**Milestone 3: AI Model Integration**

**🎯 Objective:**

To integrate the **IBM Watsonx Granite Large Language Model (LLM)** into the backend application through a **centralized service layer**, ensuring modular access, maintainability, and clean abstraction of AI-based capabilities like chat, summarization, eco tips, and report generation.

**🧩 Step 1: Central AI Service Layer Setup**

**📁 File: services/granite\_llm.py**

Encapsulate all Watsonx LLM interactions in a single module:

python

CopyEdit

import requests

from core.config import settings

def query\_watsonx(prompt: str, task: str = "summarize") -> str:

headers = {

"Content-Type": "application/json",

"Authorization": f"Bearer {settings.watsonx\_api\_key}"

}

payload = {

"model\_id": settings.watsonx\_model\_id,

"project\_id": settings.watsonx\_project\_id,

"input": [{"role": "user", "content": prompt}],

"parameters": {

"decoding\_method": "greedy",

"max\_new\_tokens": 200,

"min\_new\_tokens": 50,

}

}

url = f"{settings.watsonx\_url}/ml/v1/text/generation"

response = requests.post(url, json=payload, headers=headers)

if response.status\_code == 200:

return response.json().get("results", [{}])[0].get("generated\_text", "")

else:

raise Exception(f"Watsonx Error [{task}]: {response.status\_code} → {response.text}")

**⚙️ Step 2: Use the Granite LLM in Backend Routes**

**Example: chat\_router.py**

python

CopyEdit

from fastapi import APIRouter, HTTPException

from services.granite\_llm import query\_watsonx

router = APIRouter()

@router.post("/chat")

def get\_chat\_response(prompt: str):

try:

reply = query\_watsonx(prompt, task="chat")

return {"response": reply}

except Exception as e:

raise HTTPException(status\_code=500, detail=str(e))

**🔁 Step 3: Modular Use Across App Features**

You can now use query\_watsonx(prompt) in:

| **Feature** | **Prompt Type** |
| --- | --- |
| 🧠 Chat Assistant | Conversational |
| 📄 Policy Summarizer | Summarization |
| 🌱 Eco Tips Generator | Informational |
| 📝 Report Generator | Analytical Text |

🧩 This pattern keeps Watsonx usage consistent and testable across all modules.

**✅ Deliverables for Milestone 3**

| **Task** | **Description** | **Status** |
| --- | --- | --- |
| 🧠 Granite LLM Service | granite\_llm.py written and tested | ✅ Done |
| 🔗 API Integration | Watsonx integrated in /chat, /summarize, etc. | ✅ Done |
| 🧪 Error Handling | Robust error capture and response messaging | ✅ Done |
| 🧬 Reusability | Central service used across routes | ✅ Done |

**📌 Outcome:**

* Granite LLM integrated via **centralized, reusable service**.
* Enables prompt-driven AI functionality for chat, tips, summaries, and reporting.
* Modular architecture boosts **testability**, **debuggability**, and **scalability**.
* **Activity 3.1: Watsonx Integration**
* **✅ Step 1: Load Environment Variables**
* Use python-dotenv and pydantic-settings to load variables from .env.
* **🔧 core/config.py**
* python
* CopyEdit
* from pydantic\_settings import BaseSettings
* class Settings(BaseSettings):
* watsonx\_api\_key: str
* watsonx\_project\_id: str
* watsonx\_model\_id: str
* watsonx\_url: str
* class Config:
* env\_file = ".env"
* settings = Settings()
* **✅ Step 2: Create granite\_llm.py Service Layer**
* Central file to communicate with Watsonx Granite LLM.
* **📄 services/granite\_llm.py**
* python
* CopyEdit
* import requests
* from core.config import settings
* def query\_watsonx(prompt: str, task: str = "chat") -> str:
* headers = {
* "Authorization": f"Bearer {settings.watsonx\_api\_key}",
* "Content-Type": "application/json"
* }
* payload = {
* "model\_id": settings.watsonx\_model\_id,
* "project\_id": settings.watsonx\_project\_id,
* "input": [{"role": "user", "content": prompt}],
* "parameters": {
* "decoding\_method": "greedy",
* "max\_new\_tokens": 300,
* "min\_new\_tokens": 50
* }
* }
* url = f"{settings.watsonx\_url}/ml/v1/text/generation"
* response = requests.post(url, headers=headers, json=payload)
* if response.status\_code == 200:
* return response.json().get("results", [{}])[0].get("generated\_text", "")
* else:
* raise Exception(f"Watsonx Error: {response.status\_code} - {response.text}")
* **✅ Step 3: Define Feature-Specific Functions**
* Add simple wrappers if needed:
* python
* CopyEdit
* def generate\_summary(text: str) -> str:
* return query\_watsonx(f"Summarize this policy:\n{text}", task="summarize")
* def generate\_eco\_tip(topic: str) -> str:
* return query\_watsonx(f"Give an eco-friendly tip about {topic}", task="eco\_tip")
* def generate\_report(city\_name: str, kpi\_data: str) -> str:
* return query\_watsonx(f"Generate a sustainability report for {city\_name} using this data: {kpi\_data}", task="report")
* **✅ Step 4: Test LLM Endpoints with Dummy Prompts**
* You can create a test file or use Swagger UI to test routes like:
* **Example: test\_granite\_llm.py**
* python
* CopyEdit
* from services.granite\_llm import (
* query\_watsonx,
* generate\_summary,
* generate\_eco\_tip,
* generate\_report
* )
* # Test 1 – Chat
* print(query\_watsonx("How can smart cities reduce air pollution?", task="chat"))
* # Test 2 – Summary
* print(generate\_summary("This policy outlines renewable energy guidelines for public transportation."))
* # Test 3 – Eco Tip
* print(generate\_eco\_tip("water conservation"))
* # Test 4 – Report
* print(generate\_report("Bangalore", "Air Quality: Moderate, Water Usage: High, Energy: Efficient"))
* **✅ Deliverables for Activity 3.1**

