback in Text-to-SQL

Thomas Cook[*] **Kelly Patel**, **Sivapriya Vellaichamy**,
Saba Rahimi, **Zhen Zeng**, **Sumitra Ganesh**
JPMorganChase
`thomas.cook, kelly.patel, sivapriya.vellaichamy,`
`saba.rahimi, zhen.zeng, sumitra.ganesh@jpmchase.com`

Abstract

Foundation models (FMs) can generate SQL queries from natural language questions but struggle with database-specific schemas and tacit domain knowledge. We introduce a framework for continual learning from human feedback in text-to-SQL, where a learning agent receives natural language feedback to refine queries and distills the revealed knowledge for reuse on future tasks. This distilled knowledge is stored in a structured memory, enabling the agent to improve execution accuracy over time. We design and evaluate multiple variations of a learning agent architecture that vary in how they capture and retrieve past experiences. Experiments on the BIRD benchmark Dev set show that memory-augmented agents, particularly the Procedural Agent, achieve significant accuracy gains and error reduction by leveraging human-in-the-loop feedback. Our results highlight the importance of transforming tacit human expertise into reusable knowledge, paving the way for more adaptive, domain-aware text-to-SQL systems that continually learn from a human-in-the-loop.

1 Introduction

AI Agents, with Foundation Models (FMs) as backbones, have shown advanced capabilities performing complex tasks, but still rely on humans to supply domain-specific knowledge [1, 2]. In many applications, this information may not be readily available in formal documentation or procedures, but rather exists as tacit knowledge, disseminated through informal channels such as personal experience and word-of-mouth communication. This unstructured knowledge is often difficult to anticipate and integrate, posing a significant challenge for deploying systems in practice.

This challenge is particularly acute in text-to-SQL tasks. While FMs excel at generating syntactically correct SQL queries [3, 4, 5, 6], they often lack understanding of database-specific schema and semantics. Complex databases with numerous tables, relationships, and constraints require domain expertise rarely captured during pretraining or prompt tuning, creating a gap between general query-generation capabilities and practical requirements.

We propose a human-in-the-loop framework for continual learning that enables iterative refinement of SQL queries. By engaging human experts, our agent bridges the domain knowledge gap, distills tacit knowledge at various granularities, and maintains a dynamic memory that grows with each interaction, allowing it to apply past experiences to future tasks. Our key contributions are:

[*]Corresponding author

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: AI That Keeps Up: Workshop on Continual and Compatible Foundation Model Updates (CCFM).

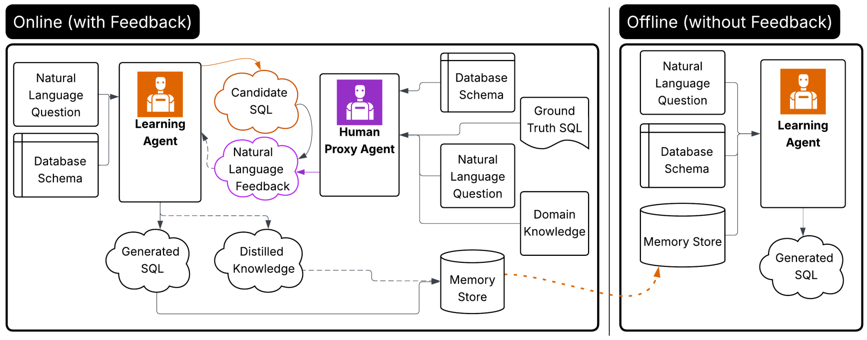

Figure 1: Overview of the two settings used for testing. In the online setting (left), the Learning Agent takes a natural language question and schema as input, generates a candidate SQL query, and iteratively refines it based on feedback from a human proxy agent. The final correct SQL and the distilled knowledge from this interaction are then stored in memory. In the offline phase (right), the Learning Agent generates SQL queries based on the natural language question, database schema, and the memory store (if available), without further feedback. The Learning Agent progressively improves execution accuracy by augmenting its memory.

- **Direct Integration of Tacit Knowledge**: We demonstrate how expert interactions can be used to capture subtle, domain-specific insights not captured through input-output example pairs, thereby enhancing the operational accuracy of text-to-SQL systems.
- **Continual Learning via Memory Augmentation**: By incorporating a structured memory mechanism that stores observed natural language queries, corresponding SQL outputs, and expert feedback distilled into procedural knowledge, our framework enables the agent to improve its performance in a cold start setting. We show that breaking down the feedback into procedural steps enables the agent to generalize concepts across similar tasks as well as provide more granular improvements to SQL generation which significantly boosts performance.
- **Comparative Agent Architectures**: We systematically construct and evaluate agent designs by incrementally enabling distinct memory types and reasoning. Starting from a baseline in-context learning setting, we incrementally develop and ablate components leading to our proposed *Procedural Agent* (PA), analyzing how each architectural enhancement affects SQL query generation accuracy and error reduction.

By addressing the intrinsic limitations of FMs in capturing domain-specific nuances, our work provides a pathway for developing AI agents that continually learn. In the sections that follow, we formalize the problem setting, detail the architecture of our proposed agents, and present empirical evaluations demonstrating the performance gains achieved through our continual learning approach.

2 Problem Setting and Technical Preliminaries

2.1 Problem Setting

We consider the setting where we observe a streaming sequence of natural language questions, $\{X_1, X_2, \dots\}$, with corresponding ground-truth SQL queries, $\{Y_1, Y_2, \dots\}$. We define a learning SQL Agent, $\mathcal{L}$, that takes as inputs the natural language question X_t and a historical memory M_t, and outputs a generated SQL query, $\hat{Y}_t$; $\mathcal{L} : \{\mathcal{X}\} \times \{\mathcal{M}\} \mapsto \{\mathcal{Y}\}$. Feedback, F_t, is also provided on intermediate generated SQL queries, meaning that the historical memory at time T depends only on $\{X_t\}_{t=1}^{T-1}$, $\{Y_t\}_{t=1}^{T-1}$, and $\{F_t\}_{t=1}^{T-1}$.

We evaluate the agent using *execution accuracy*, which asses whether the execution result of the generated SQL query is identical to that of the ground-truth[2]. We denote the accuracy with a binary

[2]The generated query can differ syntactically, but produce the same output.

variable Z_t. To assess an agent's ability to continually learn, we measure how its performance on a holdout set, $\{X'_i, Y'_i\}_{i=1}^{k}$, improves as it accumulates more experience. Over time, as the agent processes more queries and builds its accumulated experience, we monitor the proportion of queries for which the execution output matches that of the ground truth, $\bar{Z}_T = k^{-1} \sum_{i=1}^{k} Z'_i$. Improvement in this performance metric indicates that the agent is learning effectively.

2.2 Related Work

Context Augmentation for Continual Learning Large FMs can adapt to new tasks via few-shot conditioning [1], and continual learning has been explored by storing trajectories [7], reusable workflows [8], or past experiences in memory buffers [9]. Example enrichment further enables FMs to learn from interactions [10]. However, these approaches do not explicitly extract tacit domain knowledge, which is critical in text-to-SQL tasks requiring database familiarity, SQL heuristics, and domain expertise. We focus on richer representations of past experiences that capture this layered knowledge and design an agent that exploits this representation.

Text-to-SQL FMs achieve state-of-the-art results on BIRD [3] and SPIDER [4] leaderboards, often without fine-tuning [5, 6]. Current pipelines rely on query decomposition [11], schema linking [5], in-context examples [6], multiple candidate generations [6, 12, 13], and Chain-of-Though reasoning [6]. Incorporating feedback and domain adaptation have been explored via synthetic examples [14], reflective notebooks [15], and fine-tuning [16]. Our method extends these works by distilling human natural language feedback from trajectories that capture errors and revisions. The PA organizes this knowledge across multiple memory stores at different granularities, supporting continual learning specifically for chain-of-thought style procedural reasoning. Details are deferred to Appendix C.

3 Agent Design

3.1 Learning Agent

We focus on a learning agent which relies on an FM as its backbone and incorporates previous experiences through prompt augmentation. Specifically, we use GPT-4o as the FM backbone [17], and use text-embedding-ada-002 as the embedding model used for retrieval tasks [18].

We implement a baseline modeled after the GPT-4 setup reported on the BIRD leaderboard [3], which uses a templated prompt containing basic instructions, database schema, and in-context examples. While recent text-to-SQL systems incorporate increasingly complex architectures, our goal is to isolate the role of continual learning. Details are deferred to Appendix D.

3.1.1 Memory Design

The Learning Agent retrieves knowledge from a four-level memory hierarchy, which we ablate so that each higher level adds a new component. Level 0 stores NLQ–SQL pairs from previous successful runs (baseline in-context learning). Level 1 enhances these pairs with knowledge distilled from agent-human interactions. Details on the distillation process are provided in Section 3.1.2 and Appendix D. Level 2 adds an additional memory store of decomposed subtasks with affiliated SQL snippets. Level 3 adds an additional memory store containing database facts. Examples of the contents of each memory store are provided in Appendix E.

3.1.2 Knowledge Distillation

Upon successfully generating a SQL query, the Learning Agent distills knowledge from the interaction trajectory through a guided chain-of-thought reasoning process. Trajectories may be long, with multiple errors and iterative revisions, so the agent first explicitly identifies its mistakes, organizing the key failures to the end of the FM's context to prevent omitting any errors. Next, the agent isolates the information provided by the human that corrected these mistakes. This step captures tacit knowledge rather than generic SQL skills which the agent already possesses. Finally, the agent rephrases this knowledge to a broader context, generalizing the knowledge beyond the scope of the current question, to enable reuse on other questions. The resulting distilled knowledge is stored in

Level 1 memory, while the learning agent uses an additional tool to populate subtasks and database facts in Levels 2 and 3, respectively.

By abstracting domain and schema-specific insights from each interaction and storing them in memory, the agent can efficiently apply prior knowledge to new tasks, reduce redundancy, and improve its reasoning transparency. This not only enhances performance on future queries but also supports scalability and aligns with established human learning paradigms.

3.1.3 Procedural Reasoning

We evaluate two levels of procedural reasoning in our Learning Agent: no procedural reasoning (NP) and reasoning (P). In the NP condition, the agent operates under a templated prompt and is required to generate and refine SQL in a single step. The prompt includes the database schema and retrieved memories, with retrieval performed based on cosine similarity of the NLQ. In this setting, the retrieval query is fixed by the template, rather than being specified by the FM. This setup is akin to a simple FM baseline that uses in-context learning, and serves as a strong point of comparison.

