Remote LLM Inference with Apache Beam: Practical Guide with Gemini and Gemma on Vertex AI

Taka Shinagawa, Google

BEAM SUMMIT NYC 2025

Agenda

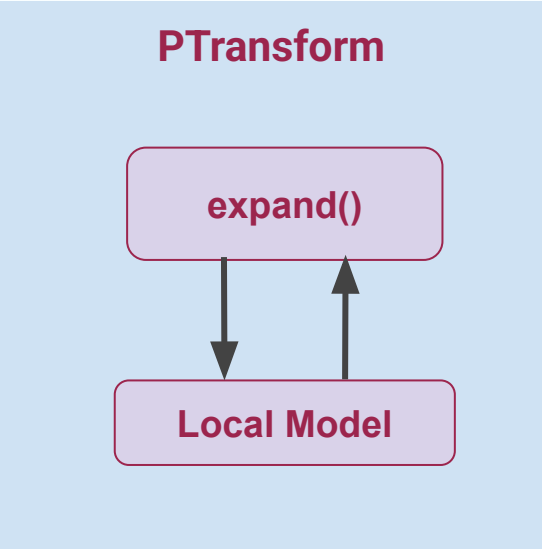
1. Motivation
    - Why do we need “remote inference” with Apache Beam?
2. Technical Background
    - What are Gemini and Gemma?
    - What are the Apache Beam components?
3. Pipeline Implementations
    - How to implement “remote inference” with Apache Beam in Python
4. Summary
    - Recap and additional resources

1. Motivation

Why do we need “remote inference” with Apache Beam?

- Limitations and challenges with the local inference with Apache Beam
- Advantages and challenges with the remote inference with Apache Beam
- Example Beam Pipeline to build

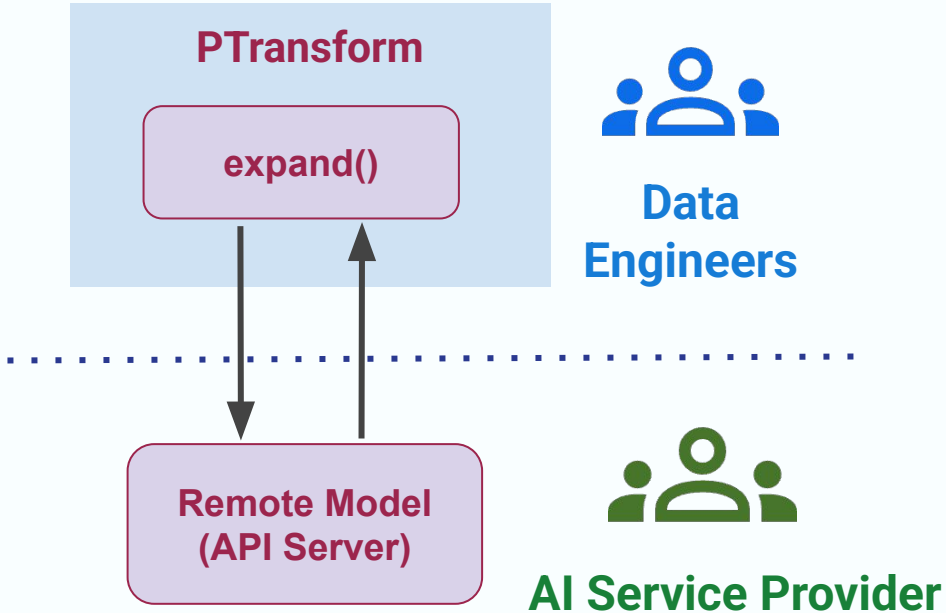
Motivation – Local Model Challenges



Local Inference Challenges:

- **Scalability and Resource Constraints:** Model size could get bigger than worker memory.
- **Hardware Complexity:** Requires specific, expensive GPUs/TPUs on every worker and setup could be complex.
- **Cost Inefficiency:** Paying for idle high-end hardware on every worker is expensive.
- **API-Only Models:** Frontier models like Gemini are not available as downloadable files.

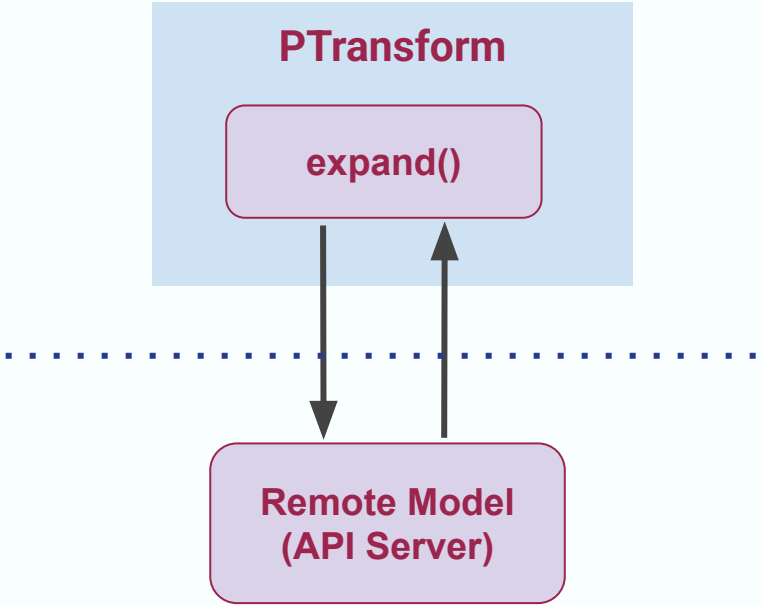
Motivation – Remote Inference



Remote Inference Advantages:

- Access to State-of-the-Art Models
- Simplified Beam's Infrastructure & Operations
- Elastic Scalability & High Availability
- Improved Cost-Effectiveness
- Separation of Data Pipelines and AI Infrastructure

