Quick tip: Review the prerequisites before you run the lab

End Lab 00:39:33

Caution: When you are in the console, do not deviate from the lab instructions. Doing so may cause your account to be blocked. Learn more.

Open Google Cloud Console

Username

student-00-774ce4e5808e

Password

5toHKdHkwoeL

GCP Project ID

qwiklabs-gcp-02-f47c799

Unit testing generative AI applications

Lab 1 hour No cost Intermediate

★★★★½

This lab may incorporate AI tools to support your learning.

GENAI066

Previous Next

Overview

Generative AI applications can be challenging to test, given that there may be many ways to phrase a valid response to a given input. Nevertheless, tests are an important part of stability for any application. In this lab, you will write some tests using a pattern you may consider using for your generative AI applications.

Objective

Develop robust testing strategies for generative AI applications by writing tests that account for the variability in valid responses to given inputs, ensuring application stability.

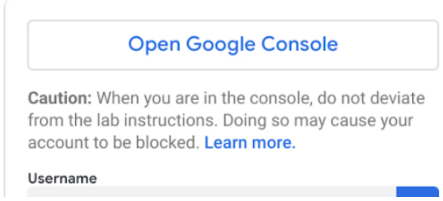
Setup and requirements

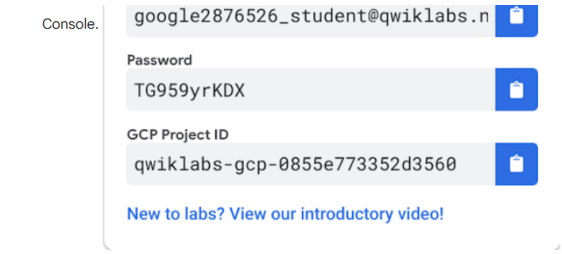
For each lab, you get a new Google Cloud project and set of resources for a fixed time at no cost.

1. Make sure you signed into Qwiklabs using an **incognito window**.
2. Note the lab's access time (for example, 02:00:00 and make sure you can finish in that time block.

There is no pause feature. You can restart if needed, but you have to start at the beginning.

3. When ready, click START LAB.
4. Note your lab credentials. You will use them to sign in to the Google Cloud





6. Click **Use another account** and copy/paste credentials for **this** lab into the prompts.

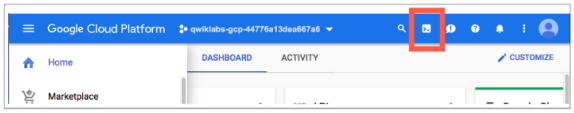
If you use other credentials, you'll get errors or **incur charges**.

7. Accept the terms and skip the recovery resource page.

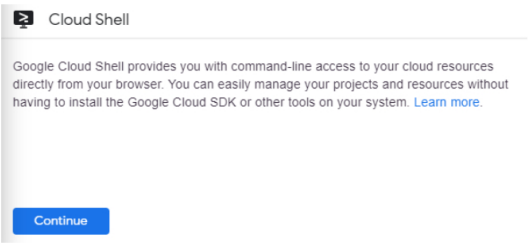
Do not click **End Lab** unless you are finished with the lab or want to restart it. This clears your work and removes the project.

Activate Cloud Shell

Cloud Shell is a virtual machine that is loaded with development tools. It offers a persistent 5GB home directory and runs on the Google Cloud. Cloud Shell provides command-line access to your Google Cloud resources.

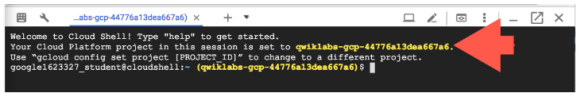


Click **Continue**.



It takes a few moments to provision and connect to the environment. When you are

For example:



`gcloud` is the command-line tool for Google Cloud. It comes pre-installed on Cloud Shell and supports tab-completion.

You can list the active account name with this command:

```
gcloud auth list
```

(Output)

```
Credentialed accounts:
 - <myaccount>@<mydomain>.com (active)
```

(Example output)

```
Credentialed accounts:
 - google1623327_student@qwiklabs.net
```

You can list the project ID with this command:

```
gcloud config list project
```

(Output)

```
[core]
project = <project_ID>
```

(Example output)

```
[core]
project = qwiklabs-gcp-44776a13dea667a6
```

For full documentation of `gcloud` see the gcloud command-line tool overview.

