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 simulations as an effective tools to iterate on the DLH. We then introduce and release to the community Eudoxia , a deterministic simulator for scheduling data workloads as cloud functions. We show that Eudoxia can simulate a wide range of workloads and enables highly customizable user implementations of scheduling algorithms, providing a cheap mechanism for developers to evaluate different scheduling algorithms against their infrastructure.

VLDB Workshop Reference Format:
Tapan Srivastava, Jacopo Tagliabue, and Ciro Greco. *Eudoxia*: a FaaS scheduling simulator for the composable lakehouse. VLDB 2025 Workshop: Third International Workshop on Composable Data Management Systems.

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/BauplanLabs/eudoxia.

1 INTRODUCTION

> "In Eudoxia, (...), a carpet is preserved in which you can observe the city's true form. At first sight nothing seems to resemble Eudoxia less than the design of that carpet (...), but if you pause and examine it carefully, you become convinced that each place in the carpet corresponds to a place in the city and all the things contained in the city are included in the design." (I. Calvino, *Invisible Cities*)

The Data Lakehouse (DLH) [39], is becoming the *de facto* cloud standard for analytics and Artificial Intelligence (AI) workloads. The DLH promises many improvements over its predecessors, the 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].

Bauplan serves interactive and batch use cases through a unified Function-as-a-Service (FaaS) runtime running on standard VMs [28]. The complexity of resource management in a dynamic, multi-language DLH thus reduces to "just" scheduling functions. Building and testing distributed systems is complex, costly, and error-prone in monolithic systems [10, 17, 37] and is even more so in composable data systems. In order to test our intuitions and safely benchmark policies, we decided to build and release a DLH simulator.

In this work, we present Eudoxia, a scheduling simulator designed for the composable DLH. Our contributions are threefold:

(1) We describe a composable lakehouse architecture from a programming and execution model perspective, showing how expressing all workloads as functions provides a simple and consistent abstraction for users and the platform alike.
(2) We formalize the scheduling problem in this setting and outline the key requirements for any viable solution.
(3) We introduce Eudoxia as a modular, open-source simulator: we detail our design choices, demonstrate typical usage patterns and provide preliminary validation using standard OLAP workloads against cloud production systems.

While Eudoxia's development was motivated by Bauplan's architecture, we release it to the community[1] with a permissive license because we believe its impact to be potentially broader – either directly as a pluggable module in similar data systems, or indirectly through its abstractions and design principles.

The paper is organized as follows. In Section 2, we introduce background on composable DLHs, which serves as the main motivation for this work; Section 3 describes the scheduling problem in detail and presents the high-level structure of the proposed system; Section 4 illustrates how to invoke and run the simulator, how to configure parameters for Eudoxia , how to register custom scheduling algorithms, and how we validated our approach to have confidence in the results produced by the simulator. We conclude by positioning our work in the context of the existing literature (Section 5) and of future developments (Section 6).

*These authors contributed equally: JT led ideation, TS design and implementation. JT, TS, CG all contributed to the final draft.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

[1] https://github.com/BauplanLabs/eudoxia

2 BACKGROUND AND MOTIVATION

The flexibility of serving interactive and batch use cases for both analytics focused (SQL) and AI focused (Python) runtimes is a distinctive feature of DLHs. To motivate the need for a new scheduler simulator, we walk backwards from the developer experience designed to simplify user interaction with heterogeneous workloads (Section 2.1), and from the corresponding architectural choices (Section 2.2): as we shall see, the Bauplan DLH is composable and modular at the logical level too.