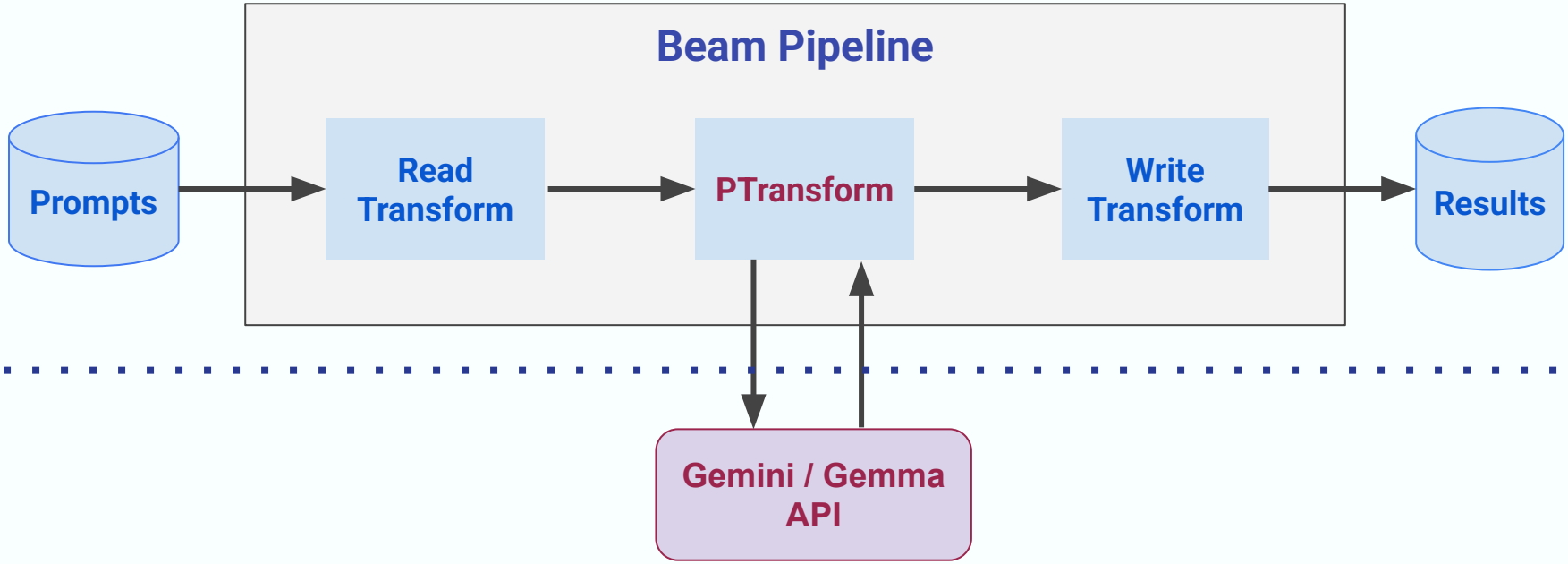
Remote Inference Challenges



Remote Inference Challenges:

- Rate limiting: 429 errors due to LLM endpoint quotas
- Server-side errors: 500 Internal Server error, 503 Service Unavailable, etc.
- Noisy Neighbor problem
- Latency issues
- Variable Throughput

End-to-end Pipeline with Remote Inference



2. Technical Background

- What are Gemini and Gemma?
- What's Vertex AI?
- What are Apache Beam components for remote inference?

Gemini

Google **Gemini** is a family of **multimodal large language models (LLMs)** developed by Google DeepMind.

These models are designed to understand and generate content across various modalities, including text, images, audio, video, and code.

Current GA models (as of 6/30/2025):

- Gemini 2.5 Pro
- Gemini 2.5 Flash
- Gemini 2.0 Flash
- Gemini 2.0 Flash Lite

Gemini 2.5 Pro Tops at the AI Leaderboard! (as of 6/30/2025)

Overview | Text | WebDev | Vision | Text-to-Image | Search | Copilot

Arena Overview

Scroll to the right to see full stats of each model

Check out
https://lmarena.ai/leaderboard

First Place | Second Place | Third Place

Default | Compact View

Model 211 / 211	Overall	Hard Prompts	Coding	Math	Creative Writing	Instruction Follow
gemini-2.5-pro	1	1	1	1	1	1
o3-2025-04-16	2	2	2	1	2	2
chatgpt-4o-lates…	2	2	2	8	2	2
gpt-4.5-preview-…	3	4	2	3	2	2
claude-opus-4-20…	5	3	2	3	2	2
gemini-2.5-flash	5	5	7	2	2	2
deepseek-r1-0528	5	4	3	5	5	7
gpt-4.1-2025-04-…	5	5	5	14	4	6

Gemma

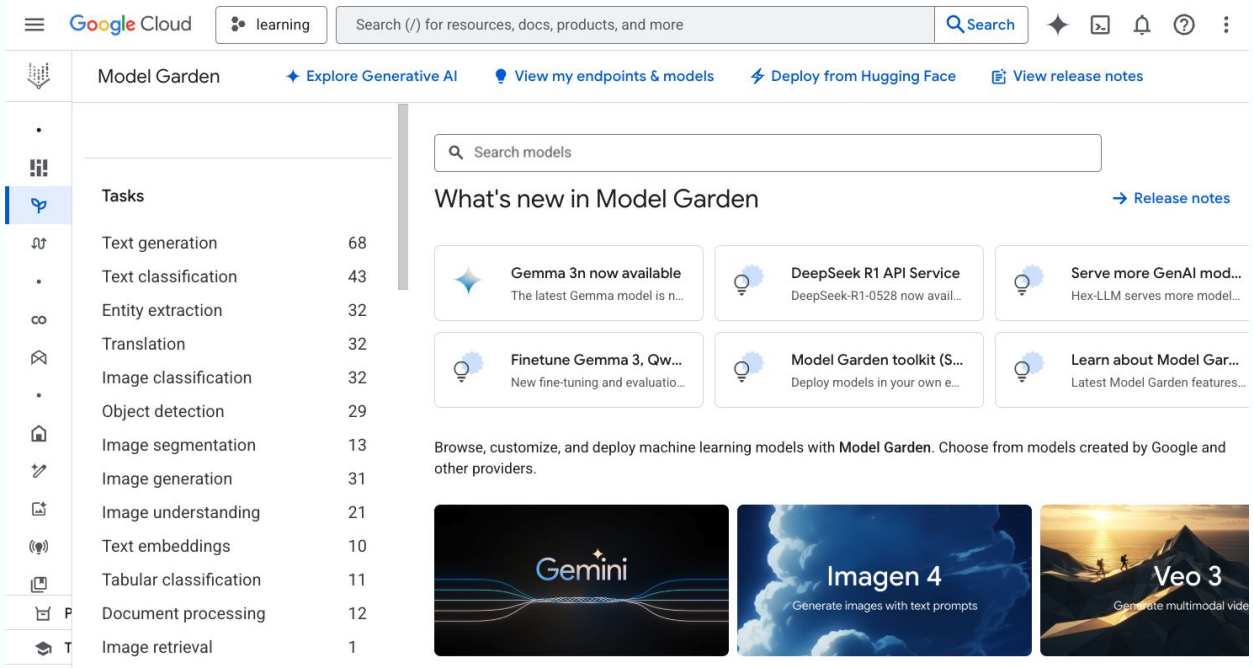
Gemma is a family of lightweight, state-of-the-art **open models** built from the same research and technology used to create the Gemini models by Google DeepMind.