In contrast, the P setting allows the agent to proceed step by step, with the specific sequence of actions (e.g., decomposition, planning, retrieval, or assembly) determined by the agent itself rather than fixed in advance. Here, the FM specifies retrieval queries and the granularity for memory retrieval, giving it substantially more flexibility in accessing relevant memories without the need to tune a similarity metric. This procedural reasoning also introduces greater variability in the agent's trajectory, since outcomes depend more heavily on the correctness of intermediate steps.

3.2 Human Proxy Agent (HPA)

To provide scalable, human-like feedback during SQL generation, we design a Human Proxy Agent (HPA) that emulates an expert database user. The HPA is built on an FM, which we use GPT-4o, and is provided with the NLQ, ground-truth SQL query, annotated expert "evidence", and database schema. The HPA is prompted to create a CoT solution to reference when providing feedback.

When the learning agent proposes a SQL query, the HPA executes it against the database and compares the output to the ground truth. If the query is correct, the HPA confirms completion, otherwise, it generates natural language feedback that identify errors in the logic of the SQL and prescribes corrections, simulating how a human expert would guide the agent. This feedback is sent to the learning agent, which refines its query accordingly. The process repeats until the correct SQL is generated, the conversation reaches 25 steps, or the maximum context length of the FM is exceeded.

4 Empirical Results

4.1 Evaluation

We evaluate the Learning Agent on the BIRD Dev set [3], which contains 11 databases spanning a variety of domains. We focus on BIRD rather than other benchmarks because it provides annotated expert evidence for each question, which is crucial for the HPA to generate realistic feedback containing tacit knowledge.

For each database, we randomly divide the questions into two equal splits—*train* and *test*—with T samples each. Our evaluation proceeds in three phases, as illustrated in Figure 1: initial performance (before learning), online performance (refinement through human feedback), and final performance (after learning).

Initial Performance. We run the offline setup with an empty memory store, establishing the performance based on database schema understanding and pretrained FM knowledge.

Online Performance. We run the online setup. The Learning Agent receives natural language feedback from the HPA until the generated SQL is correct or a maximum number of iterations is reached. Upon successful generation, the instance is saved to the agent memory store for future use, and if available, the Learning Agent distills the feedback as knowledge.

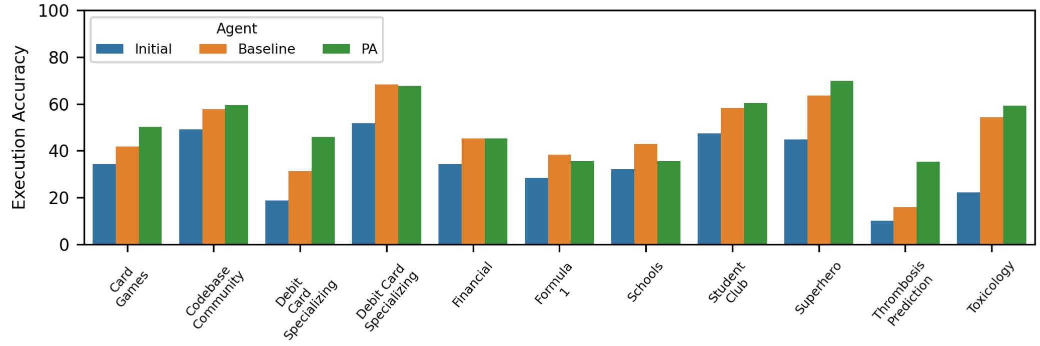

Figure 2: Execution accuracy by database for the Initial (NP-0, no memory), Baseline (NP-0), and Procedural Agent (PA, Agent label P-3).

Final Performance. We run the offline setup with the memory store built during the *Online Performance* phase. This final performance demonstrates the performance based on the database schema understanding, pretrained FM knowledge, and the retrieved knowledge from the memory store.

The difference in the *Final Performance*, $\bar{Z}_T$, and the *Initial Performance*, $\bar{Z}_0$ can be attributed to the usage of the memory store. We also evaluate $\bar{Z}_t$ for $t \in \{1, \ldots, T\}$ to quantify the impact of the quantity of instances used to construct the memory store.

4.2 Comparing Initial and Final Performance

We evaluate the Learning Agent in a setting where the online and final question splits differ: the initial and final phases are performed on the test split, and the online phase on the train split. This setup tests the agent's ability to generalize prior experiences to new tasks in the same domain, reflecting practical use. A similar analysis for the setting where all phases use the same questions is provided in Appendix B.1.

All agents improve from initial to final performance. Procedural reasoning alone increases accuracy by 2.4% while distilled knowledge (Level-1 memory) without reasoning can degrade performance. When combined with procedural reasoning, it yields substantial gains. The full Procedural Agent (P-3), combining procedural reasoning with a multi-level memory store, achieves the greatest absolute increase in accuracy, 18.9% versus 12.9% for the baseline, demonstrating the effectiveness of this approach. Figure 2 presents a per-database comparison of the full Procedural Agent (P-3) and baseline (NP-0), demonstrating the Procedural Agent's (PA) improved performance across most databases. Full ablation results are presented in Appendix B.2.

To understand these gains, we analyze common error types in Figure 3. Compared to the baseline, the PA reduces both syntax errors, such as incorrect DISTINCT usage or filters, and reasoning errors, including joins and table selection. Distilled knowledge helps the agent correct reasoning and schema comprehension errors, while procedural reasoning balances syntax and reasoning corrections. Minor increases in "Other" errors reflect additional column selections, which in practice may not impact overall query correctness.

4.3 Impact of Memory Size on Learning

Next, we examine how the number of instances in the memory store impacts performance. Figure 4 shows execution accuracy as a function of the number of instances used to construct the memory store across three databases. In more cases than not, the non-procedural baseline exhibits faster initial learning than the procedural agent, particularly within the first 10–20 instances. This likely reflects the fact that the procedural method requires more information to populate knowledge across three levels of granularity, rather than a single level. However, as the number of instances increases, the procedural method ultimately surpasses both non-procedural settings. Future work may investigate hybrid approaches that combine the rapid early gains of the non-procedural agent with the long-

Figure 3: Breakdown of error types for each agent configuration. Bars show the total counts of common error categories across different agents.

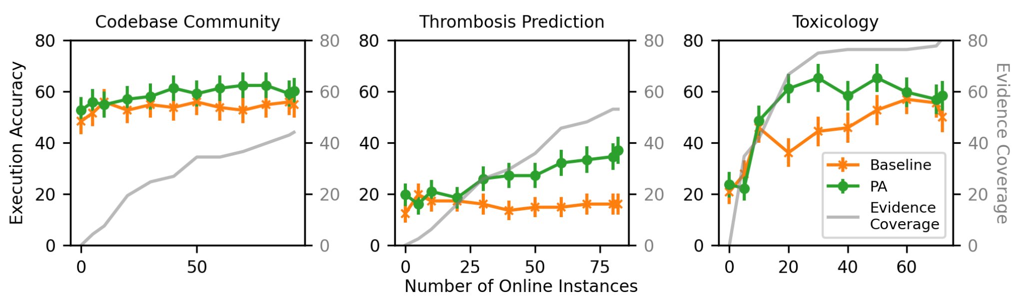

Figure 4: Execution accuracy as a function of the number of online instances used to construct the memory store. "Baseline" corresponds to the non-procedural agent (NP-0), and "PA" corresponds to the full Procedural Agent (P-3). Error bars indicate ± 1 standard deviation. "Evidence Coverage" denotes the proportion of test questions that have at least one corresponding question in the memory store whose annotated evidence field has cosine similarity ≥ 0.9 to that of the test question.

term advantages of procedural reasoning, enabling strong initial performance while progressively leveraging multi-level memory representations.

When observing the overlap in tacit knowledge between the instances used to populate the memory store and the test questions, measured as "evidence coverage" in Figure 4, the execution accuracy of the PA correlates with this metric, suggesting that the PA does effectively capture tacit knowledge. As evidence coverage plateaus or increases at a slower rate, the incremental benefit of additional tacit knowledge is outweighed by the increased complexity of retrieving from a larger memory store. This suggests that effective management of the memory size is an important direction for future work. Similar plots for all databases are provided in Appendix B.3.

5 Conclusions and Future Work

We present a framework that combines continual learning via agent memory with human-in-the-loop feedback to improve text-to-SQL performance. Our evaluation on the BIRD dev set shows that structured memory and chain-of-thought reasoning in the Procedural Agent substantially enhance query accuracy and reduce errors. Future work includes evaluating performance on more complex databases, handling noisy feedback, adapting to evolving schemas, and addressing scalability with large memory stores. This methodology can be extended to tasks requiring domain knowledge not captured during pretraining, such as tool usage, mathematical reasoning, or code generation.

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL `https://arxiv.org/abs/2005.14165`.

[2] Yingxuan Yang, Huacan Chai, Yuanyi Song, Siyuan Qi, Muning Wen, Ning Li, Junwei Liao, Haoyi Hu, Jianghao Lin, Gaowei Chang, Weiwen Liu, Ying Wen, Yong Yu, and Weinan Zhang. A survey of ai agent protocols, 2025. URL `https://arxiv.org/abs/2504.16736`.

[3] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, and Nan Huo. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 38, 2024.

[4] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018.

[5] Vladislav Shkapenyuk, Divesh Srivastava, Theodore Johnson, and Parisa Ghane. Automatic metadata extraction for text-to-sql, 2025. URL `https://arxiv.org/abs/2505.19988`.

[6] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. CHASE-SQL: Multi-path reasoning and preference optimized candidate selection in text-to-SQL. In *The Thirteenth International Conference on Learning Representations*, 2025.

[7] Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *The Twelfth International Conference on Learning Representations*, 2024.

[8] Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Forty-second International Conference on Machine Learning*, 2025.

[9] Yitao Liu, Chenglei Si, Karthik Narasimhan, and Shunyu Yao. Contextual experience replay for self-improvement of language agents, 2025. URL `https://arxiv.org/abs/2506.06698`.

[10] Aman Madaan, Niket Tandon, Peter Clark, and Yiming Yang. Memory-assisted prompt editing to improve GPT-3 after deployment. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, 2022.

