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.

The Importance of Packaging in Software Engineering Implementing a setup.py file takes a group of individual scripts and assembles them into a formal Python package, which is a fundamental practice in software assurance and to reproduce environment exactly. Packaging allows the project to be installed in "editable mode" (pip install -e .), which ensures that internal module imports work consistently across different environments, regardless of the user's current working directory. By explicitly defining dependencies and metadata, we eliminate the common problem of "it works on my machine", allowing automated tools like uv or CI/CD pipelines to recreate the exact environment needed for secure execution.

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.

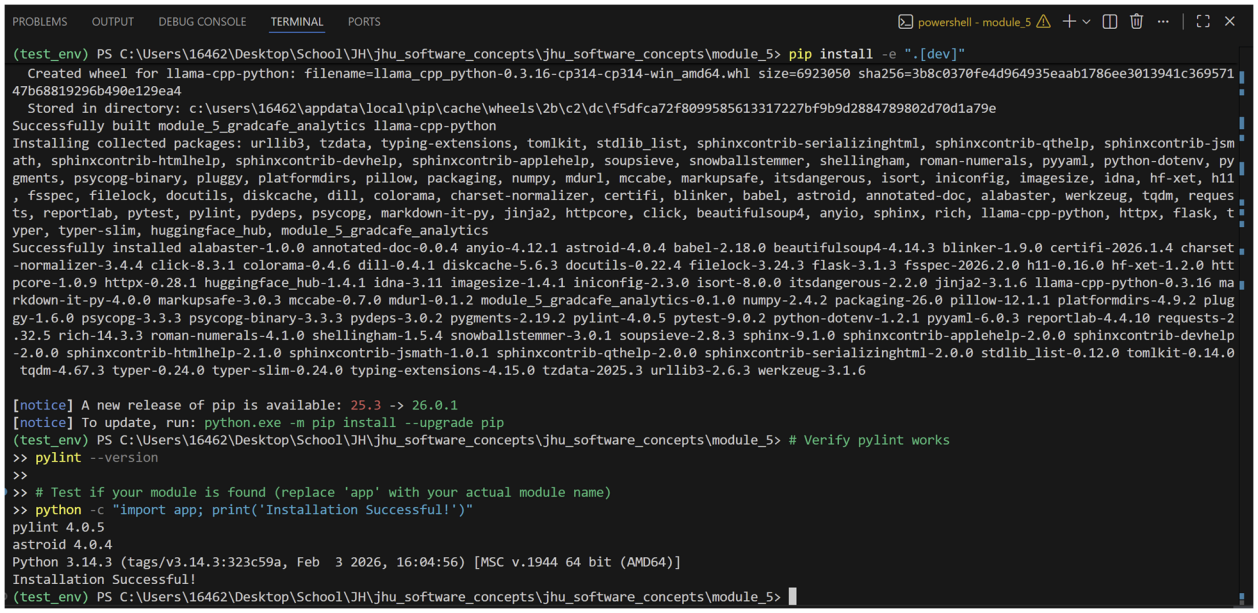
1. uv venv
2. source .venv/bin/activate (or .\.venv\Scripts\activate on Windows)
3. uv pip sync requirements.txt
4. pip install -e .

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 .

