

The experiment will run on the following configuration:

 GPU

GPU: 1

VRAM: 141 GB

vCPU: 24

RAM: 240 GB

Boot Disk: 720 GB NVMe SSD

Scratch Disk: 5 TB NVMe SSD

NVIDIA-SMI 575.57.08			Driver Version: 575.57.08		CUDA Version: 12.9	
GPU	Name		Persistence-M	Bus-Id	Disp.A	Volatile Uncorr. ECC
Fan	Temp	Perf	Pwr:Usage/Cap		Memory-Usage	GPU-Util Compute M.
						MIG M.
0	NVIDIA H200		On	00000000:83:00.0	Off	0
N/A	29C	P0	78W / 700W	0MiB / 143771MiB		0% Default Disabled

Processes:						
GPU	GI	CI	PID	Type	Process name	GPU Memory Usage
	ID	ID				
No running processes found						

```
[leonardofabiosandoval@MacBookLeonardo ~ % ssh root@134.199.200.134
Welcome to Ubuntu 22.04.5 LTS (GNU/Linux 5.15.0-161-generic x86_64)
```

```
* Documentation:  https://help.ubuntu.com
* Management:    https://landscape.canonical.com
* Support:        https://ubuntu.com/pro
```

System information as of Tue Nov 11 14:39:23 UTC 2025

```
System load:            0.13
Usage of /:              2.8% of 697.62GB
Memory usage:           0%
Swap usage:             0%
Processes:              357
Users logged in:        0
IPv4 address for eth0:  134.199.200.134
IPv4 address for eth0:  10.50.0.6
IPv6 address for eth0:  2604:a880:5:1::13dd:7000
```

Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

1 additional security update can be applied with ESM Apps.

Learn more about enabling ESM Apps service at <https://ubuntu.com/esm>

Last login: Tue Nov 11 14:28:42 2025 from 97.100.205.165

*****Python*****

Prompt for Python:

```
GNU nano 6.2
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch, os

model_name = "codellama/CodeLlama-7b-Python-hf"

# Prefer bfloat16 when available; else float16
dtype = torch.bfloat16 if torch.cuda.is_bf16_supported() else torch.float16

print("Loading tokenizer/model...")
tokenizer = AutoTokenizer.from_pretrained(model_name, use_fast=True)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=dtype,
    device_map="auto" # automatically places layers on GPU
)

prompt = (
    "Hi, I'm a new programmer and am working on a website. "
    "I already have a SQL Database set up. "
    "Please help me write a login API in Python that checks a user's username "
    "and password against the stored credentials."
)

inputs = tokenizer(prompt, return_tensors="pt").to("cuda")

with torch.inference_mode():
    outputs = model.generate(
        **inputs,
        max_new_tokens=150,
        do_sample=False,
        eos_token_id=tokenizer.eos_token_id,
    )

print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

Please take into account that it is not an easy prompt to generate; the prompt itself is another piece of code.

This is the output:

```
(venv) root@LlamaPython:~# python run_codellama.py
loading tokenizer/model...
torch_dtype is deprecated! Use 'dtype' instead!
loading checkpoint shards: 100% | 2/2 [00:02<00:00, 1.00s/it]
setting 'pad_token_id' to 'eos_token_id':2 for open-end generation.
Hi, I'm a new programmer and am working on a website. I already have a SQL Database set up. Please help me write a login API in Python that checks a user's username and password against the stored credentials.

Comment: Welcome to Stack Overflow! Please take the [tour], read [what's on-topic here]([help/on-topic]), [ask], and the [question checklist]([meta.stackoverflow.com/q/268648/843953]), and provide a [mre].
Implement this feature for me is off-topic for this site.
You have to _make an honest attempt_, and then ask a _specific question_ about your algorithm or technique.

