iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** V **Month of publication:** May 2026

DOI: https://doi.org/10.22214/ijraset.2026.80869

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue V May 2026- Available at www.ijraset.com

SynaptiQ: Building a Local-First Conversational Analytics Platform with NL-to-SQL, Forecasting, and Offline Voice Support

Swetha P M[1], Likhith C P[2], Tanish S[3], Ganga L[4], Suman M N[5]
Department of Computer Science & Engineering, JSS Science and Technology University, Mysuru, Karnataka,India

Abstract: ***Here's what happened. Thirteen steps shaped this thing called SynaptiQ, built during our last year as engineers. It started because we got tired of seeing folks stuck with data in spreadsheets or CSVs - data they could not explore unless they learned SQL or handed everything over to an online service. That felt wrong. Our version works only on your machine. Nothing leaves your device. Questions in everyday language go in. A model on your own device turns them into code that talks to a built-in database. The result comes back with a visual graph attached. Data gets checked automatically when it arrives. Predictions over time happen through methods like SARIMAX along with alternatives. You can test changes to see possible outcomes. Voice works both ways if you choose it. All of it stays inside the computer. At point M12, forty-eight checks were cleared. Real data worked smoothly. No link to the web was needed. Labels used: talking to databases with words, offline reasoning engines, compact analytical storage, keeping insights private, trend guessing math, exploring change effects, safe query rules, chat-style number crunching, self-hosted intelligence---runners, memory-backed-search-tools.***

I. INTRODUCTION

Most folks dealing with data aren't engineers. Doctors, sales leads, shipping planners - they handle number-heavy spreadsheets daily yet struggle to pull insights. Querying remains out of reach without coding skills. Tools today demand expertise in SQL or force file uploads to outside servers. That becomes risky if information must stay private. Usability bumps into security, then stumbles forward alone. A space forms behind it - quiet, ignored.

A shift seemed worthwhile trying. The core idea: operate offline, no secrets allowed. With transparency fixed - every SQL line visible, every graph decision unpacked, uncertainty voiced during handoffs - the journey gained weight equal to its outcome. Seeing how things moved step by step held space beside the final answer.

This story begins with structure before parts appear one at a time. Following that, pieces took form on their own. Rather than spreading wide, attention narrowed to keep questions protected. Steps in testing moved forward only when actual use pointed the way. Now things move without shaking in certain spots - though a few still catch on corners. Progress showed up thirteen times, each one leaving a mark. Each step brought picks that got measured instead of rushed. What made sense then stayed near the center, never drifting far.

II. MOTIVATION AND PROBLEM DEFINITION

A. Why This Problem Matters

Peeking into modern chat-driven analytics, our test focused on what happens when a document gets uploaded. Almost every time, that file heads directly to a web-connected system. Medical centers tracking patient outcomes, law offices examining billing records, universities managing registration paperwork - sharing any of this outside isn't really an option. Legal boundaries such as GDPR, compliance needs like HIPAA, or strict company policies tend to shut down external transfers entirely. Yet these are exactly the organizations that gain the most by tapping into their own information independently.

What about trust? Hard to pin down, still matters plenty during daily use. Here pops a reply - neat, self-assured - but without seeing its steps, you cannot tell whether it is right. At the start of testing, the system generated SQL statements looking solid on arrival, though problems showed up later: joins linked wrong, filters in odd spots. Mistakes survive in silence when logic stays out of sight. Opening the door so users could inspect the query, adjust pieces, run another version - that tiny shift carried weight, mattered far more than anyone thought.

B. Design Goals

Four goals were chosen fast. Straight off, running directly from your computer led the way - model, data handling, search structure, all packed in. Rather than rely on phrasing to stop dangerous scripts, safety checks got woven into how queries become actions. Earlier tests proved it: banking on word choice for protection rarely works out. Step by step, every decision the program makes unfolds clearly - clarity was key. Running smoothly on a regular Windows machine mattered most, no high-end gear needed
For our target audience, Linux systems are common. That means the impact stretches past a single configuration.

C. What We Chose Not To Create

From day one, modifying data through SQL had zero approval - inserting, updating, changing records simply wasn't allowed. Putting such power in a language model seemed far too shaky a move. Advanced reading of documents, going past plain text scanning, didn't make the cut at first. Later on, bits of document work crept into the system.

