bauplan

Zero-copy, Scale-up FaaS for Data Pipelines

Jacopo Tagliabue, Tyler Caraza-Harter, Ciro Greco
WoSC 10 @ Middleware 2024, *Hong Kong*

Backed by

innovation endeavors

South Park Commons

And by founders and executives at

aws

docker

redis

VOLTRON DATA

CLOUDERA

bauplan = (specialized FaaS runtime + data DAG abstractions)

(as we could not add a runtime to Airflow, nor a data lake to Lambda)

Today is about Bauplan!

01 | Data processing as functional DAGs

02 | Bauplan overview: architecture and abstractions

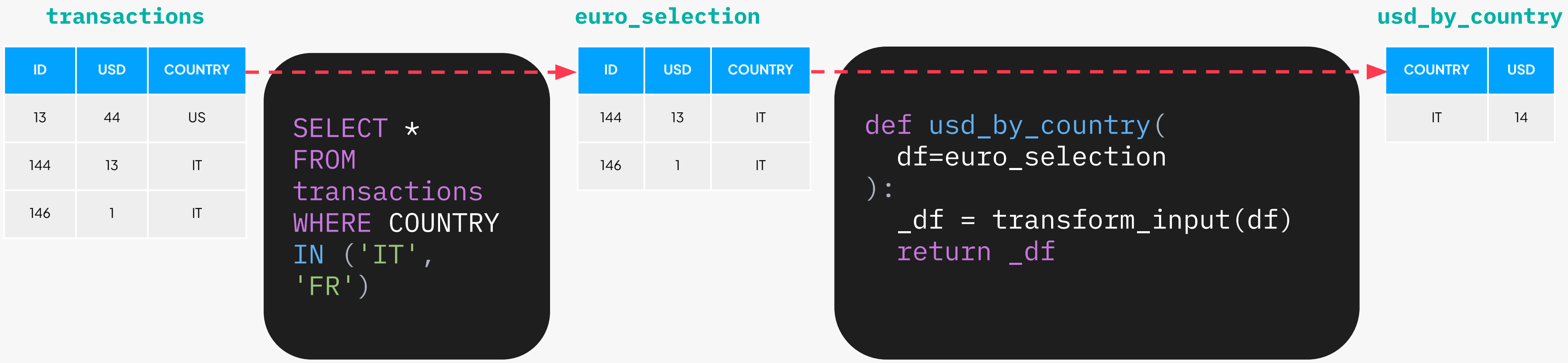
03 | Anatomy of a run

04 | Conclusion and future work

⚡ data processing as functional DAGs ⚡

Data workloads, DAGs, Faas

- Pre-processing is a core ingredient for successful data-driven use cases (analytics, BI, AI, etc.)
- Data pre-processing is done with **pipelines**, *DAGs* of *functions* wrangling raw data into cleaned datasets.



transactions

ID	USD	COUNTRY
13	44	US
144	13	IT
146	1	IT

```
SELECT *
FROM
transactions
WHERE COUNTRY
IN ('IT',
'FR')
```

euro_selection

ID	USD	COUNTRY
144	13	IT
146	1	IT

```
def usd_by_country(
  df=euro_selection
):
  _df = transform_input(df)
  return _df
```

usd_by_country

COUNTRY	USD
IT	14

Q: can we take an existing FaaS and make it data DAG aware?

Data workloads, DAGs, Faas

Data workloads are peculiar (vis-a-vis typical FaaS use cases)

Scaling up

Industry traces *p99.9* for memory is 50-200 GB! [1]

Large intermediate I/O

Functions move >100M rows multiple times in a complex DAG! [2]

Fast feedback loop

Data projects are exploratory: rapidly iterating over hypotheses is key! [3]

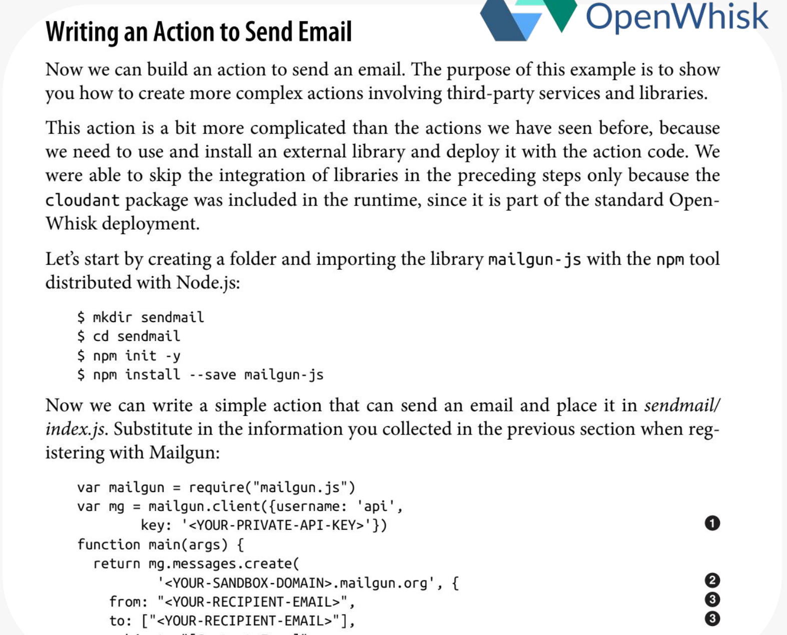
[1] van Renen et al 2024. Why TPC is not enough: An analysis of the Amazon Redshift fleet.
[2] https://github.com/jacopotagliabue/paas-data-ingestion
[3] Xin et al 2018. How Developers Iterate on Machine Learning Workflows

