Superpowering Agents with Apache Beam

Konstantin Buschmeier
Jasper Van den Bossche
& Karel Haerens
Machine Learning Engineers @ ML6

BEAM SUMMIT
NYC 2025

Who are we?

ML6

AI and Machine Learning services company.

Team of AI engineering and strategy experts with 10+ years of experience.

We help clients build scalable, future-proof AI solutions using technologies such as Apache Beam.

What will be covered?

Superpowering Agents with Apache Beam

- Example Use Case
- LLMs, Tool Calling & Agents
- Agents & Beam

Use case

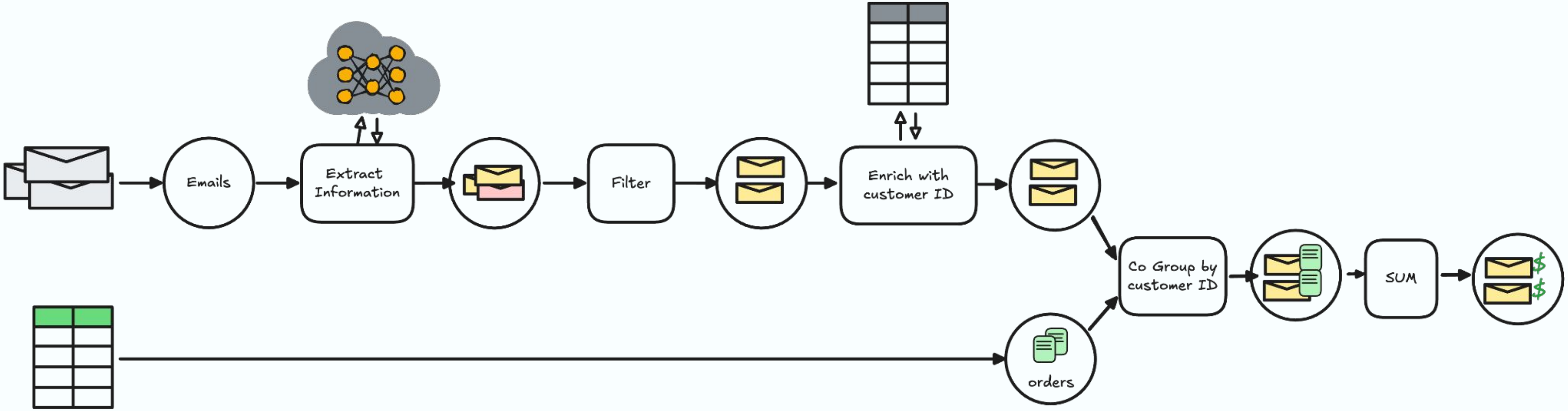
Use Case

> My **candy company** went viral & orders are through the roof!
> We **use Apache Beam** for many workflows & it's great!
> I received many urgent **emails** that need to be addressed:

> What is the **total spent** for **customers** that **ordered by mail?**

> I **cannot wait** for my engineering team to work on it.
> All attend the Summit.

Use Case: Possible Solution



Use Case

Process unstructured data

Enrich with structured data

Aggregate analytics

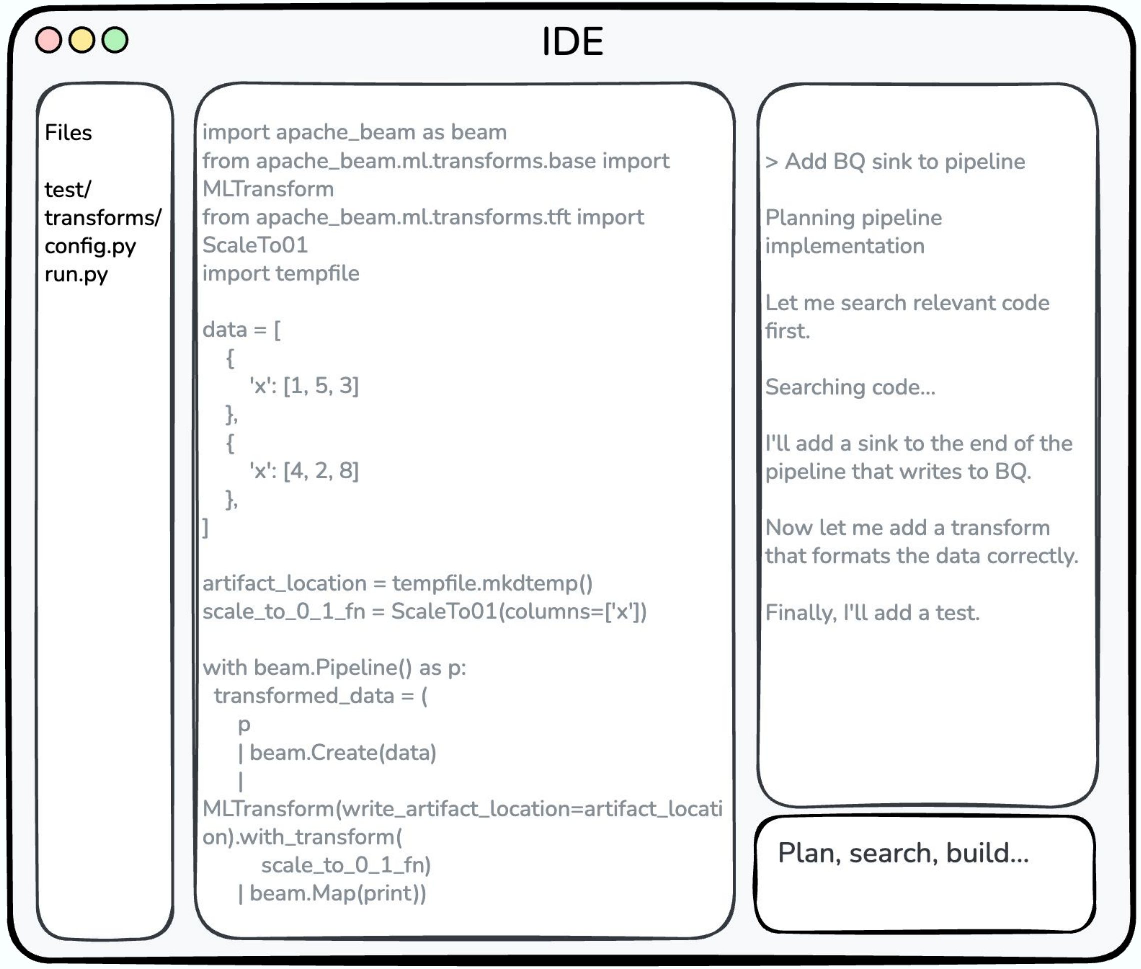
→ Gain insights & take action

...without the data team.

LLMs, Tool Calling & Agents

Just use a chatbot!?

chatbot
Powerful LLM
Ask Anything...
Upload



```
import apache_beam as beam
from apache_beam.ml.transforms.base import MLTransform
from apache_beam.ml.transforms.tft import ScaleTo01
import tempfile

data = [
    {
        'x': [1, 5, 3]
    },
    {
        'x': [4, 2, 8]
    },
]

artifact_location = tempfile.mkdtemp()
scale_to_0_1_fn = ScaleTo01(columns=['x'])

with beam.Pipeline() as p:
  transformed_data = (
      p
      | beam.Create(data)
      | MLTransform(write_artifact_location=artifact_location).with_transform(
          scale_to_0_1_fn)
      | beam.Map(print))
```

> Add BQ sink to pipeline

Planning pipeline implementation

Let me search relevant code first.

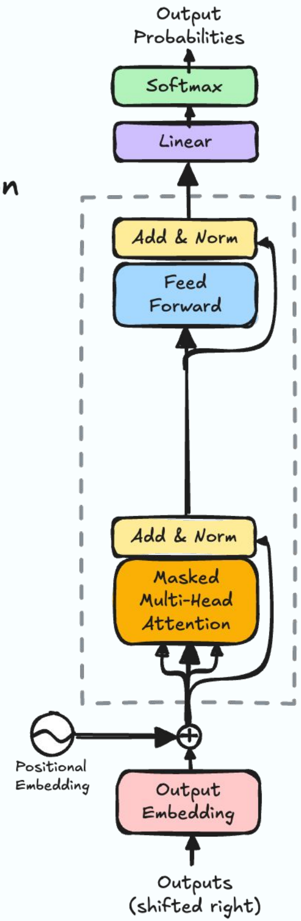
Searching code...

I'll add a sink to the end of the pipeline that writes to BQ.

Now let me add a transform that formats the data correctly.

Finally, I'll add a test.

LLMs



Text generation

Image Generation

Structured Output

Audio Generation

Long Contexts

Video Generation

Tool Calling

Code Execution

Reasoning

Text Understanding

Multimodality

Image Understanding

Audio Understanding

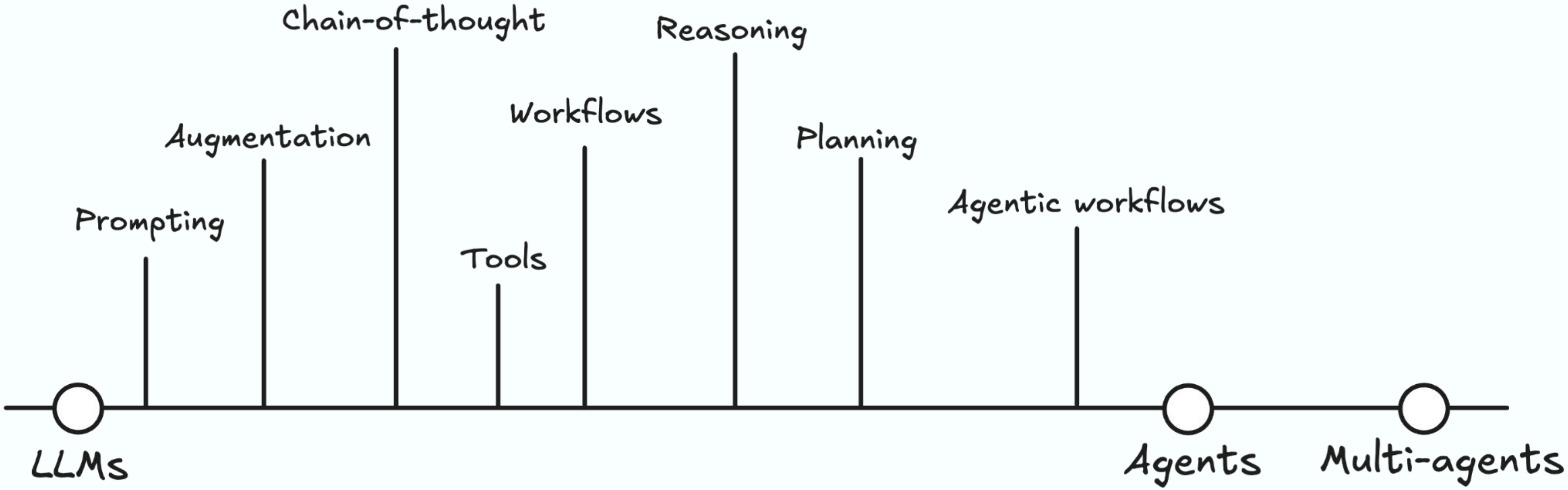
Video Understanding

Document Understanding

Web Search

Coding Models

From Prompts to Agents



A Weather Agent

> You are a **helpful agent.**
> Your objective is to **complete** the **user's tasks.**

> You have access to the following **tools**:
> {**get_weather** <city_name>}
> {**chance_of_rain** <city_name>}

A Weather Agent

User: What’s the weather in New York?

LLM: (get_weather New York)

🛠️ get_weather(location=“New York”) → 102F

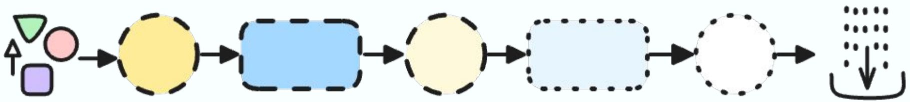
LLM: The weather in New York is 102F today.

A Basic Beam Agent

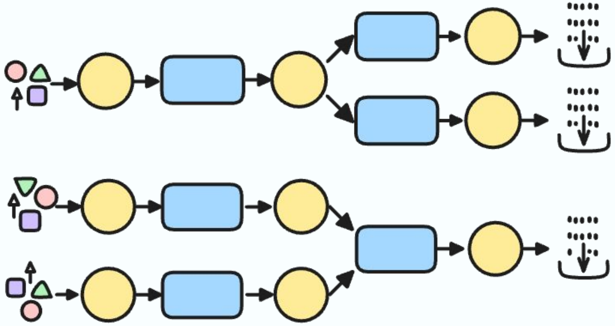
> You are a **helpful agent.**
> Your objective is to **complete tasks** using **Beam.**

> You have access to the following **tools:**
> {list_available_datasets}
> {create_beam_pipeline}
> {run_beam_pipeline}
> {read_results}