[11] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, LinZheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, Zhoujun Li, and Steven Schockaert. MAC-SQL: A multi-agent collaborative framework for text-to-SQL. In *Proceedings of the 31st International Conference on Computational Linguistics*, 2025.

[12] Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, and Jingren Zhou. Xiyan-sql: A novel multi-generator framework for text-to-sql, 2025. URL `https://arxiv.org/abs/2507.04701`.

[13] Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. Mcs-sql: Leveraging multiple prompts and multiple-choice selection for text-to-sql generation, 2024. URL `https://arxiv.org/abs/2405.07467`.

[14] Yuan Tian, Daniel Lee, Fei Wu, Tung Mai, Kun Qian, Siddhartha Sahai, Tianyi Zhang, and Yunyao Li. Text-to-sql domain adaptation via human-llm collaborative data annotation. In *Proceedings of the 30th International Conference on Intelligent User Interfaces*, 2025.

[15] Zhibo Chu, Zichong Wang, and Qitao Qin. Leveraging prior experience: An expandable auxiliary knowledge base for text-to-sql, 2024. URL `https://arxiv.org/abs/2411.13244`.

[16] Yongrui Chen, Xinnan Guo, Tongtong Wu, Guilin Qi, Yang Li, and Yang Dong. Learn from yesterday: a semi-supervised continual learning method for supervision-limited text-to-sql task streams. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence*, 2023.

[17] OpenAI. Gpt-4o: Openai's new flagship model. `https://openai.com/index/gpt-4o`, 2024.

[18] OpenAI. text-embedding-ada-002. `https://platform.openai.com/docs/guides/embeddings`, 2022.

[19] Yuan Tian, Zheng Zhang, Zheng Ning, Toby Jia-Jun Li, Jonathan K. Kummerfeld, and Tianyi Zhang. Interactive text-to-SQL generation via editable step-by-step explanations. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[20] Rakesh R. Menon, Kun Qian, Liqun Chen, Ishika Joshi, Daniel Pandyan, Jordyn Harrison, Shashank Srivastava, and Yunyao Li. Fisql: Enhancing text-to-sql systems with rich interactive feedback. In *Proceedings of the 28th International Conference on Extending Database Technology (EDBT)*, 2025.

[21] Shivanshu Gupta, Matt Gardner, and Sameer Singh. Coverage-based example selection for in-context learning. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

Disclaimer

A Deferred Experimental Details

The BIRD dataset was used under a CC BY-SA 4.0 license.L. Access to gpt-4o version 2024-05-13 and text-embedding-ada-002 are used under an Enterprise license. All presented experiments were run within a total of 48 hours on a 16gb CPU.

B Additional Results

B.1 Ablation Study: Retention of Previously Seen Questions

Before considering generalization to new tasks, it is imperative that the Learning Agent can successfully complete a task that it has previously completed successfully. To this end, we evaluate the initial, online and the final performance on the train split for each database. Note that in the online setting, the HPA is providing direct feedback for refinement, and will continue to provide feedback until the Learning Agent generates the correct SQL query. During the offline phase, to determine final performance, the Learning Agent must effectively retrieve and reason over its stored memories, accurately identifying and leveraging relevant prior experiences even when presented alongside other, potentially distracting experiences.

Table 1 shows results for this setting. Both the NP and P agents performance similarly for the initial and online performances, demonstrating similar capabilities in the absence of any memories. The procedural reasoning enables a superior ability for the Learning Agent to effectively utilize memories on the same tasks. This is especially apparent in the agent trajectories, where P-type agents can explicitly reason about the retrieved memories and recognize that the current question matches that of the retrieved memory. We note, that a considerable performance degradation occurs for NP-1 vs NP-0, as compared to any of P-1, P-2, P-3, to P-0. This suggests a positive interaction effect between knowledge distillation and the procedural reasoning of the Learning Agent. The addition of the subtask and database fact memories stores further improves the final performance.

	Memory				Reasoning	Execution Accuracy				
Agent Label	NLQ-SQL	Distilled Knowledge	Subtasks	Database Facts	Procedural Reasoning	Initial Performance	Online Performance	Final Performance	Δ_i	Δ_o
NP-0	✓	✗	✗	✗	✗	35.1	92.0	77.4	42.3	-14.6
NP-1	✓	✓	✗	✗	✗	35.1	92.0	64.5	29.4	-27.0
P-0	✓	✗	✗	✗	✓	32.9	91.3	76.9	46.2	-13.4
P-1	✓	✓	✗	✗	✓	32.9	91.3	79.1	48.9	-10.7
P-2	✓	✓	✓	✗	✓	32.9	91.3	81.3	49.5	-10.1
P-3	✓	✓	✓	✓	✓	32.9	91.3	**81.7**	**48.8**	**-9.6**

Table 1: Execution accuracy when the same set of questions is used for each phase. δ_i is the difference between the final and initial performance, while δ_o denotes the difference between the final and online performance. Performances reported are averages of 3 runs.

B.2 Ablation Study: Generalization to New Questions

We next consider the setting where the question splits differ between the online and final performance phase. We perform the initial, and final phases on the test split, and the online phase on the train split. This setting evaluates the Learning Agent's ability to generalize previously acquired experiences to new tasks in the same domain. This scenario is more representative of practical use, where the agent faces new questions sequentially.

Agent Label	Initial Performance	Final Performance	Δ_i
NP-0	34.1	47.0	12.9
NP-1	34.1	43.7	9.6
P-0	32.4	49.4	17.0
P-1	32.4	50.8	18.4
P-2	32.4	49.5	17.1
P-3	32.4	**51.3**	**18.9**

Table 2: Execution accuracy by Agent Label when questions differ between the online and offline phases, averaged across 3 runs.

All agent settings show an increase in final performance relative to the initial performance. Procedural reasoning alone improves accuracy by 2.4%, underscoring its role in continual learning. In contrast, adding distilled knowledge without reasoning is detrimental to performance, as observed in the difference between NP-1 and NP-0. However, when combined with procedural reasoning, distilled knowledge yields a substantial performance gain, indicating a positive interaction effect. The inclusion of additional memory granularities further enhances performance, with the P-3 setting achieving the highest accuracy on both previously seen and new tasks. These results demonstrate that procedural reasoning, together with a multi-level memory store, enables the most effective continual learning.

B.3 Impact of Memory Size

We present a similar figure to Figure 4 for all databases in Figure 5.

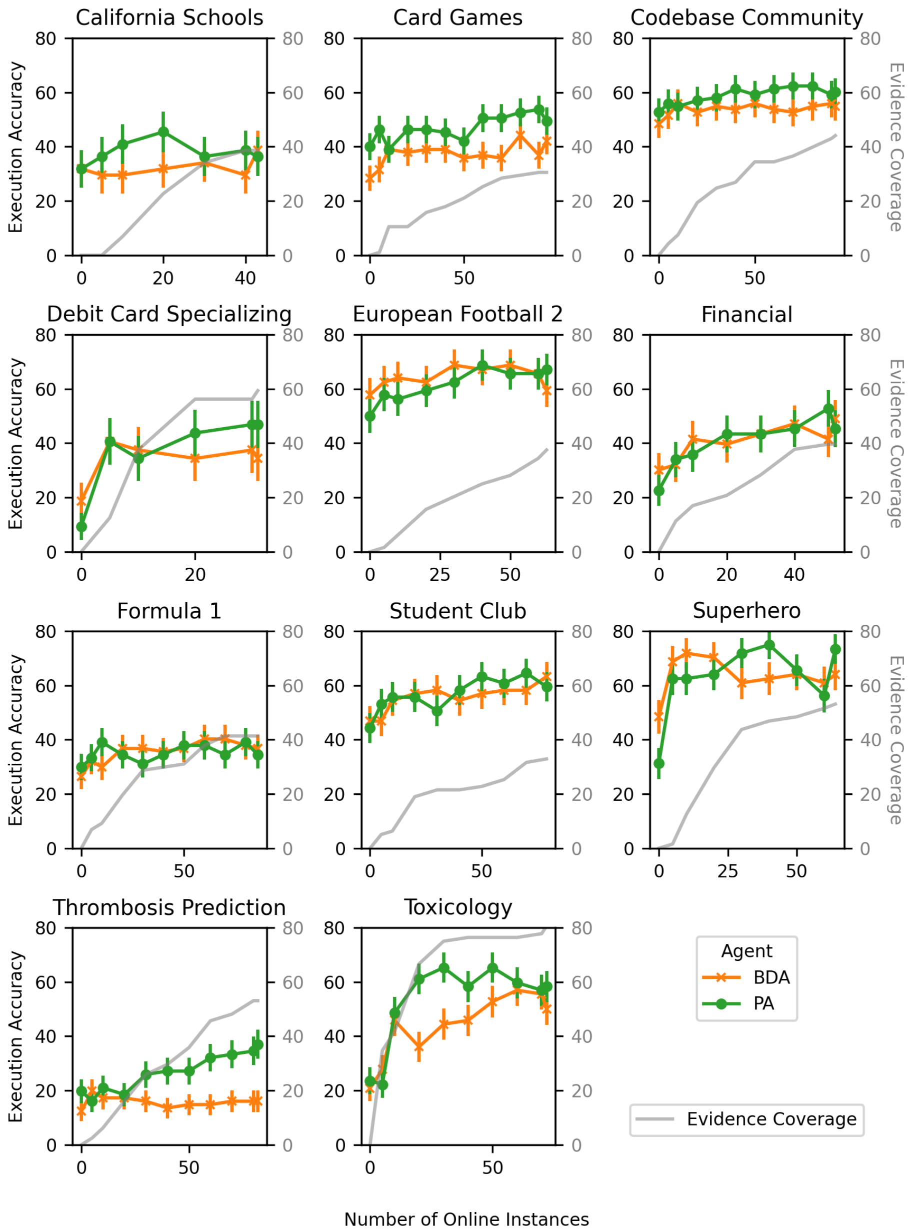

Figure 5: Execution accuracy as a function of the number of online instances used to construct the memory store for all databases in the Bird Dev benchmark. "Baseline" corresponds to the non-procedural agent (NP-0), and "PA" corresponds to the full Procedural Agent (P-3). Error bars indicate ± 1 standard deviation. "Evidence Coverage" denotes the proportion of test questions that have at least one corresponding question in the memory store whose annotated evidence field has cosine similarity ≥ 0.9 to that of the test question.