Tested the setup (Screenshot) in a test_env

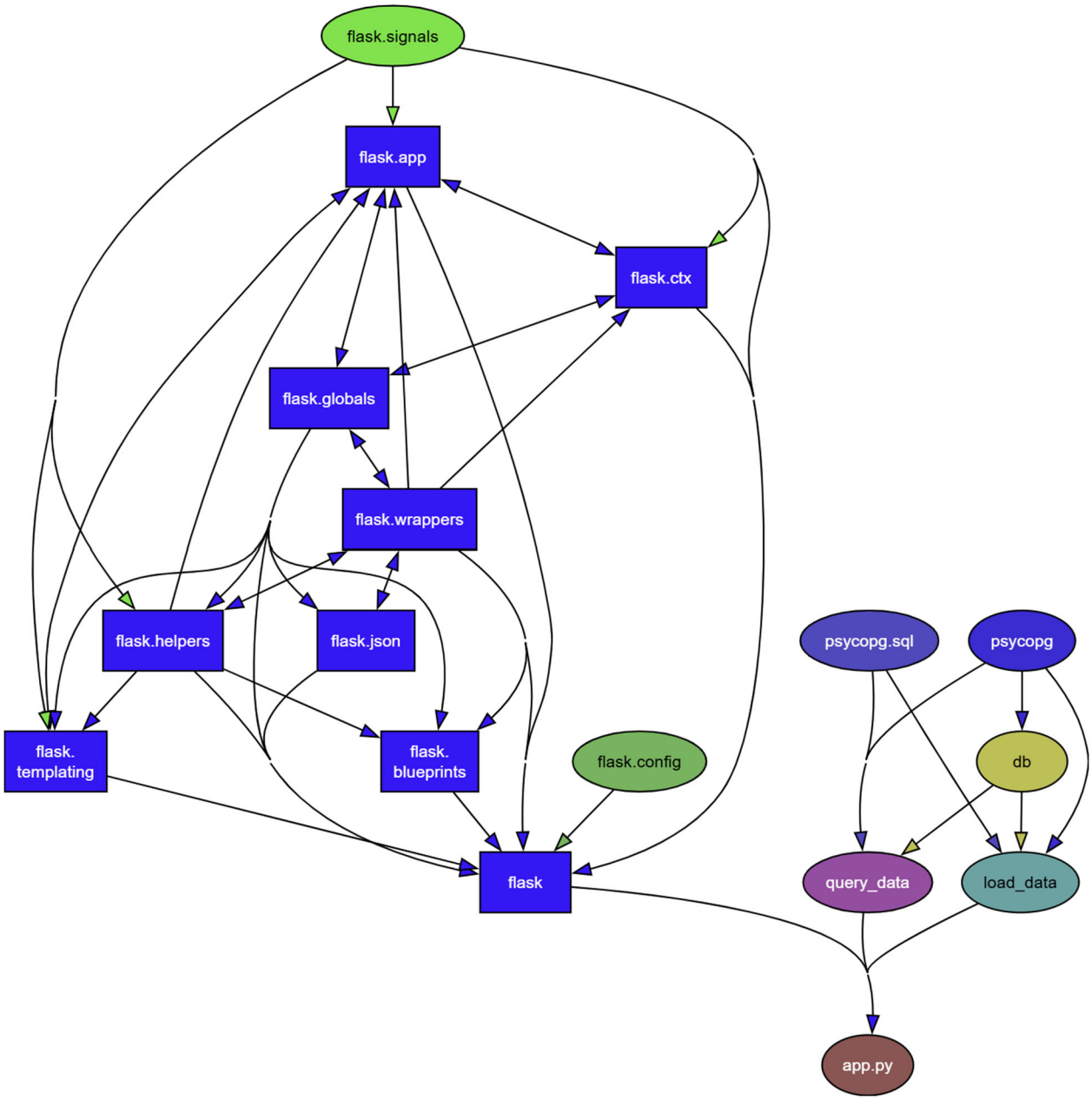
```
python -m venv test_env

.\test_env\Scripts\activate

pip install -e ".[dev]"
```



2. Dependency Graph Summary

I generated a dependency graph using pydeps src/app.py --noshow -T svg -o dependency.svg. 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)