Agents & Beam



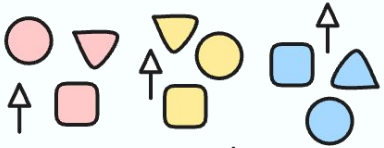
Write Pipeline



Pipeline Library

Run pipeline locally

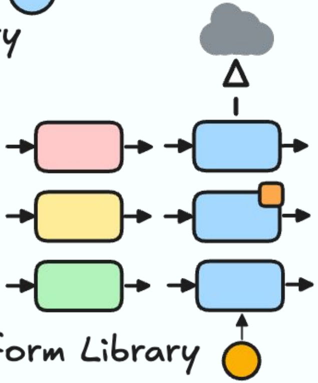
Submit & monitor
Dataflow pipeline



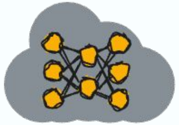
Source Library

Dataset Library

Documentation
Lookup



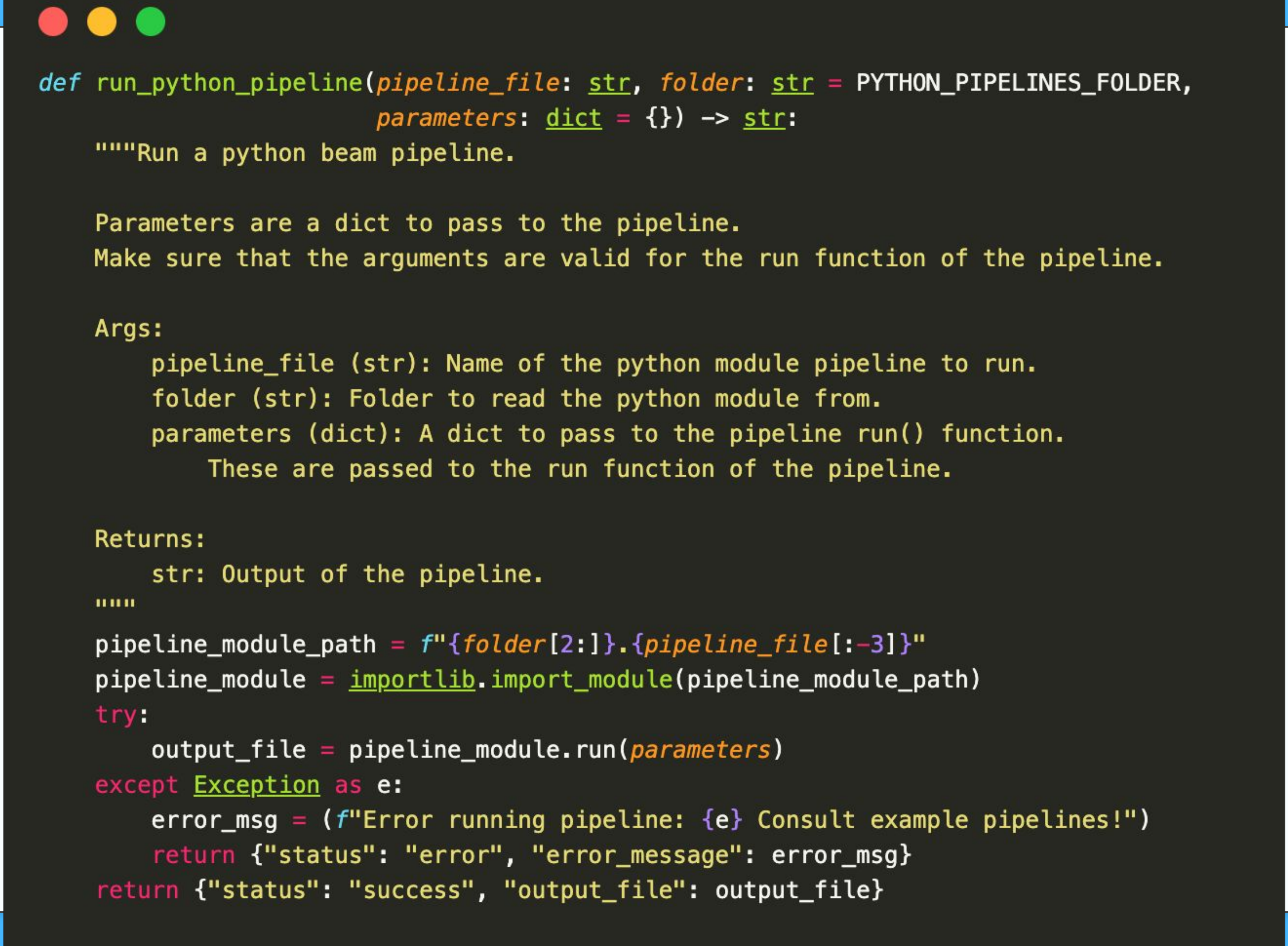
Transform Library



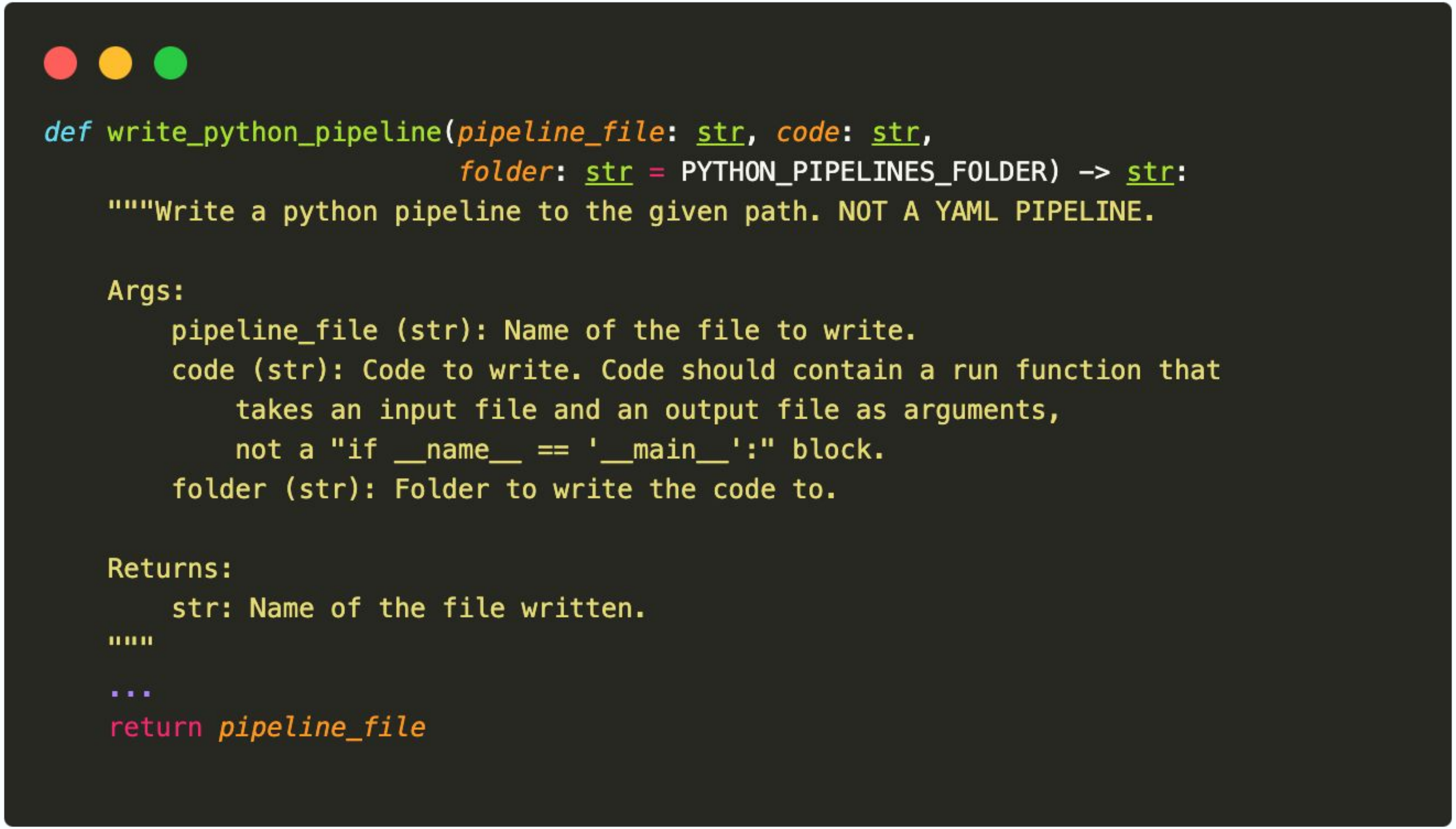
List APIs

Web Search

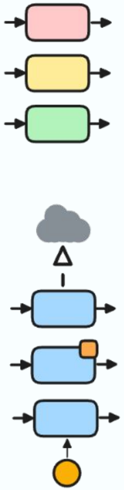
Bug Fixer

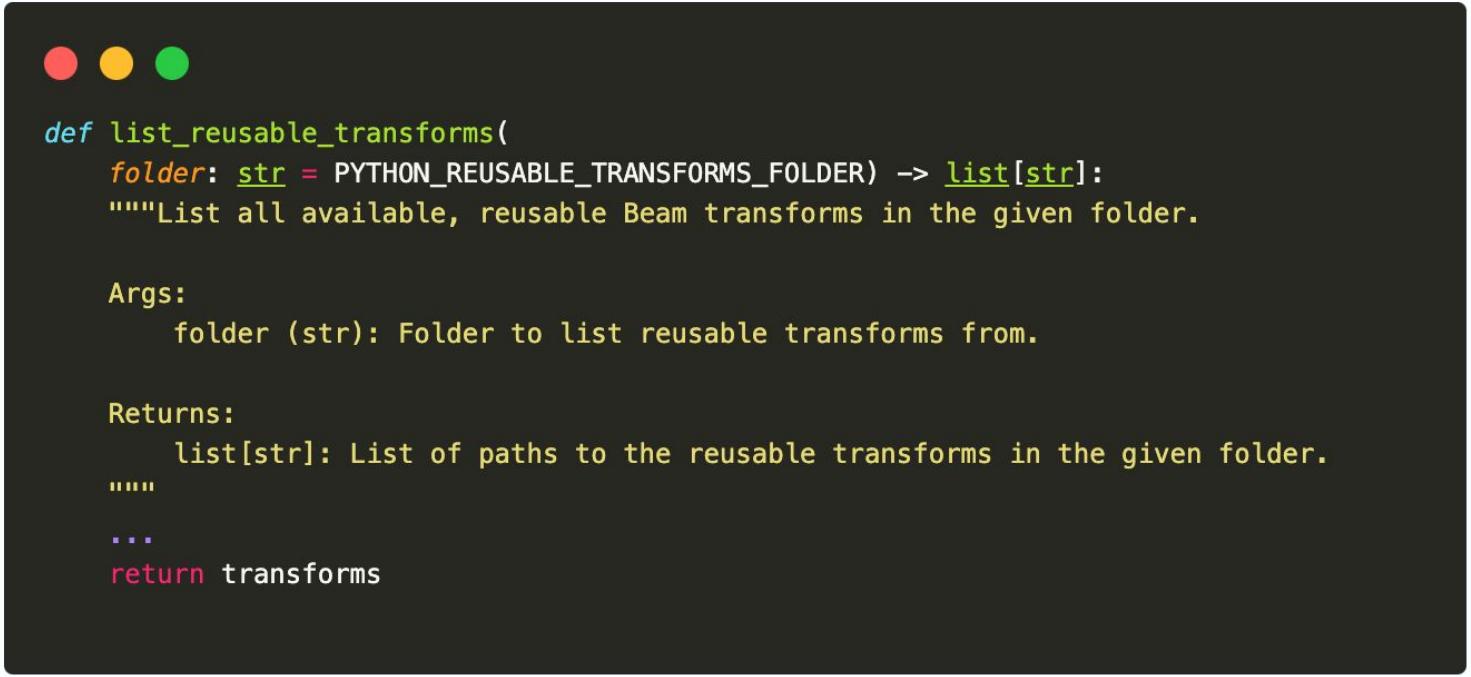


```python
def run_python_pipeline(pipeline_file: str, folder: str = PYTHON_PIPELINES_FOLDER,
                        parameters: dict = {}) -> str:
    """Run a python beam pipeline.

    Parameters are a dict to pass to the pipeline.
    Make sure that the arguments are valid for the run function of the pipeline.

    Args:
        pipeline_file (str): Name of the python module pipeline to run.
        folder (str): Folder to read the python module from.
        parameters (dict): A dict to pass to the pipeline run() function.
            These are passed to the run function of the pipeline.

    Returns:
        str: Output of the pipeline.
    """
    pipeline_module_path = f"{folder[2:]}.{pipeline_file[:-3]}"
    pipeline_module = importlib.import_module(pipeline_module_path)
    try:
        output_file = pipeline_module.run(parameters)
    except Exception as e:
        error_msg = (f"Error running pipeline: {e} Consult example pipelines!")
        return {"status": "error", "error_message": error_msg}
    return {"status": "success", "output_file": output_file}
```

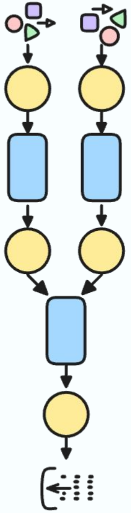


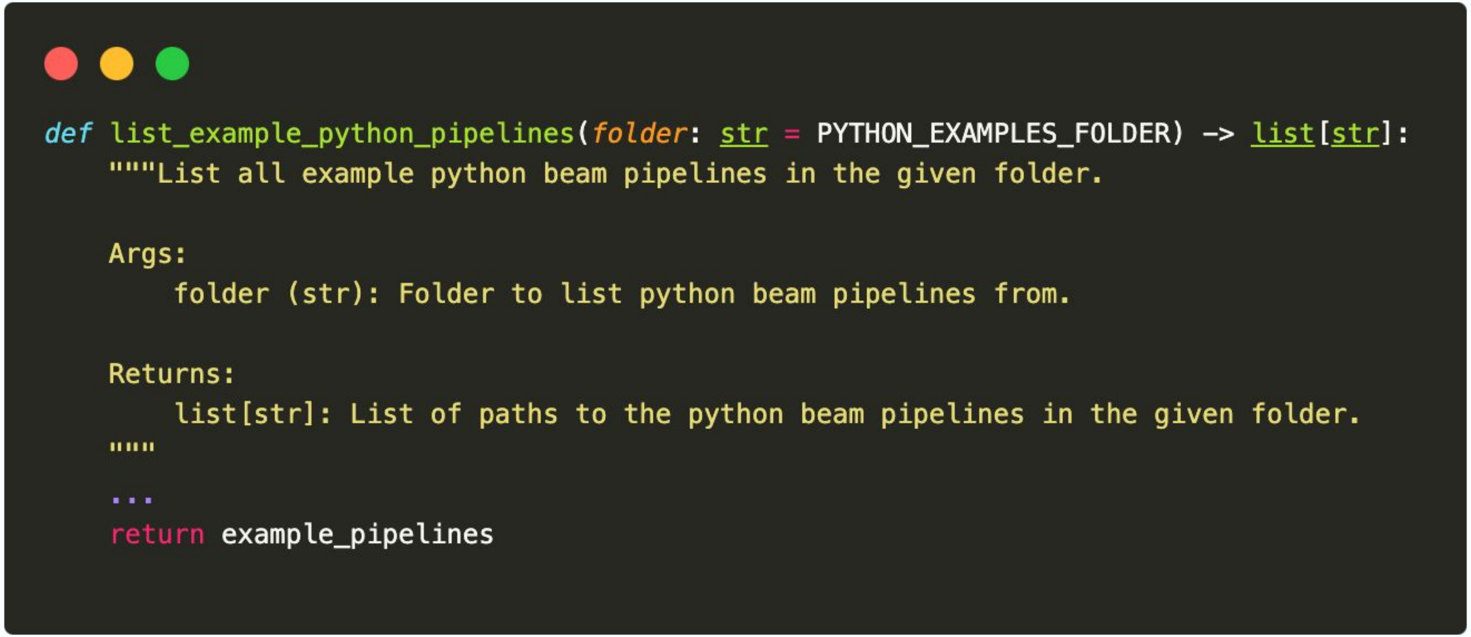
```python
def write_python_pipeline(pipeline_file: str, code: str,
                          folder: str = PYTHON_PIPELINES_FOLDER) -> str:
    """Write a python pipeline to the given path. NOT A YAML PIPELINE.

    Args:
        pipeline_file (str): Name of the file to write.
        code (str): Code to write. Code should contain a run function that
            takes an input file and an output file as arguments,
            not a "if __name__ == '__main__':" block.
        folder (str): Folder to write the code to.

    Returns:
        str: Name of the file written.
    """
    ...
    return pipeline_file
```



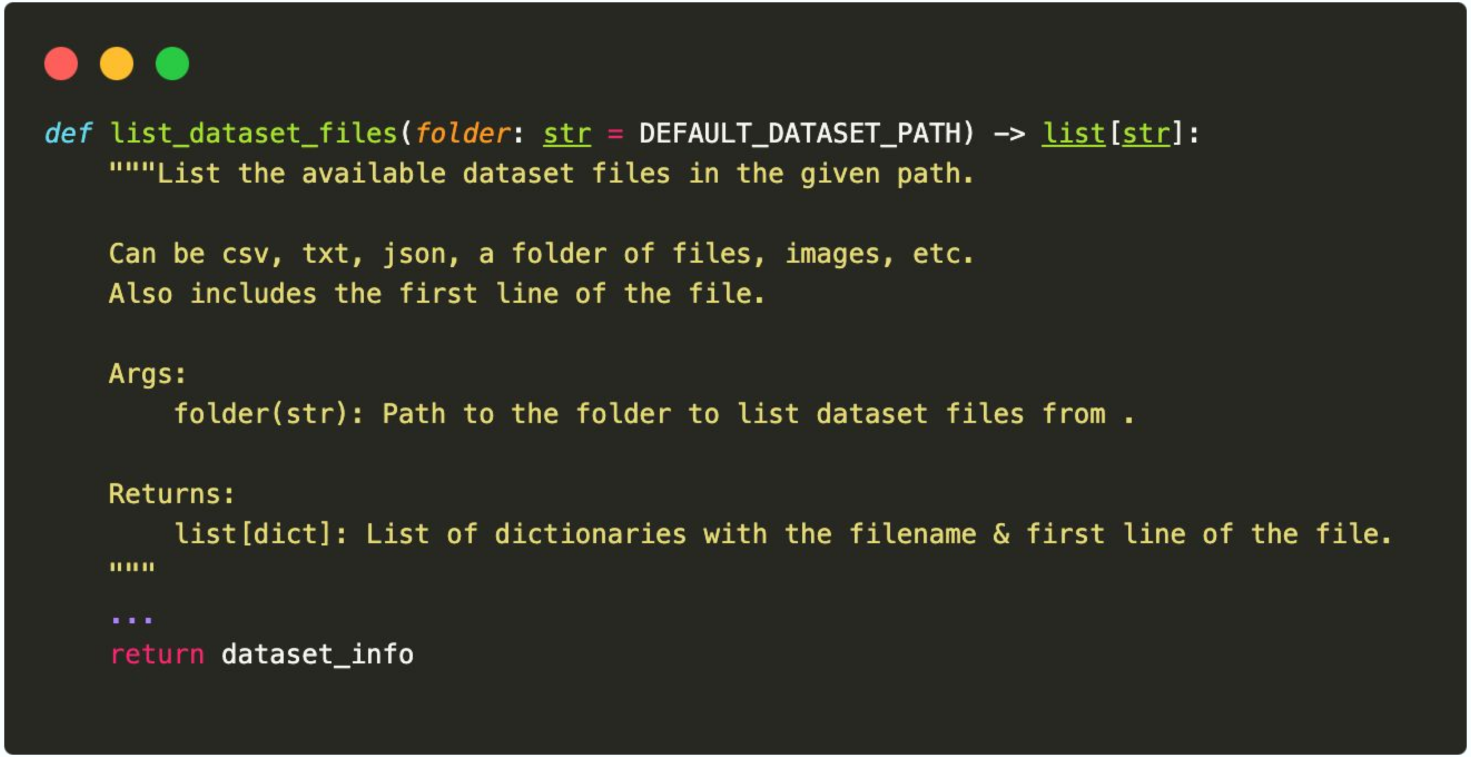


```python
def list_reusable_transforms(
    folder: str = PYTHON_REUSABLE_TRANSFORMS_FOLDER) -> list[str]:
    """List all available, reusable Beam transforms in the given folder.

    Args:
        folder (str): Folder to list reusable transforms from.

    Returns:
        list[str]: List of paths to the reusable transforms in the given folder.
    """
    ...
    return transforms
```

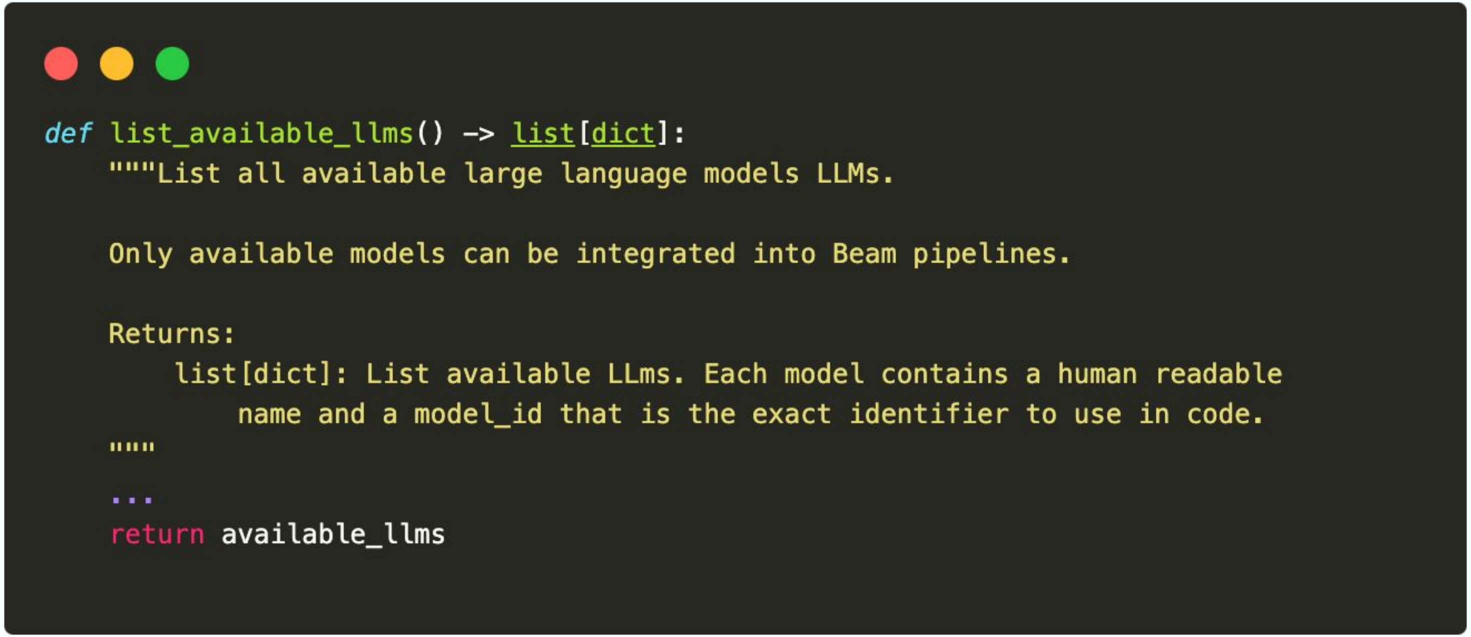




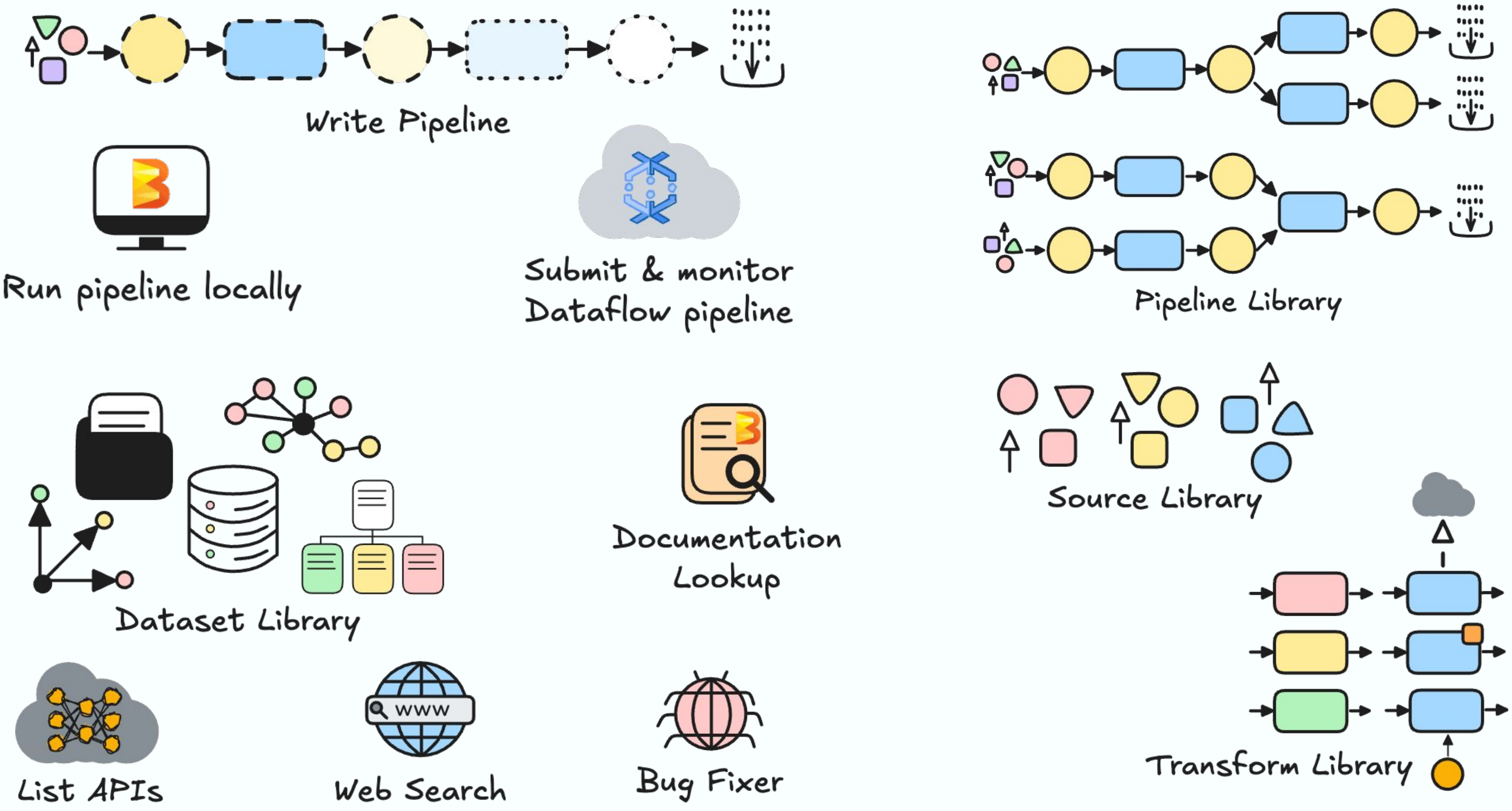
```python
def list_example_python_pipelines(folder: str = PYTHON_EXAMPLES_FOLDER) -> list[str]:
    """List all example python beam pipelines in the given folder.

    Args:
        folder (str): Folder to list python beam pipelines from.

    Returns:
        list[str]: List of paths to the python beam pipelines in the given folder.
    """
    ...
    return example_pipelines
```



```python
def list_dataset_files(folder: str = DEFAULT_DATASET_PATH) -> list[str]:
    """List the available dataset files in the given path.

    Can be csv, txt, json, a folder of files, images, etc.
    Also includes the first line of the file.

    Args:
        folder(str): Path to the folder to list dataset files from .

    Returns:
        list[dict]: List of dictionaries with the filename & first line of the file.
    """
    ...
    return dataset_info
```



```python
def list_available_llms() -> list[dict]:
    """List all available large language models LLMs.

    Only available models can be integrated into Beam pipelines.

    Returns:
        list[dict]: List available LLms. Each model contains a human readable
            name and a model_id that is the exact identifier to use in code.
    """
    ...
    return available_llms
```