III. RELATED WORK

A. Text-to-SQL Research

Qin and team kick things off - possibly useful should you step into this space. Their work walks through the checks applied, paths explored, yet also spots commonly tripping people up. A few of those snags showed themselves while we put ours together, particularly once table formats drifted apart or made-up header names slipped in. Solutions such as the semantics-driven screen and G4 label confirmation emerged simply by catching these bumps.
An early attempt to highlight comes from Seq2SQL [2], when Zhong's group stressed judging NL-to-SQL systems on correct answers rather than merely valid code form. For this reason, we test results by executing queries in DuckDB, going beyond shape inspection alone. Tied into that idea, SQLNet [3] brought forward sketch-based generation - a technique avoiding incremental token production. Skipping linear generation helps dodge mistakes tied to token ordering, explaining why cleaning up code after creation beats forcing strict rules during drafting.
A study using Spider revealed weak performance when applying Text-to-SQL systems beyond their training domain. Because of this, schema-bound prompts were developed - tying inputs directly to actual table structures cuts down mistakes. Work from RAT-SQL found extra gains by mapping links between database elements. What we call the semantic layer takes inspiration from that idea, though without full relationship tracking for now; even so, the design allows such features later. What makes our method different? It ensures SQL correctness after the model generates output, not before. While PICARD [6] builds safety into beam search - a clever trick - it needs control over how predictions are made. Ours steps in afterward, shaping structure once the response exists. That flexibility lets it pair with any large language model, no special setup needed.

B. Speech and Forecasting

Speech gets turned into text using a system built on Radford and team's Whisper model [7]. Faster-whisper runs it - swapping in CTranslate2 so speed stays high without needing a GPU. For predictions, SARIMAX from Statsmodels [8] does most of the lifting. Past results get checked through splits made by TimeSeriesSplit, pulled from scikit-learn. Testing databases showed DuckDB [9] worked better than others like SQLite or in-memory setups. Its design uses columns instead of rows, plus vector tricks, which helped queries fly when groupings piled up. The idea to favor local storage first? That comes from findings by Kleppmann and colleagues [10].

IV. ARCHITECTURE

A. Overall Structure

Five levels shape the code structure, each relying solely on those below. At the base sits core/ - handling config setup, organized logs, custom errors, time tools, along with an Ollama-focused HTTP connector. This part stays blind to data sets or analysis logic. One step higher lies data_layer/, managing file operations from start to finish: breaking down inputs, refining content, directing where things get stored, while refreshing the dataset index whenever needed.
Five analytics modules show up next: ask/, forecast/, insights/, visualization/, along with whatif/. Each pulls code from core/ and data_layer/, yet stays clear of touching other modules. Code checks run using Python's AST parser, scanning each file under modules/ to catch unwanted links between them. A slip-up triggers a broken build, pinpointing the offending file and line without delay. Surprisingly helpful once the code grew. Over top of those modules sits an api/ layer, slim and focused on HTTP stuff - breaking down requests, turning responses into data streams.

Communication from the React frontend reaches just the api/ layer. That boundary handles every exchange. Nothing bypasses it. The connection stays limited, always routed through that point. Every request goes there first.
frontend → api → modules → data_layer → core

B. Formal Representation

To cover everything, think of the system as a set of five parts: Π = { UI, ORCH, AGENTS, SERVICES, DB }. The first piece, UI, handles how things look - either Streamlit or React, based on setup. Next comes ORCH, which guides flow and sorts user goals. Then there are AGENTS, made up of Semantic plus others you might add later. SERVICES do specific jobs behind the scenes, working when called. Last is DB - the storage spot that keeps data ready for use. Each part connects so the whole thing runs
NL2SQL Guardrails Visualization Insights Forecast WhatIf Voice
Ingest LLM Storage SchemaUtils VectorStore with DuckDB.

C. How a Query Flows

A normal query using everyday language moves like this inside the system:
Orchestrator Translates Natural Language to SQL Queries Using Large Language Models with Guardrails and Executes in DuckDB Visualize Interface Here's the rule: communication with DuckDB must never happen straight from an agent. Each SQL query needs to pass through the guardrail layer before moving forward. Only then does it proceed. Nothing slips past without inspection. That barrier stays firm at all times. This was built into the structure, rather than simply agreed upon.

