Bauplan: Zero-copy, Scale-up FaaS for Data Pipelines

Jacopo Tagliabue
Bauplan Labs
New York, USA

Tyler Caraza-Harter
University of Wisconsin-Madison,
Bauplan Labs
Madison, USA

Ciro Greco
Bauplan Labs
New York, USA

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 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

• **Computer systems organization** → **Cloud computing**; **Data flow architectures**.

Keywords

data pipelines, serverless, interactive function-as-a-service

ACM Reference Format:

Jacopo Tagliabue, Tyler Caraza-Harter, and Ciro Greco. 2024. *Bauplan*: Zero-copy, Scale-up FaaS for Data Pipelines. In *10th International Workshop on Serverless Computing (WoSC10 '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3702634.3702955

1 Introduction

The growth of Artificial Intelligence and Analytics applications has created a large, fast-growing market for data pipeline tools (USD 10BN/ year, with 22% CAGR [21]). Data pipelines are typically represented as Directed Acyclic Graphs (DAGs), where the nodes are functions that transform, aggregate, or clean the "raw" data for downstream use (Fig. 1). These functions typically map one or more input dataframes to one or more output dataframes [19].

This approach allows practitioners to break down complex business logic into modular, reusable components. Data pipeline workloads can be highly variable, even for a single user working on a single DAG: it is common to start with a preliminary run on, say, 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 [18], 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 [20], and Luigi [24]) 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 areas for improvement: scaling to handle large workloads, efficiently passing intermediate dataframes between functions, and rapidly adapting to developer changes. *Second*, we introduce *bauplan*, a specialized FaaS service built for executing data pipelines. *bauplan* fills the gap between traditional FaaS platforms and pipeline abstractions, offering a truly serverless experience designed with the needs of data workloads in mind. It features a novel programming model with simple annotations for standard Python or SQL functions, and a runtime enabling platform-level optimizations similar to a database query planner. For example, unlike Lambda's Docker-based execution for custom dependencies, *bauplan* provides a declarative API for managing Python packages at the function level, leaving caching and optimization to the platform. In common data science scenarios (Table 2), *bauplan*'s build process is 15x faster than *AWS Lambda* and 7x faster than *Snowpark*.

Unlike general-purpose FaaS platforms that allow any kind of input and output (provided that users manually take care of serialization and triggering), *bauplan* stores all intermediate data as (or automatically converted to) *Arrow* tables. Arrow is an open-source in-memory tabular format with a rich ecosystem that enables zero-copy data sharing between nodes on the same host and avoids serialization costs when data is sent across the wire (§4.3). Finally, *bauplan* planning and scheduling is managed by the provider, but invocations occur on the customer's virtual machines; a single invocation can use nearly all a machine's resources, so *bauplan* provides maximal scale up.

2 Why a new FaaS? An industry perspective

The life cycle of data projects is only superficially comparable to traditional software development [23], with its own pitfalls and failure modes [8, 25]. In contrast to traditional serverless use cases (*e.g.*, endpoints for web applications [22]), pipelines need support

WoSC10 '24, December 2–6, 2024, Hong Kong, Hong Kong

ACM ISBN 979-8-4007-1336-1/24/12
https://doi.org/10.1145/3702634.3702955

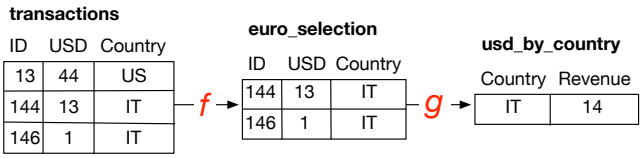

Figure 1: *A DAG of dataframes produced by transformations*: the source dataframe *transactions* is first filtered for European countries (generating *euro_selection*, then the aggregation of revenues by country is computed as *usd_by_country*.

in scaling up, handling intermediate artifacts, and boosting interactivity:

(1) *scaling up*: horizontal scaling through replicas is not as important as the ability to re-run the same function while (massively) scaling up hardware requirements between executions: recent industry traces place the *p99.9* memory footprint of a function between 50 and 200 GB for most real-world use cases [29];
(2) *large intermediate I/O*: functions often deal with large (>10 GBs) dataframes as their inputs and outputs, so the cost of serializing and moving the payload may be significant;
(3) *fast feedback loop*: unlike most software development projects, data science projects are open-ended and exploratory in nature; if no strategy is known from the start, the key to success is rapidly iterating over a set of hypotheses [30].