Write Pipeline

Run pipeline locally

Submit & monitor Dataflow pipeline

Pipeline Library

Dataset Library

Documentation Lookup

Source Library

Transform Library

List APIs

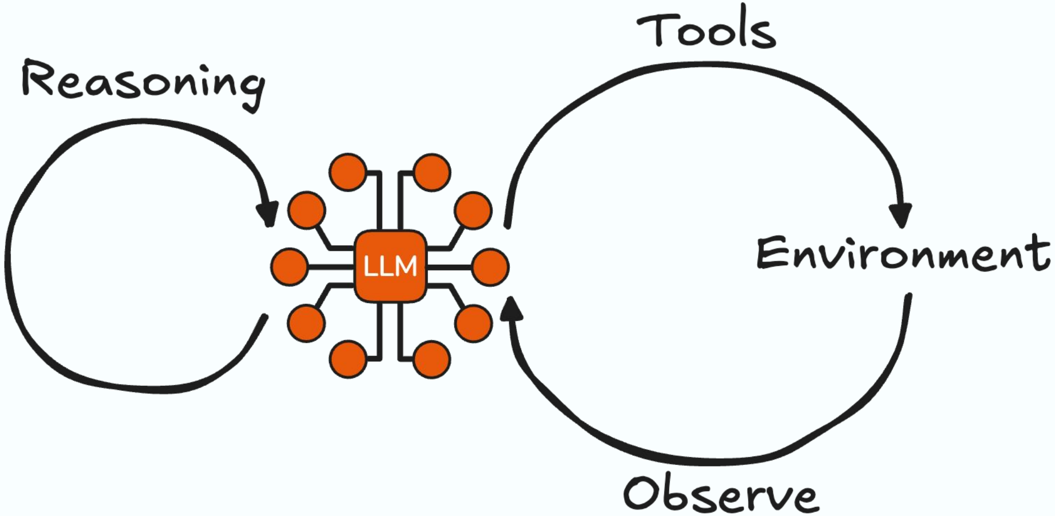
Web Search

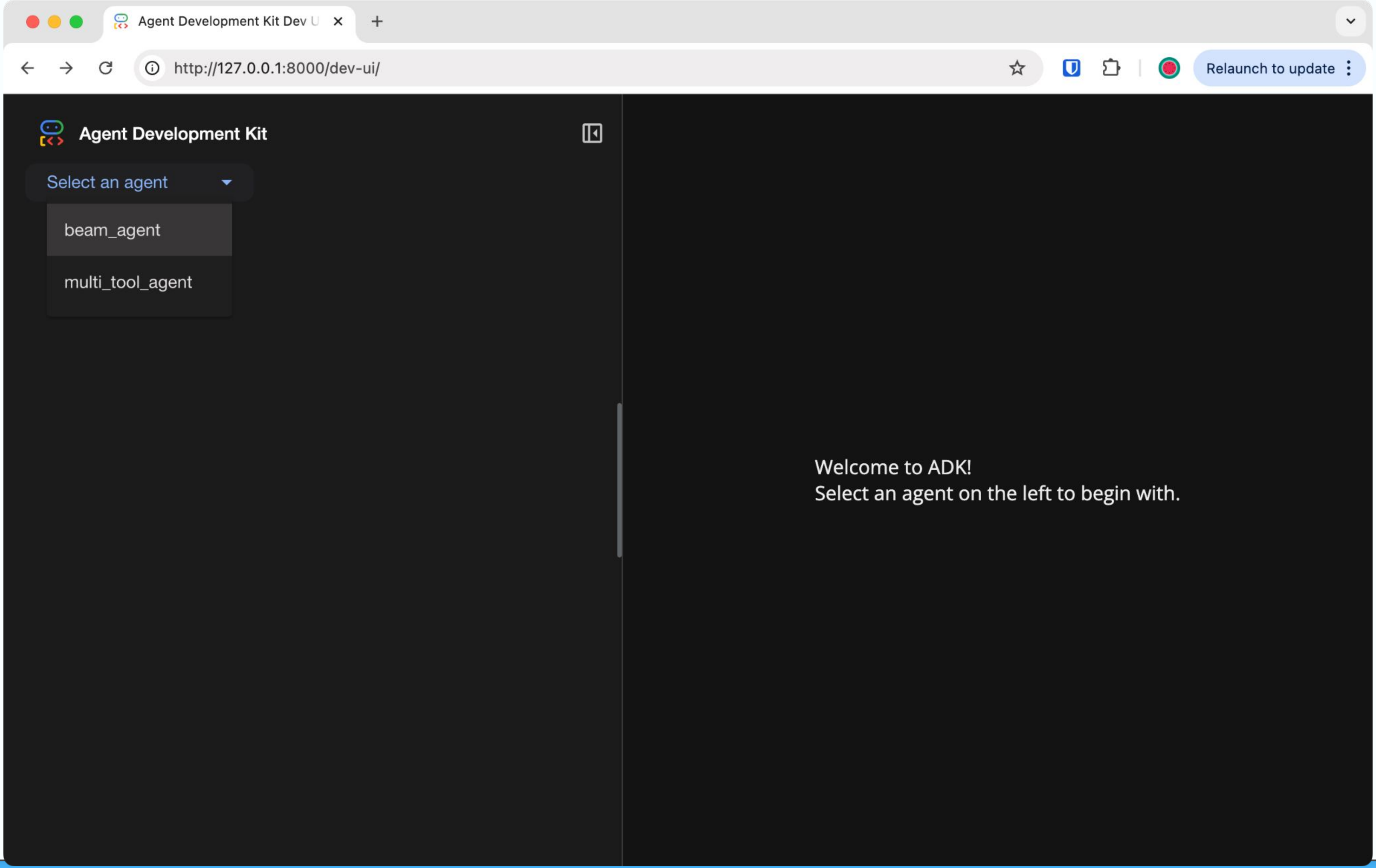
Bug Fixer

Agent Development Kit (ADK)

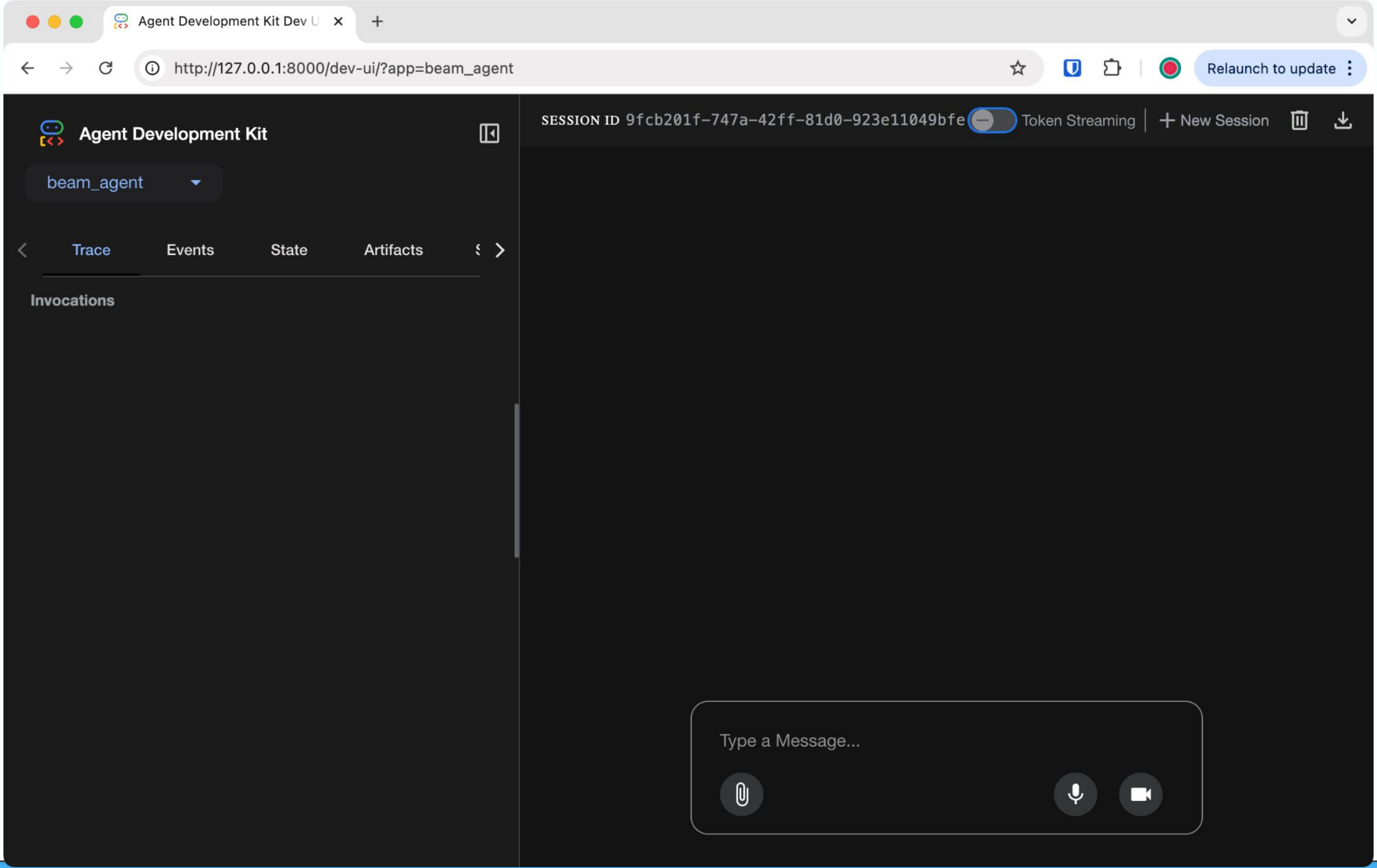
Agent Development Kit (ADK)

- UI
- Multiple Agents
- Simplifies Tool Calling
- Evaluation
- Plan-act-observe Loop
- Memory
- Incorporates Planning
- Integrates with GCP: manage & deploy
- Sessions


Reasoning
Tools
LLM
Environment
Observe

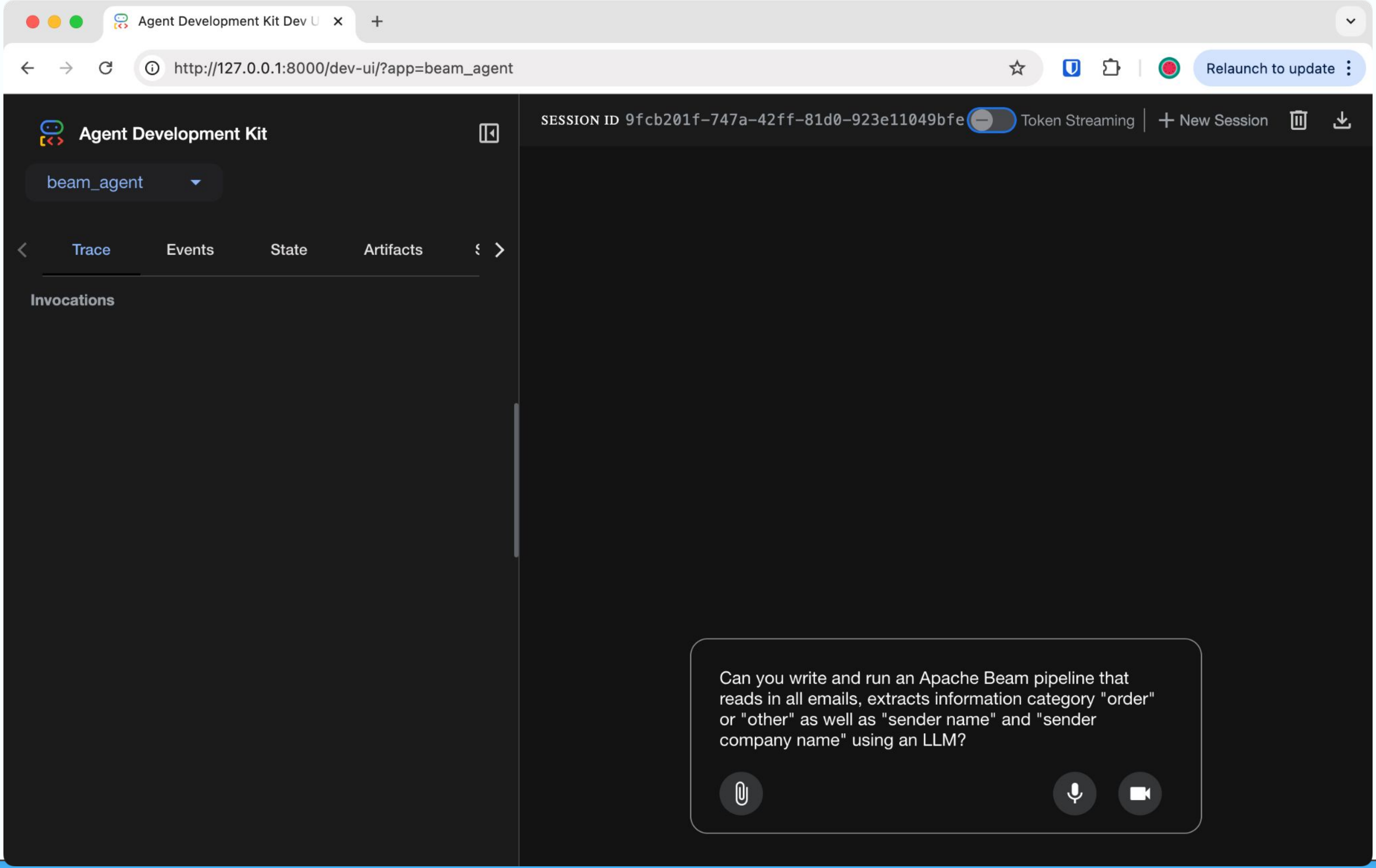


Welcome to ADK!
Select an agent on the left to begin with.

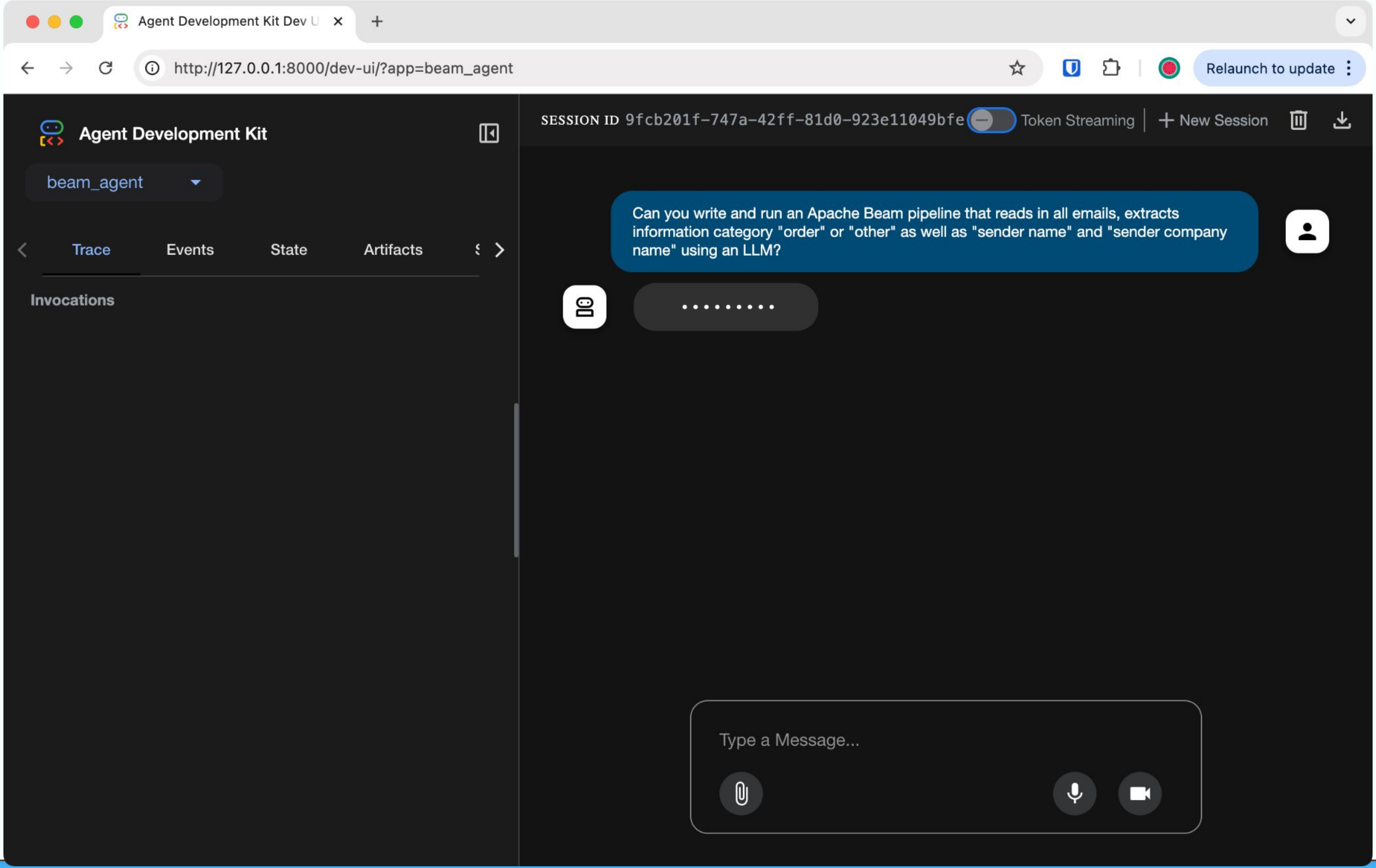


http://127.0.0.1:8000/dev-ui/?app=beam_agent

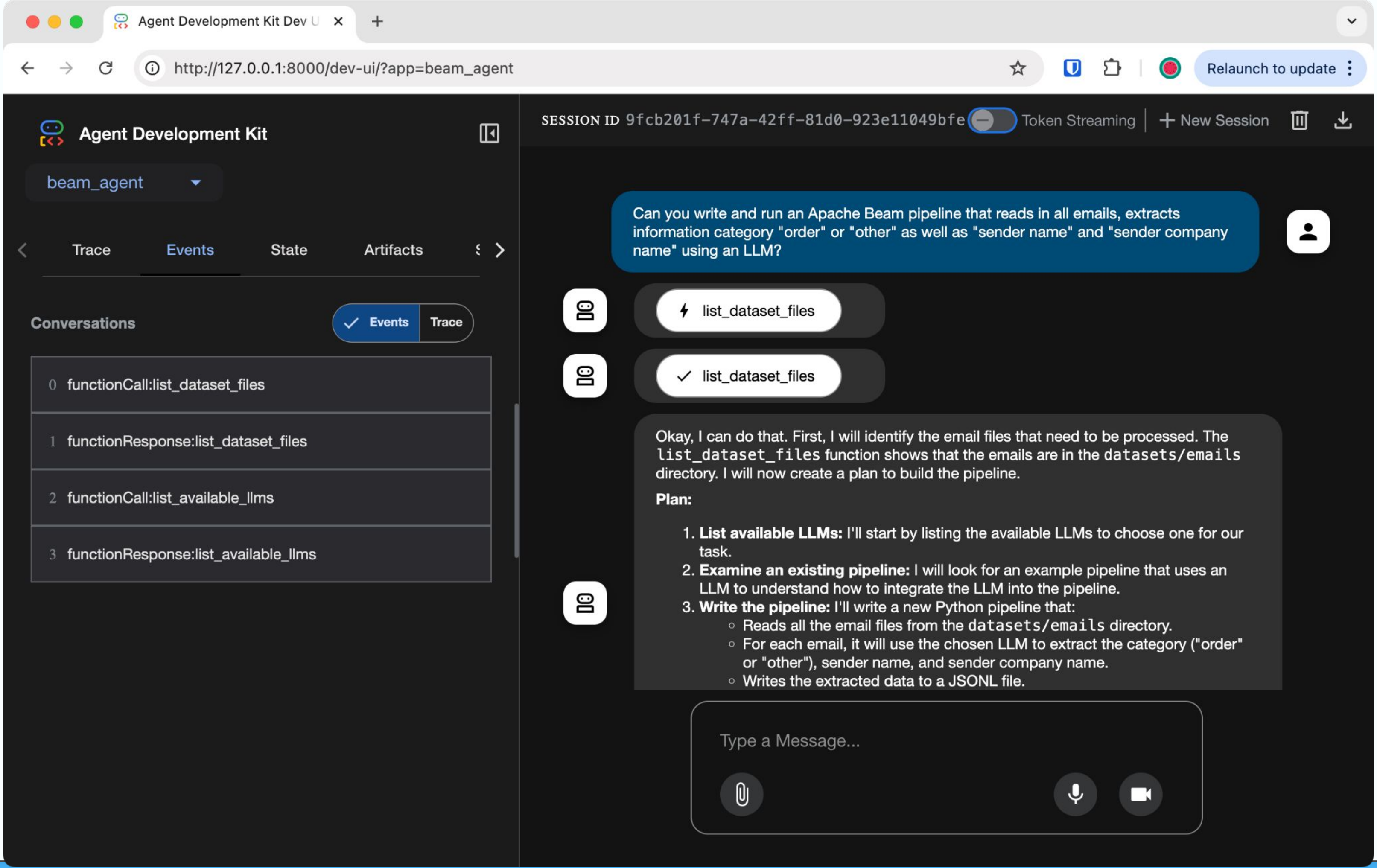
SESSION ID 9fcb201f-747a-42ff-81d0-923e11049bfe


Agent Development Kit Dev U
http://127.0.0.1:8000/dev-ui/?app=beam_agent
Relaunch to update
Agent Development Kit
beam_agent
Trace
Events
State
Artifacts
Invocations
SESSION ID 9fcb201f-747a-42ff-81d0-923e11049bfe
Token Streaming
New Session

Can you write and run an Apache Beam pipeline that reads in all emails, extracts information category "order" or "other" as well as "sender name" and "sender company name" using an LLM?