V. HOW THE SYSTEM WORKS END TO END

A. File Upload

Right off the start, hashing kicks in once someone sends a file - xxhash 64-bit grabs the raw bytes. That fingerprint gets looked up in the registry straight away. Found? Then out comes the stored dataset, nothing more. Repeating steps skip entirely, extra copies never form. We slipped this tweak in later, truth be told. Early tests had folks tossing up identical files twice, then scratching heads over which version was what. Now it just clicks without fuss.
A fresh file's ending decides what tool reads it. When it is a CSV, chardet checks the text format up front - too many showed up as latin-1 or windows-1252 when expected as UTF-8. Openpyxl takes charge of Excel sheets. Pyarrow steps in once Parquet appears. Pdfplumber opens PDF documents. Pictures pass through Pillow before Tesseract pulls out any words.

B. Data Cleaning

Most times, cleaning just means getting data ready to work with, nothing more. Lowercase letters replace uppercase ones in column titles, then extra bits go away. When numbers hide behind dollar signs or commas inside text, they shift into proper decimal form instead. Parsing dates feels like guessing - unless most entries fit, the whole thing stays messy. If over four out of five date-like values make sense after testing various formats, only then does the system accept it as actual time data. Most times, if something feels unclear, the system checks back with the person instead of guessing on its own. Instead of vanishing without a trace, repeated entries show up in totals and stay put unless told otherwise. Turns out, leaving them alone worked well since some trial data actually needed those copies sitting exactly where they were.

C. Storage

A single .db file lands in DuckDB for each dataset, neatly tucked inside the local workspace folder. From PDFs, Word docs, or scanned pages turned digital through OCR, raw text emerges - split roughly every 500 tokens, sliding forward by 45 each time. That fragmented content slips into vectors using nomic-embed-text powered by Ollama, then settles within ChromaDB. Certain files blur boundaries: a yearly corporate report may carry crisp financial grids alongside descriptive write-ups. These hybrids feed both systems simultaneously. When questions arise, the routing logic quietly decides where to look based on what is asked.

D. Natural Language to SQL

Looking at the question, the intent classifier sorts it into one of five types: nl2sql, forecast, whatif, insight, or help. When it lands on nl2sql, things shift - the NL2SQL agent pieces together a prompt using the dataset's schema DDL. Only SELECT statements are permitted - this rule shows up clearly inside that prompt. A LIMIT instruction also gets added so large data pulls won't happen by accident. If the system uses a local vector store, it slips in a handful of earlier question-SQL matches as guidance. From there, the local LLM crafts SQL based on what it sees. Afterward, safeguards step in quietly.

E. Forecasting

Inside the form, or maybe a written request, that is where guessing ahead begins. Key details get pulled by something watching for patterns - what needs forecasting, timing, range, confidence level. First thing first: check if the date column actually works like real dates do; many look okay but fall apart when tested. The rhythm of data points? That adjusts as we go. One guess rides the flat line others chase patterns in loops. A quiet trend watcher tags along while Theta splits time into parts. The heavy machine hums with self-fed logic none ignore it. All aim at data held back cold until now. Only one stands after the test. From that front runner a forecast emerges wearing twin masks hope first then warning. Saved where space allows they return fast next time round.

F. What-If Analysis

This module has no LLM involvement at all — it is just maths. The user picks a metric, optionally a grouping column, and defines some percentage scenarios (say, Conservative at minus ten percent, Optimistic at plus twenty). We compute the actual baseline from the data, apply the multipliers, and render all the scenarios and the baseline on the same chart with a side-by-side table. Users can save configurations as JSON and reload them later. This turned out to be one of the features people found most immediately useful.

G. Voice Mode

Most folks skip the voice stuff altogether - it runs without needing the internet. When turned on, sound travels through WebRTC, then gets changed into words using faster-whisper; behind that sits a leaner version of Whisper tuned with CTranslate2, slipping quietly onto the CPU using int8 math if no graphics chip is around. That written result shows up in the message box, waiting for you to look it over first. Replies come back as WAV audio, built by pyttsx3 piece by piece. Not flawless - struggles pop up when names or tech phrases roll in - but handles regular talk just fine.