2.1 Writing everything as a function

In contrast with the hard to learn, difficult to debug Big Data (e.g. Spark [32, 36]) and DAG frameworks (e.g. Airflow [38]), coding in Bauplan does not require learning new programming concepts. In particular, data computation can only be expressed through (SQL or Python) functions with signature *Table(s)* -> *Table* – environment variables are either passed as runtime argument (e.g. `bauplan run --namespace xxx`) or stored next to the code itself.[2]

We illustrate this with a concrete example. Consider the Bauplan data pipeline comprising following two files, `parent.sql` and `children.py`:

```
-- bauplan_name parent
SELECT col_1, col_2, col_3 FROM raw
```

Listing 1: parent.sql

```
@bauplan.model()
@bauplan.python("3.10", pip={"pandas": "1.5"})
def child(data=bauplan.Model("parent") ):
  return data.do_something()

@bauplan.model(materialize=True)
@bauplan.python("3.11")
def grand_child( data=bauplan.Model("child")):
   return data.do_something()
```

Listing 2: children.py

Pipelines are simply DAGs of functions chained together by naming convention: the first function to be run is `parent.sql`, whose input is a *Table* called `raw`, which is stored in object storage and registered in the system catalog. The output of this function is also represented as a *Table*. This approach is connected to the *dbt* framework, which pioneered this "functional" approach for data analysts chaining SQL queries together. The second function, `child`, contained in the Python file, takes the parent query as input and produces a new *Table*, which is in turn the input of the final function. Users interact with data and compute declaratively, using *Bauplan tables* over *data branches*, which are semantic, git-like abstractions over Apache Iceberg tables. As such, the underlying catalog and the data files are abstracted away from users.

We find three major types of interactions between users and DLHs, presented in (roughly) descending order of expected latency:

(1) *batch data pipelines*: Usually scheduled, these pipelines combine SQL and Python steps and are used in production environments. They prioritize throughput over latency, as no user is actively waiting for results.
(2) *iterative data pipelines*: Triggered during development or debugging, these pipelines benefit from fast feedback loops to improve developer productivity. While not latency-critical in production terms, delays here can slow down iteration speed and increase cognitive load.
(3) *interactive queries*: Often issued by analysts or business users in SQL or Python, these queries demand low latency and quick feedback. They represent the "live" interface with data and typically require fast, responsive infrastructure.

Because functions can read directly from base tables or from the outputs of other functions, each of the above interactions is representable by composing together Python and SQL blocks with specific signatures.

While the functional abstraction may seem limiting at first, it enables two critical features. First, it lowers the barrier to begin using the system, allowing for example interns with no prior cloud experience to push pipelines to production in their first day of work. Second, within the architecture executing pipelines boils down to orchestrating atomic blocks with the same shape and signature, "only" differing by priority.

2.2 Running everything as a function

Users often must use different interfaces to execute each of the different types of DLH workloads (batch, iterative, and interactive) as described in Section 2.1. For example, a user may run a query in a SQL editor supported by a data warehouse, develop in a notebook (supported by a Spark cluster and Jupyter server), and run pipelines as a Spark script on a schedule (supported by a `submit job` API, a cluster, and an orchestrator).

Table 1 summarizes the distinctive, composable nature of a code-first lakehouse. The uniformity at the developer experience level is mirrored by uniformity at the infrastructure level, where *all* interactions are served by composing together containerized functions over object storage as shown in Fig. 1. Due to several optimizations [28], ephemeral functions spawn in milliseconds inside off-the-shelf virtual machines (VMs), which greatly simplifies the life-cycle management of containers. Even system-level actions, such as checking out a data branch, reading from parquet files to serve a SQL query, or materializing a result back into the catalog, are written as functions and are added to the user-specified DAG by a logical planner [31]. In other words, any task executed on Bauplan is a DAG of system-provided and user-specified ephemeral functions in the view of both the user and the system. No container, warehouse, or engine exists before or after a request, as any resources or state are spawned on demand. Indeed, even the additional bidirectional communication required at the end of more interactive workloads are achieved by running an Arrow Flight server as an ephemeral container in the same model as all other functions.