Data workloads, DAGs, Faas

A: “NO!”

| Small memory footprint
| Small I/O size
| Built for deployment, not iteration speed

Platform	Memory	I/O payload	Timeout
Lambda	10 GB	6 MB	900s
OpenWhisk	2 GB	1 MB	300s



Writing an Action to Send Email

Now we can build an action to send an email. The purpose of this example is to show you how to create more complex actions involving third-party services and libraries.

This action is a bit more complicated than the actions we have seen before, because we need to use and install an external library and deploy it with the action code. We were able to skip the integration of libraries in the preceding steps only because the `cloudant` package was included in the runtime, since it is part of the standard OpenWhisk deployment.

Let’s start by creating a folder and importing the library `mailgun-js` with the `npm` tool distributed with Node.js:

```
$ mkdir sendmail
$ cd sendmail
$ npm init -y
$ npm install --save mailgun-js
```

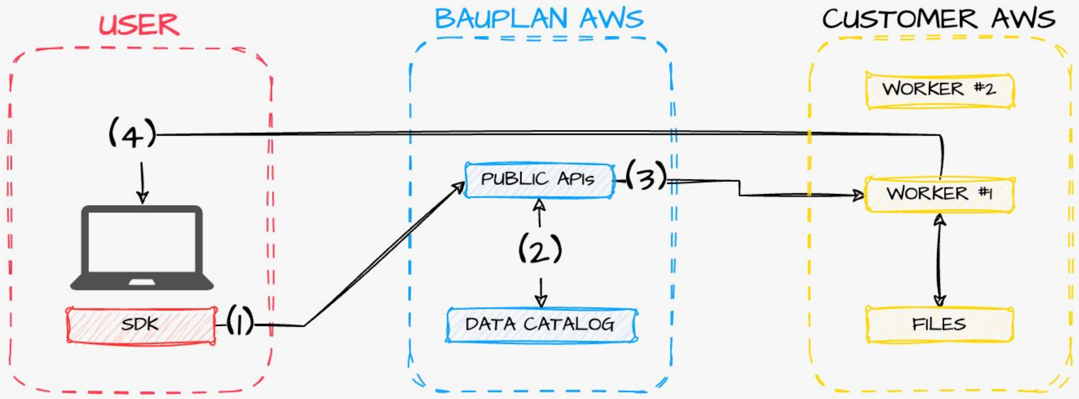
Now we can write a simple action that can send an email and place it in *sendmail/index.js*. Substitute in the information you collected in the previous section when registering with Mailgun:

```
var mailgun = require("mailgun.js")
var mg = mailgun.client({username: 'api',
        key: '<YOUR-PRIVATE-API-KEY>'})                    ❶
function main(args) {
  return mg.messages.create(
         '<YOUR-SANDBOX-DOMAIN>.mailgun.org', {            ❷
    from: "<YOUR-RECIPIENT-EMAIL>",                        ❸
    to: ["<YOUR-RECIPIENT-EMAIL>"],                        ❸
```

bauplan overview

Not a FaaS, not a db, but a secret third thing

- **CLI / SDK** to launch DAGs from a laptop (`pip install bauplan`)
- **Control plane (CP)** performs authentication and planning (metadata only).
- **Data Plane (DP)** executes the transformations in customer EC2s.



Programming model: Lambda and Airflow

dockerfile

```
FROM public.ecr.aws/lambda/python:3.11
RUN pip install -r requirements.txt
COPY app.py ${LAMBDA_TASK_ROOT}
CMD [ "app.lambda_handler" ]
```

handler.py

```
def lambda_handler(event, context):
       input_s3 = event.get('s3_path')
# transform here and save back to s3
       my_data = transform(input_s3)
       save_to_s3(my_data)

return { "status_code": 200 }
```

dag.py [1]

```
def preprocess(
s3_in_url, s3_out_bucket, s3_out_prefix
):
# Transform and save the result in S3
return "SUCCESS"

# register the function in the overall DAG
preprocess_task = PythonOperator(
task_id="preprocessing",
dag=dag
)
```

[1] https://github.com/aws-samples/sagemaker-ml-workflow-with-apache-airflow

Programming model: Lambda and Airflow

"Infra-as-code" but manual

dockerfile

```
FROM public.ecr.aws/lambda/python:3.11
RUN pip install -r requirements.txt
COPY app.py ${LAMBDA_TASK_ROOT}
CMD [ "app.lambda_handler" ]
```

handler.py

```
def lambda_handler(event, context):
       input_s3 = event.get('s3_path')
# transform here and save back to s3
       my_data = transform(input_s3)
       save_to_s3(my_data)

return { "status_code": 200 }
```