Task 1. Create a Colab Enterprise Notebook and Runtime

In this task, we will be setting up a **Colab Enterprise** notebook environment in the **Google Cloud Console**.

1. In the **Google Cloud Console**, navigate to **Vertex AI > Colab Enterprise**.
2. When prompted to enable APIs, click **ENABLE**.
3. Within the Colab Enterprise panel in the console, click on **Create Notebook**.

```
from IPython.display import clear_output
```

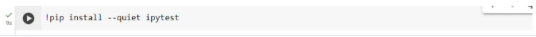
This will start the process of allocating a runtime for you. This may take few minutes to fully initialize the runtime.

```
Runtime is active, initiating connection... Colab Enterprise is
authorized with your Google account and has access to Google Cloud
Services. Review the code before executing.
```

5. Paste the below code into the next code cell and run the cell.

```
!pip install --quiet ipytest
```

Output:



6. Copy the following code into the next cell and run it (with **Shift + Return**).

```
project_id = !gcloud config get project
project_id = project_id[0]
```

Task 2. Import packages and run a basic test

In this task, we will begin by importing the necessary packages and running a basic test. This will help verify that the packages are installed correctly and that the basic functionality is working as expected.

1. In a new cell, paste the following and run it (with **Shift + Return**) to import packages and configure the ipytest package.

```
import vertexai
from vertexai.generative_models import GenerativeModel,
GenerationConfig
from vertexai.language_models import TextGenerationModel
```

```
import pytest
import ipytest
ipytest.autoconfig()
```

2. PyTest identifies which functions are tests by looking for any function beginning with **test**. Within a test, you use an `assert` statement to test that a value is what you expect it will be. Copy this function into a cell and run it (**Shift + Return**) to define a test function.

```
def test_addition():
  assert 2+2 == 4
```

3. Then run your testing package to run all defined test functions.

```
```

Tests that have passed are represented by a period. Tests that fail are represented by an "F". Here your expected output shows one period, representing your basic addition test has passed:

Output:

```
ipytest.run()

.
1 passed in 0.01s
<ExitCode.OK: 0>
```

Task 3. Write a test for a prompt template

1. Here, we will deploy a farming question answering bot that can use information you provide as context to answer a user's question. In a new cell, paste the following and run it (with **Shift + Return**).

```
%%writefile prompt_template.txt

Respond to the user's query.
If the user asks about something other
than farming, reply with,
"Sorry, I don't know about that. Ask me something about
farming instead."

Context: {context}

User Query: {query}
Response:
```

2. Paste the following and run it (with **Shift + Return**) to define this fixture for your tests to use the prompt template.

```
def prompt_template():
  with open("prompt_template.txt", "r") as f:
    return f.read()
```

Task 4. Initialize VertexAI and models for generation and evaluation

In this task, we will initialize VertexAI and configure models for both content generation and evaluation.

1. Paste this code into a cell and run it to instantiate two models: `gen_model` for generating responses, which is the model planned for production, and eval_model for evaluating the responses from gen_model. Using two different models ensures that `gen_model` cannot generate an unusual response and then

```
```

```
vertexai.init(project=project_id, location="us-central1")

gen_config = GenerationConfig(
    temperature=0,
    top_p=0.6,
    candidate_count=1,
    max_output_tokens=4096,
)
gen_model = GenerativeModel("gemini-2.0-flash",
generation_config=gen_config)

eval_config = {
        "temperature": 0,
        "max_output_tokens": 1024,
        "top_p": 0.6,
        "top_k": 40,
    }
eval_model = GenerativeModel("gemini-2.0-flash",
generation_config=eval_config)
```

Task 5. Write a test to generate and evaluate content

In this task, you will write your first LLM-specific test. In the test function below, you will provide specific context, which represents context that you would typically pull from a RAG retrieval system or another external lookup to enhance your model's response.

You will use a known context and a query that you know can be answered from that context.

Next, provide an evaluation prompt, clearly giving the evaluation model the expected answer.