This architecture reframes DLH scheduling as the problem of orchestrating functions onto pools of resources with varying latency requirements. We find that the most critical insights we've gained from both intuition and empirical evidence align with results

[2]To make this work self-contained, we briefly survey here the relevant DLH pieces. For a fuller background picture on the cloud architecture and not the developer experience, please see [28].

shared from the systems community: *first*, interleaving interactive and non-interactive workloads end up being more computational efficient than separating these workloads onto different systems (i.e. running a query on a warehouse and a pipeline on a Spark cluster) [25, 35]; *second*, using functions as building blocks nudges users to write small, re-usable code that is easier for them to maintain, and importantly easier for the scheduler to reason about [13]. Importantly, existing FaaS schedulers cannot be re-used *as is* because these systems–e.g. AWS Lambda [1], Azure Functions [2], OpenWhisk [3]–are designed to support the execution of simple, fast, stateless, standalone functions with small output sizes.

However, the uniformity of the function interface comes with a trade-off: limited horizontal scaling. While this interface easily supports long pipelines, cross-host communication, and vertical scaling of individual functions [28], each function remains the unit of scheduling and cannot be split across multiple VMs. In traditional big data systems, this has been seen as a limitation, but in practice, many modern workloads can be handled comfortably within a single high-memory VM due to the sharp drop in memory costs (e.g., 1TB fell from $4K in 2014 to $1K in 2023 [21]) and the relatively stable size of analytical datasets (i.e. most OLAP workloads today are under 250GB at the *p99.9th* percentile [34]). This perspective reflects a broader shift toward what some have called "Reasonable Scale" [20, 26, 27], a pragmatic approach that favors simplicity and efficiency over aggressive horizontal scaling at all costs.

In conclusion, we can now see how Bauplan's function-first approach benefits both users and developers. Using system and user functions as the building blocks of the runtime allows a granular understanding of workloads and provides many opportunities to interleave different workload types depending on their latency requirements. Enabling granular scheduling—pausing and resuming DAGs mid-air, moving functions between hosts etc.—is a desired consequence of our architecture but presents a challenge of finding an effective, if not optimal, scheduling algorithm. Considering a DLH that uses a single runtime makes the problem significantly more tractable and simplifies modeling the platform, but there is still a need to evaluate different scheduling algorithms. This motivates our scheduling simulator, *Eudoxia*, which we now discuss.

Table 1: Interaction types, user interfaces and infrastructure requirements for different DLH designs.

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
FaaS DLH		
Batch pipeline	`bauplan run`	Functions
Dev. pipeline	`bauplan run`	Functions
Inter. query	`bauplan query`	Functions

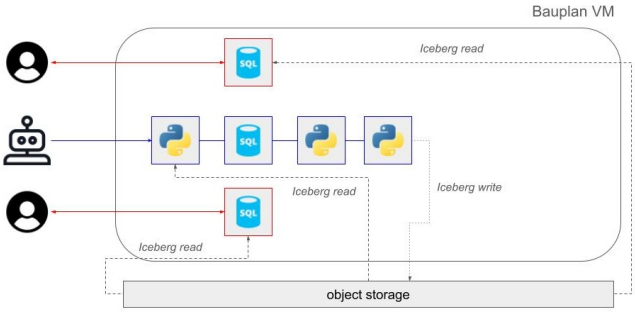

Figure 1: Bauplan workers are off-the-shelf VMs, providing stateless compute capacity over object storage. Within an organization, users and machines (Apache Airflow, AWS Lambda on a schedule etc.) may submit interactive read-only queries (red) or asynchronous read-write pipelines (blue). What scheduling policy for functions can maximize a desired metric (e.g. throughput)?

3 SIMULATOR DESIGN

We first provide an overview of the motivation and goals behind the simulator in Section 3.1 before discussing the design and major abstractions of the simulator in Section 3.2.

3.1 Overview

A composable lakehouse can be tested in a variety of ways, from cheap but case-based unit and integration tests to very expensive but general formal methods. Simulations rely on a deterministic model of the system (like formal methods) but are low-cost and experimentally driven (like integration tests) and thus are a promising way to evaluate scheduling policies in a complex cloud setup.