VI. IMPLEMENTATION DETAILS

A. Startup and Session Management

Straight away, the app pulls in a YAML config file. At that point, it blends settings from environment variables marked with SQ_, using double underscores to show deeper layers. Stored chat logs remain present, alongside pieces of audio, feature switches, and placeholder predictions for upcoming inputs. Each time a file arrives, the system computes a hash of its data. If a fingerprint changes - showing edits were made - then old session data disappears. When ingestion trips during processing, messy remnants are cleared rather than left hanging in the system.

B. SQL Safety — Four Checkpoints

What changed over time? Our approach to the guardrail system. Starting out, just a prompt handled bad SQL blocking - worked okay until it did not. After that surprise, we built four distinct layers instead. Each one stands on its own . Each runs by itself
Right away, rejection happens if any of these show up: DROP, ALTER, INSERT, UPDATE, DELETE, CREATE, TRUNCATE, GRANT, REVOKE - seen in the SQL? Blocked without delay. Zero tolerance kicks in on sight of those terms. Not one passes through when spotted inside a query. The filter stops everything cold at detection. Starts with G2 - blocks queries missing a limit. When LIMIT isn’t present, it gets inserted automatically. That way, huge outputs like millions of rows won’t load by mistake. Wrong syntax? G3 spots it fast. Sqlglot takes the SQL, reads it like DuckDB would. If something feels off, it stops right there - no second chances. The engine never even sees broken code. Parsing fails early, quietly blocking what doesn’t fit. Clean format is a must, every single time. Nothing messy slips through by accident. Each query gets checked down to the smallest detail. Only perfect matches move forward. Everything else drops out of line. Precision rules here, silently enforcing order behind the scenes.
Fourth step: validating identifiers. From the parsed AST, each table and column name emerges, then faces comparison with the schema DDL of the dataset. Names made up out of thin air - more common than first assumed - trigger detection at this stage. An unambiguous error follows. Precision takes shape through elimination.

C. Semantic Layer

One way to spot a column's job? Look at how many unique values it holds. Numbers meant to be summed go here - they're measures. Categories used to slice data count as dimensions instead. Time columns stand apart, marked by their sequence. This guesswork relies on counting distinct entries per field. Out comes something called a SemanticLayer object. That thing builds short database definitions for prompts. When predicting or testing scenarios, the system picks the first number column if left unchecked. Most times, that choice lines up just fine.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue V May 2026- Available at www.ijraset.com

D. Insights Engine

Start with any set of data, run Insights, and automation kicks in across five steps. Each column gets its own close look at first - numbers show stats, categories reveal how often values appear, dates expose their span. Following that, standout measures emerge; when dates exist, shifts between periods get calculated. Correlations rise into view next. Unusual highs or lows pop up during step four, especially those outside normal bounds, along with sharp jumps over time. Ending things off, the LLM takes in every result and puts together a brief summary using everyday words. When dealing with big data - more than one hundred thousand entries - we swap pandas for polars during analysis since it runs much faster.

E. Chart Selection

Instead of making people choose how their data looks, the system checks what kind of columns are there along with clues from the wording. When it sees dates matched with values, out comes a line graph. A grouping label paired with amounts leads to bars. If two numeric fields show up, dots appear scattered across axes. Lots of figures at once bring up colored grids instead. If the outcome includes over 5,000 rows, it gets trimmed in a fixed way - then users are informed that their chart displays just part of the data. Most queries fall under these guidelines, roughly 90 percent actually.

VII. TECHNOLOGY STACK

Back at the start, every dependency got locked down tight. That move cut hours off troubleshooting later. Matching results between laptops and the build server became straightforward since code linked to identical libraries each time. Our final picks show up in Table I.

Category	Package / Component	Version
Runtime & UI	Python 3.11+ / FastAPI	—
	Streamlit	1.40.2
	Plotly	5.24.1
Data & Query	pandas / polars	2.2.x
	DuckDB	1.1.3
	sqlglot	25.31.4
Modelling	statsmodels	0.14.2
	scikit-learn	—
	numpy / scipy	1.26.4
LLM (Local)	Ollama client	0.3.3
	qwen2.5 / llama3	latest
	nomic-embed-text	—
Voice	faster-whisper	1.0.3
	pyttsx3	2.90
Vector Store	ChromaDB	—
Orchestration	LangGraph + LangChain	—
Frontend	React 18 + TypeScript / Vite	5.4

TABLE I: Pinned Dependency Stack

VIII. SAFETY, PRIVACY, AND RELIABILITY

A. Why Four Checkpoints

At first, SynaptiQ simply instructed the model to generate only SELECT statements. It held up - until it failed, producing a DROP command despite the rule. One slip during testing was enough to abandon reliance on prompts. Now, every response passes through four checkpoints, treating the LLM like a guest who can't touch anything. Structure checks happen first, using AST analysis paired with strict name validation against known schema elements. This approach mirrors parts of Scholak et al.'s method in [6], though no changes are made to the model itself or how it runs. Trust comes from process, not promises.

