Georgia Southern University
Allen E. Paulson College of Engineering and Computing
Department of Computer Science

CSCI 3432: Database Systems

PharmaCheck

Anish Goyal

Dr. Weitian Tong
Associate Professor

December 4, 2025

Objective

This project presents PharmaCheck, a web-based drug interaction checking system. I built PharmaCheck using a MySQL database backend, a Flask API server, and a modern HTML/CSS/JavaScript frontend. The system scrapes real-time drug interaction data from Drugs.com, stores it in a normalized relational database, and uses a locally deployed Ollama large language model to translate professional medical descriptions into patient-friendly language. PharmaCheck supports doctor-patient relationships, comprehensive search history tracking, and checks for drug-drug, food-lifestyle, and disease interactions.

Table of Contents

List of Figures

1 Introduction

This semester I received a cancer diagnosis. Treatment required managing multiple medications simultaneously. I witnessed firsthand how doctors track complex drug interactions using internal hospital systems. Patients like me often lack visibility into potential medication conflicts. This experience inspired me to build an accessible system for drug interaction awareness.

Drug interactions cause over 100,000 hospitalizations annually in the United States. According to recent statistics, 82% of Americans take at least one medication. 29% take five or more medications daily. Drug interactions rank as the fourth leading cause of death in the US. Healthcare providers use proprietary internal systems that patients cannot access. Existing public tools lack transparency and real-time accuracy.

PharmaCheck is a database-managed web application that integrates artificial intelligence and advanced indexing to identify adversarial drug interactions in patients. I built this platform to address the gap between professional medical systems and patient needs. The system provides real-time drug interaction checking, food and lifestyle interaction warnings, disease interaction alerts, and AI-powered translation of medical terminology.

The main components of PharmaCheck include a MySQL relational database storing drugs, conditions, and interaction data. A Flask backend serves a RESTful API for all operations. The frontend uses vanilla HTML, CSS, and JavaScript without heavy framework overhead. A web scraping system fetches real-time data from Drugs.com. An Ollama large language model translates professional descriptions into patient-friendly language. A doctor-patient relationship system enables medical oversight. A comprehensive search history feature allows users to revisit previous queries.

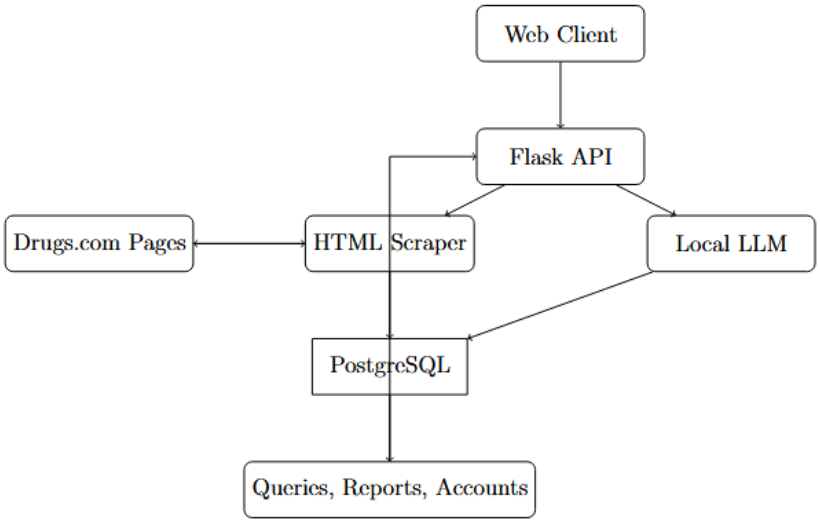

Figure 1: Technology stack showing MySQL, Flask, and frontend components

Figure 1 shows the technology stack I chose for PharmaCheck. MySQL provides reliable relational storage. Flask offers a lightweight Python web framework. The frontend prioritizes simplicity and performance.

2 Database Details

I designed the database schema through careful entity-relationship modeling. The design process began with identifying core entities: users, drugs, conditions, and interactions. I mapped relationships between these entities and planned for real-time data acquisition from external sources.