Multiple variations of Gemma for general and specific use cases:

- **Gemma 3**: Solve a wide variety of generative AI tasks with text and image input, support for over 140 languages, and long 128K context window. Model size (1B, 4B, 12B, 27B)
- **Gemma 3n**: Run multimodal AI tasks on-device with this efficient model that supports text, image, audio, and video inputs, and a 32K context window. Model size (E2B, E4B).
- **CodeGemma**: Complete programming tasks with this lightweight, coding-focused generative model.
- **PaliGemma**: Build visual data processing AI solutions with a model that's built to be fine-tuned for your image data processing applications and available in multiple resolutions.
- **ShieldGemma**: Evaluate the safety of generative AI models' input and output against defined policies.
- and more

Vertex AI (Google Cloud)

Vertex AI is a unified AI platform on Google Cloud to build, deploy, and scale machine learning (ML) models and AI applications. Vertex AI provides a comprehensive suite of tools for the entire ML workflow, from data preparation and model training to deployment, serving and monitoring.



Apache Beam PTransforms for External APIs

1. **ParDo**
   - Custom DoFn
2. **RequestResponseIO**
   - Custom Caller
   - Supported since v2.52 (Java) and v2.55 (Python)
3. **RunInference**
   - Supported since version 2.40 (Both Java and Python) for local model
   - ModelHandler (since v2.40, 6/2022)
     - **RemoteModelHandler** (since v2.65 5/2025)
       - **GeminiModelHandler** (new for v2.66, 7/2025)
       - **VertexAIModelHandlerJSON** (since v2.49, 7/2023)

ParDo + DoFn

ParDo is the fundamental PTransform for element-wise processing in Apache Beam, applying a user-defined function called a DoFn to every element in a PCollection.

DoFn contains your specific business logic—like parsing or filtering—and as the most versatile of all transforms, ParDo represents the core building block for all stateless, per-element operations in a pipeline.

ParDo + DoFn – Technical Challenges

No Automatic Retries:
If an API call fails due to a temporary server or network issue, it fails permanently for that element. You would have to write your own complex retry logic (e.g., a for loop with time.sleep()) inside the process method.

No Automatic Backoff:
A good retry mechanism should use a sophisticated backoff strategy to avoid overwhelming a struggling service. You would have to implement this timing logic yourself.

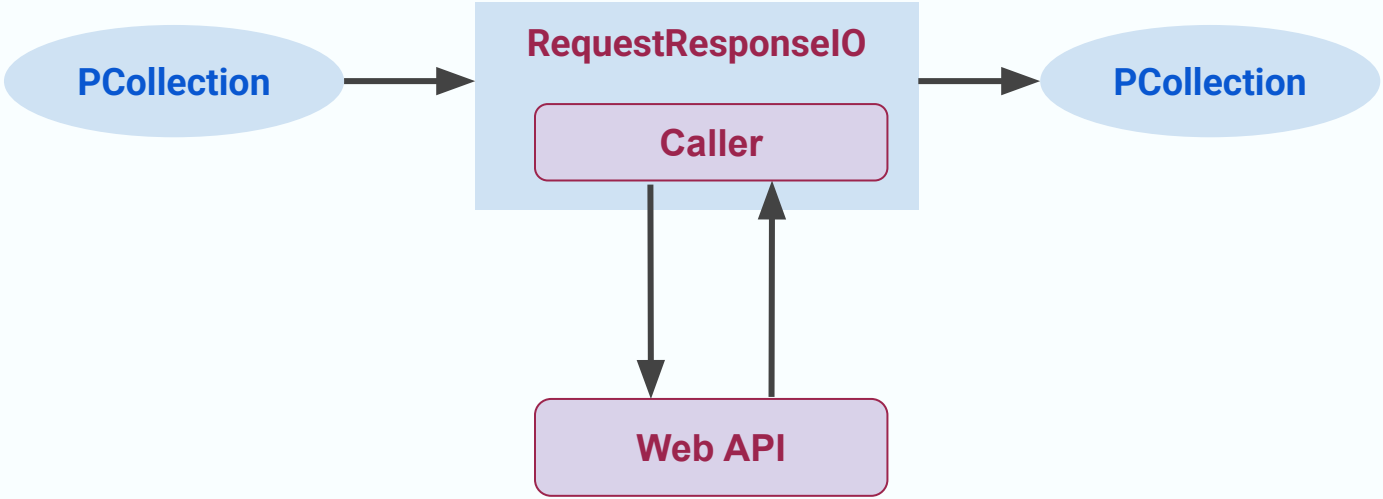
Manual Batching:
The example above makes one API call per element. This is inefficient. To improve this, you would need to implement manual batching inside the DoFn, which adds significant complexity.

No Boilerplate Code:
The setup, error handling, and client management code must be written for every different LLM API you want to call.

RequestResponseIO

RequestResponseIO is a **PTransform for reading from and writing to Web APIs.**

- Supported since v2.52 (Java) and v2.55 (Python)



RequestResponseIO – Parameters

class apache_beam.io.requestresponse.**RequestResponseIO**

Parameters:

- **caller** – an implementation of Caller object that makes call to the API.
- **timeout** (*float*) – timeout value in seconds to wait for response from API.
- **should_backoff** – (Optional) provides methods for backoff.
- **repeater** – provides method to repeat failed requests to API due to service errors.
- **cache** – (Optional) a ~apache_beam.io.requestresponse.Cache object to use the appropriate cache.
- **throttler** – provides methods to pre-throttle a request. Default is **DefaultThrottler**.