As a system providing a FaaS interface, implementing an efficient and effective scheduling algorithm is vital as users will be submitting pipelines without any consideration for hardware or scheduling. Thus our scheduling simulator must be able to evaluate different scheduling algorithm implementations both in terms of performance metrics (e.g. throughput and latency) as well as monetary cost (e.g. excess cloud resources or premium storage). A successful solution to this problem will therefore be able to give us confidence over scheduling policies without spending the time and money to evaluate the same policies in a real cloud environment.

3.2 Design Principles and Major Abstractions

We now present the architecture for our proposed solution. This design, shown in Figure 2, is *modular* to be able to test any scheduling algorithm, to allow for workload customization via parameters set by developers, and to support alternate executor models. We decompose this design into three components. Our simulator operates as a high-level loop, and during each iteration each of these three components complete whatever work is possible for them. Each iteration represents 1 CPU tick or approximately 10 microseconds.

3.2.1 Workload Generation. In a real setup, various users submit pipelines to the system at random intervals. The workload generator simulates this part of the system by generating pipelines

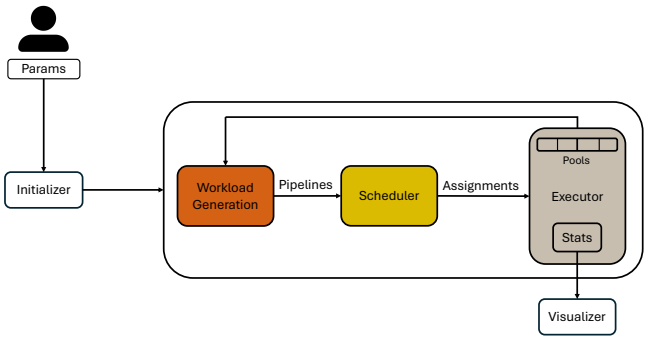

Figure 2: Simulator Architecture. Users set parameters and pass this to the initializer for Eudoxia which starts a loop of three components, the Workload Generator, Scheduler, and Executor. Once that loop completes, visualizers or other downstream applications can access execution statistics.

and sending them to the system at user-defined intervals to be scheduled and executed. The workload generator accepts a wide range of parameters which specify how frequently new pipelines arrive, how many resources pipelines require, how long pipelines will take to complete depending on the physical resources (RAM and CPU) allocated to them, among others. Full documentation is available with our artifact. Additionally, this interface allows users to format existing traces and feed them into the simulator rather than generating random ones.

We model user-submitted pipelines as directed acyclic graphs (DAGs). Each node in a pipeline is called an *operator*, which represents individual functions such as SQL queries or Python functions. Each operator is generated with some required amount of RAM to execute, representing the largest allocation of memory the operator will require to complete. Each operator is also generated with a CPU scaling function, which returns how long the operator will take to complete based on how many CPUs are allocated (for example, a heavy IO task may not scale with CPUs at all, while a stateless filter can scale linearly with more CPUs). Any value associated with a pipeline is randomly drawn from a distribution centered at one of the user-provided (or system default) parameters. Modeling pipelines as DAGs allows for user jobs to be arbitrarily split up by the scheduler to enable fine-grained parallelism, which may be helpful to ensure that large pipelines with many parallel operators with heavy RAM requirements can execute.

Finally, each pipeline has one of three Priority Levels, based on the DLH scenarios described in Section 2.1: in ascending order, we have *batch data pipelines*, *iterative data pipelines*, *interactive query*. At each tick when pipelines are generated, they are passed to the scheduler. For most ticks, no new pipelines will be generated.

3.2.2 Executor. The user also specifies how many CPUs and RAM are available to allocate to jobs and whether more resources can be accessed for additional monetary cost, i.e. using cloud scaling. The user can specify how many pools of resources there are, what the balance of resources are in each pool, and so on.