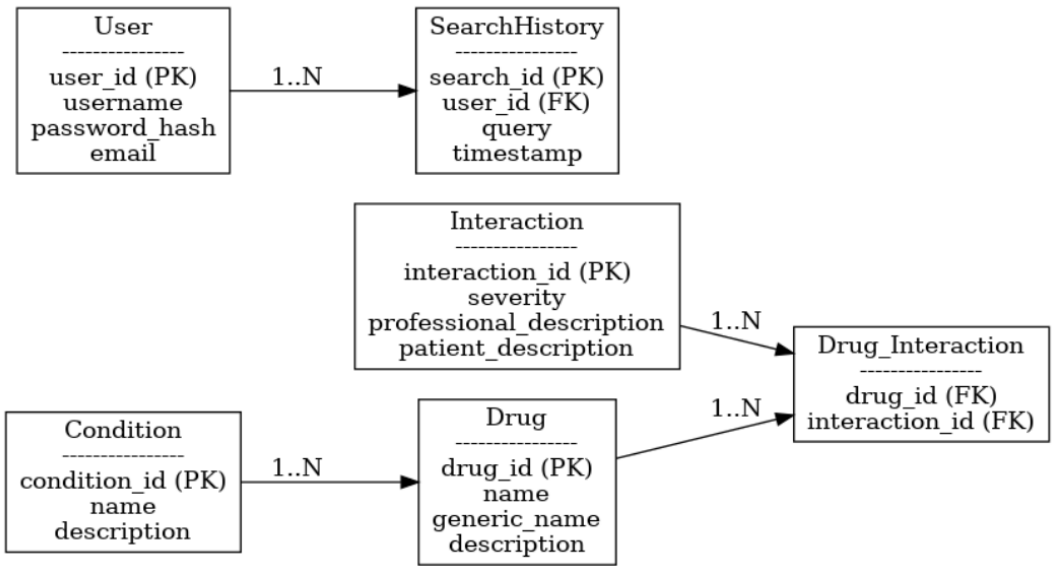

Figure 2: Entity-Relationship diagram for PharmaCheck

Figure 2 shows the ER model I created during the initial design phase. The model captures users with distinct roles, drugs with their properties, conditions they treat, and various types of interactions between drugs.

The final schema contains nine tables. The User table stores authentication credentials and role information. Each user has a unique identifier, username, password hash, email, and role designation as either PATIENT or DOCTOR. The Drug table contains medication information including name, generic name, description, URL reference, and an optional foreign key to the Condition table. The Condition table stores medical conditions with names, descriptions, and URLs. The Interaction table captures drug-drug interaction details including severity level, professional description, patient description, and cached AI-generated description.

I created a junction table called Drug_Interaction to handle the many-to-many relationship between drugs and interactions. This table contains foreign keys to both the Drug and Interaction tables along with the name of the interacting drug. The FoodInteraction table stores food and lifestyle interactions with severity, hazard level, plausibility rating, and descriptions. The DiseaseInteraction table follows a similar structure for disease-related interactions. The

SearchHistory table logs all user searches with timestamps, query text, search type, and full JSON results for restoration. The Doctor_Patient table manages the many-to-many relationship between doctors and their patients.

The functional dependencies in each table follow Third Normal Form requirements. For the User table: user_id determines username, password_hash, email, role, created_at, and updated_at. For the Drug table: drug_id determines name, generic_name, description, url, condition_id, created_at, and updated_at. For the Interaction table: interaction_id determines severity, professional_description, patient_description, ai_description, url, created_at, and updated_at. For SearchHistory: search_id determines user_id, query, search_type, search_data, and created_at.

All tables satisfy Third Normal Form. No transitive dependencies exist because all non-key attributes depend directly on the primary key. I avoided storing derived data. Each attribute represents a single fact about the entity identified by the primary key.

I implemented several constraint types to maintain data integrity. Primary keys use auto-incrementing integers for efficient indexing. Foreign keys include CASCADE rules for both UPDATE and DELETE operations. When a user deletes their account, all related search history entries delete automatically. The UNIQUE constraint applies to username and email fields in the User table to prevent duplicates. NOT NULL constraints ensure required fields always contain values. ENUM types restrict the role field to PATIENT or DOCTOR values and the severity field to Major, Moderate, Minor, or Unknown values.