C Related Works

C.1 Context Augmentation for Continual Learning

Brown et al. [1] established that large-scale foundation models can adapt to new tasks simply by conditioning on a few input-output examples in their prompt (in-context learning), without parameter updates. Madaan et al. [10] demonstrate how this phenomena naturally lends itself to continual learning in our problem setting, by continuously enriching the prompt context over time with new, more relevant experiences effectively allow an agent to *learn* within the context provided. Recent agentic systems have formalized this idea. Zheng et al. [7] propose Synapse, where agents store entire state-action trajectories as exemplars and retrieve them in context to improve performance on future tasks in multi-step control settings, thus continually learning from past experiences. Similarly, Agent Workflow Memory [8] extracts reusable workflows from past executions and includes these workflows in the agent's context to guide future task solving. Liu et al. [9] introduce Contextual Experience Replay (CER), wherein agents accumulate past experiences in a memory buffer and replay them through the prompt to better adapt to new tasks without any model fine-tuning. Collectively, these works illustrate how in-context memory augmentation enables agents to continually improve their decision-making by grounding future prompts in their accumulated experiences, however, these methods do not explicitly considering the extraction of tacit knowledge from human feedback, which we believe is critical in the text-to-SQL domain.

C.2 Text-to-SQL

The domain of converting natural language questions to executable SQL queries has seen tremendous improvements through the use of foundation models, as seen by the leaderboards of the BIRD [3] and SPIDER [4]. At the time of writing, the top two disclosed models on the BIRD leaderboard rely on a foundation model without any finetuning [5, 6].

Human Feedback in Text-to-SQL Adapting to new databases or evolving schemas remains a significant challenge in the text-to-SQL domain. Tian et al. [14] proposed SQLSynth, an LLM-based, human-in-the-loop annotation method designed to generate a synthetic dataset for fine-tuning. This approach allows humans to manually update the correctness and alignment of generated SQL queries. However, their solution is presented as a static operation rather than a continuous learning process. In contrast, Tian et al. [19] consider human feedback in the form of editing step-by-step explanations, yet they do not explore the reuse of this feedback for future steps. Addressing this gap, Menon et al. [20] introduced FISQL, which refines SQL queries by classifying natural language feedback from users and making minimal prompt adjustments, thereby enhancing the adaptability and accuracy of the text-to-SQL process.

Continual Learning in Text-to-SQL The aforementioned works do not explicitly consider the task of continual learning. Our work most closely resembles the work of Chu et al. [15], who propose LPE-SQL, which builds auxiliary notebooks that contain previous NLQ-SQL pairs augmented with reasoning paths and self-reflection generated tips. Their framework does not consider the use of natural language feedback from an expert, but instead uses a self reflection mechanism, that is based on compiler hints and ground truth SQL query, rather than including a human-in-the-loop. A key difference, is that in our set up, the refinement process is carried out to completion (or until a maximum number of steps is reached), producing a trajectory that details key mistakes and the necessary revisions. This trajectory-based distillation preserves actionable insights that can be reused for subsequent queries, rather than abstract reflections that may be vague. Moreso, the Procedural Agent makes use of multiple memory stores, which contain distilled knowledge at different granularities. Chen et al. [16] consider continual learning in Text-to-SQL and propose a combination of semi-supervised learning and and memory-replay continual learning for an LSTM model. This methodology requires access to model weights and may be cumbersome for use on large foundation models.

D Additional Agent Details

D.1 Non-procedural agent with Level 0 Memory (NLQ-SQL pairs)

The agent is an LLM-based agent that has two tools that it can call. The `generate_sql` tool takes as input the natural language question (NLQ) from the user and uses a templated prompt to perform an LLM call. This template consists of at most three similar NLQ-SQL pairs (as defined by the cosine similarity between the vector embeddings of the current NLQ and the example NLQs) from a set of saved examples from previous, successful runs if they exist in the agent's historical memory, and the database schema, and asks the LLM to generate a SQL query. The agent is specifically instructed to generate SQL queries in adherence to the database schema.

If the agent receives feedback from the human expert, then it will call its `refine_sql` tool. This tool follows a similar template, but instead seeks to refine the generated SQL based on natural language feedback, rather than generate one from scratch. The human expert may continue to provide feedback until the agent generates the correct SQL. The memory store, M_t, consists of a vector database of NLQ-SQL pairs that the agent was successful in generating, enabling continual learning for future queries. The index is the NLQ, which enables retrieval by the semantic similarity of the example NLQs and the current NLQ.

Agent System Prompt, Offline

> You are an agent who generates SQL queries for a given natural language questions to query a given set of tables. Here are some tips for generating the SQL queries successfully.
> - Each SQL query needs to be generated only once.
> - Only share the final SQL query which is the SQL query for the last sub question with the human user.
> - Be sure to refine the SQL after generation to resolve any errors that you can fix yourself. You can use the refine sql tool and pass in a blank string for the feedback argument.
> - Do not change the original question provided to you when providing the question in tool calls.
> - Do not use the generate SQL tool if you have already generated a query. You must refine the sql at that point.
> - Use the human_return communcation spec when sending a message to the human. Please separate your message and the generated SQL query accordingly.

Agent System Prompt, Online

> You are an agent who generates SQL queries for a given natural language questions to query a given set of tables. Here are some tips for generating the SQL queries successfully.
>
> - Start by using generating a SQL query, using the generate_sql tool call. Do not return control to the human after this.
> - Then request feedback from the Expert agent so that you can to receive feedback. You must request feedback from the Expert agent at least once before returning the query to the human.
> - You must refine the SQL query based on the feedback provided by the expert agent. Use the refine_sql tool for this.
> - You may request feedback from the expert multiple times.
> - You MUST get confirmation from the Expert that the generated SQL query is correct before returning it to the human. Never return and unconfirmed SQL query!!!!
> - Do not change the original question provided to you when providing the question in tool calls.
> - Do not use the generate SQL tool if you have already generated a query. You must refine the sql at that point.
> - Do not ask the expert agent to generate the SQL query or ask the ground truth query.
> - Use the return_gen_sql Communication Spec when communicating with the Expert agent.
> - Use the human_return communcation spec when sending a message to the human. Please separate your message and the generated SQL query accordingly.

Generate SQL Prompt

You are a SQL query generation agent. Instructions for your task are as follows:

Generate a single SQL query which answers the following question that can be executed in a SQLITE3 database.

Examples

{examples}

Question:
{question}

Database schema and information:
{database_schema}

Task Output:

SQL Query:

Refine SQL Prompt

You are a SQL query generation agent with the capability to refine SQL queries given feedback. Instructions for your task are as follows:

Refine the given OLD SQL for the following QUESTION based on the FEEDBACK provided.
- If a SQLite error is provided, you must take that error into account and refine the SQL based on the error as well.
- Only produce a single SQL query.

Example

Question:
What is the eligible free rate of the 10th and 11th schools with the highest enrollment for students in grades 1 through 12?

Old SQL
SELECT CAST(‘Free Meal Count (K-12)‘ AS REAL) / ‘Enrollment (K-12)‘ FROM schools ORDER BY ‘Enrollment (K-12)‘ DESC LIMIT 9, 2

Feedback
You should select from the table ’frpm’, not schools

Refined SQL
SELECT CAST(‘Free Meal Count (K-12)‘ AS REAL) / ‘Enrollment (K-12)‘ FROM frpm ORDER BY ‘Enrollment (K-12)‘ DESC LIMIT 9, 2

Question
{question}
Database name
{database_name}

Database schema
{database_schema}
Examples
{examples}
Feedback
{feedback}
old SQL
{generated_sql}

Now please fixup old SQL and generate new SQL again.

Task Output:

D.2 Procedural Agent

The *Procedural Agent* (PA) differs significantly in that it does not use a fixed workflow for SQL generation and refinement. Instead, the agent is allowed to reason step by step, and make use of its `find_memory` and `save_memory` tools, interweaving retrieval, decomposition, and reasoning steps on the fly in a procedural manner. Since the agent has more autonomy in constructing a SQL query, we correspondingly expand the memory to three vector databases, containing similar questions, subtasks, and database facts, respectively. The question memory store corresponds a store of NLQ-SQL pairs that are enhanced with *distilled knowledge*. This enhancement consists of tacit knowledge the agent extracts at the conclusion of an interaction with the human and is based on the conversation with the human, the database schema, and the correct final generated SQL.The subtask memory store contains task description-SQL chunk pairs, which correspond to elements of a decomposed SQL query, inspired by the notion of example coverage explored by [21]. Lastly, the database-fact memory store contains description-hint pairs, ranging from schema linking, table contents, filter values and SQL operation hints. For constructing entries to the subtask and database-fact memory stores, the agent makes use of the `save_memory` tool, and must specify the index, the value, and type of memory being stored.

The PA starts by retrieving similar questions and reasoning about the utility of the examples retrieved (database schema and general SQL reasoning instructions are specified in the agent's system prompt). It then decomposes the SQL query into steps, and performs retrieval on a *similar subtask* memory store, grounding components in previous experiences. Database schema and general SQL reasoning instructions are specified in the agent's system prompt.During this step, the PA may also retrieve memories on a *database-fact* memory store in order to fill in knowledge gaps that may not be tied directly to a question or subtask. It then assembles the SQL query and verifies that it does not violate the database schema, as with the non-procedural Agent with Memory Level 0 (NP-0).

Agent System Prompt Offline

Your name is "Learning SQL Agent". You are an agent who generates SQL queries for a given natural language questions to query a given set of tables. Here are some tips for generating the SQL queries successfully.
You will have access to the natural language question, the database schema, and some knowledge stores. Your generated SQL must adhere to the database schema. The query should be executable on a SQLite database.