Below, we provide preliminary benchmarks on the cost of containerization (§4.2) and data movement (§4.3), as well as a brief overview of the ergonomics of FaaS platforms for scientific computing (§5). Here, we offer a comparison of commercial and open source FaaS limitations *vis-à-vis* the peculiarities of data pipelines. On the commercial side, we picked the two largest by market share, *AWS Lambda* and *Azure Functions* [18]; on the open source side, we picked the mature OpenWhisk, due to its community support, constant evolution [3], and backing of another popular commercial platform, *IBM Cloud Functions*. Not only are the numbers in Table 1 far from the requirements above, the platforms' chaining best practices (OpenWhisk action sequences, *AWS Step Functions*, and *Azure Durable Functions*) are even more limiting, as intermediate dataframes can only be moved through object storage. Instead of the dataframe itself, functions return a placeholder, resulting in incorrect DAG semantics and sub-optimal performance (§4.3).

Given the limitations of existing FaaS platforms, are DAG tools coupled with traditional compute better equipped to satisfy these requirements? If we take the most popular framework, *Airflow*, as an example, linking the programming model to *any* kind of compute is a non-trivial operation for practitioners. Even when *tasks* can be successfully coupled to a runtime (Kubernetes pods or an EC2 fleet), the framework does not provide dynamic scale-up, nor fine-grained, containerized management of Python dependencies. Importantly, *tasks* are generic operators, not data-aware functions, leaving users to either implement their own data passing or rely on built-in, object storage backed primitives ("XComs"). §3.3 below provides a vivid example of the shortcomings of this programming model when compared to *bauplan*: unsurprisingly, the end result for the average data practitioner is that the learning curve is notoriously steep, and the developer experience sub-optimal [31].

Table 1: Max. FaaS availability for I/O, memory, timeout

Platform	*Memory*	*I/O payload*	*Timeout*
Lambda	10GB	6 MB	900s
Functions	14GB	100 MB	unlimited
OpenWhisk	2GB	1 MB	300s

3 Platform Design

We now introduce *bauplan*, a new FaaS service specialized for running data pipelines. *bauplan* is designed to scale up individual function invocations, efficiently pass large intermediate data, and allow users to interactively modify and run DAGs. The rest of this section is organized as follows: we enumerate the design principles behind *bauplan* (§3.1), describe an architecture that prioritizes privacy of user data and deployment flexibility (§3.2), and introduce a new programming model for developing data pipelines (§3.3).

3.1 Design principles

The goal of our FaaS system is to natively satisfy the three *desiderata* in §2: it is therefore convenient to state design principles which will allow us to operationalize those requirements. The biggest difference between *bauplan* and existing tools is the conviction that achieving developer interactivity at the scale of modern data pipelines (*i.e.*, hundreds of millions of rows moving across containers) is only possible if DAG abstractions and the runtime are co-designed: *i.e.*, if serverless runtimes (*e.g.*, *AWS Lambda*) have no data awareness, and pipeline frameworks (*Airflow*) have no runtime awareness, *bauplan* provides both. Our *user experience* principles are therefore a combination of architectural insights and data abstractions:

- *it runs in the cloud, but feels local*: as DAGs require moving GBs of data, only cloud bandwidth can guarantee a fast feedback loop. However, remote runtimes often come at the expense of developer convenience, with slow build times and no interactive logging (§4.2).
- *data awareness*: users should write code at the logical level of data dependencies, not at the physical level of data representation. In other words, user functions should specify as inputs *only* tables, projections, and filters – not files or endpoints. By restricting function signatures to (semantically specified) dataframes, we open the door for platform-level optimizations, such as aggressive caching, zero-copy data sharing, and versioning (§4.2).

