

Speedrunning the Lakehouse

Shipping a FaaS that looks like a database (and vice versa)

South Bay Systems
07.22.25



Ciao, I'm Jacopo!

| Co-founder and CTO at **Bauplan**.

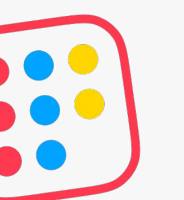
Backed by IE, SPC, Wes McKinney, Spencer Kimball, Chris Re et al.

| Started the “Reasonable Scale” movement.

Co-founder at Tooso and lead AI at TSX:CVO after the acquisition.

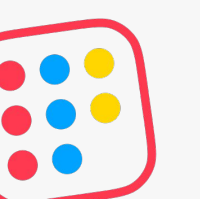
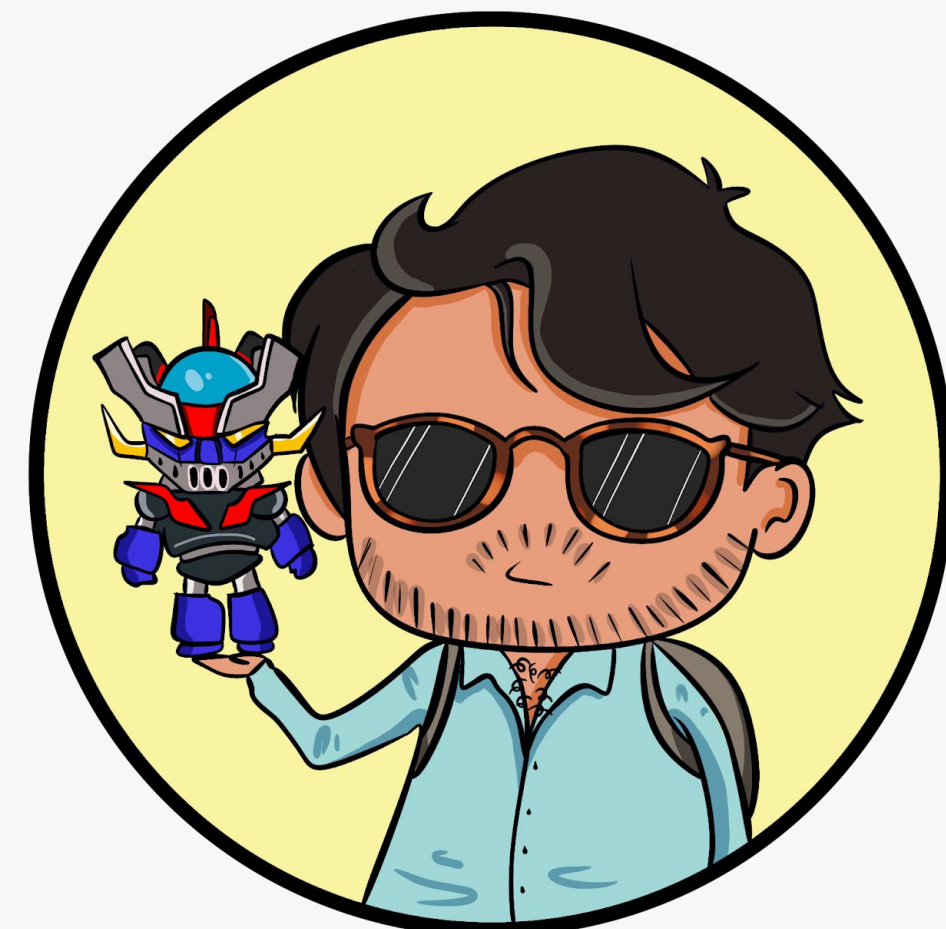
| 10 years up and down the stack in R&D, product, open source

ICML, KDD, SIGMOD, VLDB, NAACL, SIGIR et al., >2k stars, >10M+ downloads.



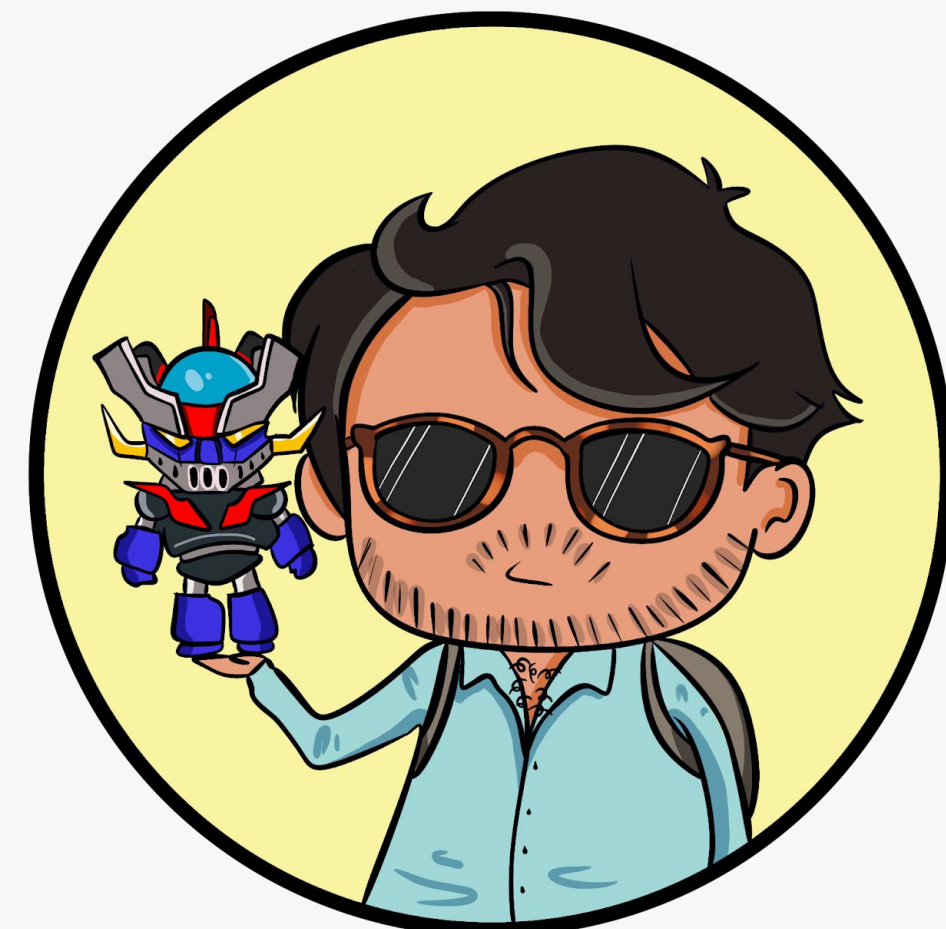
It takes a (distributed) village

| While I am the only speaker today, Matt, Ciro, Luca, Nate, Vlad (and others, unfortunately without a chibi) share with me the credit for whatever value these ideas may have.



It takes a (distributed) village

| Obviously, all the remaining mistakes are theirs 😊





speed·run

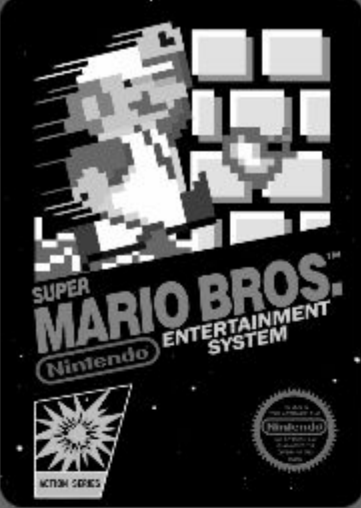
/ˈspēdˌrən/

verb

gerund or present participle: **speedrunning**

complete (a video game, or level of a game) as fast as possible.

