Module 5: Software Assurance & Security Report (With AI)

1. Installation and To Reproduce Environment

The application is packaged to ensure consistency across environments. Using a **setup.py** file we can make a formal installation and reproduce a reliable module 5.

Using uv (Recommended)

uv provides excellent software assurance through the “force synchronization”, This ensures that the environment is reproduced flawlessly and matches the specification exactly and there are no missed configurations.

1. uv venv
2. source .venv/bin/activate (or .\.venv\Scripts\activate on Windows)
3. uv pip sync requirements.txt
4. pip install -e .

Using pip

1. python -m venv .venv
2. source .venv/bin/activate (or .\.venv\Scripts\activate on Windows)
3. pip install -r requirements.txt
4. pip install -e .

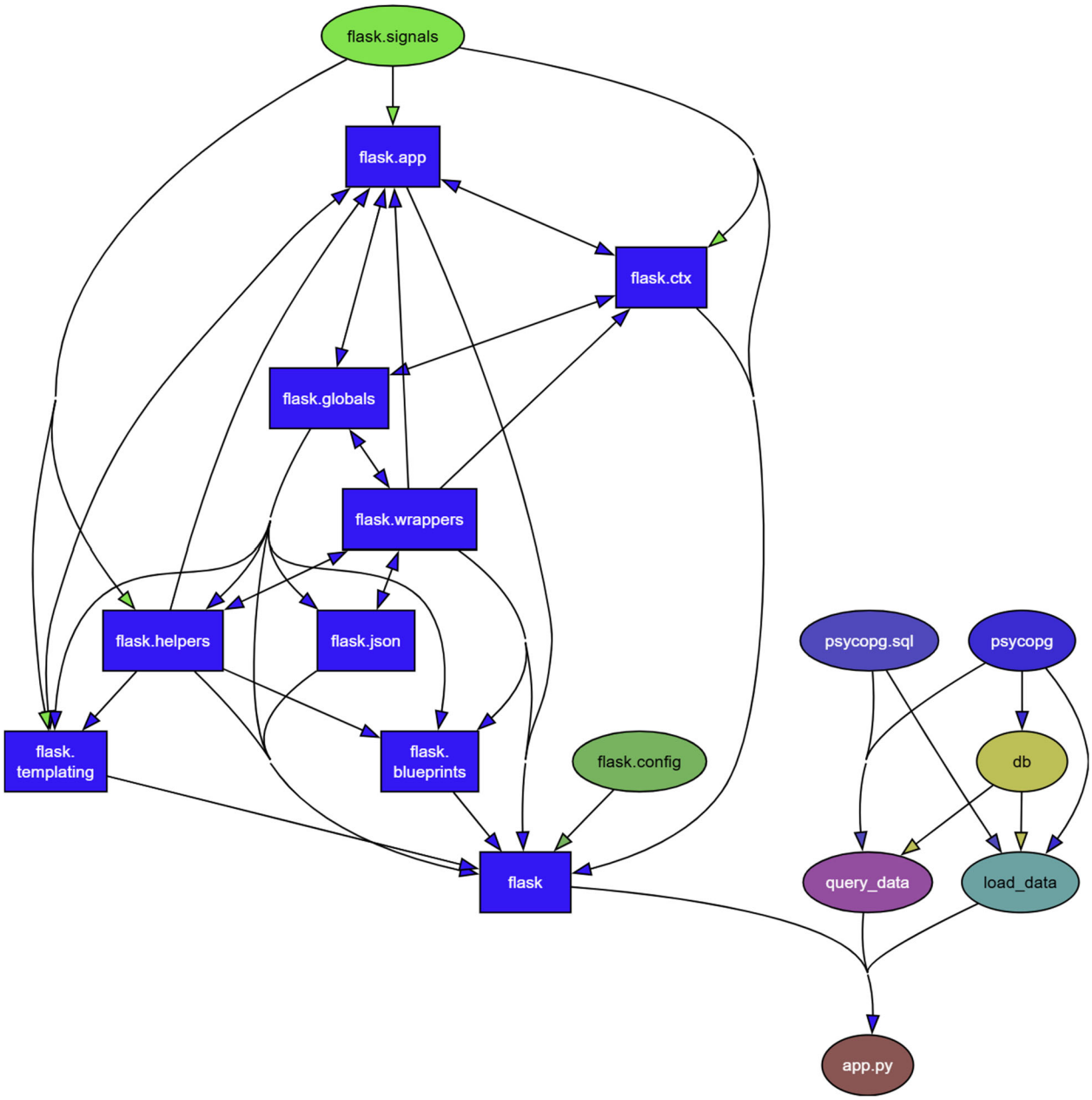
---

2. Dependency Graph Summary

The dependency graph depicts a modular Three Tier application architecture with the app.py module at its core, since it is the presentation and routing hub for my Flask app. As an entry point into the overall application flow, app.py coordinates all high-level flows of control by calling upon query_data.py for analytical outputs and load_data.py for populating the database with data. The dependency graph illustrates that query_data.py serves as a critical logic intermediary between the app.py module and the low-level database access provided through the db.py module. Additionally, since dependencies on the external libraries required for db.py's operations, such as psycopg and python-dotenv, are isolated from one another; the database connection logic and the sensitive secret management necessary to connect securely to the database are isolated from the remainder of the application. The visualization also illustrates a strict one way flow of dependencies, thus preventing circular dependencies from forming and ensuring that the scraper (scrape.py) and data loaders remain decoupled from the web front-end application. This structure shows a clear separation of concerns and will help to reduce

code complexity and improve the overall security posture of an application by isolating sensitive database operations from the rest of the app. Lastly, generate_answers_pdf.py demonstrates another layer of utility dependence on the reportlab library for creating printable documents from the processed data.

SCREENSHOT1: Dependency Graph (dependency.svg)

