Week 9 Assignment: Database Integration for Chatbot

ITEC5025 – Natural Language Processing in AI Chatbots

Author: Shruti Malik

Date: March 9, 2026

DATABASES MEET CONVERSATIONAL AI:

SQLITE INTEGRATION, DYNAMIC KNOWLEDGE RETRIEVAL, AND THE ARCHITECTURE

OF A CLINICALLY-AWARE DATA-DRIVEN CHATBOT

Introduction

When I finished Week 8, I had a chatbot that could classify user intent with a Bidirectional LSTM, recognize named entities with spaCy, analyze sentiment with VADER, and look up real patient records from CSV files. It was genuinely impressive—and genuinely fragile. Every time someone asked for patient data, the chatbot loaded a 9-megabyte CSV from scratch. Every time it generated a response, it pulled from a hardcoded JSON file it had loaded into memory at startup. There was no record of any conversation ever having happened. There was no way to update the knowledge base without editing source code.

Week 9 asked a pointed question: what if the chatbot's knowledge lived in a database instead? That question led me much further than I expected. By the end of the week I had redesigned the core data layer, imported all 100,000 patient records—along with 361,760 admissions, 361,760 diagnoses, and a 100,000-row lab sample—into an SQLite database, instrumented the chatbot to log every conversation turn automatically, and built a user-feedback mechanism that stores ratings persistently. In this paper I want to explain not just what I built, but why the design looks the way it does, where the interesting technical challenges were, and what this exercise revealed about the real role of databases in production AI systems.

---

1. Why Databases? What a CSV-Driven Chatbot Cannot Do

Before I get into implementation, I want to articulate the specific problems that motivated the database integration. Understanding the problems makes the design choices that follow much clearer.

Problem 1 – Full-File Reads on Every Query

The Week 8 chatbot loaded patients_cleaned.csv every single time a user looked up a patient. For 100,000 rows and seven columns, a pandas read_csv() call takes roughly 2 to 4 seconds on a consumer laptop. For a chatbot that is supposed to feel responsive, that is an eternity. Worse, the same memory was being allocated

and deallocated on every query—a pattern that becomes untenable in any multi-user deployment.

With a database and a properly indexed patient_id column, the equivalent lookup takes under 5 milliseconds. That is not a modest improvement; it is a qualitative change in the user experience.

Problem 2 – No Persistence

The Week 8 chatbot had no memory beyond the current session's context window. When you closed the program, every conversation was gone. There was no way to analyze usage patterns, identify frequently asked questions, or audit the chatbot's behavior over time. In a clinical context, auditability is not optional—it is a regulatory requirement. Clinical decision support tools in the United States must maintain records of queries and responses for audit and oversight purposes (Shortliffe & Biomedical Informatics, 2014).

A conversation_log table changes this completely. Every turn is recorded with a timestamp, session ID, user message, bot response, and resolved intent tag. The chatbot now has a paper trail.

Problem 3 – Static Knowledge Base

The Week 8 knowledge base—the set of patterns and responses the chatbot could use—was defined at training time and embedded in a file that required a code change to update. Adding a new type of query meant editing dataset.csv, retraining the model, and redeploying. For a rapidly evolving clinical domain, that cycle is impractical.

A relational intents table can be updated without retraining. A clinical administrator could add new response templates through a simple INSERT statement. The chatbot reads from the database at query time, not at startup, so changes take effect immediately without restarting the system.

Problem 4 – No User Feedback Loop

Machine learning systems that cannot learn from their mistakes are frozen in time. The Week 8 chatbot had no way to capture when it got something wrong. The Week 9 database adds a user_feedback table that stores numeric ratings (1 to 5) and free-text comments, permanently, with session linkage. This is the first step toward a feedback loop: you cannot improve what you cannot measure.

---

2. Database Design: Choosing the Schema

Before writing a single line of SQL, I spent time thinking about what the schema needed to express and what performance requirements it had to meet.