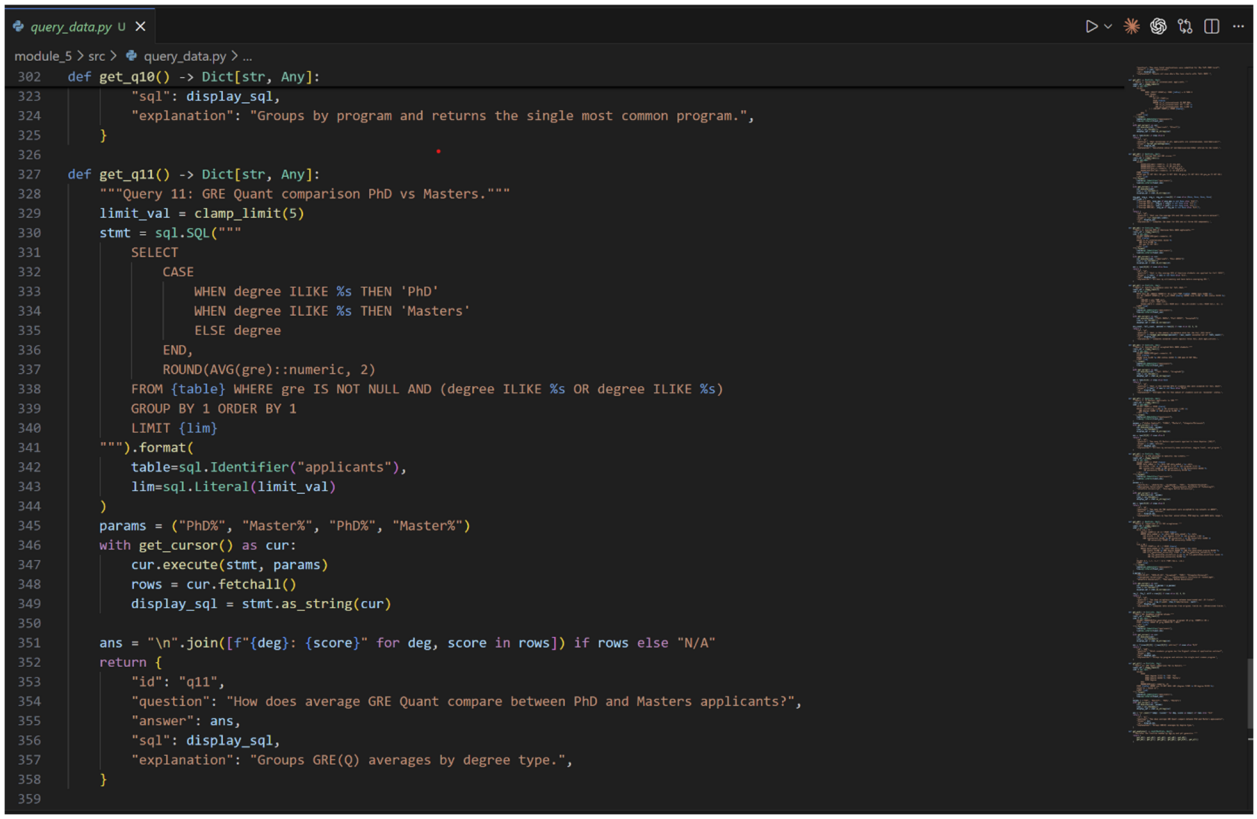

flask.signals
flask.app
flask.ctx
flask.globals
flask.wrappers
flask.helpers
flask.json
flask.
templating
flask.
blueprints
flask.config
flask
psycopg.sql
psycopg
db
query_data
load_data
app.py

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



```
query_data.py U
module_5 > src > query_data.py > ...
302  def get_q10() -> Dict[str, Any]:
323          "sql": display_sql,
324          "explanation": "Groups by program and returns the single most common program.",
325      }
326
327  def get_q11() -> Dict[str, Any]:
328      """Query 11: GRE Quant comparison PhD vs Masters."""
329      limit_val = clamp_limit(5)
330      stmt = sql.SQL("""
331          SELECT
332              CASE
333                  WHEN degree ILIKE %s THEN 'PhD'
334                  WHEN degree ILIKE %s THEN 'Masters'
335                  ELSE degree
336              END,
337              ROUND(AVG(gre)::numeric, 2)
338          FROM {table} WHERE gre IS NOT NULL AND (degree ILIKE %s OR degree ILIKE %s)
339          GROUP BY 1 ORDER BY 1
340          LIMIT {lim}
341      """).format(
342          table=sql.Identifier("applicants"),
343          lim=sql.Literal(limit_val)
344      )
345      params = ("PhD%", "Master%", "PhD%", "Master%")
346      with get_cursor() as cur:
347          cur.execute(stmt, params)
348          rows = cur.fetchall()
349          display_sql = stmt.as_string(cur)
350
351      ans = "\n".join([f"{deg}: {score}" for deg, score in rows]) if rows else "N/A"
352      return {
353          "id": "q11",
354          "question": "How does average GRE Quant compare between PhD and Masters applicants?",
355          "answer": ans,
356          "sql": display_sql,
357          "explanation": "Groups GRE(Q) averages by degree type.",
358      }
359
```

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.