The executor manages these simulated physical resources. We define an abstraction, the *Container*, which holds a set of *Operators* to execute, a number of CPUs, and amount of RAM. When created each container uses the set of operators provided to calculate how many ticks it will for that container to complete or how many ticks before it will trigger an out-of-memory error based on the parameters in the workload generator and the resources allocated.

The executor can also be extended to simulate more complex realities of FaaS system deployment. For instance, cold starts or network variability can be supported by simply adding a random number ticks, drawn from some known distribution that reflects real system observations, to the execution time of a Container. Spot instance revocation can be supported by adding a random chance of sudden failure to a Container, using the same mechanisms as failures or preemptions. The base design of the Executor is flexible enough to support as rich of a setup as necessary.

3.2.3 Scheduler. The Scheduler allocates resources to sets of Operators (as the Scheduler can subdivide pipelines in allocation) and instructs the Executor what Containers to create or preempt, i.e. terminate to free up resources. The Scheduler manages queues, makes allocation decisions, decides what jobs receive allocations sooner than others, and how priority levels are managed.

Each scheduler implementation must simply match a required type signature: accepting a set of Pipelines from the workload generator, and outputting a list of new Container allocations and Container preemptions to the Executor. At runtime, the user will register different scheduler implementations with the simulator and specify which one it should use during execution.

4 EUDOXIA 101

In this section we will describe how users interact with Eudoxia, provide a sample program and a short description of key parameter options. Then we present a preliminary validation of our simulator approach using real traces executed on Bauplan.

4.1 Developer experience

We first present how users would start a new Eudoxia instance (Section 4.1.1) before presenting the scheduling algorithms already implemented (Section 4.1.2) and how users can write and register their own implementations (Section 4.1.3).

4.1.1 Starting a New Instance. We aimed to make it as easy as possible to start working with Eudoxia's API. To start a simulator instance, users specify input parameters and select a scheduler implementation, either one of the three scheduler algorithms already implemented or a custom implementation written in Python and registered at runtime. Parameters are set in a *TOML* file, with each parameter in its own line formatted as *parameter = value*. The most important parameters here are the following:[3]

- duration: how many simulated seconds the simulator will run for. Each iteration of the primary loop corresponds to 10 microseconds, intended to roughly approximate the length of 1 CPU cycle. We call each iteration a **tick**.
- waiting_ticks_mean: on average how many ticks (10 microseconds) pass between pipelines being generated and sent to the system to be executed.

[3]Full documentation is available with the code artifact.

- NUM_POOLS: how many resource pools will exist. In general all available resources are split evenly to start.
- SCHEDULING_ALGO: what scheduling algorithm to use.

Here is how easy is to start an instance: RUN_SIMULATOR instantiates EUDOXIA with the parameters in PROJECT.TOML:

```
import eudoxia
def main():
    paramfile = "project.toml"
    eudoxia.run_simulator(paramfile)
```

Listing 3: Minimal code to start a simulation

The RUN_SIMULATOR method will then begin the core loop described in Section 3, containing the workload generation, scheduler, and executor. EUDOXIA will use the DURATION parameter to compute the number of iterations the loop runs for and will pass each parameter to its appropriate component(s). Once EUDOXIA is launched, each component will log its current actions, and CPU and RAM utilization will be logged after each tick for each pool of resources.

4.1.2 New Scheduling Protocols. EUDOXIA has three built-in implementations for schedulers.

The first is the NAIVE scheduler, which uses one pool of resources. It assigns all available resources to the next pipeline. When that pipeline completes, it repeats with the next pipeline in the queue.

The next is the PRIORITY scheduler, which also assumes one pool of resources. It accounts for both the size of the pool and the priority of pipeline that was submitted (either BATCH, QUERY, or INTERACTIVE). New workloads are assigned a container with 10% of the *total* amount of resources. The scheduler proceeds until it has allocated all resources. If a pipeline completes, those resources are allocated to the next pipeline in the queue.