The key design decision was how many tables to create and how to separate the chatbot's knowledge base from its operational data from the clinical records. I settled on six tables organized into two tiers:

Knowledge Base Tables:

```
intents – Maps (tag, pattern, response) triples for intent-driven
```

```
chatbot responses. Populated from dataset.csv (386 rows,
19 intent categories). This is the chatbot's core knowledge.
```

```
patients_summary – Pre-aggregated statistics from the patient population
(gender distribution, age stats, language breakdown, etc.).
Used as a fast-path answer to insights queries before the
full patient table is consulted.
```

Operational Tables:

```
conversation_log – Every chat turn: session_id, user_message, bot_response,
intent_tag, timestamp (auto-set by database DEFAULT).
```

```
user_feedback – Numeric ratings (1-5) and free-text comments linked to
session IDs. Created by the 'feedback <rating> <comment>'
chatbot command.
```

Clinical Records Tables (added in the Week 9 patient import):

```
patients – 100,000 demographic records (patient_id PRIMARY KEY, gender,
date_of_birth, race, marital_status, language, poverty_pct).
```

```
admissions – 361,760 hospital admission events (patient_id FK, admission_id,
admission_start, admission_end).
```

```
diagnoses – 361,760 ICD-10 diagnosis records (patient_id FK, admission_id,
diagnosis_code, diagnosis_description).
```

```
labs – 100,000-row sample of lab results (patient_id FK, admission_id,
lab_name, lab_value, lab_units, lab_datetime).
```

The separation of concerns here is intentional. The intents table is small (386 rows) and frequently read—every chatbot response reads from it. The clinical tables are large and read infrequently—only when a specific patient is looked up. Keeping them separate makes it easy to optimize each tier independently. In a production deployment the clinical tables might live in a different database server or be partitioned by year; the architecture supports that transition without restructuring the chatbot code.

Foreign Key Relationships:

I declared patient_id as a FOREIGN KEY from admissions, diagnoses, and labs back to the patients table, and enabled PRAGMA foreign_keys = ON at connection time. This enforces referential integrity at the database level: you cannot have an admission record for a patient who does not exist. In a production clinical system this kind of constraint catches data entry errors that would otherwise silently corrupt the dataset.

---

3. Technology Choice: Why SQLite?

SQLite has an curious reputation in the software engineering world. It is sometimes dismissed as a "toy database" for learning exercises, but that characterization is badly wrong. SQLite is the most widely deployed database engine in the world—it runs inside every Android device, every iPhone, every Firefox installation, and every Adobe application (Hipp, 2023). For applications that need a local, embedded database with full SQL semantics and ACID compliance, SQLite is often the correct

choice.

For this project, SQLite is the right tool for several specific reasons:

```
– Zero configuration: No server process, no credentials, no connection pooling. The database is a single file. You can copy it, back it up, and inspect it with any SQLite browser without stopping the application.
```

```
– Sufficient write throughput: SQLite can handle tens of thousands of writes per second with Write-Ahead Logging (WAL) mode enabled. Our chatbot writes at most a few turns per minute. SQLite is not the bottleneck.
```

```
– Full SQL semantics: The aggregation queries used for population statistics (GROUP BY gender, AVG(poverty_pct), julianday age calculations) are valid standard SQL. If we ever need to migrate to PostgreSQL for a multi-user deployment, changing the connection string is almost all that is required.
```

```
– Python standard library: sqlite3 ships with Python. No pip install, no dependency conflicts, no licensing issues. For an assignment that also needs TensorFlow, pandas, spaCy, and NLTK, eliminating one more dependency is genuinely valuable.
```

The alternative I considered was using pandas DataFrames as an in-memory database. This would have worked for the statistics queries, but it would have provided no persistence, no ACID guarantees, no foreign key enforcement, and would have required loading the entire 100,000-patient CSV into memory at startup. The 17-millisecond SQLite lookup is not just faster—it uses less RAM, starts instantly, and is queryable by any external tool.

