Panda: Performance Debugging for Databases using LLM Agents

Vikramank Singh
AWS AI Labs
vkramas@amazon.com

Kapil Eknath Vaidya
Amazon Web Services
kapilvaidya24@gmail.com

Vinayshekhar Bannihatti Kumar
AWS AI Labs
vinayshk@amazon.com

Sopan Khosla
AWS AI Labs
sopankh@amazon.com

Balakrishnan Narayanaswamy
AWS AI Labs
muralibn@amazon.com

Rashmi Gangadharaiah
AWS AI Labs
rgangad@amazon.com

Tim Kraska
Amazon Web Services, MIT
kraska@mit.edu

ABSTRACT

Debugging a performance issue in databases is notoriously hard. Wouldn't it be convenient if there exists an oracle or a co-pilot for every database system which users can query in natural language (NL) — '*what's wrong?*', or even better— '*How to fix it?*'. Large Language Models (LLMs), like ChatGPT, seem to be a natural surrogate to this oracle given their ability to answer a wide range of questions by efficiently encoding vast amount of knowledge for e.g., a major chunk of the internet. However, prompting ChatGPT with database performance queries often results in 'technically correct' but highly 'vague' or 'generic' recommendations typically rendered useless and untrustworthy by experienced Database Engineers (DBEs).

In this work we propose **Panda**, a framework to provide context grounding to pre-trained LLMs in order to generate more 'useful' and 'in-context' troubleshooting recommendations. Panda draws inspiration from the way experienced DBEs perform debugging, and puts a system in place with necessary components required to robustly deploy pre-trained LLMs in production for debugging. The 4 key components of Panda are: (1) *Grounding*; (2) *Verification*; (3) *Affordance*; and (4) *Feedback*. We describe the necessity and usefulness of each component and how they communicate internally to transform a given pre-trained LLM into generating in-context, actionable, useful and accurate recommendation for debugging a given database system.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

database debugging, large language models

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

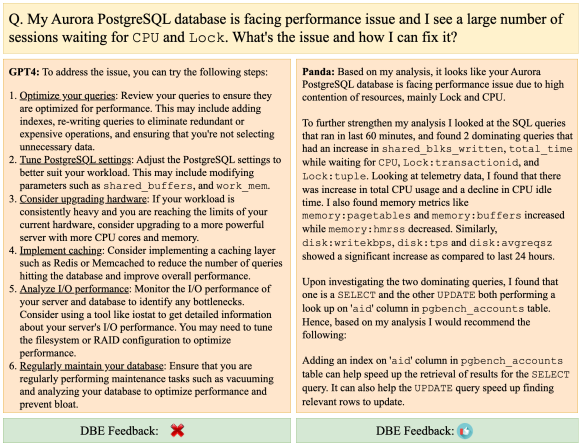

Figure 1: Pre-trained LLMs (like GPT-4) generate recommendations that are universally true but aren't actionable given the context of query, and hence deemed useless by DBEs for troubleshooting. Panda builds the right context and generates a specific actionable recommendation using a smaller cost-effective pre-trained LLM (GPT-3.5).

ACM Reference Format:
Vikramank Singh, Kapil Eknath Vaidya, Vinayshekhar Bannihatti Kumar, Sopan Khosla, Balakrishnan Narayanaswamy, Rashmi Gangadharaiah, and Tim Kraska. 2018. Panda: Performance Debugging for Databases using LLM Agents. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

We propose Panda, a simple yet effective mechanism which enables database owners to quickly analyze a database performance degradation and obtain recommendations to fix. Given that foundation models like LLMs [7, 18] are trained on a vast majority of internet corpus, they naturally appear to be a good candidate to answer these NL debugging questions as compared to classical approaches that require building complex models from telemetry or system logs, followed by human labeling to either identify the root causes