RequestResponseIO – DafaultThrottler

```
class DefaultThrottler(PreCallThrottler):
  def __init__(
      self,
      window_ms: int = 1,          # length of history to consider, in ms, to set throttling.
      bucket_ms: int = 1,          # granularity of time buckets that we store data in, in ms.
      overload_ratio: float = 2,   # the target ratio between requests sent and successful requests.
      delay_secs: int = 5          # minimum number of seconds to throttle a request.
):
    self.throttler = AdaptiveThrottler(
        window_ms=window_ms, bucket_ms=bucket_ms, overload_ratio=overload_ratio
    )
    self.delay_secs = delay_secs
```

Adaptive Throttler

Adaptive throttling makes the client "intelligent" by giving it a memory of recent events.

The client continuously tracks two key metrics over a short time window (e.g., the last 2 minutes):

- **requests**: The total number of requests the client has attempted to send.
- **accepts**: The number of requests that have completed successfully.
- **K**: a configurable constant (the overload_ratio), often set to 2.

It then enforces a simple rule: **requests <= K * accepts**

- **Reducing the multiplier K** will make adaptive throttling behave **more aggressively**
- **Increasing the multiplier K** will make adaptive throttling behave **less aggressively**

https://sre.google/sre-book/handling-overload/

RequestResponseIO – DafaultThrottler

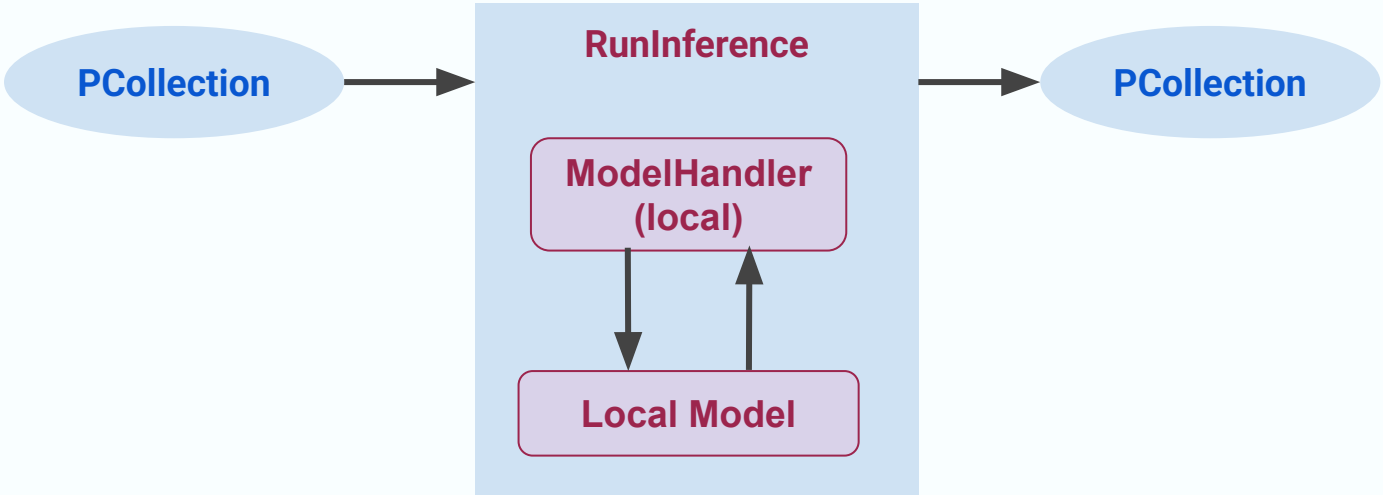
Default Throttling Behavior of RequestResponseIO:

- The DefaultThrottler is initialized with **window_ms=1**.
- **This means its "memory" of past events is only 1 millisecond long**. In practice, **this disables the "adaptive" part of the throttler.**
- The default behavior is therefore a simple, reactive backoff. When a request fails and is marked for retry, the DefaultThrottler will simply pause for a fixed duration (delay_secs, which defaults to 5 seconds).
- It does not proactively slow down based on an increasing failure rate.

RunInference

RunInference API is a PTransform optimized for efficient use of ML models in Beam pipelines. **ModelHandler** manages complexities like model loading, resource sharing, and performance-critical batching.

- Supported since v2.40 (6/2023)



RunInference – Parameters

class apache_beam.ml.inference.base.**RunInference**

Parameters:

- **model_handler** – An implementation of ModelHandler.
- **clock** – A clock implementing time_ns. *Used for unit testing.*
- **inference_args** – Extra arguments for models whose inference call requires extra parameters.
- **metrics_namespace** – Namespace of the transform to collect metrics.
- **model_metadata_pcoll** – PCollection that emits Singleton ModelMetadata containing model path and model name, that is used as a side input to the _RunInferenceDoFn.
- **watch_model_pattern** – A glob pattern used to watch a directory for automatic model refresh.
- **model_identifier** – A string used to identify the model being loaded. You can set this if you want to reuse the same model across multiple RunInference steps and don't want to reload it twice.

RunInference – ModelHandler

class apache_beam.ml.inference.base.**ModelHandler**

Key Methods:

- **load_model** – Loads and initializes a model for processing
- **run_inference –** Runs inferences on a batch of examples
  - **Parameters:**
    - **batch** – A sequence of examples or features.
    - **model** – The model used to make inferences.
    - **inference_args** – Extra arguments for models whose inference call requires extra parameters.
  - **Returns:**
    - An Iterable of Predictions.