The Hardening Process (Step-by-Step)

1. **Stripping Administrative Attributes**: Used ALTER ROLE to set account powers to NOCREATEDB and NOCREATEROLE, ensuring compromised credentials cannot be used to bypass security or delete the database.
2. **Implementing "Zero Trust" and Fixing Ownership**: Realized the user could still DROP the table as the default owner; transferred ownership to the postgres admin to close this loophole and revoked all permissions from the PUBLIC role.
3. **Final Lockdown**: Restricted the user to a "Default Deny" environment where they have only CONNECT privileges on the database and SELECT privileges on specific tables.

Hardening Commands Executed:

-- Prevent instance leaks

REVOKE ALL ON SCHEMA public FROM PUBLIC;

REVOKE ALL ON DATABASE module3_db FROM PUBLIC;

-- Lock down the schema

ALTER SCHEMA public OWNER TO postgres;

GRANT USAGE ON SCHEMA public TO module3_user;

-- Secure the actual table data

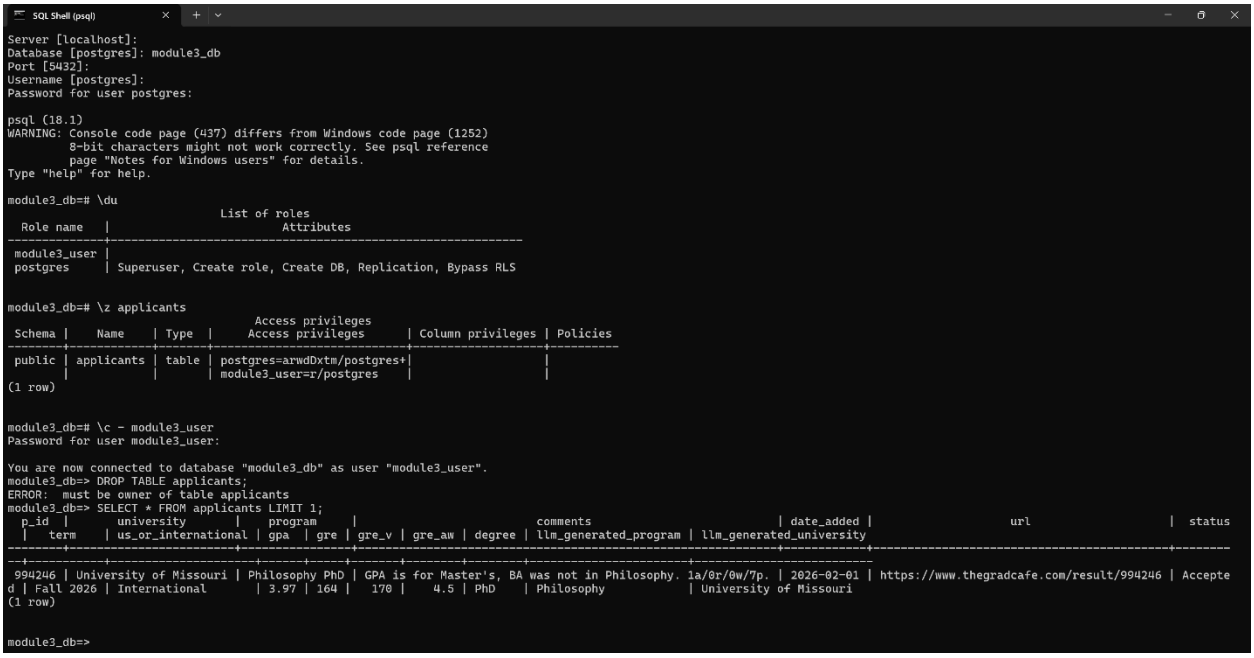
ALTER TABLE applicants OWNER TO postgres;

REVOKE ALL ON TABLE applicants FROM module3_user;

GRANT SELECT ON TABLE applicants TO module3_user;