---

4. The Modular Architecture: db_manager.py as the Gatekeeper

One of the most important software engineering decisions I made was to keep all raw SQL out of chatbot_w9.py. Instead, every database operation routes through a dedicated module, db_manager.py, that exposes a Python function interface to the rest of the application.

This is the Repository pattern (Evans, 2003)—a layer of abstraction that decouples the application logic from the persistence mechanism. The chatbot does not know whether its data is stored in SQLite, PostgreSQL, or a cloud API. It only knows that it can call:

```
db_manager.get_intent_response("greeting")
db_manager.log_conversation(session_id, user_msg, bot_response, intent)
db_manager.get_patient_from_db("P000042")
db_manager.get_patient_admissions("P000042")
db_manager.get_patient_diagnoses("P000042")
db_manager.get_patient_labs("P000042")
db_manager.get_population_stats_from_db("gender")
db_manager.add_user_feedback(session_id, rating, comment)
db_manager.search_intents("glucose")
```

Every one of these functions opens its own connection, executes its query, commits if needed, and closes the connection. This is a deliberate choice—it avoids the subtle concurrency bugs that arise from sharing connections across threads, and it

ensures that every query runs with a clean transaction state.

The functions also implement the fallback hierarchy that makes the chatbot resilient to partial deployment states. If the clinical patient tables have not been populated yet (because import_patients.py has not been run), get_patient_from_db() returns a message explaining how to run it, rather than crashing. If the database file is missing entirely, connect() raises FileNotFoundError with a clear message. The chatbot degrades gracefully to CSV-based or hardcoded responses rather than throwing an unhandled exception.

This kind of defensive design is not overhead. In a clinical setting, a chatbot that crashes silently or returns a garbled error is worse than useless—it actively undermines trust in the system.

---

5. Importing 100,000 Patients: Performance Engineering at Scale

The patient data import is where I encountered the most interesting technical challenges. Loading four CSV files into SQLite sounds straightforward, but the naive approach—read the whole file with pandas, then call to_sql()—has serious problems at the scale of this dataset.

The Data Volumes:

```
patients_cleaned.csv : 100,000 rows, 7 columns, 9.4 MB
admissions_cleaned.csv : 361,760 rows, 4 columns, 28.9 MB
diagnoses_cleaned.csv : 361,760 rows, 4 columns, 35.9 MB
labs_cleaned.csv : > 1,000,000 rows, 6 columns, 1.03 GB
```

Loading all of this naively would require several gigabytes of RAM and would block the Python process for an extended period. I made three strategic engineering decisions to address this.

Decision 1 – Chunked Reads for Large Files

For admissions and diagnoses (each about 361K rows), I read the CSV in 20,000-row chunks using pandas' chunksize parameter and wrote each chunk to SQLite before reading the next:

```
for chunk in pd.read_csv(path, chunksize=20_000):
chunk.to_sql("admissions", conn, if_exists="append", index=False)
```

This keeps peak memory usage at roughly 20,000 × (row size) rather than 361,000 × (row size)—a 10-to-1 reduction in memory footprint. It also provides a progress indicator (I printed the running row count) so the user can see that something is happening during the multi-minute import.

Decision 2 – Sampling the Labs File

The labs_cleaned.csv file is 1.03 gigabytes. Importing the full dataset into SQLite would produce a database file of several hundred megabytes—not prohibitive, but impractical for an assignment submission and for reasonable laptop storage. More importantly, one million lab rows are more than sufficient for the chatbot's use cases: lab lookup by patient ID typically returns no more than a few dozen rows,

and the statistics queries do not need a 99.99% sample to be representative.

I implemented a configurable sampling mechanism: the import reads rows in 10,000-row chunks, accumulates them until the sample size is reached, then stops. The default is 100,000 rows (adjustable via --labs-sample <n>). This produces a representative sample that answers most realistic lab queries while keeping the database to a manageable size.