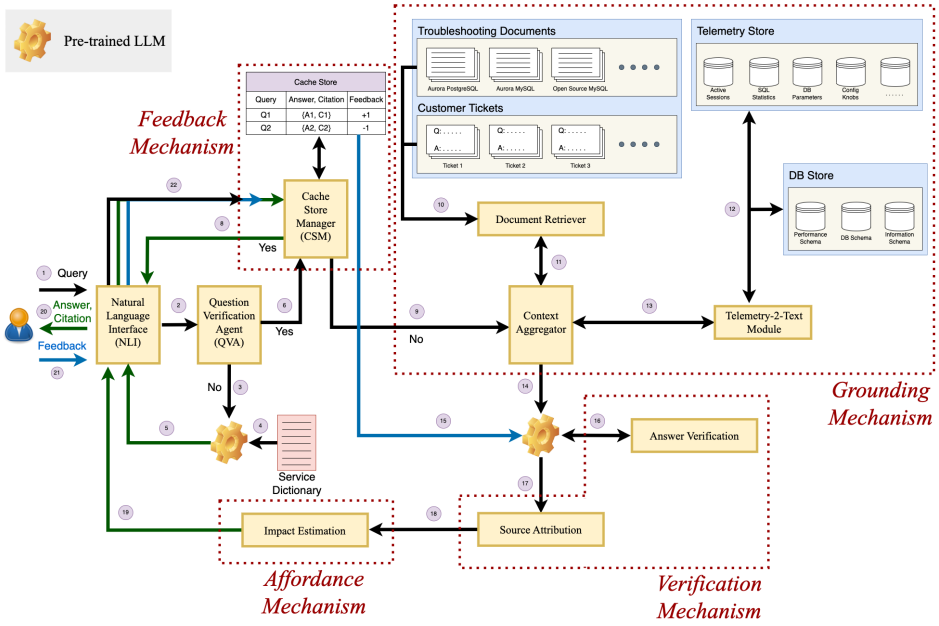

Figure 2: Panda Architecture: The system consists of– (a) *Grounding* mechanism that provides the necessary context about the user query; (b) *Verification* mechanism verifies for correctness and attributes sources in form of troubleshooting docs and resolved customer tickets; (c) *Affordance* mechanism estimates the performance impact for each of the generated recommendations; and (d) *Feedback* mechanism collects and stores feedback that LLM uses as context to improve its generation.

[9, 14, 16, 23] or potential knobs to tune [17, 22]. However, using these pre-trained LLMs in their natural form pose 2 key limitations that quickly render the generated recommendations useless— (a) Debugging is always contextual, i.e root cause of a performance degradation can vary from database to database, and time to time. (b) Debugging requires analysis of multi-modal data, i.e unstructured texts in form of logs, troubleshooting docs, and telemetry in form of database metrics (eg: query latency, memory pressure, cpu utilization, etc). Lacking the right context of the problem makes the LLM generate answers that are true in expectation but aren't particularly useful for the given issue. For e.g., in Fig 1 all the 6 recommendations made by GPT-4 [3] are technically correct, but are mere facts or best practices that hold true in almost every situation. Thus, to an experienced DBE these recommendations don't add any value in identifying the root cause or fixing the issue. This is a common problem seen in other domains like Robotics [5] where knowledge of physical world is necessary for the LLM to generate contextually correct answers. Retrieval Augmented Generation (RAG) [12] has emerged as an effective prompt tuning technique to augment the user prompt with necessary information to generate context specific answers without fine-tuning the model. RAG relies on an Information Retrieval (IR) approach where it uses an embedding model to embed the user prompt and a set of natural language documents, retrieves the relevant documents using a distance based similarity search, and adds them to the context before feeding it to the LLM. Although this has shown to tremendously improve the quality of final output, it is non-trivial to do RAG with multi-modal data, e.g., text and telemetry, which is a key modality for performance troubleshooting. Furthermore, there are additional concerns when it comes to applying the generated recommendation on a real production database like, Trust: '*where is this knowledge coming from?*'; Impact: '*what would happen to my database if I make this recommended change?*'; Feedback: '*this recommendation was useless, keep it in mind for future*'; and Risk: '*how catastrophic is this advice? (e.g., dropping tables?)*' Humans can respond to these questions relatively easily either using domain knowledge or by gathering experimental data. But they are fairly non-trivial for a billion parameter black-box LLM, rendering it unsafe/untrustworthy to deploy in real-world. Thus the natural question to ask here is— '***What are the essential building blocks that we need in order to safely deploy a LLM for debugging in production which can generate accurate, verifiable, actionable, and useful recommendations?***'