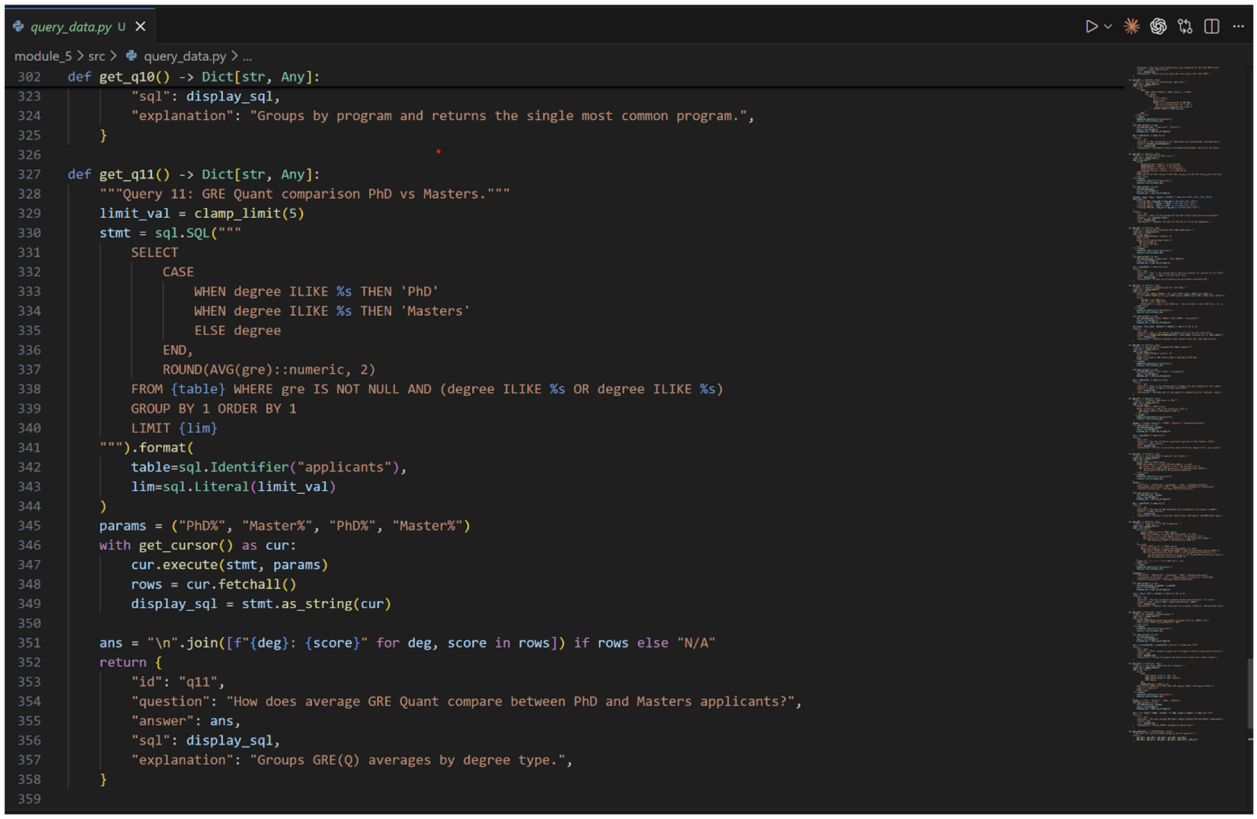


3. SQL Injection Defenses

To protect the database from injection attacks, the application logic was rewritten to move away from string formatting and toward safe query composition.

- **Safe Composition:** I use psycopg.sql to build queries. Table names and identifiers are wrapped in sql.Identifier, and static values are wrapped in sql.Literal.
- **Separation of Concerns:** SQL statements are defined as objects separately from their execution.