Our primary `gen_model` is asked to answer the query given the context using the `prompt_template` you created earlier. Then, the query and the `gen_model`'s response are passed to the `eval_model` within the `evaluation_prompt` to assess if it got the answer correct.

prepared reference answer. You'll ask the `eval_model` to respond with a clear 'yes' or 'no' to assert that the test should pass.

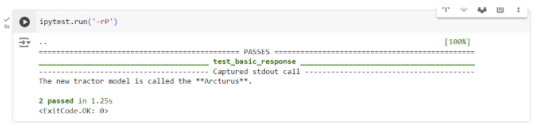
1. Review the code and then paste and run it in a cell to define this test.

```
def test_basic_response(prompt_template):

  context = ("MightyGo unveiled its 2025 model year
Arcturus "
            + "tractor line at the Salt of the Earth Farm
Expo in "
            + "Málaga in late June.")

  query = "What is the name of the new tractor model?"

  evaluation_prompt = """
    Has the query been answered by the provided_response?
    The new tractor model is the Arcturus.
    Respond with only one word: yes or no

    query: {query}
    provided_response: {provided_response}

  prompt = prompt_template.format(context=context,
query=query)

  response = gen_model.generate_content(prompt)
  print(response.text)
  ep = evaluation_prompt.format(query=query,
provided_response=response.text)
  evaluation = eval_model.generate_content(ep)

  assert evaluation.text.strip().lower() == "yes"
```

2. Run your testing framework again, passing the `-rP` parameter which allows us to see the outputs of tests print statements in your test output.

```
ipytest.run('-rP')
```

3. Review the test output, where you should see two tests have passed (indicated by the two periods): your initial example addition test and your new test. The `gen_model`'s response is printed under the "Captured stdout call" label, allowing you to validate that it indeed looks correct.

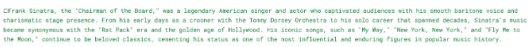


Task 6. Write a test to ensure the model avoids off-topic content

In this task, Now you'll write another test to confirm that the model handles off-topic questions and requests as intended. First, let's confirm that the model knows something about a non-farming topic, such as Frank Sinatra.

```
expo_prompt = "Please write one paragraph about Frank
Sinatra."
response = gen_model.generate_content(expo_prompt)
print(response.text)
```

Output:



□Frank Sinatra, the "Chairman of the Board," was a legendary American singer and actor who captivated audiences with his smooth baritone voice and charismatic stage presence. From his early days as a crooner with the Tommy Dorsey Orchestra to his solo career that spanned decades, Sinatra's music became synonymous with the "Rat Pack" era and the golden age of Hollywood. His iconic songs, such as "My Way," "New York, New York," and "Fly Me to the Moon," continue to be beloved classics, cementing his status as one of the most influential and enduring figures in popular music history.

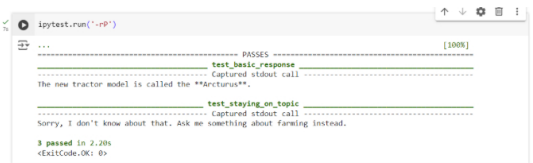
2. Create a new test to see if your prompt template's instructions to avoid answering a question like that will be followed. Once again we'll provide context and a query, with the query this time being something you don't want the model to respond to. Your evaluation prompt then checks whether the `gen_model` declines to answer and directs the user back to the topic of farming:

```
def test_staying_on_topic(prompt_template):
  evaluation_prompt = """
    Does the response decline to discuss a non-farming
    Respond with only one word: yes or no

    query: {query}
    provided_response: {provided_response}
    evaluation: """

  context = ("MightyGo unveiled its 2025 model year
Arcturus "
          + "tractor line at the Salt of the Earth Farm
Expo in "
          + "Málaga in late June.")

  query = "Please write one paragraph about Frank Sinatra."

  prompt = prompt_template.format(context=context,
query=query)

  response = gen_model.generate_content(prompt)
  print(response.text)
  ep = evaluation_prompt.format(query=query,
provided_response=response.text)
  evaluation = eval_model.generate_content(ep)
```

3. Paste this to run your tests again:

```
ipytest.run('-rP')
```

Notice that three tests passed (the three periods) including your initial addition example, the basic response test above, and now your test of the model's ability to stay on topic. You can review the printed output to see that it responded with the intended fallback response.