"I used to be able to speedrun this game in less than 20 minutes"



Super Mario Bros. (1985)

Super Mario Series

NES SNES WiiVC +14

Category extensions Discord

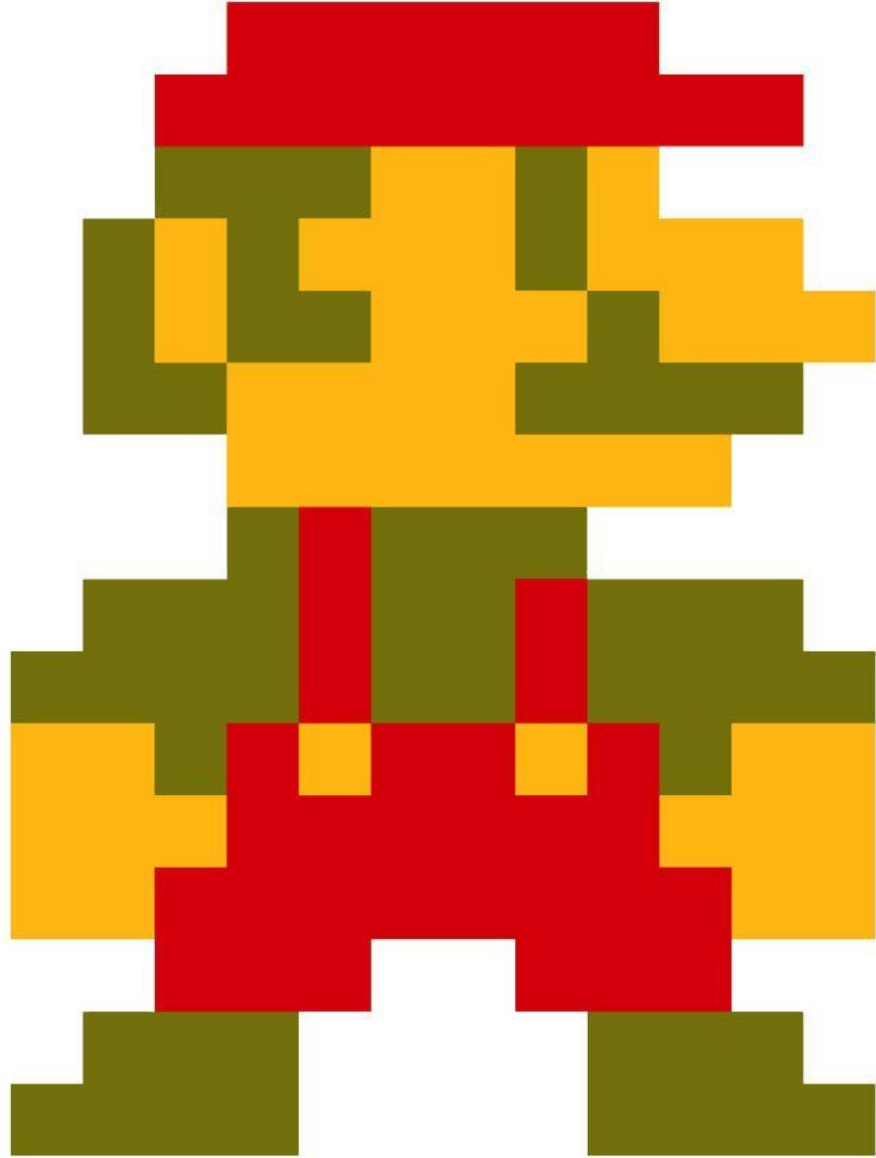
Leaderboards News 8 Guides 42 Resources 44

Any% Warpless Any% All-Stars Warpless All-Stars

Version NTSC PAL

Filters Show rules

#	Player	Time
★	Niftski	4m 54s 565ms
	averge11	4m 54s 748ms
	Tree_05	4m 54s 864ms



Time

4m 54s 565ms

4m 54s 748ms

4m 54s 864ms

What is a Lakehouse?

- | Data on S3
- | Multi-use case
- | Multi-language (SQL, Python)

Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics

Michael Armbrust¹, Ali Ghodsi^{1,2}, Reynold Xin¹, Matei Zaharia^{1,3}

¹Databricks, ²UC Berkeley, ³Stanford University

Abstract

This paper argues that the data warehouse architecture as we know it today will wither in the coming years and be replaced by a new architectural pattern, the Lakehouse, which will (i) be based on open direct-access data formats, such as Apache Parquet, (ii) have first-class support for machine learning and data science, and (iii) offer state-of-the-art performance. Lakehouses can help address several major challenges with data warehouses, including data staleness, reliability, total cost of ownership, data lock-in, and limited use-case support. We discuss how the industry is already moving toward Lakehouses and how this shift may affect work in data management. We also report results from a Lakehouse system using Parquet that is competitive with popular cloud data warehouses on TPC-DS.

1 Introduction

This paper argues that the data warehouse architecture as we know it today will wane in the coming years and be replaced by a new architectural pattern, which we refer to as the Lakehouse, characterized by (i) open direct-access data formats, such as Apache Parquet and ORC, (ii) first-class support for machine learning and data science workloads, and (iii) state-of-the-art performance.

The history of data warehousing started with helping business leaders get analytical insights by collecting data from operational databases into centralized warehouses, which then could be used for decision support and business intelligence (BI). Data in these warehouses would be written with schema-on-write, which ensured

quality and governance downstream. In this architecture, a small subset of data in the lake would later be ETLED to a downstream data warehouse (such as Teradata) for the most important decision support and BI applications. The use of open formats also made data lake data directly accessible to a wide range of other analytics engines, such as machine learning systems [30, 37, 42].

From 2015 onwards, cloud data lakes, such as S3, ADLS and GCS, started replacing HDFS. They have superior durability (often >10 nines), geo-replication, and most importantly, extremely low cost with the possibility of automatic, even cheaper, archival storage, e.g., AWS Glacier. The rest of the architecture is largely the same in the cloud as in the second generation systems, with a downstream data warehouse such as Redshift or Snowflake. This two-tier data lake + warehouse architecture is now dominant in the industry in our experience (used at virtually all Fortune 500 enterprises).

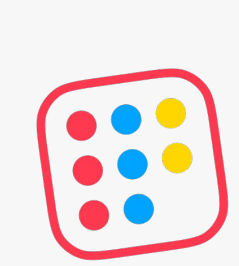
This brings us to the challenges with current data architectures. While the cloud data lake and warehouse architecture is ostensibly cheap due to separate storage (e.g., S3) and compute (e.g., Redshift), a two-tier architecture is highly complex for users. In the first generation platforms, all data was ETLED from operational data systems directly into a warehouse. In today's architectures, data is first ETLED into lakes, and then again ELTed into warehouses, creating complexity, delays, and new failure modes. Moreover, enterprise use cases now include advanced analytics such as machine learning, for which *neither* data lakes nor warehouses are ideal. Specifically, today's data architectures commonly suffer from four problems:

Reliability. Keeping the data lake and warehouse consistent is difficult and costly. Continuous engineering is required to ETL data

What is a Lakehouse?

- | Data on S3
- | Multi-use case
- | Multi-language (SQL, Python)

Interaction	UX	Infrastructure
Traditional DLH		
Batch pipeline	Submit API	One-off cluster
Dev. pipeline	Notebook Session	Dev. cluster
Inter. query	Web Editor (JDBC Driver)	Warehouse



4:17:01 (!?!)

YouTube

Search

+ Create

SPARK **ARCHITECTURE**

The diagram illustrates the Spark architecture. On the left, a stick figure labeled '1 TB' points towards the 'Driver program' box, which contains 'SparkContext'. In the center is the 'Cluster Manager'. On the right are two 'Worker Node' boxes, each containing an 'Executor' (with 'Cache' and 'Task' sub-components) and another 'Task' box. Arrows show the flow of data and control: from the Driver program to the Cluster Manager, and from the Cluster Manager to the Worker Nodes. Handwritten green annotations include '1 TB' next to the stick figure, 'W1' and 'W2' next to the Worker Nodes, and a green box around the Cluster Manager.

19:38 / 4:17:01

Tutorial (From Zero to Hero)

Masterclass



Ansh Lamba
46.6K subscribers

Join

Subscribe

4.8K



Share

Download



All

From Ansh Lamba

Big Data

Cloud cor



niniaOne



All In One RMM Solution





```
pip install bauplan  
bauplan checkout my-branch  
bauplan run
```



Speedrun a lakehouse → simplicity at the core

| Easy to reason about

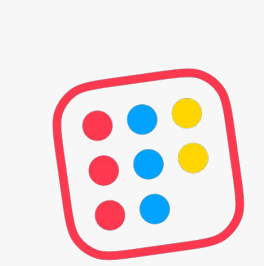
- Simple abstractions, “looks like code”
- A unified compute model, “everything is a function”



[VLDB 2023: Building a serverless Data Lakehouse from spare parts](#)

Functions everywhere!





