Gemini : Example 1:

```
from langchain_google_genai import ChatGoogleGenerativeAI
import os

# Initialize Gemini LLM via LangChain
llm = ChatGoogleGenerativeAI(
    model="gemini-1.5-flash",
    google_api_key="AIza…",
    temperature=0.3
)
prompt = """
Solve the math problem step-by-step, then give the final answer.

Problem: If 12 people sit at a round table, how many unique seating arrangements (up to
rotation) exist?

Step-by-step:
"""

resp = llm.invoke( prompt)
print(resp.content)
```

—----------

Install ollama: !pip install ollama
cmd: ollama pull llama2

Eg1:

```
import ollama

response = ollama.chat(model="llama2", messages=[
    {"role": "user", "content": "Write a short poem about the moon"}
])

print(response["message"]["content"])
```

Eg2:

```
import ollama
from langchain_core.prompts import ChatPromptTemplate
from langchain_ollama import ChatOllama
```

```
def create_directory(dir_name):
        os.makedirs(dir_name, exist_ok=True)
        return f"Directory '{dir_name}' created successfully."

# 2. Connect to local Ollama
response = ollama.chat(
    'llama3.1',
    messages=[{
        'role': 'user',
        'content': 'Create the folder "test-directory"',
    }],
    tools=[create_directory],
)

print(response)

eg3)
import os
from langchain_ollama import ChatOllama
from langchain.agents import Tool, create_tool_calling_agent, AgentExecutor
from langchain.prompts import PromptTemplate

# 1. Define the tool function
import os

def create_directory(dir_input: str) -> str:
    current_path = os.getcwd()
    full_path = os.path.join(current_path, dir_input.strip().replace("/", os.sep))
    os.makedirs(full_path, exist_ok=True)
    return f"Directory '{full_path}' created successfully."


# 2. Wrap it as a LangChain Tool
create_directory_tool = Tool(
    name="create_directory",
    func=create_directory,
    description="Use this tool to create a folder. Provide the full path or name of the folder to be
created."
)

# 3. Connect to Ollama via LangChain
llm = ChatOllama(model="llama3.1")
```

```
# 4. Define the prompt
prompt = PromptTemplate.from_template(
    "You are a helpful assistant that can create folders using the tool provided. "
    "When the user asks to create nested folders, extract the full folder path and use the
'create_directory' tool with it. "
    "Always preserve the folder structure. {agent_scratchpad}"
)



# 5. Create the agent
agent = create_tool_calling_agent(llm=llm, tools=[create_directory_tool], prompt=prompt)

# 6. Wrap in executor
agent_executor = AgentExecutor(agent=agent, tools=[create_directory_tool], verbose=True)

# 7. Run the agent
response = agent_executor.invoke({
    "input": "Create a folder named 'test'. Inside that create a new folder 'docs'"
})

print(response)
```

eg4) https://aistudio.google.com/app/apikey

```
!pip install -U langchain-google-genai
import getpass
import os

if not os.environ.get("GOOGLE_API_KEY"):
  os.environ["GOOGLE_API_KEY"] = 'AIza…..'

from langchain.chat_models import init_chat_model

model = init_chat_model("gemini-2.5-flash", model_provider="google_genai")
model.invoke("Hello, world!")
```

eg) Chain Pattern: Output of one becomes input to other llm

```
!pip install google-generativeai
!pip install langchain-google-genai
```

```
from langchain_google_genai import ChatGoogleGenerativeAI

# Initialize Gemini model
llm = ChatGoogleGenerativeAI(
    model="gemini-1.5-flash",
    google_api_key="AIza..",
    temperature=0.3
)

from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import ChatPromptTemplate
from langchain.chains import LLMChain, SimpleSequentialChain

# Define prompts
analyze_prompt = ChatPromptTemplate.from_template("Analyze this text and identify the main themes:\n\n{text}")
examples_prompt = ChatPromptTemplate.from_template("For each theme, provide specific examples:\n\n{text}")
summary_prompt = ChatPromptTemplate.from_template("Synthesize the themes and examples into a summary:\n\n{text}")
questions_prompt = ChatPromptTemplate.from_template("Generate 3 discussion questions based on:\n\n{text}")

# Define chains
analyze_chain = LLMChain(llm=llm, prompt=analyze_prompt)
examples_chain = LLMChain(llm=llm, prompt=examples_prompt)
summary_chain = LLMChain(llm=llm, prompt=summary_prompt)
questions_chain = LLMChain(llm=llm, prompt=questions_prompt)

# Combine sequentially
overall_chain = SimpleSequentialChain(chains=[analyze_chain, examples_chain, summary_chain, questions_chain])

# Run
sample_text = """
The rise of artificial intelligence has transformed multiple industries in the past decade.
Healthcare has seen improvements in diagnosis accuracy and treatment planning.
Manufacturing has become more efficient with predictive maintenance and automated quality control.
However, these advancements also raise important ethical questions about privacy, job displacement,
and the role of human decision-making in an increasingly automated world.
```

```
"""

final_result = overall_chain.invoke( sample_text)
print(final_result)
```

