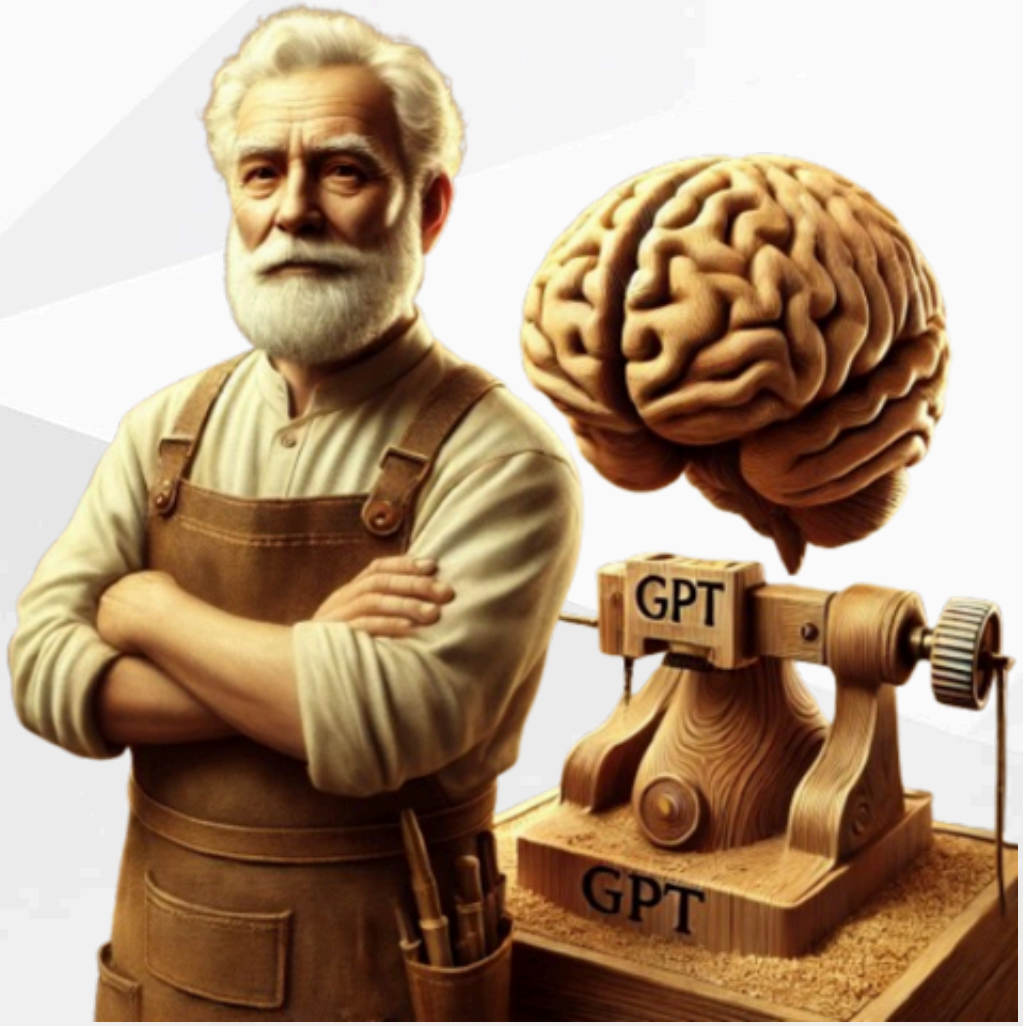
Developing Open LLM applications with

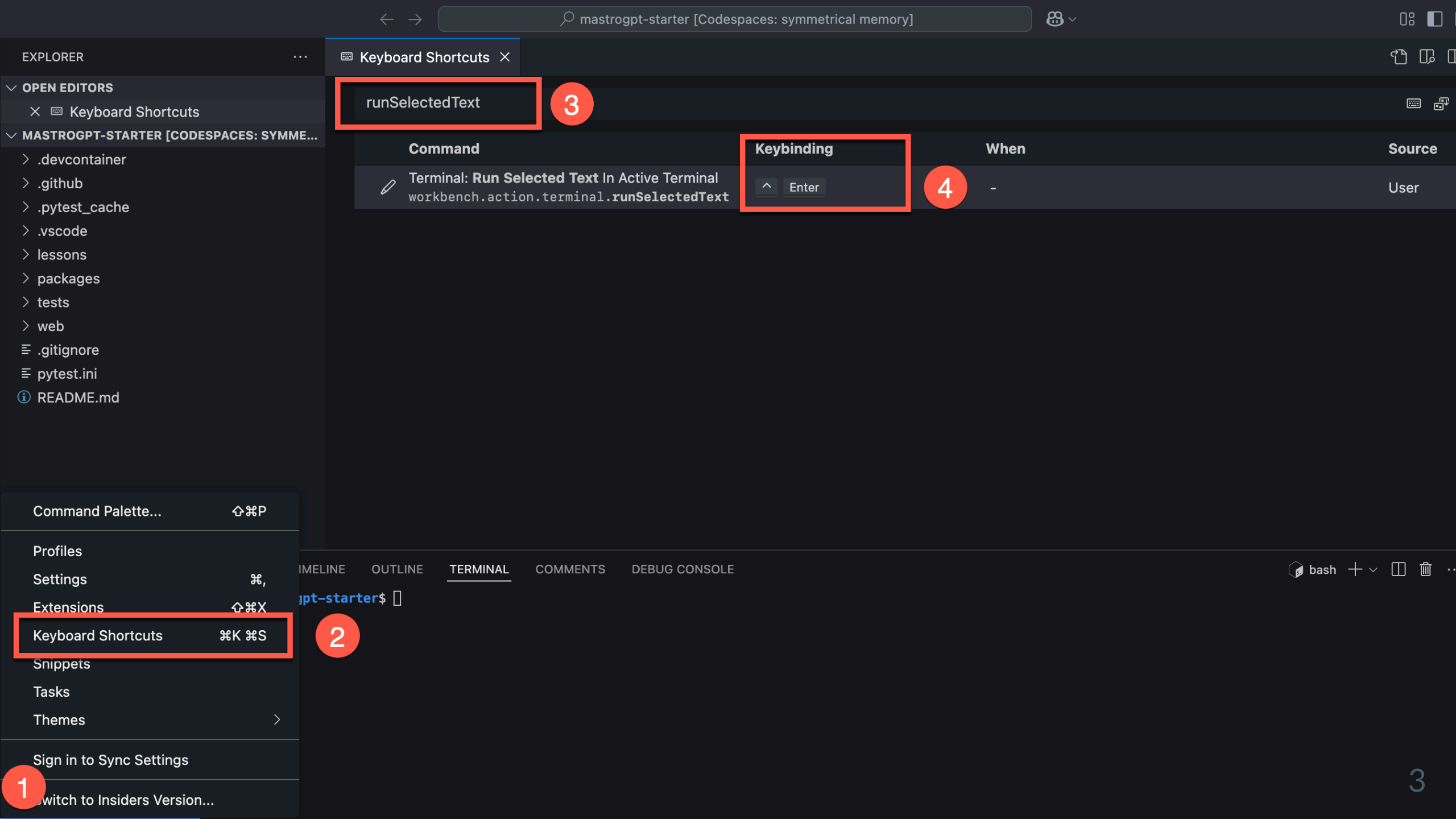
Lesson 2

An LLM chat with streaming



Agenda

- Accessing the LLM
- Managing Secrets
- How to stream
- Exercise: Streaming LLM


mastrogpt-starter [Codespaces: symmetrical memory]
EXPLORER
OPEN EDITORS
Keyboard Shortcuts
MASTROGPT-STARTER [CODESPACES: SYMME...
.devcontainer
.github
.pytest_cache
.vscode
lessons
packages
tests
web
.gitignore
pytest.ini
README.md
Keyboard Shortcuts
runSelectedText
3
Command
Keybinding
When
Source
Terminal: Run Selected Text In Active Terminal
workbench.action.terminal.runSelectedText
Enter
4
-
User
Command Palette... ⇧⌘P
Profiles
Settings ⌘,
Extensions ⇧⌘X
Keyboard Shortcuts ⌘K ⌘S
Snippets
Tasks
Themes
Sign in to Sync Settings
Switch to Insiders Version...
1
2
TIMELINE
OUTLINE
TERMINAL
COMMENTS
DEBUG CONSOLE
bash
gpt-starter$
3

Accessing the LLM

Getting credentials

- Credentials are available in the environment:
  - `OLLAMA_HOST` is the url
  - `AUTH` are the credentials
    - **for the dev environment only!**
- Enter the CLI: `ops ai cli`
- Checking you have the credentials:

```
import os
os.getenv("OLLAMA_HOST")
os.getenv("AUTH")
```

Accessing Ollama

```
args = {}
host = args.get("OLLAMA_HOST", os.getenv("OLLAMA_HOST"))
auth = args.get("AUTH", os.getenv("AUTH"))
base = f"https://{auth}@{host}/"
```

Test it!

```
!curl {base}
```

```
Ollama is Running
```

Talking with a model in Ollama

```
# using llama 3.1 8 Billions
MODEL = "llama3.1:8b"
inp = "Who are you?"
# preparing a request
msg = { "model": MODEL, "prompt": inp, "stream": False}
url = f"{base}/api/generate"
# making a request
import requests as req
res = req.post(url, json=msg).json()
out = res.get('response', 'error')
print(out)
```

Putting all togegether in an action

Checking the code:

```
!code packages/chat/simple.py
```

Deploying the action

Low level commands **DO NOT DO THIS WAY - FOR ILLUSTRATION**

```
!ops package create chat
!ops action update chat/simple packages/chat/simple.py \
     --web=true --param AUTH {auth} --param OLLAMA_HOST {host}
```

Managing Secrets

From where the `OLLAMA_HOST` and `AUTH` are coming from?

Multiple places!