A sample pipeline

transactions

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

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

euro_selection

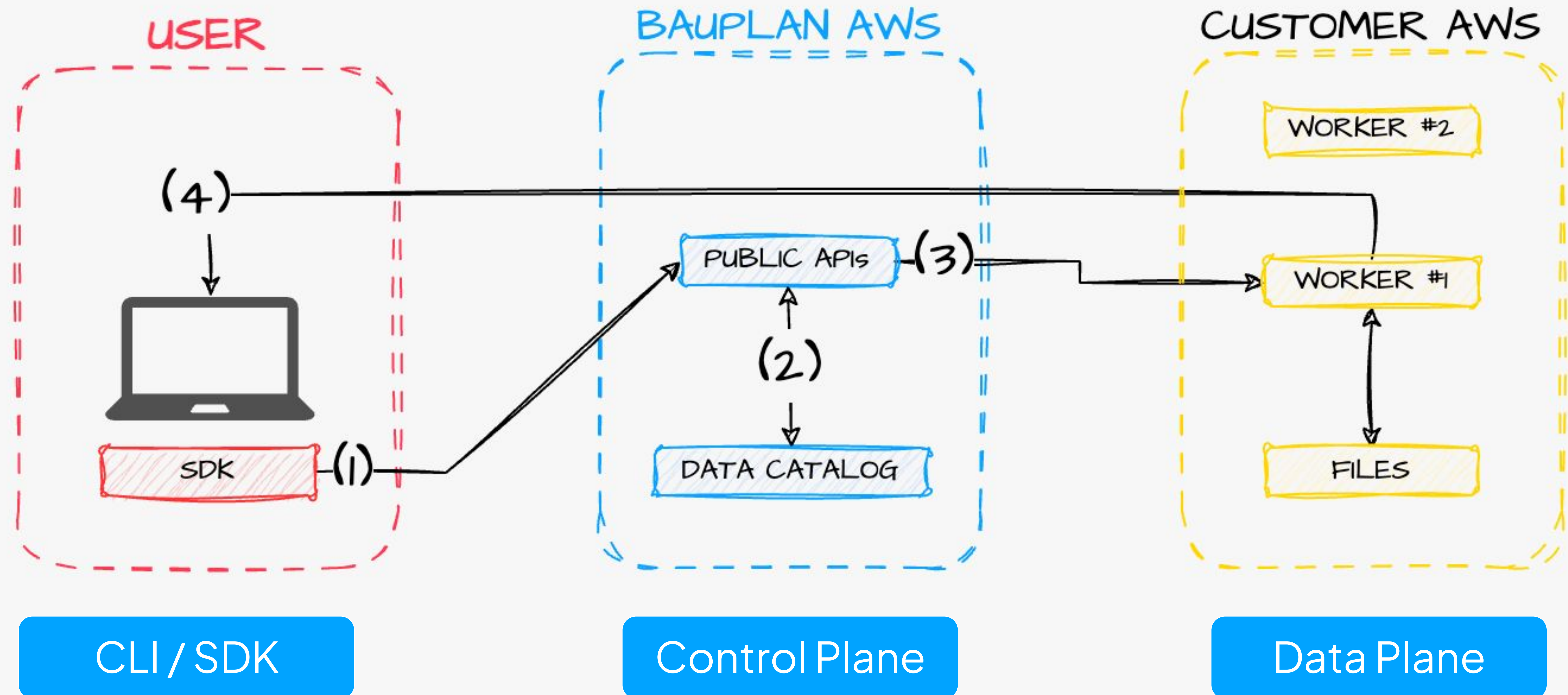
ID	USD	COUNTRY
144	13	IT
146	1	IT

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

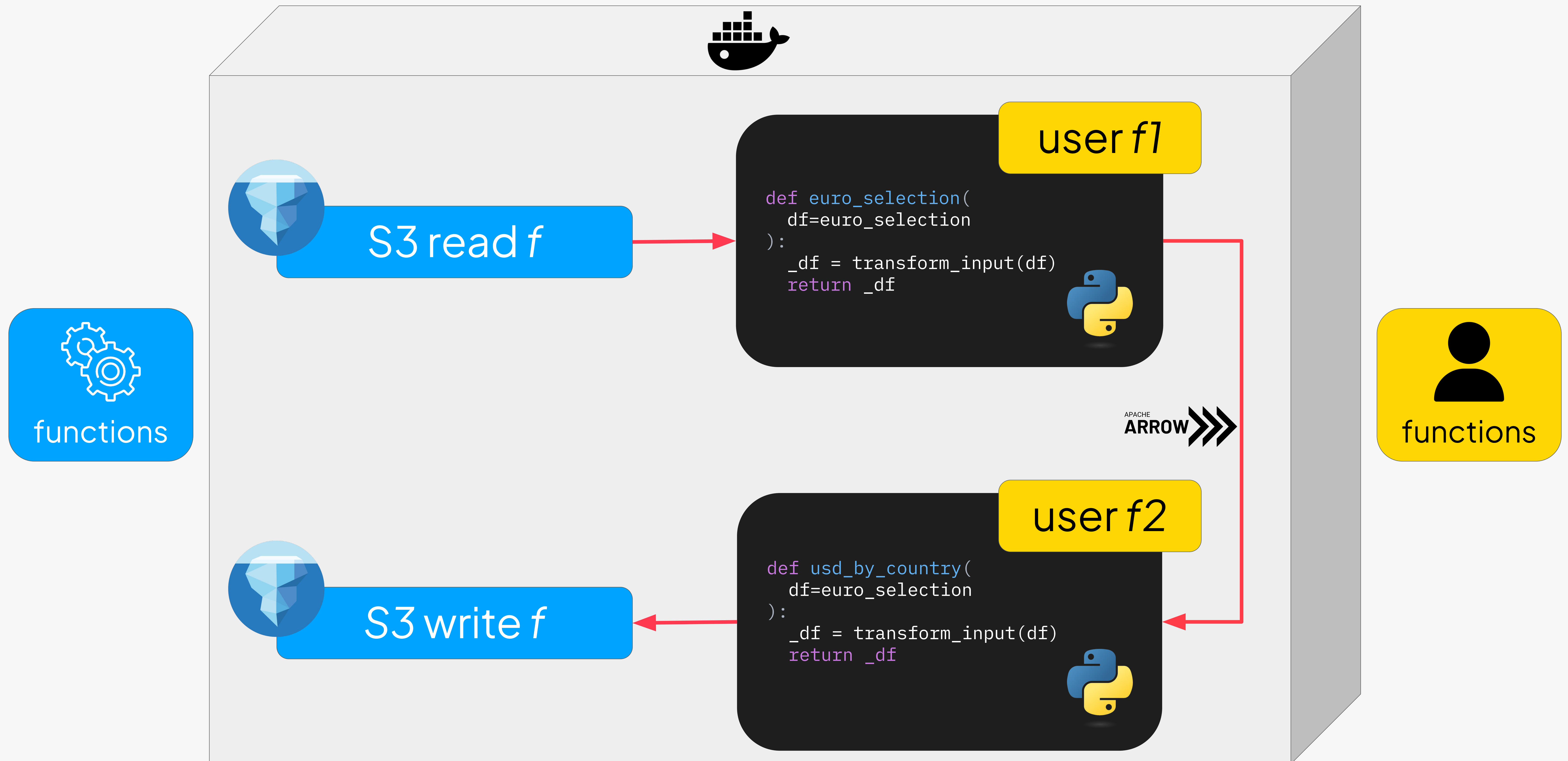
usd_by_country

COUNTRY	USD
IT	14

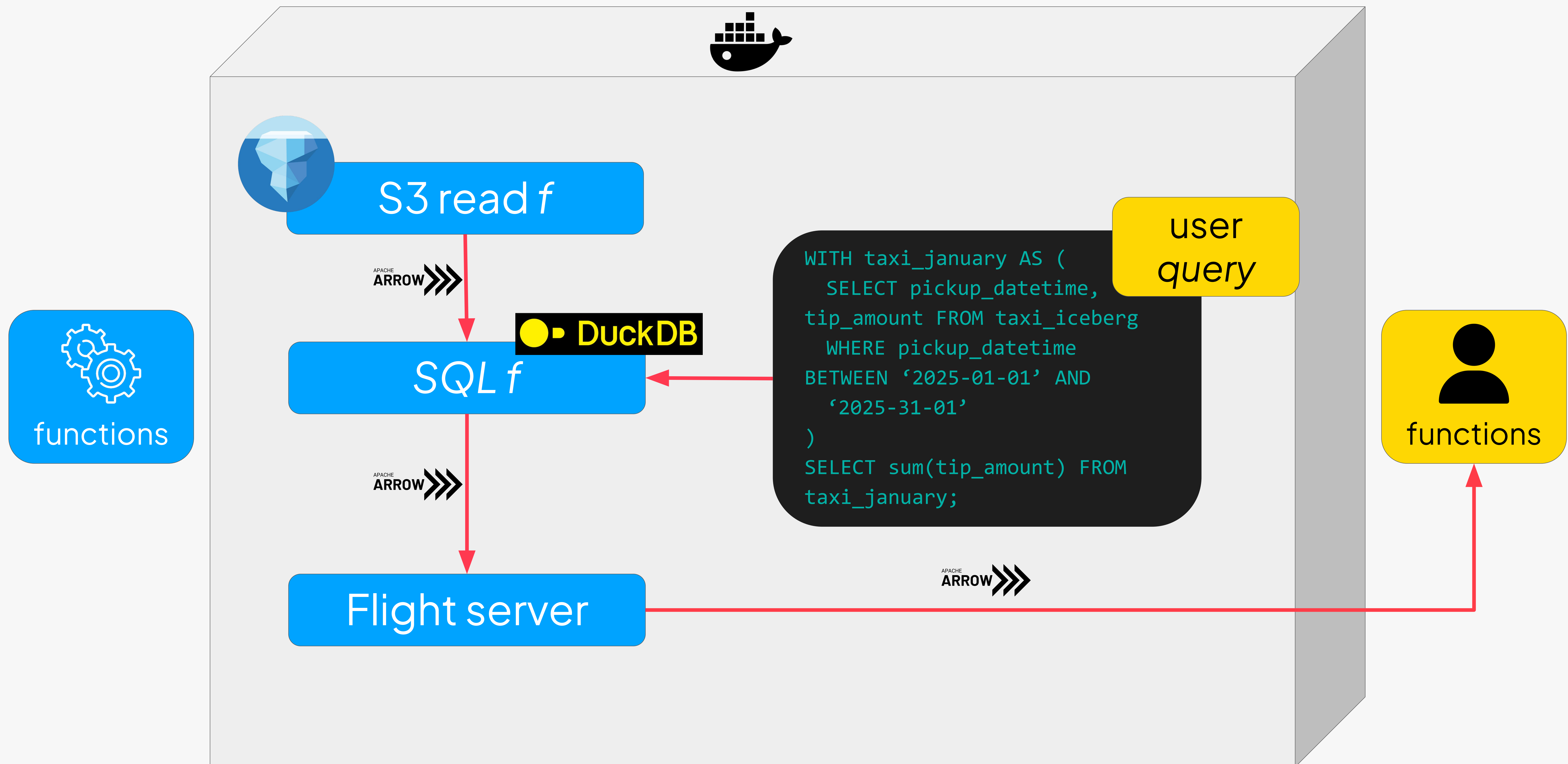
🎯 Not a FaaS, not a DB, but a secret third thing

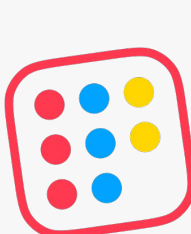


Pipelines are chained functions (Batch / Dev)



Queries are chained functions as well!