- **Parameterization:** All user provided data or variables are passed as secondary arguments to the execute() method, using %s placeholders. The database driver handles the escaping, making it impossible for a malicious string to be interpreted as a command.

SCREENSHOT2: Code snippet showing psycopg.sql implementation

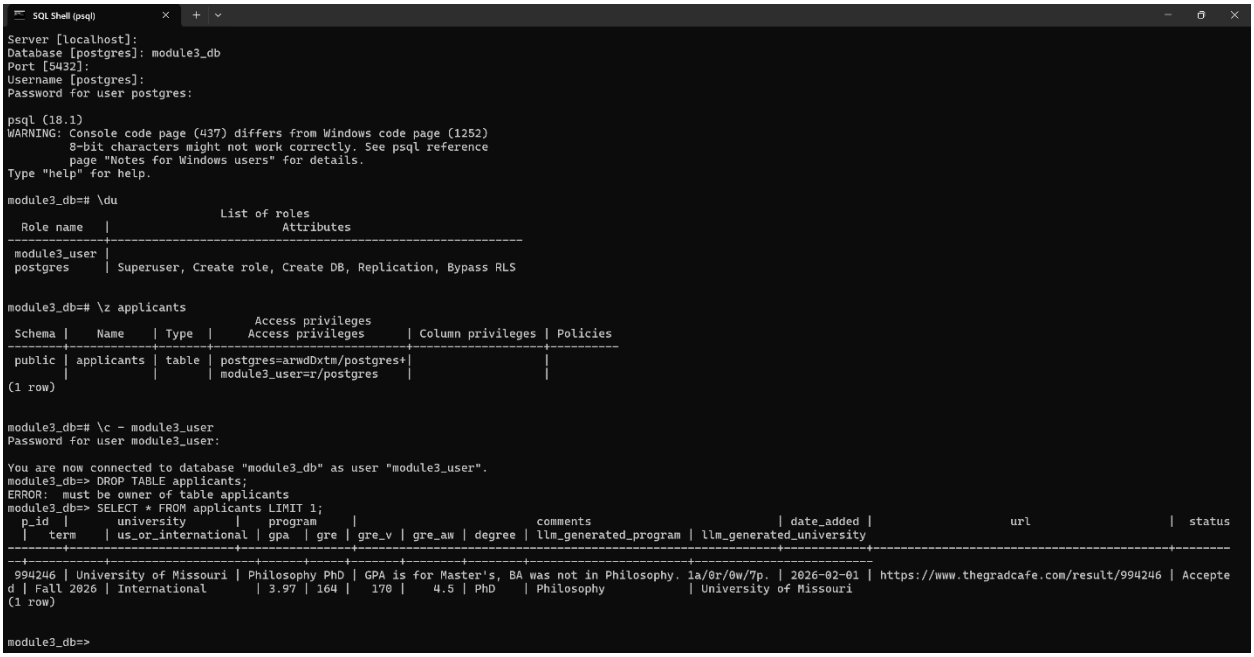


4. Database Hardening & Least Privilege

The database has been secured by applying the "Principle of Least Privilege" to the module3_user account. This ensures a "Default Deny" environment where the application can only perform the actions necessary for its function.