The following SQL shows the User table definition:

```sql
CREATE TABLE User (
    user_id INT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(64) NOT NULL UNIQUE,
    password_hash CHAR(60) NOT NULL,
    email VARCHAR(255) NOT NULL UNIQUE,
    role ENUM('PATIENT', 'DOCTOR') NOT NULL,
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
        ON UPDATE CURRENT_TIMESTAMP
);
```

The SearchHistory table definition includes the search_data column for storing complete JSON results:

```sql
CREATE TABLE SearchHistory (
    search_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    user_id INT NOT NULL,
    query TEXT NOT NULL,
```

```
    search_type ENUM('DRUG', 'CONDITION', 'INTERACTION',
        'FOOD_INTERACTION', 'DISEASE_INTERACTION') DEFAULT 'DRUG',
    search_data TEXT,
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES User(user_id)
        ON UPDATE CASCADE ON DELETE CASCADE,
    INDEX idx_search_user (user_id),
    INDEX idx_search_created (created_at)
);
```

I added indexes on frequently queried columns. The drug name and generic name columns have indexes for fast autocomplete searches. The search history table has indexes on user_id and created_at for efficient retrieval of recent searches. Full-text indexes on drug and condition names support future advanced search features.

3 Functionality Details

I implemented both basic and advanced functions in PharmaCheck. The basic functions handle user authentication, drug searching, and simple interaction checks. The advanced functions provide multi-drug analysis, AI translation, and doctor-patient management.

User registration accepts a username, email, password, and role selection. Patients can optionally select a doctor during registration for immediate oversight assignment. The system hashes passwords using bcrypt before storage. Upon successful registration, the API returns a JWT token for subsequent authenticated requests. Login accepts username or email with password verification. The JWT token expires after a configurable duration, defaulting to one hour.

Drug search uses database-backed autocomplete. When you type at least two characters, the system queries the Drug table using prefix matching. The database contains 15,775 drugs imported from Drugs.com. If the database returns no matches, the system falls back to a JSON file search. Results return drug names, URLs, and generic names when available.

The multi-drug interaction checker accepts up to five drugs simultaneously. For each drug in the list, the system fetches interactions from Drugs.com and checks if other drugs in your list appear in the interaction results. The checker uses both exact string matching and fuzzy matching with Levenshtein distance to catch variations in drug naming. The following pseudocode describes the core logic:

```
for each drug in user_drug_list:
    interactions = fetch_interactions(drug)
    for each interaction in interactions:
        for each other_drug in user_drug_list:
            if other_drug != drug:
                if other_drug in interaction.name or
                   is_similar(other_drug, interaction.name):
                    add_to_results(drug, other_drug, interaction)
```

Food and lifestyle interaction checking fetches data from a separate Drugs.com endpoint. The scraper parses HTML to extract interaction names, severity levels, hazard ratings, plausibility scores, and detailed descriptions. Disease interaction checking works similarly, extracting applicable conditions and clinical recommendations.

The AI translation feature uses a locally deployed Ollama instance running the Llama 3.2 model. When you request a translation, the system sends the professional description to Ollama with a prompt instructing it to act as a clinical physician translating for patient comprehension. The translated text caches in the database to avoid repeated API calls. The translation function appears below:

```
def translate_professional_to_consumer(professional_description):
    prompt = f"""Pretend you are a clinical physician.
    Translate the following professional drug interaction
    description into a more consumer-friendly description.
    Write the consumer-friendly description only:

    {professional_description}"""

    response = requests.post(
        f"{OLLAMA_BASE_URL}/api/generate",
        json={"model": "llama3.2:3b",
              "prompt": prompt,
              "stream": False},
        timeout=60
    )
    return response.json().get("response", "")
```

The doctor-patient relationship system allows patients to request oversight from registered doctors. Doctors can view all assigned patients and access their search histories. This feature enables medical professionals to monitor what drug combinations their patients investigate. Patients retain control and can remove doctor assignments at any time.

Search history tracking stores complete interaction results as JSON. When you click a previous search, the system restores the full results without re-scraping. This improves performance and provides consistent historical data even if Drugs.com content changes. The search type field distinguishes between drug searches, condition searches, drug-drug interactions, food interactions, and disease interactions.

4 Implementation Details

I chose Python 3 with Flask for the backend because of its simplicity and extensive library ecosystem. MySQL serves as the primary database due to its reliability and wide support. The frontend uses vanilla HTML, CSS, and JavaScript to minimize dependencies and maximize performance.

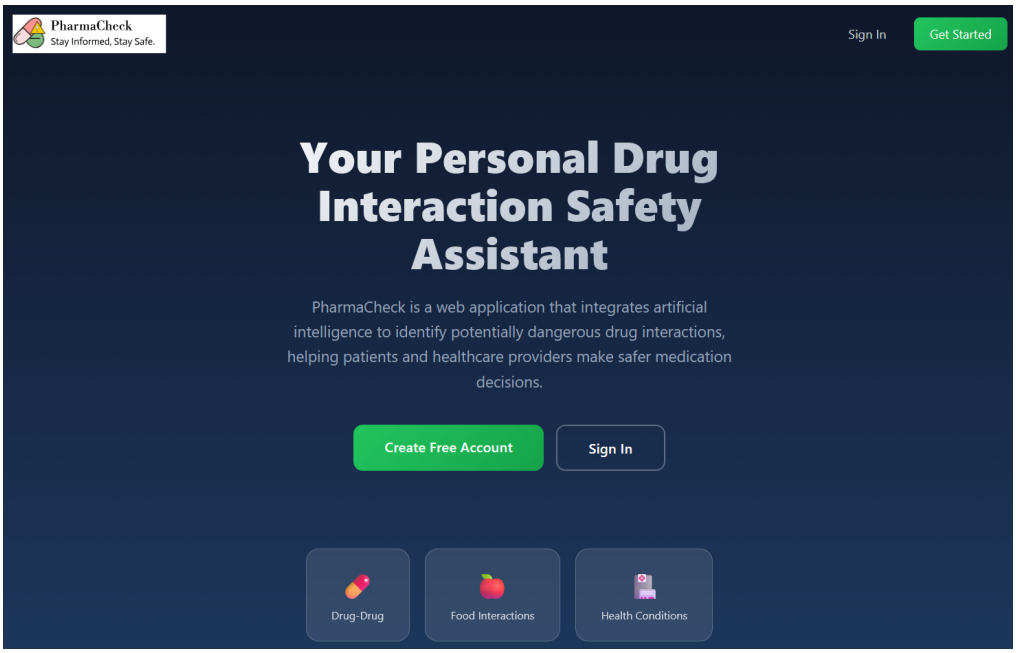

Figure 3: Welcome page showing the PharmaCheck landing interface

Figure 3 shows the welcome page that greets users. The design emphasizes clarity and medical professionalism. Navigation links direct users to registration, login, or the main dashboard.

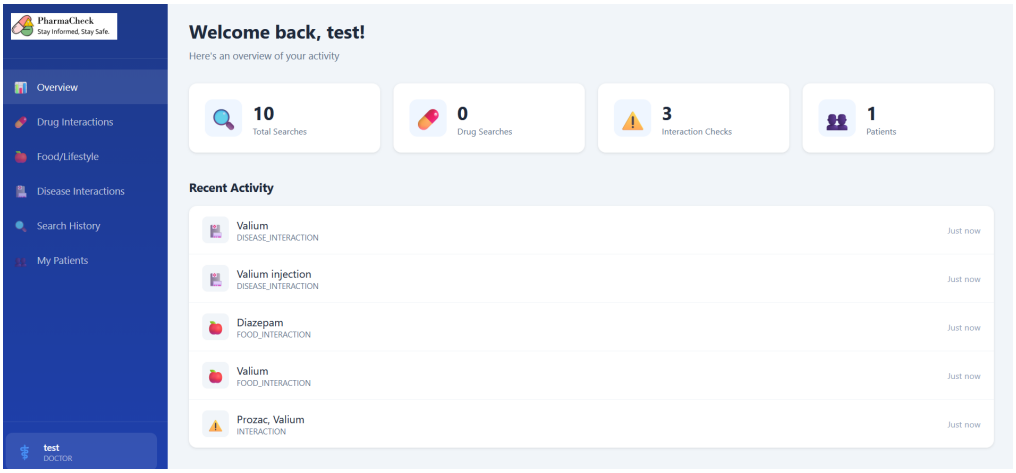

Figure 4: Dashboard interface with interaction checking options

The dashboard in Figure 4 provides access to all system features. Users can check drug interactions, view food and lifestyle warnings, explore disease interactions, and review their search history.

The Flask application structure follows RESTful principles. Authentication endpoints handle registration and login at /auth/register and /auth/login. Drug search endpoints provide autocomplete at /drugs/autocomplete. Interaction checking occurs at /check_drug_interactions for multi-drug analysis. The API returns JSON responses for all endpoints.

SQLAlchemy provides the object-relational mapping layer. I configured connection pooling with pool_size=10 and max_overflow=20 to handle concurrent requests efficiently. Scoped sessions ensure thread safety. The teardown_appcontext decorator automatically closes sessions after each request. The database URL reads from environment variables for deployment flexibility.

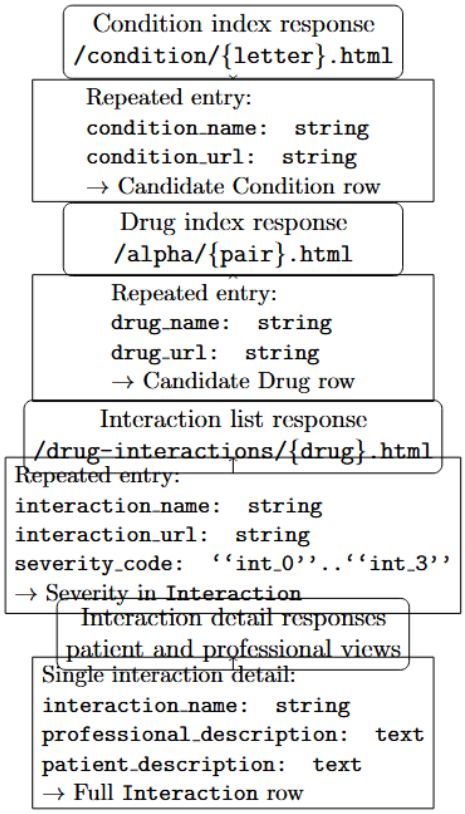

Figure 5: Web scraping workflow from Drugs.com to database

Figure 5 illustrates the web scraping workflow. The DrugInteractionChecker class manages all scraping operations. When you request interactions for a drug, the system first checks the database cache. If cached data exists and use_cache=True, the system returns stored results immediately. Otherwise, the scraper fetches fresh data from Drugs.com, parses the HTML using BeautifulSoup, extracts interaction details, stores them in MySQL, and returns the results.

Brand name to generic name resolution handles cases where drugs have multiple names. Prozac and fluoxetine refer to the same medication but use different URL patterns on Drugs.com. The scraper checks the drug's main page to find the generic name, then uses that for interaction lookups. This ensures complete interaction data regardless of which name you enter.

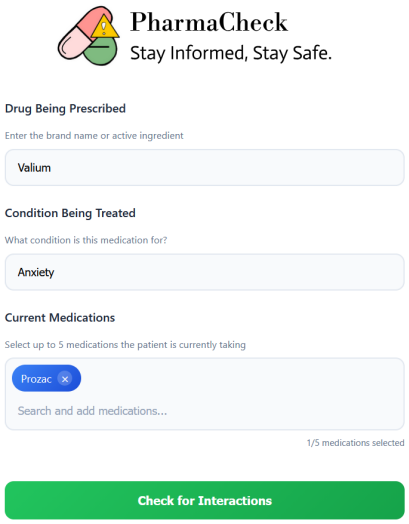

Figure 6: Drug interaction checker showing severity-coded results

Figure 6 displays the interaction checker interface. Results appear color-coded by severity. Major interactions show in red. Moderate interactions appear in yellow. Minor interactions display in green. Each result expands to show full professional and patient-friendly descriptions.

Authentication uses bcrypt for password hashing with automatic salt generation. JWT tokens encode user_id and role claims. The login_required decorator validates tokens on protected routes. The role_required decorator restricts certain endpoints to doctors only.

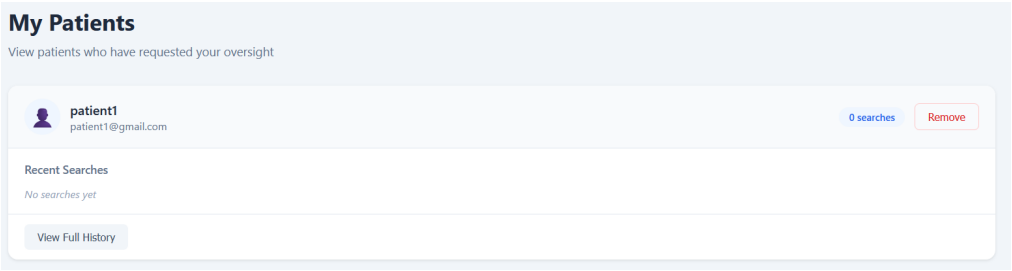

Figure 7: Doctor's patient management view

Figure 7 shows the doctor's view of assigned patients. Doctors see patient usernames and recent search activity. Clicking a patient reveals their complete search history. This enables doctors to monitor patient medication research and provide informed guidance.

Initial data loading imports 2,123 conditions and 15,773 drugs from JSON files. The import script processes records in batches of 100 to prevent memory issues. Progress feedback prints

every 100 records. The entire import completes in approximately two minutes on standard hardware.

The frontend communicates with the backend through fetch API calls. All requests include the JWT token in the Authorization header. Error responses display user-friendly messages. Loading indicators show during long operations like web scraping or AI translation.

5 Experiences

I encountered several challenges during development. The HTML structure on Drugs.com changed multiple times during the project. My initial selectors stopped working when the site updated its CSS classes. I solved this by implementing multiple fallback selectors in the parsing code. The scraper tries several patterns until one succeeds. This approach provides resilience against future site changes.

Brand name resolution proved more complex than expected. Searching for "Valium" required finding "diazepam" for the interaction lookup. The drugs use entirely different URL paths. I implemented a lookup function that visits the brand name page, extracts the generic name from the HTML, and uses that for subsequent requests. This adds an extra HTTP request but ensures accurate results.

Performance became a concern when initial scraping took over two minutes per multi-drug check. Users experienced unacceptable wait times. I implemented intelligent caching that stores scraped interactions in MySQL. Subsequent requests for the same drug return cached data instantly. The use_cache parameter allows forcing fresh data when needed. This reduced typical response times to under one second for cached drugs.

Schema evolution mid-development taught me the importance of migration planning. I initially designed only drug-drug interactions. User feedback requested food and disease interactions. Adding the FoodInteraction and DiseaseInteraction tables required careful migration. I created a separate migration SQL file to add the new columns and modify ENUM values. Future projects will include schema versioning from the start.

Working with real-world web data reinforced the need for defensive programming. HTML parsing can fail in countless ways. Missing elements, changed class names, and unexpected content all cause errors. I wrapped parsing operations in try-except blocks and provided sensible defaults. The system degrades gracefully rather than crashing on unexpected input.

The doctor-patient feature design went through multiple iterations. My original design had doctors adding patients to their list. User testing revealed this felt intrusive. I reversed the flow so patients request doctor oversight. Patients retain control while doctors can still monitor those who opt in. This matches the real-world dynamic better.

Several lessons emerged from this project. Real databases require planning for change. Schema evolution happens in every long-lived application. Building migration support early saves time later. Web scraping needs resilience and fallback strategies. External data sources change without notice. Caching dramatically improves user experience for repeated queries.

Future development could extend PharmaCheck in several directions. A background job queue using Celery would allow asynchronous scraping. Users could submit requests and receive notifications when results are ready. Elasticsearch integration would enable full-text search across drug descriptions. A React Native mobile application would reach users on smartphones. PDF report generation would let users share interaction reports with healthcare providers.

Integration with pharmacy systems could pre-populate current medications. An analytics dashboard would help doctors track patient adherence patterns.

The project demonstrates practical application of database concepts from this course. Normalization ensures data integrity. Indexing enables fast searches. Foreign key constraints maintain referential integrity. Transaction management prevents data corruption. These concepts moved from theoretical to practical through hands-on implementation.

References

[1] R. Ramakrishnan and J. Gehrke, *Database management systems*, 3rd Edition. McGraw-Hill, 2003.

[2] Drugs.com, "Drugs.com drug interaction checker." https://www.drugs.com/drug_interactions.html, 2024.

[3] Ollama, "Ollama: Run large language models locally." https://ollama.ai, 2024.

[4] Pallets Projects, "Flask: A python microframework." https://flask.palletsprojects.com, 2024.

[5] SQLAlchemy, "SQLAlchemy: The python SQL toolkit and ORM." https://www.sqlalchemy.org, 2024.