PROs: no heterogenous infra + trivial to learn

| Can we re-use existing FaaS? **NO!!!**

- Hardware limitations
- No “DAG awareness”
- Slow feedback loop

Interaction	UX	Infrastructure
Traditional DLH		
Batch pipeline	Submit API	One-off cluster
Dev. pipeline	Notebook Session	Dev. cluster
Inter. query	Web Editor (JDBC Driver)	Warehouse

CONs: need a new a FaaS-for-data

- | New programming model
 - Express data and code dependencies
- | New runtime
 - Function lifecycle
 - Scheduling

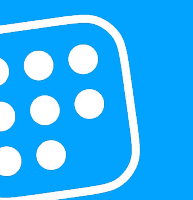
Interaction	UX	Infrastructure
Traditional DLH		
Batch pipeline	Submit API	One-off cluster
Dev. pipeline	Notebook Session	Dev. cluster
Inter. query	Web Editor (JDBC Driver)	Warehouse

New programming model





**clear “division of labor”
between platform and users**



New programming model

bauplan.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
```

bauplan.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
```

New programming model

Signature Table(s) -> Table

bauplan.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
```

bauplan.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
```


New programming model

Infra-as-code

bauplan.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
```

bauplan.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
```

New programming model

I/O chaining

bauplan.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
```

bauplan.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
```


New programming model

User code here!

bauplan.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
```

bauplan.py

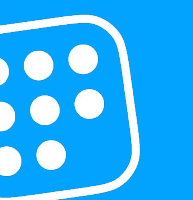
```
@bauplan.model(materialize=True)
@bauplan.python(
    "3.10",
    pip={"pandas": "1.5.3"}
)
def usd_by_country(
    data=bauplan.Model("euro_selection")
):
    # aggregation here
    # return a dataframe
    return _df
```

New runtime

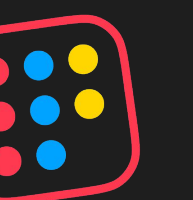




**we can't just “run user functions”, which
is a challenge and opportunity**



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

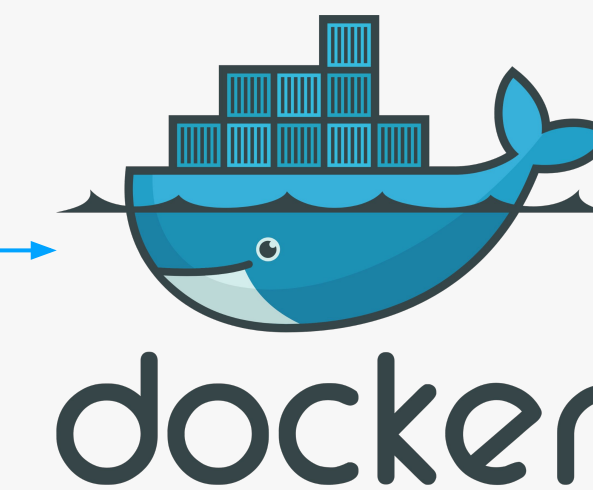


Planning

USER CODE

baup.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
```



RUN ...

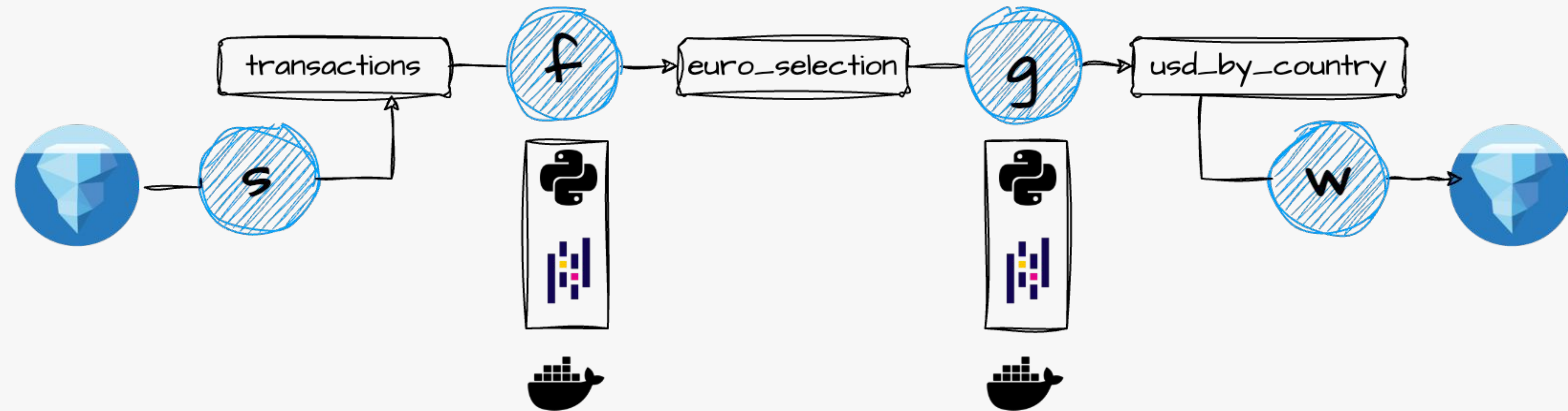


obj.get(Range='bytes=32-64')['Body']