Apache Airflow

dag.py [1]

```
def preprocess(
s3_in_url, s3_out_bucket, s3_out_prefix
):
# Transform and save the result in S3
return "SUCCESS"

# register the function in the overall DAG
preprocess_task = PythonOperator(
task_id="preprocessing",
dag=dag
)
```

[1] https://github.com/aws-samples/sagemaker-ml-workflow-with-apache-airflow

Programming model: Lambda and Airflow

dockerfile

```
FROM public.ecr.aws/lambda/python:3.11
RUN pip install -r requirements.txt
COPY app.py ${LAMBDA_TASK_ROOT}
CMD [ "app.lambda_handler" ]
```

handler.py

```
def lambda_handler(event, context):
       input_s3 = event.get('s3_path')
# transform here and save back to s3
       my_data = transform(input_s3)
       save_to_s3(my_data)

return { "status_code": 200 }
```

Generic inputs

dag.py [1]

```
def preprocess(
s3_in_url, s3_out_bucket, s3_out_prefix
):
# Transform and save the result in S3
return "SUCCESS"

# register the function in the overall DAG
preprocess_task = PythonOperator(
task_id="preprocessing",
dag=dag
)
```

[1] https://github.com/aws-samples/sagemaker-ml-workflow-with-apache-airflow

Programming model: Lambda and Airflow

dockerfile

```
FROM public.ecr.aws/lambda/python:3.11
RUN pip install -r requirements.txt
COPY app.py ${LAMBDA_TASK_ROOT}
CMD [ "app.lambda_handler" ]
```

handler.py

```
def lambda_handler(event, context):
       input_s3 = event.get('s3_path')
# transform here and save back to s3
       my_data = transform(input_s3)
       save_to_s3(my_data)

return { "status_code": 200 }
```

I/O as side-effect

Apache Airflow

dag.py [1]

```
def preprocess(
s3_in_url, s3_out_bucket, s3_out_prefix
):
# Transform and save the result in S3
return "SUCCESS"

# register the function in the overall DAG
preprocess_task = PythonOperator(
task_id="preprocessing",
dag=dag
)
```

[1] https://github.com/aws-samples/sagemaker-ml-workflow-with-apache-airflow

Q: can we design better FaaS abstractions for data DAGs?

Programming model: bauplan

bau.py

```python
@bauplan.model()
@bauplan.python(
   "3.11",
   pip={"pandas": "2.2"}
)
def euro_selection(
    data=bauplan.Model(
       "transactions",
       columns=["id", "usd", "country"],
       filter="eventTime BETWEEN 2023-01-01 AND
2023-02-01"
   )
):
    # filtering here
    # return a dataframe
    return _df
```

bau.py

```python
@bauplan.model(materialize=True)
@bauplan.python(
    "3.10",
    pip={"pandas": "1.5.3"}
)
def usd_by_country(
    data=bauplan.Model("euro_selection")
):
    # aggregation here
    # return a dataframe
    return _df
```

Programming model: bauplan

Infra-as-code

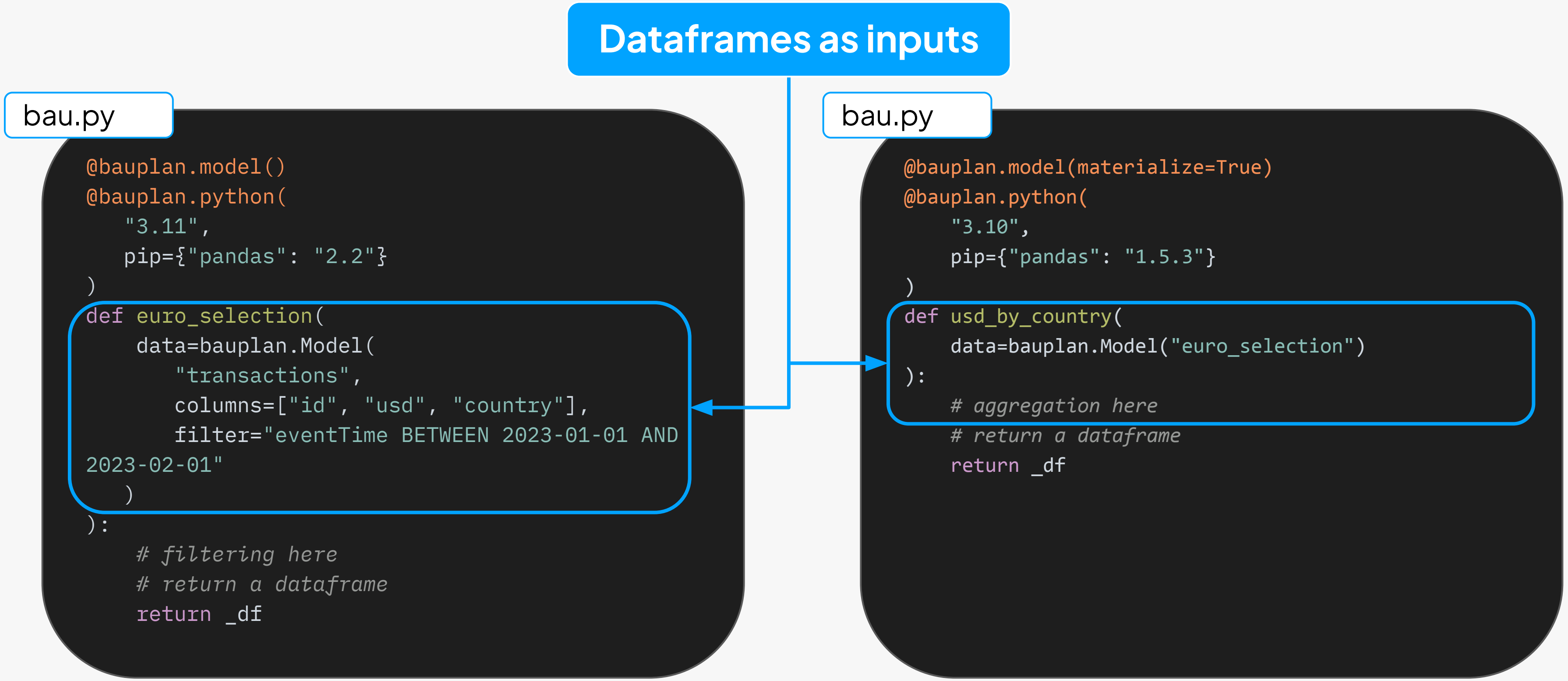
bau.py

```python
@bauplan.model()
@bauplan.python(
   "3.11",
   pip={"pandas": "2.2"}
)
def euro_selection(
    data=bauplan.Model(
       "transactions",
       columns=["id", "usd", "country"],
       filter="eventTime BETWEEN 2023-01-01 AND
2023-02-01"
   )
):
    # filtering here
    # return a dataframe
    return _df
```