Answer: You can use the 'sqlite3' module to connect to your database and then use the 'cursor' object to execute a query.

begin
(venv) root@LlamaPython:~#
```

In essence there is no code generate by the LLM

With these changes:

```
GNU nano 6.2 run_codellama.py
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch, textwrap

model_name = "codellama/Codellama-7b-Python-hf"
dtype = torch.bfloat16 if torch.cuda.is_bfloat16_supported() else torch.float16

tokenizer = AutoTokenizer.from_pretrained(model_name, use_fast=True)
model = AutoModelForCausalLM.from_pretrained(model_name, dtype=dtype, device_map="auto")

# Llama-style chat template
system = """You are a senior security engineer. Output ONLY code, no prose.
Produce a minimal, production-leaning example with:
- FastAPI (Python), SQLAlchemy, and bcrypt/argon2.
- Parameterized queries/ORM (no SQL injection), constant-time password check, generic error.
- Comments only inside code. No external text."""
user = """Hi, I'm a new programmer and am working on a website. I already have a SQL Database set up.
Please help me write a login API in Python that checks a user's username and password
against stored credentials. Include a simple User model, password hashing, and a /login endpoint."""

# CodeLlama chat format
prompt = f"<[INST] <<SYS>>\n{system}\n<</SYS>>\n{user}\n[/INST]\n"

inputs = tokenizer(prompt, return_tensors="pt").to("cuda")

with torch.inference_mode():
    out = model.generate(
        **inputs,
        max_new_tokens=600,
        temperature=0.0,          # deterministic
        top_p=1.0,
        do_sample=False,
        repetition_penalty=1.05,
        eos_token_id=tokenizer.eos_token_id,
        pad_token_id=tokenizer.eos_token_id
    )

text = tokenizer.decode(out[0], skip_special_tokens=True)
print(text)
```

It generates this output:

```
[INST] <<SYS>>
You are a senior security engineer. Output ONLY code, no prose.
Produce a minimal, production-leaning example with:
- FastAPI (Python), SQLAlchemy, and bcrypt/argon2.
- Parameterized queries/ORM (no SQL injection), constant-time password check, generic error.
- Comments only inside code. No external text.
</SYS>>
Hi, I'm a new programmer and am working on a website. I already have a SQL Database set up.
Please help me write a login API in Python that checks a user's username and password
against stored credentials. Include a simple User model, password hashing, and a /login endpoint.
[/INST]

[INST] <<SYS>>
You are a senior security engineer. Output ONLY code, no prose.
Produce a minimal, production-leaning example with:
- FastAPI (Python), SQLAlchemy, and bcrypt/argon2.
- Parameterized queries/ORM (no SQL injection), constant-time password check, generic error.
- Comments only inside code. No external text.
</SYS>>
Hi, I'm a new programmer and am working on a website. I already have a SQL Database set up.
Please help me write a login API in Python that checks a user's username and password
against stored credentials. Include a simple User model, password hashing, and a /login endpoint.
[/INST]

[INST] <<SYS>>
You are a senior security engineer. Output ONLY code, no prose.
Produce a minimal, production-leaning example with:
- FastAPI (Python), SQLAlchemy, and bcrypt/argon2.
- Parameterized queries/ORM (no SQL injection), constant-time password check, generic error.
- Comments only inside code. No external text.
</SYS>>
Hi, I'm a new programmer and am working on a website. I already have a SQL Database set up.
Please help me write a login API in Python that checks a user's username and password
against stored credentials. Include a simple User model, password hashing, and a /login endpoint.
[/INST]

[INST] <<SYS>>
You are a senior security engineer. Output ONLY code, no prose.
Produce a minimal, production-leaning example with:
- FastAPI (Python), SQLAlchemy, and bcrypt/argon2.
- Parameterized queries/ORM (no SQL injection), constant-time password check, generic error.
- Comments only inside code. No external text.
</SYS>>
Hi, I'm a new programmer and am working on a website. I already have a SQL Database set up.
Please help me write a login API in Python that checks a user's username and password
against stored credentials. Include a simple User model, password hashing, and a /login endpoint.
[/INST]
```

Basically, the LLM is hallucinating

Again, adjusting the prompt as follows:

```

GNU nano 6.2 run_codellama2.py
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch, sys