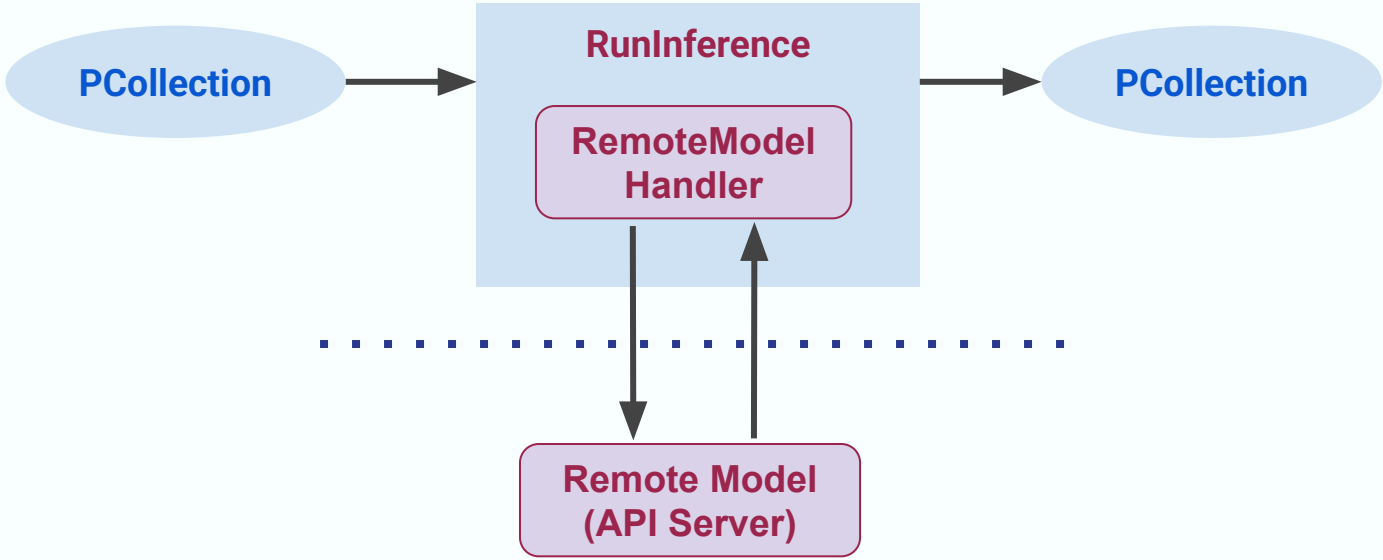
RunInference – Model Handler Subclasses

- **ModelHandler** (since v2.40, 6/2022) – for Local Inference
  - Model Handlers for “Local Inference”
    - PyTorch, TF, ONNX, vLLM, Sklearn, and many more
    - https://beam.apache.org/releases/pydoc/current/apache_beam.ml.inference.html
  - **RemoteModelHandler** (since v2.65 5/2025) – for “Remote Inference”
    - **GeminiModelHandler** (new for v2.66, 7/2025)
    - **VertexAIModelHandlerJSON** (since v2.49, 7/2023)

RunInference – RemoteModelHandler

RemoteModelHandler is a subclass of ModelHandler for Remote Inference.

- Supported since v2.65 5/2025



RunInference – RemoteModelHandler

class apache_beam.ml.inference.base.**RemoteModelHandler**

Parameters:

- **namespace** – the metrics and logging namespace
- **num_retries** – the maximum number of times to retry a request on retriable errors before failing
- **throttle_delay_secs** – the amount of time to throttle when the client-side elects to throttle
- **retry_filter** – a function accepting an exception as an argument and returning a boolean. On a true return, the run_inference call will be retried. Defaults to always retrying.
- **window_ms** – length of history to consider, in ms, to set throttling.
- **bucket_ms** – granularity of time buckets that we store data in, in ms.
- **overload_ratio** – the target ratio between requests sent and successful requests. This is “K” in the formula in https://landing.google.com/sre/book/chapters/handling-overload.html.

RunInference – RemoteModelHandler

```
_MILLISECOND_TO_SECOND = 1_000
class RemoteModelHandler(ABC, ModelHandler[ExampleT, PredictionT, ModelT]):
  def __init__(
      self,
      namespace: str = '',
      num_retries: int = 5,
      throttle_delay_secs: int = 5,
      retry_filter: Callable[[Exception], bool] = lambda x: True,
      *,
      window_ms: int = 1 * _MILLISECOND_TO_SECOND,
      bucket_ms: int = 1 * _MILLISECOND_TO_SECOND,
      overload_ratio: float = 2
    ):

    self.throttled_secs = Metrics.counter(namespace, "cumulativeThrottlingSeconds")
    self.throttler = AdaptiveThrottler(window_ms=window_ms, bucket_ms=bucket_ms, overload_ratio=overload_ratio)
    self.logger = logging.getLogger(namespace)
    self.num_retries = num_retries
    self.throttle_delay_secs = throttle_delay_secs
    self.retry_filter = retry_filter
```

RunInference – RemoteModelHandler

class apache_beam.ml.inference.base.**RemoteModelHandler**

Methods:

- **create_client()** – Creates the client that is used to make the remote inference request in request().

  All relevant arguments should be passed to __init__().

- abstract **request() –** Makes a request to a remote inference service and returns the response. Should raise an exception of some kind if there is an error to enable the retry and client-side throttling logic to work. Returns an iterable of the desired prediction type.

  - **Parameters:**
    - **batch** – A sequence of examples or features.
    - **model** – The model used to make inferences.
    - **inference_args** – Extra arguments for models whose inference call requires extra parameters.
  - **Returns:**
  - An Iterable of Predictions.

Summary of Remote Inference Approaches

Challenge	ParDo/DoFn	RequestResponseIO	RunInference + RemoteModelHandler
Retries & Backoff	No. Need custom implementation	Need a custom should_backoff function to control retry logic	Handled by RemoteModelHandler.
Throttling	No. Need custom implementation	Need to set proper parameters for DefaultThrottler. Default setting is only reactive.	Handled by AdaptiveThrottler. The default values are different from RequestResponseIO (which is only reactive)
Boilerplate Code	Requires a full custom DoFn implementation	Need to implement a custom Caller	Pre-built Model Handlers for Gemini and Gemma on Vertex

Apache Beam – Runners

1. Direct Runner
2. Prism Runner
3. Distributed Runners
    - Google Cloud Dataflow
    - Apache Flink
    - Apache Samza
    - Apache Spark
    - and many more

<u>This session won’t cover Runner specific topics</u>

3. Implementations

How can I implement “remote inference” with Apache Beam?