YOU MUST ADHERE TO THE FOLLOWING STEPS, IF YOU DEVIATE FROM THESE STEPS YOU WILL FAIL.
1. You should first find the most similar questions. Use the find memory_tool for this step. Evaluate how similar the memories are and elaborate on what information may be explicitly useful.
2. You should then decompose the SQL query down step by step. This should create a logical plan of how the SQL query will come together.
3. Based on this plan, you should use find_memory tool to find related sub_tasks and

database_facts. You must use this tool before generating the SQL. It has useful information you have accumulated over time.
4. After these steps, you should reason about generating SQL query. Think out loud, step by step. Feel free to go back to searching for more memories to work on the query. YOU SHOULD REALLY USE THE MEMORIES.
5. Generate the SQL query. This will be a candidate answer, contained in a thought, that you will reason about before sharing.
6. Although you are breaking the SQL query down step by step and creating queries for each step, you must avoid using subqueries in the final SQL. You should use joins when possible. Assemble these subtasks through joins. Rework the query if necessary.
6. Verify the generated SQL query by referencing memories. Elaborate on if any portion of the SQL query conflicts information from memories, and make the necessary updates.
7. Once you are satisfied with the SQL query, share the SQL query with the human, along with an explanation of the logic of the SQL query.

Some helpful tips:
- Do not generate and immediately share it with the human. You are REQUIRED pose a candidate answer to yourself, and check it first!
- You can reason with yourself by sending a message to yourself. Note that if the most recent message is from yourself, "Learning SQL Agent", you should build off of that message, not just repeat it.
- Use the human_return communication spec when sending a message to the human. Please separate your message and the generated SQL query accordingly.
- Be especially careful about the values used for filtering.
- Do not try to assemble a message and thought at the same time. Do these actions one at a time.

Here is the database schema:
{db_schema}

Agent System Prompt Online

Your name is "Learning SQL Agent". You are an agent who generates SQL queries for a given natural language questions to query a given set of tables. Here are some tips for generating the SQL queries successfully.

- You will have access to the natural language question, and the database schema. Your generated SQL must adhere to the database schema. The query should be executable on a SQLite database.
- You should first break the SQL query down step by step.
- You should reason about a plan to assemble the SQL query by completing these steps.
- Although you are breaking the SQL query down step by step and creating queries for each step, you must avoid using subqueries in the final SQL. You should use joins when possible. Assemble these subtasks through joins.
- Be especially careful about the values used for filtering.
- You can message a database expert agent to request feedback. You should get confirmation that the query is correct before returning it to the human. You MUST message the expert at least once.
- When the expert provides feedback, you should reflect on the feedback and plan a revision. With this plan, refine the SQL query to incorporate the feedback. You must request feedback after any refinement.
- Once the expert says the sql query is correct, only then can you share the SQL with the human. Share an explanation of the SQL query as well.
- You can reason with yourself by sending a message to yourself. Note that if the most recent message is from yourself, "Learning SQL Agent", you should build off of that message, not just repeat it.
- Use the human_return communication spec when sending a message to the human. Please separate your message and the generated SQL query accordingly.

- Do not save memories until the human instructs you to.

- Do not try to generate a message and thought at the same step. YOU CAN ONLY DO ONE OF THESE PER STEP!!!!

Here is the database schema:
{db_schema}

Distilling Knowledge Upon successful generation of the SQL query, one of the following messages is sent to the learning Agent. If the learning agent did not receive any feedback (it generated a correct SQL initially), the following message is sent:

Please summarize what you learned from generating this SQL query. Think about it out loud in many steps, and then give me a concise and specific summary.

Do these steps sequentially. The output of the previous step is used for the subsequent steps.

1. First send yourself a message which describes any domain specific knowledge that was needed for generating this SQL query. Explicitly highlight what reasoning steps that you took that were correct.
2. Second, send yourself a message which details what database knowledge was revealed. Link this is to the schema. Be very specific, and SQL heavy.
3. Third, look at the thoughts generated by steps 1 and 2. Rephrase the findings as facts that can be useful in the future.

The only response that I want back is the set of facts. Leave the "generated_sql" field empty in your response.

Otherwise, the following message is sent:

Please summarize what you learned from interacting with the Expert agent. Think about it out loud in many steps, and then give me a concise and specific summary.

Do these steps sequentially. The output of the previous step is used for the subsequent steps.

1. First send yourself a message which describes the mistakes you made. This should explicitly state what component of the SQL query was incorrect. You should also pinpoint where this error occurred in your reasoning.
2. Second, send yourself a message which details what database knowledge was revealed through feedback to correct this mistake. Link this is to the schema. Be very specific, and SQL heavy. This may contain information about the reasoning you had about the database.
3. Third, look at the thought generated by step 2. Rephrase the findings as facts. Provide some context as to when these facts would be useful in the future.

The only response that I want back is the set of facts. Leave the "generated_sql" field empty in your response.

The learning agent then responds with distilled knowledge. This knowledge is added to the NLQ-SQL pair and is saved to the similar_question memory store. Next, the following message is sent to the learning agent to save memories to the sub task and database fact memory stores:

Now save the distilled knowledge, one chunk at a time, for future use by using your save_memory tool. You can decompose the SQL query and save the chunks as sub tasks as useful reusable components in the future. You can also save database_facts, which contain useful information that may be necessary in the future in a more general set of tasks.

You can save up to 5 memories. Plan and decide which are most important before you start. Here are some helpful tips:
- Be sure to make the index of the memory useful. A useful index is one that will be easy to search for in the future. For example, dont just say "join table1 and table2". Say something like "join table1 and table2 by attribute", where attribute is the common column.
- When saving a similar subtask, the index should generally describe the subtask. Always make the passage contain a chunk of SQL. The index must contain information about the values filtered for in the SQL if you keep the values in the passage. Otherwise, you should replace the filters with placeholders, and the index should reflect the generality.
- Don't save basic SQL knowledge like "how to limit". This is information you already know. You could instead say "how to sort for __column name__" and then the passage can show that you order and use a desc and limit clause in a SQL query.

Some examples of what not to do:

'query_string': 'use of JOIN operations to connect tables',
'knowledge_string': 'JOIN operations are used to connect tables based on common columns to retrieve related data.',
'memory_type': 'database_fact'

Reason: This does not provide any useful information about the database and is just common SQL knowledge.

'query_string': 'filter by exact text values in SQL',
'knowledge_string': "When constructing SQL queries involving the 'frequency' column, it is essential to use the exact text values defined in the schema for accurate filtering. This ensures the query retrieves the intended results based on the issuance frequency.",
'memory_type': 'database_fact'

Reason: You should instead save "filter by issuance frequency" as the index, and the the passage should be showing the portion of the SQL query of how to do this. If available, you should specify what the filter value corresponds to in the index.

'query_string': "filter accounts by frequency 'POPLATEK TYDNE'",
'knowledge_string': "SELECT * FROM account WHERE frequency = 'POPLATEK TYDNE';",
'memory_type': 'similar_subtask'

Reason: The query string is not helpful since you would need to know to search for POPLATEK TYDNE in the first place. The tricky part is knowing what this value semantically means. Instead the query string should be "filter accounts for weekly issuance"

D.3 Memory Tools

The save_memory tool is used at the conclusion of an online iteration, after the PA has distilled the knowledge. Note that this tool is used for sub_task and database_fact memory types, as the similar_question memory type is hardcoded to prevent question rewording and to provide clarity between questions and subtasks.

Save Memory Tool

```
def save_memory(self, query_string:str, knowledge_string: str, memory_type :
    str) -> str:
'''
Tool for saving information from the current iteration for future in-context
    learning. When using this tool, do not use any aliases.
Args:
- query_string (str): the piece of text you will be used as an index in a
    vector db. This is NOT the sql query. For example, "how to calculate
    eligible free rate".
- knowledge_string (): The knowledge attached to the index in the vector
    database. This can be a SQL query, a snippet of a SQL query, or a
    database fact.
- memory_type (str): One of ['similar_subtask', 'database_fact'], Specifies
    what the type of information you are searching for.
'similar_subtask' memory type is helpful for understanding how to perform
    particular operations on this database. This information is critical when
     assembling a SQL query.
'database_fact' memory type is helpful for understand critical database
    information which you may have over looked in the past. This is
    incredibly useful for ensuring filters are correct.
'''
    if self.memory_counter >=5:
        return "You have exceeded the number of memories that you can save
    for this question. Do not save any more memories."
    memory_store = load_vector_db(memory_type)
    if query_string not in memory_store.index:
        memory_store.insert(query_string, knowledge_string)
        memory_store.save()
        self.memory_counter += 1
    return "Saved."
```

The find_memory tool is used to retrieve memories during offline testing.

Find Memory Tool

```
def find_memory(self, query_string:str, memory_type : str) -> str:
    '''
    Tool for retrieving information from previous experiences for in-context
    learning.

    Args:
    - query_string (str): the piece of text you want to search for. This is
    NOT the sql query. For example, "how to calculate eligible free rate".
    - memory_type (str): One of ['similar_question', 'similar_subtask', '
    database_fact'], Specifies what the type of information you are searching
     for.

    'similar_question' memory type is helpful for seeing natural language
    question-sql query pairs. This is helpful for planning. DO NOT CHANGE THE
     WORDING OF THE NATURAL LANGUAGE QUESTION FROM HOW THE HUMAN STATED IT.
    DO NOT PARAPHRASE.
    'similar_subtask' memory type is helpful for understanding how to perform
     particular operations on this database. This information is critical
    when assembling a SQL query.
    'database_fact' memory type is helpful for understand critical database
    information which you may have over looked in the past. This is
    incredibly useful for ensuring filters are correct.
    '''
    memory_store = memory_store.load(memory_type)

    return memory_store.retrieve(query_string, k = 3, max_distance = 0.28)
```

E Sample Trajectories

E.1 Non-procedural agent with Memory Level 0 - Offline Sample Trajectory

Human → SQL agent

Generate a SQL query for the following question. I have provided name of the database that is relevant to this query.
question: Among the accounts who have loan validity more than 12 months, list out the accounts that have the highest approved amount and have account opening date in 1993.
database name: financial

SQL agent Tool Call: generate_sql

```
{
    'question': 'Among the accounts who have loan validity more
    than 12 months,
list out the accounts that have the highest approved amount and
    have account
opening date in 1993.',
    'database_name': 'financial'
}
```

