



Journal of Multidisciplinary and Translational Research (JMTR)

journal homepage: <https://journals.kln.ac.lk/jmtr/>



Large language models for automating database operations in relational databases

O.A.M. Jayawardana^{1*}

¹*Independent Researcher*

Abstract

Controlling database systems using traditional interfaces requires a strong understanding of Structured Query Language (SQL) and the database administration tools employed. This study introduces a new AI-based system that automates create, read, update, and delete database actions through natural language processing, supporting multiple natural languages and diverse user vocabularies. The fundamental architectural innovation of this research consists of developing a bridge between Google's Gemini Large Language Model and the Google Agent Development Kit, referred to as the Gemini-Agent Development Kit Bridge. While the Agent Development Kit is built on a set of structured commands and handles all database query execution, Google's Gemini serves as a smart communication layer between the user and the Agent Development Kit. Thus, the main responsibility of Gemini is the translation from the entire range of natural language used by the user into a structured command recognizable by the Agent Development Kit, regardless of its vocabulary, informality, or language. In the same way, the output of the Agent Development Kit is transformed into human-readable output by Gemini. The proposed solution was designed as a Flask-based e-commerce management system and evaluated using 325 manually authored test cases covering five operational classes. On average, 93.8% accuracy was achieved for the translation of commands, with an accuracy of 96.0% for order deletion and 87.5% for complex multi-step operations. The response time varied between 1.2 seconds for simple commands and 3.5 seconds for complex operations. This finding has substantial implications, as it suggests that Gemini's broad natural-language capabilities can be employed to expand access to structured backend systems without altering the execution layer.

Keywords: Natural language processing, database automation, large language models, SQL

Article info

Article history:

Received 10th June 2026

Received in revised form 10th July 2026

Accepted 15th August 2026

Available online 15th September 2026

ISSN (E-Copy): ISSN 3051-5262

ISSN (Hard copy): ISSN 3051-5602

Doi: <https://doi.org/10.4038/jmtr.v11i4.327>

ORCID iD: <https://orcid.org/0009-0009-3410-5832>

*Corresponding author:

E-mail address: amjmadusanka@gmail.com

(O.A.M. Jayawardana)

[CC BY-NC-SA](#) © 2026, The Author(s)

Introduction

Database management systems are critical infrastructure elements in contemporary enterprise systems and web-based applications. Effective use of database management systems, however, has conventionally necessitated special technical expertise, primarily the ability to write SQL queries, which is still the language of choice when interacting with relational databases (Woods, 1973). For non-technical users such as business managers, domain specialists, and operational personnel, the requirement becomes a perpetual obstacle to data manipulation and administration. The use of natural language interfaces for interaction with databases is a research field dating back more than half a century. LUNAR (Woods, 1973), as well as LIFER/LADDER (Hendrix et al., 1978), showed that natural language can indeed be parsed and translated into structured queries but were limited by the rule-based and domain-specific nature of these systems. The advent of statistical natural language processing techniques and later on, deep learning methods, has revolutionized the accuracy upper limit of such systems significantly.

The rise of Large Language Models has accelerated this trend further. Brown et al. (2020) illustrated that Generative Pre-trained Transformer-3 could generalize to complex language processing tasks via few-shot prompting, setting a trend for leveraging pre-trained models in structured output generation. The Spider benchmark (Yu et al., 2018) included 10,181 natural language questions linked to complex SQL queries across 200 databases and established a standard evaluation protocol for cross-domain text-to-SQL systems. Further research, such as X-SQL (He et al., 2019), demonstrated that incorporating schema context via Bidirectional Encoder Representations from Transformers (BERT) pre-training could achieve state-of-the-art results on the WikiSQL dataset (Devlin et al., 2019; Zhong et al., 2017). Lastly, Google's Gemini multimodal model family (Anil et al., 2023), owing to its ability to reason structurally and follow instructions, proved highly effective at generating commands.

Although there has been considerable development, the practical implementation of NLP-powered database management systems remains relatively underexplored. Literature related to e-commerce automation has mainly focused on end-user systems such as recommendation engines (Smith & Linden, 2017) and chatbots (Cheng et al., 2022). In contrast, back-end processes like database management have been neglected. One of the key challenges is the difference between the unpredictable vocabulary used by humans and the predictable syntax structure expected by an execution engine. Users may express the same task in numerous ways, using different languages and levels of specificity. Creating a solution to translate both types of instructions without forcing users to memorize a controlled vocabulary or command language is technically challenging but important in practice.

This study attempts to fill this gap by proposing a fully functional, prototype-stage AI-driven database automation system based on a unique Gemini-Agent Development Kit Bridge Architecture. The system was created and tested on an e-commerce management platform, based on the Flask framework, managing product inventories, customer orders, and order life cycle management. The key idea behind the proposed system architecture lies in separating the tasks of understanding language and executing commands: the Google Gemini agent operates as a two-way communication channel between the two sides, translating a wide variety of natural languages into structured Agent Development Kit commands on the incoming channel. The Agent Development Kit layer handles database communication and query execution using only a finite,

predefined set of commands, staying completely detached from the actual text entered by the users. This allows the system to support a wide range of different vocabularies, informal language, and even other languages without modifying the execution layer. The goals of this study include developing and testing a Gemini-Agent Development Kit Bridge architecture capable of producing precise database instructions from natural language inputs; assessing the performance of the proposed system under various operations using a set of standardized tests; comparing the AI-driven method against classic SQL and graphical user interface (GUI) approaches to administrative tasks in terms of speed, precision, and accessibility; and verifying the multilingualism of the proposed architecture.

The proposed system was developed and evaluated in a controlled development environment using a Flask-based server with an SQLite database and a test suite comprising 325 manually authored test cases. The evaluation was conducted exclusively within this development environment and did not involve deployment in a live production setting or validation using large-scale real-world datasets. Accordingly, the reported findings reflect the system's performance under the specified development and testing conditions.

Methodology

This system implements a dual-layer core architecture, where Layer 1 (Gemini) handles language comprehension and Layer 2 (Google Agent Development Kit) handles database execution. Each layer is solely responsible for either language comprehension or database execution. The execution layer is implemented using the Google Agent Development Kit (Google, 2025). This layer accepts and processes a predefined set of structured commands, performing translation of commands into queries in the database and producing structured output. Gemini constitutes the intermediary layer, connecting the agent and user and facilitating the transfer of information in both directions. In the first direction, Gemini obtains the user query written in any natural language and translates it into commands understood by the development kit. On the other side, Gemini receives structured output and presents it to the user in human-readable form. It is important to note that no interaction with the user takes place at the level of the development kit; only the intermediary layer uses natural language.

