Using LLMs with Beam and RunInference

Reza Rokni
Jack R. McCluskey

BEAM SUMMIT

September 4-5, 2024

Sunnyvale, CA. USA

Jack has

I ~~also have~~ a dog.

Kenna would be upset if she didn't get a mention here.

BEAM SUMMIT

Why Use an LLM in a Beam Pipeline?

- LLMs are versatile models that can handle a variety of tasks
- Particularly excel with unstructured text input
  - Sentiment analysis
  - Summerization
- Prompt engineering and model tuning can adapt these models for data processing workloads relatively well

BEAM SUMMIT

Working through an example

GEMMA

Gemma for Streaming ML with Dataflow

AUG 16, 2024

Reza Rokni
Google Senior Staff
Dataflow

Ravin Kumar
Google Data Scientist
Language Applications

Share

BEAM SUMMIT

A users chat history

Chat ID : 4116

user_id 221:

Hay I am really annoyed that your menu includes a pizza with pineapple on it!

user_id 331:

Sorry to hear that , but pineapple is nice on pizza

user_id 221:

What a terrible thing to say! Its never ok, so unhappy right now!

Prompt the model

prompt_template = """

<prompt>

Provide the results of doing these two tasks on the chat history provided below for the user {}

task 1 : assess if the tone is happy = 1 , neutral = 0 or angry = -1

task 2 : summarize the text with a maximum of 512 characters

Output the results as a **json** with fields [sentiment, summary]

@@@{}@@@

<answer>

"""

JSON

Analyse the reponse

{

"sentiment":

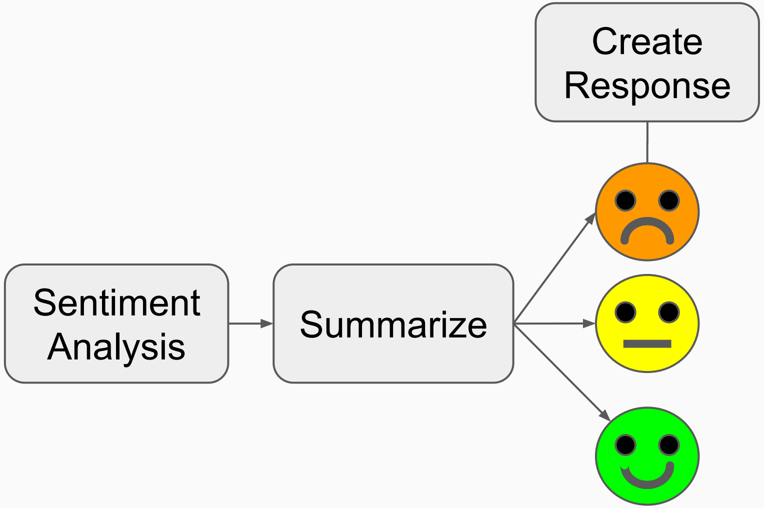
-1,

"summary":

"User 221 is very unhappy about the presence of pineapple on pizza."