bau.py

```python
@bauplan.model(materialize=True)
@bauplan.python(
    "3.10",
    pip={"pandas": "1.5.3"}
)
def usd_by_country(
    data=bauplan.Model("euro_selection")
):
    # aggregation here
    # return a dataframe
    return _df
```

Programming model: bauplan

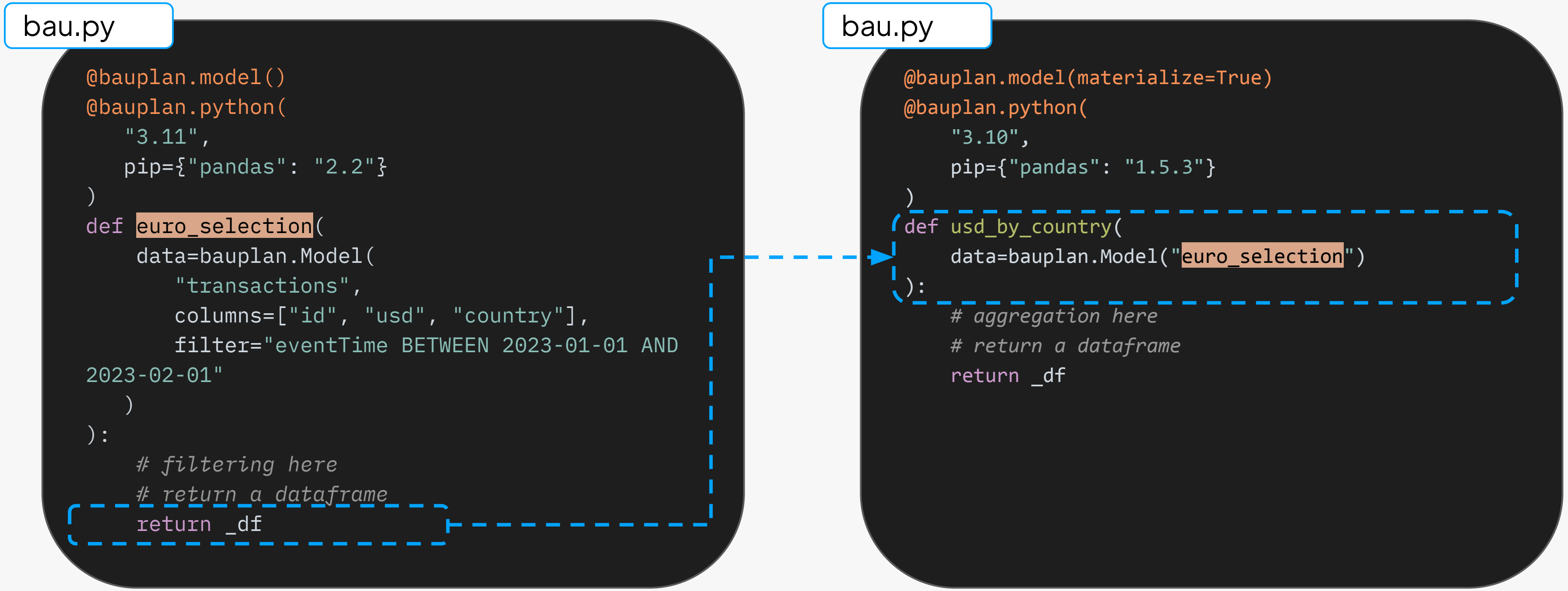


```
@bauplan.model()
@bauplan.python(
   "3.11",
   pip={"pandas": "2.2"}
)
def euro_selection(
    data=bauplan.Model(
       "transactions",
       columns=["id", "usd", "country"],
       filter="eventTime BETWEEN 2023-01-01 AND
2023-02-01"
   )
):
    # filtering here
    # return a dataframe
    return _df
```

```
@bauplan.model(materialize=True)
@bauplan.python(
    "3.10",
    pip={"pandas": "1.5.3"}
)
def usd_by_country(
    data=bauplan.Model("euro_selection")
):
    # aggregation here
    # return a dataframe
    return _df
```

