



**Developing Open LLM
applications with**



Apache OpenServerless

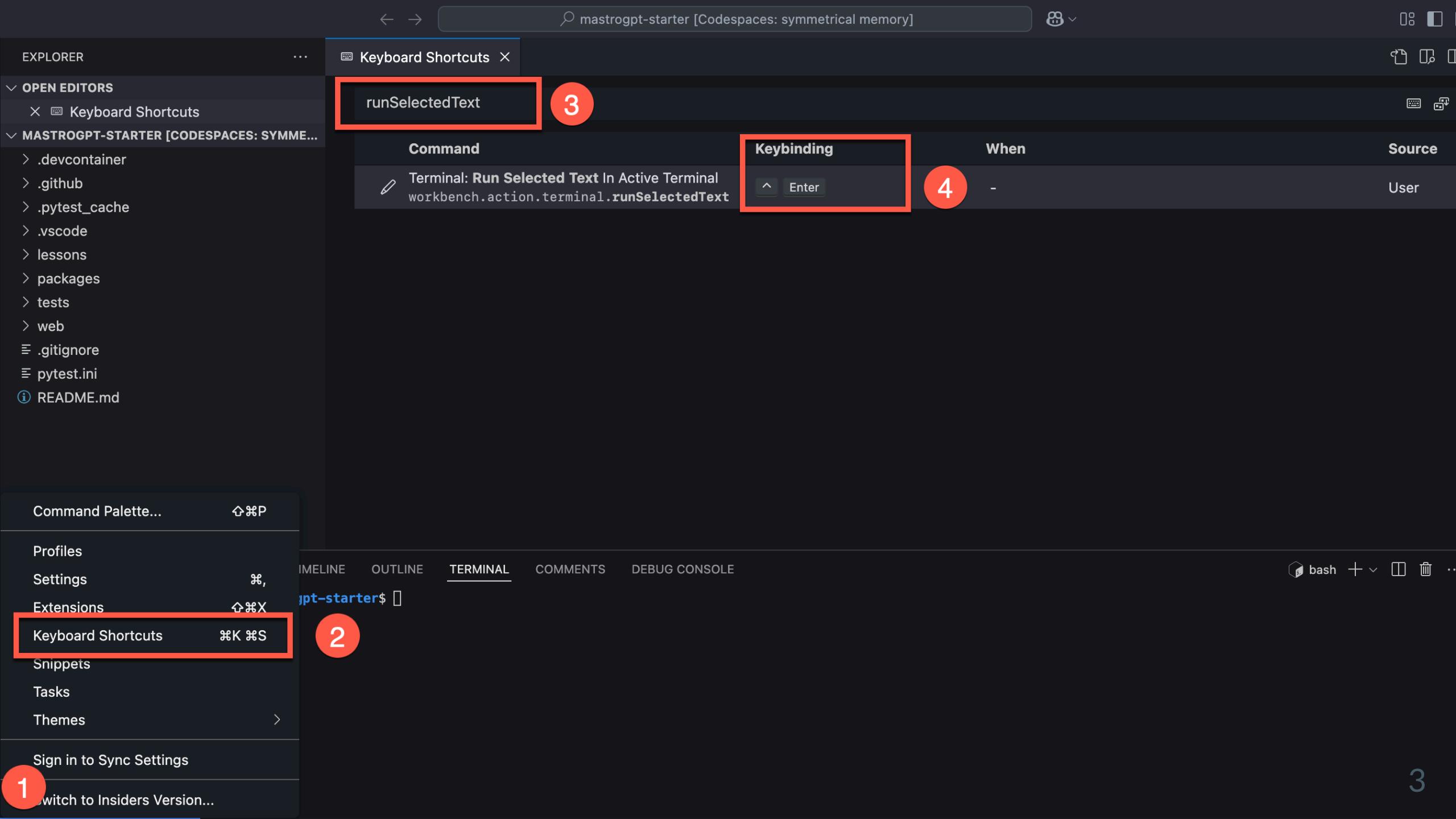
Lesson 2

**An LLM chat
with streaming**

An LLM chat with streaming

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





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 together in an action

Checking the code:

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

Deploying the action "manually"

YOU DO NOT DO THIS WAY - FOR ILLUSTRATION PURPOSE

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

mastrogpt/mastrogpt-starter x | 2-stream.md — mastrogpt-st x Pinocchio - Nuvolaris x +

https://demouser.openserverless.dev

Chat Simple Side View

Simple

Chat

- Simple
- Countdown
- Chat

Test

- Hello
- Demo

demouser

BOT 12:43

Welcome to llama3.1:8b

YOU 12:43

hello

BOT 12:43

Hello! How are you today? Is there something I can help you with or would you like to chat?

YOU 12:43

who are you?

BOT 12:43

I'm an artificial intelligence model known as Llama. Llama stands for "Large Language Model Meta AI."

Enter your message... Send

9

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 packages/.env tests/.env
```

(no output)

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
```

- Then 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* variables are **different** from the *deployment* variables

In test you use *local* services instead of production ones

Best practices

Always get secrets from `args` and default `getenv`

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

Add the `#--param` in the main file of the action or `__main__`

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

Pass args to the action deploying 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)
```

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

```
b'{"model":"llama3.1:8b","response":"R","done":false}'
b'{"model":"llama3.1:8b","response":"ome","done":false}'
```

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
- we receive the socket location in `args` as:
 - `STREAM_HOST` usually fixed
 - `STREAM_PORT` always changing

```
sock = args.get("STREAM_HOST")
port = int(args.get("STREAM_PORT"))
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((sock, port)) # double parenthesis NOT error
    # ... prepare a msg in json ...
    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) #; print(line, end=' ')
    except Exception as e:
        traceback.print_exc(e)
        out = str(e)
    return out
```

streamock: a mock to test the stream

```
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

```
# WARN: exit for cli to avoid busy ports!
code packages/chat/countdown.py
code tests/chat/test_countdown.py
```

Note the "streaming": true to enable streaming

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

Checking the test and deploying

```
ops ide deploy chat/countdown.py
ops ide deploy mastrogpt/index
```

mastrogpt/mastrogpt-starter x | 2-stream.md — mastrogpt-st x Pinocchio - Nuvolaris x +

https://demouser.openserverless.dev

Chat

Simple

Countdown

Chat

Test

Hello

Demo

BOT 12:44

Input a number > 0 to countdown

Countdown

Side View

YOU 12:44

12

BOT 12:44

12...
11...
10...
9...
8...
7...
6...
5...
4...
3...
2...
1...
Go!

Enter your message...

Send

25

Exercise: Streaming LLM

Exercise 1: Add the secrets

You have to add the code to retrieve the secrets from the environment.

Search `TODO:E2.1` add params for accessing and authorizing Ollama access

Hint:

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

```
args.get("XXX", os.getenv("XXX"))
```

Exercise 2: Fix the streaming

You have to change the default streaming function from extracting the response from the answer.

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

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

Result: `test_stateless.test_stream` should pass

Exercise 3: model switcher

Search `TODO:E2.3` add if to switch to `llama3.1:8b` or `deepseek-r1:32b` on input 'llmama' or 'deepseek'

- if input is `deepseek` change the model to `deepseek-r1:32b`
- if input is `llama` change the model to `llama3.1:8b`

Bonus: fix `stream` to replace `<think>` with `[think]` (no hints)

Result: test if you can change the model

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: Building a RAG