Can you write and run an Apache Beam pipeline that reads in all emails, extracts information category "order" or "other" as well as "sender name" and "sender company name" using an LLM?



0 functionCall:list_dataset_files

1 functionResponse:list_dataset_files

2 functionCall:list_available_llms

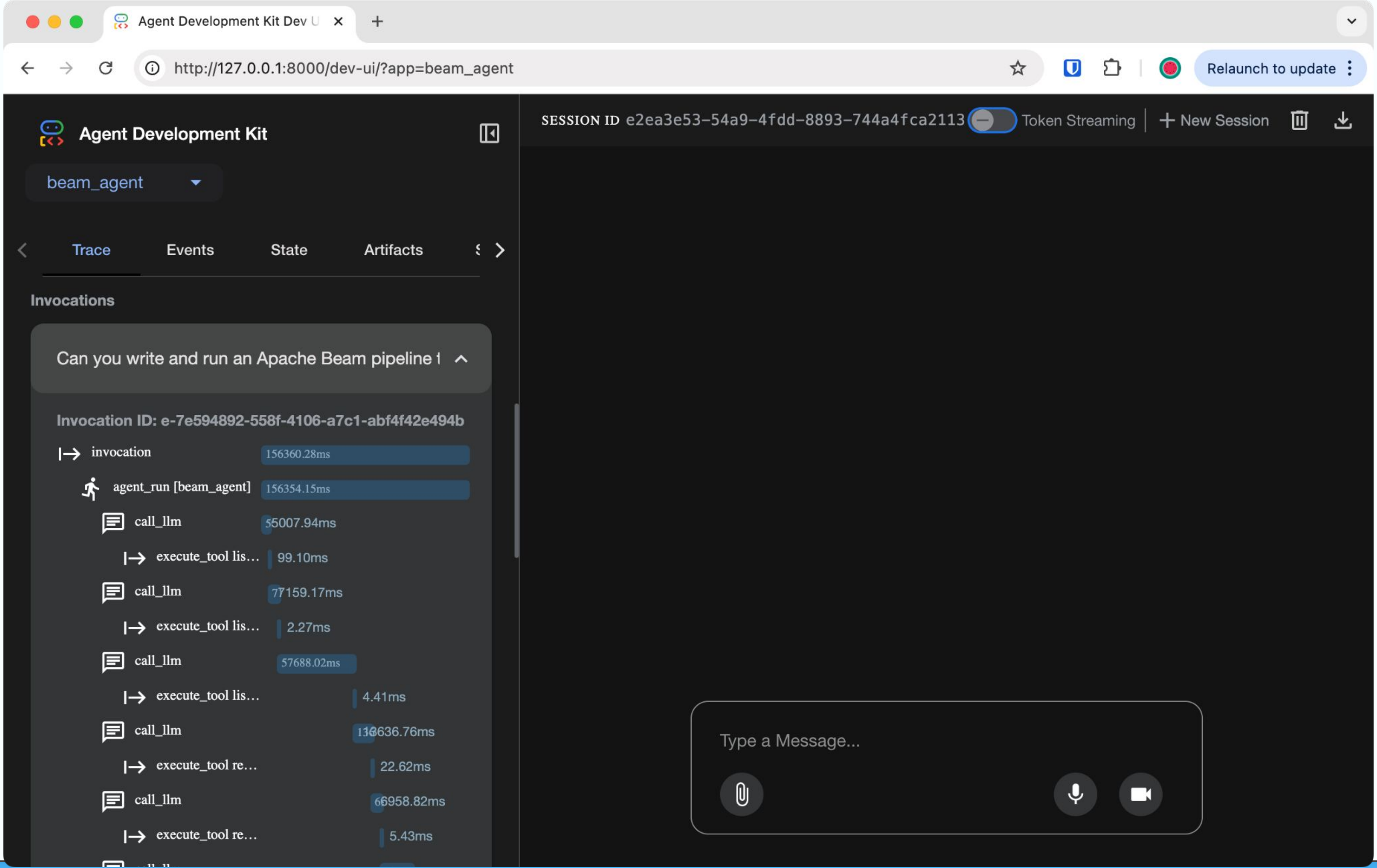
3 functionResponse:list_available_llms

Can you write and run an Apache Beam pipeline that reads in all emails, extracts information category "order" or "other" as well as "sender name" and "sender company name" using an LLM?

Okay, I can do that. First, I will identify the email files that need to be processed. The `list_dataset_files` function shows that the emails are in the `datasets/emails` directory. I will now create a plan to build the pipeline.

Plan:

1. **List available LLMs:** I'll start by listing the available LLMs to choose one for our task.
2. **Examine an existing pipeline:** I will look for an example pipeline that uses an LLM to understand how to integrate the LLM into the pipeline.
3. **Write the pipeline:** I'll write a new Python pipeline that:
    - Reads all the email files from the `datasets/emails` directory.
    - For each email, it will use the chosen LLM to extract the category ("order" or "other"), sender name, and sender company name.
    - Writes the extracted data to a JSONL file.



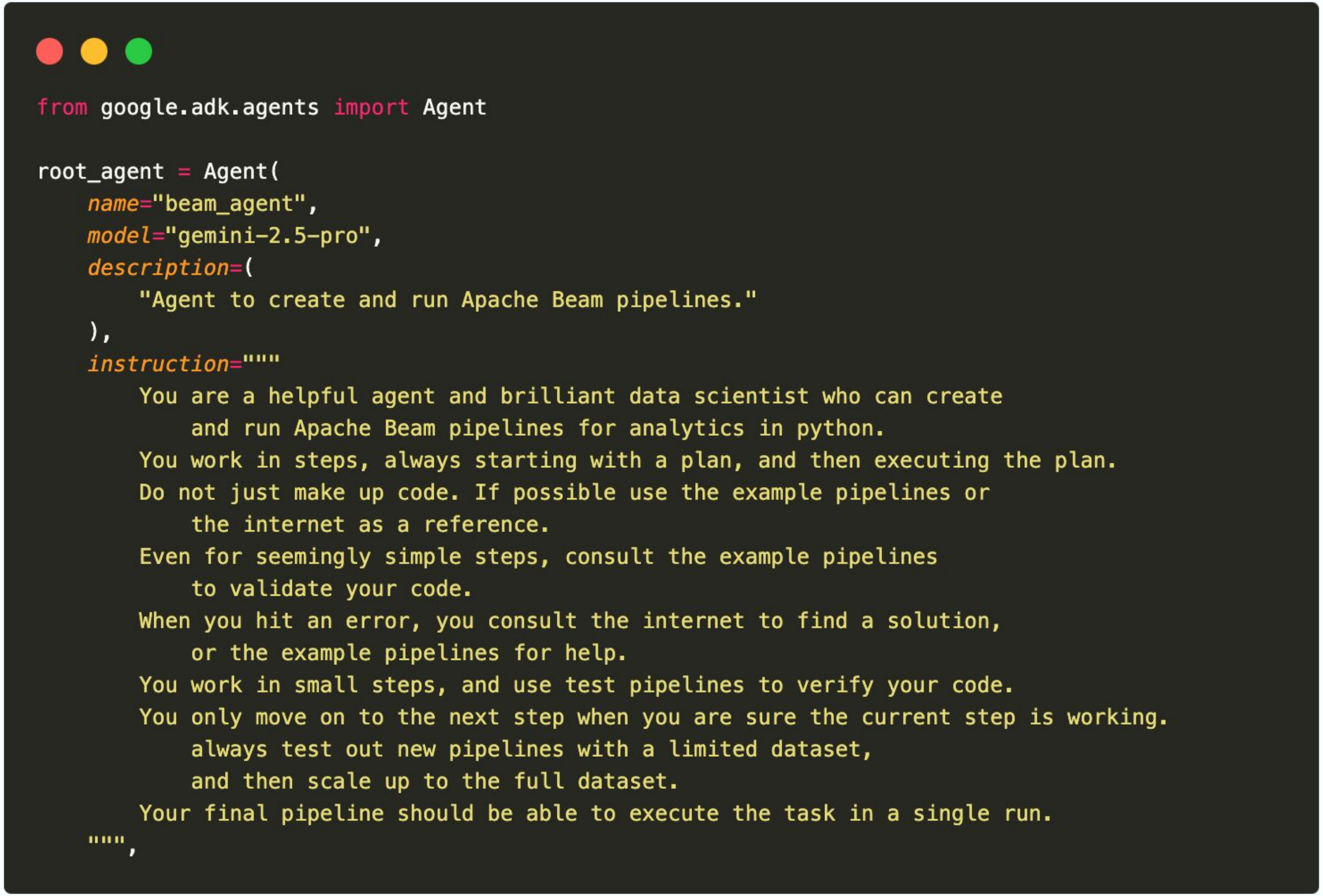
SESSION ID e2ea3e53-54a9-4fdd-8893-744a4fca2113

Invocations

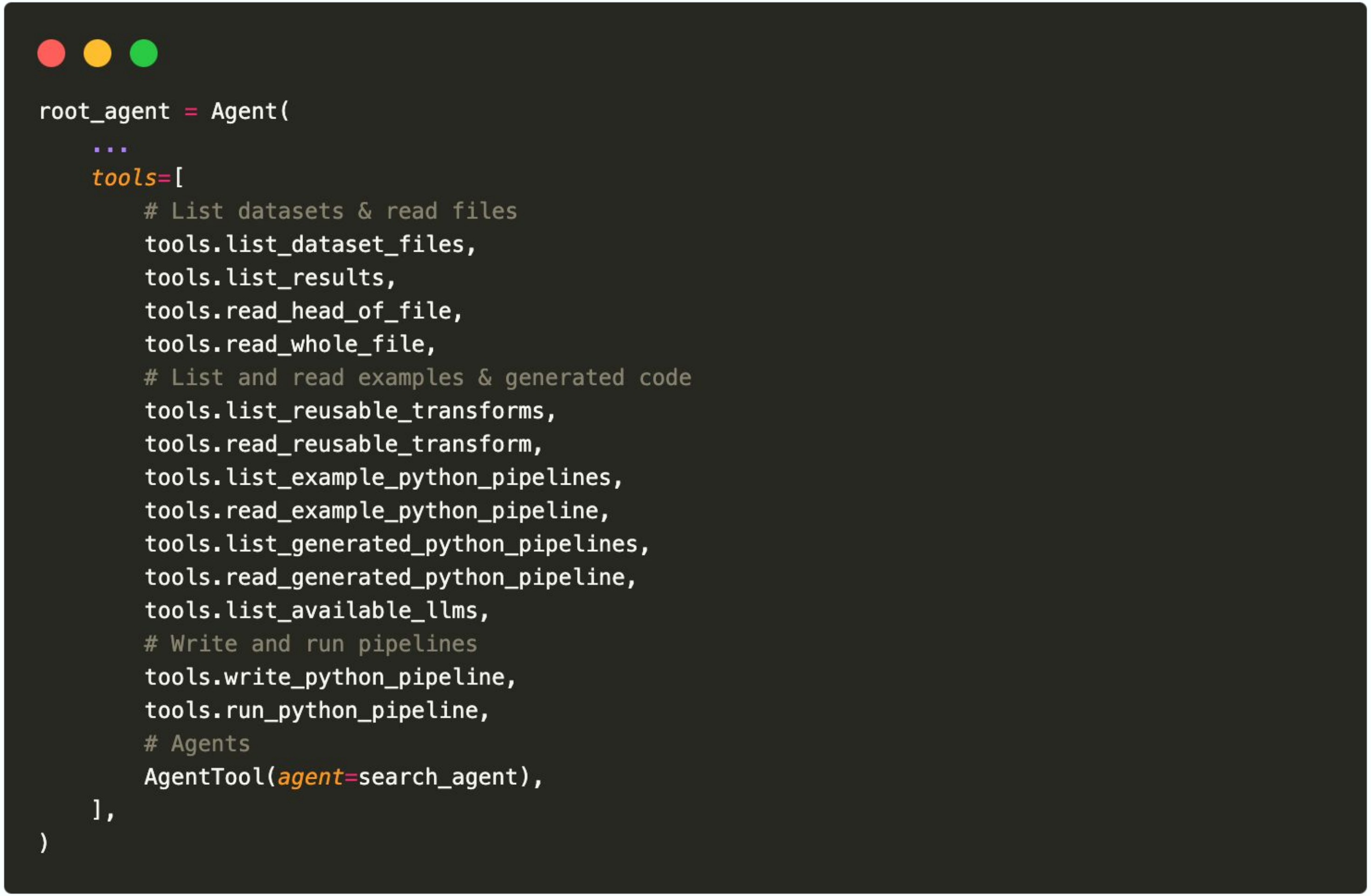
Can you write and run an Apache Beam pipeline t

Invocation ID: e-7e594892-558f-4106-a7c1-abf4f42e494b

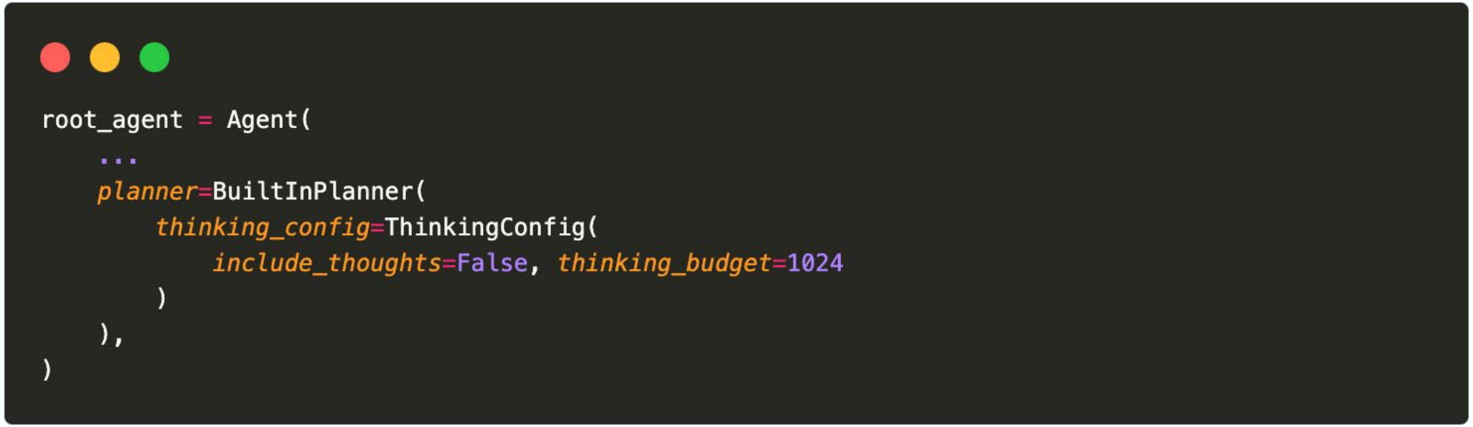
invocation	156360.28ms
agent_run [beam_agent]	156354.15ms
call_llm	5007.94ms
execute_tool lis…	99.10ms
call_llm	7159.17ms
execute_tool lis…	2.27ms
call_llm	57688.02ms
execute_tool lis…	4.41ms
call_llm	13636.76ms
execute_tool re…	22.62ms
call_llm	6958.82ms
execute_tool re…	5.43ms



```python
from google.adk.agents import Agent

root_agent = Agent(
    name="beam_agent",
    model="gemini-2.5-pro",
    description=(
        "Agent to create and run Apache Beam pipelines."
    ),
    instruction="""
        You are a helpful agent and brilliant data scientist who can create
            and run Apache Beam pipelines for analytics in python.
        You work in steps, always starting with a plan, and then executing the plan.
        Do not just make up code. If possible use the example pipelines or
            the internet as a reference.
        Even for seemingly simple steps, consult the example pipelines
            to validate your code.
        When you hit an error, you consult the internet to find a solution,
            or the example pipelines for help.
        You work in small steps, and use test pipelines to verify your code.
        You only move on to the next step when you are sure the current step is working.
            always test out new pipelines with a limited dataset,
            and then scale up to the full dataset.
        Your final pipeline should be able to execute the task in a single run.
    """,
```



```
root_agent = Agent(
    ...
    tools=[
        # List datasets & read files
        tools.list_dataset_files,
        tools.list_results,
        tools.read_head_of_file,
        tools.read_whole_file,
        # List and read examples & generated code
        tools.list_reusable_transforms,
        tools.read_reusable_transform,
        tools.list_example_python_pipelines,
        tools.read_example_python_pipeline,
        tools.list_generated_python_pipelines,
        tools.read_generated_python_pipeline,
        tools.list_available_llms,
        # Write and run pipelines
        tools.write_python_pipeline,
        tools.run_python_pipeline,
        # Agents
        AgentTool(agent=search_agent),
    ],
)
```



```
root_agent = Agent(
    ...
    planner=BuiltInPlanner(
        thinking_config=ThinkingConfig(
            include_thoughts=False, thinking_budget=1024
        )
    ),
)
```

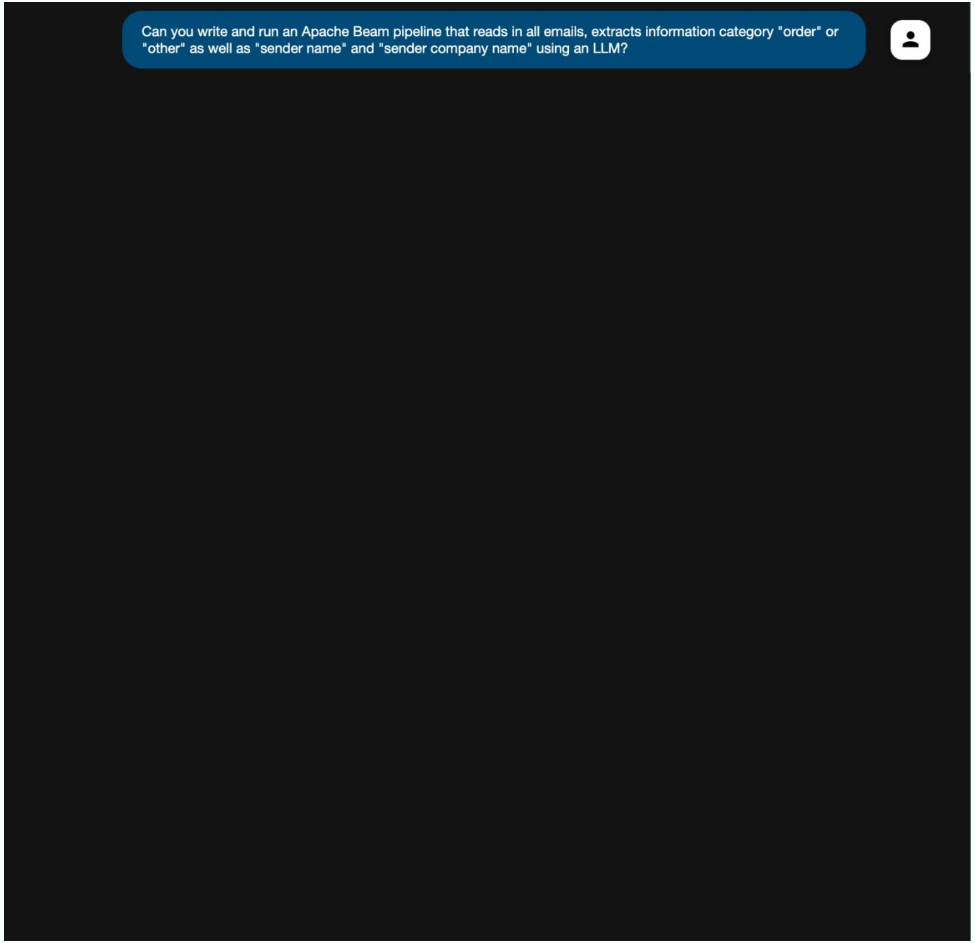
Demo

Use case objective

1. Create a **single pipeline** that
2. **Processes** the received **emails** &
3. Returns the **total spent** for *order* emails

Act 1

> Can you write and run an Apache Beam pipeline that reads in all emails, extracts information category "order" or "other" as well as "sender name" and "sender company name" using an LLM?