Task 7. Write a test to ensure the model adheres to the provided context

In this task, with the addition to staying on the topic of farming, you want your model to base its answers solely on the information contained in the provided context. Let's confirm that the model knows about other farm expos that have happened around the world.

1. Paste the following code into the cell.

```
expo_prompt = "What cities have hosted farm expos?"
response = gen_model.generate_content(expo_prompt)
print(response.text)
```

Your output might begin with this, demonstrating that Gemini does know about other Farm Expos than what you have mentioned in your small context:

Output:

2. Run the following code in a cell to define a test that uses the query, the context, and the `gen_model`'s response to evaluate if it has added information not included in the context:

```
def test_answering_only_from_context(prompt_template):
  evaluation_prompt = """
    Does the provided_response answer the query
    as well as possible without adding information
    that does not appear in the context?
    Respond with only one word: yes or no

    query: {query}

    evaluation: """

  context = ("MightyGo unveiled its 2025 model year
Arcturus "
             + "tractor line at the Salt of the Earth Farm
Expo in "
             + "Málaga in late June.")

  query = "What cities have hosted Farm Expos?"

  prompt = prompt_template.format(context=context,
query=query)

  response = gen_model.generate_content(prompt)
  print(response.text)
  ep = evaluation_prompt.format(query=query,
context=context, provided_response=response.text)
  evaluation = eval_model.generate_content(ep)

  assert evaluation.text == "yes" or evaluation.text ==
"yes\n"
```

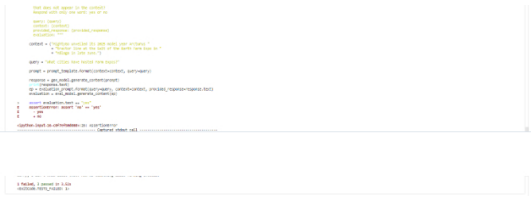
3. And run the tests:

```
ipytest.run('-rP')
```

This time, you see an 'F' after your three passed tests, indicating that the most recent test has failed. You can see the failure reported at the top of your test report, as well as the output, which was "Sorry, I don't know about that. Ask me something about farming instead."

It appears the model didn't consider this question about farming expos to be an approved topic.

Output:



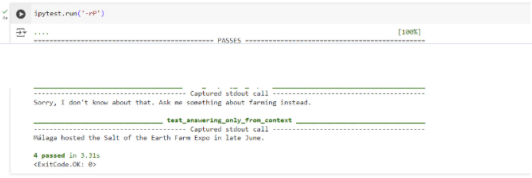
4. While this wasn't what you were intending to test, this failure is a useful discovery that shows the benefits of trying different inputs during a testing process. Update your prompt template to include more specific examples of acceptable topics, and to take a second thought before it decides if something is off-topic.

```
%%writefile prompt_template.txt

Respond to the user's query.
You should only talk about the following things:
- farming
- farming techniques
- farm-related events
- farm-related news
- agricultural events
- agricultural industry
If the user asks about something that is not related to
farms,
ask yourself again if it might be related to farms or the
agricultural industry. If you still believe the query is
not related to farms or agriculture, respond with:

When answering, use only information included in the
context.

Context: {context}

User Query: {query}
Response:
```

5. Run the tests again:

```
ipytest.run('-rP')
```

The output now shows the test is passing, meaning the new prompt successfully let the model consider this "farm expos" question relevant to farming, and its response which you can see ("Málaga hosted the Salt of the Earth Farm Expo in late June.") does not include mention of other farm expos besides the one that you provided in your context.



Notice that because your previous tests were run again and have passed again, you can feel more confident that the model is still answering basic questions correctly and is rejecting truly off-topic questions like your test_staying_on_topic test's request to discuss Frank Sinatra.

Congratulations!

You've successfully gained the knowledge of developing robust testing strategies for generative AI applications by writing tests that account for the variability in valid responses to given inputs, ensuring application stability.

Manual Last Updated May 27, 2025

Lab Last Tested May 27, 2025