Database name: financial
../../datasets/bird/dataset/dev/dev/_databases/financial/database/_description/account.csv ***
../../datasets/bird/dataset/dev/dev/_databases/financial/database/_description/card.csv ***
../../datasets/bird/dataset/dev/dev/_databases/financial/database/_description/client.csv ***
../../datasets/bird/dataset/dev/dev/_databases/financial/database/_description/disp.csv ***
../../datasets/bird/dataset/dev/dev/_databases/financial/database/_description/district.csv ***
../../datasets/bird/dataset/dev/dev/_databases/financial/database/_description/loan.csv ***
../../datasets/bird/dataset/dev/dev/_databases/financial/database/_description/order.csv ***
../../datasets/bird/dataset/dev/dev/_databases/financial/database/_description/trans.csv ***
Using examples

Similar Example

Question: Among the accounts who have loan validity more than 24 months, list out the accounts that have the lowest approved amount and have account opening date before 1997.
SQL:

```
SELECT a.account_id
FROM loan l
JOIN account a ON l.account_id = a.account_id
WHERE l.duration > 24 AND STRFTIME('%Y', a.date) < '1997'
ORDER BY l.amount ASC
LIMIT 1
```

SQL agent Tool Response: generate_sql

SQL query generated! Generate SQL is

```
SELECT a.account_id
FROM loan l
JOIN account a ON l.account_id = a.account_id
WHERE l.duration > 12 AND STRFTIME('%Y', a.date) = '1993'
ORDER BY l.amount DESC
LIMIT 1;
```

SQL agent → Human

Here is the SQL query to find the accounts with the highest approved amount among those with loan validity more than 12 months and account opening date in 1993.

```
SELECT a.account_id
FROM loan l
JOIN account a ON l.account_id = a.account_id
WHERE l.duration > 12 AND STRFTIME('%Y', a.date) = '1993'
ORDER BY l.amount DESC
LIMIT 1;
```

Query Result

```
[(10451,)]
```

Output match!

E.2 Non-procedural agent with Memory Level 0 Online Sample Trajectory

Human → SQL agent

Generate a SQL query for the following question. I have provided name of the database that is relevant to this query.
question: In the branch where the second-highest number of crimes were committed in 1995 occurred, how many male clients are there?
database name: financial

SQL agent Tool Call: generate_sql

```
{
    'question': 'In the branch where the second-highest number
    of crimes were
committed in 1995 occurred, how many male clients are there?',
    'database_name': 'financial'
}
```

Database name: financial
../../datasets/bird/dataset/dev/dev/_databases/financial/database/_description/account.csv ***
../../datasets/bird/dataset/dev/dev/_databases/financial/database/_description/card.csv ***
../../datasets/bird/dataset/dev/dev/_databases/financial/database/_description/client.csv ***
../../datasets/bird/dataset/dev/dev/_databases/financial/database/_description/disp.csv ***
../../datasets/bird/dataset/dev/dev/_databases/financial/database/_description/district.csv ***
../../datasets/bird/dataset/dev/dev/_databases/financial/database/_description/loan.csv ***
../../datasets/bird/dataset/dev/dev/_databases/financial/database/_description/order.csv ***
../../datasets/bird/dataset/dev/dev/_databases/financial/database/_description/trans.csv ***
Not using examples

SQL agent Tool Response: generate_sql

SQL query generated! Generate SQL is

```
SELECT COUNT(c.client_id)
FROM client c
JOIN disp d ON c.client_id = d.client_id
JOIN account a ON d.account_id = a.account_id
JOIN district dis ON a.district_id = dis.district_id
WHERE c.gender = 'male'
AND dis.district_id = (
    SELECT district_id
    FROM district
    ORDER BY A15 DESC
    LIMIT 1 OFFSET 1
);
```

SQL agent → Human proxy agent

Could you please review the generated SQL query for correctness?

```
SELECT COUNT(c.client_id)
FROM client c
JOIN disp d ON c.client_id = d.client_id
JOIN account a ON d.account_id = a.account_id
JOIN district dis ON a.district_id = dis.district_id
WHERE c.gender = 'male'
AND dis.district_id = (
    SELECT district_id
    FROM district
    ORDER BY A15 DESC
    LIMIT 1 OFFSET 1
);
```

Human proxy agent Tool Call: evaluate_sql

```
{
    'generated_sql': "SELECT COUNT(c.client_id) \nFROM client c\
    nJOIN disp d ON
c.client_id = d.client_id\nJOIN account a ON d.account_id = a.
    account_id\nJOIN
district dis ON a.district_id = dis.district_id\nWHERE c.gender
    = 'male' \nAND
dis.district_id = (\n    SELECT district_id \n    FROM district
    \n    ORDER BY
A15 DESC \n    LIMIT 1 OFFSET 1\n);"
}
```

../../datasets/bird/dataset/dev/dev_databases\financial\financial.sqlite
Output match fail!

Human proxy agent Tool Response: evaluate_sql

```
{
    'test_sql': "SELECT COUNT(c.client_id) \nFROM client c\nJOIN
     disp d ON
c.client_id = d.client_id\nJOIN account a ON d.account_id = a.
    account_id\nJOIN
district dis ON a.district_id = dis.district_id\nWHERE c.gender
    = 'male' \nAND
dis.district_id = (\n    SELECT district_id \n    FROM district
    \n    ORDER BY
A15 DESC \n    LIMIT 1 OFFSET 1\n);",
    'gold_sql': "SELECT COUNT(T1.client_id) FROM client AS T1
    INNER JOIN
district AS T2 ON T1.district_id = T2.district_id WHERE T1.
    gender = 'M' AND
T2.A15 = (SELECT T3.A15 FROM district AS T3 ORDER BY T3.A15 DESC
     LIMIT 1, 1)",
    'outcome': 'output-match-fail'
}
```

Human proxy agent → Human proxy agent (Thought)

The generated SQL query does not match the expected output. I need to analyze the differences and provide feedback to the SQL agent on how to correct the query.

Human proxy agent → Human proxy agent (Thought)

The generated SQL query does not produce the correct output. The main issues are:
1. **Gender Filtering**: The query uses `c.gender = 'male'`, but the correct filter should be `c.gender = 'M'` to match the expected format in the database.
2. **District Identification**: The subquery correctly identifies the district with the second-highest number of crimes using `LIMIT 1 OFFSET 1`, which is equivalent to `LIMIT 1, 1`. However, the subquery should select `A15` instead of `district_id` to match the number of crimes.
3. **Table Joins**: The query unnecessarily joins the `disp` and `account` tables, which are not needed for this task. The `client` table can be directly joined with the `district` table using `district_id`.
The SQL agent should adjust the query to address these issues.

Human proxy agent → SQL agent

The generated SQL query has a few issues that need to be addressed:
1. **Gender Filtering**: The query uses `c.gender = 'male'`, but it should use `c.gender = 'M'` to match the expected format in the database.
2. **District Identification**: The subquery should select `A15` instead of `district_id` to correctly identify the district with the second-highest number of crimes.
3. **Table Joins**: The joins with the `disp` and `account` tables are unnecessary for this task. You can directly join the `client` table with the `district` table using `district_id`.
Please adjust the query accordingly to address these issues.

SQL agent Tool Call: refine_sql

```
{
    'question': 'In the branch where the second-highest number
    of crimes were
committed in 1995 occurred, how many male clients are there?',
    'generated_sql': "SELECT COUNT(c.client_id) \nFROM client c\
    nJOIN disp d ON
c.client_id = d.client_id\nJOIN account a ON d.account_id = a.
    account_id\nJOIN
district dis ON a.district_id = dis.district_id\nWHERE c.gender
    = 'male' \nAND
dis.district_id = (\n    SELECT district_id \n    FROM district
    \n    ORDER BY
A15 DESC \n    LIMIT 1 OFFSET 1\n);",
    'feedback': "The generated SQL query has a few issues that
    need to be
addressed:\n\n1. **Gender Filtering**: The query uses `c.gender
    = 'male'`, but
it should use `c.gender = 'M'` to match the expected format in
    the
database.\n\n2. **District Identification**: The subquery should
     select `A15`
instead of `district_id` to correctly identify the district with
     the
second-highest number of crimes.\n\n3. **Table Joins**: The
    joins with the
`disp` and `account` tables are unnecessary for this task. You
    can directly
join the `client` table with the `district` table using `
    district_id`.",
    'database_name': 'financial'
}
```

SQL agent Tool Response: refine_sql

SQL refined! Refined SQL is

```
SELECT COUNT(c.client_id)
FROM client c
JOIN district dis ON c.district_id = dis.district_id
WHERE c.gender = 'M'
AND dis.district_id = (
    SELECT district_id
    FROM district
    ORDER BY A15 DESC
    LIMIT 1 OFFSET 1
);
```

SQL agent → Human proxy agent

Could you please review the refined SQL query for correctness?

```
SELECT COUNT(c.client_id)
FROM client c
JOIN district dis ON c.district_id = dis.district_id
WHERE c.gender = 'M'
AND dis.district_id = (
    SELECT district_id
    FROM district
    ORDER BY A15 DESC
    LIMIT 1 OFFSET 1
);
```

Human proxy agent Tool Call: evaluate_sql

```
{
    'generated_sql': "SELECT COUNT(c.client_id) \nFROM client c\
   nJOIN district
dis ON c.district_id = dis.district_id\nWHERE c.gender = 'M' \
   nAND
dis.district_id = (\n    SELECT district_id \n    FROM district
   \n    ORDER BY
A15 DESC \n    LIMIT 1 OFFSET 1\n);\n"
}
```

../../datasets/bird/dataset/dev/dev_databases\financial\financial.sqlite
Output match!

Human proxy agent Tool Response: evaluate_sql


```
{
    'test_sql': "SELECT COUNT(c.client_id) \nFROM client c\nJOIN
     district dis
ON c.district_id = dis.district_id\nWHERE c.gender = 'M' \nAND
    dis.district_id
= (\n    SELECT district_id \n    FROM district \n    ORDER BY
    A15 DESC \n
LIMIT 1 OFFSET 1\n);\n",
    'gold_sql': "SELECT COUNT(T1.client_id) FROM client AS T1
    INNER JOIN
district AS T2 ON T1.district_id = T2.district_id WHERE T1.
    gender = 'M' AND
T2.A15 = (SELECT T3.A15 FROM district AS T3 ORDER BY T3.A15 DESC
     LIMIT 1, 1)",
    'outcome': 'output-match'
}
```


Human proxy agent → SQL agent

The refined SQL query is now correct and produces the expected output. Great job on making the necessary adjustments!