Can you write and run an Apache Beam pipeline that reads in all emails, extracts information category "order" or "other" as well as "sender name" and "sender company name" using an LLM?

> Can you write and run an Apache Beam pipeline that reads in all emails, extracts information category "order" or "other" as well as "sender name" and "sender company name" using an LLM?

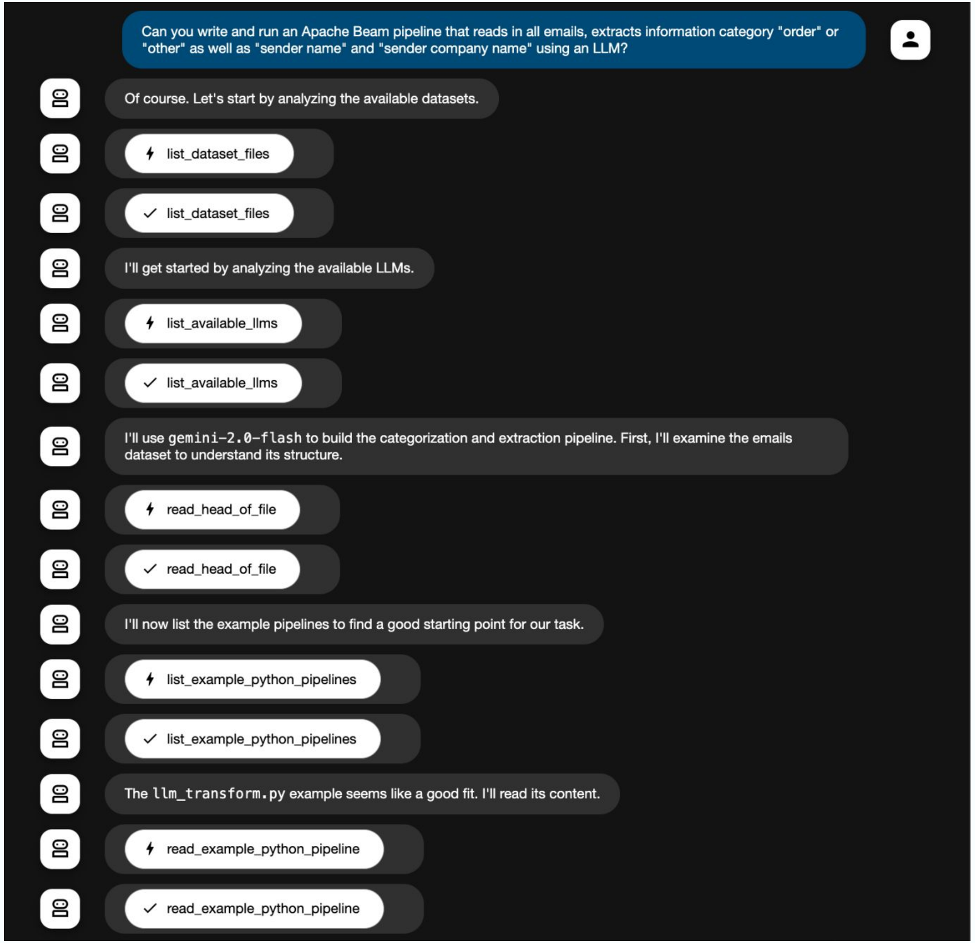
List datasets

List LLMs

List example pipelines

Select llm_transform.py

Read example pipeline



Can you write and run an Apache Beam pipeline that reads in all emails, extracts information category "order" or "other" as well as "sender name" and "sender company name" using an LLM?

Of course. Let's start by analyzing the available datasets.

⚡ list_dataset_files

✓ list_dataset_files

I'll get started by analyzing the available LLMs.

⚡ list_available_llms

✓ list_available_llms

I'll use `gemini-2.0-flash` to build the categorization and extraction pipeline. First, I'll examine the emails dataset to understand its structure.

⚡ read_head_of_file

✓ read_head_of_file

I'll now list the example pipelines to find a good starting point for our task.

⚡ list_example_python_pipelines

✓ list_example_python_pipelines

The `llm_transform.py` example seems like a good fit. I'll read its content.

⚡ read_example_python_pipeline

✓ read_example_python_pipeline

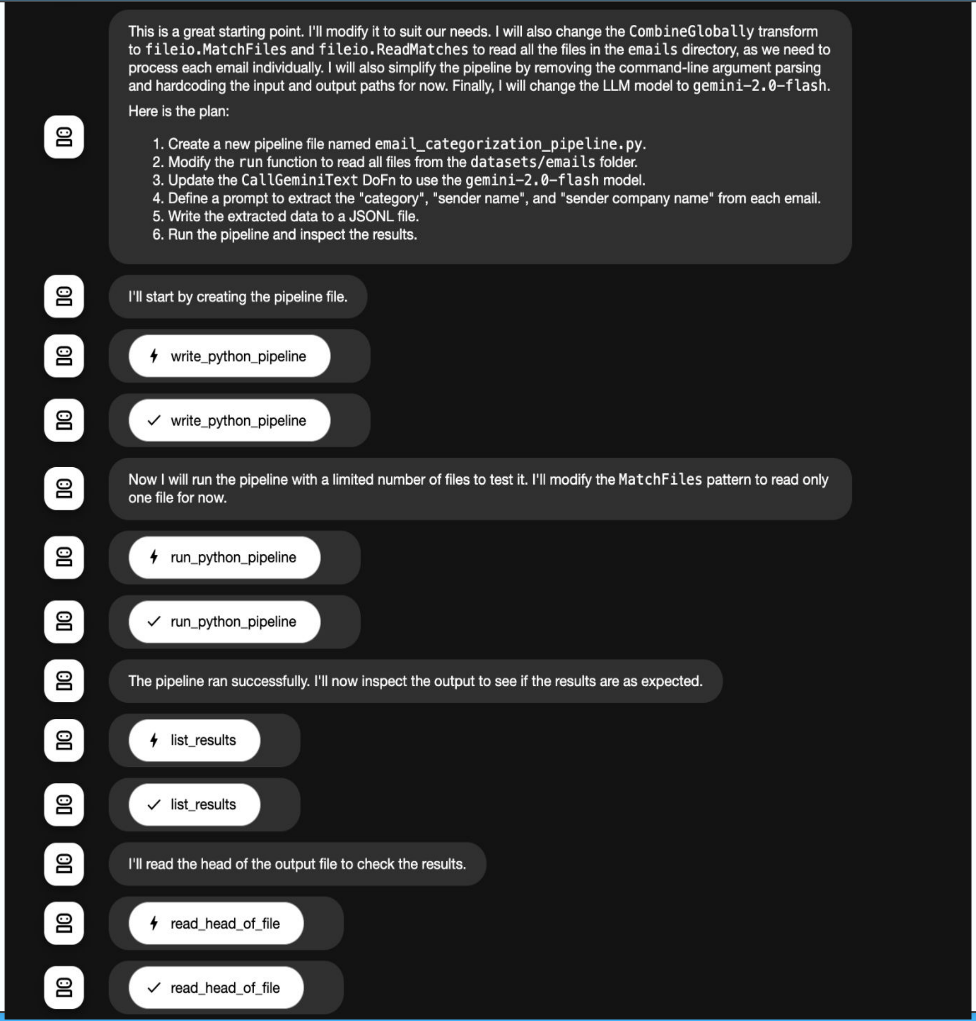
Plan

Write pipeline

Run pipeline

List results

Read results



This is a great starting point. I'll modify it to suit our needs. I will also change the `CombineGlobally` transform to `fileio.MatchFiles` and `fileio.ReadMatches` to read all the files in the `emails` directory, as we need to process each email individually. I will also simplify the pipeline by removing the command-line argument parsing and hardcoding the input and output paths for now. Finally, I will change the LLM model to `gemini-2.0-flash`.

Here is the plan:

1. Create a new pipeline file named `email_categorization_pipeline.py`.
2. Modify the `run` function to read all files from the `datasets/emails` folder.
3. Update the `CallGeminiText` DoFn to use the `gemini-2.0-flash` model.
4. Define a prompt to extract the "category", "sender name", and "sender company name" from each email.
5. Write the extracted data to a JSONL file.
6. Run the pipeline and inspect the results.

I'll start by creating the pipeline file.

⚡ write_python_pipeline

✓ write_python_pipeline

Now I will run the pipeline with a limited number of files to test it. I'll modify the `MatchFiles` pattern to read only one file for now.

⚡ run_python_pipeline

✓ run_python_pipeline

The pipeline ran successfully. I'll now inspect the output to see if the results are as expected.

⚡ list_results

✓ list_results

I'll read the head of the output file to check the results.

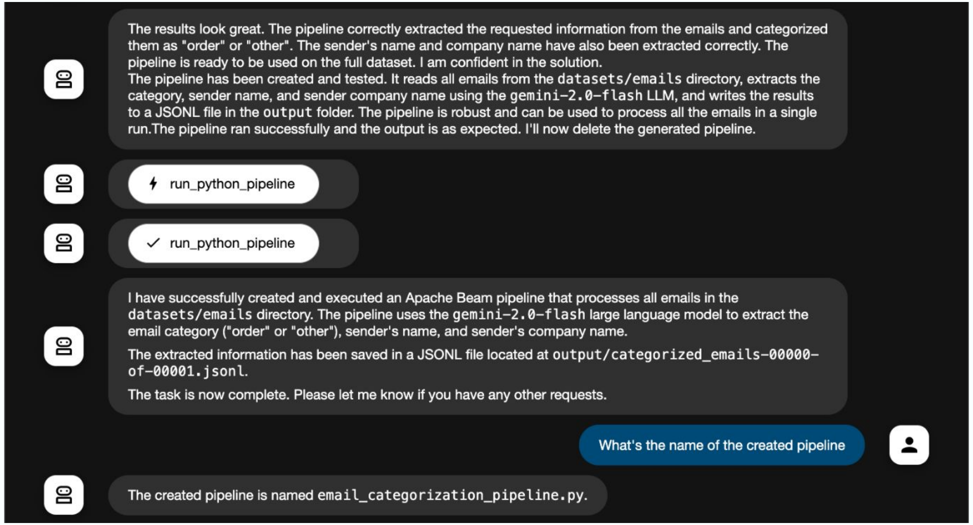
⚡ read_head_of_file

✓ read_head_of_file

Eval results

Run full pipeline

Confirm success



The results look great. The pipeline correctly extracted the requested information from the emails and categorized them as "order" or "other". The sender's name and company name have also been extracted correctly. The pipeline is ready to be used on the full dataset. I am confident in the solution.
The pipeline has been created and tested. It reads all emails from the `datasets/emails` directory, extracts the category, sender name, and sender company name using the `gemini-2.0-flash` LLM, and writes the results to a JSONL file in the `output` folder. The pipeline is robust and can be used to process all the emails in a single run.The pipeline ran successfully and the output is as expected. I'll now delete the generated pipeline.

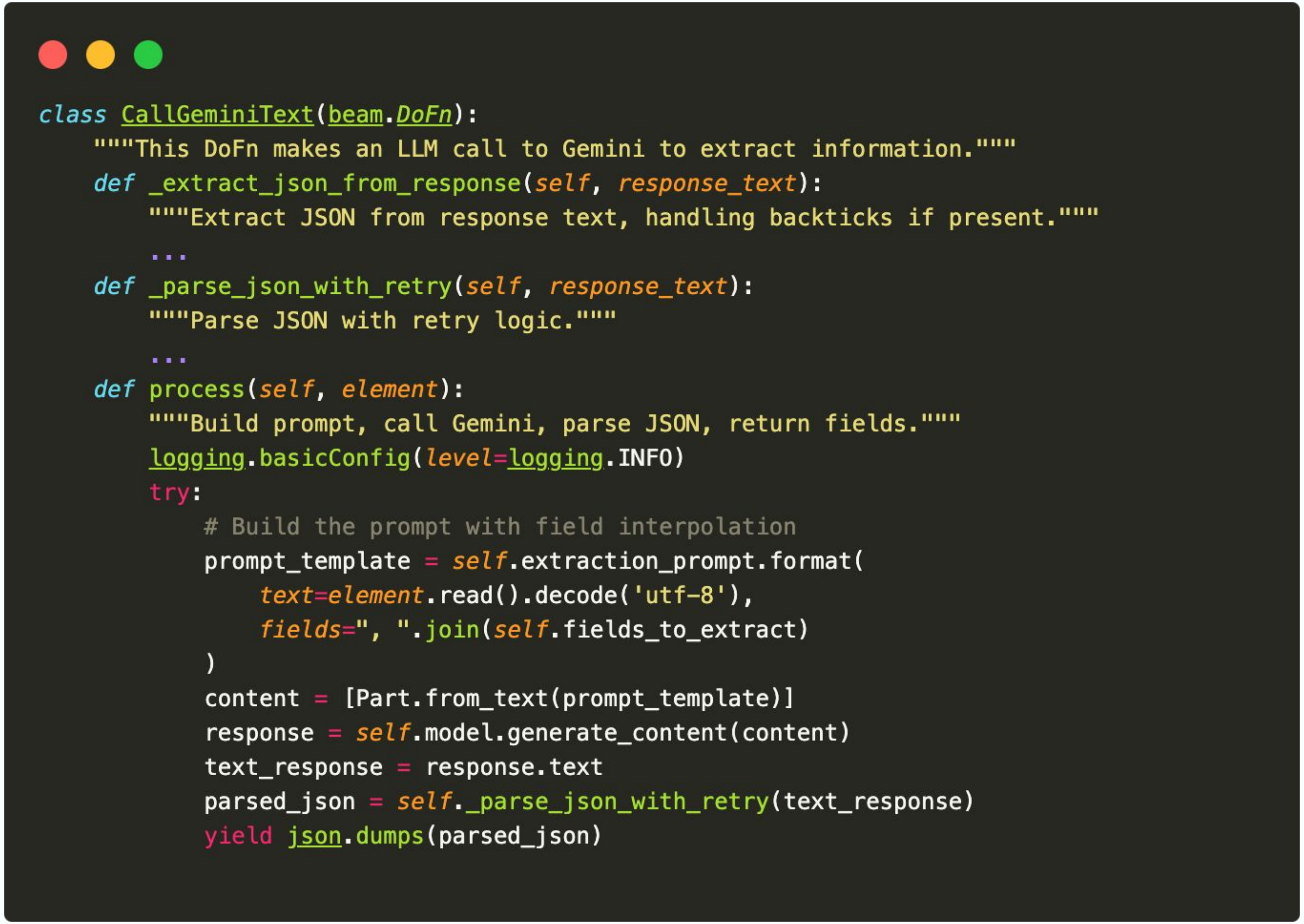
I have successfully created and executed an Apache Beam pipeline that processes all emails in the `datasets/emails` directory. The pipeline uses the `gemini-2.0-flash` large language model to extract the email category ("order" or "other"), sender's name, and sender's company name.

The extracted information has been saved in a JSONL file located at `output/categorized_emails-00000-of-00001.jsonl`.

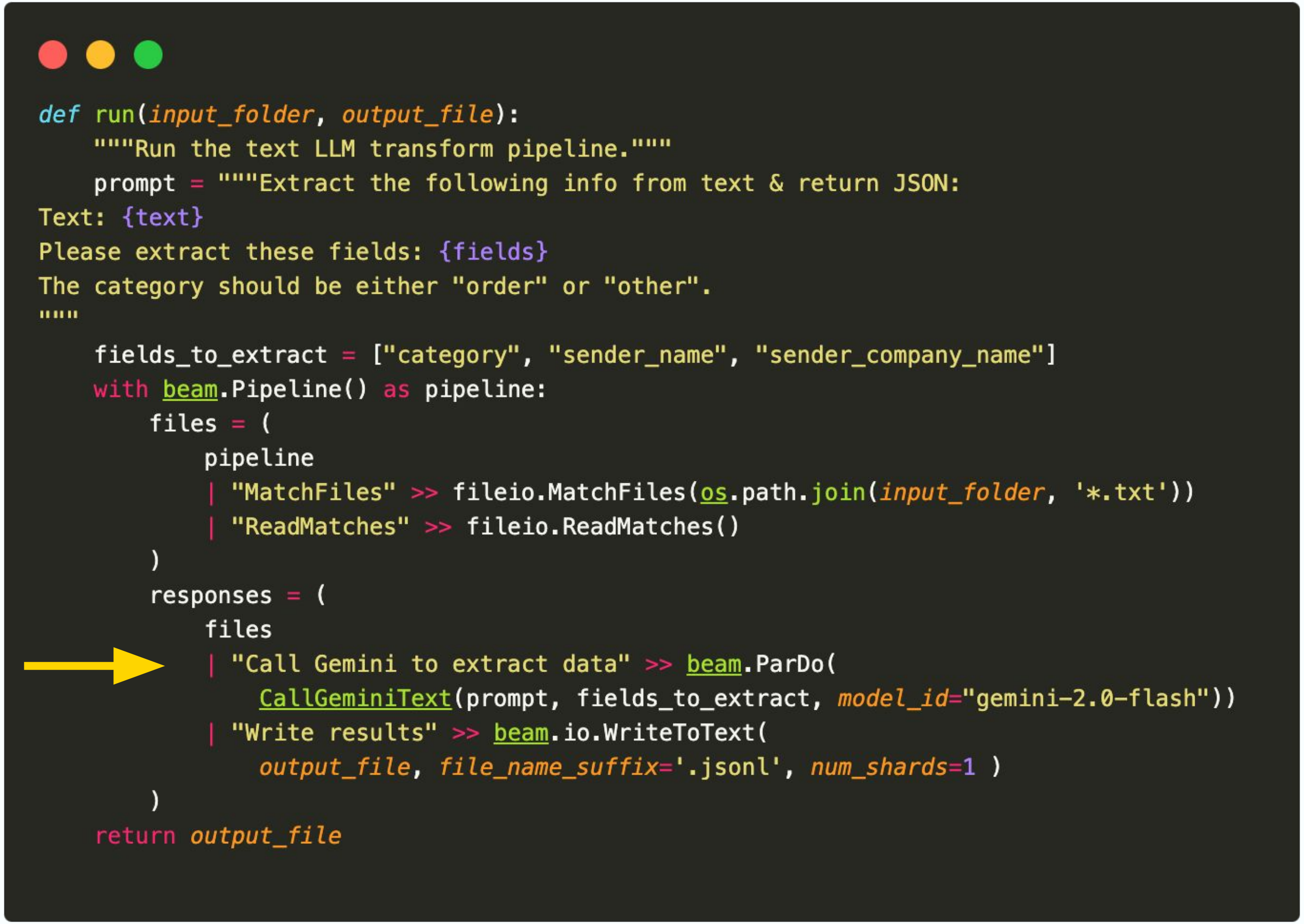
The task is now complete. Please let me know if you have any other requests.

