SQL injection defenses:

To defend against SQL injection, I refactored my database layer so that every SQL query uses **parameterized statements** instead of building SQL strings with concatenation or f-strings. The main change was in query_data.py, where the query functions (ex: q1–q9, extra_1, extra_2) were updated to pass dynamic values (like term, limits, and filter text) as **separate parameters** to the driver rather than embedding them directly into the SQL text. This matters because if user-controlled input is inserted into a query string, a malicious value could change the meaning of the SQL (for example by adding OR 1=1 or attempting to append additional statements). By switching to placeholders (such as %s) and supplying a parameter tuple/list, PostgreSQL treats the input as **data only**, not executable SQL, so the structure of the query cannot be altered.

My db_config.py supports this safety improvement by centralizing how the application connects to the database and ensuring credentials are not hard-coded anywhere in the source. db_config.py reads DB_HOST, DB_PORT, DB_NAME, DB_USER, and DB_PASSWORD from environment variables (optionally loading them locally using python-dotenv). This keeps secrets out of the codebase and also makes it easier to control which database user is being used (for example, the least-privilege gradcafe_app role). With connection settings in one place, all query functions use the same controlled connection logic, which reduces the chance of “quick hacks” like hard-coded credentials or unsafe ad-hoc connections being introduced later.

In query_data.py, I specifically changed any query that previously relied on dynamically constructed SQL to instead use safe parameter binding. For example, the term filter in q1 (count of Fall 2026 rows) is now passed as a query parameter rather than concatenated into the WHERE term = ... clause. Any “limit” values are clamped/validated in Python first and then passed as parameters instead of being injected into the SQL string, preventing someone from using the limit field to inject SQL. I also ensured that any string matching logic (such as program/university filters used in later questions) does not embed raw user text inside SQL; instead the query text stays constant and only the user-provided value is passed in separately. These changes make the system safe because even if an attacker supplies special characters, quotes, or SQL keywords, the driver escapes/binds them properly and PostgreSQL will not execute them as part of the SQL command.

Least-privilege DB configuration:
Permissions granted:

```
CREATE USER gradcafe_app WITH PASSWORD '********';

GRANT CONNECT ON DATABASE gradcafe TO gradcafe_app;
GRANT USAGE ON SCHEMA public TO gradcafe_app;

GRANT SELECT, INSERT ON TABLE public.applicants TO gradcafe_app;
GRANT USAGE, SELECT ON SEQUENCE public.applicants_p_id_seq TO gradcafe_app;
```

The database user gradcafe_app was created following the principle of least privilege, meaning it was granted only the minimum permissions required for the application to function. First, the user was granted CONNECT on the gradcafe database so it can establish a database session. Without this permission, the application would not be able to log in at all. This permission alone does not provide access to any data; it only allows connection to the database.

Next, the user was granted USAGE on the public schema. The application's applicants table resides within this schema, and without USAGE, the user would not be able to access objects inside it even if table-level permissions were granted. This permission allows referencing objects in the schema but does not allow creating, altering, or dropping schema objects.

At the table level, the user was granted SELECT on public.applicants because the Flask application runs multiple analytics queries that read data from this table. Without SELECT, the dashboard queries (such as counts, averages, and percentages) would fail. Additionally, the user was granted INSERT on public.applicants because the ETL pipeline loads newly scraped and cleaned records into the database. This allows the application to add new rows but does not permit updating or deleting existing data.

Since the applicants table uses a SERIAL primary key, PostgreSQL automatically creates a sequence (applicants_p_id_seq). To allow row insertion, the user was also granted USAGE and SELECT on this sequence so it can generate new primary key values during inserts. Without this permission, insert operations would fail due to insufficient sequence privileges.

Importantly, no elevated privileges were granted. The gradcafe_app user is not a superuser and does not have DROP, ALTER, UPDATE, DELETE, database creation, role creation, or ownership-level permissions. This ensures that even if the credentials were compromised, the user could not modify the schema, drop tables, escalate privileges, or damage the database structure. The permissions granted strictly align with the functional requirements of the application while minimizing security risk.

```
gradcafe=# SELECT grantee, privilege_type
gradcafe-# FROM information_schema.role_table_grants
gradcafe-# WHERE table_name = 'applicants'
gradcafe-# AND grantee = 'gradcafe_app';
   grantee    | privilege_type
--------------+----------------
 gradcafe_app | INSERT
 gradcafe_app | SELECT
(2 rows)
```

Dependency graph summary:

The dependency graph shows that **src.app** is the central module of the application and depends on several supporting modules within the src package. The src.scrape module is responsible for collecting raw Grad Cafe data, while src.clean processes and structures that scraped data before it is written to disk or loaded into the database. The src.load_data module handles inserting cleaned records into PostgreSQL, and src.query_data contains the SQL queries used to generate analytics results for the dashboard. Both src.load_data and src.query_data depend on src.db_config, which centralizes database connection logic and reads credentials from environment variables, ensuring secure and consistent database access. The arrows in the graph illustrate how data flows through the ETL pipeline (scrape -> clean -> load) and then into the analytics layer consumed by app.py. Because app.py uses relative imports inside the src package, pydeps must be run against the package rather than the file directly. This ensures the dependency graph correctly resolves internal module relationships without triggering import errors.

Step 5: Why packaging matters?

Adding a setup.py file makes the project installable as a Python package, which ensures consistent import behavior across development, testing, and CI environments. By using pip install -e ., the project is installed in editable mode, allowing changes to the source code to immediately reflect without reinstalling. This reduces common “works on my machine” path issues, especially when using relative imports inside the src package. Packaging also enables dependency tools like uv to synchronize environments accurately. Overall, setup.py improves reproducibility, portability, and maintainability of the project.

How to install/run via pip and uv (Screenshot taken from my README.md):

```
Fresh Install Instructions

To set up the project from scratch using pip, first create a virtual
environment using python -m venv .venv, then activate it (on Windows use
.venv\Scripts\activate, and on Mac/Linux use source .venv/bin/activate). Once
activated, upgrade pip and install all required dependencies using pip install
--upgrade pip followed by pip install -r requirements.txt. After installing
dependencies, install the project itself in editable mode using pip install -e
.. Editable installation ensures imports behave consistently across local
development, testing, and CI environments, preventing common path-related
issues.

To install using uv, create a virtual environment with uv venv, then
synchronize dependencies exactly as specified in requirements.txt using uv pip
sync requirements.txt. This guarantees that the environment matches the
requirements file precisely, improving reproducibility. After syncing, install
the package in editable mode using uv pip install -e ..
```

The application requires database configuration via environment variables. You may either define DB_HOST, DB_PORT, DB_NAME, DB_USER, and DB_PASSWORD, or provide a full DATABASE_URL. These values can be placed in a .env file (which should not be committed to version control). An example configuration file is included to demonstrate required variable names.

To verify correctness, run pytest to execute the full test suite. The project enforces 100% coverage and includes tests for SQL injection safety, database behavior, Flask routes, and end-to-end flows. To verify code quality, run pylint src, which should complete without errors and achieve a full lint score.

To run the web application, execute python -m src.app and open http://localhost:8080 in your browser. The interface allows you to trigger the ETL pipeline, update analysis results, and view formatted analytics output.

This project includes a setup.py file so it can be installed as a proper Python package. Packaging ensures consistent import behavior across environments, supports editable installs, reduces “works on my machine” issues, improves CI reliability, and allows tools such as uv to synchronize dependencies accurately. If a fresh install works using either pip or uv, pytest passes with 100% coverage, pylint passes cleanly, and the Flask app runs successfully, then Step 5 requirements are fully satisfied.