| **Task** | **Description** | **Status** |
| --- | --- | --- |
| 🔐 Env Loaded | .env values loaded via config.py | ✅ Done |
| 🧠 Granite LLM Service | granite\_llm.py implemented | ✅ Done |
| 🧪 Dummy Prompt Testing | Sample prompts return AI responses | ✅ Done |

**Activity 3.2: Implement LLM Service Functions**

Objective: Add modular functions to granite\_llm.py for different types of tasks using IBM Watsonx Granite LLM.

**📄 services/granite\_llm.py**

python

CopyEdit

import requests

from core.config import settings

def \_call\_watsonx(prompt: str) -> str:

headers = {

"Authorization": f"Bearer {settings.watsonx\_api\_key}",

"Content-Type": "application/json"

}

payload = {

"model\_id": settings.watsonx\_model\_id,

"project\_id": settings.watsonx\_project\_id,

"input": [{"role": "user", "content": prompt}],

"parameters": {

"decoding\_method": "greedy",

"max\_new\_tokens": 300,

"min\_new\_tokens": 50

}

}

url = f"{settings.watsonx\_url}/ml/v1/text/generation"

response = requests.post(url, json=payload, headers=headers)

if response.status\_code == 200:

return response.json().get("results", [{}])[0].get("generated\_text", "")

else:

raise Exception(f"Watsonx Error {response.status\_code}: {response.text}")

**✅ 1. ask\_granite(prompt: str) — Chat Assistant**

python

CopyEdit

def ask\_granite(prompt: str) -> str:

full\_prompt = f"You are a sustainability assistant. Answer the following question clearly:\n{prompt}"

return \_call\_watsonx(full\_prompt)

**✅ 2. generate\_summary(text: str) — Policy Summarizer**

python

CopyEdit

def generate\_summary(text: str) -> str:

full\_prompt = f"Summarize this smart city policy:\n{text}"

return \_call\_watsonx(full\_prompt)

**✅ 3. generate\_eco\_tip(topic: str) — Eco Tips Generator**

python

CopyEdit

def generate\_eco\_tip(topic: str) -> str:

full\_prompt = f"Provide an actionable eco-friendly tip related to {topic}."

return \_call\_watsonx(full\_prompt)

**✅ 4. generate\_city\_report(kpi\_data: str) — Sustainability Report Generator**

python

CopyEdit

def generate\_city\_report(kpi\_data: str) -> str:

full\_prompt = (

"Generate a smart city sustainability report based on the following KPI data:\n"

f"{kpi\_data}\nInclude insights, recommendations, and an overall sustainability score."

)

return \_call\_watsonx(full\_prompt)

**✅ Final Checklist for Activity 3.2**

| **Function** | **Purpose** | **Status** |
| --- | --- | --- |
| ask\_granite() | Natural language Q&A | ✅ Done |
| generate\_summary() | Policy summarization | ✅ Done |
| generate\_eco\_tip() | Environmental suggestions | ✅ Done |
| generate\_city\_report() | KPI-based reporting | ✅ Done |