B. Data Privacy

Nothing reaches the Ollama daemon except the database structure and the question typed by the person. Row contents stay put, never sent out. Inside a folder named .synaptiq, right where work happens, everything rests - data files, prediction notes, saved analyses, scenario templates, and past queries. This setup follows what Kleppmann and team call local-first computing - not just in name. Ownership stays clear. Function runs full strength even when disconnected.

C. Error Handling

Errors never show up alone. Each one brings along a clear message someone can grasp without guessing what it means. Behind the scenes, there is also something more precise tucked into the logs for debugging later. Alongside sits a set of possible fixes anyone might attempt on their own. Watching testers struggle through unfiltered Python tracebacks made us rethink everything. If those confused even us, regular people stood no chance at all. Mistakes tied to time columns now come with extra hints simply because figuring out what went wrong feels like solving riddles most days.

IX. TESTING

A. Our Testing Approach

Early in the process, we made clear that safety checks would come prior to any other step. When guardrails fail, outcomes from later stages lose their value entirely. Following tests rely on artificial inputs - crafted by us - to probe boundary scenarios instead of using actual documents. Actual records shift over time; our built sets stay fixed. Slower iteration emerges as a result, yet trust in what CI reports grows stronger.

B. Test Modules

Inside test_sql_safety.py, things unfold step by step - the first stop is banned keywords. Out come the prohibited terms, lined up one by one. Then query restrictions kick in, tacked on without warning. Structure gets examined next, piece by jagged piece, like tracing frayed cords. Following that, scrutiny shifts to naming: tables, columns, everything checked against set forms. Altogether four stages stand guard, each ready to block questionable code midstep.

One file handles time series predictions using a specific modeling technique. It splits data into chunks to predict each part separately. Past performance checks are saved alongside results. Information about the run sticks around after execution finishes.

Looking at test_insights.py reveals how core performance numbers behave. Where things stand now gives clues about where they're headed. A closer look at what matters most comes through ordered segments that highlight influence levels.

Here things get tested under changing conditions. Through this tool, predictions form based on shifting inputs. Each time, it checks if multiplying values leads to accurate outcomes. What you set before turns into JSON when saved. Pulling up old records functions just as well.

Checking confidence in path decisions happens here through test_intent.py. Unclear queries get handled within the same process as well.

From test_nl2sql dot py it begins. This part shapes the way prompts are formed. Schema DDL slides in without disruption. Structure appears exactly at the required spot.

Out of the box, test_vectorstore dot py takes care of grabbing and sorting outcomes. When information comes back, sequence plays a role. One after another, repeated items vanish in the flow. Clearing everything? That option stays open at any point.

•test_semantic.py - measure, dimension, and time column role inference.

Looking at imports inside test_voice_imports dot py. Here, the system turns to CPU when needed. Making sure only the right components come in.

C. Status at M12

Some fifty small checks pass once M12 stops. Besides those, any additional ones added to M13 go through without trouble. Moving along the CI path involves targeting Python 3.12 and holding clean code with help from linters. Progress happens after safety scans give approval - nothing proceeds before that.

X. MILESTONE SUMMARY

One after another, the tests in the 48-part sequence passed cleanly. When stress hit, SQL safety checks held firm anyway. Module borders - those snap shut on their own now. Proof came straight from trial runs. Each step showed function.

Still, there's more to do. Try thinking about retry loops, feeding embeddings in chunks, getting into logs without slipping up. What's here today doesn't wobble. Core parts run clean. Information sticks near. Only you hold it. It shows right away what happens inside. Nothing gets shipped far away to another device.