Eg: 5) Parallel Pattern: single prompt executed with multiple inputs

```
from concurrent.futures import ThreadPoolExecutor
from langchain_google_genai import ChatGoogleGenerativeAI

# Initialize Gemini LLM via LangChain
llm = ChatGoogleGenerativeAI(
    model="gemini-1.5-flash",
    google_api_key="AIza…",
    temperature=0.3
)

def llm_call(prompt_question):
    response = llm.invoke(prompt_question)
    return response.content

def parallel(prompt, inputs, n_workers=3):
    with ThreadPoolExecutor(max_workers=n_workers) as executor:
        futures = [executor.submit(llm_call, f"{prompt}:\n {inp}") for inp in inputs]
    return [f.result() for f in futures]

# Example usage
prompt = "Summarize this paragraph in one sentence"
inputs = [
    """
    Climate change is causing rising sea levels and extreme weather patterns globally.
    Scientists have observed accelerating ice melt in polar regions, leading to coastal
    flooding in low-lying areas. Additionally, communities worldwide are experiencing
    more frequent and severe hurricanes, droughts, and other extreme weather events.
    These changes pose significant risks to agriculture, infrastructure, and human
    populations, particularly in vulnerable coastal regions and developing nations.
    """,
    """
    The development of quantum computers represents a major technological breakthrough
    with far-reaching implications. These powerful machines leverage quantum mechanical
    properties to perform complex calculations exponentially faster than classical computers.
    In the field of cryptography, quantum computers could break many current encryption
    methods while enabling new unbreakable encryption protocols. For drug discovery,
```

```
    quantum computers could simulate molecular interactions with unprecedented accuracy,
    potentially accelerating the development of new medicines and treatments for diseases.
    """,
    """
    Social media platforms have transformed how people communicate and share information
    in the modern digital age. Platforms like Facebook, Twitter, and Instagram have created
    virtual communities where users can instantly connect with friends and family across the
    globe. These platforms enable the rapid spread of news, ideas, and cultural trends,
    while also raising concerns about privacy, misinformation, and the impact on mental
    health. The rise of social media has also revolutionized marketing, activism, and
    how businesses engage with their customers.
    """
]

results = parallel(prompt, inputs)
for r in results:
    print("-", r)
```

Eg6:

Pydantic is a Python library used for data validation and data parsing.

It lets you define models (classes) where you describe the shape of your data (fields, types, constraints), and then it makes sure any input matches those rules — otherwise it raises an error.

Example without Pydantic

```
data = {"name": "Alice", "age": "30"}  # age is a string, not int
```

If you use this directly, `"30"` is still a string — you might forget to convert it, and bugs creep in.

Example with Pydantic

```
from pydantic import BaseModel


class Person(BaseModel):
```

```
    name: str
    age: int   # must be int


person = Person(**{"name": "Alice", "age": "30"})
print(person.age)  # 30 (Pydantic auto-converts string → int)
```

✅ **Pydantic:**

- **Converts `"30"` into `30` automatically**
- **Raises an error if the field is missing or invalid**

Why use it with LLMs?

When an LLM gives you JSON/text, you can validate it with Pydantic:

- **If the model says `{"selected_profile": "banana"}`, validation fails because `"banana"` is not in `[hr, software engineer, product manager]`.**
- **Ensures your app only processes clean, expected data.**

```
from pydantic import BaseModel, Field

class User(BaseModel):
   username: str = Field(min_length=3, max_length=20, regex="^[a-zA-Z0-9_]+$")
   email: str
```

min_length, max_length → string length limits

regex → must match regex pattern

3. Numbers

```
class Product(BaseModel):
    price: float = Field(gt=0)       # strictly greater than 0
    discount: float = Field(ge=0)    # greater or equal to 0
```

```
    stock: int = Field(le=1000)      # less or equal to 1000
```

- **`gt`, `ge` → greater than, greater or equal**
- **`lt`, `le` → less than, less or equal**

---

4. Lists / Arrays

```
from typing import List

class Order(BaseModel):
    items: List[str] = Field(min_items=1, max_items=10)
```

