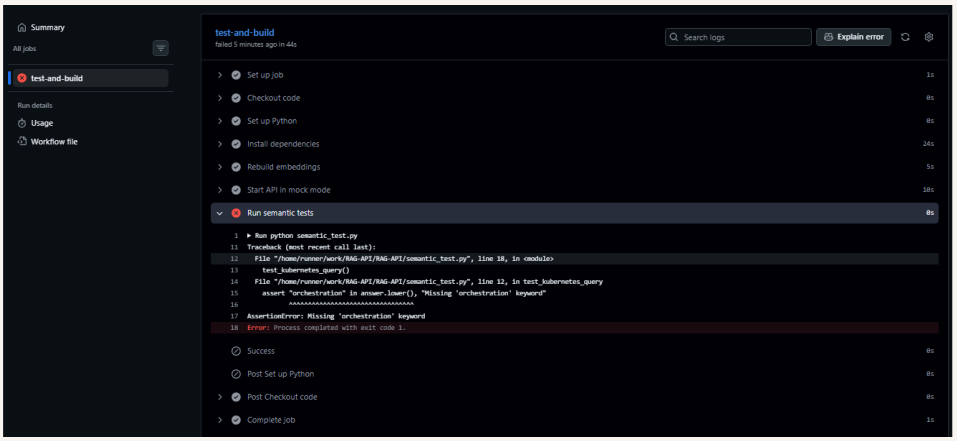
Automate Testing with GitHub Actions



Introducing Today's Project!

In this project, I will demonstrate how to build a CI/CD pipeline with GitHub Actions that automatically tests a RAG (Retrieval-Augmented Generation) API application every time code is pushed to the repository.I'm doing this project to learn how to automate testing and deployment workflows, implement continuous integration practices, and use GitHub Actions to run automated quality checks that verify API responses are accurate and complete ensuring bugs are caught before they reach production.

Key services and concepts

Services I used were GitHub Actions as the main CI/CD automation platform, FastAPI to build and serve the RAG API, ChromaDB to store and query document embeddings, Ollama to run the tinyllama language model locally, and Uvicorn as the web server to host the API during development and testing. Git and GitHub were also central to the whole workflow, from version control to triggering automated pipelines.Key concepts include how RAG systems work by combining document retrieval with language model generation to produce context-aware answers. I also what embeddings are and why rebuilding them matters whenever source documents change. A big one was understanding non-determinism in LLMs and why it breaks automated testing, then solving that by implementing mock LLM mode to return raw retrieved context instead. I learnt how semantic tests differ from regular unit tests since they check for meaning, not just format. I also picked up how GitHub Actions workflows are structured us

Challenges and wins

This project took me approximately five hours to complete from start to finish. The most challenging part was honestly nothing too dramatic — everything moved pretty smoothly throughout. The setup steps were straightforward, the code made sense as I went along, and even the parts that were meant to "break" things worked exactly, which actually made the learning feel really clean and intentional. It was most rewarding to see the RAG system running under GitHub Actions and reporting errors on time. Watching the pipeline catch a missing keyword in the knowledge base, then seeing that failure notification land straight in my GitHub email, made everything click. That moment of "oh, this is how teams catch issues before they hit production" was genuinely satisfying and made the five hours feel well spent.

Why I did this project

I did this project because I wanted to learn how GitHub Actions actually works under the hood, not just at a surface level. I also wanted to understand how RAG systems are built and how they process documents to generate answers, how non-deterministic LLM outputs can be converted into deterministic ones for reliable testing, and how industry professionals structure and automate code in real working environments. One thing I'll apply from this is the mock mode pattern — the idea of stripping out the unpredictable part of a system during testing so you can actually trust your results. That's something I can carry into future projects well beyond RAG. The project absolutely met my goals. It answered every question I came in with and showed me that building reliable AI systems is less about the model and more about the systems and discipline you wrap around it.

Setting Up Your RAG API

I'm setting up my RAG API by making sure I have a working FastAPI application that can load documents into a knowledge base and answer questions using AI. A RAG API retrieves information by searching through documents in a knowledge base to find relevant information, then uses that retrieved information to augment a prompt, and finally generates a natural language answer using a Large Language Model. This foundation is needed for CI/CD because the automated pipeline needs an actual API to test and validate the CI/CD workflow will automatically run tests against this API every time code is pushed to ensure the answers are accurate and complete before deployment.