SQL agent → Human

The SQL query has been successfully generated and verified. Here is the final query for your request:

```
SELECT COUNT(c.client_id)
FROM client c
JOIN district dis ON c.district_id = dis.district_id
WHERE c.gender = 'M'
AND dis.district_id = (
    SELECT district_id
    FROM district
    ORDER BY A15 DESC
    LIMIT 1 OFFSET 1
);
```

Query Result

```
[(96,)]
```

Output match!

Knowledge Summary

In the context of determining the number of male clients in the branch where the second-highest number of crimes were committed in 1995, the following knowledge gaps were identified and addressed:
1. **Understanding of Crime Data**: The number of crimes committed in 1995 is stored in the `A15` column of the `district` table. This column should be used to determine the branch with the second-highest number of crimes.
2. **Branch Identification**: The `district_id` column is used to identify branches across different tables. This column is crucial for linking the `district` table with the `client` table to find the correct branch.
3. **Gender Filtering**: The `gender` column in the `client` table is used to filter for male clients. The value to filter for male clients is `'M'`.
4. **Joining Tables**: To find the number of male clients in a specific branch, it is necessary to join the `client` table with the `district` table using the `district_id` column.
5. **Handling of Rankings**: To find the second-highest number of crimes, it is necessary to order the districts by the number of crimes in descending order and then select the second entry.
By addressing these gaps, the correct SQL query can be constructed to accurately answer the user's question.

E.3 Procedural Agent with Memory Level 3 - Offline Sample Trajectory

Human → SQL agent

Generate a SQL query for the following question. I have provided name of the database that is relevant to this query.
question: Among the accounts who have loan validity more than 12 months, list out the accounts that have the highest approved amount and have account opening date in 1993.
database name: financial

SQL agent Tool Call: find_memory

```
{
    'query_string': 'accounts with loan validity more than 12
   months and
highest approved amount',
    'memory_type': 'similar_question'
}
```

SQL agent Tool Response: find_memory

Index: Among the accounts who have loan validity more than 24 months, list out the accounts that have the lowest approved amount and have account opening date before 1997.
SQL:

```
SELECT a.account_id
FROM loan l
JOIN account a ON l.account_id = a.account_id
WHERE l.duration > 24 AND STRFTIME('%Y', a.date) < '1997'
ORDER BY l.amount ASC
LIMIT 1
```

Knowledge: 1. **Ordering and Limiting Results**: When you need to find the minimum or maximum value in a dataset, use ORDER BY to sort the results and LIMIT to restrict the output to the desired number of records. This is a standard SQL practice for efficiently retrieving extreme values.
2. **Date Filtering with STRFTIME**: Use the STRFTIME function to extract specific components (like year, month, day) from date columns for accurate filtering. This is particularly useful when you need to filter records based on a specific time period.
3. **Schema Understanding**: Familiarize yourself with the columns in each table and their data types. This knowledge is crucial for constructing accurate SQL queries that meet specific requirements, such as filtering based on date or retrieving specific columns.
[Additional memory entries omitted for brevity]

SQL agent → SQL agent (Thought)

The similar question found involves filtering accounts based on loan duration and account opening date, but it focuses on the lowest approved amount and a different year. However, it provides useful insights on how to structure the query:
1. **Join the `loan` and `account` tables**: This is necessary to access both loan details and account opening dates. 2. **Filter by loan duration**: Use a condition to filter loans with a duration greater than 12 months. 3. **Filter by account opening date**: Use the `STRFTIME` function to extract the year from the account opening date and filter for the year 1993. 4. **Order by approved amount**: Use `ORDER BY` to sort the results by the approved amount in descending order to find the highest. 5. **Limit the results**: Use `LIMIT` to ensure only the account with the highest approved amount is returned.
Next, I will search for any relevant subtasks or database facts that could assist in constructing the SQL query.

SQL agent → SQL agent (Final Thought)

The SQL query aligns with the information from the memories:

- The join between `loan` and `account` tables is correctly implemented.
- The filter for loan duration greater than 12 months is correctly applied.
- The use of `STRFTIME` to filter accounts opened in 1993 is consistent with the memory.
- The ordering by approved amount in descending order and limiting the results to 1 is correctly implemented.

I am confident that this SQL query is accurate and ready to be shared with the human.

SQL agent → Human

Here is the SQL query to find the accounts with loan validity more than 12 months, the highest approved amount, and account opening date in 1993.

```
SELECT a.account_id
FROM loan l
JOIN account a ON l.account_id = a.account_id
WHERE l.duration > 12 AND STRFTIME('%Y', a.date) = '1993'
ORDER BY l.amount DESC
LIMIT 1;
```

Query Result

```
[(10451,)]
```

Output match!

E.4 Procedural Agent with Memory Level 3 - Online Sample Trajectory

Human → SQL agent

Generate a SQL query for the following question. I have provided name of the database that is relevant to this query.
question: In the branch where the second-highest number of crimes were committed in 1995 occurred, how many male clients are there?
database name: financial

SQL agent → SQL agent (Thought)

To solve this query, I need to follow these steps:
1. Identify the branch (district) with the second-highest number of crimes committed in 1995. This information is available in the 'district' table, specifically in the 'A15' column.
2. Once the district is identified, I need to find the number of male clients in that district. This requires joining the 'client' table with the 'district' table using the 'district_id'.
3. Filter the clients by gender to count only male clients.
4. Assemble the SQL query using these steps and ensure it adheres to the database schema.
Next, I will start by identifying the district with the second-highest number of crimes in 1995.

SQL agent → SQL agent (Thought)

To find the district with the second-highest number of crimes in 1995, I will use the 'district' table and order the districts by the 'A15' column in descending order. I will then select the second entry from this ordered list.
The SQL query for this step would look like this:

```
SELECT district_id FROM district ORDER BY A15 DESC LIMIT 1
    OFFSET 1;
```

This query will give me the district_id of the branch with the second-highest number of crimes in 1995.
Next, I will plan how to count the number of male clients in this district.

SQL agent → SQL agent (Thought)

Now that I have the district_id with the second-highest number of crimes in 1995, I need to count the number of male clients in that district.
To do this, I will join the 'client' table with the 'district' table using the 'district_id'. I will then filter the clients by gender to count only male clients.
The SQL query for this step would look like this:

```
SELECT COUNT(*) FROM client WHERE gender = 'male' AND
district_id = (SELECT district_id FROM district ORDER BY A15
    DESC LIMIT 1
OFFSET 1);
```

This query will give me the count of male clients in the district with the second-highest number of crimes in 1995.
I will now send this query to the database expert agent for feedback.

SQL agent → Human proxy agent

Please review the following SQL query to ensure it correctly counts the number of male clients in the district with the second-highest number of crimes in 1995.

```
SELECT COUNT(*) FROM client WHERE gender = 'male' AND
district_id = (SELECT district_id FROM district ORDER BY A15
    DESC LIMIT 1
OFFSET 1);
```

Human proxy agent Tool Call: evaluate_sql

```
{
    'generated_sql': "SELECT COUNT(*) FROM client WHERE gender =
     'male' AND
district_id = (SELECT district_id FROM district ORDER BY A15
    DESC LIMIT 1
OFFSET 1);"
}
```

../../datasets/bird/dataset/dev/dev_databases\financial\financial.sqlite
Output match fail!

Human proxy agent Tool Response: evaluate_sql

```
{
    'test_sql': "SELECT COUNT(*) FROM client WHERE gender = '
    male' AND
district_id = (SELECT district_id FROM district ORDER BY A15
    DESC LIMIT 1
OFFSET 1);",
    'gold_sql': "SELECT COUNT(T1.client_id) FROM client AS T1
    INNER JOIN
district AS T2 ON T1.district_id = T2.district_id WHERE T1.
    gender = 'M' AND
T2.A15 = (SELECT T3.A15 FROM district AS T3 ORDER BY T3.A15 DESC
     LIMIT 1, 1)",
    'outcome': 'output-match-fail'
}
```

Human proxy agent → Human proxy agent (Thought)

The SQL query provided by the SQL agent does not match the expected output. The main issues are:
1. **Gender Filtering**: The query uses `gender = 'male'`, but the correct filter should be `gender = 'M'` as per the database schema.
2. **Subquery for District**: The subquery correctly identifies the district with the second-highest number of crimes, but it should return the `A15` value instead of `district_id` to match the expert solution.
3. **Table Join**: The query does not join the `client` and `district` tables, which is necessary to ensure the correct district is being referenced.
I will provide feedback to the SQL agent to address these issues.

Human proxy agent → SQL agent

The SQL query needs some adjustments to correctly count the number of male clients in the district with the second-highest number of crimes in 1995:
1. **Gender Filtering**: Use `gender = 'M'` instead of `gender = 'male'` to match the database schema.
2. **Subquery for District**: Ensure the subquery returns the `A15` value for the number of crimes, not the `district_id`. This is crucial for identifying the correct district.
3. **Table Join**: Implement an `INNER JOIN` between the `client` and `district` tables on `district_id` to ensure the correct district is being referenced.
Please revise the query with these points in mind.

SQL agent → SQL agent (Thought)

To revise the SQL query, I need to make the following changes based on the feedback:
1. Change the gender filter to `gender = 'M'` to match the database schema.
2. Correct the subquery to ensure it returns the `district_id` of the district with the second-highest number of crimes in 1995.
3. Implement an `INNER JOIN` between the `client` and `district` tables on `district_id` to ensure the correct district is being referenced.
The revised SQL query should look like this:

```
SELECT COUNT(*)
FROM client
INNER JOIN district ON client.district_id = district.district_id
WHERE client.gender = 'M'
AND district.district_id = (SELECT district_id FROM district
    ORDER BY A15 DESC LIMIT 1 OFFSET 1);
```

I will now send this revised query to the database expert agent for feedback.

SQL agent → Human proxy agent

Please review the revised SQL query to ensure it correctly counts the number of male clients in the district with the second-highest number of crimes in 1995.

```
SELECT COUNT(*)
FROM client
INNER JOIN district ON client.district_id = district.district_id
WHERE client.gender = 'M'
AND district.district_id = (SELECT district_id FROM district
    ORDER BY A15 DESC LIMIT 1 OFFSET 1);
```

Human proxy agent Tool Call: evaluate_sql

```
{
    'generated_sql': "SELECT COUNT(*) \nFROM client \nINNER JOIN
     district ON
client.district_id = district.district_id \nWHERE client.gender
    = 'M' \nAND
district.district_id = (SELECT district_id FROM district ORDER
    BY A15 DESC
LIMIT 1 OFFSET 1);"
}
```

../../datasets/bird/dataset/dev/dev_databases\financial\financial.sqlite
Output match!