Programming model: bauplan

I/O chaining



bau.py

```
@bauplan.model()
@bauplan.python(
    "3.11",
    pip={"pandas": "2.2"}
)
def euro_selection(
     data=bauplan.Model(
        "transactions",
        columns=["id", "usd", "country"],
        filter="eventTime BETWEEN 2023-01-01 AND
2023-02-01"
    )
):
     # filtering here
     # return a dataframe
     return _df
```

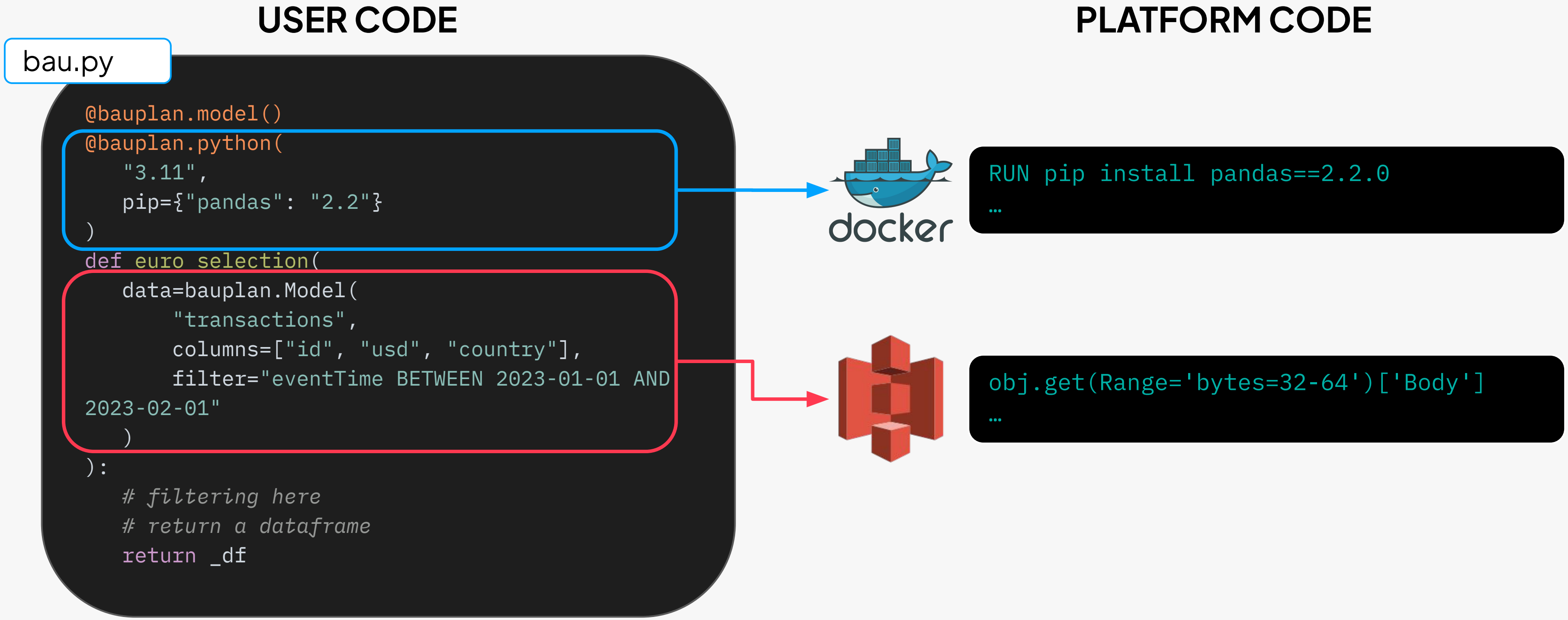
bau.py

```
@bauplan.model(materialize=True)
@bauplan.python(
     "3.10",
     pip={"pandas": "1.5.3"}
)
def usd_by_country(
     data=bauplan.Model("euro_selection")
):
     # aggregation here
     # return a dataframe
     return _df
```

🫀 anatomy of a run 🫀

$$\text{bauplan run} = \left(\begin{array}{c} \text{plan} \\ + \\ \text{environment} \\ + \\ \text{data movement} \end{array} \right)$$