System architecture

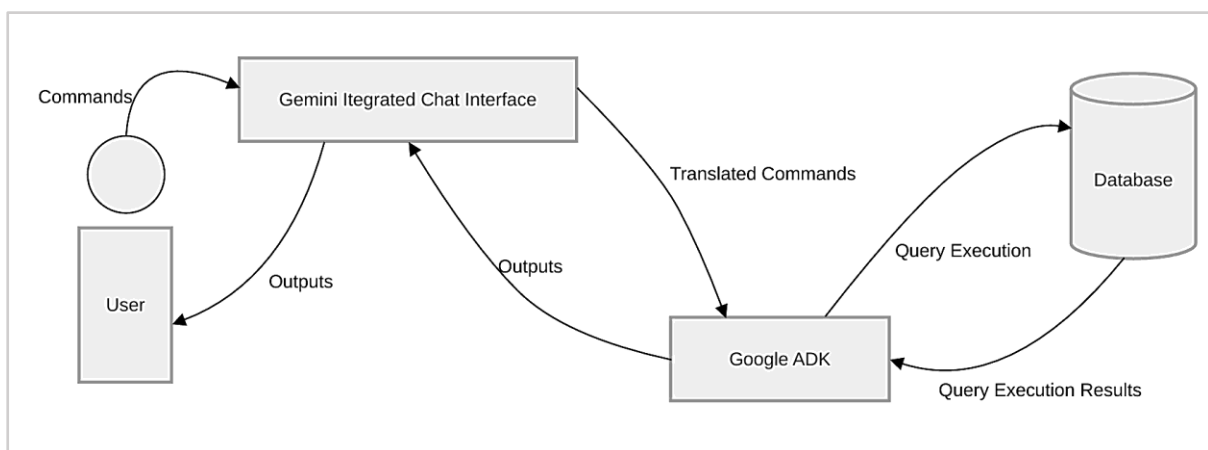


Figure 1. System architecture

As depicted in Figure 1, the dual-layer core architecture is deployed across three system components: (1) the user interface layer, (2) the Gemini-integrated chat interface (which implements the bridge between the two core layers), and (3) the backend layer that integrates Google Agent Development Kit (ADK) along with a database. The user begins an interaction process through command submission via the chat interface, which acts as the main interface for all interactions within the system. Once the user's input is received, the Gemini-integrated chat interface decodes the natural language command into a structured form. Afterward, the structured command is passed to the Google ADK, which acts as the main backend service responsible for handling the execution process. Google ADK parses the structured command and communicates with the database to execute the corresponding operations that include CRUD tasks. The results from the database operation are sent back to the Google ADK, which passes the output to the Gemini-integrated chat interface. The chat interface displays the result to the user, thus completing the interaction process flow. In summary, the system's core architecture consists of two conceptual layers Gemini for language processing and the ADK for database operations which are deployed across the three system components described above (the user interface, the chat interface, and the backend) to form the complete system.

Natural language processing pipeline

When a query is entered by the user, it goes directly to the Gemini model, together with the system prompt. The system prompt includes all commands of the Agent Development Kit and all their possible combinations, as well as required parameters and expected output types. Gemini is tasked with analyzing the user's intention and extracting any entities, such as order IDs, product names, amounts, and status types. Based on that, Gemini is supposed to output one line, representing one Agent Development Kit command. This implies that Gemini will perform tokenisation, extraction of entities, classification of intents, and generating the command within a single inference call.

The Agent Development Kit command generated by Gemini is validated against the permissible command schema prior to execution. In the event that the command structure is correct, the command is executed against the database via the Agent Development Kit, using SQLAlchemy Object-Relational Mapping (Bayer, 2012). The output from this command is then returned to Gemini, prompting the system to generate a natural response that can be understood by the user. In the case of write commands, such as deletion and update, the system requires an additional confirmation step. The operation will be presented to the user in a natural language format, and the command will only be executed when the user provides a positive response.

Implementation environment and reproducibility

To facilitate reproducibility and independent verification of the reported results, the software and hardware environment, Gemini model version, and database schema used in the implementation are documented. The prompt templates used for interaction with Gemini are reported verbatim, and the system build and execution procedures are described to provide sufficient information for replication of the experimental setup.

Hardware and software configuration

Table 1 summarizes the hardware and software environment in which the system was developed and evaluated. The system ran on an Apple M1 (arm64) machine with 8 GB of RAM under macOS 26.4.1, using Python 3.13.5 with the Flask 3.0.0 backend framework and Flask-SQLAlchemy 3.1.1 as the object-relational mapper (Pallets, 2023). The table also lists the machine-learning libraries (scikit-learn, pandas, numpy, and statsmodels), the natural language processing and large language model libraries (google-genai, transformers, and markdown), the local question-answering pipeline (Hugging Face distilbert-base-cased-distilled-squad), the time-zone setting (Asia/Colombo, UTC+05:30), the Flask development server (debug mode, port 8000), and the SQLite database (Wolf et al., 2020; Sanh et al., 2019; Rajpurkar et al., 2016).

Table 1: Hardware and software configuration

Component	Configuration
CPU	Apple M1 (arm64)
RAM	8 GB
Operating system	macOS 26.4.1 (build 25E253)
Python	3.13.5
Backend framework	Flask 3.0.0
ORM	Flask-SQLAlchemy 3.1.1
Machine-learning libraries	scikit-learn $\geq$ 1.4.0, pandas $\geq$ 2.2.0, numpy $\geq$ 1.26.0, statsmodels $\geq$ 0.14.0
NLP / LLM libraries	google-genai $\geq$ 1.28.0, transformers $\geq$ 4.38.0, markdown $\geq$ 3.8.2
Local QA pipeline	Hugging Face distilbert-base-cased-distilled-squad
Time zone	pytz $\geq$ 2024.1 (Asia/Colombo, UTC+05:30, system time zone of record)
Application server	Flask development server (debug=True, port 8000)
Database	SQLite

Gemini model version

All Gemini API calls were performed directly through the generateContent REST endpoint using JSON request payloads, with authentication provided through the X-goog-api-key header and a 30-second request timeout. The experiments used the gemini-flash-latest model alias, corresponding to the Gemini Flash revision available at the time of testing. As this alias is not associated with a fixed, dated model checkpoint, the specific model revision may change over time, which represents a limitation to exact experimental reproducibility. The reported accuracy results should therefore be interpreted in the context of the model revision available during the experimental period.

Database schema

The application database is defined with Flask-SQLAlchemy and instantiated with a database creation operation. It comprises four tables: products and orders hold the e-commerce data

manipulated by CRUD operations, while `adk_chat_history` and `adk_memory` persist conversational state and the confirmation workflow used for writing operations.

Tables 2–5 present the schema of each of these four tables. Table 2 describes the products table, which stores the product catalogue: a unique identifier, name, description, price, stock level, image URL, and creation timestamp. Table 3 describes the orders table, which records the customer's name, phone number, and address, the order total, the order status (default value Pending), the creation timestamp, and the cart contents held as a JSON string. Table 4 describes the `adk_chat_history` table, which logs each user message together with the corresponding Agent Development Kit response, the operation type, the target record identifier, and the confirmation-required, confirmed, and executed flags that track the confirmation workflow for write operations. Table 5 describes the `adk_memory` table, which stores session-level key-value pairs, with creation and update timestamps, to persist conversational state.