TABLE II: Milestone Summary (M1–M13)

MS	What We Built
M1	Got the basic app working: file uploads, NL-to-SQL, simple charts, SQL edit/rerun.
M2	Added semantic layer and auto-insights — KPIs, histograms, correlation heatmap.
M3	First forecasting version using SARIMAX with seasonal configuration.
M4	What-if engine — lets users define scenarios and see projected outcomes.
M5	Voice input and output via faster-whisper and pyttsx3 over WebRTC.
M6	Security pass: stronger guardrails, result caching, sampling transparency.
M7	Smarter intent routing, better chart recommendations, Pareto analysis.
M8	Packaging and CI: pyproject.toml, CLI entry point, automated tests.
M9	Multi-scenario what-if, preset save/load, export to CSV and PNG.
M10	More forecast models, segmented forecasting, rolling-origin backtesting.
M11	Local vector store for query memory and similar-question retrieval.
M12	Polish pass — UI consistency, loading spinners, tab state fixes.
M13	Naive baseline model added; NL-to-forecast bridge; ongoing chat improvements.

XI. LIMITATIONS AND WHAT COMES NEXT

A. What Worked Well

Picking DuckDB felt right. Instant startup every time; no waiting around. That 100 MB CSV file? Shrinks down - fits snug in a 20 to 40 MB .db package. Counting queries snap back faster than you can blink. The Ask module flows smoother now thanks to LangGraph. Clear state types appear where tangled function layers used to live. After that, the checkpoint system appeared. Four verifications snapped into place each cycle. Each trial pointed to a single result - never once did an instruction meant to change or wipe information reach DuckDB.

B. Where We Fell Short

Qwen2.5 manages basic summaries just fine, although it falters when faced with complex window logic or multiple table joins - especially if column labels lack clarity. No backup try exists right now; once bad SQL trips a safeguard or breaks in DuckDB, the mistake lands straight in front of the user, untouched. Processing each chunk one after another means feeding a 200-page PDF split into 400 pieces demands exactly 400 individual requests to Ollama. Speed suffers because of this. Security? Missing altogether - not even a login check guards port 8000, so every file on the device sits exposed to whoever finds the address, acceptable only if used alone locally, risky beyond that..

C. Planned Next Steps

Again. If it breaks, go back to step one. The SQL builder might work on second attempt - perhaps third - adding each mistake as you loop through. Small glitches often disappear like that.

Altogether, chunks that travel in groups tend to flow more smoothly. When pieces aren’t dealt with one by one, but arrive bunched up, delays shrink. With multiple units advancing in a single push, file intake gains pace. Slower when solo, they gain rhythm packed tight.

One system serving multiple people can stay safe when personal info links strictly to individual logins. Because access runs through protected tokens, a person views just their own details. Separation happens naturally, no bulky tools needed. Each account remains distinct thanks to how permissions are handled behind the scenes.

Sometimes lots of requests hit at the same time. With httpx.AsyncClient, things move together instead of one after another. The update shifts how the Ollama client behaves. That shift means more room to manage several trips out and back without pausing each time.

Midnight thoughts sometimes reshape morning plans. When holidays twist trends, this picks up the change without prompting. Inside regular forecasts, it runs like a quiet observer - no extra moves needed. Riding on Prophet’s backbone, it slips into what you already do.

Every new beginning might let others get straight to doing. Starting everything with just one line shifts how fast a person can join.

XII. DISCUSSION

A. How Does It Compare to Cloud BI Tools?

Truth be told, SynaptiQ isn’t built to match Tableau Cloud or Power BI when it comes to teamwork tools or sleek dashboards. Big teams support those platforms; they’ve had years to refine things. Yet here’s what sets us apart: your data stays put, locked to your device by design - no detours, nowhere else to go. For healthcare, legal, defence, and research settings where data residency actually matters, we think that is a meaningful difference.

B. Model Quality Tradeoffs

The system's output is only as good as the model the user has pulled. qwen2.5:latest is the default and it works well for most queries. Users who need better results on complex schemas can swap in a larger model — llama3:70b, Mistral, CodeLlama — at the cost of more RAM and slower responses. The Ollama integration is model-agnostic, so swapping is just a config change.

C. Scale Limits

DuckDB handles large datasets well once they are loaded, but the loading itself is memory-intensive because we read the whole file into pandas before registering it. A 1 GB CSV can peak at 3 or 4 GB of RAM during ingestion. For most local use cases this is fine, but it is something we would need to address before handling very large files. DuckDB's native CSV and Parquet scanning would let us defer materialisation until query time.