- **`min_items`, `max_items` restrict list length**

---

5. Enums / Choices

```
from enum import Enum

class Role(str, Enum):
    admin = "admin"
    user = "user"
    guest = "guest"

class Person(BaseModel):
    role: Role
```

✅ **Enforces allowed values.**

Or simpler with `Literal`:

```
from typing import Literal
```

```
class Person(BaseModel):
    role: Literal["admin", "user", "guest"]
```

---

6. Nested Models

```
class Address(BaseModel):
    city: str
    zip_code: str

class User(BaseModel):
    name: str
    address: Address
```

✅ **Auto-parses nested dicts into models.**

---

7. Default Values & Optional

```
from typing import Optional

class User(BaseModel):
    nickname: Optional[str] = None
    age: int = 18  # default
```

---

8. Custom Validators

Pydantic v1

```
from pydantic import BaseModel, validator

class User(BaseModel):
    age: int

    @validator("age")
    def check_age(cls, v):
```

```
        if v < 0:
            raise ValueError("Age must be positive")
        return v
```

Pydantic v2

```
from pydantic import BaseModel, field_validator

class User(BaseModel):
    age: int

    @field_validator("age")
    def check_age(cls, v):
        if v < 0:
            raise ValueError("Age must be positive")
        return v
```

---

9. Strict Mode

```
from pydantic import BaseModel, StrictStr, StrictInt

class User(BaseModel):
    name: StrictStr   # must be a str, no coercion
    age: StrictInt    # must be an int, "25" will fail
```

Example: Router Pattern

```
from enum import Enum
from typing import List
from pydantic import BaseModel, Field
import google.generativeai as genai
import json
import os
import  re

# Configure Gemini client
genai.configure(api_key="AIza….")
```

```
# Pydantic model for validation
class ProfileRouter(BaseModel):
    selected_profile: str = Field(description="One of [hr, software engineer, product manager]")

def llm_call_route(problem: str) -> str:
    """Selects which profile should solve the problem."""

    prompt = f"""
    You are a router. You must select ONE profile to solve the given problem.
    Allowed profiles:
    - hr
    - software engineer
    - product manager

    Problem: {problem}

    Respond ONLY in JSON with this format:
    {{
      "selected_profile": "software engineer"
    }}

    No markdown, no code fences, no explanations.
    """

    model = genai.GenerativeModel("gemini-1.5-flash")
    response = model.generate_content(prompt)

    raw_text = response.text.strip()
    print(raw_text)
    #  🔹 Clean up code fences if Gemini adds them
    if raw_text.startswith("```"):
        raw_text = re.sub(r"^```[a-zA-Z]*\n", "", raw_text)  # remove opening fence
        raw_text = raw_text.rstrip("`")                      # remove closing fence

    #  🔹 Parse JSON safely
    try:
        parsed = json.loads(raw_text)
        validated = ProfileRouter(**parsed)
        return validated.selected_profile
    except Exception as e:
        raise ValueError(f"Invalid response: {raw_text}") from e


# Example usage
```

```
print(llm_call_route("I need to fix this bug in our codebase"))
```

Eg : Create subdirectories and folders with langchain

```
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.tools import tool
from langchain.agents import initialize_agent, AgentType
import os

os.environ["GOOGLE_API_KEY"] ='AIza….'

def create_path(path: str, is_file: bool = False, content: str = "") -> str:
    """Utility to create file or folder with optional content."""
    if is_file:
        os.makedirs(os.path.dirname(path), exist_ok=True)
        with open(path, "w") as f:
            f.write(content)
        return f"File created at {path}"
    else:
        os.makedirs(path, exist_ok=True)
        return f"Directory created at {path}"

# Wrap our function as a LangChain tool
@tool
def create_path_tool(path: str, is_file: bool = False, content: str = "") -> str:
    """Create a file or folder with optional content."""
    return create_path(path, is_file, content)

# Initialize Gemini model
llm = ChatGoogleGenerativeAI(model="gemini-1.5-flash", temperature=0.2)
# Build agent with our tool
agent = initialize_agent(
    tools=[create_path_tool],
    llm=llm,
    agent=AgentType.STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION,  # <--
works with Gemini
    verbose=True,
)
# Example user request
prompt = """
Create a directory called 'project' with subfolders 'src' and 'tests'.
Inside 'src', create a file 'main.py' with content: print("Hello Gemini").
```

```
Inside 'tests', create a file 'test_main.py' with content: assert 1+1==2.
"""

agent.run(prompt)
```