Planning



bau.py

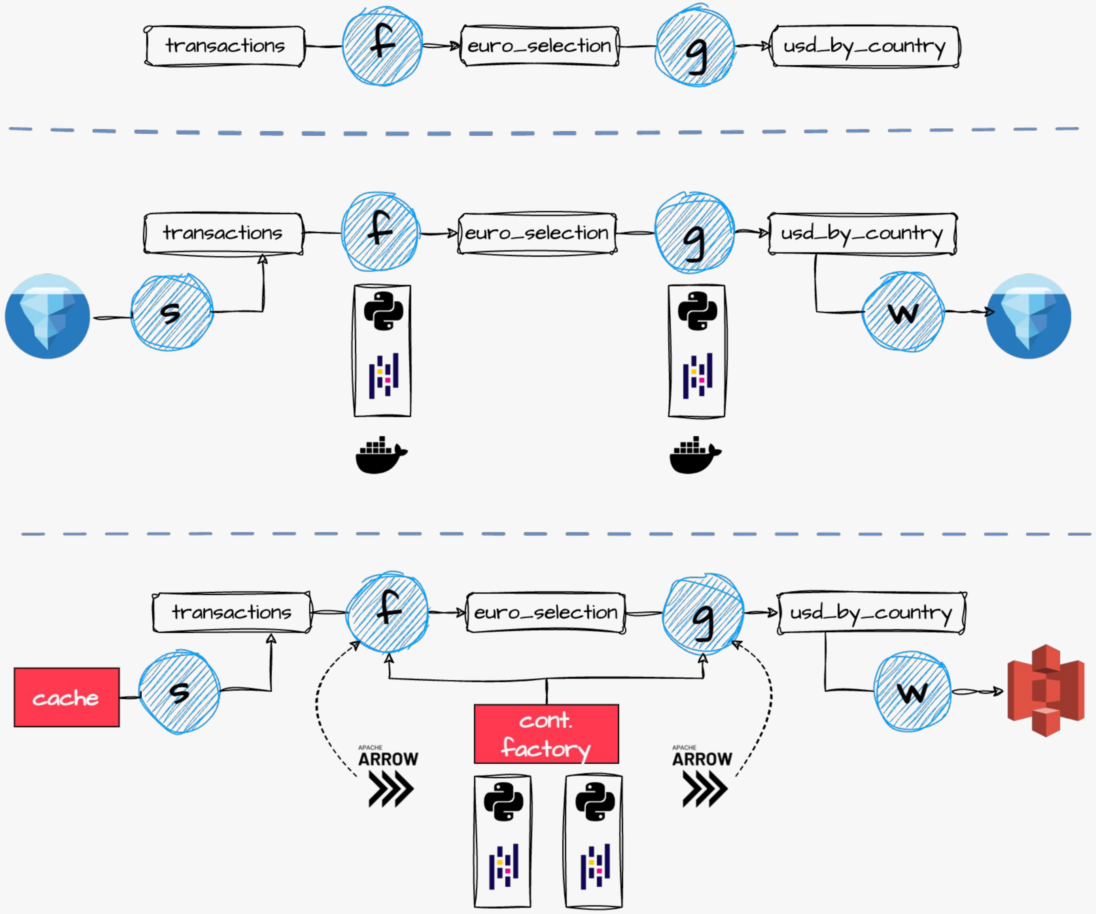
```python
@bauplan.model()
@bauplan.python(
    "3.11",
    pip={"pandas": "2.2"}
)
def euro_selection(
    data=bauplan.Model(
        "transactions",
        columns=["id", "usd", "country"],
        filter="eventTime BETWEEN 2023-01-01 AND
2023-02-01"
    )
):
    # filtering here
    # return a dataframe
    return _df
```

```
RUN pip install pandas==2.2.0
…
```

```
obj.get(Range='bytes=32-64')['Body']
…
```

Planning

- **Logical view.** What are the dependencies as expressed in USER CODE?
- **Physical view.** What are the Docker and Iceberg instructions to finalize PLATFORM CODE?
- **Worker view.** What artifacts and packages we have already / what do we need to fetch?



Building function environments

- **Assemble, don’t build.** Functions run in dockerized environments which mount packages from a local cache. [1]
- No Docker on the client, no PyPI bandwidth bottlenecks, no ECR update: adding a package is **15× faster than AWS Lambda!**

