- How to install/run via **pip** and **uv**
  - **pip**
    - cd jhu_software_concepts/module_5
    - python -m venv .venv
    - source .venv/bin/activate
    - pip install --upgrade pip
    - pip install -r requirements.txt
    - pip install -e .
    - Ensure ../.env exists (copy from .env.example and set DB values).
    - python src/run.py
    - Open http://localhost:8080
  - **uv**
    - cd jhu_software_concepts/module_5
    - uv venv
    - source .venv/bin/activate
    - uv pip sync requirements.txt (exactly matches requirements.txt)
    - uv pip install -e .
    - Ensure ../.env exists and DB is configured.
    - python run.py
    - Open http://localhost:8080
  - **Key difference**
    - If using the pip method (pip install -r requirements.txt), then any missing packages will be installed, but there may be extra packages already in the environment, which means the environment would not be an exact reproduction of the specified runtime environment
    - Using the uv method (uv pip sync requirements.txt) ensures that the environment matches for specified runtime environment exactly for better reproducibility
  - **Importance of setup.py**
    - Including setup.py defines the project as an installable Python package, which allows you to run 'pip install –e .' or 'uv pip install –e .' so that all imports behave as expected when reproducing the environment.

- Dependency graph summary (5–7 sentences)
  - The only external packages required by my entire pipeline are bs4, flask, psycopg, psycopg.sql, and huggingface_hub. 'src/scrape.py' requires bs4 for parsing html content from TheGradCafe. 'src/load_data.py' and 'src/query_data.py' both require psycopg and psycopg.sql for connecting to the postgresql database to populate it with data and run analysis queries. 'src/llm_hosting/app.py' requires llama_cpp and huggingface_hub to serve the local LLM for standardizing the scraped inputs. 'lrc/llm_hosting/app.py',

‘src/board/pages.py’, and ‘src/board/__init__.py’ all require flask for running flask web pages.

- Your SQL injection defenses (what changed and why it’s safe)
- **What changed**
  - **Dynamic SQL construction**
    - BEFORE: load_data.py used raw f-string for database creation
      - cur.execute(f'CREATE DATABASE {dbname}')
    - AFTER: load_data.py uses psycopg composition with identifier quoting
      - sql.SQL('CREATE DATABASE {db_name}').format(db_name=sql.Identifier(DBNAME))
  - **Query execution interface**
    - BEFORE: query_data.py execute_query(query) accepted raw SQL strings and executed directly.
    - AFTER: query_data.py execute_query(query, params, limit) requires psycopg.sql.Composable, rejects raw strings, and executes with bound params.
  - **LIMIT enforcement**
    - BEFORE: no central hard cap.
    - AFTER: all analysis queries go through a composed wrapper with a clamped limit (1..100) in execute_query().
  - **Placeholder-safe pattern handling**
    - query_data.py: static SQL builder escapes literal % (for ILIKE '%...%') before execution with params, avoiding placeholder confusion and accidental interpolation behavior.
  - **Read-query bounds in loader**
    - BEFORE: load_data.py had an unbounded ‘SELECT url FROM admissions;’
    - AFTER: paginated bounded fetch with LIMIT/OFFSET and cap logic (MAX_QUERY_LIMIT).
  - **Least-privilege schema provisioning path**
    - BEFORE: app always attempted CREATE TABLE/INDEX.
    - AFTER: app checks table existence; if missing, provisioning runs through admin path and grants only the required rights
  - **Why these changes are safer**
    - Dynamic identifiers are quoted safely (sql.Identifier), not string-concatenated.
    - Values are now passed as bound parameters instead of being injected into SQL text.
    - Raw string SQL is blocked by the API contract in execute_query().
    - Queries are bounded by enforced max LIMIT, reducing accidental large-result exposure/DoS risk.

        - Least-privilege DB model reduces impact even if app credentials are compromised.

- Least-privilege DB configuration (what permissions and why)
    - Role:
        - Dedicated app login role grad_app (NOSUPERUSER NOCREATEDB NOCREATEROLE NOINHERIT NOREPLICATION)
          Why: The app can authenticate, but cannot administer server, create DBs/roles, or escalate privileges.
    - Database-level grant:
        - GRANT CONNECT ON DATABASE grad_data TO grad_app;
          Why: Allows connection only to the target DB.
    - Schema-level grant (inside grad_data):
        - GRANT USAGE ON SCHEMA public TO grad_app;
          Why: It lets my app access objects in public without giving ownership/admin rights.
    - Table-level grants:
        - GRANT SELECT, INSERT, UPDATE ON TABLE public.admissions TO grad_app;
          Why: Only grant permission for actions that the analysis page needs to run (reads, inserts, and updates); no DELETE, ALTER, or DROP.
    - Sequence grants:
        - GRANT USAGE, SELECT ON SEQUENCE public.admissions_p_id_seq TO grad_app;
          Why: Required for SERIAL/ID generation and reading sequence values during inserts.
    - Explicitly omitted:
        - SUPERUSER, CREATEDB, CREATEROLE, and object DROP/ALTER privileges.
          Why: Limits the potential for damage if app credentials are exposed.