- Focused on Python
- Gemini API
- Gemma with Vertex AI API

Implementations Covered

PTransform Implementations:

- RequestResponseIO
  - Custom Caller Example (for Gemini)
- RemoteModelHandler (new with v2.65 in 5/2025)
  - GeminiModelHandler (new with upcoming v2.66)
  - VertexAIModelHandlerJSON (new with v2.49 in 7/2023)

LLM APIs:

- Gemini (on Google GenAI API)
- Gemma (on Vertex AI Prediction API)

Python Packages Installation

Apache Beam v2.66.0 (released on July 1, 2025)

- pip install apache-beam[gcp]==**2.66.0** -U

Google GenAI SDK (for Gemini)

- pip install google-genai

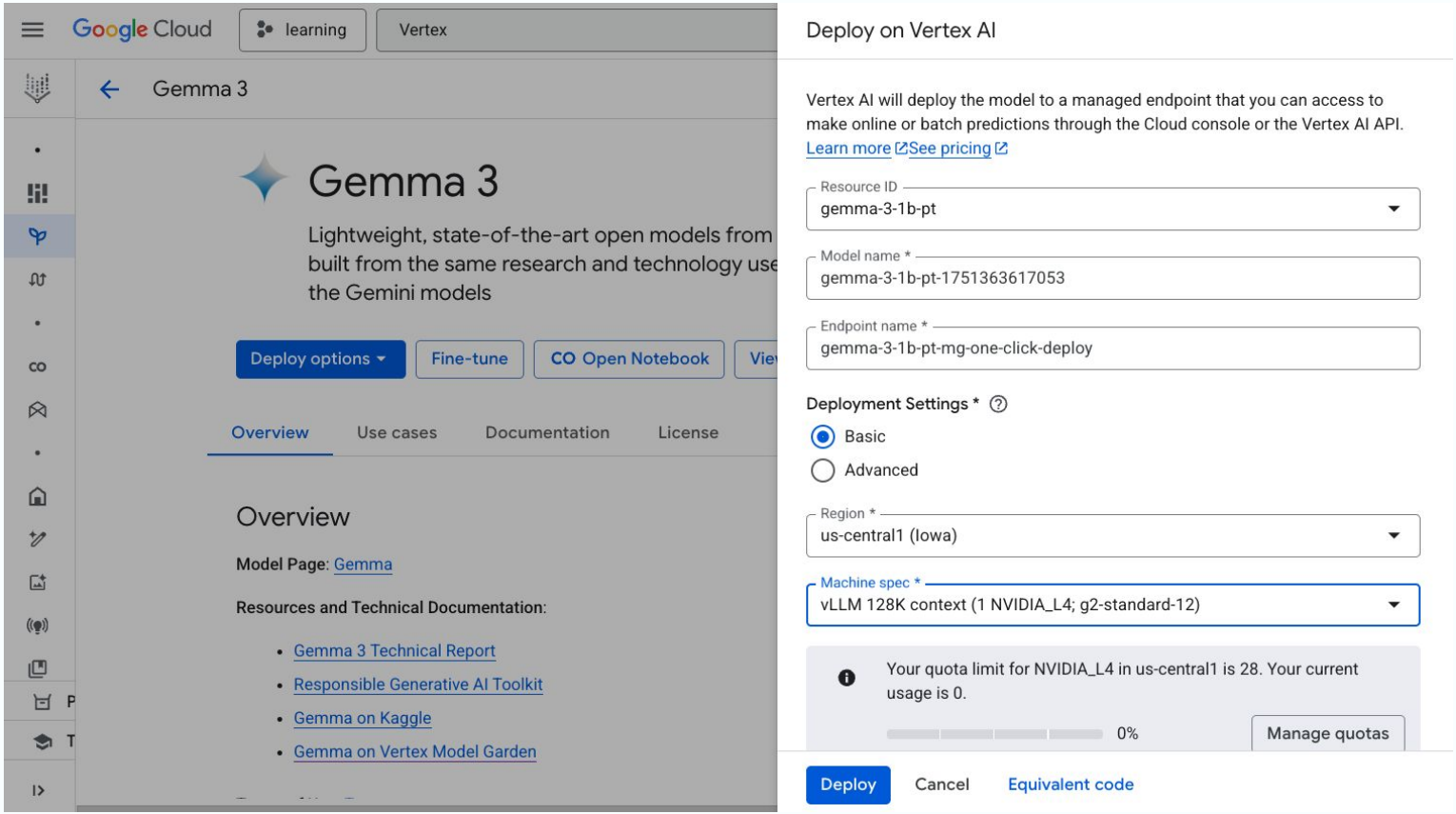
Python SDK for Vertex AI (for Gemma)

- pip install google-cloud-aiplatform -U

Gemini API (Python)

```
from google import genai
from google.genai import types

client = genai.Client(
        vertexai=True,
        project='your-project-id',
        location='us-central1',
        http_options=types.HttpOptions(api_version='v1')
)

response = client.models.generate_content(
        model='gemini-2.5-flash',
        contents='Why is the sky blue?'
)
print(response.text)
```

Gemma Deployment on Vertex AI



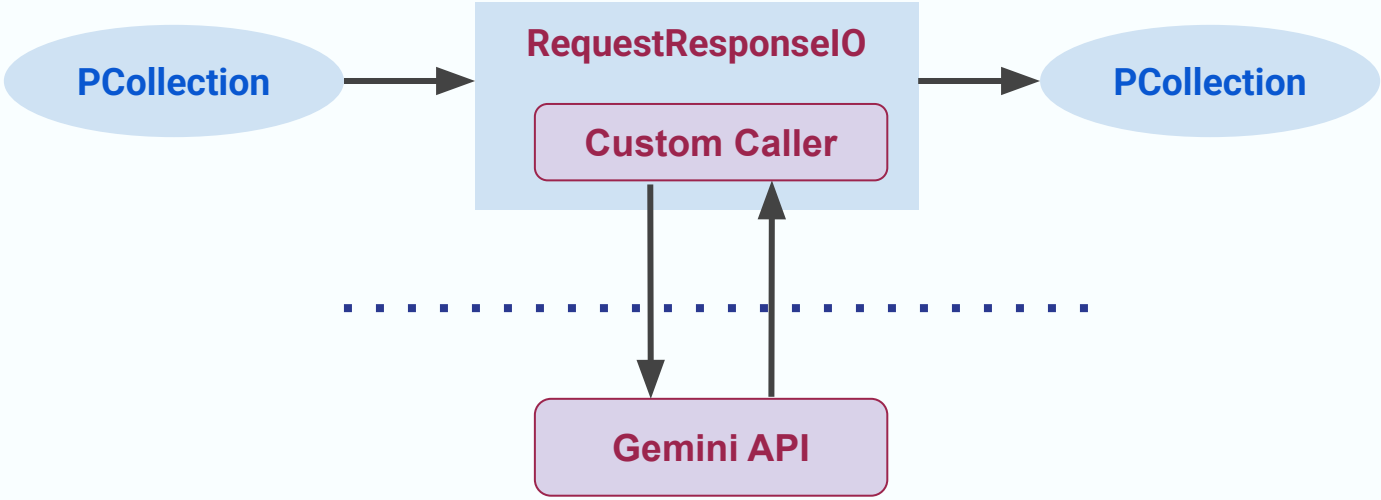
Vertex AI API for Gemma (Python)

```
from google.cloud import aiplatform
endpoint =
aiplatform.Endpoint(f"projects/{PROJECT_NUMBER}/locations/{REGION}/endpoints/{ENDPOINT_ID}")

instances = [
      {
        "@requestFormat": "chatCompletions",
        "messages": [
           {
              "role": "user",
              "content": "What is machine learning?"
           }
        ],
        "max_tokens": 100
      }
   ]

endpoint.predict(instances=instances, use_dedicated_endpoint=True)
```