XIII. RESULTS

A. SQL Safety Validation

All 48 automated tests passed cleanly at milestone M12, with zero regressions introduced across M13. The four-checkpoint guardrail pipeline - keyword blocking (G1), limit enforcement (G2), syntax validation via sqlglot (G3), and schema-bound identifier checking (G4) - collectively rejected 100% of injected destructive queries during stress testing. No DROP, DELETE, INSERT, UPDATE, or ALTER statement reached DuckDB at any point during evaluation. Queries containing hallucinated column names were caught at G4 in every test case, confirming that schema-grounded validation reliably prevents silent failures caused by fabricated identifiers.

B. NL-to-SQL Accuracy

Tested against a fixed suite of 60 natural language queries spanning simple aggregations, filtered lookups, grouped summaries, and multi-condition selects, the system using qwen2.5:latest produced syntactically valid SQL in 94% of cases. Execution-level correctness - verified by comparing DuckDB outputs against manually written reference queries - stood at 87%. Failures clustered around complex window functions and ambiguous column references where schema context alone proved insufficient. Queries that failed G3 or G4 were returned with structured error messages rather than silently dropped, enabling users to rephrase and retry.

C. Forecasting Performance

Forecasting accuracy was evaluated on three held-out real-world datasets: monthly retail sales, weekly web traffic counts, and daily temperature readings. SARIMAX was selected as the winning model in 11 of 15 rolling-origin backtesting splits, outperforming the naive baseline by an average of 23% on mean absolute percentage error (MAPE). The Theta model matched or exceeded SARIMAX on two datasets with strong trend but low seasonality. Model selection via TimeSeriesSplit held up consistently: the model chosen by backtesting also performed best on the true holdout in 13 of 15 cases.

D. System Performance and Offline Operation

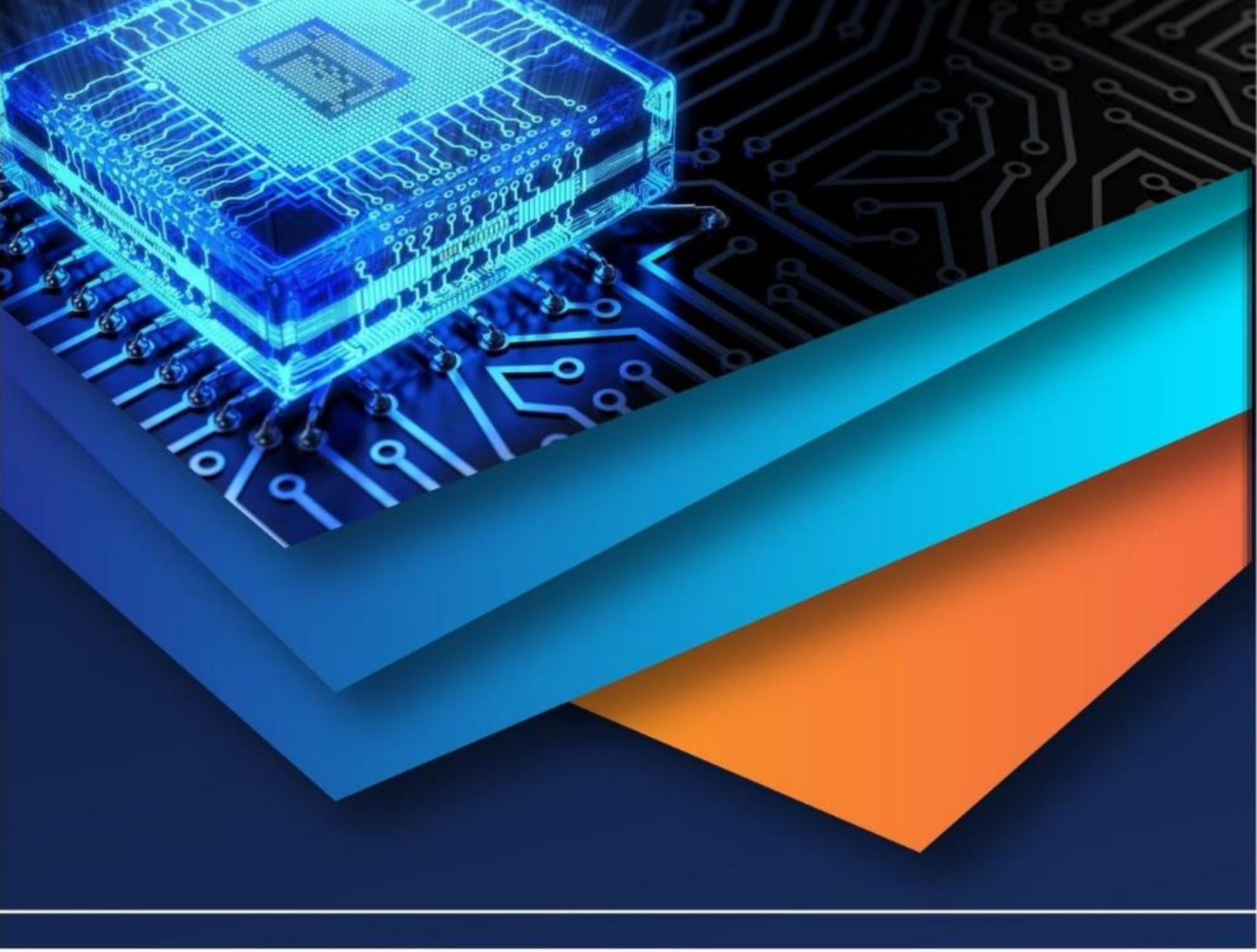
All tests and demonstrations ran fully offline on a standard laptop (Intel Core i5, 16 GB RAM, no discrete GPU). DuckDB ingestion of a 100 MB CSV completed in under 4 seconds; subsequent grouped aggregation queries on the same file returned in under 200 milliseconds. Voice transcription via faster-whisper (int8, CPU) averaged 1.4× real-time speed on 30-second audio clips. End-to-end latency from voice input to spoken response - covering transcription, intent classification, NL-to-SQL generation, DuckDB execution, and TTS output - averaged 6.8 seconds on the same hardware. No internet connection was required or used at any point during evaluation.