Our *system design* principles are focused on infrastructure and deployment:

- *ephemeral functions*: execution is truly stateless, as function instances live only for the duration of an invocation. In contrast, standard FaaS platforms typically reuse function instances across multiple invocations, making it harder for the average developer to reason about the program life cycle [13]. Starting fresh each time simplifies reasoning about invocations, and it also allows instantly re-executing functions with

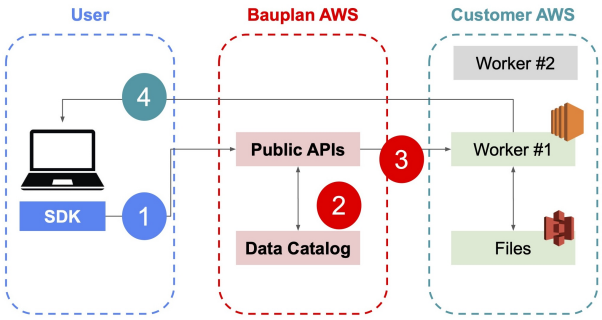

Figure 2: *End-to-end architecture*: 1) A user requests to run a DAG; 2) the APIs parse the request and send an execution plan to a worker; 3) an existing (bin-packing) or on-demand worker runs the required operations over customer data inside the customer cloud; 4) print statements and data previews are streamed back to the user.

different memory limits, which is a common requirement in data science (*e.g.*, running a pipeline first on January data, then on the full year);

- *off-the-shelf infrastructure*: cloud VMs provide the greatest level of customization and hardware diversity, which are both important levers when dealing with heterogeneous packages and large artifacts. VMs are also more portable than any serverless offering, appealing to enterprises who are sensitive to data security, and enabling *bauplan* to support multiple deployment models in every major cloud, from a managed service (with or without a private link), to a full Bring-Your-Own-Cloud.

Taken together, our design principles are a considerable departure from FaaS standards: ephemeral existence, VM portability, and data awareness are necessary to satisfy data pipeline requirements, but these properties are less useful (and perhaps counterproductive) for traditional use cases involving relatively independent functions.

3.2 Architecture

The basic modules involved in an *bauplan* run are depicted in Fig. 2. The system is built with a separation between the *Control Plane* (CP) and the *Data Plane* (DP). The CP is in *bauplan*'s cloud account and exposes multi-tenant APIs and only ever deals with *metadata*; for each customer we then have a single-tenant DP, a fleet of one or more off-the-shelf VMs which can be deployed (as in the picture) directly in a customer account. Workers are the only part of the system that interact with customer data; workers can be deployed in any cloud with ease as a Golang binary.

Crucially, the user and the workers are connected through bi-directional gRPC, so that every `print` statement in user code and system logs are visible in real-time in the user terminal / IDE. This is in stark contrast with platforms such as *AWS Lambda*, which provide only asynchronous observability through Cloudwatch at additional cost. Debugging a run in *bauplan* is a self-contained, real-time, and free operation; the execution feels as if it were local to the user.

3.3 Programming model

bauplan provides a CLI tool and Python SDK installable with *pip* (*i.e.*, `pip install bauplan`): the CLI allows users to initiate pipeline runs and navigate their data assets. The SDK provides decorators for user code and a client to interact with the platform from any Python process. The best way to understand the developer experience is to see how the DAG in Fig.1 is expressed in *bauplan*:

Listing 1: A sample DAG in bauplan

```
@bauplan.model()
@bauplan.python("3.11", pip={"pandas": "2.0"})
# the table name is the name of the function producing it
def euro_selection(
    # its parent node is referenced as the input
    data=bauplan.Model(
        "transactions",
        # columns and filters are expressed for
        # pushdown to object storage
        columns=["id", "usd", "country"],
        filter="eventTime BETWEEN 2023-01-01 AND 2023-02-01"
    )
):
    # do pre-processing here and return the
    # cleaned dataframe directly
    return _df

# specify that the output needs to be written back to S3
@bauplan.model(materialize=True)
@bauplan.python("3.10", pip={"pandas": "1.5.3"})
def usd_by_country(
    data=bauplan.Model("euro_selection")
):
    # aggregation code here - return as usual a dataframe
    return _df
```

Users express transformations as Python functions with the signature $f(dataframe(s)) \rightarrow dataframe$. The code is straightforward, but it is worth noting a few details:

- the DAG topology is *implicitly* expressed through function inputs;
- infrastructure is declarative: users specify the desired Python version and packages for each function, and the platform is in charge of deploying the corresponding stack – two functions may use different interpreters and versions of *pandas* within the same DAG;
- data management is declarative: users specify the desired input dataframe(s) for their code, and the platform is in charge of making it available inside the function by fetching it from object storage or from a parent. The optional hints on columns and filters enable optimizations such as predicate push-downs and columnar caching. Outputs follow the same principle: users return dataframes, and the platform is in charge of persisting them when required.