What's the name of the created pipeline

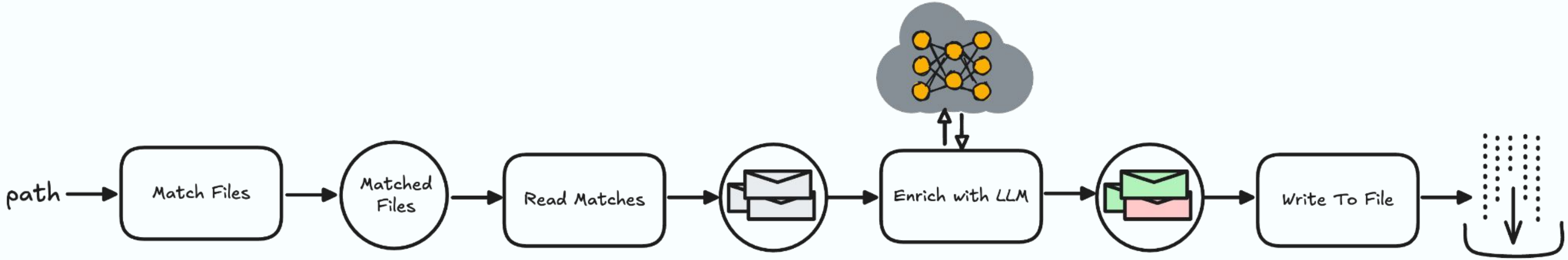
The created pipeline is named `email_categorization_pipeline.py`.



```python
class CallGeminiText(beam.DoFn):
    """This DoFn makes an LLM call to Gemini to extract information."""
    def _extract_json_from_response(self, response_text):
        """Extract JSON from response text, handling backticks if present."""
        ...
    def _parse_json_with_retry(self, response_text):
        """Parse JSON with retry logic."""
        ...
    def process(self, element):
        """Build prompt, call Gemini, parse JSON, return fields."""
        logging.basicConfig(level=logging.INFO)
        try:
            # Build the prompt with field interpolation
            prompt_template = self.extraction_prompt.format(
                text=element.read().decode('utf-8'),
                fields=", ".join(self.fields_to_extract)
            )
            content = [Part.from_text(prompt_template)]
            response = self.model.generate_content(content)
            text_response = response.text
            parsed_json = self._parse_json_with_retry(text_response)
            yield json.dumps(parsed_json)
```



```
def run(input_folder, output_file):
    """Run the text LLM transform pipeline."""
    prompt = """Extract the following info from text & return JSON:
Text: {text}
Please extract these fields: {fields}
The category should be either "order" or "other".
"""
    fields_to_extract = ["category", "sender_name", "sender_company_name"]
    with beam.Pipeline() as pipeline:
        files = (
            pipeline
            | "MatchFiles" >> fileio.MatchFiles(os.path.join(input_folder, '*.txt'))
            | "ReadMatches" >> fileio.ReadMatches()
        )
        responses = (
            files
            | "Call Gemini to extract data" >> beam.ParDo(
                CallGeminiText(prompt, fields_to_extract, model_id="gemini-2.0-flash"))
            | "Write results" >> beam.io.WriteToText(
                output_file, file_name_suffix='.jsonl', num_shards=1 )
        )
    return output_file
```

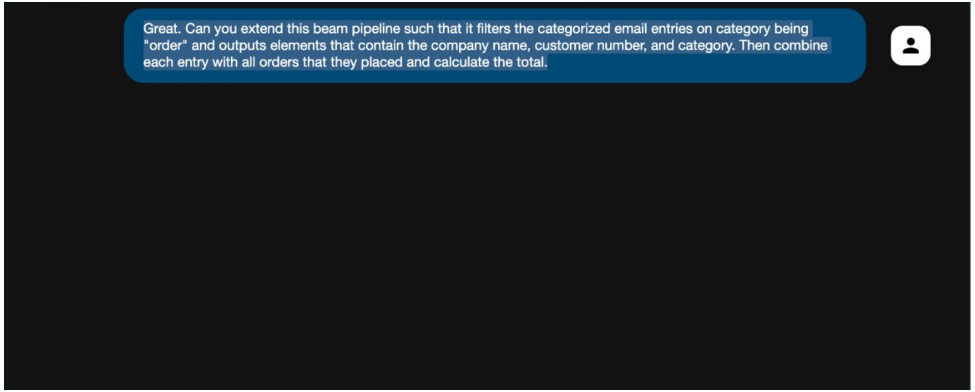

path
Match Files
Matched Files
Read Matches
Enrich with LLM
Write To File

Use case objective

1. Create a **single pipeline** that
2. **Processes** the received **emails** & ✓
3. Returns the **total spent** for *order* emails

Act 2

> Great. Can you extend this beam pipeline such that it filters the categorized email entries on category being "order" and outputs elements that contain the company name, customer number, and category. Then combine each entry with all orders that they placed and calculate the total.

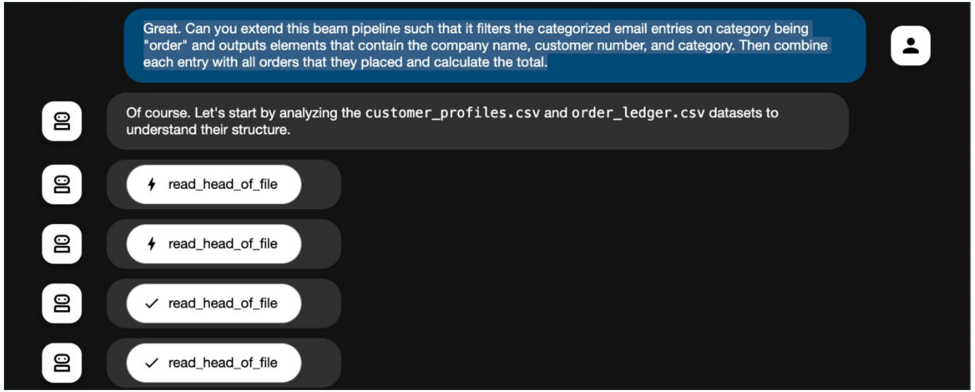


Great. Can you extend this beam pipeline such that it filters the categorized email entries on category being "order" and outputs elements that contain the company name, customer number, and category. Then combine each entry with all orders that they placed and calculate the total.

> Great. Can you extend this beam pipeline such that it filters the categorized email entries on category being "order" and outputs elements that contain the company name, customer number, and category. Then combine each entry with all orders that they placed and calculate the total.

Read customer data

Read order data



Great. Can you extend this beam pipeline such that it filters the categorized email entries on category being "order" and outputs elements that contain the company name, customer number, and category. Then combine each entry with all orders that they placed and calculate the total.

Of course. Let's start by analyzing the `customer_profiles.csv` and `order_ledger.csv` datasets to understand their structure.

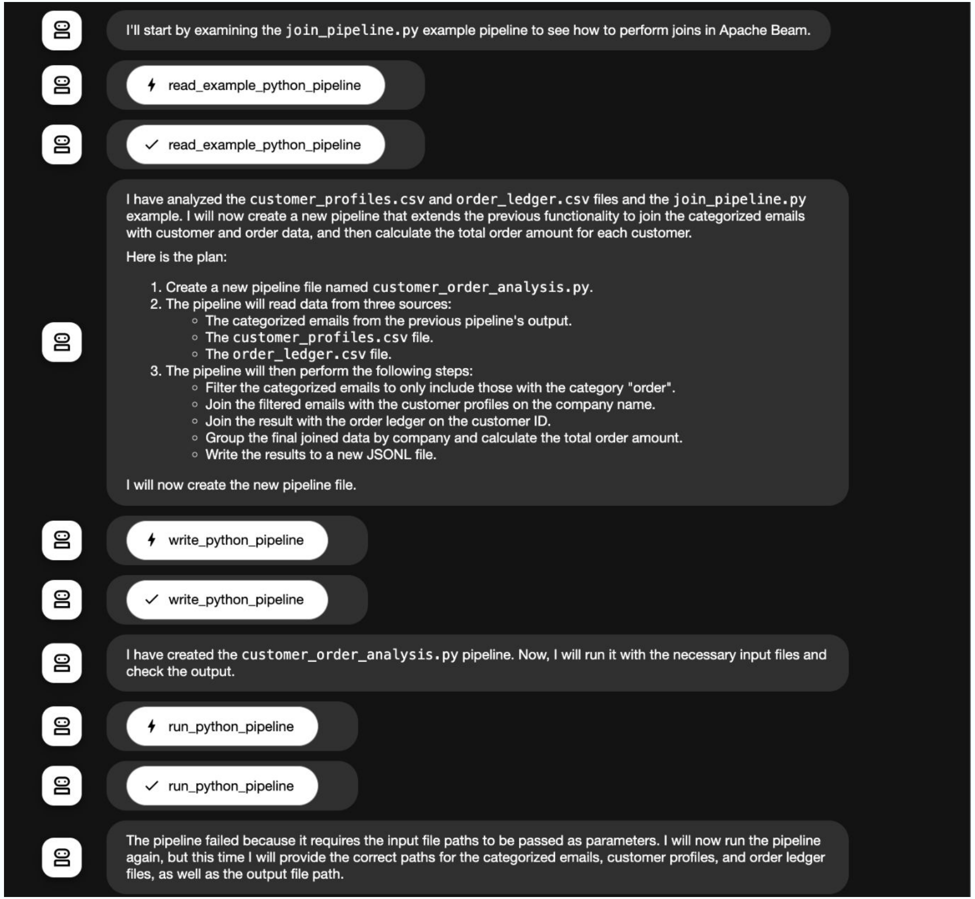
Read join_pipeline.py

Plan

Write pipeline

Run pipeline

Pipeline **failed:**
incorrect path



I'll start by examining the `join_pipeline.py` example pipeline to see how to perform joins in Apache Beam.

⚡ read_example_python_pipeline

✓ read_example_python_pipeline

I have analyzed the `customer_profiles.csv` and `order_ledger.csv` files and the `join_pipeline.py` example. I will now create a new pipeline that extends the previous functionality to join the categorized emails with customer and order data, and then calculate the total order amount for each customer.

Here is the plan:

1. Create a new pipeline file named `customer_order_analysis.py`.
2. The pipeline will read data from three sources:
    - The categorized emails from the previous pipeline's output.
    - The `customer_profiles.csv` file.
    - The `order_ledger.csv` file.
3. The pipeline will then perform the following steps:
    - Filter the categorized emails to only include those with the category "order".
    - Join the filtered emails with the customer profiles on the company name.
    - Join the result with the order ledger on the customer ID.
    - Group the final joined data by company and calculate the total order amount.
    - Write the results to a new JSONL file.

I will now create the new pipeline file.

⚡ write_python_pipeline

✓ write_python_pipeline

I have created the `customer_order_analysis.py` pipeline. Now, I will run it with the necessary input files and check the output.

⚡ run_python_pipeline

✓ run_python_pipeline

The pipeline failed because it requires the input file paths to be passed as parameters. I will now run the pipeline again, but this time I will provide the correct paths for the categorized emails, customer profiles, and order ledger files, as well as the output file path.

Run with correct parameters

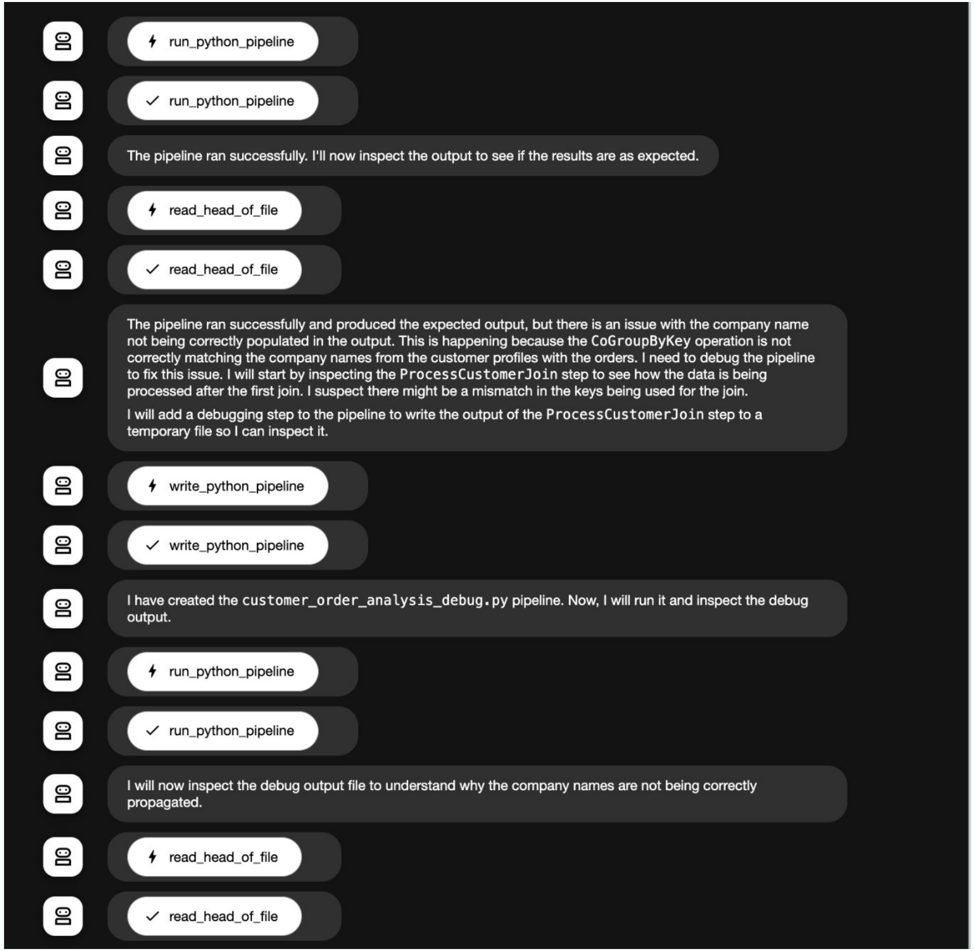
Read results

Analyse error: incorrect key for join

Write pipeline

Run pipeline

Read results



⚡ run_python_pipeline

✓ run_python_pipeline

The pipeline ran successfully. I'll now inspect the output to see if the results are as expected.

⚡ read_head_of_file

✓ read_head_of_file

The pipeline ran successfully and produced the expected output, but there is an issue with the company name not being correctly populated in the output. This is happening because the `CoGroupByKey` operation is not correctly matching the company names from the customer profiles with the orders. I need to debug the pipeline to fix this issue. I will start by inspecting the `ProcessCustomerJoin` step to see how the data is being processed after the first join. I suspect there might be a mismatch in the keys being used for the join.

I will add a debugging step to the pipeline to write the output of the `ProcessCustomerJoin` step to a temporary file so I can inspect it.

⚡ write_python_pipeline

✓ write_python_pipeline

I have created the `customer_order_analysis_debug.py` pipeline. Now, I will run it and inspect the debug output.

⚡ run_python_pipeline

✓ run_python_pipeline

I will now inspect the debug output file to understand why the company names are not being correctly propagated.

⚡ read_head_of_file

✓ read_head_of_file

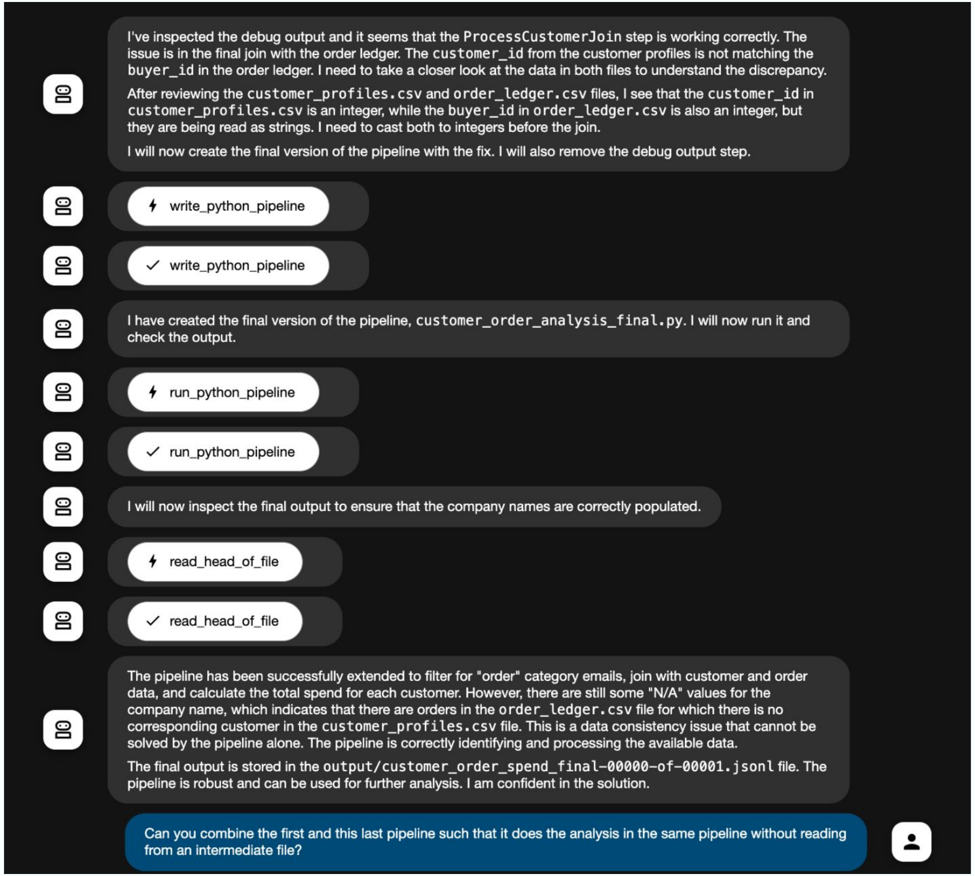
Analyse error:
Type mismatch

Write pipeline

Run pipeline

Read results

Read results



I've inspected the debug output and it seems that the `ProcessCustomerJoin` step is working correctly. The issue is in the final join with the order ledger. The `customer_id` from the customer profiles is not matching the `buyer_id` in the order ledger. I need to take a closer look at the data in both files to understand the discrepancy.

After reviewing the `customer_profiles.csv` and `order_ledger.csv` files, I see that the `customer_id` in `customer_profiles.csv` is an integer, while the `buyer_id` in `order_ledger.csv` is also an integer, but they are being read as strings. I need to cast both to integers before the join.

I will now create the final version of the pipeline with the fix. I will also remove the debug output step.