This is an open question, and a highly ambiguous one given the definition of 'accurate', 'verifiable' and 'actionable' are contextual and subjective. However, in this work we propose a system that attempts to address these open concerns. More specifically, we argue that a LLM-driven autonomous database debugging agent should possess following 4 properties to be ready for 'real-world':

- **Grounding**: Given a NL user query, system should be able to extract relevant information from appropriate sources to build the necessary context before generating an answer. This involves analyzing multi-modal data like logs, related historical tickets, troubleshooting docs, and telemetry.
- **Verification**: The system should be able to verify the generated answer using relevant sources and produce the citation along with its output so the end user can verify it.

	DBE Scorers	Trust	Understand	Useful
GPT-4	Beginner	4 (8%)	18 (36%)	15 (30%)
	Intermediate	3 (6%)	11 (22%)	4 (8%)
	Advanced	3 (6%)	12 (24%)	3 (6%)
Panda	Beginner	**46 (92%)**	**32 (64%)**	**35 (70%)**
	Intermediate	**47 (94%)**	**39 (78%)**	**46 (92%)**
	Advanced	**47 (94%)**	**38 (76%)**	**47 (94%)**

Table 1: Human evaluation of the recommendation quality.

- **Affordance**: If the recommendation is actionable, system should be able to estimate and inform the user about consequences of that action along with explicitly highlighting high-risk actions (e.g., DROP, DELETE, etc).
- **Feedback**: The system should be able to accept feedback from the user and account for those when generating responses.

Given its ubiquity we use the example of cloud databases (for e.g., Aurora PostgreSQL [2], and MySQL [1]) to explain Panda. However, the proposed system is general in that it can be easily extended to any database system.

2 PANDA ARCHITECTURE

As illustrated in Figure 2, Panda consists of 5 key components: (1) *Question Verification Agent* (QVA) to identify and filter out the irrelevant queries; (2) *Grounding Mechanism* comprising of Document Retriever that extracts relevant documents; Telemetry-2-Text (T2T) that extract features from telemetry and converts them to text, and Context Aggregator combines all of this to augment the prompt; (3) *Verification Mechanism* comprising of Answer Verification and Source Attribution; (4) *Feedback Mechanism*; and (5) *Affordance Mechanism*. All these mechanisms work in the background of a NL interface which user interacts with. We now discuss each of these components in detail.

2.1 Question Verification Agent (QVA)

Panda introduces QVA as a gatekeeper that rejects the queries that are deemed '*out-of-context*' to the given use-case/domain, in this case database performance debugging. Thus, given a user query i, QVA produces a binary label $\{Yes, No\}$ on whether its a relevant query or not, as shown by arrows numbered 2,3,6 in Fig 2. However, just saying '*No*' doesn't solve user problem, and hence QVA also outputs the name of the service/tool which can answer that given user query (e.g., if the question is related to reading an S3 bucket, or spinning up an EC2 instance, etc). To achieve this goal, Panda adopts the RAG approach [12] and uses an embedding model to first obtain a vector representation of user query i. It then creates a vector store where it stores the embeddings of all available troubleshooting docs and historical customer tickets using the same embedding model, and then runs an embedding based similarity search to compute a distance score (e.g., cosine) between the query embedding and elements in vector store [8] and returns the document with highest score. If the returned score is below a set threshold λ, Panda deems the query to be '*out-of-context*'. We set $\lambda = 0.85$ but can be tuned as per the use-case. For a given '*out-of-context*' query, Panda then performs the same embedding based similarity search b/w the query and embeddings of AWS service description documents [1] to identify the name of service that most likely relates to the user query (See arrows 3,4,5 in Fig 2). For query that doesn't overlap with any of the listed AWS service (e.g., *How's the weather today?*), Panda requests the user to revise the query or provide more context.

2.2 Grounding Mechanism

Panda builds two types of contexts to ground the LLM— *Global* and *Local*. Former gathers information regarding *'How is this problem usually solved?', 'what metrics to look at', 'best-practices'*, etc; while latter gathers information regarding user's database, recent queries, configurations, etc. Both these contexts are key to understanding the problem, and producing a useful solution.