}

Create a respone



Assist our support staff

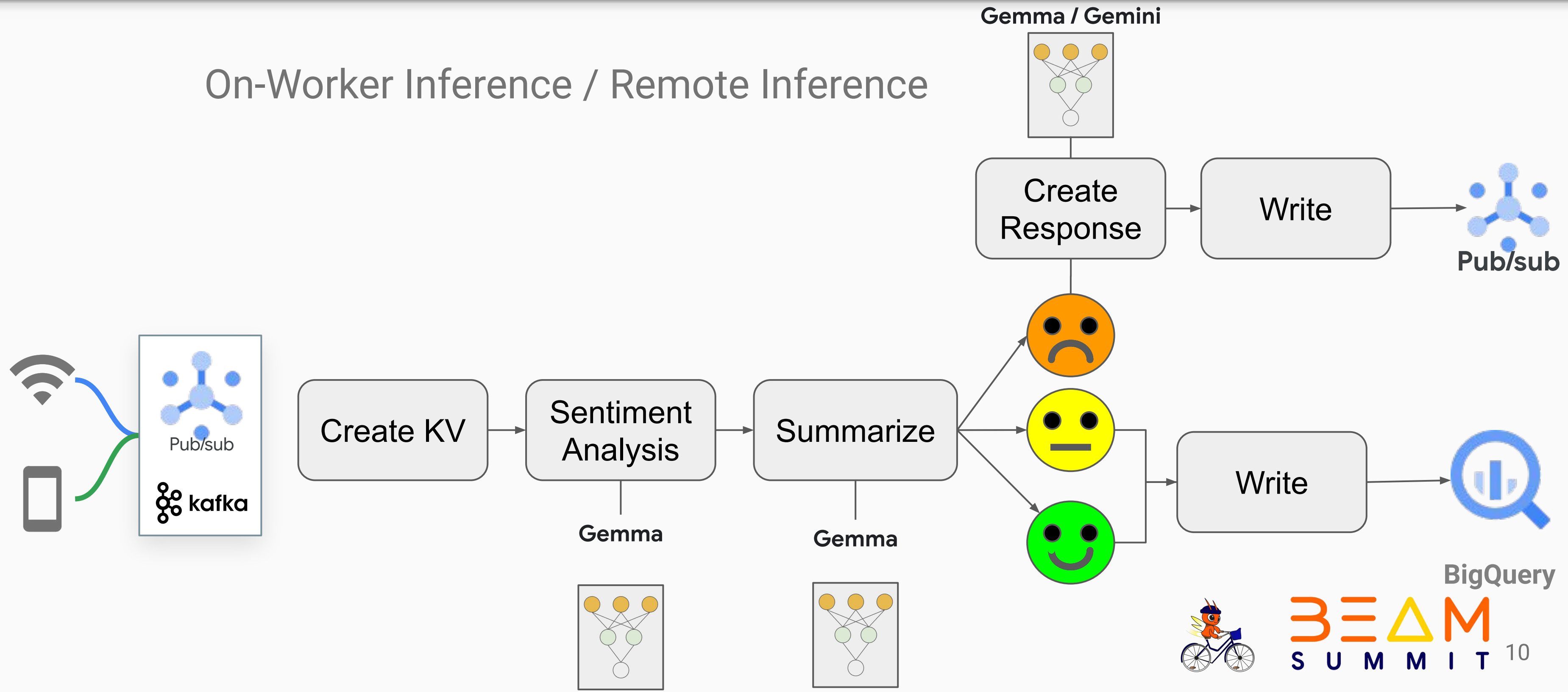
Prompt:

"Generate an apology response for the user in this chat text: {}"

Response:

"I understand that you're upset about the pineapple pizza. It's a very personal preference, and I apologize that it might have caused you some frustration. We strive to offer a diverse menu to cater to a wide range of tastes, and we're always open to feedback. Would you like to share your thoughts on the pizza with pineapple?"

Apache Beam for Async LLM flows



Two Ways to Approach The Problem

1. On-Worker Inference
2. Remote Inference

On-Worker Inference

RunInference

```
with pipeline as p:
  predictions = (
      p
      | beam.ReadFromSource('a_source')
      | RunInference(MODEL_HANDLER))
```

BEAM SUMMIT

Worker Considerations

- LLMs are *big* models
  - Workers will need sufficient memory to load and serve the models
  - GPUs generally provide better performance for inference
- Beam Python workers can run multiple copies of the worker harness on a single machine
  - How do we avoid OOM errors with these big models without enforcing one worker harness instance per machine?

Model Loading

- Implement the share_model_across_processes() method in the model handler
  - Or pass the large_model parameter to one of the existing RunInference model handlers
- Enforces loading the model once per worker instead of per process

Model Staging

- Another consideration: where is the model staged before runtime?
- Loading externally
    - No extra work beyond pulling the model from a source
    - Can incur a lot of network traffic and significant loading times
- Packaging the model into a custom worker container
    - All required files are present in the worker at startup, avoiding network use
        - Essential if workers cannot have public IPs for security purposes
    - Container sizes are *big*

BEAM SUMMIT

Developement considerations

1. Build out the LLM calls first with its own Unit Tests
2. Use the DirectRunner along with a GPU enabled host
3. When using local mode use containers
    a. Create a docker container with the model
    b. Create a nother docker image that used the first as base and hosts your pipeline code

Gemma Model Handler (KerasNLP)

Model Loading

```
def __init__(
        self,
        model_name: str = "",
    ):
        self._model_name = model_name
        self._env_vars = {}

def share_model_across_processes(self)  -> bool:
        return True

def load_model(self) -> GemmaCausalLM:
        return keras_nlp.models.GemmaCausalLM.from_preset(self._model_name)
```

Gemma Model Handler (KerasNLP)