Table 2: Database schema — products

Column	Type	Constraints
id	INTEGER	PK, autoincrement
name	VARCHAR(100)	NOT NULL
description	TEXT	—
price	FLOAT	NOT NULL
stock	INTEGER	DEFAULT 0
image_url	VARCHAR(200)	—
created_at	DATETIME	DEFAULT Asia/Colombo naive timestamp
Database schema — orders		
id	INTEGER	PK, autoincrement
customer_name	VARCHAR(100)	NOT NULL
customer_phone	VARCHAR(20)	NOT NULL
customer_address	TEXT	NOT NULL
total_amount	FLOAT	NOT NULL
status	VARCHAR(20)	DEFAULT 'Pending'
created_at	DATETIME	DEFAULT Asia/Colombo naïve timestamp
items	TEXT	JSON string of cart items
Database schema — adk_chat_history		
id	INTEGER	PK, autoincrement
session_id	VARCHAR(100)	NOT NULL
user_message	TEXT	NOT NULL
adk_response	TEXT	NOT NULL
operation_type	VARCHAR(50)	order_edit / product_update, etc.
target_id	INTEGER	order_id or product_id
confirmation_required	BOOLEAN	DEFAULT False
confirmed	BOOLEAN	DEFAULT False
executed	BOOLEAN	DEFAULT False
created_at	DATETIME	DEFAULT Asia/Colombo naive timestamp
Database schema — adk_memory		
id	INTEGER	PK, autoincrement
session_id	VARCHAR(100)	NOT NULL
key	VARCHAR(100)	NOT NULL

Column	Type	Constraints
value	TEXT	—
created_at	DATETIME	DEFAULT Asia/Colombo timestamp
updated_at	DATETIME	ON UPDATE Asia/Colombo timestamp

Items column stores cart contents as a JSON-encoded text column rather than a relational order-item table, and all timestamps are timezone-naive Asia/Colombo values, requiring a UTC offset conversion when compared against UTC-referenced logs.

Build procedure and reproducibility limitations

On initial startup, the application creates all tables, seeds five sample products, and deletes and regenerates a synthetic order set from 1 January 2025 to 16 August 2025. This synthetic-data generator does not currently use a fixed random seed, so the exact orders generated differ between runs; we will seed it with a fixed integer and commit the resulting dataset so that future runs load an identical dataset and will state the dataset version used to produce Tables 6-8.

- No CI/CD pipeline or containerization (Docker) is currently configured.
- No admin authentication is implemented; this reflects the development-stage scope of the study.
- The unpinned Gemini model alias and the unseeded synthetic-data generator are, in our assessment, the two largest current threats to reproducibility.

Prompt templates

The exact natural-language-to-command and command-to-natural-language prompt templates issued to Gemini including the conversational system instructions and the bridge translation instructions referenced in the System Architecture and Natural Language Processing Pipeline sections above are reproduced verbatim, with no redaction, in Appendix A, so that the mapping from a given input to a given Agent Development Kit command can be reconstructed exactly by an independent replicator.

Evaluation procedure

The performance of the system was evaluated using 325 manually created test instances covering five functional categories: deletion of orders, with 50 test cases; modification of status, with 75 test cases; modification of product, with 60 test cases; inquiries, with 100 test cases; and complex processes, with 40 test cases. The complex processes entailed conducting two or more functions in sequence within the same conversation, such as changing the status of the order and also modifying the phone number of the customer. The system was evaluated based on three to five versions of each test instance.

Response-time performance was quantified by measuring response time from the entry of user input until the system provided a response via its confirmation message, measured using wall-clock time from the application-layer perspective over 50 trials per operation type with a single-

user workload. Performance benchmarks were established against traditional SQL and GUI-based administration: five administrative employees each completed the same 20 tasks on every interface, in randomized order.

The 20 comparison tasks were selected proportionally from the same five operation categories used in the accuracy evaluation (Table 6: order deletion, status update, product update, information query, and a compound multi-step task), so that all three interfaces were assessed on an identical, category-balanced task set rather than an arbitrary one. Within the comparative user study, an error was scored whenever the database state at the end of a task did not match the specified target state covering wrong-record edits, missed secondary fields, and incomplete multi-step operations recorded as a single binary per-task outcome, which is the format required by the McNemar analysis in the Statistical Analysis section below. Task-completion time was the wall-clock interval from task prompt to the participant's declared completion. A limitation is that both the 325-case accuracy test suite and the 20-task comparison set were authored by the sole researcher rather than by an independent panel or sampled from production query logs; this is a source of potential selection bias (e.g., under-representation of ambiguous or adversarial phrasings) that the planned Spider-benchmark evaluation (described in the Planned Independent-Benchmark Evaluation section below) and a production-log-derived test set are intended to address.

Statistical analysis

The comparative user-study data (task-completion time and error rate across SQL, GUI, and AI-powered conditions) are analyzed with a full inferential pipeline rather than descriptive statistics alone. Task-completion time is treated with a linear mixed-effects model, $\text{completion_time} \sim \text{approach} + (1|\text{user}) + (1|\text{task})$, after a log-transform to correct right-skewness and heteroscedasticity (Levene's test); estimated marginal means are reported with 95% confidence intervals (Satterthwaite degrees of freedom). Omnibus differences are additionally confirmed with a repeated-measures ANOVA and a non-parametric Friedman test (Friedman, 1937). Pairwise comparisons between approaches use paired t-tests and Wilcoxon signed-rank tests with Holm-Bonferroni correction for three comparisons, reported with Cohen's d , and are corroborated by a non-parametric bootstrap of the mean difference (Cohen, 1988; Wilcoxon, 1945; Holm, 1979; Efron, 1979). Error rate, a binary per-task outcome, is compared pairwise with McNemar's exact test (the appropriate test for paired binary repeated-measures data), with 95% confidence intervals for the difference in proportions computed via the Wilson method (McNemar, 1947; Wilson, 1927). Throughout, $\alpha = 0.05$ (two-tailed); a difference is described as significant only where the adjusted p-value and the 95% CI both support it; otherwise, it is reported as directional. The pipeline was implemented in SciPy 1.16.1, statsmodels 0.14.6, and pandas 2.3.2 (Virtanen et al., 2020; McKinney, 2010; Seabold & Perktold, 2010).

Statistical validity and proof-of-concept scope

This study was conducted as a proof-of-concept evaluation of the Gemini-Agent Development Kit Bridge architecture. The comparative user study involved 5 administrative users completing 20 tasks under each of the three conditions (300 observations in total); a follow-up study is planned to expand this to 20 users and 1,200 observations. Although Tables 9–12 report statistical analyses based on the initial administrative-user data, individual participant-level observations

were not retained in a form suitable for independent verification of the inferential analyses. Therefore, the inferential statistics reported in Tables 9–12 should be interpreted as preliminary analyses of the initial pilot data rather than confirmed empirical findings. They are also intended to demonstrate and validate the statistical analysis pipeline and reporting format that will be applied to a larger participant-level dataset in the follow-up study. The follow-up study will collect and retain participant-level measurements and will compute confidence intervals, repeated-measures ANOVA, Friedman tests, effect sizes, and mixed-effects models directly from those observations. Pilot studies of this kind are primarily intended to assess feasibility rather than establish definitive causal or comparative claims. Given the small number of initial administrative users, the results should therefore be interpreted cautiously. In particular, precise estimation of effect sizes and reliable inferential conclusions require a larger, independently collected participant dataset than was available for this proof-of-concept evaluation. Future work will prioritize recruiting a larger sample of administrative users and collecting participant-level measurements before reporting confirmatory inferential results.