Decision 3 – Write-Ahead Logging and Performance PRAGMAs

SQLite in its default mode writes synchronously to disk at the end of every transaction. For bulk inserts, this is extremely slow. I enabled three SQLite PRAGMAs that together made a dramatic difference in import speed:

```
PRAGMA journal_mode = WAL — Write-Ahead Logging decouples writes from
reads. Concurrent read operations do not
block during the bulk insert.
```

```
PRAGMA synchronous = NORMAL — Reduces disk sync frequency. Some atomicity
guarantees are relaxed, but this is acceptable
during a controlled import operation.
```

```
PRAGMA cache_size = -64000 — Allocates 64 MB of page cache in memory.
Reduces the frequency of disk reads during
write operations.
```

With these optimizations, the import of all 100,000 patient rows completed in about 25 seconds on my laptop. Admissions and diagnoses (each 361K rows) took approximately 3 minutes each. The full import—patients, admissions, diagnoses, and the 100K lab sample—completed in under 8 minutes total. Without the optimizations, preliminary tests suggested times 5 to 10 times longer.

Decision 4 – Indexes for Fast Lookup

After the import, I created indexes on the patient_id columns of the admissions, diagnoses, and labs tables:

```
CREATE INDEX idx_admissions_patient ON admissions(patient_id);
CREATE INDEX idx_diagnoses_patient ON diagnoses(patient_id);
CREATE INDEX idx_labs_patient ON labs(patient_id);
```

A database index is a pre-computed data structure (typically a B-tree) that allows the database engine to find rows matching a WHERE clause without scanning every row in the table. Without an index, looking up admissions for patient "P000042" in a 361,760-row table requires examining all 361,760 rows. With the index, the engine navigates the B-tree in O(log n) time and finds the matching rows in milliseconds (Date, 2003). For a chatbot where users expect instant responses, this is not an optimization—it is a prerequisite.

---

6. Population Statistics: SQL vs. Pandas—A Principled Comparison

The Week 8 chatbot computed population statistics (gender distribution, age breakdown, poverty analysis, etc.) by loading the full 100,000-row patients CSV into a pandas DataFrame at query time. In Week 9, the primary source for these

statistics is the patients SQL table, queried directly.

I want to be transparent about why I believe the SQL approach is objectively better for this use case, and where pandas still has advantages.

Advantages of SQL for Population Stats:

```
1. No data loading cost. The database is already open. A GROUP BY query over
the indexed patients table runs in milliseconds; loading a 9MB CSV takes
2-4 seconds.
```

```
2. Precise aggregation. SQLite's julianday() function computes exact age in
fractional years from date_of_birth—the same result pandas date arithmetic
would produce, but expressed in six lines of SQL rather than twelve lines
of Python.
```

```
3. Single source of truth. The patients table is now the canonical record.
The CSV file can be archived. There is no risk of the statistics drifting
out of sync with the actual data.
```

Advantages of Pandas (where it still applies):

```
1. Complex operations. Pandas is better for operations that SQL cannot express
cleanly: applying custom Python functions to each row, producing complex
visualizations, or performing machine learning feature engineering.
```

```
2. Exploratory analysis. An interactive Jupyter notebook with a pandas DataFrame
is more flexible than a SQL prompt for exploring unknown datasets.
```

In the chatbot_w9.py fallback chain, I kept the CSV-based pandas path as a secondary source for cases where the clinical tables are not yet loaded. This preserves backward compatibility with Week 8 and demonstrates both approaches in the same codebase.

The priority order the chatbot uses:

```
1. SQLite patients table (fastest, always current)
2. pandas / patients_cleaned.csv (slower, works without db import running)
3. pre-computed patients_summary values (Week 8 fallback)
4. Hardcoded defaults
```

---

7. New Chatbot Commands and the Expanded User Interface

Week 9 adds five new command types to the chatbot's interface, each backed by a direct SQL query:

'patient <id>'

```
Queries the patients SQL table for exact ID match, then case-insensitive match.
Returns a formatted 6-field record. Falls back to CSV if the clinical tables
are not loaded.
```

'admissions <id>'

```
Queries the admissions table for all hospital stays associated with the patient.
Returns admission IDs, start dates, and end dates in chronological order.
```