- **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



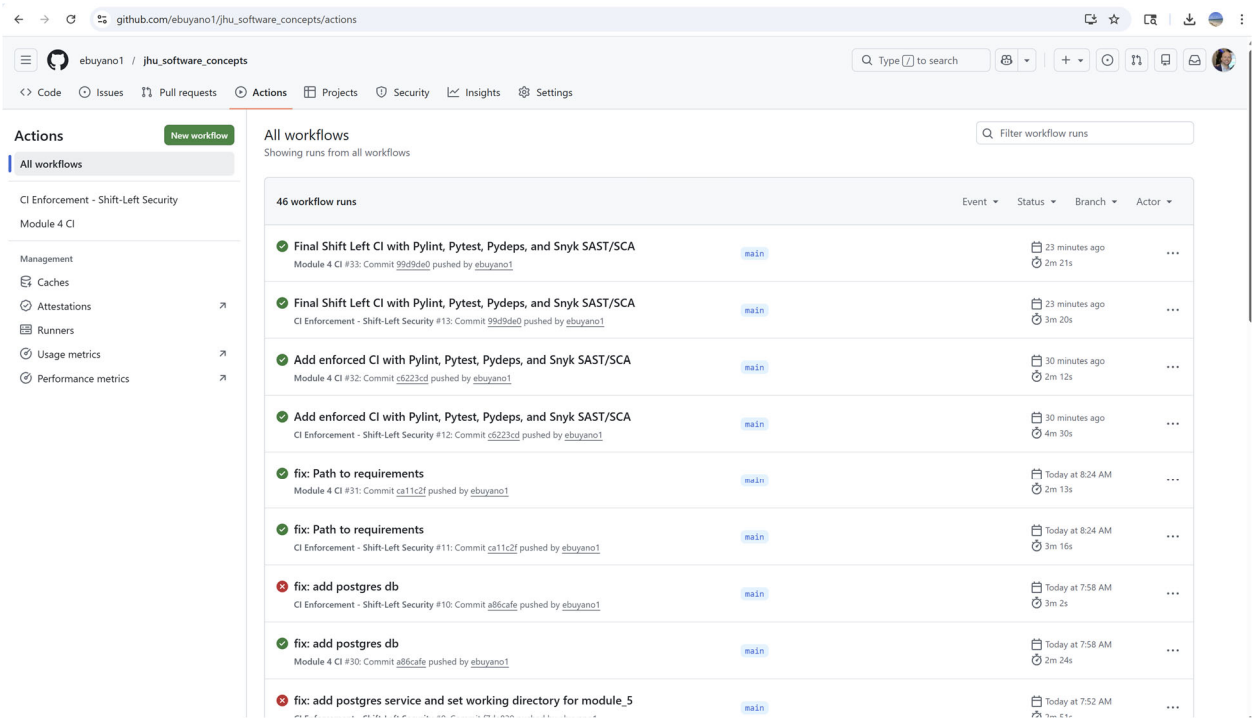
5. Requirements Met for SQL

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)