If a pipeline fails due to insufficient resources, i.e. an out-of-memory (OOM) error, then those resources are freed but the pipeline re-enters the waiting queue of the scheduler with information about what resources were allocated to the container which failed.

If a previously-failed pipeline arrives, the scheduler attempts to double the resources previously allocated up to a maximum of 50% of total CPU or RAM, at which point the scheduler returns the failure to the user. If there are not sufficient available resources to double the allocation, the job is put back on the queue to wait.

Finally, in the event that all resources are allocated and a high priority pipeline, such as a QUERY, arrives, the scheduler scans the currently running containers for any which are running a low-priority job (such as a BATCH workload). That container is preempted, freeing its resources to be used for the QUERY. The BATCH pipeline is put back on the waiting queue with a log of what resources were last allocated to it; however, this pipeline does not also receive the flag indicating it failed. So when the BATCH pipeline next arrives the scheduler will allocate the *same* resources it allocated previously.

The third scheduling algorithm is the PRIORITY-POOL scheduler. This operates similarly to the PRIORITY scheduler but with multiple resource pools in the Executor. Every time the Scheduler considers a new pipeline, it identifies which pool has the most available resources and allocates a container on that pool. It also handles preemption in the same way, but this time on multiple pools.

4.1.3 Registering New Scheduler Implementations. EUDOXIA lets users write custom schedulers by writing an initialization function, writing a scheduler function, and using two decorators.

The *initialization function* accepts an instance of the SCHEDULER class and returns nothing. This function initializes any needed data structures within the SCHEDULER. The *scheduler function must* accept three parameters and return two values. The parameters are:

(1) An instance of the SCHEDULER class.
(2) A list of pipelines which failed in the previous tick
(3) A list of pipelines which were newly created in this tick

The list of newly created pipelines is often empty, as the workload generation step creates new pipelines at random intervals. Similarly, the list of failures only includes jobs which the executor failed, such as for an OOM error. This does not include pipelines which the scheduler preempted. If the scheduler wishes to preempt pipelines it must manage those queues itself to ensure no pipelines fall through the cracks. Finally, the algorithm must return two values:

(1) SUSPENSIONS: these are a set of pipelines that the scheduler is instructing the Executor to preempt so that its resources may be freed. For the priority scheduler, it places pipelines to be preempted in an internal SUSPENDING queue, which after one tick it moves back into the standard waiting queues.
(2) ASSIGNMENTS: the second return value is a list of new assignments instructing the Executor what resources to allocate to a container and what job to run inside that container.

Putting these requirements together, extending EUDOXIA with a custom scheduler is as simple as the snippets below – note how the *two decorators* in ALGORITHM.PY and the parameter in PROJECT.TOML reference the same *key*:[4]

```
from eudoxia.core import Scheduler
from eudoxia.core import Failure, Assignment, Pipeline
from eudoxia.algorithm import register_scheduler,
    register_scheduler_init
from typing import List

@register_scheduler_init(key="my-scheduler")
def scheduler_init(sch: Scheduler):
    ...

@register_scheduler(key="my-scheduler")
def scheduler_algo(sch: Scheduler, f: List[Failure], p:
    List[Pipeline]):
    ...
    return suspends, assignments
```

Listing 4: algorithm.py: *scheduler function.*

```
scheduling_algo = "my-scheduler"
```

Listing 5: project.toml: custom parameter.

```
from algorithm import scheduler_init, scheduler_algo
import eudoxia
def main():
    paramfile = "project.toml"
    eudoxia.run_simulator(paramfile)
```

Listing 6: main.py: custom imports and instantiation.

[4]Users should import SCHEDULER_INIT and SCHEDULER_ALGO in MAIN.PY first so that the decorators register the keys before the instance starts.

4.2 Preliminary Validation

While developer experience, clarity in abstractions and extensibility are crucial aspects for its adoption, Eudoxia's utility depends on reliability and robustness.