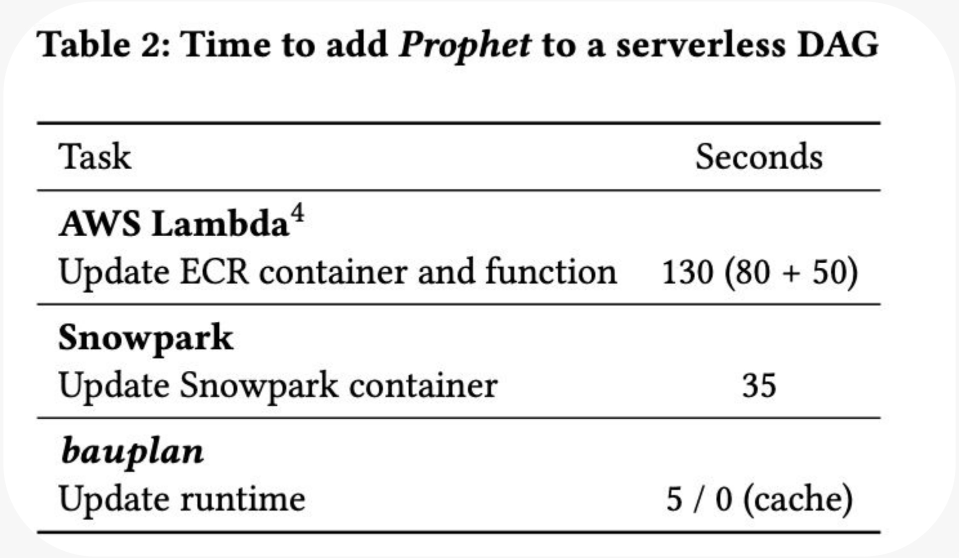


Table 2: Time to add *Prophet* to a serverless DAG

Task	Seconds
AWS Lambda[4]	
Update ECR container and function	130 (80 + 50)
Snowpark	
Update Snowpark container	35
bauplan	
Update runtime	5 / 0 (cache)

[1] Hendrickson et al. 2016, Serverless Computation with OpenLambda

I/O and data movement

Arrow everywhere. Dataframe layout in-memory and over the wire is Apache Arrow.

“Zero-copy”:

- Within a worker, tables can be zero-copy shared between functions (100× faster than S3)
- Across workers, an Arrow stream is as fast as local parquet files (no-serialization cost)
- Arrow fragments in a worker cache can be combined in a “view” (saves 30% S3 reads on TCP-H 100)

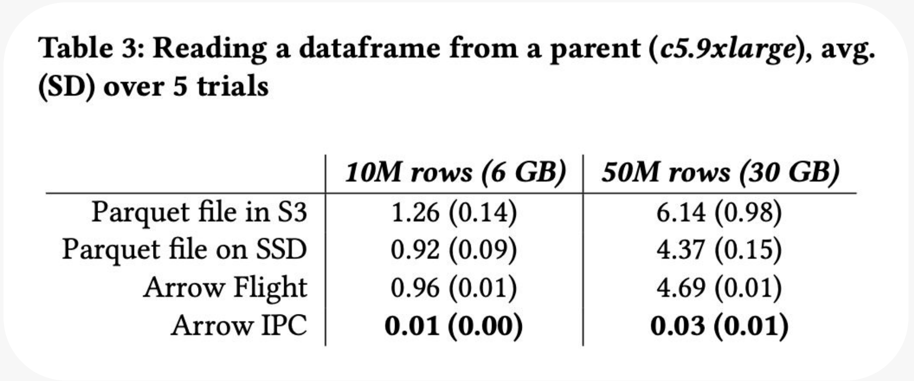
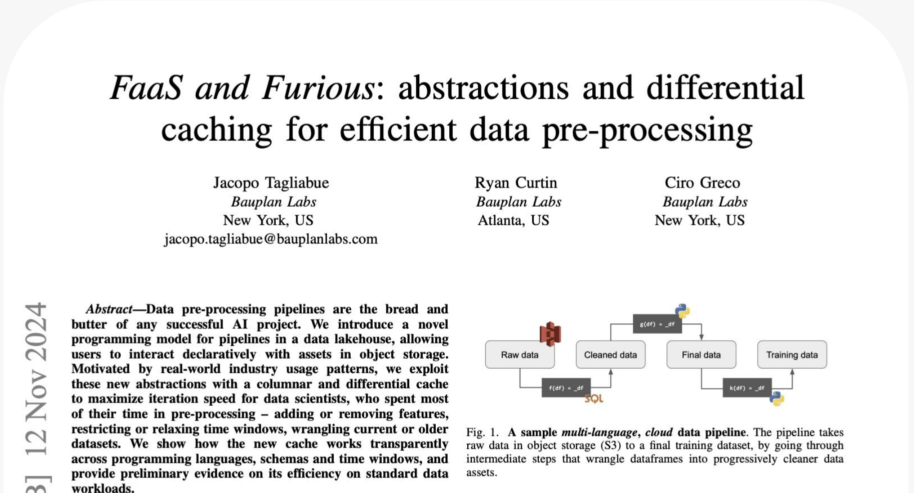


Table 3: Reading a dataframe from a parent (*c5.9xlarge*), avg. (SD) over 5 trials

	10M rows (6 GB)	*50M rows (30 GB)*
Parquet file in S3	1.26 (0.14)	6.14 (0.98)
Parquet file on SSD	0.92 (0.09)	4.37 (0.15)
Arrow Flight	0.96 (0.01)	4.69 (0.01)
Arrow IPC	**0.01 (0.00)**	**0.03 (0.01)**



FaaS and Furious: abstractions and differential caching for efficient data pre-processing