Document Retriever. Panda collects high-level information in form of troubleshooting documents, and historically resolved customer tickets to build the right *global* context. Panda utilizes the same mechanism described in Sec 2.1 to retrieve the relevant documents. These are typically 2-4 page long documents[2] written by domain experts with defined sections like *'Likely causes', 'Corner cases', 'Actions'*, etc, while tickets are short Q/A pairs with a concise problem statement and a verified answer.

Feature Extractor. Panda also extracts relevant telemetry data related to the user query. From a provided list of telemetry metric names and their descriptions[3], Panda first identifies the top-K (K=3) relevant metric names from retrieved docs using the same embedding based similarity search. It then queries the data lake to collect the raw time series data for those metrics. The default setting is to collect last 24 hours of data, but can be tuned based on the use-case. Panda uses a change point detection algorithm [21] to divide the time series into *problematic* and *baseline* regions and runs a set of classic time series detectors, on this data to extract features relevant for debugging performance issues. Inspired from what DBEs look at when analyzing telemetry data, we extract interpretable features like trend, seasonality, mean, p95, and correlation score for each metric. Panda then uses these features to identify *problematic* metrics. Furthermore, it also identifies the *problematic* SQL queries that had at-least one query statistics[4] identified as *anomalous*. However, the telemetry alone isn't sufficient to make an informed recommendation, for e.g., in order to recommend adding an index on a particular column, Panda needs to know if there already exists an index on that column or not? Thus, to provide this domain specific knowledge, Panda accesses 2 key things– (a) For queries identified as problematic, it collects their raw SQL statements; (b) Database and Information Schema necessary to understand the tables, columns, views, etc.

Context Aggregator. All these extracted features are stored as text in form of a structured object (e.g., JSON), and combined along with the retrieved docs and tickets. We found JSON formatting of prompt to work really well as compared to unstructured large text paragraphs, but other structures like bulleted, numbered, etc too give similar results. As LLMs have a fixed size context window,

[1] An example of AWS Service List
[2] An example of Aurora troubleshooting document
[3] An example of telemetry list and their descriptions
[4] A list of SQL statistics for PostgreSQL

Engine	Problem Type	Dominating Wait Event	Description
APG	Poor Application Design	`Lock:Tuple`	High number of concurrent sessions trying to acquire conflicting lock for same tuple by running UPDATE and DELETE statements
	Poor Transaction Management	`Lock:TransactionId`	Result of long running transactions that hold longer locks blocking other transcations from running or high concurrency
	Network Congestion	`Client:ClientRead`	Connection is in idle transaction state and is waiting for a client to send more data or issue a command
AMS	Workload Spike	`io/table/sql/handler`	Greatly increase the rate of I/O transactions
	Storage Latency	`io/redo_log_flush`	Database doing excessive commits and write operations
	Poor Configuration or Poor Application Design	`synch/mutex/innodb/ buf_pool_mutex`	A thread has acquired a lock on the InnoDB buffer pool to access a page in memory

Table 2: 6 common performance degradation scenarios used in our experiments (out of total 50).

we make a trade-off to restrict the prompt size by further breaking down large documents into smaller chunks and storing only top-3 relevant chunks. Similarly, we only include the top-3 most problematic SQL query texts.

2.3 Feedback Mechanism

Cache Store Manager (CSM). Any real-world system needs to improve over time, and utilizing feedback is one way of doing it. Panda achieves this by requesting the user to provide a binary feedback in terms of '*Helpful*' or '*Not Helpful*' for each generated response. This feedback along with the user query and its response are stored in a vector store indexed by the vector embedding of the user query (See arrows numbered 22, 15 in Fig 2). When generating a new response, Panda first retrieves top-K similar user responses from this vector store (if they exist) by using the similarity search on query embedding, and adds them to the context. Panda retrieves at-least 1 and at-most 3 responses of both positive and negative feedback. This kind of few-shot in-context learning techniques are shown to significantly improve the output quality [7].