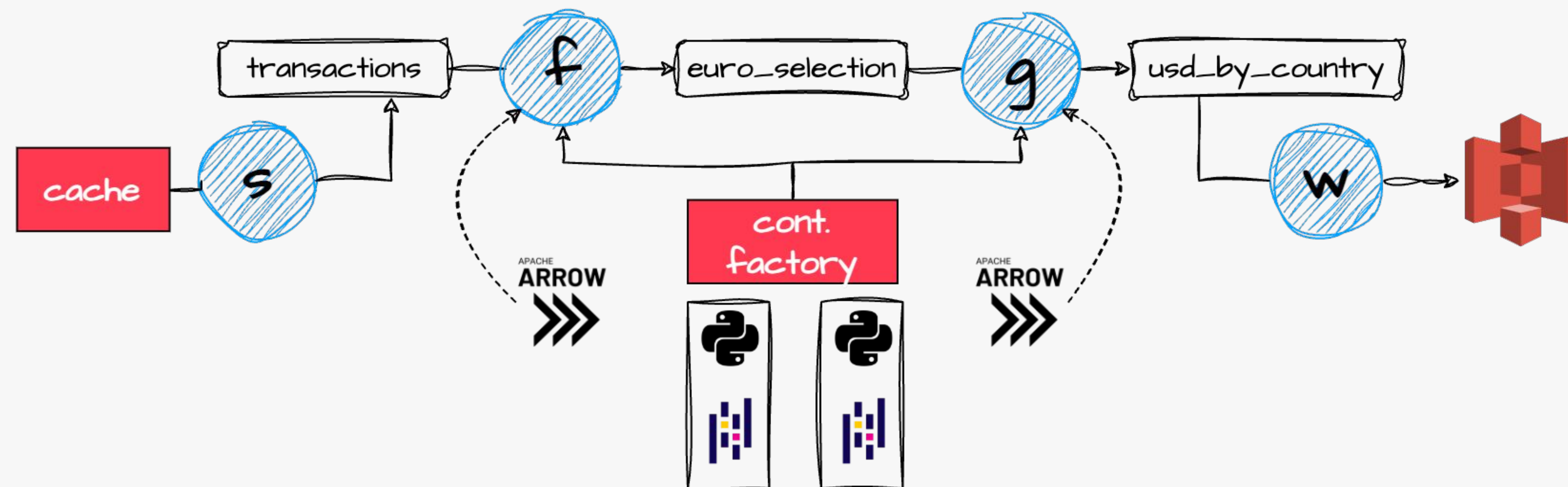
Planning



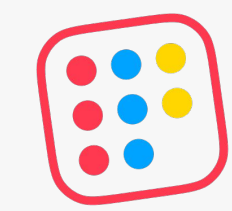
Logical



Physical

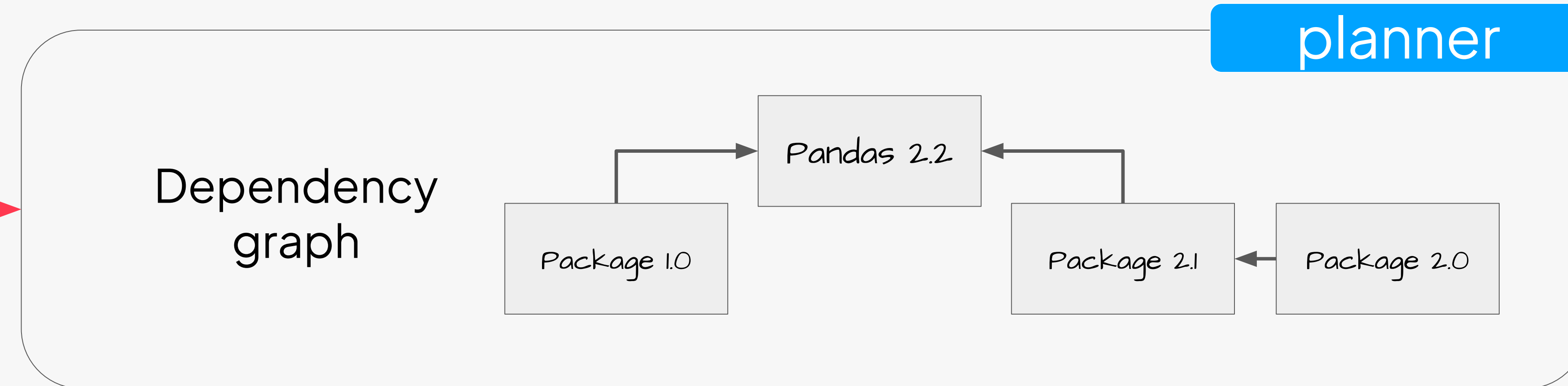
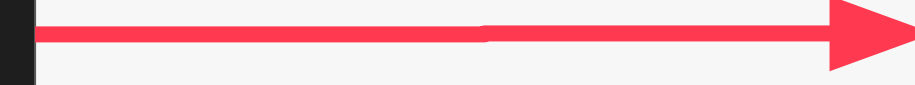


Worker



Environment

```
@bauplan.python(  
    "3.11",  
    pip={"pandas": "2.2"}  
)
```

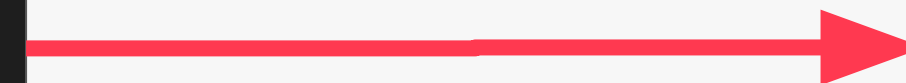


bauplan
cloud

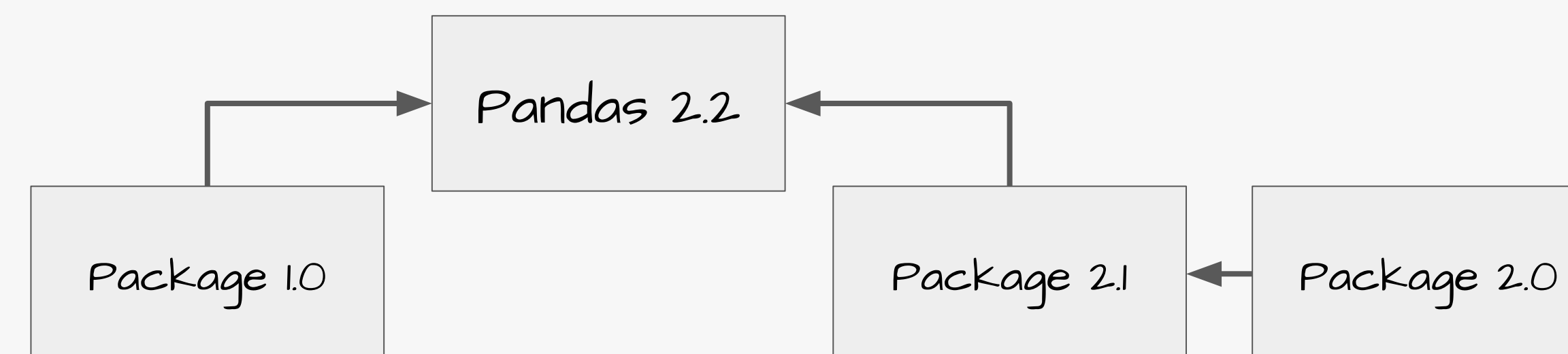


Environment

```
@bauplan.python(  
    "3.11",  
    pip={"pandas": "2.2"}  
)
```



Dependency
graph

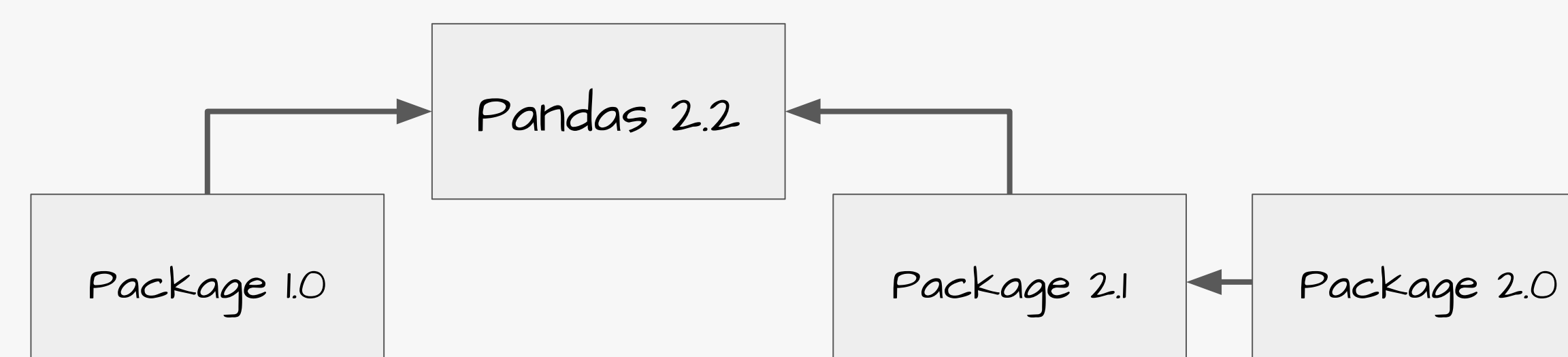
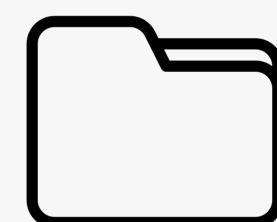


planner

bauplan
cloud

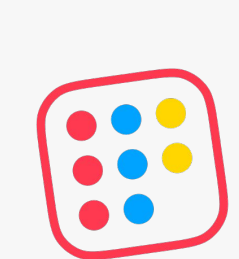


Install to ...



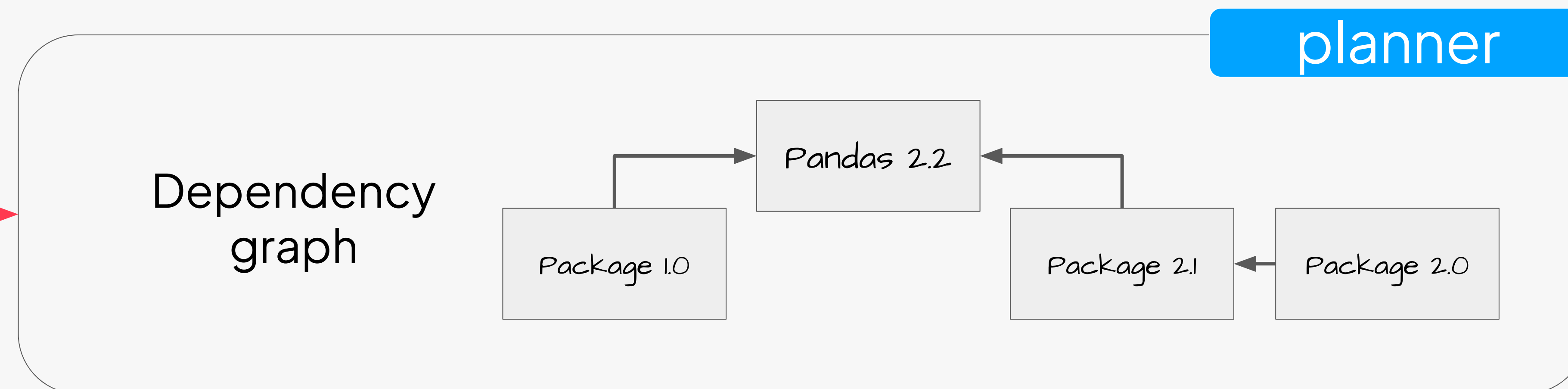
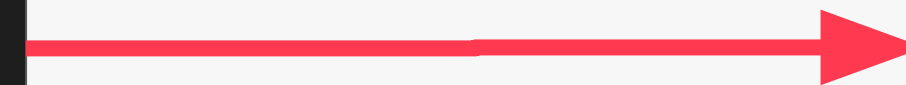
worker

