Generative AI Use Reflection

As we have said in our syllabus, the habits you adopt in using generative models are yours to choose, and you should strive to work with these technologies responsibly as a study aid, not a fast-track to an answer. Generative AI technologies possess powerful capabilities that will facilitate not only your coursework now but activities in your professional work later on. There is a nascent but active community of researchers studying how AI tools can be used to help student education, and this reflection is based on key insights from that community.

The purpose of this reflection is to promote beneficial usage of generative AI and to encourage you to leverage those tools to aid in your learning process. Generative AI is ubiquitous, and we encourage you to use these tools to improve your learning process. The intended outcome of working through these questions is a focus on your learning process, rather than your learning output.

If you used any generative AI tool (e.g., ChatGPT, Copilot, Gemini) at any point in an assignment, you must complete the following questions for that assignment. As you complete this reflection, discuss in your own words, and do not use generative AI.

Question 1. Structured self-reflection as a part of work involving generative AI usage improves transparency for instructors and supports students in that they tend to rely less on generated content.[1]

(a) What generative AI tool(s) did you use? Provide the name(s) and model version(s).

Generative AI Tools: Chatgpt

Model Version: GPT 5.1

(b) Include the dialogues with those tools as screenshots attached to this reflection submission.

Dialogue 1:

I have a hugging face space, how do i push my streamlit frontend code to it?

1. Prerequisites

Make sure you have:

- A Hugging Face account
- `git` installed locally
- Your Streamlit app files (e.g., `app.py`, `requirements.txt`, etc.)

Optional but recommended:

```bash
pip install huggingface_hub
```

Then log in:

```bash
huggingface-cli login
```

2. Create the Space

You can either:

- Go to https://huggingface.co/spaces/new,
- Name your space (e.g., `username/streamlit-demo`),
- Set **SDK** to `Streamlit`,
- Choose **Python** as the runtime,
- Click **Create Space**.

This will create a Git repository for your space (like `https://huggingface.co/spaces/username/streamlit-demo`).

3. Clone the Space Repo

Once created, clone it to your machine:

```bash
git clone https://huggingface.co/spaces/username/streamlit-demo
cd streamlit-demo
```

4. Add Your Streamlit App Code

Copy your Streamlit files (from your local project) into this directory.
At minimum, you need:

```
app.py
requirements.txt
```

Example `requirements.txt`:

```
streamlit
pandas
numpy
```

If your entry file isn't named `app.py`, you **must** include a file named `app.py` or add a `README.md` with:

```yaml
---
title: My Streamlit App
sdk: streamlit
app_file: your_script_name.py
---
```

5. Commit and Push to Hugging Face

```bash
git add .
git commit -m "Initial Streamlit app push"
git push
```

After the push, Hugging Face will automatically **build** and **launch** your Space in about 1–2 minutes.
You can watch the logs in the "**Logs**" tab on your Space page if something fails.

[1] Combrinck, C., Loubser, N. Student self-reflection as a tool for managing GenAI use in large class assessment. Discov Educ 4, 72 (2025). https://doi.org/10.1007/s44217-025-00461-2

Dialogue 2:

how to see if im on the hugging face git or the github git

🕵️ 1. Check what repo you're currently in

From your terminal, run:

```bash
git remote -v
```

You'll see output like one of these:

✅ **Example (Hugging Face repo)**

```ruby
origin  https://huggingface.co/spaces/dcorcoran/deployment_test (fetch)
origin  https://huggingface.co/spaces/dcorcoran/deployment_test (push)
```

That means your commits will go to **Hugging Face Spaces**.

✅ **Example (GitHub repo)**

```perl
origin  https://github.com/yourname/your-repo.git (fetch)
origin  https://github.com/yourname/your-repo.git (push)
```

That means you're in a **GitHub** repo.

🧭 2. See which folder you're in

You can also confirm with:

```bash
git status
```

If it shows tracked files and a branch (like `main`), then you're inside a Git repository.
If it says:

```python
fatal: not a git repository (or any of the parent directories): .git
```

Dialogue 3:

how can i access the UI to see health and all endpoints?

1. Health check

If you added the optional root endpoint (`/`) like this:

```python
@app.get("/")
def root():
    return {"status": "ok", "message": "Render backend is running!"}
```

Just open in your browser:

```cpp
https://YOUR-RENDER-URL/
```

You should see something like:

```json
{"status": "ok", "message": "Render backend is running!"}
```

---

2. FastAPI interactive docs

FastAPI automatically provides **two UI docs**:

Endpoint	Description
`/docs`	Swagger UI (interactive, click endpoints, send test requests)
`/redoc`	ReDoc (more detailed API reference)

So if your Render backend URL is:

```arduino
https://deployment-testing-0v1x.onrender.com
```

You can access:

```bash
https://deployment-testing-0v1x.onrender.com/docs
https://deployment-testing-0v1x.onrender.com/redoc
```

On `/docs` you can:

- See all endpoints (`GET /`, `POST /search`, etc.)
- Click **"Try it out"** for `/search`
- Hit **Execute** → see the JSON response right in the browser

---

3. Testing `/search` without UI

If you want to test from terminal:

```bash
curl -X POST "https://deployment-testing-0v1x.onrender.com/search"
```

You should get a random city JSON response.