It is crucial to note that *bauplan*'s declarative nature creates a principled division of labor between the system (infrastructure and data management) and the data scientist (business logic and choice of libraries). Furthermore, this decoupling is necessary to run the same pipeline over different versions of the same table (*e.g.*, running

today's code on last Friday's table [26]), or over different *physical realizations* of the same asset (*e.g.*, Parquet files in S3, Arrow streams over the wire, or locally cached data). In this regard, it is worth noting how *bauplan* enables a true FaaS programming experience, in contrast to frameworks that instead couple the physical representation of a DAG with code. For example, consider this *AWS Airflow* reference implementation for a Python DAG [28]:

Listing 2: A pre-processing function in Airflow.

```
def preprocess(
      s3_in_url, s3_out_bucket, s3_out_prefix
   ):
   # Do pre-processing here and save the result in
   # "s3_out_bucket / s3_out_prefix"
   return "SUCCESS"

# the function gets registered in the overall DAG
preprocess_task = PythonOperator(
   task_id="preprocessing",
   dag=dag,
   python_callable=preprocess.preprocess
)
```

Not only does the pre-processing function operate at the physical level of the raw data path instead of at a logical DAG level, but it saves its output as a side effect, instead of returning the cleaned dataframe to the caller. Hiding data artifacts from the scheduler prevents any further optimization, such as caching or compression.

4 Implementation: the anatomy of a DAG run

To understand how all of the pieces fit together, we will decompose a run and dive deeper in some of the system optimizations. To get a first-person perspective on the developer experience (real time log streams, function building, data passing, etc.), the reader is invited to pause here and watch a recorded run before continuing: https://www.loom.com/share/99ac0d5b5f944fc9aeef132bfaea0881.

A successful DAG run involves three main operations, which we will describe in more detail. First, translate user code to a physical plan. User code is declarative, so the platform must fill the gap between *logical* requests (*e.g.*, "I want *transactions* with columns *ID, USD, COUNTRY*") and system operations (*e.g.*, "read files XYZ from S3") (§4.1). Second, optimize environment construction so that the code can run in quickly provisioned ephemeral functions (§4.2). Third, optimize I/O and data movement (§4.3). In each step, *bauplan* leverages data awareness and runtime awareness to perform optimizations not found in other comparable systems.

4.1 Logical and physical plan

Unlike other FaaS platforms, which execute user code "as is", *bauplan* acts more like a database, where user code needs "translation" before being executed. When a run is requested from the CLI / SDK, the user code is uploaded to the control plane (CP) for processing and scheduling (Fig. 2). The CP parses the code and reconstructs the DAG topology from the inputs / outputs of the functions: using database jargon, the result is the "logical plan", a structured representation of user code, expressing dependencies between steps with dataframe semantics (Fig. 3, *top*). This representation is too

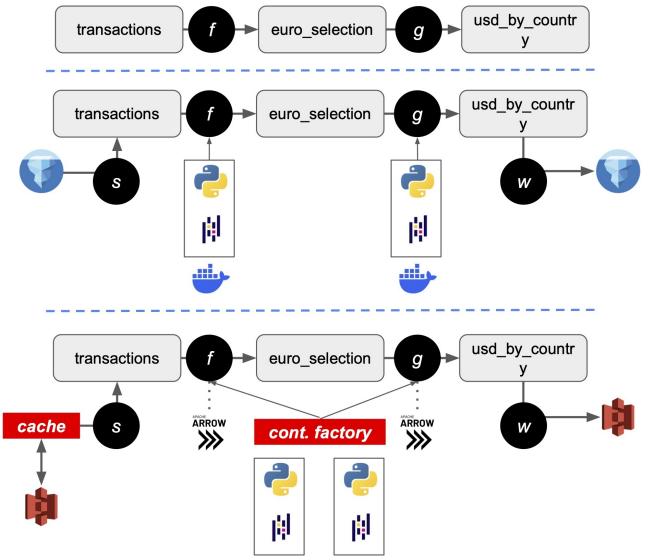

Figure 3: *From logical plan to execution*, with three levels of representation for every run: (top to bottom) 1) logical plan, obtained by parsing user code, 2) physical plan, obtained by adding system functions, 3) the actual worker-level execution, transparently managing data and package caches to further speed up execution.

abstract to be sent to workers, so the CP will also produce a "physical plan" (Fig. 3, *middle*), which contains explicit instructions for the containerized runtime of the transformation functions f and g (*e.g.*, *euro_selection* and *usd_by_country*), and maps dataframe semantics to S3-backed tables. The CP accomplishes this by leveraging *Iceberg* as its table format (providing schema evolution and per-table snapshots) and *Nessie* [10] as a data catalog (providing cross-table transactions and data lake branching). Transparent to the user, the runtime has support for optimized procedures to read the *transactions* dataframe from S3 and write back *usd_by_country* as the final dataset. Throughout this procedure, no customer data is ever visible to the CP.