customer
cloud

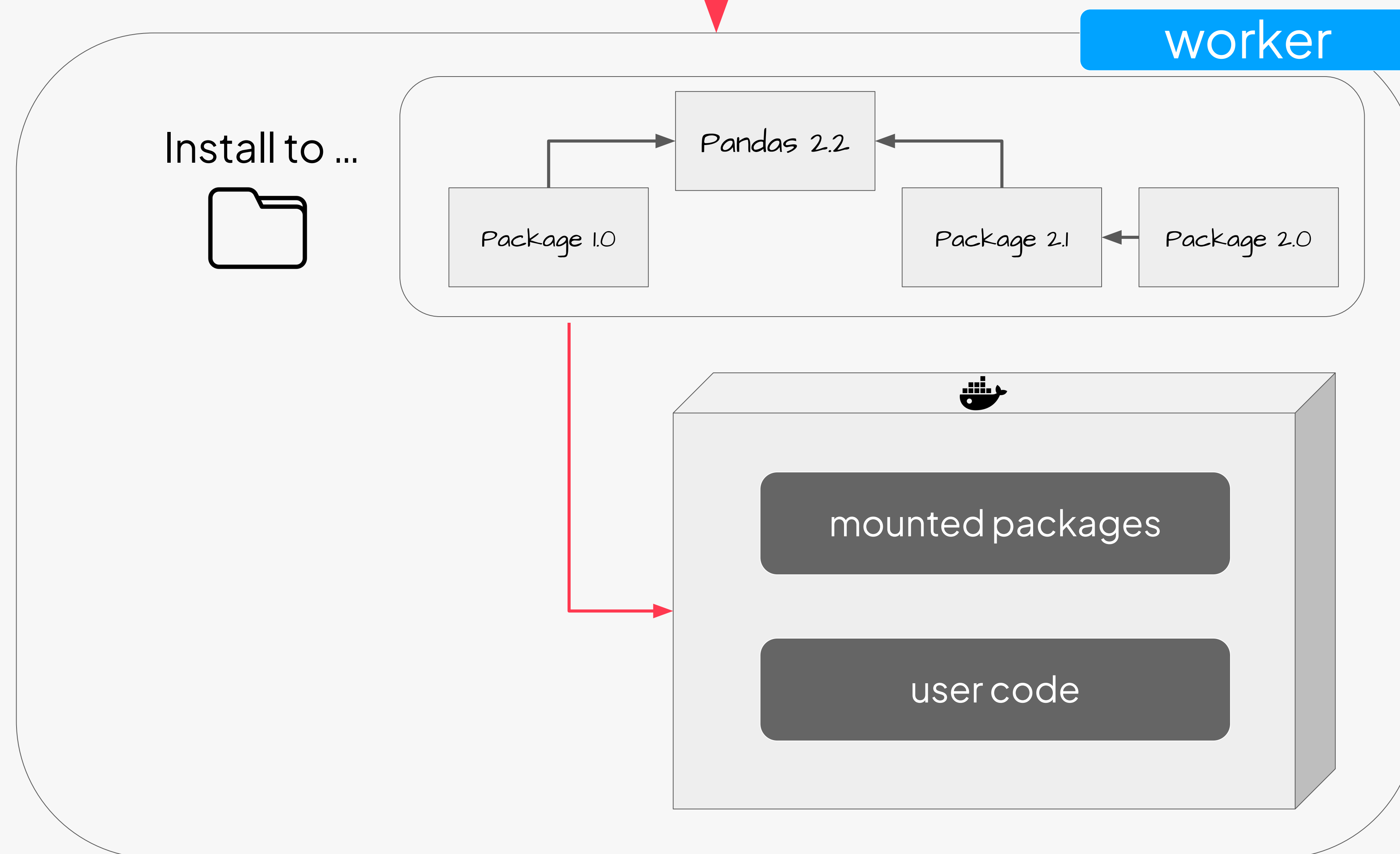


Environment

```
@bauplan.python(  
    "3.11",  
    pip={"pandas": "2.2"}  
)
```



bauplan
cloud



customer
cloud

Environment: assemble, don't build

- | **NO** Docker, **NO** bandwidth bottlenecks, **NO** ECR update
- | Functions are ephemeral: no lifecycle management.
- | **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
<i>bauplan</i>	
Update runtime	5 / 0 (cache)

Data movement: Arrow everywhere + zero-copy



```
data=bauplan.Model(  
    "transactions",  
    columns=["id", "usd", "country"],  
    filter="..."  
)
```

- | Across workers, an Arrow stream is as fast as local parquet files (**B**)
- | Within a worker, tables can be zero-copy shared between functions (**C**)
- | RQ: is **D** feasible?

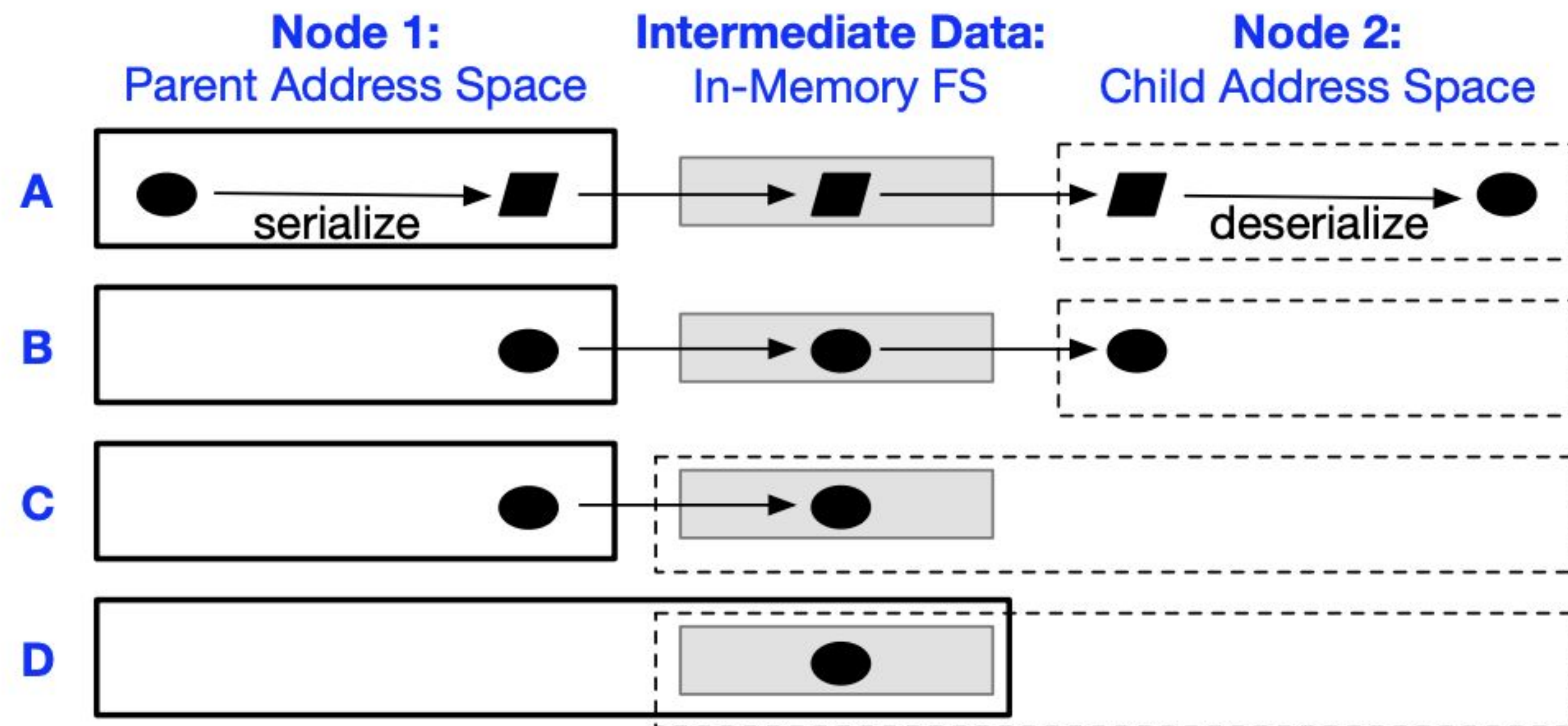


Figure 1: Communication: Degrees of Zero Copy

Data movement: Arrow everywhere + zero-copy

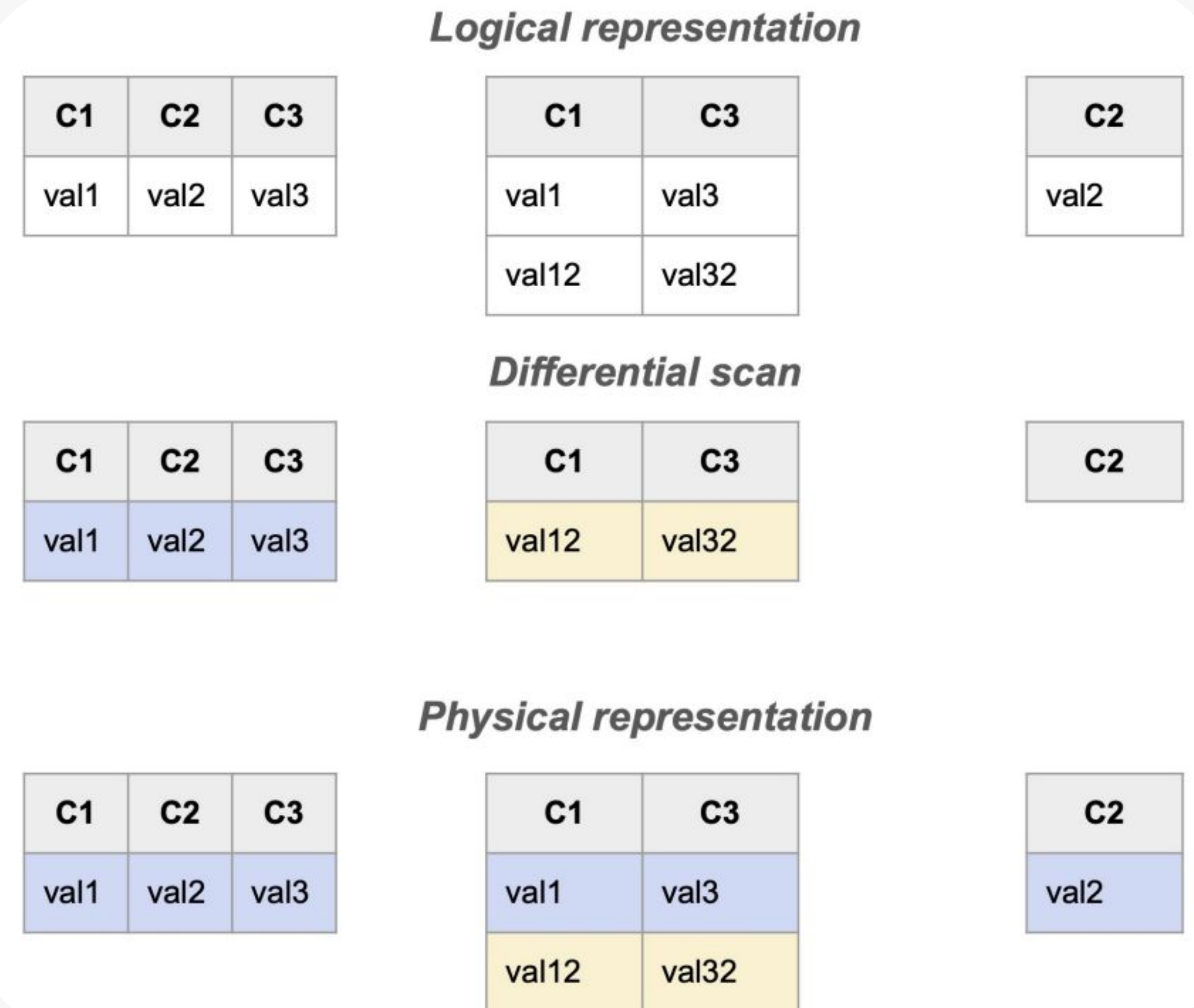
Table 3: Reading a dataframe from a parent (*c5.9xlarge*), avg. (SD) over 5 trials

	<i>10M rows (6 GB)</i>	<i>50M rows (30 GB)</i>
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)

Scans do not repeat themselves, but they often rhyme

Differential cache:

- | U1: “SELECT c1, c2, c3 FROM t WHERE eventTime BETWEEN 2023-01-01 AND 2023-02-01”
- | U2: “SELECT c1, c3 ... BETWEEN 2023-01-01 AND 2023-03-01”
- | U1: “SELECT c2 ... BETWEEN 2023-01-01 AND 2023-01-02”



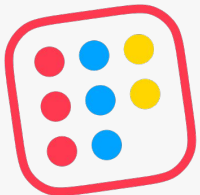
We barely scratched the surface!



Script everything: “devOps” = Python

```
1 import bauplan
2
3 client = bauplan.Client()
4 # run agents in parallel on branches
5 for i, agent in enumerate(agents):
6     agent_branch = client.create_data_branch(
7         branch=f"{i}_agent",
8         from_ref='main'
9     )
10    run_state = client.run(
11        dir=my_pipeline,
12        branch=agent_branch
13    )
14
15 # merge the best version
16 client.merge_data_branch(
17     source_ref=my_best_branch,
18     into_branch='main'
19 )
20
```

```
1 import bauplan
2
3 @bauplan.model()
4 @bauplan.python(pip={'pandas': '2.2.0'})
5 def clean_dataset(
6     input_table='nyc_taxi',
7     columns=[col1, 'col2']
8     filter="datetime = '2022-12-15'"
9 ):
10     import pandas as pd
11     return clean_dataset
12
13
14 @bauplan.model()
15 @bauplan.python(pip={'torch': '2.6.0'})
16 def train_model(input_table='clean_dataset'):
17     import torch
18     return predictions
19
```

True zero-copy and function scheduling

Under review

v2 [cs.OS] 13 May 2025

Zerrow: True Zero-Copy Arrow Pipelines in *Bauplan*

Yifan Dai*, Jacopo Tagliabue*, Andrea Arpaci-Dusseau*,
Remzi Arpaci-Dusseau*, Tyler R. Caraza-Harter**
* *University of Wisconsin–Madison*, * *Bauplan Labs*

Abstract. Bauplan is a FaaS-based lakehouse specifically built for data pipelines: its execution engine uses Apache Arrow for data passing between the nodes in the DAG. While Arrow is known as the “zero copy format”, in practice, limited Linux kernel support for shared memory makes it difficult to avoid copying entirely. In *this* work, we introduce several new techniques to eliminate nearly all copying from pipelines: in particular, we implement a new kernel module that performs de-anonymization, thus eliminating a copy to intermediate data. We conclude by sharing our preliminary evaluation on different workloads types, as well as discussing our plan for future improvements.

1 Introduction

Data pipelines are a popular programming paradigm for data analysis and machine-learning workloads. Data pipelines are frequently implemented as DAGs (Directed Acyclic Graphs), where each node of the DAG describes a transformation to perform on the data [2, 22, 26, 37]. Given fine-grained nodes, efficient communication between nodes is especially important for good performance [9, 20, 36]. Much work has addressed how to efficiently communicate between

be mapped by multiple downstream nodes. Unfortunately, simply using Arrow for inter-node communication does not eliminate several sources of copying and duplication in data pipelines. First, many tools and libraries that return Arrow data allocate space with malloc, which uses anonymously mapped memory without a backing file; operating systems (including Linux) do not typically support sharing of anonymous memory, so unless all libraries in the Arrow ecosystem are rewritten to use shared memory, a copy to shared memory is necessary. Second, DAG nodes must perform copies when Arrow output overlaps with Arrow input (*e.g.*, the node adds a column to an input table), as the existing Arrow IPC protocol does not provide a way to identify or reference such overlap. Finally, when independent DAGs deserialize the same data from on-disk formats (*e.g.*, Parquet files) to Arrow, different processes contain identical copies of Arrow data.

In pursuit of “true zero-copy” in a data DAG, we introduce Zerrow, an experimental system focused on i) our new kernel support for sharing anonymous memory and ii) our new extended Arrow IPC protocol. Zerrow introduces several new techniques. First, Zerrow introduces *de-anonymization*; our

| Zerrow: “true” zero-copy Arrow through kernel hacking

| Eudoxia: FaaS simulator for lakehouse scheduling policies

9 May 2025

Eudoxia: a FaaS scheduling simulator for the composable lakehouse

Tapan Srivastava*
tapansriv@uchicago.edu
University of Chicago
Chicago, Illinois, USA

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

Ciro Greco
ciro.greco@bauplanlabs.com
Bauplan Labs
New York, USA

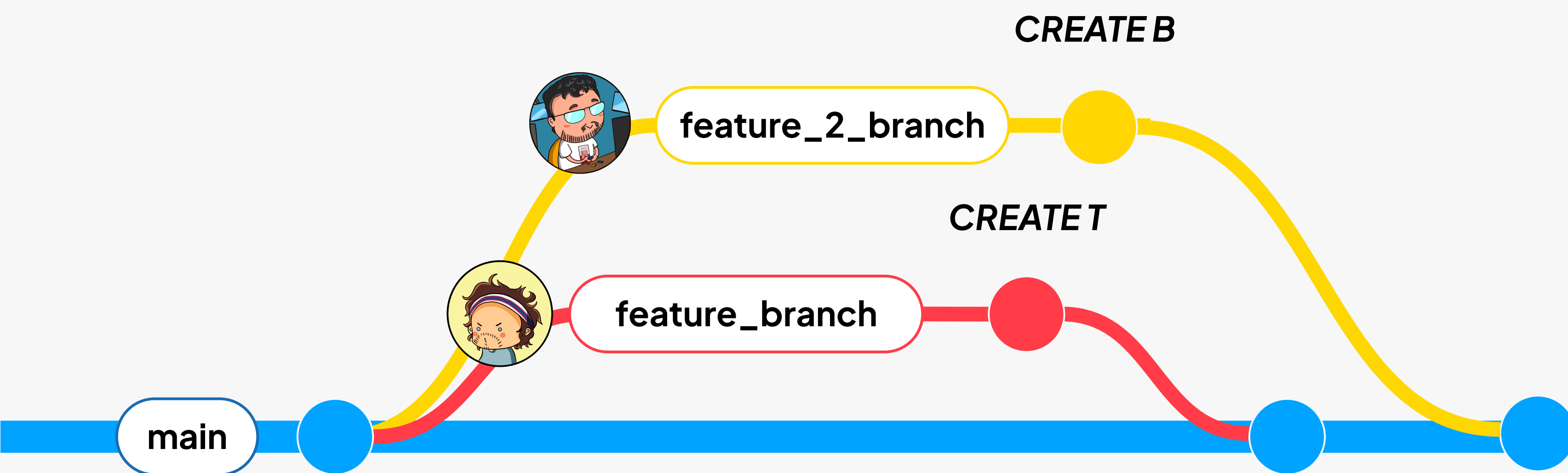
ABSTRACT

Due to the variety of its target use cases and the large API surface area to cover, a data lakehouse (DLH) is a natural candidate for a composable data system. *Bauplan* is a composable DLH built on “spare data parts” and a unified Function-as-a-Service (FaaS) runtime for SQL queries and Python pipelines. While FaaS simplifies both building and using the system, it introduces novel challenges in scheduling and optimization of data workloads. In this work, starting from the programming model of the composable DLH, we characterize the underlying scheduling problem and motivate

data lake and warehouse, such as cheap and durable foundation through object storage, compute decoupling, multi-language support, unified table semantics, and governance [19].

The breadth of DLH use cases makes it a natural target for the philosophy of composable data systems [23]. In this spirit, *Bauplan* is a DLH built from “spare parts” [31]: while presenting to users a unified API for assets and compute [30], the system is built from modularized components that reuse existing data tools through novel interfaces: *e.g.* Arrow fragments for differential caching [29], Kuzu for DAG planning [18], DuckDB as SQL engine [24], Arrow Flight for client-server communication [6].

How much “Git” is in Git-for-data?



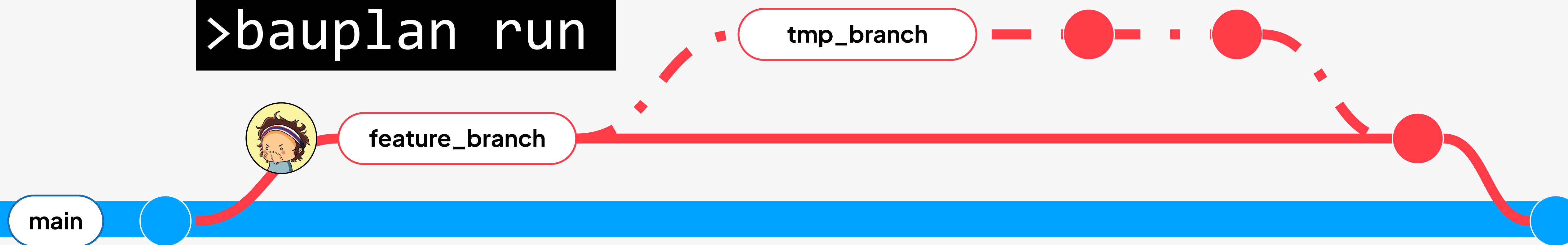
engineering Jul 17, 2025 Written by [Ciro Greco](#) and [Jacopo Tagliabue](#)

Git for Data: Formal Semantics of Branching, Merging, and Rollbacks (Part 1)

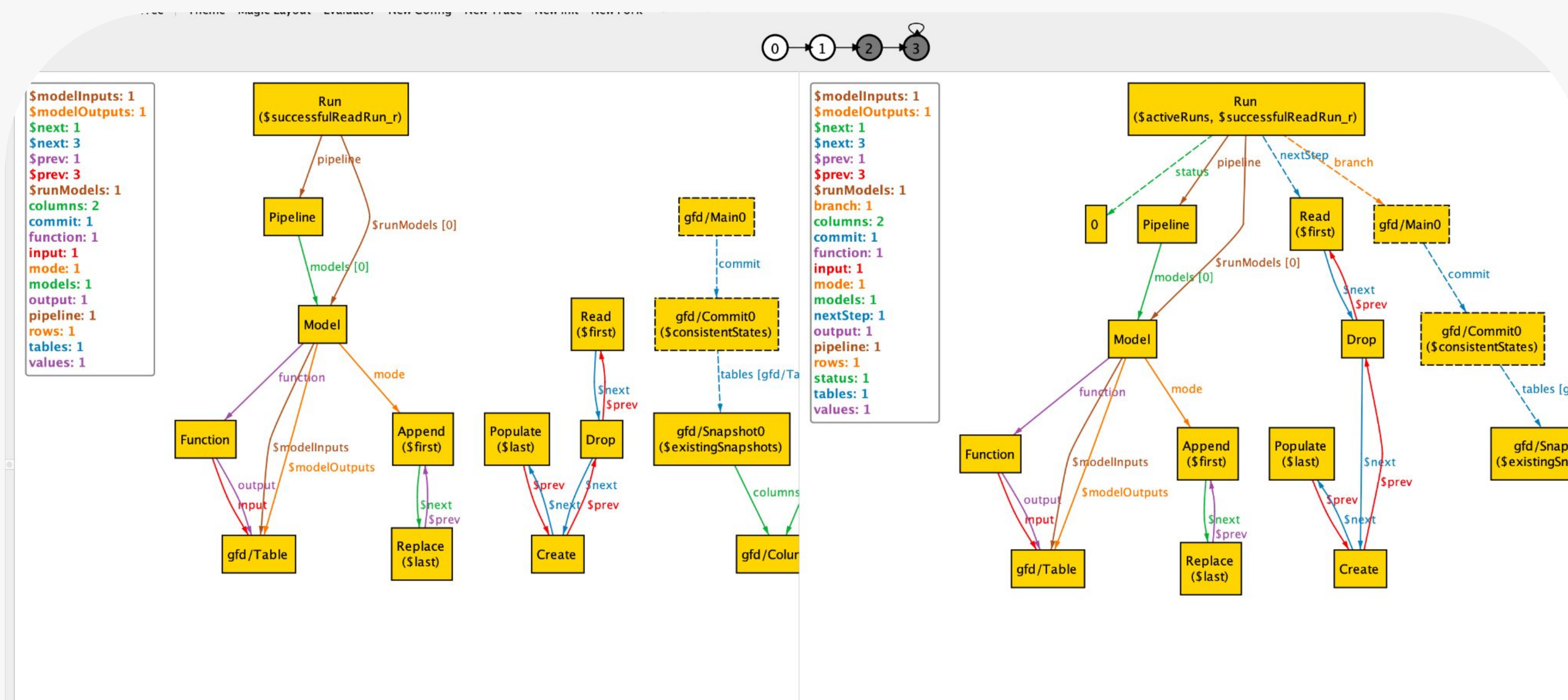
How formal methods help ensure safe, reproducible workflows in data lakehouses

How much “database” is in Git-for-data?

```
>bauplan run
```



“We have discovered a truly marvelous proof of this, which this slide is too narrow to contain”



Want to know more?

2023

- [CDMS@VLDB 2023](#)

2024

- [SIGMOD 2024](#)
- [MIDDLEWARE 2024](#) (with UMadison-Wisconsin)
- [BIG DATA 2024](#)

2025

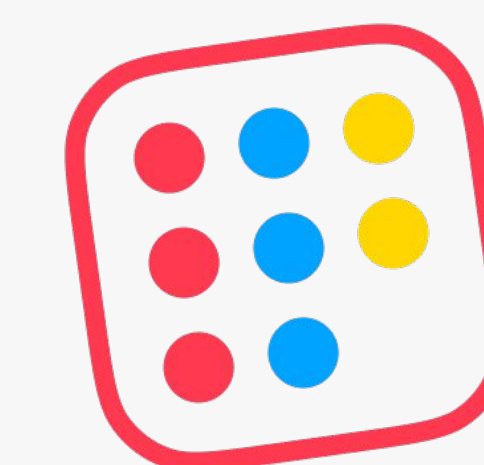
- [UNDER REVIEW 2025](#) (with UMadison-Wisconsin)
- [CDMS@VLDB 2025](#) (with UChicago)

| **(Most) lakehouse** use cases can be served by
functions

| **Co-designing abstractions + FaaS** leads to a
simple, powerful system



- | Want to chat?
jacopo.tagliabue@bauplanlabs.com
- | Are you coming to VLDB? Come to our
keynote: we are organizing a dinner!



bauplan