- From the config **loaded at the login**
- From the `.env` file
- From the `packages/.env` for actions
- from the `tests/.env` for tests

(the later override the former)

Where is `OLLAMA_HOST`?

- in `tests/.env` for actions
- in `packages/.env` for tests

```
!grep OLLAMA_HOST packages/.env
!grep OLLAMA_HOST tests/.env
```

`OLLAMA_HOST=ollama.nuvolaris.io`

`OLLAMA_HOST=ollamatest.nuvolaris.io`

note that they are different!

Where is `AUTH`?

Is is NOT in any `.env`

```
grep AUTH .env packages/.env tests/.env
```

`#(nothing)#`

It is in the config!

```
ops -config -dump | grep AUTH
```

`AUTH=<uid>:<secret>`

How to propagate secrets to actions

- When you login you get the secrets for deployment

```
ops -config -dump
```

- You can then propagate the secrets to actions adding:

```
#--web true
#--param OLLAMA_HOST $OLLAMA_HOST
#--param AUTH $AUTH
```

- The use `ops ide deploy chat/simple.py`

Important!

Tests and CLI see:
config, `.env` and `tests/.env`

Deployment see:
config, `.env` and `packages/.env`

The test environment is different from the deployemnt environment

Best practices

Always get secrets from args and env

```
host = args.get("OLLAMA_HOST", os.getenv("OLLAMA_HOST"))
```

Put the args in the main file of the action

```
#--param OLLAMA_HOST $OLLAMA_HOST
```

Deploy with

```
ops ide deploy [<action>]
```

About `ops ide deploy`

- Built on top of `ops actions` and `ops packages`
- Works **currently** only for `python`, `node` and `php`
- Create packages for actions
- Create a zip for multi-file actions
- Resolve Dependencies: `requirements.txt`, `packages.json`
- Extract command line arguments from `#--<arg> <val>`
  - Required to propagate secrets: `#--param AUTH $AUTH`
- Integrates with `ops ide devel` for incremental deploy

How to stream

Let's see the streaming in `ops ai cli`

```
# prepare
import os, requests as req
url = f'https://{os.getenv("AUTH")}@{os.getenv("OLLAMA_HOST")}/api/generate'
# streaming request
msg = {"model": "llama3.1:8b", "prompt": "Capital of Italy", "stream": True}
res = req.post(url, json=msg)
```

Result from the LLM is a stream:

```
lines = res.iter_lines()
for line in lines:
  print(line)
```

Countdown generator

```
import time
def count_to_zero(n):
  while n > 0:
    yield f"{n}...\n"
    n -= 1
    time.sleep(1)
  yield "Go!\n"
```

Test:

```
for line in count_to_zero(10):
  print(line, end='')
```

To stream we use a socket!

- The action is long running
- We send intermediate results in the socket

Connect:

```
sock = args.get("STREAM_HOST")
port = int(args.get("STREAM_PORT"))
with socket.connect((sock, port)) as s:
```

Send a json `msg`:

```
s.sendall(json.dumps(msg).encode('utf-8'))
```

The `stream` function for an iterator

```
import json, socket, traceback
def stream(args, lines):
  sock = args.get("STREAM_HOST") ; port = int(args.get("STREAM_PORT"))
  out = ""
  with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((sock, port))
    try:
      for line in lines:
        msg = {"output": line}
        s.sendall(json.dumps(msg).encode("utf-8"))
        out += str(line)
    except Exception as e:
      traceback.print_exc(e)
      out = str(e)
  return out
```

Testing a stream with a mock!

```
import sys; sys.path.append("tests")
import streamock
args = streamock.args()
mock = streamock.start(args)
```

Running the tests:

```
lines = count_to_zero(10) # extracting the generator
stream(args, lines)       # streaming the generated items
```

Collecting the results:

```
streamock.stop(mock)
```

`countdown.py` and `test_countdown.py`

```
!code packages/chat/countdown.py
```

Note the `"streaming": true` to enable streaming

```
return { "output": out, "streaming": True }
```

Checking the test and deploying

```
!code packages/chat/countdown.py
!ops ide deploy chat/countdown.py
!ops ide deploy mastrogpt/index
```

Exercise: Streaming LLM

Exercise 1: Add the secrets

Search `TODO: E2.1` and add the parameters

```
#--param XXX $XXX
```

Result: test_stateless.test_url should pass

Exercise 2: Add the streaming

Search `TODO: E2.2` and insert the `stream` implementation, changing:

```
msg = {"output": line}
out += line
```

to:

```
dec = json.loads(line.decode("utf-8")).get("response", "")
msg = {"output": out}
out += dec
```

Result: test_stateless.test_stream should pass

Exercise 3: model switcher

if input is `deepseek` change to `deepseek-r1:32b`

if input is `llama` change to `llama3.1:8b`

Bonus: replace `<think>` to `[think]`

Result: I can change to llama and deeepseek

What is Next?

Lesson 3 - Form

Support for form, display and advanced rendering

More lessons

- Lesson 4: Building an Assistant
- Lesson 5: Vision Support
- Lesson 6: VectorDB
- Lesson 7: Bulding a RAG