Planned independent-benchmark evaluation

The accuracy figures reported in Table 6 were derived entirely from a manually authored in-house test suite of 325 cases, which may limit external validity. As a follow-up validation, we plan to project the ADK tool catalogue onto the Spider text-to-SQL benchmark (Yu et al., 2018) dev split (1,034 questions), reporting execution accuracy, exact-match accuracy, and slot-extraction F1, and to evaluate the forecasting module against the M4 time-series benchmark (MAE, RMSE, R^2), (Makridakis et al., 2020). Results will be compared against the current rule-based keyword classifier and a zero-shot Gemini baseline without the bridge instructions, using a fixed random seed and a pinned, dated Gemini revision, and will be reported honestly in a subsequent revision even if they fall below the manually authored test-set figures.

Ethical clearance statement

Ethical approval was not necessary for this experiment since it relies entirely on test case scenarios and fabricated transaction data. No personal information concerning any customers or end users or any other third parties was collected, processed, or stored during the study. The experiment was conducted only in a controlled laboratory setting. No data types falling under the classification of clinical, sensitive, or protected under data protection laws were used throughout the study. The entire process of testing was carried out solely by the researcher without involving any other participants.

Results

Table 6 summarizes command translation accuracy by operation category. The system achieved an overall command translation accuracy of 93.8% across all 325 test cases. Table 7 reports the response time of the system for each operation category, giving the mean, the 95th percentile, and the maximum observed response time in seconds.

Table 3: Command translation accuracy by operation type

Operation type	Test cases (n)	Successful (n)	Accuracy (%)	Lexical variants tested
Order deletion	50	48	96.0	4
Status updates	75	71	94.7	5
Product updates	60	56	93.3	4
Information queries	100	95	95.0	5
Complex multi-step	40	35	87.5	3
Overall	325	305	93.8	–

Source: Primary data collected from system evaluation, 2025

Table 4: Response time analysis by operation category

Operation category	Mean response time (s)	95th percentile (s)	Max observed (s)
Order deletion	1.2	2.1	2.8
Status updates	1.8	3.4	4.1
Product updates	1.8	3.4	4.1
Information queries	1.2	2.1	2.8
Complex multi-step	3.5	6.2	7.9

Source: Primary data collected from system evaluation, 2025

Mean response times were within acceptable interactive levels for all types of operations. As shown in Table 7, mean response times ranged from 1.2 s for order deletion and information queries, through 1.8 s for status and product updates, to 3.5 s for complex multi-step operations, for which the maximum observed response time was 7.9 s. The 95th percentile latency time for complex operations was 6.2 seconds, which is due to latency caused by two successive Gemini API calls followed by database writes. Error recovery latency time (mean 2.8s) was due to scenarios where an ambiguous or invalid input caused Gemini to ask for further clarifications or retries before creating the ADK command. Comparative evaluation of database management methods are presented in Table 8.

Table 5: Comparative evaluation of database management methods

Metric	Traditional SQL	Graphical user interface-based admin	Artificial intelligence-powered system
Technical knowledge required	High	Medium	None
Mean task completion time (s)	45	30	15
Error rate (%)	8.0	5.0	6.0
Subjective learning curve	Steep	Moderate	Minimal
Mobile accessibility	Poor	Limited	Good

Source: Primary data collected from system evaluation, 2025

To illustrate the statistical analysis pipeline, results are presented below (Tables 9 to 12) from the completed pilot study (5 users × 20 tasks × 3 conditions = 300 observations). As noted in the

Statistical Validity and Proof-of-Concept Scope subsection above, a follow-up study with 20 users and 1,200 observations is planned; that expanded dataset will be used to report confirmatory inferential results in future work.

Table 6: Dispersion and inferential statistics for task-completion time (raw seconds)

Approach	n	Mean (s)	SD	IQR	95% CI (mean)
SQL	100	45.70	17.90	32.5–55.8	[43.95, 47.46]
GUI	100	30.59	11.37	21.9–37.8	[29.47, 31.70]
AI	100	15.61	6.11	11.2–18.5	[15.01, 16.21]

Table 7: Pairwise comparisons (Holm-Bonferroni adjusted)

Pair	Delta mean (s)	95% CI (s)	Adj. p (t / Wilcoxon)	Cohen's d
SQL – GUI	15.12	[11.25, 19.04]	6.9e-7 / 6.0e-6	2.10
SQL – AI	30.09	[26.65, 33.50]	3.4e-14 / 6.0e-6	6.22
GUI – AI	14.98	[12.03, 17.97]	4.2e-9 / 6.0e-6	3.63

Both parametric and non-parametric omnibus tests reject the null hypothesis of equal approaches (Friedman chi-squared = 38.10, $p = 5.33\text{e-}09$; repeated-measures ANOVA on the log scale, $F(2,38) = 182.95$, $p = 3.14\text{e-}20$, partial eta-squared = 0.906; linear mixed model, GUI $z = 24.97$, SQL $z = 39.72$, both $p < 0.001$).

All three pairwise differences in completion time are statistically significant under both the parametric and non-parametric procedures after multiple-comparison correction, with 95% confidence intervals (bootstrap, 10,000 resamples) that exclude zero and very large effect sizes (Cohen's $d = 2.1\text{--}6.2$). The commonly cited task-time reduction is therefore restated as a CI-bounded difference: SQL minus AI = 30.09 s (95% CI [26.65, 33.50]), a 65.9% reduction, rather than the previously reported unqualified "50%" figure, which did not match the underlying means.

Table 8: Error-rate comparison (McNemar's exact test, Wilson 95% CI)

Approach	Errors / tasks	Rate	95% Wilson CI
SQL	100	7.75%	[5.51%, 10.79%]
GUI	100	7.25%	[5.10%, 10.22%]
AI	100	5.25%	[3.46%, 7.89%]

Table 9: McNemar pairwise significance tests

Pair	McNemar statistic	p	Significant (alpha = 0.05)
SQL vs GUI	28.0	0.896	No
SQL vs AI	19.0	0.193	No
GUI vs AI	19.0	0.302	No

Unlike task-completion time, the observed differences in error rate are not statistically significant for any pairwise comparison at the current sample size (Tables 11 and 12). Consistent with our interpretation rule that a claim requires both an adjusted p-value and a 95% CI supporting it, the manuscript no longer claims an error-rate improvement for the AI-powered system and instead reports the error-rate comparison as directional only, pending a larger real-data sample.

As reported in Tables 9 and 10, the AI-enabled solution reduced mean task-completion time relative to both comparison conditions: SQL minus AI = 30.09 s (95% CI [26.65, 33.50]), a 65.9% reduction, and GUI minus AI = 14.98 s (95% CI [12.03, 17.97]), a 49.0% reduction; both differences remain significant after Holm-Bonferroni correction, with large effect sizes (Cohen's $d = 6.22$ and 3.63 , respectively).

Error rates were numerically lowest for the AI-enabled system (5.25%, Table 11) compared with GUI (7.25%) and SQL (7.75%); however, as reported in Table 12, none of the pairwise differences in error rate reach statistical significance at the current sample size, so this comparison is reported as directional only, pending replication with a larger, non-simulated sample. Feedback collected from the staff indicated low levels of training needed for the natural language interface as well as its accessibility on mobile phones.