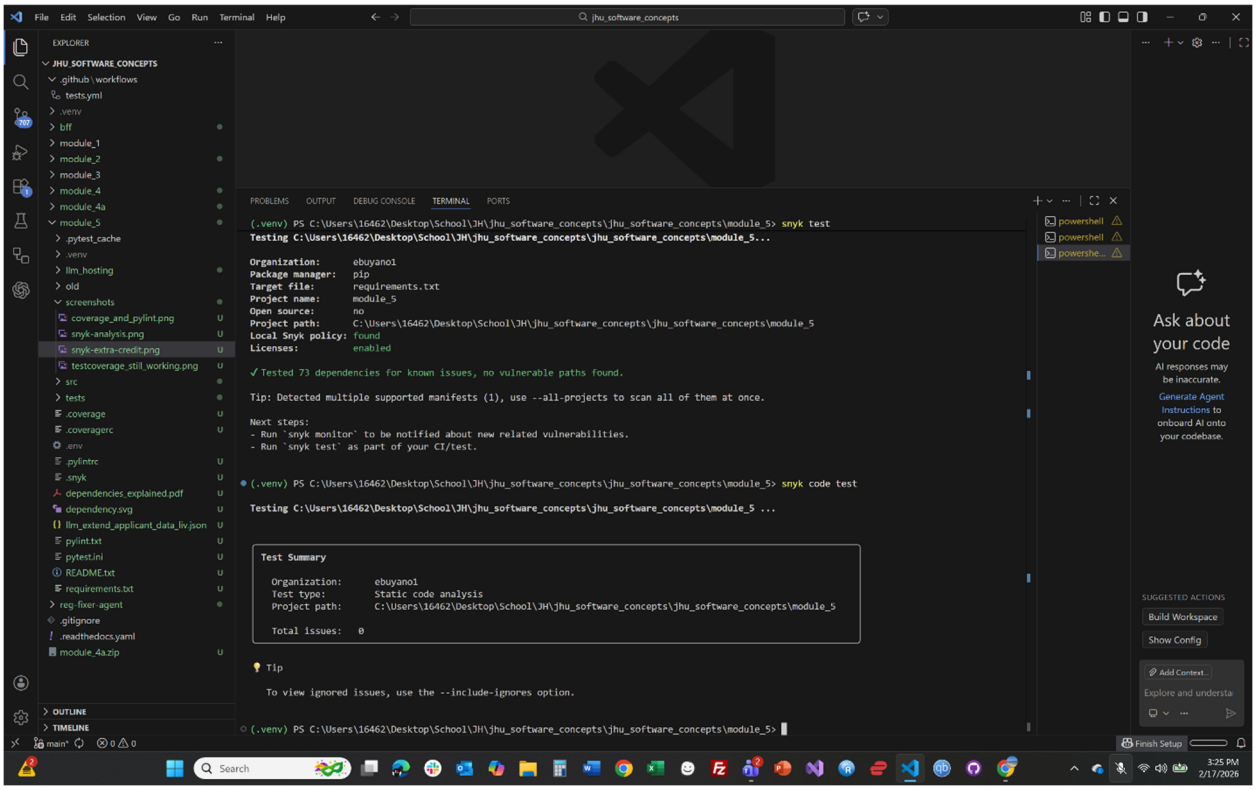

github.com/ebuyano1/jhu_software_concepts/actions
ebuyano1 / jhu_software_concepts
Type / to search
Code
Issues
Pull requests
Actions
Projects
Security
Insights
Settings
Actions
New workflow
All workflows
CI Enforcement - Shift-Left Security
Module 4 CI
Management
Caches
Attestations
Runners
Usage metrics
Performance metrics
All workflows
Showing runs from all workflows
Filter workflow runs
46 workflow runs
Event
Status
Branch
Actor
Final Shift Left CI with Pylint, Pytest, Pydeps, and Snyk SAST/SCA
Module 4 CI #33: Commit 99d9de0 pushed by ebuyano1
main
23 minutes ago
2m 21s
Final Shift Left CI with Pylint, Pytest, Pydeps, and Snyk SAST/SCA
CI Enforcement - Shift-Left Security #13: Commit 99d9de0 pushed by ebuyano1
main
23 minutes ago
3m 20s
Add enforced CI with Pylint, Pytest, Pydeps, and Snyk SAST/SCA
Module 4 CI #32: Commit c6223cd pushed by ebuyano1
main
30 minutes ago
2m 12s
Add enforced CI with Pylint, Pytest, Pydeps, and Snyk SAST/SCA
CI Enforcement - Shift-Left Security #12: Commit c6223cd pushed by ebuyano1
main
30 minutes ago
4m 30s
fix: Path to requirements
Module 4 CI #31: Commit ca11c2f pushed by ebuyano1
main
Today at 8:24 AM
2m 13s
fix: Path to requirements
CI Enforcement - Shift-Left Security #11: Commit ca11c2f pushed by ebuyano1
main
Today at 8:24 AM
3m 16s
fix: add postgres db
CI Enforcement - Shift-Left Security #10: Commit a86cafe pushed by ebuyano1
main
Today at 7:58 AM
3m 2s
fix: add postgres db
Module 4 CI #30: Commit a86cafe pushed by ebuyano1
main
Today at 7:58 AM
2m 24s
fix: add postgres service and set working directory for module_5
main
Today at 7:52 AM

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.