2.4 Verification Mechanism

Answer Verification. Panda frames the answer verification problem as a Natural Language Inference (NLI) task [6] where it reuses the pre-trained LLM to act as a verifier and produce a label as *accept*, *reject*, or *neutral* given a 'hypothesis' (i.e, the generated answer) and a 'premise' (i.e, the retrieved documents and tickets). If the model answers *reject* or *neutral*, Panda reruns the generation process to obtain a new answer and verifies it. After K unsuccessful tries (we set $K = 3$), Panda produces a failure message saying, *"Apologies, I do not possess enough information to answer this question. You can retry by rephrasing the query. Also, here are some relevant documents that you can use to get started"*. Note that customer tickets are removed from the list of relevant docs shown to the user as they may contain sensitive data.

Source Attribution. Backing the recommendations with authentic external sources is a reasonable way to ensure user's trust in these generative models. Once the answer is verified, Panda along with the final output, provides the retrieved sources in form of passages from retrieved docs (excluding the customer tickets) as references used by its LLM to generate the output. Furthermore, Panda splits the output and the reference passages in individual sentences and provides inline citation if the output sentences are taken verbatim from the reference passages as shown in Table 3, 4.

2.5 Affordance Mechanism

Impact Estimation. Goal of this module is to provide user with an estimate of change in the database performance (measured w.r.t a target metric, e.g., number of active sessions, avg. query latency, cpu util, etc) for a given fix that will help them quantify how likely it is for their problem to get resolved if they choose to apply it. There are two ways to answer this— (a) Generic (less precise): infer what's the usual impact seen on similar databases captured in retrieved documents and tickets; (b) Specific: use external models to predict the change in performance for a given action (e.g., adding an index, tuning a knob, etc). Latter requires having an access to a backup database or learned behavior models, something in which prior works [22] have shown promise. Panda in its current form mainly relies on the former, while using simple statistical behavior models for some of the important actions (e.g., changes in instance class, engine version, etc) to predict impact on performance metrics like no. of active sessions [14], and query latency [16]. Furthermore, Panda maintains a list of potentially high-risk actions apriori (e.g., dropping a table, changing isolation levels, etc) and highlights them in the output if present. We are continuously working to increase the scope of this module by incorporating existing behavior models like Learned Cost and Materialized View Estimators [15].

3 CASE STUDY ON REAL WORKLOADS

Setup. We conduct a small experimental study of real-world problematic database workloads to test the reliability and usefulness of Panda. We use GPT-3.5 as the base pre-trained LLM in Panda and compare its performance with a pre-trained GPT-4. For embeddings we use the `text-embedding-ada-002` model by OpenAI [3]. We use the wait event docs for both Aurora PostgreSQL (APG) and MySQL (AMS) constituting a total of $2.5K$ documents, i.e, a total of $6.25M$ tokens. The telemetry data consists of a total of 250 database metrics collected over a period of 7 days at 1-minute granularity combining a total of $2.5M$ data points. We manually design debugging queries around the most common wait events that are commonly asked by Performance Insights [4] customers. We use 50 queries, 25 each for AMS and APG. Due to space constraints, in Table 2 we only show 6 most common problem types out of the