'diagnoses <id>'

```
Queries the diagnoses table for up to 20 unique ICD-10 coded diagnoses
for the patient. De-duplicates entries in Python before display.
```

'labs <id>'

```
Queries the labs table for the most recent 15 lab results, sorted
descending by lab_datetime. Shows lab name, value, units, and date.
```

'db info'

```
Calls db_manager.get_db_stats() and formats the response as a table of
all eight database tables with their current row counts. After the full
patient import, this shows:
intents : 386 rows
patients_summary : 51 rows
conversation_log : (grows with use)
user_feedback : (grows with use)
patients : 100,000 rows
admissions : 361,760 rows
diagnoses : 361,760 rows
labs : 100,000 rows (sampled)
```

The existing 'search <keyword>', 'feedback <1-5> <comment>', and all Week 8 NLP commands (sentiment analysis, NER, context-aware follow-up) remain intact.

---

8. Conversation Logging: The Chatbot's Memory

Every time chatbot_w9.py generates a response, it calls:

```
db_manager.log_conversation(
session_id = self.session_id, # UUID for this run
user_message = user_input,
bot_response = response,
intent_tag = resolved_intent,
)
```

The conversation_log table stores these rows with an auto-incremented id and a DEFAULT CURRENT_TIMESTAMP. The session_id is a UUID substring generated at startup, so multiple concurrent sessions remain distinct in the log.

What does this enable?

```
– Usage analytics: Which intents are most frequently requested? What time of
day is the chatbot busiest? Which patient IDs are looked up most often?
```

```
– Error detection: Filter conversation_log for entries where intent_tag =
'unknown'. High rates of unknown-intent responses indicate gaps in the
training data that need addressing.
```

```
– Audit trail: In a HIPAA context, every access to a patient record must be
logged with a user identifier and timestamp. The conversation_log provides
the raw data for that audit, though a production system would need to capture
authenticated user IDs rather than session GUIDs.
```

```
– Feedback correlation: Join conversation_log with user_feedback on session_id
to understand which types of interactions generate negative ratings.
```

The smoke test I ran during development—10 predefined queries—produced 20 rows in conversation_log (2 per query because the chatbot generates a response and logs it). This confirmed that the logging pipeline works correctly in an automated test

environment.

---

9. The SQL-to-JSON Export: Closing the Loop

One Week 9 requirement I found particularly interesting was the need to export SQL data back into a JSON file. This initially seemed like an odd requirement for a database integration assignment—why export to JSON if we already have SQL?—but I think the exercise captures something real about how data pipelines work in practice.

Systems rarely exist in isolation. The Hypotify chatbot needs a JSON file for its JSON-based fallback path. The training pipeline that produced the Week 8 model reads JSON. If another team wanted to build a different chatbot with the same knowledge base, they might prefer a JSON file to SQLite access. The SQL database is the authoritative source, but JSON is a portable interchange format.

The export_intents_to_json.py script:

```
1. Queries SELECT tag, pattern, response FROM intents
2. Groups rows by tag using a Python defaultdict
3. Structures the output as { "intents": [ { "tag": ..., "patterns": [...],
"responses": [...] }, ... ] }
4. Writes to intents.json using json.dump with indent=2
```

The resulting file (41,606 bytes, 19 intent tags) is format-compatible with the intents.json that the Week 7 and Week 8 chatbots expect. Running the export and then the chatbot demonstrates the full cycle: CSV → database → SQL query → JSON export → chatbot response.

---

10. Reflection: What Database Integration Actually Changes

I want to close by being honest about what changed between Week 8 and Week 9, beyond the obvious technical additions.

The Week 8 chatbot was what I would call a self-contained artifact. All of its knowledge lived in files that were part of the application. Consulting it meant consulting those files. Improving it meant modifying those files.

The Week 9 chatbot is the beginning of a system. Its knowledge lives in a database that is queryable and updatable independently of the application code. Its conversations are logged to a table that is queryable in real time. Its user feedback is stored in a record that could be used to drive automated retraining. These are not just technical features—they represent a different relationship between the AI system and the organization that deploys it.