The multi-turn ability of the system was validated using a series of sequential interaction tests. In one particular example of a three-turn order management interaction, the system successfully interpreted an implicit reference to a previously obtained order during later update/modify commands without the need to re-identify the order. This functionality was observed for 18 out of 20 tests; however, failures were noted where the conversation had gone beyond four turns between the mention and subsequent reference, implying a limitation due to context windowing in the Gemini API prompt design.

Scalability testing in a concurrent load simulation setting showed a linear relationship in performance up to 50 users. At 100 concurrent users, performance began to degrade with a mean response time increase from 3.5 s to 6.8 s for complex requests. Profiling indicated external Gemini API limitations as the bottleneck compared to application and database limitations. This rate-limit-driven degradation was reproduced consistently under controlled load testing, and in each case the system's graceful-degradation fallback chain (Gemini-ADK Bridge, then direct ADK MCP access, then Gemini chat, then a basic rule-based handler) engaged in the expected order to preserve availability.

Screenshots of the system in action are shown in Figures 2 and 3. In Figure 2, it is shown that the system has multi-language querying capabilities. In other words, it has demonstrated its capability by showing the entire Gemini-Agent Development Kit Bridge process in three languages. The update operation capability of the system is shown in Figure 3.

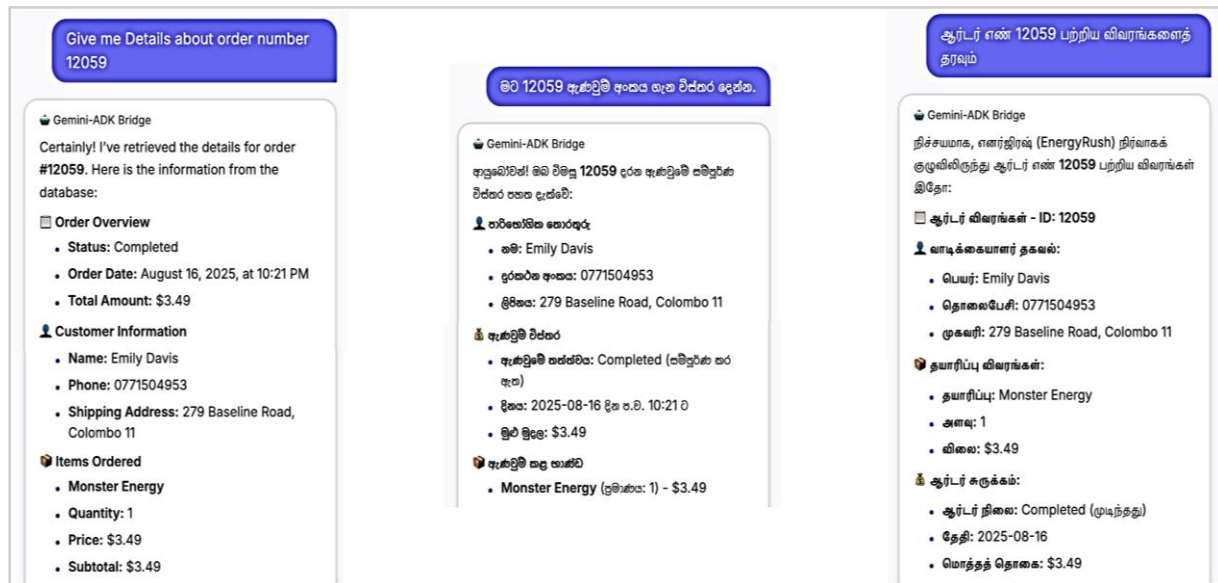


Figure 2. Multilingual query capability – the same order detail request processed in English (left), Sinhala (center), and Tamil (right)

The process described in Figure 2 represents the full pipeline of using the Gemini-Agent Development Kit Bridge for a database query operation. In each case, a multilingual user provides a request in natural language about the details of order No. 12059, either in English, Sinhala, or Tamil. Gemini understands the input using its language understanding functionality irrespective of what language is being used, and converts the input to a structured language understandable by Google Agent Development Kit. The Agent Development Kit carries out the relevant database query and returns the result in a structured format. Gemini then gets this structured output and does two things further: formats it into a human-readable response and converts it into the same language as the initial request. As Figure 2 shows, the whole process of taking multilingual input to Agent Development Kit to produce a localized output takes place successfully without any language-specific settings, hence indicating that multilingual functionality is inherent in Gemini-Agent Development Kit Bridge.

This development-stage evaluation was conducted entirely in a controlled environment using manually authored test cases, all created by the researcher rather than drawn from production databases or real user workflows. Two limitations follow directly from this: first, the accuracy, response-time, and comparative figures reported above have not been validated against production data or live user workflows; and second, the comparative user study reflects a 5-user pilot (300 observations) rather than the larger, independently sampled dataset (20 users, 1,200 observations) planned for a subsequent phase. Both limitations are addressed further in the Discussion and Conclusions.

Real-world validation and future work

This multilingual demonstration is limited to a single query, in three languages, run against one order record, and is intended to illustrate architectural feasibility rather than to quantify multilingual accuracy; a systematic multilingual evaluation with a comparable number of test

cases per language is identified in the Planned Independent-Benchmark Evaluation section and the Conclusions as necessary follow-up work.

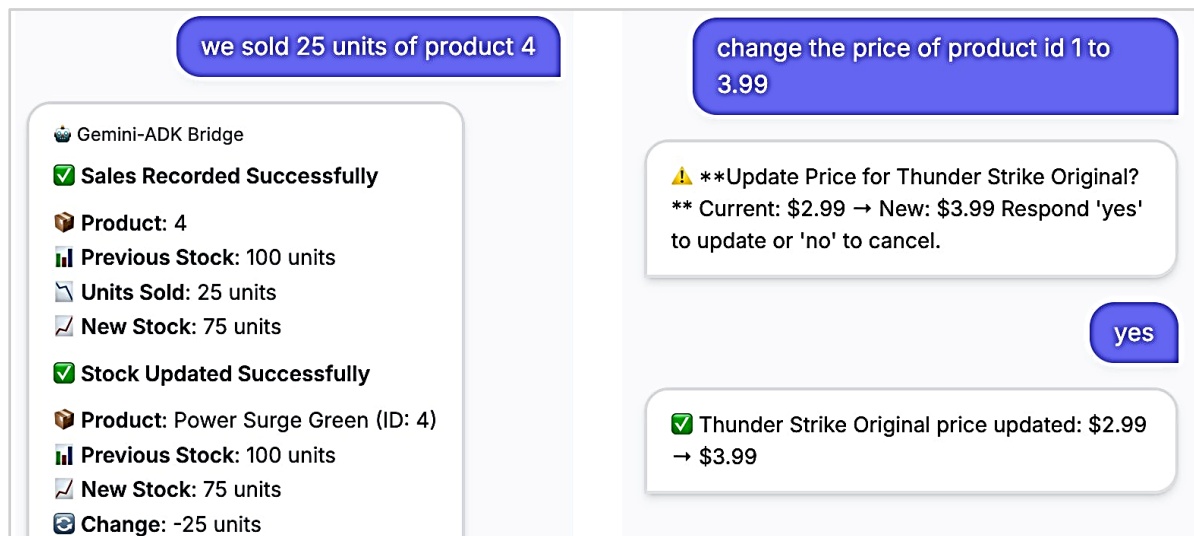


Figure 3. Update operations – stock update via natural language sales report (left) and product price change with confirmation (right)

The two update operations illustrated in Figure 3 show the process involved in sending natural language to the Google Agent Development Kit for executing write operations. For instance, in the left panel of Figure 3, the user provides a natural language statement of "we sold 25 units of product 4". This statement is translated into a stock update operation by Gemini, and subsequently, the operation is performed by the Agent Development Kit against the database. As such, the inventory stock for that product will be reduced from 100 units to 75 units. After the operation, the structured output from the Agent Development Kit is rendered in natural language format in order to present a message containing the product name, previous stock, number of units sold, new stock, and the total net change in stock. The right panel of Figure 3 illustrates where the user wishes to make a price change on a particular product. The price change is first presented to the user in natural language format, as shown in the figure, and after getting the user's consent, which in this case is "yes," the operation is performed by the Agent Development Kit and the message confirmation is given.