Query: For my Aurora PostgreSQL database, I see a lot of sessions waiting for `lock:tuple` and `CPU`. What's the issue and how can I fix it?
Panda: Based on the information provided, it appears that the problem with your database is related to high concurrency and lock contention. The increase in shared block writes and the presence of waiting events for tuple locks suggests that multiple sessions are trying to modify the same rows, causing resource contention and high waiting times. To bring down the number of sessions, I recommend investigating your application logic and identifying any blocker sessions that have been in the idle in transaction state for a long time. Consider ending these blocker sessions using the `pg_terminate_backend` function as a short-term solution[s_1]. For a long-term solution, adjust your application logic to reduce concurrency when it is high. Additionally, you can use the `aurora_stat_backend_waits` function to analyze past `Lock:tuple` events and identify any patterns or common causes[s_2]. Based on the database counters, it would be helpful to monitor and optimize queries that are causing high shared block writes and increasing the usage of CPU and locks. This can be achieved by identifying and tuning the problematic SQL queries that are causing the contention. ***Sources:*** [s_1]: Amazon Aurora User Guide for Aurora Pg.1397, L.2. [s_2]: Amazon Aurora User Guide for Aurora Pg.1397, L.20.
GPT-4: If you're experiencing an increase in the number of sessions waiting for locks and CPU resources it could indicate performance issues or contention problems within the. database. Here are some steps you can take to diagnose and address the problem: ***Identify the source of contention:*** Check the database logs and monitoring tools to determine which queries or operations are causing the most contention. Look for long-running or poorly optimized queries that might be causing excessive lock contention or high CPU usage. ***Optimize queries:*** Review the queries identified in the previous step and ensure they are properly optimized. Consider adding or modifying indexes, rewriting queries, or adjusting the database schema to improve query performance. ***Consider scaling up:*** If your database is consistently reaching its resource limits and the optimizations mentioned above do not provide sufficient relief, it may be necessary to scale up your Aurora PostgreSQL instance by increasing its compute and storage capacity.
DBE Feedback: Panda offered concrete advice tailored to the problem with sources that I could look into and trust. However, GPT-4 offers a group of relevance but general practice advice that are universally true. Between the two I'd prefer Panda especially if I am a non-expert and don't know where to start from.

Table 3: Comparison of recommendations produced by Panda and GPT-4 rated by DBEs.

Query: For my Aurora MySQL database, I see a lot of sessions waiting for `cpu` and `io/redo_log_flush`. What's the issue, and how can I fix it?
Panda: Based on the task information and reference documents, the likely cause of the performance issue is an increase in wait events related to cpu, redo log flush. This is typically caused by excessive commits and write I/O operations[s_1]. To fix this issue, the recommended action is to group your write operations together to reduce the number of commits[s_2]. Instead of inserting records row by row or in small batches, you should consider larger batch sizes[s_3] or using bulk insert operations. By reducing the, number of commits you can minimize the wait events related to write I/O operations and, mutex lock contention thereby improving the performance of your Aurora MySQL database. ***Sources:*** [s_1]: Amazon Aurora User Guide for Aurora Pg.901, L.26. [s_2]: Amazon Aurora User Guide for Aurora Pg.902, L.22. [s_3]: Amazon Aurora User Guide for Aurora Pg.903, L.44.
GPT-4: If you are experiencing a high number of sessions waiting for 'redo log flush', and 'CPU' in your Aurora MySQL database, it could indicate potential performance issues: Optimize your database design and queries to reduce the number and duration of transactions. Ensure your transactions are committing or rolling back promptly. Consider breaking down large transactions into smaller ones to reduce contention. Monitor and tune the `innodb_buffer_pool_size` and `innodb_buffer_pool_instances` configuration parameters to optimize memory usage. Monitor your disk I/O performance to identify any bottlenecks. Consider optimizing your queries and transaction management to minimize the amount of data written to the redo log. Ensure that your database server has sufficient disk throughput and IOPS for your workload. Adjust the `innodb_log_file_size` configuration parameter to an appropriate value based on your workload.
DBE Feedback: GPT-4 is incorrect. Aurora doesn't use `innodb_log_file_size`. Also, "*breaking down large transactions into smaller ones to reduce contention*" is the opposite of what should be done here. Panda clearly outperforms GPT-4 in this scenario.

Table 4: Comparison of recommendations produced by Panda and GPT-4 rated by DBEs.

50 used in experiments. As human evaluation [14, 16] is key in estimating the usefulness of performance recommendations, similar to [23], we gather a group of 3 DBEs with 3 different level of competency based on their experience: *Preliminary DB Knowledge* (e.g.,SQL knowledge or undergraduate course on databases), *DB Usage Experience*, and *DB Research* or *DBE Experience*. Each DBE is shown two anonymized answers and asked to score them out of 3. Similar to [11], the 3 dimensions used for scoring are– (1) given the two answers, which one do they **trust** more; (2) which explanation do they **understand** better; and (3) which explanation do they find more **useful**. If all the criteria are met, the score is 3, if none is met, score is 0, and so on. For each query we also asked them to justify in few words what they liked in particular about an answer.

