Install Stuff

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

```
[ ] from IPython.display import clear_output
```

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

```
[ ] !pip install --quiet ipytest
```

```
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 24.9 MB/s eta 0:00:00
```

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

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.

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()
```

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.

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

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

```
[ ] ipytest.run()
```

```
    .                                                                    [100%]
    1 passed in 0.01s
    <ExitCode.OK: 0>
```

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:

Write a test for a prompt template

In this task, we will use the %%writefile cell magic command to save your application's primary content generation prompt template to a file.

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,

```
[ ] %%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:
```

```
    Writing prompt_template.txt
```

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

```
[ ] @pytest.fixture
    def prompt_template():
      with open("prompt_template.txt", "r") as f:
        return f.read()
```

Initialize VertexAI and models for generation and evaluation

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

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 inaccurately evaluate itself as having given a good response.

```
[ ] 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)
```

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.

The eval_model can evaluate if the substance of the response is correct, even if the generative model has responded with full sentences that may not exactly match a pre-prepared reference answer. You'll ask the eval_model to respond with a clear 'yes' or 'no' to assert that the test should pass.

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}
        evaluation: """

      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"
```

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

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.

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

```
..                                                                       [100%]
============================================ PASSES ============================================
_____________________________________ test_basic_response _____________________________________
------------------------------------- Captured stdout call -------------------------------------
The new tractor model is called Arcturus.

2 passed in 0.72s
<ExitCode.OK: 0>
```

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.

Paste this code into a cell and run it to confirm that the model could indeed answer a question like this if not restricted by your prompt template:

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

```
Frank Sinatra, nicknamed "Ol' Blue Eyes" and "The Voice," was one of the most iconic and influential singers of the 20th century. His smooth baritone voice, impeccable phr
```

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 related topic
        and encourage the user to ask about farming instead?
        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)

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

Paste this to run your tests again:

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

```
...                                                                      [100%]
========================================== PASSES ==========================================
_____________________________________ test_basic_response _____________________________________
----------------------------------------- Captured stdout call -----------------------------------------
The new tractor model is called Arcturus.

_____________________________________ test_staying_on_topic _____________________________________
----------------------------------------- Captured stdout call -----------------------------------------
Sorry, I don't know about that. Ask me something about farming instead.

3 passed in 1.24s
<ExitCode.OK: 0>
```

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.

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.

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)
```

```
Farm expos, also known as agricultural fairs or trade shows, are held in many cities around the world. Here are some examples of cities that have hosted significant farm e

**United States:**

*   **Decatur, Illinois:** Home to the Farm Progress Show, one of the largest outdoor farm shows in the U.S.
*   **Boone, Iowa:** Another location for the Farm Progress Show (it alternates between Iowa and Illinois).
*   **Louisville, Kentucky:** Host of the National Farm Machinery Show, the largest indoor farm show in the U.S.
*   **Tulare, California:** Home to the World Ag Expo, a major agricultural exposition.
*   **Hershey, Pennsylvania:** Site of the Pennsylvania Farm Show, one of the largest indoor agricultural events in the country.
*   **Indianapolis, Indiana:** Host of the Performance Racing Industry Trade Show, which includes many vendors related to agricultural machinery and technology.
*   **Various State Capitals:** Many state capitals host their respective state's agricultural fairs (e.g., Sacramento, California; Springfield, Illinois; Des Moines, Iowa

**Canada:**

*   **Regina, Saskatchewan:** Home to Canada's Farm Progress Show.
*   **Toronto, Ontario:** Host of the Royal Agricultural Winter Fair.
*   **Edmonton, Alberta:** Site of Farmfair International.

**Europe:**

*   **Hanover, Germany:** Host of Agritechnica, one of the world's largest agricultural machinery exhibitions.
*   **Paris, France:** Site of the Salon International de l'Agriculture (SIA), a major agricultural show.
*   **Bologna, Italy:** Home to EIMA International, an international exposition of machinery for agriculture and gardening.
*   **Zaragoza, Spain:** Host of FIMA, International Fair of Agricultural Machinery.
*   **Stoneleigh Park, Warwickshire, UK:** Former home of the Royal Show (no longer held).

**Asia:**
```

```
*   **Bangkok, Thailand:** Site of Agritechnica Asia.
*   **New Delhi, India:** Host of various agricultural expos and trade shows.
*   **Tokyo, Japan:** Site of Agri Week Tokyo.
*   **Shanghai, China:** Host of various agricultural technology and equipment exhibitions.

**Australia:**

*   **Melbourne, Victoria:** Site of Agribition Australia.
*   **Toowoomba, Queensland:** Home to AgQuip.

**South America:**

*   **Ribeirão Preto, Brazil:** Host of Agrishow, one of the largest agricultural technology fairs in Latin America.
*   **Buenos Aires, Argentina:** Site of Exposición Rural.

**Important Considerations:**

*   **Scale and Focus:** Farm expos vary greatly in size and focus. Some are massive international events showcasing the latest technology, while others are smaller, regi
*   **Recurring Events:** Many farm expos are annual or biennial events, so the host city remains consistent.
*   **Changing Locations:** Some expos rotate between different cities or locations.

This list is not exhaustive, but it provides a good overview of cities that are known for hosting significant farm expos. To find specific events, it's best to search onl
```

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}
        context: {context}
        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 = "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"
```

And run the tests

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.

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

```
...F                                                                    [100%]
========================================== FAILURES ==========================================
_____________________________ test_answering_only_from_context ______________________________

prompt_template = '\nRespond to the user\'s query.\nIf the user asks about something other\nthan farming, reply with,\n"Sorry, I don\'t know about that. Ask me something a

    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}
        context: {context}
        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 = "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"
E     AssertionError: assert ('no\n' == 'yes'
E
E       - yes
E       + no or 'no\n' == 'yes\n'
E
E       - yes
E       + no)

<ipython-input-17-cc0f2a13a4c2>:26: AssertionError
---------------------------------------- Captured stdout call ----------------------------------------
Sorry, I don't know about that. Ask me something about farming instead.

========================================== PASSES ==========================================
                                    test_basic_response
```

```
_____ test_basic_response _____
---------- Captured stdout call ----------
The new tractor model is called Arcturus.

_____ test_staying_on_topic _____
---------- Captured stdout call ----------
Sorry, I don't know about that. Ask me something about farming instead.

1 failed, 3 passed in 2.06s
<ExitCode.TESTS_FAILED: 1>
```

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.

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:
    "Sorry, I don't know about that. Ask me something about farming instead."
    When answering, use only information included in the context.

    Context: {context}

    User Query: {query}
    Response:
```

```
Overwriting prompt_template.txt
```

Run the tests again

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

```
....                                                                     [100%]
========== PASSES ==========
_____ test_basic_response _____
---------- Captured stdout call ----------
The new tractor model is called Arcturus.

_____ test_staying_on_topic _____
---------- Captured stdout call ----------
Sorry, I don't know about that. Ask me something about farming instead.

_____ test_answering_only_from_context _____
---------- Captured stdout call ----------
Málaga has hosted a Farm Expo.

4 passed in 1.93s
<ExitCode.OK: 0>
```

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.

```
[ ]
```