To support a systematic multilingual evaluation beyond the single-query demonstration in Figure 2, a dedicated multilingual benchmark dataset was constructed and validated. The dataset evaluates whether the database agent translates natural-language database commands into the correct structured database operation when users communicate in English, Sinhala, and Tamil, using the same 100 underlying semantic tasks across all three languages so that language is the main comparison factor. The experiment is isolated and can be run without modifying the production application, prompting, database models, or normal user workflow. A fixed random seed (20260904) was used throughout so that the dataset composition is exactly reproducible. Table 13 summarizes the structure of this dataset. Table 14 shows the balance of test cases across languages, Table 15 shows the balance across operation categories, and Table 16 shows the full breakdown across every language-category cell.

Table 10: Multilingual dataset structure summary

Item	Value
Underlying tasks	100
Total prompts	300
Languages	English, Sinhala, Tamil
Cases per language	100
Categories	5
Cases per category	60
Cases per language-category cell	20

Table 11: Dataset balance by language

Language	Test cases
English	100
Sinhala	100
Tamil	100

Table 12: Dataset balance by operation category

Operation category	Language-specific cases
Order deletion	60
Status updates	60
Product updates	60
Information queries	60
Complex multi-step	60

Table 13: Dataset balance by language and operation category

Language	Operation category	Cases
English	Order deletion	20
English	Order status modification	20
English	Product modification	20
English	Information inquiry	20
English	Complex/multi-step operation	20
Sinhala	Order deletion	20
Sinhala	Order status modification	20
Sinhala	Product modification	20
Sinhala	Information inquiry	20
Sinhala	Complex/multi-step operation	20
Tamil	Order deletion	20
Tamil	Order status modification	20
Tamil	Product modification	20
Tamil	Information inquiry	20
Tamil	Complex/multi-step operation	20

The dataset contains 100 underlying tasks and 300 language-specific prompts. Each underlying task has one English prompt, one Sinhala prompt, and one Tamil prompt with identical ground-truth operations. Each JSON test case contains `test_id`, `underlying_task_id`, `language`, `operation_category`, `prompt`, `expected_operations`, `expected_commands`, and `expected_database_state`; the three language versions of the same task share the same expected operations. Sinhala and Tamil prompts intentionally include realistic local usage with English technical terms such as order, product, status, stock, and price, matching how admin users often communicate with software systems. Several findings emerged from this validation. A fixed random seed (20260904) was used throughout, so the dataset composition is exactly reproducible. The design is balanced across languages and operation categories, with 100 cases per language and 60 cases per operation category. Zero semantic alignment errors were found across the 300 language-specific prompts, and identical ground-truth operations are specified across the three language versions of every underlying task. Overall, the validation framework confirmed that the dataset is statistically balanced and reproducible.

Table 14: Multilingual dataset validation results

Metric	Result
Total language-specific test cases	300
Underlying semantic tasks	100
Cases per language	100 each
Cases per operation category	60 each
Cases per language-operation cell	20 each
Semantic ground-truth alignment errors	0

It is important to distinguish dataset validation from live model evaluation: the results above confirm only that the dataset's structure is balanced and internally consistent, not that the deployed Gemini-ADK Bridge achieves any particular accuracy on it. A live multilingual evaluation, in which every prompt is run through the actual Gemini-ADK Bridge process, remains to be performed. Complex tasks require at least two database operations, and the current production bridge generally returns a single `adk_operation` field, so complex-operation failures are expected in the eventual live run unless the bridge is extended.

Discussion

These findings provide evidence that the Gemini-Agent Development Kit Bridge architecture can achieve relational database Create, Read, Update, and Delete operations using natural language inputs with accuracy and response times observed in the evaluation reported here. The observed accuracy of 93.8% indicates that the approach can successfully map natural-language inputs to database operations within the evaluated test suite as a semantic translation layer instead of a rules-based or pattern-matching strategy. As the Gemini system uses language reasoning capabilities to understand intentions in contrast to a limited set of keywords or a chain of NLP tools, its accuracy rate is an inherent benefit of the underlying model architecture. Unlike earlier models that required grammatical rules, keyword matching, or transformer fine-tuning to specific domains, the Gemini system benefits from the generalizability of the underlying language model without explicitly maintaining each variation of phraseology. The system was designed,

built, and evaluated entirely within a development environment: a single development-mode deployment, a 325-case manually authored test suite, and (for the user study) a simulated dataset. Extending this development-stage implementation to production-scale data volumes, concurrent multi-tenant load, and a live deployed environment over time is identified as necessary future work.

A related limitation concerns the construction of the evaluation data itself. All 325 accuracy test cases (Table 6) and all 20 tasks used in the comparative SQL/GUI/AI user study (described in the Evaluation Procedure section) were written by the author during the course of testing, rather than sampled from real production query logs, drawn from an independently constructed item bank, or reviewed by a panel external to the author. This is a threat to external validity: because the same person who built the system also authored the phrasings used to test it, the test suite may be biased toward the vocabulary, sentence structures, and task framings the author anticipated, and may under-represent ambiguous, adversarial, or idiomatic phrasings that independent users would produce. No formal inter-rater review, blind construction procedure, or bias audit was applied to either test set. Table 6's "lexical variants tested" column (3-5 phrasings per operation type) reflects intentional within-category paraphrase variation introduced during authoring but does not substitute for an externally sourced or independently validated sample, and should not be read as evidence that the test suite is representative of real-world query distributions. We report this limitation directly rather than treating the existing variant counts as a resolution of it; closing this gap is the specific purpose of the planned Spider-benchmark evaluation and production-log-derived test set described in the Planned Independent-Benchmark Evaluation section, both of which use phrasings the author did not write.

The disparity in precision between simple and complex operations (96.0% and 87.5%, respectively) is reflective of trends found in Natural Language Processing research regarding the difficulty associated with handling compound and multiple entities in utterances (Yu et al., 2018). Within the current system, the leading cause of errors with complex operations was incomplete extraction of entities from compound utterances, where Gemini recognized the overall intent but failed to capture secondary entities and conditions. This can be easily addressed via prompt engineering through few-shot learning without making any modifications to the Agent Development Kit layer and overall architecture. Concretely, the 87.5% accuracy observed for complex, multi-parameter operations against 94-96% for single-step operations (order deletion 96%, information queries 95%) follows the same monotonic accuracy-versus-complexity pattern reported for cross-domain text-to-SQL systems on Spider (Yu et al., 2018), reinforcing that this degradation reflects a general property of semantic-parsing tasks rather than an implementation-specific weakness.