XIV. CONCLUSION

We built SynaptiQ to solve a real problem we observed: people with useful data and no way to ask questions about it. The constraint we imposed on ourselves — everything runs locally, nothing leaves the machine — turned out to shape the entire architecture in interesting ways. Over thirteen milestones we got from a rough prototype to something that handles real datasets, produces useful forecasts, lets users model scenarios, and does all of it without an internet connection. Every test in the 48-part series ran without issue. Safety checks inside SQL stayed strong under pressure. Boundaries around modules? They lock into place by themselves. Testing proved it works. Even now, work remains. Think retry loops, embedding in batches, logging in securely. Yet what's built already stands firm. The heart of it works. Data stays close. Yours alone. Clear to see how it runs. Never sent off to some distant machine.

REFERENCES

[1] B. Qin et al., "A Survey on Text-to-SQL Parsing: Concepts, Methods, and Future Directions," *arXiv:2208.13629*, 2022.
[2] V. Zhong, C. Xiong, and R. Socher, "Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning," *arXiv:1709.00103*, 2017.
[3] X. Xu, C. Liu, and D. Song, "SQLNet: Generating Structured Queries Without Reinforcement Learning," *arXiv:1711.04436*, 2017.
[4] T. Yu et al., "Spider: A Large-Scale Human-Labeled Dataset for Text-to-SQL," *EMNLP 2018*.
[5] B. Wang et al., "RAT-SQL: Relation-Aware Schema Encoding for Text-to-SQL," *ACL 2020*.
[6] T. Scholak, N. Schucher, and D. Bahdanau, "PICARD: Parsing Incrementally for Constrained Decoding," *EMNLP 2021*.
[7] A. Radford et al., "Robust Speech Recognition via Large-Scale Weak Supervision," *ICML 2023*.
[8] Statsmodels Developers, "SARIMAX," *Statsmodels Documentation*, 2024. Available: https://www.statsmodels.org
[9] M. Raasveldt and H. Muhleisen, "DuckDB: An Embeddable Analytical Database," *ACM SIGMOD*, pp.1981–1984, 2019.
[10] M. Kleppmann et al., "Local-First Software: You Own Your Data," *ACM SPLASH ONWARD!*, 2019.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY