Drug Interaction Checker

1. Introduction

Medication safety really matters in healthcare. When two drugs interact badly, it can mean serious problems—bleeding, toxicity, or your treatment just doesn't work like it should. The Drug Interaction Checker isn't some clinical tool, but a simple, educational web app. It shows how you can spot known drug-drug interactions using a structured dataset and straightforward logic.

With this app, you pick a few meds and it warns you if there's a known issue between any of them. The whole system's built to be simple, modular, and easy to understand. It's meant for learning, prototyping, or demos—not actual medical decisions.

This report breaks down how the app's designed, how data and logic are set up, and why we picked the tech and algorithms we did.

2. System Architecture and Design Decisions

2.1 Layered Architecture

The app uses a three-layer setup:

1. Data Layer
2. Logic Layer
3. Presentation (UI) Layer

Keeping things separate like this makes the code easier to maintain, read, and test. If you want to change one part, you don't have to worry about accidentally breaking something else.

2.2 Data Layer Design

The DrugDatabase class handles the Data Layer. Its only job is to store and manage the known drug-drug interaction data.

Why do it this way?
- It uses in-memory data, not an external database.
- Interaction descriptions are easy to read.
- Clarity comes first—even if it means not being totally comprehensive.

With this setup, there's no need for complicated configs or outside dependencies. The app stays lightweight and easy to run.

2.3 Logic Layer Design

The Logic Layer lives in the InteractionChecker class. This is where all the logic for finding interactions happens.

Here's what it does:
- Cleans up user input.
- Looks at all combinations of meds.
- Checks for known interactions.
- Sends clear results to the UI.

By keeping logic separate from the UI, you can reuse the interaction checker in unit tests, APIs, command-line tools, or even plug it into bigger healthcare systems down the line.

2.4 User Interface Layer

For the UI, we use Streamlit. It's a Python framework that makes building data apps quick and painless.

Why Streamlit?

- Hardly any boilerplate code.
- Built-in interactive widgets.
- Instant visual feedback.
- Perfect for fast prototyping and demos.

The UI stays simple on purpose. There’s no clutter—just the interactions you care about.

3. Data Structures

3.1 Interaction Storage Structure

At the heart of the app is a nested Python dictionary:

```
{
  "Drug A": {
    "Drug B": "Interaction description"
  }
}
```

Why stick with dictionaries?
- You get super-fast lookups.
- It’s easy to see which drugs interact and how.
- Adding new entries is simple.
- The whole thing is clean and easy to maintain.

This setup lets you catch interactions quickly, without needing complex queries or database joins.

3.2 Bidirectional Interaction Handling

In real-world drug databases, not all drug interactions go both ways. Instead of storing both “A interacts with B” and “B interacts with A,” this system saves each interaction in just one direction. Then, at runtime, it checks both ways.

This keeps the dataset smaller, avoids conflicting updates, and means you don’t have a bunch of duplicate data.

3.3 Medication List Aggregation

The app builds the list of available meds by combining all the top-level and nested dictionary keys. That way, no medication gets left out, and if you add new data, the UI updates automatically.

4. Algorithmic Design and Rationale

4.1 Interaction Detection Algorithm

The app checks for interactions by comparing every possible pair of selected medications.

Here’s the process:
1. Take the list of meds the user picked.
2. Generate all the unique pairs.
3. Check each pair in the interaction database.
4. If there’s an interaction, record it.

Pseudocode:
```
for each medication i:
    for each medication j where j > i:
        if interaction exists between i and j:
            record interaction
```

4.2 Computational Complexity

If you pick n medications:

- Time: $O(n^2)$ at worst.
- Space: $O(k)$, with k being the number of interactions found.

Is this a problem? Not really. Most users won't select more than 5 or 10 meds at once. With those numbers, the quadratic approach works fine. It keeps the logic simple, which is better for reliability and maintenance.

We skipped fancier algorithms—like graph traversal—on purpose. Keeping things clear and straightforward just makes life easier for everyone.

4.3 Why Not Use Advanced Algorithms?

You might wonder why we skipped things like graph databases, rule engines, or machine learning models. The truth is, we just didn't need them. The dataset's small, the project's meant to teach, and above all, we wanted everything to stay clear and easy to follow. That's why we picked an approach that's straightforward and easy to understand, even if it's not built to handle giant datasets.

5. Error Handling and Validation

5.1 Input Validation

We don't let users leave selections empty, and they need to pick at least two medications. Any extra spaces? Gone. This keeps things tidy and blocks most bad data from slipping through. It also helps cut down on random errors during use.

5.2 Safety Considerations

Since this deals with medical info, we built in clear warning messages and educational disclaimers. The app never spits out recommendations or dosage advice on its own. We set it up this way to keep things ethical and lower the risk of someone using it the wrong way.

6. Technical Limitations

Right now, there are a bunch of things the app doesn't do:

- The interaction dataset is pretty limited and nowhere near complete.
- There's no way to sort interactions by severity.
- It doesn't consider individual factors like age, dosage, or health conditions.
- Nobody's validated it externally, and it's not approved by any regulators.

We left these gaps on purpose and want users to know about them.

7. Future Improvements

Here's what could come next:

- Add severity ratings for drug interactions.
- Normalize drug names without worrying about capitalization.
- Hook up to external drug interaction APIs.
- Store data in a database that sticks around.
- Bring in unit tests and continuous integration.
- Give the UI more features, like expandable details for each interaction.

The way the app's built now makes it easy to add these features later without overhauling everything.

8. Conclusion

The Drug Interaction Checker shows how you can build a system that's modular, readable, and efficient with Python and Streamlit—tackling a real healthcare challenge, at least in theory.

By sticking to clean architecture, simple data, and transparent algorithms, the project hits its educational targets and leaves plenty of room to grow. It’s not meant for actual clinical use, but it’s a solid place to start if you want to dig into medical informatics, decision-support tools, or healthcare software engineering.