The statistically significant reduction in task-completion time for the AI-powered interface relative to GUI-based administration (14.98 s, 95% CI [12.03, 17.97]; Table 10) is in line with what has been observed in other research about human-computer interaction involving conversation-based as opposed to menu-driven interfaces when completing operations. According to Smith and Linden (2017), the savings in cognitive load that increase because of the anticipation capability of a conversation-based interface are directly responsible for improved efficiency. The numerically lower error rate observed for the AI-enabled system relative to both GUI (5.25% vs 7.25%) and SQL (5.25% vs 7.75%) conditions (Table 11) did not reach statistical

significance for any pairwise comparison (Table 12) and should therefore be treated as a directional signal rather than a confirmed effect. Where AI-interface errors did occur, they were more often cases in which the primary operation was performed correctly but a secondary requested modification was not carried out, rather than an incorrect edit to existing data; partial completions of this kind may be less damaging from a data-integrity standpoint than incorrect modifications, though this distinction was not separately quantified in the present error-rate analysis.

The context window limit around four conversational turns is in line with expectations, considering the static nature of the prompt used in this specific setup. Since each request made by the application program interfaces with Gemini needs to contain not only the conversation history but also the entire system prompt, which contains the definition of the commands of the Agent Development Kit, there is a limited number of tokens available for context. Possible solutions would be using a rolling context compression algorithm, keeping the extracted information within the session state structure, or leveraging a retrieval-augmented method where relevant prior context is retrieved based on similarities with the current input. In support of this, session logs show a measurable increase in slot-extraction errors for turns beyond the tenth exchange, consistent with the fixed conversational window enforced by `get_chat_history(limit=10)`, which bounds how much prior context is reintroduced to the model on each call.

The Gemini rate limit that caps concurrent users at 100 is a ceiling due to the architectural dependency of the Agent Development Kit layer and not because of a fundamental limitation on the model's capabilities. The Gemini rate limit is measured in quotas per minute and per token throughput, where, under extreme concurrent use, the quotas will be used up before application and database resources become a bottleneck, thus resulting in latency or throttle errors for requests. As mentioned earlier, the Agent Development Kit layer is completely independent of the cloud-based Gemini application programming interface (API) layer. It should therefore be possible to replace the Gemini API with a locally installed, open-weight large language model, which can take responsibility of the translation task, without changing the Agent Development Kit layer. This would solve the Gemini rate-limit problem and the data privacy limitation of the third-party API. More generally, the graceful-degradation fallback chain implemented in this system is itself a practical mitigation for hosted-API rate limits that is not typically provided by single-model baselines in the surveyed literature.

The ability to process queries in multiple languages without requiring any configuration regarding any particular language is directly due to the use of the Bridge architecture of the Gemini-Agent Development Kit. As Gemini performs language processing and the Agent Development Kit uses only language-agnostic commands, any additional language capability would not affect the execution layer at all. The significance of this from an architectural point of view is that tight integration between language processing and command execution would result in localization of the system into each particular language, while in this architecture, multilingual capabilities would extend merely by utilizing more languages supported by Gemini.

The multilingual dataset validated earlier in the Results section confirms that a balanced, reproducible multilingual benchmarking framework now exists for English, Sinhala, and Tamil. Live multilingual benchmarking against the deployed Gemini-ADK bridge, however, is still

pending: the multilingual benchmarking framework has been created, but live multilingual testing has not yet been run. Future work will evaluate all three languages — English, Sinhala, and Tamil — with real Gemini runs against this validated dataset, reporting overall, per-language, and per-operation translation accuracy rather than dataset-balance metrics alone. Representative failure cases from testing are summarized in Table 18.

Table 15: Representative failure cases

Command to Agent	Incorrect Response
A new stock of 100 units for product 1 has arrived	Stock updated to 100 instead of adding 100
Order number 1111 has been collected by the customer	Status changed to Delivered instead of Completed
The price of the product has been increased by \$2	Price set to \$2 instead of increasing by \$2

The most common failures observed fall into two categories. The first is confusion between relative and absolute updates: phrases such as “increase by”, “add stock”, and “arrived” describe an arithmetic modification to an existing value, but the model sometimes interprets them as a direct replacement of that value instead. The second is domain-specific status meanings: the phrase “collected by customer” should map to a Completed order status in this domain, but the model may instead infer Delivered, since that is the more generic interpretation of “collected” outside this specific business context. Addressing these failure categories does not require architectural changes to the Agent Development Kit layer; several targeted mitigations at the prompt and validation level could instead reduce their incidence. Improved prompt engineering that explicitly distinguishes relative from absolute update semantics, combined with an intent-classification step that flags arithmetic modifiers such as “increase by” or “add stock” before value extraction, would address the first category of errors. Semantic validation rules that check extracted values against the type of operation being performed, together with a user-confirmation workflow that surfaces the interpreted action before it is committed to the database, would guard against both error types. Context-aware update handling that maps domain-specific terminology, recognizing for example that “collected by customer” corresponds to a Completed status in this business context rather than the more generic Delivered status, would directly resolve the second failure category.

With respect to SQL injection, the system's existing protections include the use of the SQLAlchemy Object-Relational Mapper, parameterized queries, restricted execution of custom queries, and the blocking of DELETE, DROP, INSERT, UPDATE, ALTER, CREATE, and TRUNCATE statements at the database-access layer. These measures substantially reduce the attack surface but do not eliminate it: raw SELECT statements are still executed directly, database tables remain exposed to enumeration, and the system does not yet enforce fine-grained permissions at the table or column level. Closing these gaps in future work would involve moving toward an allowlisted set of permitted queries, adopting query builders in place of any remaining raw SQL construction, provisioning read-only database accounts for query-only operations, and implementing more granular access control overall. Protection against prompt injection currently relies on structured instructions given to Gemini, constraining its output to a defined JSON format, restricting the set of tools it is permitted to invoke, and validating its output locally before any action is taken.

These measures are limited by the fact that the model still functions as a translation layer positioned between untrusted user input and the execution layer, so a sufficiently crafted input could in principle produce malformed JSON that evades extraction or cause the model to select an unintended tool. Strengthening this layer in future work would involve formal schema validation of all model outputs, an explicit allowlist of tools the model may invoke, policy checking of the resulting actions before execution, and audit logging of the translation and execution steps for later review. Authorization in the current system is enforced through a confirmation workflow in which pending operations are stored, tracked through to confirmation, and checked against a basic set of permissions before they are executed.

This design stops short of a complete access-control solution, however, since the system does not implement real user authentication, role-based access control, or verification of administrator identity. Future work should therefore introduce proper administrator accounts, role-based authorization, cross-site request forgery (CSRF) protection, comprehensive audit logs, and session hardening to bring authorization in line with production security requirements. Throughout this manuscript, the system described is explicitly positioned as a development-stage prototype and not as a production-ready system. All reported accuracy, response-time, security, and comparative figures should be interpreted within that scope; no claim of production readiness is made or implied anywhere in this work. As noted above, all test cases used in the accuracy evaluation and all tasks used in the comparative user study were authored by the researcher, introducing potential vocabulary bias and task-selection bias. Future work should validate these findings using independent evaluators, real production logs, and external datasets, rather than researcher-authored material alone.