Local API verification

I tested my RAG API by starting the uvicorn server with the command uvicorn app:app --reload, then opening a new terminal tab and sending a POST request to the /query endpoint using curl with the question "What is Kubernetes?". The API responded with a JSON response containing an answer about Kubernetes being a container orchestration platform that manages containers at scale. This confirms that my RAG API is working correctly it successfully received the question, searched the embedded documents from k8s.txt for relevant context, sent that context and question to the tinyllama LLM via Ollama, and returned a natural language answer generated by the AI model.

```
StatusCode        : 200
StatusDescription : OK
Content           : {"answer":"In short, Kubernetes is a container orchestration platform that is designed to automate
                    the deployment, scaling, and management of Kubernetes clusters. It offers a wide range of features
                    in...
RawContent        : HTTP/1.1 200 OK
                    Content-Length: 450
                    Content-Type: application/json
                    Date: Tue, 03 Feb 2026 19:40:26 GMT
                    Server: uvicorn

                    {"answer":"In short, Kubernetes is a container orchestration platform that...
Forms             : {}
Headers           : {[Content-Length, 450], [Content-Type, application/json], [Date, Tue, 03 Feb 2026 19:40:26 GMT],
                    [Server, uvicorn]}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : mshtml.HTMLDocumentClass
RawContentLength  : 450
```

Initializing Git and Pushing to GitHub

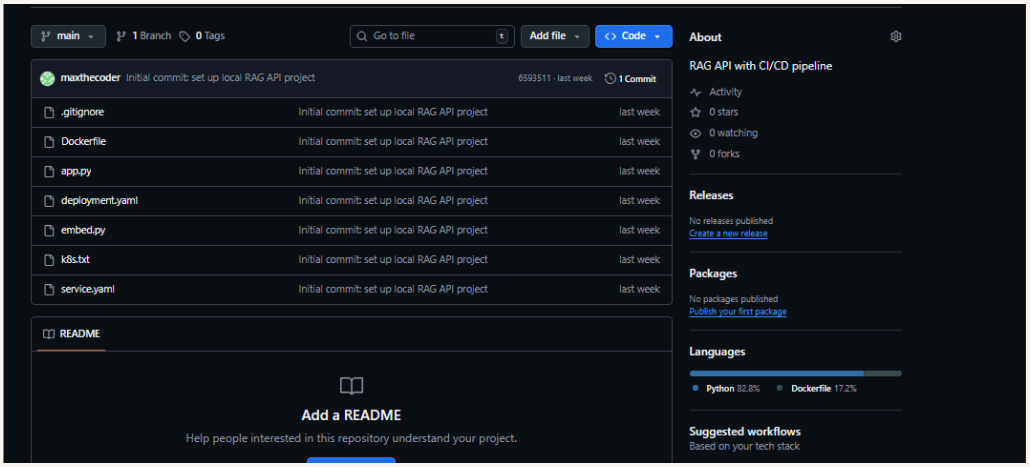
I'm initializing Git by setting up Git version control in my project folder, creating a GitHub repository to host my code online, and pushing my local code to that repository. Git tracks changes by recording snapshots of my project files over time, allowing me to see what changed, when it changed, and revert to earlier versions if needed. Version control enables CI/CD to automatically trigger workflows whenever code changes are pushed to the repository, ensuring that every code update is tested and validated before deployment.

Git initialization and first commit

I initialized Git by running git init in my project directory, which created a hidden .git folder to start tracking changes. Then, I staged and committed the project files using git add . followed by git commit -m "Initial commit: set up local RAG API project" to save the first snapshot of the code. The .gitignore file helps by preventing unnecessary, generated, large, or sensitive files like virtual environments and databases from being tracked in the repository.

Pushing to GitHub for CI/CD

Pushing to GitHub means uploading your local code from your computer to GitHub's remote servers, creating a backup in the cloud and making your code accessible for collaboration and automation. This enables CI/CD because GitHub Actions can automatically detect code changes when you push, trigger automated tests and workflows, and deploy your application - all without manual intervention, forming the foundation for continuous integration and continuous deployment pipelines.


main
1 Branch
0 Tags
Go to file
Add file
Code
maxthecoder Initial commit: set up local RAG API project
6593511 · last week
1 Commit
.gitignore Initial commit: set up local RAG API project last week
Dockerfile Initial commit: set up local RAG API project last week
app.py Initial commit: set up local RAG API project last week
deployment.yaml Initial commit: set up local RAG API project last week
embed.py Initial commit: set up local RAG API project last week
k8s.txt Initial commit: set up local RAG API project last week
service.yaml Initial commit: set up local RAG API project last week
README
Add a README
Help people interested in this repository understand your project.
About
RAG API with CI/CD pipeline
Activity
0 stars
0 watching
0 forks
Releases
No releases published
Create a new release
Packages
No packages published
Publish your first package
Languages
Python 82.8%
Dockerfile 17.2%
Suggested workflows
Based on your tech stack

Creating Semantic Tests

I'm creating semantic tests that verify the RAG system returns answers with the correct meaning and relevant concepts. Unlike unit tests that check code logic, semantic tests validate the data quality and semantic accuracy of AI-generated responses - for example, checking that an answer about "What is Kubernetes?" includes key concepts like "orchestration" and "containers" rather than just verifying the response format. These tests ensure quality by running locally first to understand system behavior before automating with CI/CD, allowing me to experiment with the knowledge base and observe how the RAG API performs under different conditions.