In a real clinical environment, the knowledge base would be maintained by clinical informatics staff who might not be Python programmers. The conversation logs would be reviewed by clinical AI safety officers. The feedback records would feed into a quality improvement process. None of that is possible with a file-based system.

The hardest part of this week was not the SQL. SQLite's API is clean, well-documented, and works exactly as advertised. The hard part was designing the fallback chain correctly—deciding what the chatbot should do when the clinical tables are not loaded yet, when the database file is missing, when a patient ID does not exist in the DB or the CSV. Getting the error paths right is what makes a system trustworthy rather than just functional, and it is something I underestimated going in.

The Week 8 model is not production-ready. The Week 9 system is a step closer—not because of any individual feature, but because it has the structural properties that production systems require: persistence, auditability, updatability, and graceful degradation. That is what database integration actually changes.

Conclusion

Week 9 transformed the Hypotify chatbot from a stateless inference engine into a stateful, data-driven system. An SQLite database now stores the chatbot's knowledge (386 intent triples), the clinical knowledge base (100,000 patients, 361,760 admissions and diagnoses each, 100,000 lab samples), every conversation turn in a timestamped log, and every user feedback rating. A dedicated db_manager.py module ensures that all database interactions are encapsulated behind clean Python functions, with robust error handling and graceful fallback to CSV-based data when needed.

The performance improvements over the Week 8 CSV-driven approach are substantial: millisecond patient lookups instead of 2-4 second CSV reads, indexed aggregation queries instead of full-table scans, and persistent storage that survives session boundaries. The new chatbot commands—admissions, diagnoses, labs, db info—give users direct access to the clinical database through natural language.

The most important lesson from this week is architectural: a database is not just faster storage. It is a shared, queryable, auditable record of what the system knows and what it has done. For an AI system operating in a clinical context, that distinction is the difference between a useful tool and a trustworthy one.

References

Date, C. J. (2003). An introduction to database systems (8th ed.). Addison-Wesley. `https://dl.acm.org/doi/book/10.5555/861569`

Evans, E. (2003). Domain-driven design: Tackling complexity in the heart of `software. Addison-Wesley Professional. https://www.oreilly.com/library/view/domain-driven-design-tackling/0321125215/`

Hipp, D. R. (2023). SQLite documentation: Features and use cases. `https://www.sqlite.org/whentouse.html`

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural `Computation, 9(8), 1735–1780.`

https://doi.org/10.1162/neco.1997.9.8.1735

Hutto, C. J., & Gilbert, E. (2014). VADER: A parsimonious rule-based model for sentiment analysis of social media text. Proceedings of the 8th International AAAI Conference on Weblogs and Social Media (ICWSM-14). https://ojs.aaai.org/index.php/ICWSM/article/view/14550

Jurafsky, D., & Martin, J. H. (2023). Speech and language processing (3rd ed. draft). Stanford University. https://web.stanford.edu/~jurafsky/slp3/

Kreimeyer, K., Foster, M., Pandey, A., Arya, N., Halford, G., Jones, S. F., Forshee, R., Walderhaug, M., & Botsis, T. (2017). Natural language processing systems for capturing and standardizing unstructured clinical information: A systematic review. Journal of Biomedical Informatics, 73, 14–29. https://doi.org/10.1016/j.jbi.2017.07.012

Shortliffe, E. H., & Cimino, J. J. (Eds.). (2014). Biomedical informatics: Computer applications in health care and biomedicine (4th ed.). Springer. https://doi.org/10.1007/978-1-4471-4474-8

SQLite Consortium. (2023). SQLite FAQ: Is SQLite appropriate for use in production? https://www.sqlite.org/faq.html

Stonebraker, M., & Cattel, R. (2011). 10 rules for scalable performance in simple operation datastores. Communications of the ACM, 54(6), 72–80. https://doi.org/10.1145/1953122.1953144

End of Paper