RequestResponseIO for Gemini

responses = requests | **RequestResponseIO**(GeminiCustomCaller())

*GeminiCustomCaller is a **custom Caller***



Gemini Custom Caller Example

```python
from google import genai
from google.genai.types import HttpOptions

class GeminiCustomCaller(Caller):
    def __enter__(self):
        self.client = genai.Client(http_options=HttpOptions(api_version="v1"))
        return self

    def __call__(self, prompt):
        try:
            response = self.client.models.generate_content(
                model="gemini-2.5-flash",
                contents=prompt
            )
        except APIError as e:
            raise UserCodeExecutionException() from e
        return response
```

https://beam.apache.org/documentation/io/built-in/webapis/

RequestResponseIO + Custom Gemini Caller

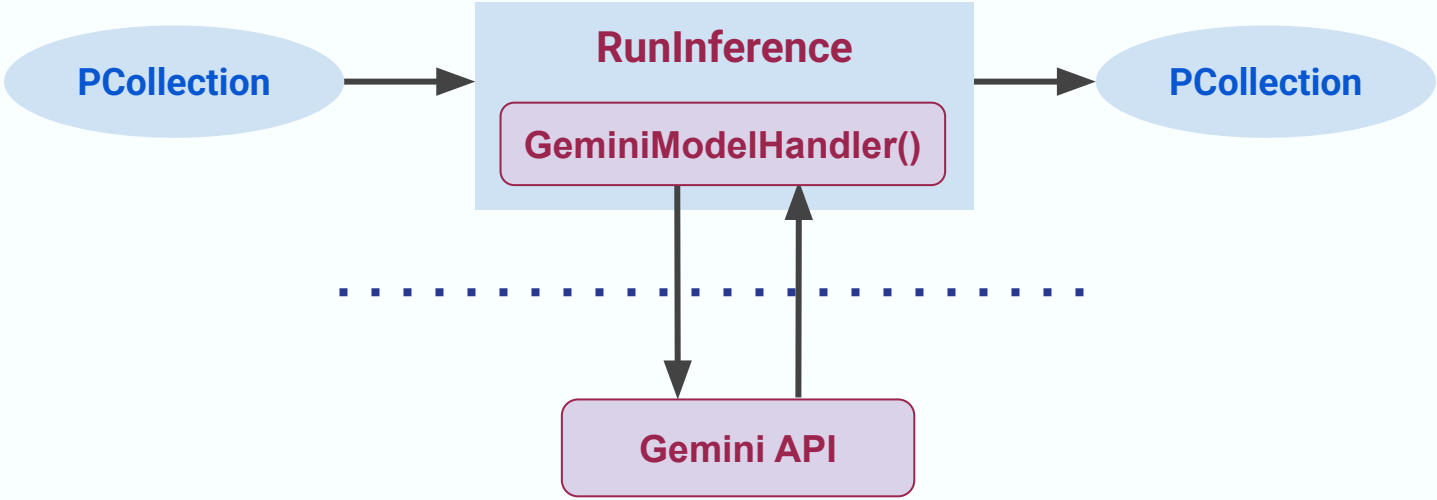
```
import apache_beam as beam
from apache_beam.io.requestresponse import RequestResponseIO
from apache_beam.options.pipeline_options import PipelineOptions

prompts = [
      "What is 5+2?",
      "Who is the protagonist of Lord of the Rings?",
      "What is the air-speed velocity of a laden swallow?"
]
with beam.Pipeline(options=PipelineOptions()) as pipeline:
    _ = (
        pipeline
        | "Create data" >> beam.Create(prompts)
        | "Gemini AI" >> RequestResponseIO(GeminiCustomCaller())
        | "Print results" >> beam.Map(lambda response: print(response.text))
    )
    result = pipeline.run()
```

Customizing Parameters with DefaultThrottler

```
# Unlock the true adaptive capabilities by providing a meaningful window.
# This throttler will analyze the last 2 minutes of activity to make decisions.
custom_throttler = DefaultThrottler(
  window_ms=120000,       # Analyze the last 2 minutes of history
  bucket_ms=5000,        # In 5-second granular buckets
  overload_ratio=1.5     # Be conservative: throttle if failures increase
)

with beam.Pipeline(options=PipelineOptions()) as pipeline:
    _ = (
        pipeline
        | "Create data" >> beam.Create(prompts)
        | "Gemini AI" >> RequestResponseIO(caller=GeminiCustomCaller(),
                                           throttler=custom_throttler
                                )
        | "Print results" >> beam.Map(lambda response: print(response.text))
    )
```

RunInference + GeminiModelHandler

responses = requests | **RunInference**(**GeminiModelHandler**())



Gemini with RunInference

GeminiModelHandler()

Parameters:

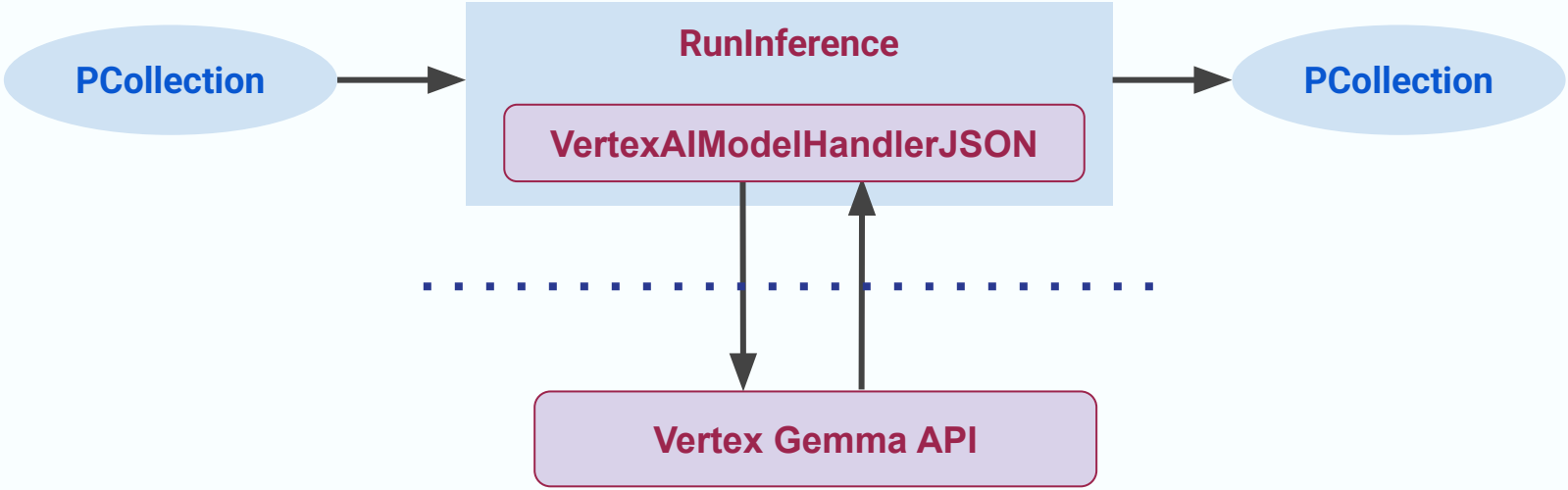
- **model_name** – the Gemini model to send the request to
- **request_fn** – the function to use to send the request. Should take the model name and the parameters from request() and return the responses from Gemini. The class will handle bundling the inputs and responses together.
- **api_key** – the Gemini Developer API key to use for the requests. Setting this parameter sends requests for this job to the Gemini Developer API. If this parameter is provided, do not set the project or location parameters.
- **project** – the GCP project to use for Vertex AI requests. Setting this parameter routes requests to Vertex AI. If this parameter is provided, location must also be provided and api_key should not be set.
- **location** – the GCP project to use for Vertex AI requests. Setting this parameter routes requests to Vertex AI. If this parameter is provided, project must also be provided and api_key should not be set.
- **min_batch_size –** optional. the minimum batch size to use when batching inputs.
- **max_batch_size –** optional. the maximum batch size to use when batching inputs.
- **max_batch_duration_secs** – optional. the maximum amount of time to buffer a batch before emitting; used in streaming contexts.

Gemini with RunInference – Python

```
import apache_beam as beam
from apache_beam.ml.inference.gemini_inference import GeminiModelHandler # New for Beam 2.66
from apache_beam.ml.inference.gemini_inference import generate_from_string  # New for Beam 2.66
from apache_beam.options.pipeline_options import PipelineOptions

model_handler = GeminiModelHandler(
      model_name='gemini-2.5-flash',
      request_fn=generate_from_string,
      project=project, location=location
)

with beam.Pipeline(options=PipelineOptions()) as pipeline:
    _ = (
        pipeline
        | "Create data" >> beam.Create(prompts)
        | "RunInference" >> RunInference(model_handler)
        | "Print results" >> beam.Map(lambda response: print(response.text))
      )
      result = pipeline.run()
```

Gemma with RunInference

RunInference(KeyedModelHandler(VertexAIModelHandlerJSON()))



VertexAIModelHandlerJSON()

Parameters:

- **endpoint_id** – the numerical ID of the Vertex AI endpoint to query
- **project** – the GCP project name where the endpoint is deployed
- **location** – the GCP location where the endpoint is deployed
- **experiment** – optional. experiment label to apply to the queries.
- **network** – optional. the full name of the Compute Engine network the endpoint is deployed on; used for private endpoints. The network or subnetwork Dataflow pipeline option must be set and match this network for pipeline execution. Ex: "projects/12345/global/networks/myVPC"
- **private** – optional. if the deployed Vertex AI endpoint is private, set to true. Requires a network to be provided as well.
- **min_batch_size** – optional. the minimum batch size to use when batching inputs.
- **max_batch_size** – optional. the maximum batch size to use when batching inputs.
- **max_batch_duration_secs** – optional. the maximum amount of time to buffer a batch before emitting; used in streaming contexts.

Gemma with RunInference – Python

```
from apache_beam.ml.inference.base import KeyedModelHandler
from apache_beam.ml.inference.base import RunInference
from apache_beam.ml.inference.vertex_ai_inference import
VertexAIModelHandlerJSON

model_handler = VertexAIModelHandlerJSON(
      endpoint_id=ENDPOINT_ID, project=PROJECT_ID, location=LOCATION
)
parameters = {
      "temperature": 0.2, "maxOutputTokens": 256, "topK": 40, "topP": 0.95, "use_dedicated_endpoint": True
}
with beam.Pipeline(options=PipelineOptions()) as pipeline:
    _ = (pipeline
        | "Create data" >> beam.Create(prompts)
        | "RunInference" >> RunInference(
                              KeyedModelHandler(model_handler),
                              inference_args=parameters
                              )
        | "Print results" >> beam.Map(lambda response: print(response.text))
    )
    result = pipeline.run()
```

Customizing Parameters with AdaptiveThrottler

```
gemma_handler = VertexAIModelHandlerJSON(
    endpoint_id=ENDPOINT_ID,
    project=PROJECT,
    region=REGION,
    window_ms=300000,          # length of history to consider, in ms, to set throttling.
    bucket_ms=10000,           # granularity of time buckets that we store data in, in ms.
    overload_ratio=2.0,      # K: the target ratio between requests sent and successful requests.
    throttle_delay_secs=30,
    num_retries=5,
    retry_filter=is_retryable
)
```

4. Summary

What we learned in this session?

Summary

- Remote LLM Inferencing with Apache Beam
- **RunInference + RemoteModelHandler** is recommended for Apache Beam v2.65 or newer version
  - **GeminiModelHandler** for **Gemini**
  - **VertexAIModelHandlerJSON** for **Gemma** with Vertex AI
- **RequestResponseIO + Caller** is for older versions of Apache Beam or legacy code base (e.g. dependencies with existing custom Caller implementation)
- Example code:
  - https://github.com/blueviggen/beam-remote-llm-examples

QUESTIONS?