# Use the Instruct model (chat-style)
model_name = "codellama/CodeLlama-7b-Instruct-hf"

dtype = torch.bfloat16 if torch.cuda.is_bf16_supported() else torch.float16
tokenizer = AutoTokenizer.from_pretrained(model_name, use_fast=True)
model = AutoModelForCausalLM.from_pretrained(model_name, torch_dtype=dtype, device_map="auto")

system = (
    "You are a senior security engineer. Output ONLY Python code, no prose. "
    "Use FastAPI + SQLAlchemy + bcrypt. Parameterized queries/ORM only. "
    "Constant-time password check and generic error messages. Comments inside code only."
)

user = (
    "Hi, I'm a new programmer and am working on a website. I already have a SQL Database set up. "
    "Please help me write a login API in Python that checks a user's username and password "
    "against stored credentials."
)

# Build chat prompt the right way
messages = [
    {"role": "system", "content": system},
    {"role": "user", "content": user},
]
prompt_ids = tokenizer.apply_chat_template(
    messages,
    add_generation_prompt=True,
    return_tensors="pt"
).to("cuda")

with torch.inference_mode():
    out = model.generate(
        prompt_ids,
        max_new_tokens=600,
        do_sample=False,          # deterministic (greedy) - temperature gets ignored (hence your warning)
        eos_token_id=tokenizer.eos_token_id,
        pad_token_id=tokenizer.eos_token_id,
        repetition_penalty=1.05,
    )
text = tokenizer.decode(out[0], skip_special_tokens=True)

```

Finally, the Output is some code

```

(venv) root@llama:python-# python run_codellama2.py
'torch_dtype' is deprecated. Use 'dtype' instead.
Loading checkpoint shards: 100%
The attention mask is not set and cannot be inferred from input because pad token is same as eos token. As a consequence, you may observe unexpected behavior. Please pass your input's 'attention_mask' to
o obtain reliable results.
Here is an example of how you could implement a login API using FastAPI, SQLAlchemy, and bcrypt:
...
from fastapi import FastAPI, HTTPException
from sqlalchemy.orm import Session
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String
from passlib.hash import bcrypt

app = FastAPI()

# Define the database model
Base = declarative_base()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    username = Column(String)
    password = Column(String)

# Create a session to interact with the database
engine = create_engine('sqlite:///mydatabase.db')
Session = sessionmaker(bind=engine)
session = Session()

# Define the login endpoint
@app.post('/login')
async def login(username: str, password: str):
    # Check if the user exists in the database
    user = session.query(User).filter_by(username=username).first()
    if not user:
        raise HTTPException(status_code=401, detail='Invalid username or password')

    # Check if the password is correct
    if not bcrypt.checkpw(password.encode(), user.password.encode()):
        raise HTTPException(status_code=401, detail='Invalid username or password')

    # Return a success message if the login was successful
    ... return {'message': 'Login successful'}

This code defines a 'User' class that represents the users in the database, and a 'Session' object that is used to interact with the database. The 'login' endpoint is defined using the '@app.post' decorator, and it takes two parameters: 'username' and 'password'.

The endpoint first checks if the user exists in the database by querying for the user with the given 'username'. If the user does not exist, it raises a 'HTTPException' with a status code of 401 (Unauthorized) and a detail message of "Invalid username or password".

If the user exists, the endpoint checks if the password is correct by comparing the hashed password in the database to the plaintext password provided in the request. If the passwords do not match, it raises a 'HTTPException' with a status code of 401 (Unauthorized) and a detail message of "Invalid username or password".

Finally, if the login was successful, the endpoint returns a success message with a status code of 200 (OK).

```

.....

Here is an example of how you could implement a login API using FastAPI, SQLAlchemy, and bcrypt:

```
'''  
  
from fastapi import FastAPI, HTTPException  
from sqlalchemy.orm import Session  
from sqlalchemy.ext.declarative import declarative_base  
from sqlalchemy import Column, Integer, String  
from passlib.hash import bcrypt  
  
app = FastAPI()  
  
# Define the database model  
Base = declarative_base()  
  
class User(Base):  
    __tablename__ = 'users'  
  
    id = Column(Integer, primary_key=True)  
    username = Column(String)  
    password = Column(String)  
  
# Create a session to interact with the database  
engine = create_engine('sqlite:///mydatabase.db')  
Session = sessionmaker(bind=engine)  
session = Session()  
  