- **Restricted Attributes:** The user account was stripped of administrative powers (NOCREATEDB, NOCREATEROLE).
- **Default Deny:** I revoked all permissions from the PUBLIC role on the database and schema to prevent "ghost permissions" from leaking through.
- **Ownership Transfer:** I transferred ownership of the applicants table to the postgres superuser. This is a critical security boundary: in PostgreSQL, only the owner or a superuser can DROP or TRUNCATE a table.
- **Selective Access:** The module3_user was granted only CONNECT to the database and SELECT on the table.

SCREENSHOT3: Terminal output showing "ERROR: must be owner of table applicants" when trying to drop a table as module3_user



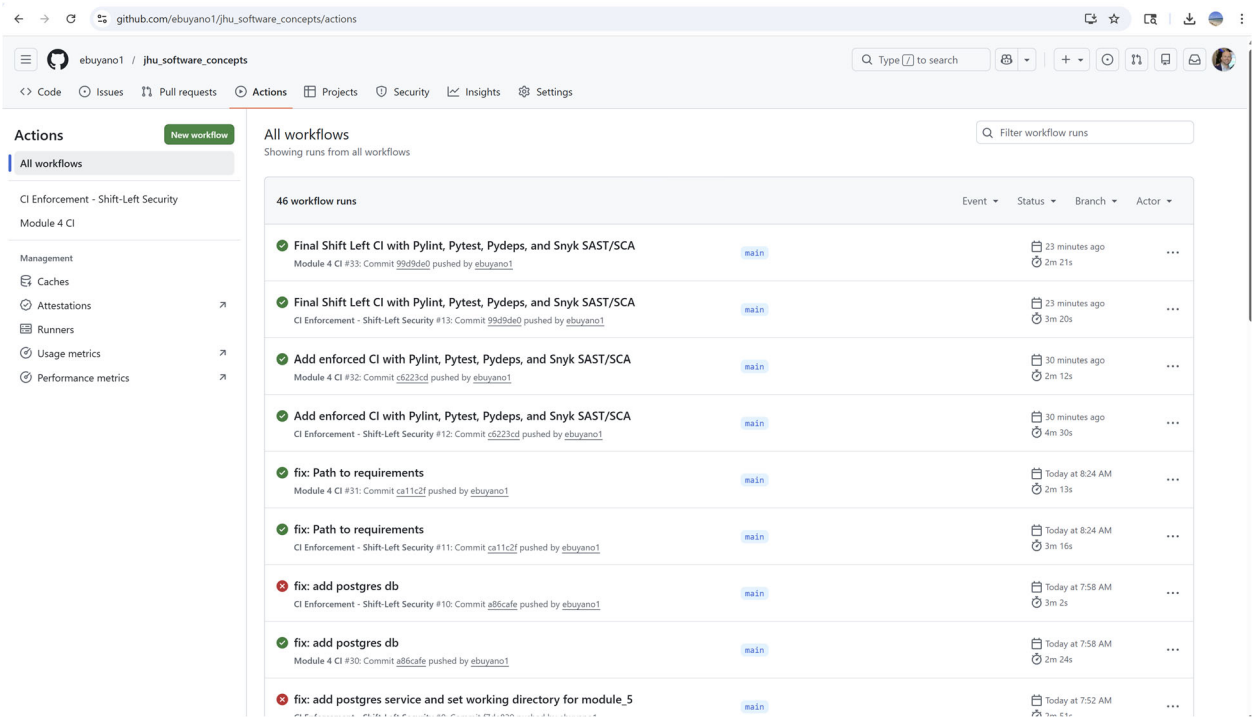
5. Requirements Met for SQL

Requirement	Implementation Detail
LIMIT Enforced	Every query in query_data.py uses a clamp_limit() function to ensure results never exceed 100 rows.
Separated Execution	Queries are defined as sql.SQL objects before being passed to cur.execute().
Safe Parameterization	Placeholder %s syntax is used for all variable data.

6. CI Enforcement with GitHub Actions

I implemented a "Shift-Left" security strategy using GitHub Actions. The build fails if quality or security gates are not met.