The physical plan is now specific enough to be sent to a worker (Fig. 3, *bottom*), which is in charge of executing it while taking into account local optimizations and constraints. In particular, the worker can access the files in the customer S3 bucket, as well as cleverly leverage space, time, and team locality: similar users with similar DAGs would access similar packages and S3 objects, opening up the possibility of performing significant optimizations in package (§4.2) and data re-use (§4.3). Notably, all these optimizations require no code changes nor additional cognitive effort on the user side, and they would not be possible without *bauplan*'s declarative programming model.

4.2 Towards a fully interactive FaaS

Data practitioners frequently add or remove data science packages or update the versions of those packages. Like many other FaaS platforms, *bauplan* guarantees code portability ("infrastructure-*is*-code") and selective maintenance: a new function can benefit from *pandas==2.0* without forcing breaking changes in old functions

with *1.5.3*. However, due to its focus on interactivity and dynamic scaling, *bauplan* significantly departs from existing FaaS along two dimensions: ephemeral function building and data caching.

As mentioned earlier (§3.1), functions only exist for the duration of a single invocation: two subsequent runs will build *two* containers for the same *euro_selection* function (possibly with different hardware resources), which would result in unacceptable latency if re-building functions were a slow operation. To give a sense of the usual re-containerization flow for data science packages in popular platforms, Table 2 reports the latencies we recorded[1] when re-running a DAG on a given target stack after adding a new Python package (the *prophet* prediction library). *bauplan* is 7x faster than *Snowpark* – a data-aware, serverless experience built within a cloud warehouse – and 15x faster than *AWS Lambda* at closing the developer loop, yet requires no additional tools, nor special knowledge. This improvement comes from two major insights. First, the worker leverages a local container factory to avoid re-installing common packages across runs, thus removing the dependency from network calls to PyPI. Second, our narrower use case compared to general platforms allows us to avoid relying on image layers (as for example *AWS Lambda* does), with their associated building and network costs. In particular, our atomic building blocks for environments are the Python packages themselves. To the best of our knowledge, we implemented OpenLambda-style package initialization [11] in a Docker-compatible runtime as a novel experiment before any other research team. As a result, *bauplan* containers are assembled in 100s of milliseconds by mounting relevant modules from the local file system.

Given all workers are single-tenant (Section 3.2), host disk and memory can be more easily shared between subsequent executions: instead of exposing APIs for manual management of temporary storage (like *Lambda*), the *bauplan* data cache works without user intervention, exploiting its signature with no side-effects. Once again, *bauplan* data awareness makes database-like optimizations (not available in other platforms) possible:

- intermediate dataframes are produced by DAGs functions, and *bauplan* tracks both code and data changes, so it is possible to cache and re-use intermediate steps to avoid unnecessary re-computations when iterating;
- the semantics of reading *Iceberg* tables follows relational algebra, so the cache can be *columnar and differential*: for example, after reading from *transactions* once, a subsequent request for *ID, USD, COUNTRY, CLIENT_ID* re-uses *ID, USD, COUNTRY* from the cache and only downloads *CLIENT_ID* from object storage;
- inputs over object storage map to immutable files (through the *Iceberg* manifest), so dataframe changes are identified with data commits such that the cache knows with certainty when a table is stale.

4.3 Intermediate dataframes

As data workloads involve reading, writing, and moving around large dataframes, a *cloud-only* design is a necessity, with cloud virtual machines (*e.g.*, EC2 instances) offering network bandwidth up to 50 Gbps. However, running cloud functions with manual data management is still sub-optimal, as data scientists are likely to end up with slow implementations, low throughput, and complex logic to maintain. *bauplan* constrains user inputs / outputs to be dataframes: raw dataframes in object storage are stored in Parquet files as *Iceberg* tables, following lakehouse standards [27, 32] and maximizing interoperability. On the other hand, *bauplan* represents intermediate dataframes as Arrow tables. Arrow is a popular open-source columnar format for in-memory and over-the-wire tabular data [2]. Arrow is built for modern CPUs and vectorized execution and has a fast-growing ecosystem built around it [9, 12]. The Arrow format carefully avoids the use of pointers to memory addresses, preferring instead offset buffers to represent location and bitmaps to represent null values. Avoiding pointers allows the same Arrow data to be mapped to different locations in different address spaces with minimal modification; as a result, Arrow avoids serialization and deserialization overheads and supports zero-copy transfers. This results in significant performance gains relative to traditional columnar formats like Parquet or row-based wire protocols like JDBC. In the context of a DAG (such as the one in the demonstration above), Arrow is a crucial component for a fast feedback loop.

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

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

As a pipeline is executed, the platform transparently picks a sharing mechanism: shared memory or local disk (for co-located functions) or Arrow Flight (across workers). To give a sense of the performance gains, Table 3 shows how long it takes to read an intermediate dataframe into a user function, depending on serialization and storage type[2]; since existing platforms can only support S3-backed data passing (§2), moving dataframes in *bauplan* can *be hundreds of time faster than alternatives*: perhaps counter-intuitively, cross-machine communication through Arrow Flight is nearly as fast as local Parquet reading. Moreover, Arrow is very flexible: functions can transparently read tables from shared memory, memory-map them from disk (with standard OS primitives), or stream them from gRPC (with Flight) depending on resource availability.

Importantly, tables can often be shared with no movement at all: when children functions can be scheduled in the same worker, *bauplan* performs a zero-copy sharing of the parent output. The Arrow IPC module allows children to transparently reference the underlying memory buffer, thus avoiding unnecessary copies: a 10 GBs table with three children only requires 10 (not 30) GBs, saving considerable time and resources.

[1]Code snippets are available at https://github.com/BauplanLabs/WoSC-2024.

[2]Code snippets are available at: https://gist.github.com/jacopotagliabue/57bb14c675a5375338d4a57a88cea32a.

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)**

5 Related Work

Aside from platforms already discussed (§2), there is academic interest in serverless systems that address some of the challenges we have encountered. For example, Mahgoub *et al.* [17] and Lopez *et al.* [16] investigate data passing from a serverless perspective, but provide no abstractions for data-awareness nor zero-copy capabilities: as Table 3 shows, tackling both storage *and* serialization is critical. *Lithops* [15] and Carreira *et al.* [6] both focus on serverless for scientific use cases (Montecarlo simulations and Machine Learning, respectively), but their focus make DAGs hard to represent within the framework. Jia *et al.* [14] also use shared memory to allow inter-function communication, but their stated goal is "interactive microservices", resulting in a low-latency, low-complexity system, with no affordances for data practitioners and no principled way to deal with storage and serialization trade-offs. Importantly, none of these systems is known to be used in production, making it hard to assess how the design choices actually fit real-world scenarios, especially given the known unreliability of analytics benchmarks [29]. In general, data practitioners are more familiar with SQL [7] and Python DAG frameworks [1, 20, 24]. While some abstractions of these frameworks resemble *bauplan*'s, they are not able to make runtime and data optimizations for the user: working within these frameworks often results in a tight coupling between business logic and data representation (§3.3), with sub-optimal performance and a non-FaaS experience.

6 Conclusion

While chaining is supported by major FaaS platforms, we have argued that their generality prevents them from fully supporting modern data pipelines. We outlined *bauplan*, a purpose-built FaaS programming model and runtime, which effectively trades off control to achieve large gains in performance and developer experience. Notwithstanding its novelty, *bauplan* is already used in by large enterprises: as the platform further matures, we look forward to sharing with the community the next steps of our journey.

References

[1] Apache. 2024. Airflow. https://github.com/apache/airflow.
[2] Apache. 2024. Arrow. https://github.com/apache/arrow.
[3] Apache. 2024. OpenServerless. https://incubator.apache.org/projects/openserverless.html.
[4] Apache. 2024. OpenWhisk. https://github.com/apache/openwhisk.
[5] AWS. 2024. AWS Developer Guide. https://docs.aws.amazon.com/lambda/latest/api/API_Invoke.html.
[6] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3357223.3362711
[7] dbt-labs. 2024. dbt-core. https://github.com/dbt-labs/dbt-core.
[8] Dimensional Research. 2022. What Data Scientists Tell Us About AI Model Training Today. https://content.alegion.com/dimensional-researchs-survey
[9] Dremio. 2024. dremio-oss. https://github.com/dremio/dremio-oss.
[10] Dremio. 2024. Nessie. https://github.com/projectnessie/nessie.
[11] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson
[12] Influx Data. 2024. influxdb. https://github.com/influxdata/influxdb.
[13] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal Foundations of Serverless Computing. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 149 (oct 2019), 26 pages. https://doi.org/10.1145/3360575
[14] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 152–166. https://doi.org/10.1145/3445814.3446701
[15] Lithops. 2024. Lithops. https://github.com/lithops-cloud/lithops.
[16] Pedro García López, Aitor Arjona, Josep Sampé, Aleksander Slominski, and Lionel Villard. 2020. Triggerflow: trigger-based orchestration of serverless workflows. *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems* (2020). https://api.semanticscholar.org/CorpusID:219708396
[17] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 285–301. https://www.usenix.org/conference/atc21/presentation/mahgoub
[18] Microsoft. 2024. Azure Functions. https://azure.microsoft.com/en-us/products/functions/.
[19] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. 2020. Towards Scalable Dataframe Systems. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2033–2046. https://doi.org/10.14778/3407790.3407807
[20] PrefectHQ. 2024. Prefect. https://github.com/PrefectHQ/prefect.
[21] Research and Markets. 2024. Global Data Pipeline Tools Market by Component. https://www.researchandmarkets.com/report/data-pipeline-tools
[22] Michele Sciabarrà. 2019. Learning Apache OpenWhisk: Developing Open Source Serverless Solutions. https://www.oreilly.com/library/view/learning-apache-openwhisk/9781492046158/
[23] Shreya Shankar, Rolando Garcia, Joseph M. Hellerstein, and Aditya G. Parameswaran. 2022. Operationalizing Machine Learning: An Interview Study. https://doi.org/10.48550/ARXIV.2209.09125
[24] Spotify. 2024. Luigi. https://github.com/spotify/luigi.
[25] Jacopo Tagliabue, Hugo Bowne-Anderson, Ville Tuulos, Savin Goyal, Romain Cledat, and David Berg. 2023. Reasonable Scale Machine Learning with Open-Source Metaflow. *ArXiv* abs/2303.11761 (2023).
[26] Jacopo Tagliabue and Ciro Greco. 2024. Reproducible data science over data lakes: replayable data pipelines with Bauplan and Nessie. In *Proceedings of the Eighth Workshop on Data Management for End-to-End Machine Learning* (Santiago, AA, Chile) *(DEEM '24)*. Association for Computing Machinery, New York, NY, USA, 67–71. https://doi.org/10.1145/3650203.3663335
[27] Jacopo Tagliabue, Ciro Greco, and Luca Bigon. 2023. Building a Serverless Data Lakehouse from Spare Parts. *ArXiv* abs/2308.05368 (2023). https://api.semanticscholar.org/CorpusID:260775634
[28] Rajesh Thallam and Martin Dominguez. 2024. Build end-to-end machine learning workflows with Amazon SageMaker and Apache Airflow. https://github.com/aws-samples/sagemaker-ml-workflow-with-apache-airflow/blob/master/src/dag_ml_pipeline_amazon_video_reviews.py
[29] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Eknath Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is not enough: An analysis of the Amazon Redshift fleet. In *VLDB 2024*.
[30] Doris Xin, Litian Ma, Shuchen Song, and Aditya G. Parameswaran. 2018. How Developers Iterate on Machine Learning Workflows - A Survey of the Applied Machine Learning Literature. *ArXiv* abs/1803.10311 (2018). https://api.semanticscholar.org/CorpusID:4378544
[31] Jerin Yasmin, Jiale Wang, Yuan Tian, and Bram Adams. 2024. An Empirical Study of Developers' Challenges in Implementing Workflows as Code: A Case Study on Apache Airflow. *ArXiv* abs/2406.00180 (2024). https://api.semanticscholar.org/CorpusID:270213226
[32] Matei A. Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *Conference on Innovative Data Systems Research*.