Jacopo Tagliabue
Bauplan Labs
New York, US
jacopo.tagliabue@bauplanlabs.com

Ryan Curtin
Bauplan Labs
Atlanta, US

Ciro Greco
Bauplan Labs
New York, US

3] 12 Nov 2024

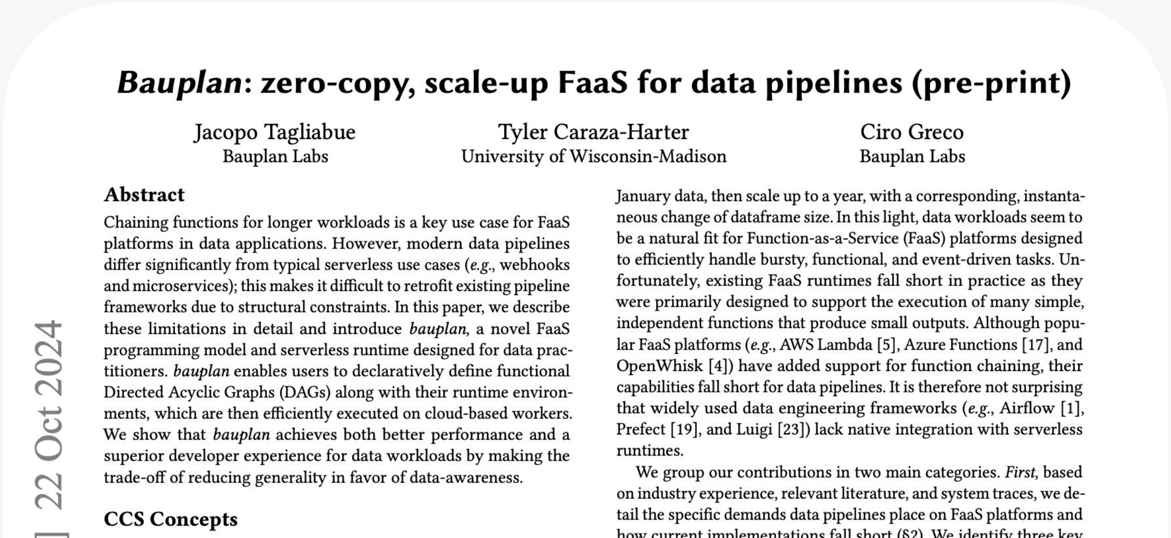
***Abstract*—Data pre-processing pipelines are the bread and butter of any successful AI project. We introduce a novel programming model for pipelines in a data lakehouse, allowing users to interact declaratively with assets in object storage. Motivated by real-world industry usage patterns, we exploit these new abstractions with a columnar and differential cache to maximize iteration speed for data scientists, who spent most of their time in pre-processing – adding or removing features, restricting or relaxing time windows, wrangling current or older datasets. We show how the new cache works transparently across programming languages, schemas and time windows, and provide preliminary evidence on its efficiency on standard data workloads.**

Fig. 1. **A sample *multi-language*, *cloud* data pipeline**. The pipeline takes raw data in object storage (S3) to a final training dataset, by going through intermediate steps that wrangle dataframes into progressively cleaner data assets.

🏁 conclusion 🏁

Conclusion and future work

- TL;DR: by co-designing *abstractions* (unique signature) and *runtime* features (Arrow layout, Python-SQL only) for data workloads, *bauplan* provides better performance and FaaS ergonomics to data practitioners.
- We look forward to sharing with the community further results in memory optimization and function scheduling in a single-tenant environment.



Bauplan: zero-copy, scale-up FaaS for data pipelines (pre-print)

Jacopo Tagliabue, Bauplan Labs
Tyler Caraza-Harter, University of Wisconsin-Madison
Ciro Greco, Bauplan Labs

Abstract

Chaining functions for longer workloads is a key use case for FaaS platforms in data applications. However, modern data pipelines differ significantly from typical serverless use cases (*e.g.*, webhooks and microservices); this makes it difficult to retrofit existing pipeline frameworks due to structural constraints. In this paper, we describe these limitations in detail and introduce *bauplan*, a novel FaaS programming model and serverless runtime designed for data practitioners. *bauplan* enables users to declaratively define functional Directed Acyclic Graphs (DAGs) along with their runtime environments, which are then efficiently executed on cloud-based workers. We show that *bauplan* achieves both better performance and a superior developer experience for data workloads by making the trade-off of reducing generality in favor of data-awareness.

CCS Concepts

January data, then scale up to a year, with a corresponding, instantaneous change of dataframe size. In this light, data workloads seem to be a natural fit for Function-as-a-Service (FaaS) platforms designed to efficiently handle bursty, functional, and event-driven tasks. Unfortunately, existing FaaS runtimes fall short in practice as they were primarily designed to support the execution of many simple, independent functions that produce small outputs. Although popular FaaS platforms (*e.g.*, AWS Lambda [5], Azure Functions [17], and OpenWhisk [4]) have added support for function chaining, their capabilities fall short for data pipelines. It is therefore not surprising that widely used data engineering frameworks (*e.g.*, Airflow [1], Prefect [19], and Luigi [23]) lack native integration with serverless runtimes.

We group our contributions in two main categories. *First*, based on industry experience, relevant literature, and system traces, we detail the specific demands data pipelines place on FaaS platforms and how current implementations fall short (§2). We identify three key bauplan