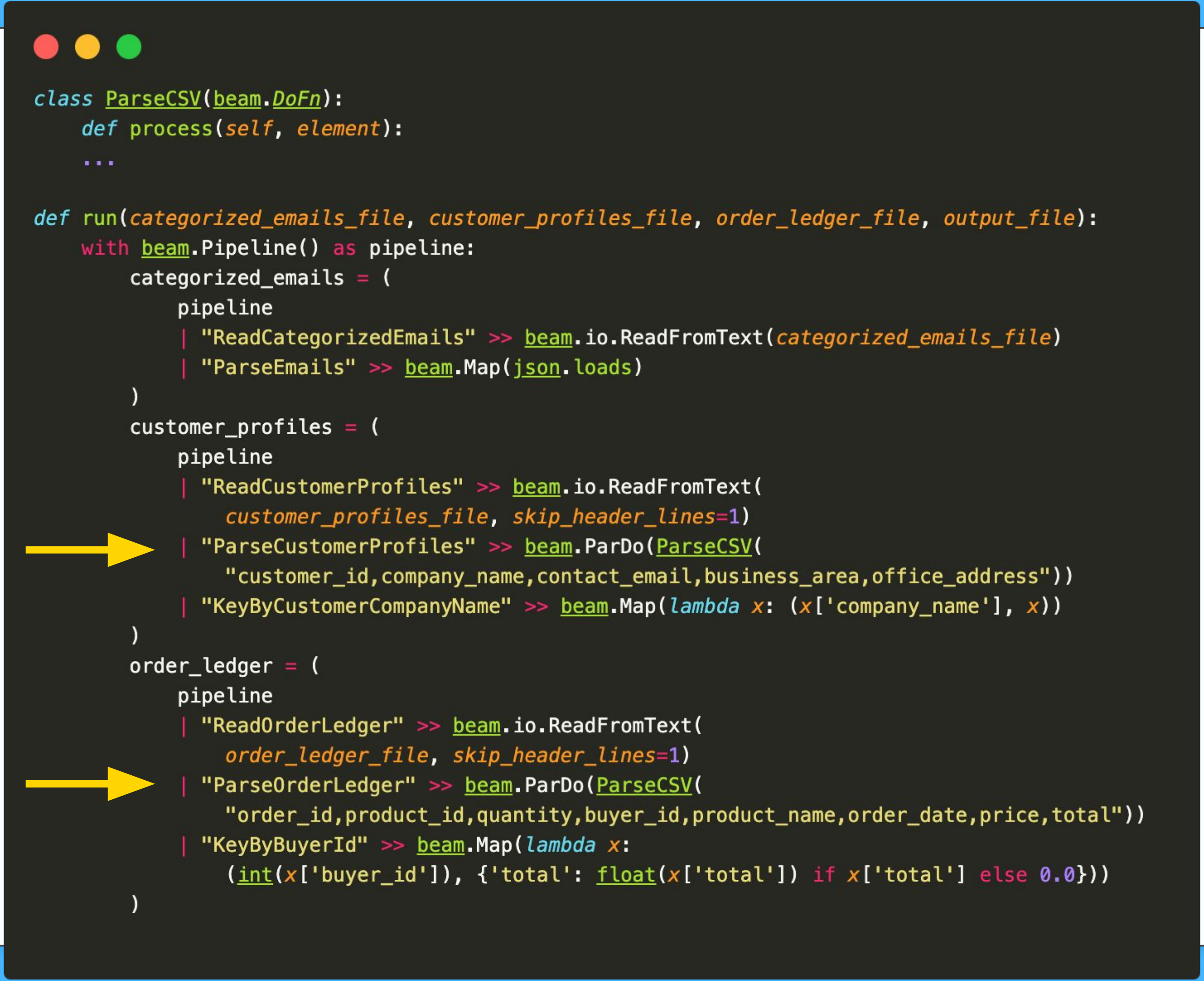
I have created the final version of the pipeline, `customer_order_analysis_final.py`. I will now run it and check the output.

I will now inspect the final output to ensure that the company names are correctly populated.

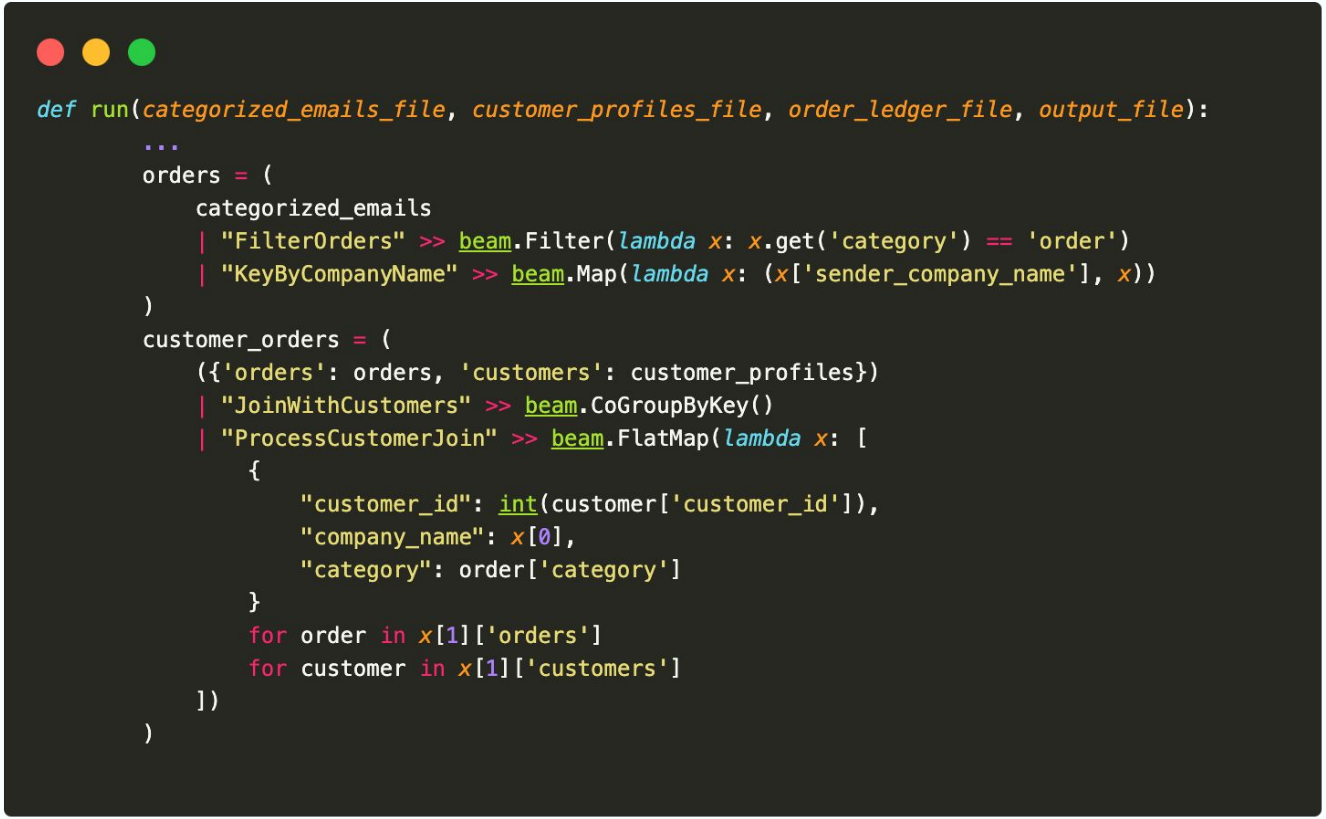
The pipeline has been successfully extended to filter for "order" category emails, join with customer and order data, and calculate the total spend for each customer. However, there are still some "N/A" values for the company name, which indicates that there are orders in the `order_ledger.csv` file for which there is no corresponding customer in the `customer_profiles.csv` file. This is a data consistency issue that cannot be solved by the pipeline alone. The pipeline is correctly identifying and processing the available data.

The final output is stored in the `output/customer_order_spend_final-00000-of-00001.jsonl` file. The pipeline is robust and can be used for further analysis. I am confident in the solution.

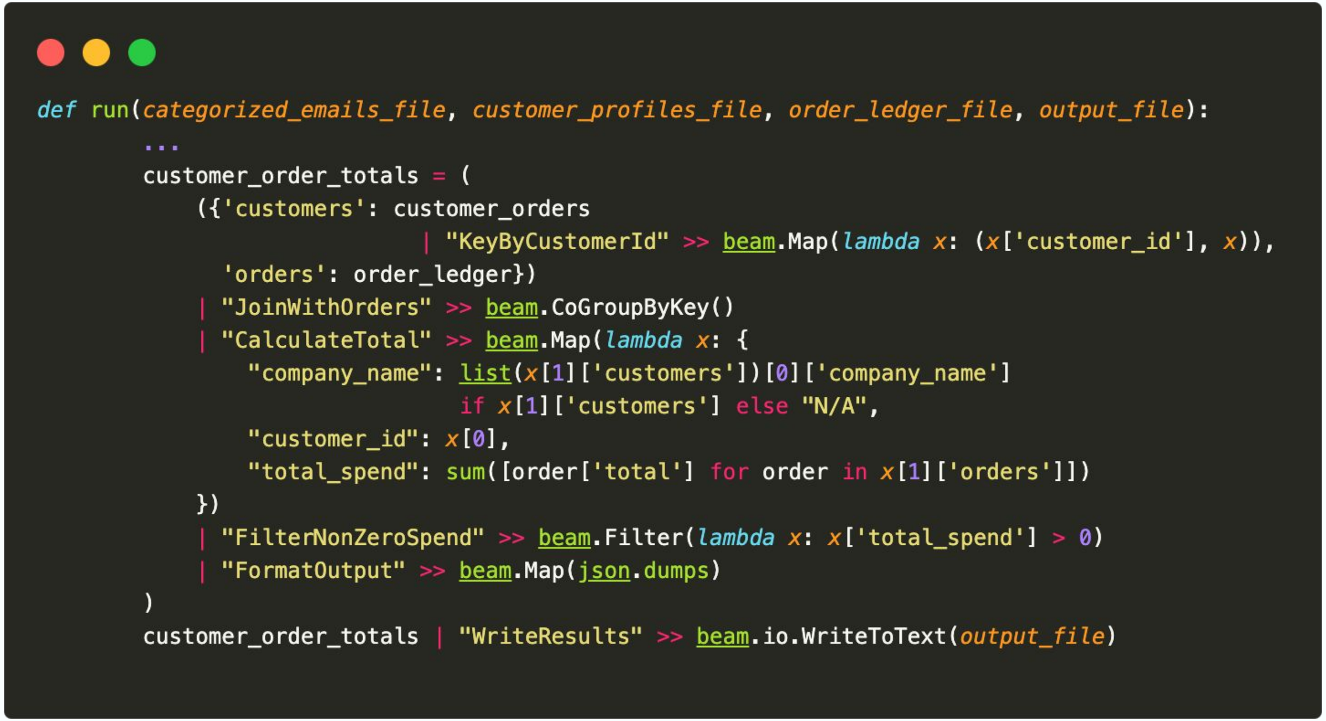
Can you combine the first and this last pipeline such that it does the analysis in the same pipeline without reading from an intermediate file?



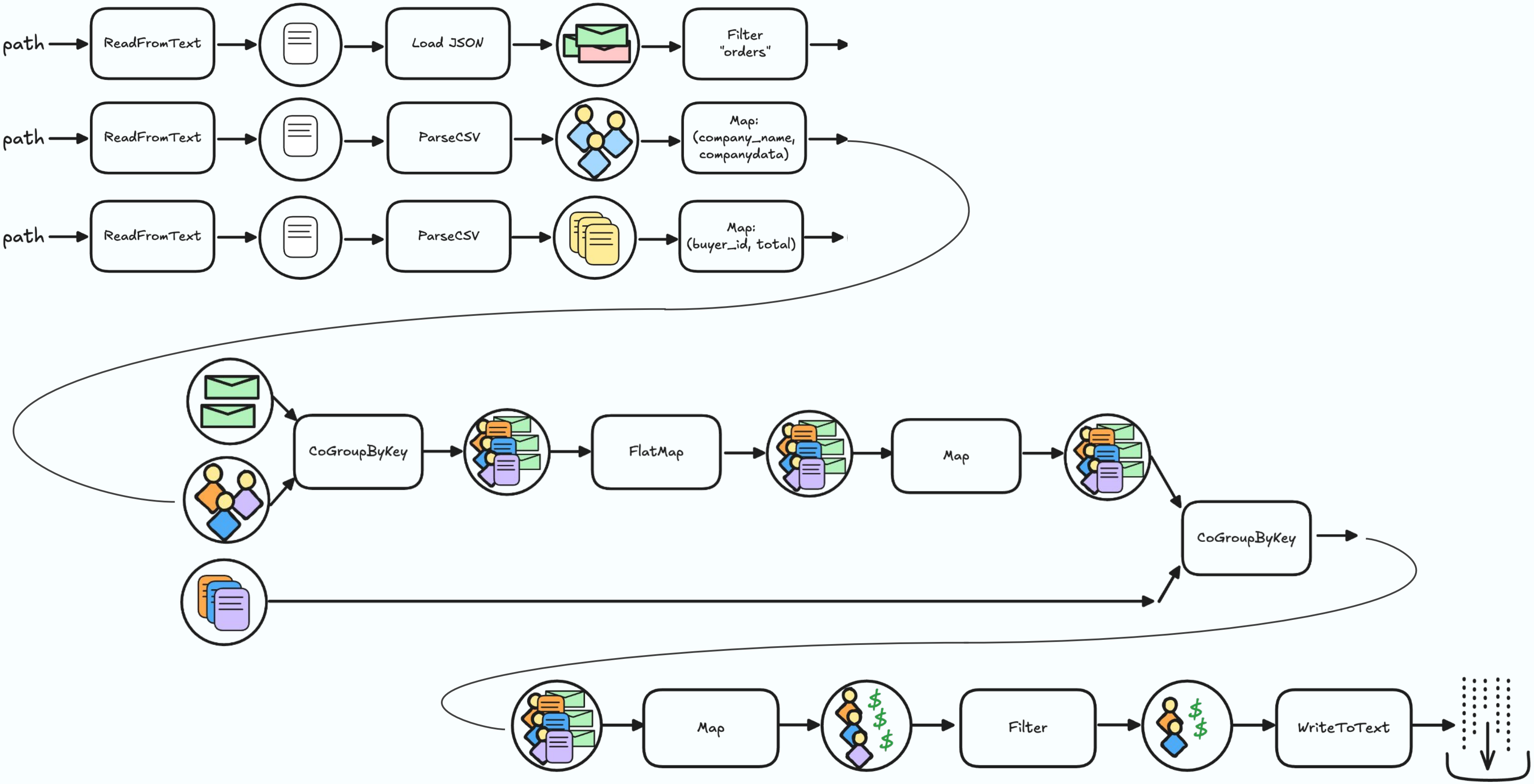
```python
class ParseCSV(beam.DoFn):
    def process(self, element):
    ...

def run(categorized_emails_file, customer_profiles_file, order_ledger_file, output_file):
    with beam.Pipeline() as pipeline:
        categorized_emails = (
            pipeline
            | "ReadCategorizedEmails" >> beam.io.ReadFromText(categorized_emails_file)
            | "ParseEmails" >> beam.Map(json.loads)
        )
        customer_profiles = (
            pipeline
            | "ReadCustomerProfiles" >> beam.io.ReadFromText(
                customer_profiles_file, skip_header_lines=1)
            | "ParseCustomerProfiles" >> beam.ParDo(ParseCSV(
                "customer_id,company_name,contact_email,business_area,office_address"))
            | "KeyByCustomerCompanyName" >> beam.Map(lambda x: (x['company_name'], x))
        )
        order_ledger = (
            pipeline
            | "ReadOrderLedger" >> beam.io.ReadFromText(
                order_ledger_file, skip_header_lines=1)
            | "ParseOrderLedger" >> beam.ParDo(ParseCSV(
                "order_id,product_id,quantity,buyer_id,product_name,order_date,price,total"))
            | "KeyByBuyerId" >> beam.Map(lambda x:
                (int(x['buyer_id']), {'total': float(x['total']) if x['total'] else 0.0}))
        )
```



```python
def run(categorized_emails_file, customer_profiles_file, order_ledger_file, output_file):
        ...
        orders = (
            categorized_emails
            | "FilterOrders" >> beam.Filter(lambda x: x.get('category') == 'order')
            | "KeyByCompanyName" >> beam.Map(lambda x: (x['sender_company_name'], x))
        )
        customer_orders = (
            ({'orders': orders, 'customers': customer_profiles})
            | "JoinWithCustomers" >> beam.CoGroupByKey()
            | "ProcessCustomerJoin" >> beam.FlatMap(lambda x: [
                {
                    "customer_id": int(customer['customer_id']),
                    "company_name": x[0],
                    "category": order['category']
                }
                for order in x[1]['orders']
                for customer in x[1]['customers']
            ])
        )
```



```
def run(categorized_emails_file, customer_profiles_file, order_ledger_file, output_file):
        ...
        customer_order_totals = (
            ({'customers': customer_orders
                             | "KeyByCustomerId" >> beam.Map(lambda x: (x['customer_id'], x)),
              'orders': order_ledger})
            | "JoinWithOrders" >> beam.CoGroupByKey()
            | "CalculateTotal" >> beam.Map(lambda x: {
                "company_name": list(x[1]['customers'])[0]['company_name']
                                if x[1]['customers'] else "N/A",
                "customer_id": x[0],
                "total_spend": sum([order['total'] for order in x[1]['orders']])
            })
            | "FilterNonZeroSpend" >> beam.Filter(lambda x: x['total_spend'] > 0)
            | "FormatOutput" >> beam.Map(json.dumps)
        )
        customer_order_totals | "WriteResults" >> beam.io.WriteToText(output_file)
```

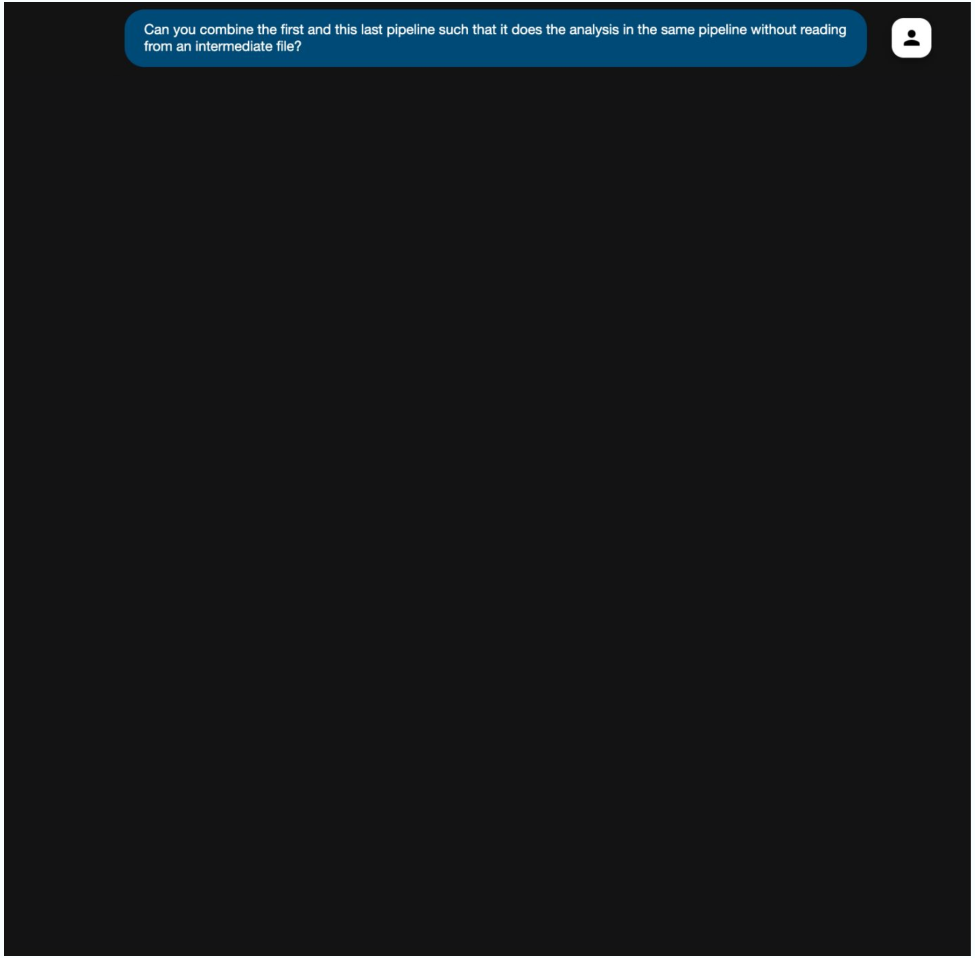

path
ReadFromText
Load JSON
Filter "orders"
path
ReadFromText
ParseCSV
Map: (company_name, companydata)
path
ReadFromText
ParseCSV
Map: (buyer_id, total)
CoGroupByKey
FlatMap
Map
CoGroupByKey
Map
Filter
WriteToText

Use case objective

1. Create a **single pipeline** that
2. **Processes** the received **emails** & ✓
3. Returns the **total spent** for *order* emails ✓

Act 3

> Can you combine the first and this last pipeline such that it does the analysis in the same pipeline without reading from an intermediate file?



Can you combine the first and this last pipeline such that it does the analysis in the same pipeline without reading from an intermediate file?

> Can you combine the first and this last pipeline such that it does the analysis in the same pipeline without reading from an intermediate file?