SCREENSHOT4: GitHub Actions passing (Green Checkmarks)

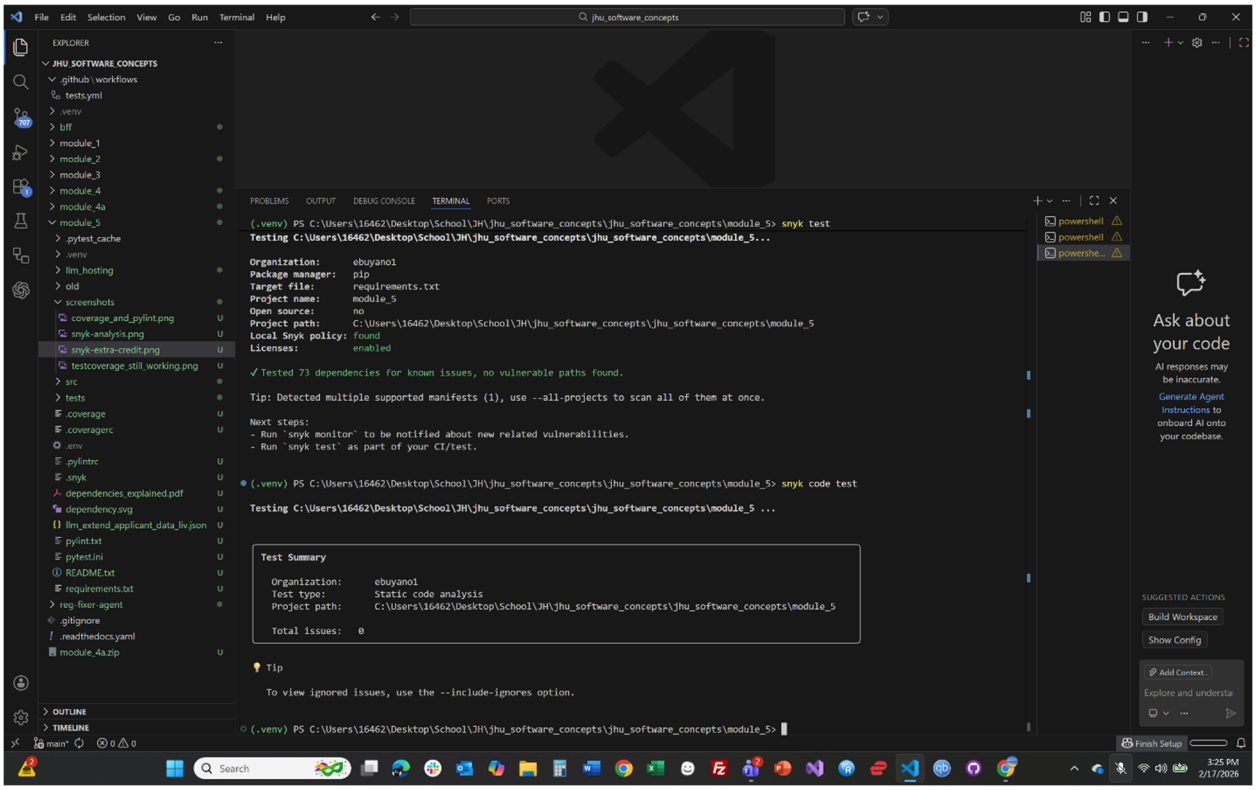


7. Extra Credit: Snyk Security Evidence

A Snyk scan was performed on the project dependencies. While some high-severity issues were found in sub-dependencies (like pillow), they were remediated by pinning pillow==12.1.1 in the requirements.txt. For issues with no direct patch (e.g., diskcache), a .snyk ignore policy was implemented with a documented rationale, ensuring the CI build remains secure and passing.

For the diskcache vulnerability (where no patch is available), I implemented a Snyk ignore policy with a 30-day expiry, ensuring the vulnerability is tracked but does not block the CI pipeline while awaiting a maintainer update.

SCREENSHOT5: Snyk test output showing "No vulnerable paths found"



8. Shift Left Security CI (ALL PASS)

I enforced **100% Code Coverage** using pytest-cov. Since our CI pipeline will literally fail if coverage drops below 100%, that's a massive software assurance achievement. The CI passes Pylint, Pytest, Snyk Test, Snyk Code Test, Generates dependencies svg . Github Actions shows all passed, Pylint, Pytest, Snyk Test, Snyk Code Test (extra credit), Dependency graph generated and can be downloaded for verification.