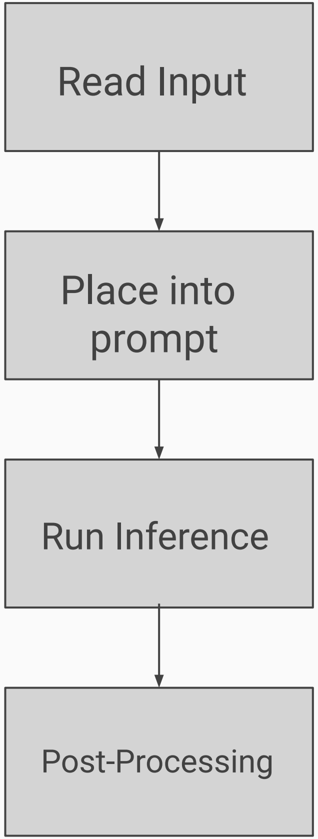
Run Inference

```
def run_inference(
        self,
        batch: Sequence[str],
        model: GemmaCausalLM,
        inference_args: Optional[Dict[str, Any]] = None
    ) -> Iterable[PredictionResult]:
        predictions = []
        for one_text in batch:
            result = model.generate(one_text, max_length=1024)
            predictions.append(result)
        return utils._convert_to_result(batch, predictions, self._model_name)
```

BEAM SUMMIT

Gemma Model Handler (KerasNLP)

Processing Flow



Gemma Model Handler (KerasNLP)

Example Prompt Template

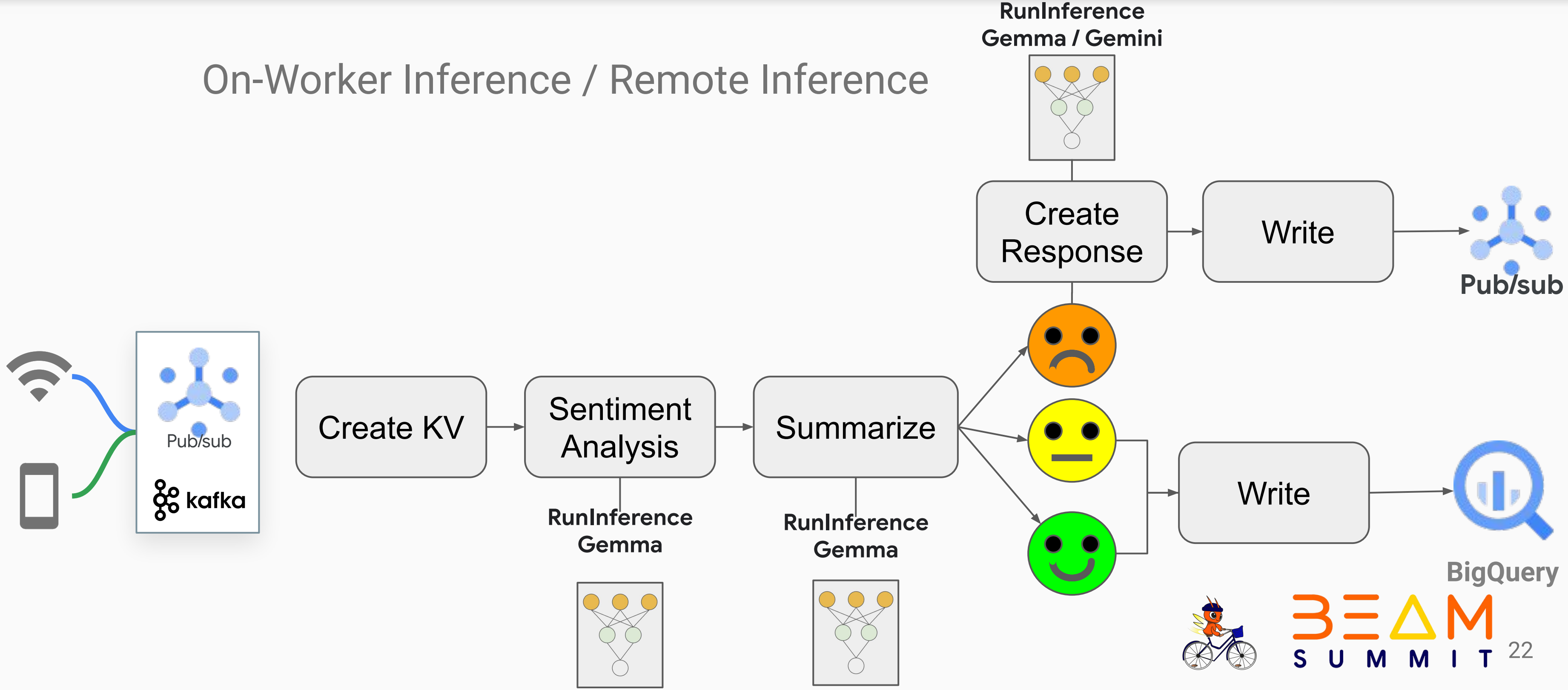
```
prompt_template = """
<prompt>
Provide the results of doing these two tasks on the chat history provided
below for the user {}
task 1 : assess if the tone is happy = 1 , neutral = 0 or unhappy = -1
task 2 : summarize the text with a maximum of 512 characters
Return the answer as a JSON string with fields [sentiment, summary] do NOT
explain your answer

@@@{}@@@
<answer>
"""
```

?