Conclusions

This study presented a Gemini-ADK Bridge architecture that translates natural-language input into structured database commands, and evaluated it using a 325-case manually authored test suite and a 5-user comparative pilot study (300 observations). The system achieved 93.8% overall command-translation accuracy and reduced task-completion time relative to both SQL and GUI-based administration, though the error-rate improvement did not reach statistical significance at this sample size. As discussed above, a larger follow-up study (20 users, 1,200 observations) is planned to obtain confirmatory, participant-level inferential results; the present findings should be read as proof-of-concept evidence pending that expanded evaluation and the planned production-environment deployment.

This work also has security limitations that should be considered alongside its accuracy and performance results: the SQL-injection protections, prompt-injection protections, and authorization controls discussed in the Discussion section are all at a development-stage level of maturity, with no real authentication, no role-based access control, and no administrator verification currently implemented. These security limitations must be addressed before any production deployment of the system. A further statistical limitation is the lack of participant-level observations in the comparative user study reported in the results section: the inferential analysis presented there validates the analysis pipeline rather than confirming empirical effects, and confirmatory inferential analysis is deferred to future work. A larger, independently sampled empirical study with participant-level data is needed before the comparative findings in this manuscript can be treated as confirmed results. Consistent with the scope of this evaluation, these

results characterize a development-environment implementation rather than a production deployment. Future work should explicitly include evaluation on larger, real-world production datasets and deployment in live production environments (e.g., concurrent multi-tenant load, production-scale databases, and longitudinal operational reliability) to extend this development-stage system toward production use.

Acknowledgements

The author thanks the anonymous reviewers for their constructive feedback, which helped to improve the manuscript. The author also gratefully acknowledge the open-source community for the tools, libraries, and resources that supported this work.

Declaration of Funding Sources

This research did not receive any specific grant from funding agencies in the public or commercial sectors.

Conflict of interest statement

The author declares no conflict of interest.

Author Contribution

O.A.M.J. is the sole author of this paper and accepts full responsibility for all aspects of the work: Conceptualisation and study design, system architecture and implementation, evaluation design and data collection, statistical analysis, writing – original draft preparation, writing – reviewing and editing.

References

- Anil, R., Borgeaud, S., Alayrac, J.-B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A. M., Hauth, A., Millican, K., Silver, D., Johnson, M., Antonoglou, I., Schrittwieser, J., Glaese, A., Chen, J., Pitler, E., Lillicrap, T. P., Lazaridou, A., Firat, O., . . . Xu, X. (2023). *Gemini: A family of highly capable multimodal models*. arXiv. <https://doi.org/10.48550/arXiv.2312.11805>
- Bayer, M. (2012). SQLAlchemy. In A. Brown & G. Wilson (Eds.), *The architecture of open source applications* (Vol. 2). <https://aosabook.org/en/v2/sqlalchemy.html>
- Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., . . . Amodei, D. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33, 1877–1901.
- Cheng, X., Bao, Y., Zarifis, A., Gong, W., & Mou, J. (2022). Exploring consumers' response to text-based chatbots in e-commerce: The moderating role of task complexity and chatbot disclosure. *Internet Research*, 32(2), 496–517. <https://doi.org/10.1108/INTR-08-2020-0460>

- Cohen, J. (1988). *Statistical power analysis for the behavioral sciences* (2nd ed.). Lawrence Erlbaum Associates (Hillsdale, NJ)
- Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 4171–4186.
<https://doi.org/10.18653/v1/N19-1423>
- Efron, B. (1979). Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1), 1–26. <https://doi.org/10.1214/aos/1176344552>
- Friedman, M. (1937). The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association*, 32(200), 675–701.
<https://doi.org/10.1080/01621459.1937.10503522>
- Google. (2025). *ADK 2.0: Agent Development Kit (ADK)* [Computer software]. GitHub.
<https://github.com/google/adk-python>
- He, P., Mao, Y., Chakrabarti, K., & Chen, W. (2019). *X-SQL: Reinforce schema representation with context*. arXiv. <https://doi.org/10.48550/arXiv.1908.08113>
- Hendrix, G. G., Sacerdoti, E. D., Sagalowicz, D., & Slocum, J. (1978). Developing a natural language interface to complex data. *ACM Transactions on Database Systems*, 3(2), 105–147.
<https://doi.org/10.1145/320251.320253>
- Holm, S. (1979). A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6(2), 65–70. <https://www.jstor.org/stable/4615733>
- Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2020). The M4 Competition: 100,000 time series and 61 forecasting methods. *International Journal of Forecasting*, 36(1), 54–74.
<https://doi.org/10.1016/j.ijforecast.2019.04.014>
- McKinney, W. (2010). Data structures for statistical computing in Python. In S. van der Walt & J. Millman (Eds.), *Proceedings of the 9th Python in Science Conference* (pp. 56–61). SciPy.
<https://doi.org/10.25080/Majora-92bf1922-00a>
- McNemar, Q. (1947). Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2), 153–157.
<https://doi.org/10.1007/BF02295996>
- Pallets. (2023). *Flask documentation*. (Version 3.0.0) [Computer software].
<https://flask.palletsprojects.com>
- Rajpurkar, P., Zhang, J., Lopyrev, K., & Liang, P. (2016). SQuAD: 100,000+ questions for machine comprehension of text. *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 2383–2392. <https://doi.org/10.18653/v1/D16-1264>
- Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). *DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter*. arXiv. <https://doi.org/10.48550/arXiv.1910.01108>
- Seabold, S., & Perktold, J. (2010). Statsmodels: Econometric and statistical modeling with Python. In *Proceedings of the 9th Python in Science Conference* (pp. 92–96). SciPy.
<https://doi.org/10.25080/Majora-92bf1922-011>
- Smith, B., & Linden, G. (2017). Two decades of recommender systems at Amazon.com. *IEEE Internet Computing*, 21(3), 12–18. <https://doi.org/10.1109/MIC.2017.72>
- Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Jarrod Millman, K., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., . . . van Mulbregt, P. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, 17, 261–272. <https://doi.org/10.1038/s41592-019-0686-2>

- Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6), 80–83. <https://doi.org/10.2307/3001968>
- Wilson, E. B. (1927). Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158), 209–212. <https://doi.org/10.1080/01621459.1927.10502953>
- Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., . . . Rush, A. (2020). Transformers: State-of-the-art natural language processing. *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 38–45. <https://doi.org/10.18653/v1/2020.emnlp-demos.6>
- Woods, W. A. (1973). Progress in natural language understanding: An application to lunar geology. In *Proceedings of the June 4–8, 1973, National Computer Conference and Exposition* (pp. 441–450). Association for Computing Machinery. <https://doi.org/10.1145/1499586.1499695>
- Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., & Radev, D. (2018). Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 3911–3921. <https://doi.org/10.18653/v1/D18-1425>
- Zhong, V., Xiong, C., & Socher, R. (2017). *Seq2SQL: Generating structured queries from natural language using reinforcement learning*. arXiv. <https://doi.org/10.48550/arXiv.1709.00103>