If you want, I can give a **quick tip to make the `/search` endpoint show example responses in the Swagger UI** so you can see the random city result structure directly. That's really handy for testing before connecting Streamlit.

Dialogue 4:

is it bad to inlcude the bm25 code in main.py?

Yes — it's **not ideal** to put your full BM25 implementation directly in `main.py`. Here's why and what to do instead.

Why it's not recommended

1. **Separation of concerns**
   `main.py` should only handle:
   - FastAPI routes
   - Request validation
   - Orchestrating retrieval / feature calls

   All the logic for scoring, indexing, or BM25 math is better placed in a dedicated module.
2. **Readability / Maintainability**
   A large BM25 implementation will make `main.py` long and hard to navigate.
   Future changes to BM25 (tuning `k1`, `b`, adding IDF weighting) will require editing `main.py` unnecessarily.
3. **Testing / Prototyping**
   If BM25 is in a separate file:
   - You can write standalone scripts to test it (`python -m src.features.bm25_baseline`)
   - You can run unit tests easily without spinning up FastAPI
4. **Reusability**
   - You might want to reuse BM25 in other scripts, CLI tools, or notebooks.
   - If it's in `main.py`, you'd have to import the whole API just to access it.

Recommended structure

css — Copy code

```
backend/
└── src/
    ├── api/
    │   main.py       ← FastAPI endpoints only
    └── features/
        bm25_baseline.py  ← BM25 scoring logic, indexing, utility functions
```

Then in `main.py`:

python — Copy code

```python
from src.features.bm25_baseline import BM25, rank_documents

@app.post("/search")
def search(req: SearchRequest):
    ...
    # Use BM25 here
    scores = rank_documents(query=req.query, corpus=documents)
```

This keeps `main.py` clean and your BM25 logic modular.

TL;DR

- **Bad:** Put BM25 code directly in `main.py` — messy, hard to test, hard to maintain.
- **Good:** Keep BM25 in `features/bm25_baseline.py` and import it in `main.py`.

If you want, I can draft a **clean skeleton for integrating BM25 into your FastAPI backend** without bloating `main.py`. This would show exactly where to call it and how to handle scoring + filtering.

Dialogue 5:

d\bm25> python bm25_package.py "small coastal towns in Spain with artisan markets" 25
Traceback (most recent call last):
  File "C:\Users\DCorc\OneDrive\Documents\Georgetown\4-Fall-2025\DSAN_6700_ML_App_Deployment\Project\dsan6700_app_dev_project\backend\bm25\bm25_package.py", line 7, in <module>
    from data_collection.read_blogs_db import posts
ModuleNotFoundError: No module named 'data_collection'

This error happens because Python can't find your `data_collection` package. On Windows/PowerShell, you need to make sure the Python path includes the directory that contains `data_collection`. Here's how to fix it:

Option 1: Run from the project root

Assuming your project structure is something like:

Copy code

```
project_root/
├─ backend/
│  ├─ bm25/
│  │  ├─ bm25_package.py
├─ data_collection/
│  ├─ read_blogs_db.py
```

Run the script **from the root folder** (`project_root`) like this:

powershell — Copy code

```
cd C:\Users\DCorc\OneDrive\Documents\Georgetown\4-Fall-2025\DSAN_6700_ML_App_Deployment\Project\ds
python backend\bm25\bm25_package.py "small coastal towns in Spain with artisan markets" 25
```

Python will now see `data_collection` as a top-level package.

Option 2: Add the root to `PYTHONPATH`

You can tell Python where to look by prepending the project root to `PYTHONPATH`:

powershell — Copy code

```
$env:PYTHONPATH = "C:\Users\DCorc\OneDrive\Documents\Georgetown\4-Fall-2025\DSAN_6700_ML_App_Deplo
python backend\bm25\bm25_package.py "small coastal towns in Spain with artisan markets" 25
```

Option 3: Modify `sys.path` inside the script

At the top of `bm25_package.py`, add:

python — Copy code

```python
import sys
import os
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "../../")))
```

This tells Python to look two levels up (to the project root) for packages like `data_collection`.

💡 **Recommendation:** Option 1 is the simplest for testing scripts locally. Option 2 is better if you want to run scripts from anywhere.

If you want, I can rewrite your BM25 scripts with a **built-in project root path fix** so you won't need to worry about running from the correct folder. Do you want me to do that?