Apache Beam for Async LLM flows

On-Worker Inference / Remote Inference



Remote Inference

Why Use Remote Inference?

- Want to use a larger model for the business needs
- Restrictions on available worker hardware
- Concerns about load times/worker container sizes
- Want to use an already existing service

Adapting RunInference for Remote Calls

- Model loading becomes a lightweight process
  - Can do some sort of sanity checking that the remote service being called exists and any arguments are valid
- Inputs have to be formatted to be sent over the wire
- Errors from the external service have to be handled

Being a Good Client

- Understanding and respecting HTTP error codes when returned is essential to keeping your pipeline running cleanly
  - Common error to consider: HTTP 429: Too Many Requests
- Mechanically, the DoFn needs to know when to retry the request and when an error is unrecoverable
- Respecting 429 errors and backing off will allow the pipeline to continue once the external service can handle traffic again

Vertex AI Model Handler

Model Loading

VertexAIModelHandler

```
def load_model(self) -> aiplatform.Endpoint:
    """Loads the Endpoint object used to build and send prediction request to
    Vertex AI.
    """
    # Check to make sure the endpoint is still active since pipeline
    # construction time
    ep = self._retrieve_endpoint(
        self.endpoint_name, self.location, self.is_private)
    return ep
```

Vertex AI Model Handler

Model Loading

```
def _retrieve_endpoint(
    self, endpoint_id: str, location: str,
    is_private: bool) -> aiplatform.Endpoint:
  if is_private:
    endpoint: aiplatform.Endpoint = aiplatform.PrivateEndpoint(
        endpoint_name=endpoint_id, location=location)
    LOGGER.debug("Treating endpoint %s as private", endpoint_id)
  else:
    endpoint = aiplatform.Endpoint(
        endpoint_name=endpoint_id, location=location)
    LOGGER.debug("Treating endpoint %s as public", endpoint_id)
  try:
    mod_list = endpoint.list_models()
  except Exception as e:
    raise ValueError(
        "Failed to contact endpoint %s, got exception: %s", endpoint_id, e)
  if len(mod_list) == 0:
    raise ValueError("Endpoint %s has no models deployed to it.",
                     endpoint_id)
  return endpoint
```

Vertex AI Model Handler

Inference Calls

```
def run_inference(
      self,
      batch: Sequence[Any],
      model: aiplatform.Endpoint,
      inference_args: Optional[Dict[str, Any]] = None
  ) -> Iterable[PredictionResult]:
    # Endpoint.predict returns a Prediction type with the prediction values
    # along with model metadata
    prediction = self.get_request(
        batch, model, throttle_delay_secs=5, inference_args=inference_args)

    return utils._convert_to_result(
        batch, prediction.predictions, prediction.deployed_model_id)
```

Vertex AI Model Handler

Request-Response Loop

```
@retry.with_exponential_backoff(
      num_retries=5, retry_filter=_retry_on_appropriate_gcp_error)
def get_request(
      self,
      batch: Sequence[Any],
      model: aiplatform.Endpoint,
      throttle_delay_secs: int,
      inference_args: Optional[Dict[str, Any]]):
    while self.throttler.throttle_request(time.time() * MSEC_TO_SEC):
      time.sleep(throttle_delay_secs)
      self.throttled_secs.inc(throttle_delay_secs)
    try:
      req_time = time.time()
      prediction = model.predict(
          instances=list(batch), parameters=inference_args)
      self.throttler.successful_request(req_time * MSEC_TO_SEC)
      return prediction
    except TooManyRequests as e:
      LOGGER.warning("request was limited by the service with code %i", e.code)
      raise
    except Exception as e:
      LOGGER.error("unexpected exception raised as part of request, got %s", e)
      raise
```

A Confession

- These techniques and design considerations apply for any large model deployed in a Beam pipeline, not just LLMs!
- Managing your model effectively (where the model is being loaded from, the number of copies, handling remote inference calls) is key to building the most effective pipeline

Thank you!

Questions?

LinkedIn -

linkedin.com/in/jrmccluskey

linkedin.com/in/rezarokni