Non-deterministic output observation

When I ran the query multiple times, I noticed that the LLM's responses were non-deterministic - sometimes the answer included "orchestration" even though I had removed it from k8s.txt, and sometimes it didn't. The output varied each time I ran the same query. This is a problem because the LLM (tinyllama) was trained on data that includes "Kubernetes" and "orchestration" together, so it sometimes adds concepts based on its training data rather than strictly using the source document. For CI/CD to work reliably, we need consistent, predictable test results - tests that randomly pass or fail make automation useless because you can't distinguish between actual code problems and LLM variability, leading to false alarms and lost trust in the testing pipeline.

Adding Mock LLM Mode

I'm adding mock LLM mode to the API so it returns retrieved context directly instead of generating answers with the LLM. This solves the non-determinism problem by making the same query always return the same retrieved document without probabilistic LLM variation. Reliable testing requires deterministic outputs that isolate retrieval quality from LLM generation so tests can run consistently in CI/CD without model or platform variability.

Mock LLM mode for CI testing

Mock LLM mode returns the retrieved text directly, which makes tests fast, predictable, and focused only on whether the retrieval step works. Without mock mode, tests would depend on a real LLM generating answers, which introduces randomness, external dependencies like Ollama, and inconsistent results. For automated CI, we need deterministic behavior that runs anywhere without extra services, so mock mode ensures reliable tests that always reflect the actual state of the indexed documents.

Creating GitHub Actions Workflow

I’m creating a GitHub Actions workflow file that defines automated CI steps for my RAG API project. The workflow automates testing by setting up the environment, installing dependencies, and running deterministic tests using the mock LLM mode. When I push code, it will automatically execute these checks on GitHub’s servers to verify that retrieval and API behavior still work correctly.

Workflow automation and CI testing

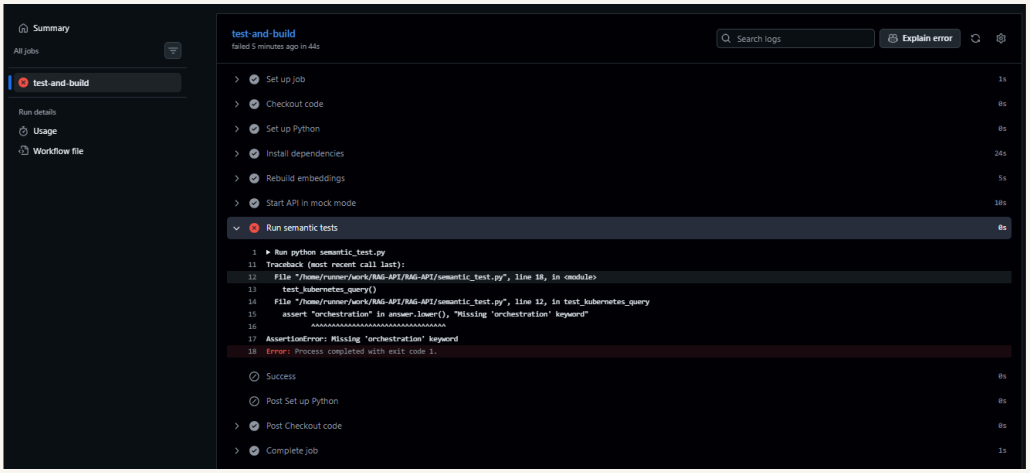
I created the workflow file in the `.github/workflows/ci.yml` path inside my repository so GitHub can detect it as an Actions workflow. I pushed it using `git add .github/workflows/ci.yml`, `git add semantic_test.py`, `git commit -m "Add GitHub Actions CI workflow and semantic test script"`, and `git push`. Once on GitHub, the workflow will automatically run the CI pipeline to rebuild embeddings, start the API in mock mode, and execute semantic tests whenever relevant project files are updated.

Testing Data Quality

I’m triggering the CI workflow by pushing a change to the knowledge base file so GitHub Actions runs the pipeline. The workflow will test the RAG system by rebuilding embeddings, starting the API in mock mode, and running semantic tests against the updated data. I expect it to fail because the knowledge base change removed required content, so the retrieval output no longer satisfies the semantic test conditions.