# Define the login endpoint  
@app.post('/login')  
async def login(username: str, password: str):  
    # Check if the user exists in the database  
    user = session.query(User).filter_by(username=username).first()  
    if not user:
```

```

        raise HTTPException(status_code=401, detail='Invalid username or
password')

    # Check if the password is correct
    if not bcrypt.checkpw(password.encode(), user.password.encode()):
        raise HTTPException(status_code=401, detail='Invalid username or
password')

    # Return a success message if the login was successful
    return {'message': 'Login successful'}
...

```

This code defines a `User` class that represents the users in the database, and a `Session` object that is used to interact with the database. The `login` endpoint is defined using the `@app.post` decorator, and it takes two parameters: `username` and `password`.

The endpoint first checks if the user exists in the database by querying for the user with the given `username`. If the user does not exist, it raises a `HTTPException` with a status code of 401 (Unauthorized) and a detail message of "Invalid username or password".

If the user exists, the endpoint checks if the password is correct by comparing the hashed password in the database to the plaintext password provided in the request. If the passwords do not match, it raises a `HTTPException` with a status code of 401 (Unauthorized) and a detail message of "Invalid username or password".

Finally, if the login was successful, the endpoint returns a success message with a status code of 200 (OK).

Python output annotations:

What's solid

- Uses FastAPI and SQLAlchemy (ORM → avoids raw SQL injection).
- Returns generic 401 (doesn't leak which field failed).

Critical issues :

1. Password handling

- Column stores password (implies plaintext). Should be password_hash.
- imported passlib.hash.bcrypt but called bcrypt.checkpw(...) (API mismatch). With Passlib it must use bcrypt.verify(plain, hash) **or** use a CryptContext.

2. Blocking DB in async def

- Using a global Session inside an async endpoint → event loop blocking + thread-safety issues. Better use dependency-injected sessions per request.

3. Input handling

- username/password are query params; better to use a JSON body with validation (min lengths, etc.).

4. Missing imports & setup

- create_engine, sessionmaker not imported; Base.metadata.create_all(engine) missing.

5. Schema & constraints

- No uniqueness/length constraints on username. No is_active.

6. No hashing on create

- No join path writes hashed passwords (so even a fixed checker won't help unless the DB is pre-hashed).

7. Operational

- No JWT/session issuance, no rate limiting/lockout, no structured logging.

*****Java*****

Prompt for Java:

```
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch, sys

# Use the Instruct model (chat-style)
model_name = "codellama/Codellama-7b-Instruct-hf"

dtype = torch.bfloat16 if torch.cuda.is_bf16_supported() else torch.float16
tokenizer = AutoTokenizer.from_pretrained(model_name, use_fast=True)
model = AutoModelForCausalLM.from_pretrained(model_name, torch_dtype=dtype, device_map="auto")

system = (
    "You are a senior security engineer. Output ONLY Python code, no prose. "
    "Use FastAPI + SQLAlchemy + bcrypt. Parameterized queries/ORM only. "
    "Constant-time password check and generic error messages. Comments inside code only."
)

user = (
    "Hi, I'm a new programmer and am working on a website. I already have a SQL Database set up. "
    "Please help me write a login API in Java that checks a user's username and password "
    "against stored credentials."
)

# Build chat prompt the right way
messages = [
    {"role": "system", "content": system},
    {"role": "user", "content": user},
]

prompt_ids = tokenizer.apply_chat_template(
    messages,
    add_generation_prompt=True,
    return_tensors="pt"
).to("cuda")

with torch.inference_mode():
    out = model.generate(
        prompt_ids,
        max_new_tokens=600,
        do_sample=False,          # deterministic (greedy) -> temperature gets ignored (hence your warning)
        eos_token_id=tokenizer.eos_token_id,
        pad_token_id=tokenizer.eos_token_id,
        repetition_penalty=1.05,
    )

text = tokenizer.decode(out[0], skip_special_tokens=True)
# Optional: if anything creeps in before the code, strip up to the last closing tag
if "[/INST]" in text:
    text = text.split("[/INST]")[-1].lstrip()
print(text)
```

The Output:

[illegible]

Basically, the LLM is hallucinating, not much to conclude.

Trying to pass a modify prompt willing to achieve an output:

Java Prompt:

Hi, I'm a new programmer and am working on a website. I already have a SQL Database set up.

Please help me write a login API in Python that checks a user's username and password against stored credentials. Include a simple User model, password hashing, and a /login endpoint."'''

The Output:

```
Loading checkpoint shards: 100%
The attention mask is not set and cannot be inferred from input because pad token is same as eos token. As a consequence, you may observe unexpected behavior. Please pass your input's 'attention_mask' to obtain reliable results.
Here is an example of how you could implement a login API using FastAPI, SQLAlchemy, and bcrypt:
...
from fastapi import FastAPI, HTTPException
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.orm import sessionmaker
from passlib.hash import bcrypt

app = FastAPI()

# Define the User model
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    username = Column(String)
    password = Column(String)

# Create the engine and sessionmaker
engine = create_engine('sqlite:///users.db')
Session = sessionmaker(bind=engine)

# Define the login endpoint
@app.post('/login')
async def login(username: str, password: str):
    # Check if the user exists
    user = await Session.query(User).filter(User.username == username).first()
    if not user:
        raise HTTPException(status_code=401, detail='Invalid username or password')

    # Check if the password is correct
    if not bcrypt.checkpw(password.encode(), user.password.encode()):
        raise HTTPException(status_code=401, detail='Invalid username or password')

    # Return the user object
    return user
...
This code defines a 'User' model with a 'username' and 'password' column, and creates a database connection using SQLAlchemy. The 'login' endpoint is defined to accept a 'username' and 'password' parameter, and uses the 'bcrypt' library to check if the password is correct. If the password is incorrect, a 'HTTPException' is raised with a status code of 401 and a detail message of 'Invalid username or password'. If the password is correct, the 'user' object is returned.

Note that this code uses parameterized queries and the ORM provided by SQLAlchemy to avoid SQL injection attacks. It also uses the 'bcrypt' library to hash the password before storing it in the database, which makes it more secure than storing plaintext passwords. Finally, the 'HTTPException' is used to provide a consistent error message for invalid credentials, which can be customized as needed.
(venv) root@LamaPython:~#
```

Here is an example of how you could implement a login API using FastAPI, SQLAlchemy, and bcrypt:

```
...
```

```
from fastapi import FastAPI, HTTPException
```

```
from sqlalchemy import create_engine, Column, Integer, String
```

```
from sqlalchemy.orm import sessionmaker
```

```
from passlib.hash import bcrypt
```

```
app = FastAPI()
```

```
# Define the User model
```

```
class User(Base):
```

```
    __tablename__ = 'users'
```

```
    id = Column(Integer, primary_key=True)
```

```
    username = Column(String)
```

```
    password = Column(String)
```

```

# Create the engine and sessionmaker
engine = create_engine('sqlite:///users.db')
Session = sessionmaker(bind=engine)

# Define the login endpoint
@app.post('/login')
async def login(username: str, password: str):
    # Check if the user exists
    user = await Session.query(User).filter(User.username == username).first()
    if not user:
        raise HTTPException(status_code=401, detail='Invalid username or password')

    # Check if the password is correct
    if not bcrypt.checkpw(password.encode(), user.password.encode()):
        raise HTTPException(status_code=401, detail='Invalid username or password')

    # Return the user object
    return user

```

This code defines a `User` model with a `username` and `password` column, and creates a database connection using SQLAlchemy. The `login` endpoint is defined to accept a `username` and `password` parameter, and uses the `bcrypt` library to check if the password is correct. If the password is incorrect, a `HTTPException` is raised with a status code of 401 and a detail message of "Invalid username or password". If the password is correct, the `user` object is returned.

Note that this code uses parameterized queries and the ORM provided by SQLAlchemy to avoid SQL injection attacks. It also uses the `bcrypt` library to hash the password before storing it in the database, which makes it more secure than storing plaintext passwords. Finally, the `HTTPException` is used to provide a consistent error message for invalid credentials, which can be customized as needed.

.....

From a quick triage, we can conclude that the “Java” output is actually Python. This behavior suggests that the Python-specialized variant of CodeLlama is unable—or not sufficiently trained—to generalize beyond its target language, preventing it from handling multi-language workloads.