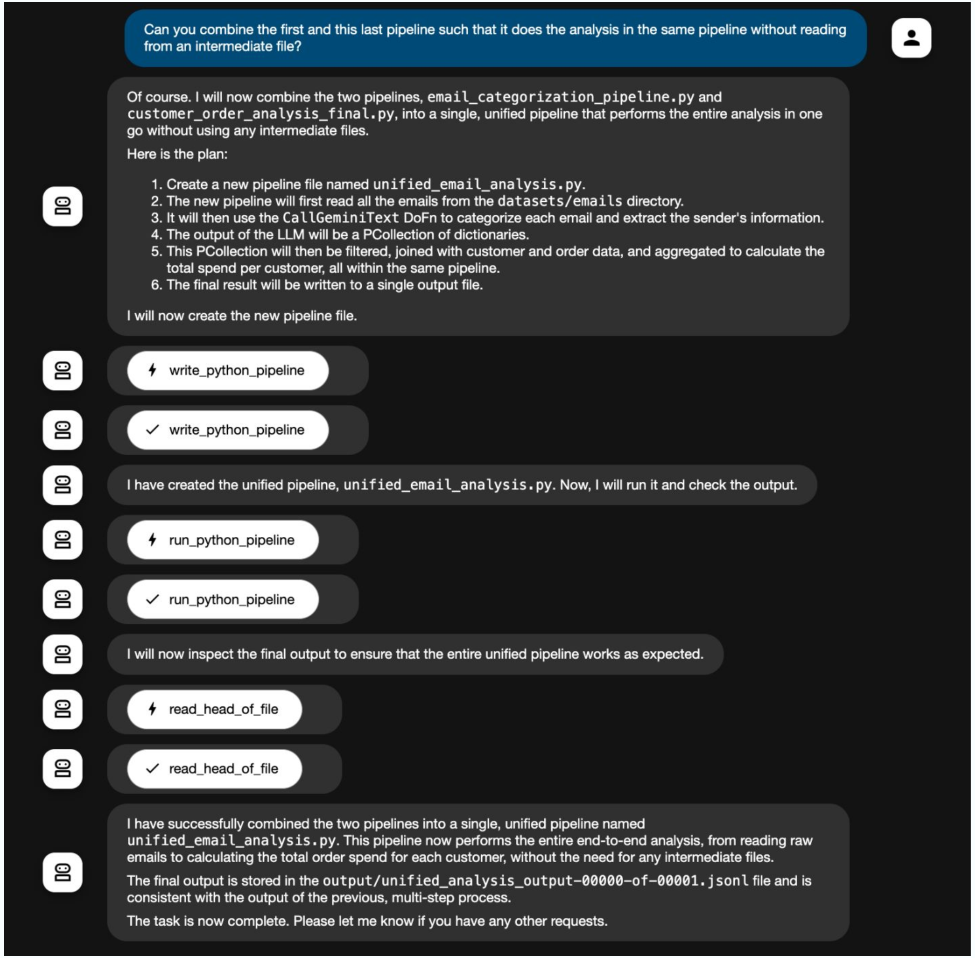
Plan

Write pipeline

Run pipeline

List results

Completed task unified_pipeline.py



Can you combine the first and this last pipeline such that it does the analysis in the same pipeline without reading from an intermediate file?

Of course. I will now combine the two pipelines, `email_categorization_pipeline.py` and `customer_order_analysis_final.py`, into a single, unified pipeline that performs the entire analysis in one go without using any intermediate files.

Here is the plan:

1. Create a new pipeline file named `unified_email_analysis.py`.
2. The new pipeline will first read all the emails from the `datasets/emails` directory.
3. It will then use the `CallGeminiText` DoFn to categorize each email and extract the sender's information.
4. The output of the LLM will be a PCollection of dictionaries.
5. This PCollection will then be filtered, joined with customer and order data, and aggregated to calculate the total spend per customer, all within the same pipeline.
6. The final result will be written to a single output file.

I will now create the new pipeline file.

⚡ write_python_pipeline

✓ write_python_pipeline

I have created the unified pipeline, `unified_email_analysis.py`. Now, I will run it and check the output.

⚡ run_python_pipeline

✓ run_python_pipeline

I will now inspect the final output to ensure that the entire unified pipeline works as expected.

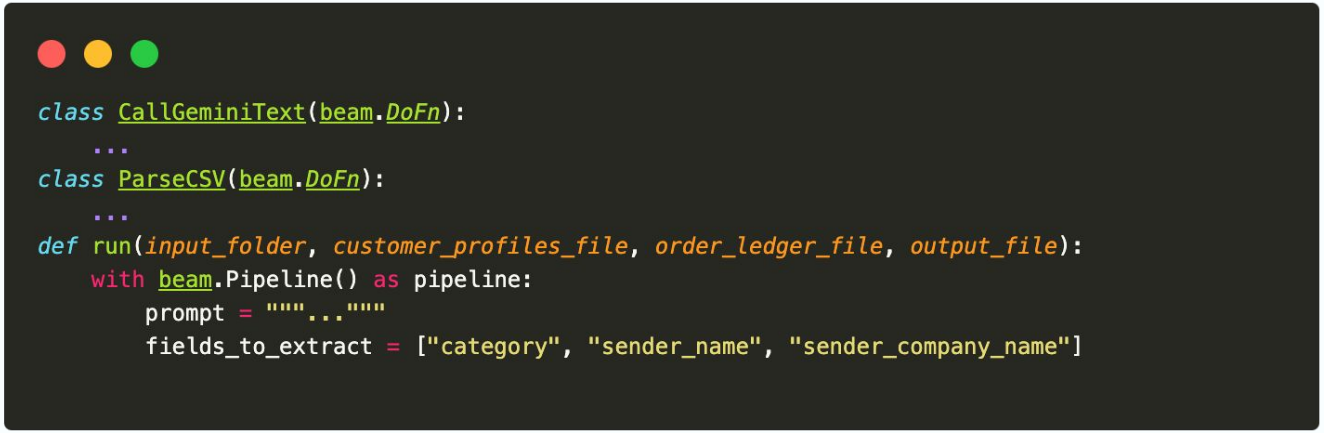
⚡ read_head_of_file

✓ read_head_of_file

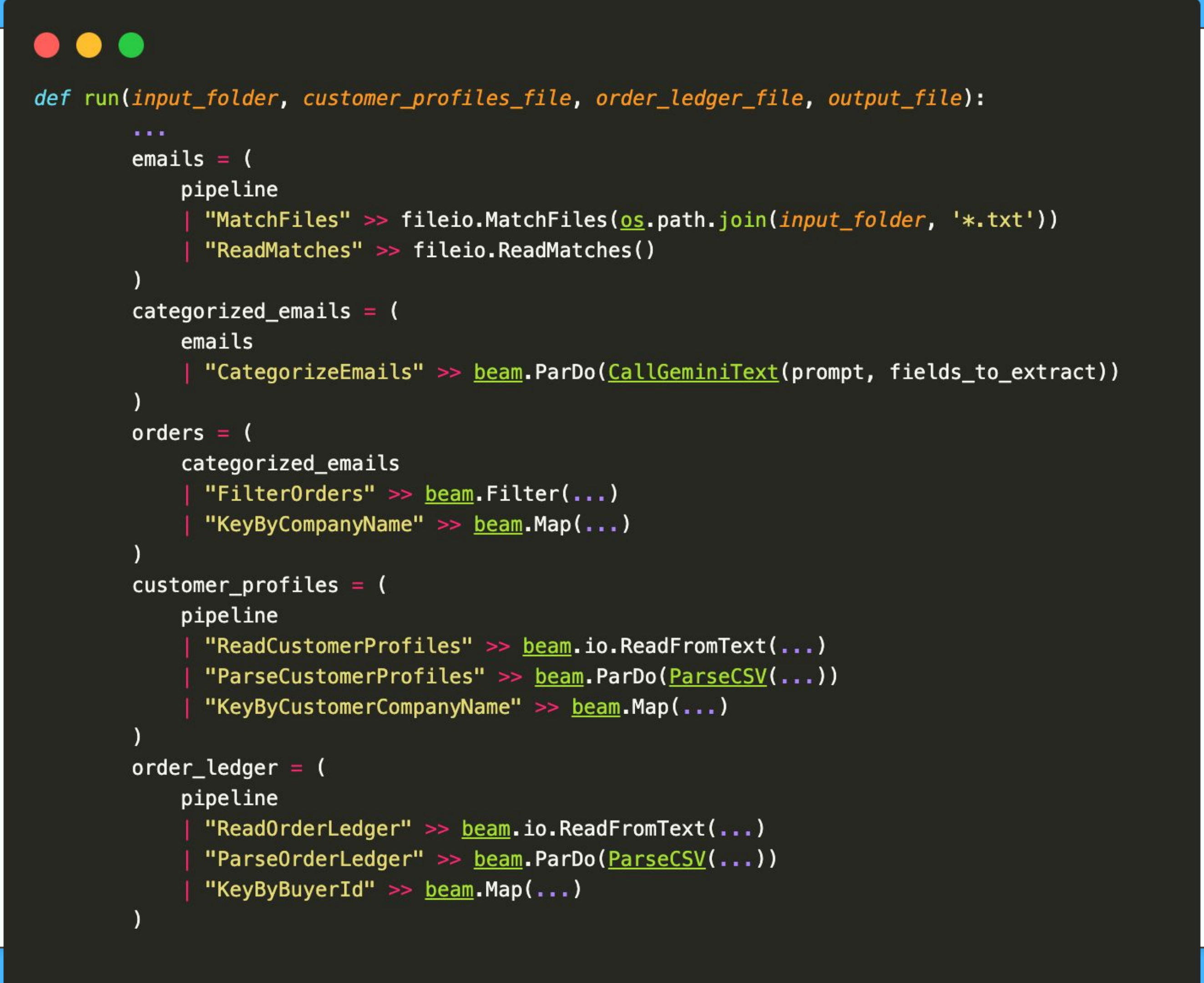
I have successfully combined the two pipelines into a single, unified pipeline named `unified_email_analysis.py`. This pipeline now performs the entire end-to-end analysis, from reading raw emails to calculating the total order spend for each customer, without the need for any intermediate files.

The final output is stored in the `output/unified_analysis_output-00000-of-00001.jsonl` file and is consistent with the output of the previous, multi-step process.

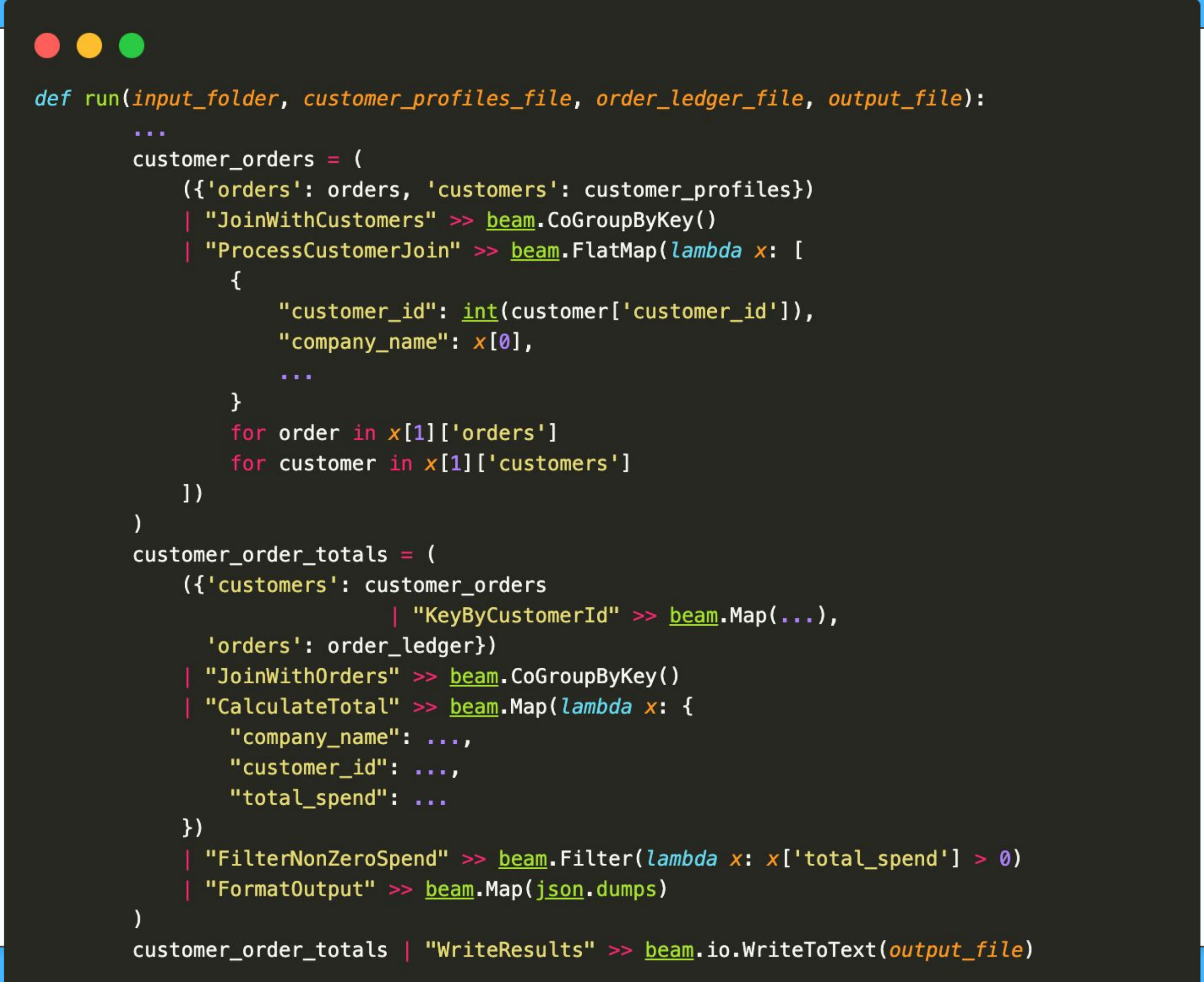
The task is now complete. Please let me know if you have any other requests.



```
class CallGeminiText(beam.DoFn):
    ...
class ParseCSV(beam.DoFn):
    ...
def run(input_folder, customer_profiles_file, order_ledger_file, output_file):
    with beam.Pipeline() as pipeline:
        prompt = """..."""
        fields_to_extract = ["category", "sender_name", "sender_company_name"]
```



```
def run(input_folder, customer_profiles_file, order_ledger_file, output_file):
        ...
        emails = (
            pipeline
            | "MatchFiles" >> fileio.MatchFiles(os.path.join(input_folder, '*.txt'))
            | "ReadMatches" >> fileio.ReadMatches()
        )
        categorized_emails = (
            emails
            | "CategorizeEmails" >> beam.ParDo(CallGeminiText(prompt, fields_to_extract))
        )
        orders = (
            categorized_emails
            | "FilterOrders" >> beam.Filter(...)
            | "KeyByCompanyName" >> beam.Map(...)
        )
        customer_profiles = (
            pipeline
            | "ReadCustomerProfiles" >> beam.io.ReadFromText(...)
            | "ParseCustomerProfiles" >> beam.ParDo(ParseCSV(...))
            | "KeyByCustomerCompanyName" >> beam.Map(...)
        )
        order_ledger = (
            pipeline
            | "ReadOrderLedger" >> beam.io.ReadFromText(...)
            | "ParseOrderLedger" >> beam.ParDo(ParseCSV(...))
            | "KeyByBuyerId" >> beam.Map(...)
        )
```



```python
def run(input_folder, customer_profiles_file, order_ledger_file, output_file):
        ...
        customer_orders = (
            ({'orders': orders, 'customers': customer_profiles})
            | "JoinWithCustomers" >> beam.CoGroupByKey()
            | "ProcessCustomerJoin" >> beam.FlatMap(lambda x: [
                {
                    "customer_id": int(customer['customer_id']),
                    "company_name": x[0],
                    ...
                }
                for order in x[1]['orders']
                for customer in x[1]['customers']
            ])
        )
        customer_order_totals = (
            ({'customers': customer_orders
                            | "KeyByCustomerId" >> beam.Map(...),
              'orders': order_ledger})
            | "JoinWithOrders" >> beam.CoGroupByKey()
            | "CalculateTotal" >> beam.Map(lambda x: {
                "company_name": ...,
                "customer_id": ...,
                "total_spend": ...
            })
            | "FilterNonZeroSpend" >> beam.Filter(lambda x: x['total_spend'] > 0)
            | "FormatOutput" >> beam.Map(json.dumps)
        )
        customer_order_totals | "WriteResults" >> beam.io.WriteToText(output_file)
```

Use case objective

1. Create a **single pipeline** that ✓
2. **Processes** the received **emails** & ✓
3. Returns the **total spent** for *order* emails ✓

The Good & the Bad

Agent is able to solve the use case!
...but not always in one go.

Beam tools enable data processing at scale with robust built-ins.

Examples transforms help. Instruct agent to use them where possible.

Defining tools correctly is crucial & requires iteration.

Python pipelines work better than YAML pipelines.

Data engineers are not obsolete:
Validation is important & works best step-by-step.

Taking it Further

More tools

Transform library

Search documentation tool

Tools combinations

Cloud Runners & Streaming Pipelines

Defining tools is the new prompt engineering

Standardize tools using Model Context Protocol (MCP)

Evaluation

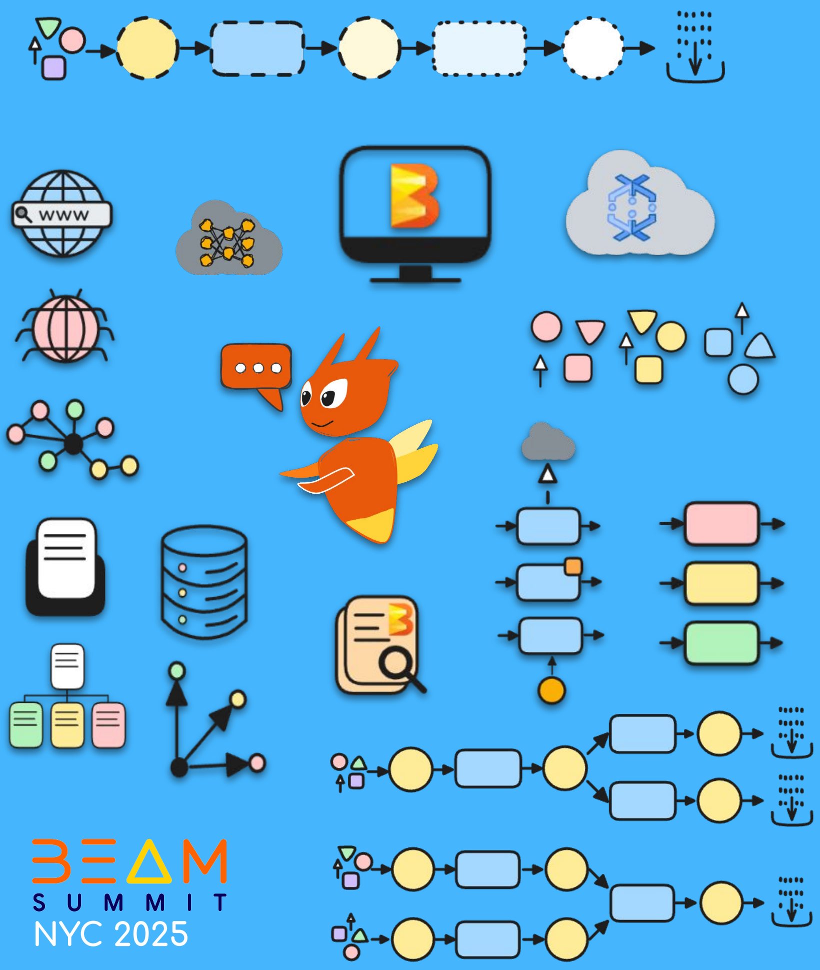
Proactivity reporting

Defined Beam Schemas

Food for Thought

How to make a framework successful in the era AI-assisted coding?

Summary



Superpowered agent with beam tools to solve the use case.

Tools enable LLMs to act.

With reasoning & planning they can solve complex tasks.

Defining tools is the new *prompt engineering*.

Human-in-the-loop & step-by-step

BEAM SUMMIT
NYC 2025

Konstantin Buschmeier
Jasper Van den Bossche & Karel Haerens

QUESTIONS?

ML6 is hiring

👉 https://jobs.ml6.eu/

BEAM SUMMIT
NYC 2025