Preliminary Results. Table 1 summarizes the aggregated statistics of the study. Intermediate and Advanced DBEs demonstrated similar behavior where they found Panda answers to be both- more trustworthy (94% of times) based on its source citations, and more useful (94% of times) based on its correctness arising due to grounding with telemetry and troubleshooting documents. On the other hand, although GPT-4 answers sometimes where more understandable (~25% of times), they were found to be often too generic and/or incorrect (See examples in Table 3 and 4). Interestingly, Panda was rated relatively lower on its *understandability* which the DBEs attributed to the fact that sometimes Panda produced correct but incomplete answers that lacked enough context. This was further validated in ratings provided by *Beginner* DBEs for *usefulness*. Although they too found Panda to be superior across all 3 dimensions, in some cases (30%) they found Panda's answers to be too specific or even verbatim from troubleshooting documents that lacked enough context for a novice to utilize it as compared to GPT-4 answers, despite the lack of trust in those answers (8%). However, on the other hand both *intermediate* and *expert* DBEs found that specific quality of Panda to be very useful. This contrast in usefulness of Panda wasn't immediately obvious when we started the experiment.

Furthermore, we use a two-sample T-Test to check for statistical significance b/w the aggregate scores (i.e, total score out of 3) obtained by the two systems for each scorer. We use a p-value of 0.05 to test the null hypothesis that `mean` score of two populations are same. For each of the 3 scorers, we obtain a p-value $> 10^{-20}$, thus rejecting the null hypothesis and proving the statistical superiority of Panda over vanilla GPT-4 in this initial study of 50 cases.

4 RESEARCH DIRECTIONS

Improving IR with custom embeddings. One of the building blocks of Panda is RAG based prompt tuning technique which relies heavily on the underlying embedding model and a retriever function (e.g., cosine distance). General purpose embedding models [19] aren't effective at embedding domain specific texts and concepts, leading to both False Positives and Negatives in retrieval processes.

Moreover, learning custom embeddings is expensive in both, data-quantity and cost. Cheaper heuristics exist[5], for e.g., learning a linear transformation on top of general purpose embedding via a contrastive loss to bias the embeddings to a given domain. However, this is an open area that will impact usefulness of IR for niche domains, e.g., database systems.

Handling long multi-modal documents. Troubleshooting docs and tickets contain useful information in different modalities, e.g., screenshots, code snippets, and text which makes it difficult to parse and embed them using standard text embedding models. Also, usually problems require information that is spread across multiple docs and tickets, and accommodating everything in context quickly hits the limit. Some common ways to solve these problems are to ignore all other modalities except text, and use overlapping windows to breakdown a long document into chunks and retrieve relevant chunks. This is effective but leads to incomplete information in chunks that impact the answer quality.

Telemetry to text. Telemetry data is sequential and lengthy like text, but LLMs aren't trained on them. Successful emergent behaviors in LLMs are demonstrated on toy telemetry datasets [13] but telemetry in real world is long, multi-dimensional, and noisy. There exists some work on converting them to text [10], including what we propose in Panda, but what is the most effective way to— (a) segment out relevant parts of telemetry; (b) convert those into texts, are open research problems that will impact many telemetry driven domains that like finance, healthcare, etc.

Verification and Attribution. Panda uses textual overlap as a safe-guarding mechanism to attribute sources which isn't robust as LLMs are known to paraphrase and as a result hallucinate. Recent study [24] shows that only 51.5% of LLM generated answers truly support their citations, thus undermining their trustworthiness in real world. Human verification is reliable but unscalable and costly. Both verification and attribution are open research problems.

LLMs as tool manager. Inability of LLMs to perform complicated mathematical operations limits Panda from computing accurate affordances, which is key for user to decide whether or not to follow an advice. Recent works [20] show that using LLMs as reasoners which can query right external tools can solve this problem. This is a new and relevant research area where given the right context, Panda acts as a controller and selects the right behavior model (e.g., OtterTune [22], MB2 [15]) to estimate affordances where all the computational complexity resides within these external models.