(c) Describe in a few sentences the places in the assignment where you used generative AI and how it was used. Why was it used in some places and not others?

I used generative AI throughout the assignment, especially when I encountered unfamiliar tooling such as Hugging Face Spaces and FastAPI endpoint testing. For example, AI walked me through how to push a Streamlit frontend to Hugging Face, including repo setup and required file structure. It also helped me determine whether I was working inside a Hugging Face Git repo or a GitHub repo and showed how to access the FastAPI docs UI and test my health and /search endpoints. When structuring the backend, AI explained why the BM25 code should not live in main.py and provided a better modular layout. AI also helped debug Python import errors and offered multiple ways to fix module path issues. I used AI in these places because the tasks involved configuration, tool-specific workflows, or debugging processes that are hard to discover just by reading error messages.

Question 2. Over-reliance on generative AI models has been found to diminish learning outcomes. Thus, evaluating and verifying AI outputs, such as by consulting trusted sources like textbooks and journal articles, leads to users gaining better domain knowledge and avoiding that over-reliance.[2]

(a) How have you verified AI generations in the work you submitted?

I tested each solution directly in my project environment. For example, when AI showed how to check which Git remote I was connected to, I validated it by running the exact commands and confirming the output matched the explanation. Likewise, when AI explained how to access FastAPI's automatically generated docs, I tested the /docs and /health endpoints in a browser and confirmed the behavior. For BM25 restructuring, I validated the recommendations by reorganizing the code and verifying that FastAPI still ran cleanly. For the import-path fix , I tried each proposed solution and confirmed which one successfully resolved ModuleNotFoundError. These hands-on tests ensured that AI suggestions were not just theoretically correct but actually functional for my environment.

(b) What modifications did you make to the content generated by the tools you used, and why did you make them?

Although AI provided great guidance, I adapted the suggestions to fit my project. For Hugging Face deployment, the AI provided a general workflow, but I adjusted the folder structure and filenames to match my existing Streamlit app. For FastAPI, I modified the sample health endpoint to integrate with my actual layout. When AI recommended moving BM25 logic into a separate module, I tailored the file structure and import paths to my backend's existing folder organization. These modifications were necessary because AI outputs are generic, and my project had its own directory layout, naming conventions, and environment constraints.

Question 3. Reflective practice allows us to focus on process—not output—and these practices prioritize meaningful learning over the substance of the final outputs of interacting with a generative model.[3]

(a) Let's reflect more broadly. What did you learn in this assignment?

This project helped me build practical skills in deploying applications across multiple tools and services. I learned how to upload and configure a Streamlit frontend on Hugging Face Spaces, how Git remotes determine where code pushes go, and how FastAPI exposes automatic documentation and endpoint testing through /docs. I also gained experience structuring backend code more cleanly, such as separating BM25 logic into its own module instead of embedding it in main.py. I also learned how Python module resolution works and how to fix import errors using project-root paths. The project strengthened my understanding of deployment workflows, API design, and debugging techniques grounded in real-world development practices.

(b) Thinking about the process of your learning rather than the output, how was your learning process impacted by the use of generative AI?

I think my learning process was positively impacted by generative AI in this assignment. Coming into the assignment, Docker, MLflow, and containerized deployments were completely new to me, and I was initially overwhelmed by the complexity of getting everything to work together. Generative AI helped me understand how to fix specific errors, why those errors occurred, how the different components interact with each other. AI provided conceptual explanations alongside practical solutions, which helped my understanding of the microservices architecture. For example, when I had the iris model loading directly in my API, generative AI explained why a separate inference server was better and walked me through the architectural changes needed.

(c) If you hadn't used generative AI, what would have gone differently?

---

[2] Zhai, C., Wibowo, S. & Li, L.D. The effects of over-reliance on AI dialogue systems on students' cognitive abilities: a systematic review. Smart Learn. Environ. 11, 28 (2024). https://doi.org/10.1186/s40561-024-00316-7

[3] Estaphan, S., Kramer, D., and Witchel, H.J. Navigating the frontier of AI-assisted student assignments: challenges, skills, and solutions. Advances in Physiology Education 49.3 (2025): 633-639. https://doi.org/10.1152/advan.00253.2024

Without generative AI, I would have struggled through far more trial and error. For example, setting up Hugging Face Spaces would likely have taken hours as I tried to find the correct repo structure and deployment steps. Understanding Git remote origins without AI's examples would have led to pushing code to the wrong location multiple times. FastAPI debugging would have been slower without knowing about the automatic docs and how to test endpoints properly. Without AI's explanation of why BM25 shouldn't live inside main.py, I might have kept a messy backend structure that caused maintainability issues, especially as my teammates used the syntax I craeted for BM25 for the remaining features.