Data quality and CI protection

The missing keyword was “orchestration” in the k8s.txt knowledge base file. The semantic test failed because after embeddings were rebuilt, the retrieved context no longer contained that required term, so the deterministic mock-mode test detected the gap. Without CI, this degraded content would have been pushed and deployed, causing the RAG system to return incomplete or incorrect answers without anyone noticing until users encountered errors.

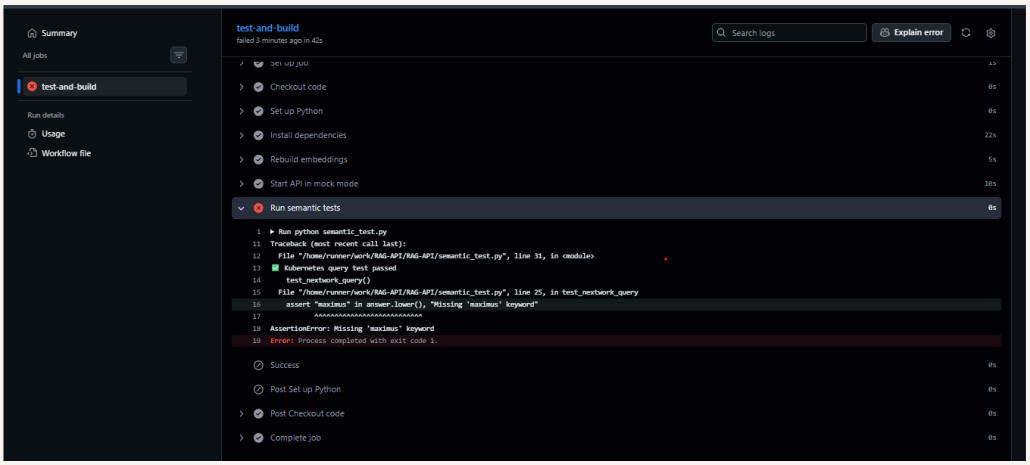

Summary
All jobs
test-and-build
Run details
Usage
Workflow file
test-and-build
failed 5 minutes ago in 44s
Search logs
Explain error
Set up job 1s
Checkout code 0s
Set up Python 0s
Install dependencies 24s
Rebuild embeddings 5s
Start API in mock mode 10s
Run semantic tests 0s
1 ▶ Run python semantic_test.py
11 Traceback (most recent call last):
12 File "/home/runner/work/RAG-API/RAG-API/semantic_test.py", line 18, in <module>
13 test_kubernetes_query()
14 File "/home/runner/work/RAG-API/RAG-API/semantic_test.py", line 12, in test_kubernetes_query
15 assert "orchestration" in answer.lower(), "Missing 'orchestration' keyword"
16 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
17 AssertionError: Missing 'orchestration' keyword
18 Error: Process completed with exit code 1.
Success 0s
Post Set up Python 0s
Post Checkout code 0s
Complete job 1s

Scaling with Multiple Documents

I'm restructuring the project to handle multiple knowledge documents instead of a single file so the RAG system can ingest and retrieve from a growing collection of sources. The new folder structure supports organizing many documents in a dedicated directory, enabling automated ingestion, embedding generation, and CI testing across all files consistently. This approach scales better because adding or updating documents no longer requires code changes, retrieval tests can cover the entire knowledge base automatically, and the system mirrors real-world RAG setups that manage large, evolving document sets reliably.

Docs folder structure and CI scaling

The docs folder organizes files by separating knowledge documents from application code, creating a clean, scalable structure where all source texts live in one dedicated location. The embed_docs.py script handles automatic discovery and embedding of every .txt file inside docs, so new documents are included without changing code or configuration. CI validated all documents and found a mismatch between the semantic test expectation and the actual base2.txt content, proving that the pipeline now checks the entire knowledge base rather than a single file. This structure supports growth by allowing unlimited documents to be added, embedded, tested, and version-controlled consistently, which mirrors how production RAG systems manage large evolving corpora.


Summary
All jobs
test-and-build
Run details
Usage
Workflow file
test-and-build
failed 3 minutes ago in 42s
Search logs
Explain error
Checkout code
Set up Python
Install dependencies
Rebuild embeddings
Start API in mock mode
Run semantic tests
Run python semantic_test.py
Traceback (most recent call last):
File "/home/runner/work/RAG-API/RAG-API/semantic_test.py", line 31, in <module>
Kubernetes query test passed
test_nextwork_query()
File "/home/runner/work/RAG-API/RAG-API/semantic_test.py", line 25, in test_nextwork_query
assert "maximus" in answer.lower(), "Missing 'maximus' keyword"
AssertionError: Missing 'maximus' keyword
Error: Process completed with exit code 1.
Success
Post Set up Python
Post Checkout code
Complete job