5 CONCLUSION

In this paper we present Panda, a novel system designed for autonomous debugging of database systems using NL agents. We show Panda's capability to infer and reject irrelevant queries, build relevant multi-modal context, estimate affordances, provide citations with the answer, and also improve over time using feedback, all of which are key for generating useful and trustworthy recommendations. We also present various open research questions that we encountered in the early stages of this journey and encourage the database and systems communities to join us in answering them and re-imagining the entire debugging process for databases in general.

[5]OpenAI Custom embeddings

REFERENCES

[1] [n. d.]. Amazon Aurora MySQL. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.AuroraMySQL.html.

[2] [n. d.]. Amazon Aurora PostgreSQL. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.AuroraPostgreSQL.html.

[3] [n. d.]. OpenAI (2023). https://cdn.openai.com/papers/gpt-4.pdf.

[4] [n. d.]. Performance Insights. https://aws.amazon.com/rds/performance-insights/.

[5] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. 2022. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691* (2022).

[6] Samuel R Bowman, Gabor Angeli, Christopher Potts, and Christopher D Manning. 2015. A large annotated corpus for learning natural language inference. *arXiv preprint arXiv:1508.05326* (2015).

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[8] Zui Chen, Zihui Gu, Lei Cao, Ju Fan, Sam Madden, and Nan Tang. 2023. Symphony: Towards natural language query answering over multi-modal data lakes. In *Conference on Innovative Data Systems Research, CIDR*. 8–151.

[9] Brad Glasbergen, Michael Abebe, Khuzaima Daudjee, and Amit Levi. 2020. Sentinel: universal analysis and insight for data systems. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2720–2733.

[10] Shima Imani, Frank Madrid, Wei Ding, Scott E Crouter, and Eamonn Keogh. 2020. Introducing time series snippets: a new primitive for summarizing long time series. *Data Mining and Knowledge Discovery* 34 (2020), 1713–1743.

[11] Tushar Khot, Daniel Khashabi, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2020. Text modular networks: Learning to decompose tasks in the language of existing models. *arXiv preprint arXiv:2009.00751* (2020).

[12] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[13] Xin Liu, Daniel McDuff, Geza Kovacs, Isaac Galatzer-Levy, Jacob Sunshine, Jiening Zhan, Ming-Zher Poh, Shun Liao, Paolo Di Achille, and Shwetak Patel. 2023. Large Language Models are Few-Shot Health Learners. *arXiv preprint arXiv:2305.15525* (2023).

[14] Xiaoze Liu, Zheng Yin, Chao Zhao, Congcong Ge, Lu Chen, Yunjun Gao, Dimeng Li, Ziting Wang, Gaozhong Liang, Jian Tan, et al. 2022. PinSQL: Pinpoint Root Cause SQLs to Resolve Performance Issues in Cloud Databases. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2549–2561.

[15] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. 2021. MB2: decomposed behavior modeling for self-driving database management systems. In *Proceedings of the 2021 International Conference on Management of Data*. 1248–1261.

[16] Minghua Ma, Zheng Yin, Shenglin Zhang, Sheng Wang, Christopher Zheng, Xinhao Jiang, Hanwen Hu, Cheng Luo, Yilin Li, Nengjun Qiu, et al. 2020. Diagnosing root causes of intermittent slow queries in cloud databases. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1176–1189.

[17] Andrew Pavlo, Matthew Butrovich, Lin Ma, Prashanth Menon, Wan Shen Lim, Dana Van Aken, and William Zhang. 2021. Make your database system dream of electric sheep: towards self-driving operation. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3211–3221.

[18] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[19] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019).

[20] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761* (2023).

[21] Charles Truong, Laurent Oudre, and Nicolas Vayatis. 2020. Selective review of offline change point detection methods. *Signal Processing* 167 (2020), 107299.

[22] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*. 1009–1024.

[23] Dong Young Yoon, Ning Niu, and Barzan Mozafari. 2016. Dbsherlock: A performance diagnostic tool for transactional databases. In *Proceedings of the 2016 international conference on management of data*. 1599–1614.

[24] Xiang Yue, Boshi Wang, Kai Zhang, Ziru Chen, Yu Su, and Huan Sun. 2023. Automatic evaluation of attribution by large language models. *arXiv preprint arXiv:2305.06311* (2023).