Human proxy agent Tool Response: evaluate_sql


```
{
    'test_sql': "SELECT COUNT(*) \nFROM client \nINNER JOIN
    district ON
client.district_id = district.district_id \nWHERE client.gender
    = 'M' \nAND
district.district_id = (SELECT district_id FROM district ORDER
    BY A15 DESC
LIMIT 1 OFFSET 1);",
    'gold_sql': "SELECT COUNT(T1.client_id) FROM client AS T1
    INNER JOIN
district AS T2 ON T1.district_id = T2.district_id WHERE T1.
    gender = 'M' AND
T2.A15 = (SELECT T3.A15 FROM district AS T3 ORDER BY T3.A15 DESC
     LIMIT 1, 1)",
    'outcome': 'output-match'
}
```


Human proxy agent → SQL agent

The revised SQL query is now correct and accurately counts the number of male clients in the district with the second-highest number of crimes in 1995. Great job!

SQL agent → Human

The SQL query has been successfully generated and verified. It counts the number of male clients in the district with the second-highest number of crimes in 1995. Here is the query:

```
SELECT COUNT(*)
FROM client
INNER JOIN district ON client.district_id = district.district_id
WHERE client.gender = 'M'
AND district.district_id = (SELECT district_id FROM district
    ORDER BY A15 DESC LIMIT 1 OFFSET 1);
```

This query works by first identifying the district with the second-highest number of crimes in 1995 using a subquery. It then counts the number of male clients in that district by joining the client and district tables on district_id and filtering for male clients.

Query Result

```
[(96,)]
```

Output match!

Human → SQL agent

Please summarize what you learned from interacting with the Expert agent. Think about it out loud in many steps, and then give me a concise and specific summary. Do these steps sequentially. The output of the previous step is used for the subsequent steps.
1. First send yourself a message which describes the mistakes you made. This should explicitly state what component of the SQL query was incorrect. You should also pinpoint where this error occured in your reasoning. 2. Second, send yourself a message which details what database knowledge was revealed through feedback to correct this mistake. Link this is to the schema. Be very specific, and SQL heavy. This may contain information about the reasoning you had about the database. 3. Third, look at the thought generated by step 2. Rephrase the findings as facts. Provide some context as to when these facts would be useful in the future.
The only response that I want back is the set of facts. Leave the "generated_sql" field empty in your response.

SQL agent → SQL agent (Thought)

The mistakes I made in the initial SQL query were:
1. **Gender Filtering**: I used `gender = 'male'` instead of `gender = 'M'`. This mistake occurred because I assumed the gender values were stored as full words rather than single characters.
2. **Subquery for District**: I incorrectly assumed that the subquery should return the `district_id` directly without considering the need to order by the number of crimes (`A15`). This error was due to a lack of attention to the specific requirement of identifying the district with the second-highest number of crimes.
3. **Table Join**: I initially did not implement an `INNER JOIN` between the `client` and `district` tables, which is necessary to ensure the correct district is being referenced.

SQL agent → SQL agent (Thought)

The feedback revealed the following database knowledge:
1. **Gender Filtering**: The `client` table stores gender as a single character ('M' for male, 'F' for female). This is evident from the schema where the `gender` column is of type `text`.
2. **Subquery for District**: To find the district with the second-highest number of crimes in 1995, the subquery must order the `district` table by the `A15` column in descending order and then select the second entry. This ensures the correct district is identified based on the number of crimes.
3. **Table Join**: An `INNER JOIN` between the `client` and `district` tables on `district_id` is necessary to reference the correct district when counting clients. This join operation is crucial for combining data from both tables based on the shared `district_id`.

SQL agent → SQL agent (Thought)

Facts derived from the feedback:
1. **Gender Representation**: In the `client` table, gender is represented by single characters ('M' for male, 'F' for female). This is crucial when filtering clients by gender in SQL queries.
2. **Identifying Districts by Crime Rate**: To find a district based on crime rates, use the `A15` column in the `district` table. Order the districts by this column in descending order to identify districts with the highest number of crimes.
3. **Joining Tables on District ID**: When needing to reference district-specific data for clients, use an `INNER JOIN` between the `client` and `district` tables on `district_id`. This join is essential for combining client data with district data accurately.

SQL agent → Human

1. **Gender Representation**: In the `client` table, gender is represented by single characters ('M' for male, 'F' for female). This is crucial when filtering clients by gender in SQL queries.
2. **Identifying Districts by Crime Rate**: To find a district based on crime rates, use the `A15` column in the `district` table. Order the districts by this column in descending order to identify districts with the highest number of crimes.
3. **Joining Tables on District ID**: When needing to reference district-specific data for clients, use an `INNER JOIN` between the `client` and `district` tables on `district_id`. This join is essential for combining client data with district data accurately.

Human → SQL agent

Now save the distilled knowledge, one chunk at a time, for future use by using your save_memory tool. You can decompose the SQL query and save the chunks as sub tasks as useful reusable components in the future. You can also save database_facts, which contain useful information that may be necessary in the future in a more general set of tasks. You can save up to 5 memories. Plan and decide which are most important before you start. Here are some helpful tips: - Be sure to make the index of the memory useful. A useful index is one that will be easy to search for in the future. For example, dont just say "join table1 and table2". Say something like "join table1 and table2 by attribute", where attribute is the common column. - When saving a similar subtask, the index should generally describe the subtask. Always make the passage contain a chunk of SQL. The index must contain information about the values filtered for in the SQL if you keep the values in the passage. Otherwise, you should replace the filters with placeholders, and the index should reflect the generality. - Don't save basic SQL knowledge like "how to limit". This is information you already know. You could instead say "how to sort for __column name__" and then the passage can show that you order and use a desc and limit clause in a SQL query.
Some examples of what not to do:
'query_string': 'use of JOIN operations to connect tables', 'knowledge_string': 'JOIN operations are used to connect tables based on common columns to retrieve related data.', 'memory_type': 'database_fact'
Reason: This does not provide any useful information about the database and is just common SQL knowledge.
'query_string': 'filter by exact text values in SQL', 'knowledge_string': "When constructing SQL queries involving the 'frequency' column, it is essential to use the exact text values defined in the schema for accurate filtering. This ensures the query retrieves the intended results based on the issuance frequency.", 'memory_type': 'database_fact'
Reason: You should instead save "filter by issuance frequecy" as the index, and the the passage should be showing the portion of the SQL query of how to do this. If available, you should specify what the filter value corresponds to in the index.
'query_string': "filter accounts by frequency 'POPLATEK TYDNE'", 'knowledge_string': "SELECT * FROM account WHERE frequency = 'POPLATEK TYDNE';", 'memory_type': 'similar_subtask'
Reason: The query string is not helpful since you would need to know to search for POPLATEK TYDNE in the first place. The tricky part is knowing what this value semanticly means. Instead the query string should be "filter accounts for weekly issuance"

SQL agent Tool Call: save_memory

```
{
    'query_string': 'filter clients by gender',
    'knowledge_string': "SELECT * FROM client WHERE gender = 'M
    ';",
    'memory_type': 'similar_subtask'
}
```

SQL agent Tool Response: save_memory

Saved.

SQL agent Tool Call: save_memory

```
{
    'query_string': 'identify district with second-highest
    crimes in 1995',
    'knowledge_string': 'SELECT district_id FROM district ORDER
    BY A15 DESC
LIMIT 1 OFFSET 1;',
    'memory_type': 'similar_subtask'
}
```

SQL agent Tool Response: save_memory

Saved.

SQL agent Tool Call: save_memory

```
{
    'query_string': 'join client and district tables by
    district_id',
    'knowledge_string': 'SELECT * FROM client INNER JOIN
    district ON
client.district_id = district.district_id;',
    'memory_type': 'similar_subtask'
}
```

SQL agent Tool Response: save_memory

Saved.

SQL agent Tool Call: save_memory

```
{
    'query_string': 'gender representation in client table',
    'knowledge_string': "In the client table, gender is
    represented by single
characters ('M' for male, 'F' for female).",
    'memory_type': 'database_fact'
}
```

SQL agent Tool Response: save_memory

Saved.

SQL agent Tool Call: save_memory

```
{
    'query_string': 'sort districts by number of crimes in
    1995',
    'knowledge_string': 'To find a district based on crime rates
    , use the A15
column in the district table. Order the districts by this column
     in descending
order to identify districts with the highest number of crimes.',
    'memory_type': 'database_fact'
}
```

SQL agent Tool Response: save_memory

Saved.

SQL agent → Human

I have saved the most important pieces of knowledge from our interaction for future use. This includes how to filter clients by gender, identify districts with the second-highest crimes, join client and district tables by district_id, and some key database facts about gender representation and sorting districts by crime rates.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The three main claims are supported by empirical evidence. The feedback mechanism incorporates tacit knowledge. Continual learning is achieved by memory augmentation, and the three agent architectures are thoroughly compared.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Final paragraph of the paper explicitly discusses the limitation.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.

- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: No theoretical results are presented.

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: All prompts are disclosed to reproduce agent set up. Data preparation is discussed and we make use of an open source benchmark.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.

- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: Code will not be released, but the dataset used is open source. All details to reproduce the code is made available.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

Answer: [Yes]

Justification: Full details are contained in the main text and appendix.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   All results are averaged across 3 independent runs. Error bars are plotted in Figure 4.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
   - If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer:[Yes]

   Justification: Presented in Appendix A

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

Answer: [Yes]

Justification:

Guidelines:

- The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer:[NA] .

    Justification: This paper presents a methodology for continual learning in text-to-SQL pipelines that assumes a human expert providing feedback. Malicious or unintended use is beyond the scope of the paper.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: The paper proposes an improvement to text-to-SQL methodology.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.

- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Provided in appendix A.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [NA]

    Justification: No new assets are released.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer:[NA]

    Justification:No crowdsourcing or research with human subjects is performed.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.

- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA] .

    Justification: see answer above.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: The methodology uses an LLM based agent. An LLM is critical for usage of the core methods in this research.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.