The simulation generates pipelines which have two key values: (1) how the number of CPUs allocated impacts the pipeline's execution time (if at all) and (2) the minimum RAM allocation needed to avoid an out-of-memory error. The scheduling algorithms do not have access to these values or scaling functions; however, once the pipeline is allocated to a container of resources, those values are used to determine what the true execution time for a pipeline on a container will be. We believe that this is a realistic setup that can effectively represent any kind of workload in the appropriate and necessary dimensions.

We first validate this approach by running data workloads against a Bauplan cloud instance, measuring runtime statistics such as CPU and RAM utilization along with runtime and comparing this to the runtime estimated by Eudoxia on a pipeline with similar statistics. We run the common data analytics benchmark TPC-H [33] and run its 22 queries against a 10GB dataset on a instance running on an AWS c5ad.4xlarge instance with 16 vCPUs and 32GB of RAM. As described, each query is compiled by Bauplan into a small number of execution blocks (i.e. functions), and we observe CPU and RAM usage during execution. Each query is run alone on the instance, and we disable caching. For three queries (11, 16, and 22), the runtime was so short that resource utilization statistics could not be gathered from underlying telemetry systems. The percent error in runtime of the scheduler versus the true runtime as executed on a Bauplan instance ranges from 0.44% to 3.08% with an average of 1.74% error. We additionally plot the real and simulated runtimes for a subset of TPC-H queries for ease of visual interpretation in Figure 3. We see that the simulated runtime well approximates the true execution time, indicating that the simulator can be relied upon to give realistic results.

Furthermore, because Eudoxia supports varying CPU scaling functions and enables real traces to be plugged in rather than using random generation, the system can easily emulate how the benchmark's performance would vary if different compute resources are allocated or if the benchmark ran on larger datasets. The modular design enables users to reproduce other results cheaply, test how algorithms would hold up against different types of workloads, or consider how a current implementation would fare against a changing setup.

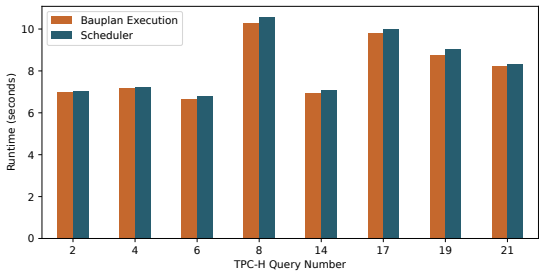

Figure 3: Distribution of percent error of simulator estimates for runtime vs. real runtime for TPC-H queries at 10GB.

5 RELATED WORK

Composable data systems. The FaaS lakehouse modeled by Eudoxia is built in the *composable data system* tradition [23]. In a sense, the deconstructed lakehouse [31] is the natural generalization of the "Deconstructed Database" [14]. The rapid growth of *DataFusion* [15] in the composable data community is fostering an eco-system of novel single-node systems [4, 5] that could benefit from the simulation methodology and code in Eudoxia .

Cloud Scheduling. There is a broad range of work in the realm of scheduling or scheduling workloads in cloud environments that is relevant to Eudoxia . Motlagh et. al. [9] provides an analytical framework to evaluate different scheduling approaches. Similarly, Hai et. all [12] proposes a new scheduling approach but does so with a broad cloud usage pattern in mind. In contrast, Eudoxia is designed for specifically a data lakehouse/composable data system environment. Rather than trying to survey a range of approaches or techniques, Eudoxia focuses specifically on an application deployment on a single Bauplan instance on an EC2 node.

Another common goal for schedulers is to abide by quality-of-service (QoS) guidelines. While this is a vital part of the cloud ecosystem, our goal behind Eudoxia was to consider, experiment, and use that simulator to evaluate future scheduling approaches.

Finally, a broad range of literature covers scheduling under power constraints. Specifically, as power-constrained applications throttle query performance in some instances as they limit CPU frequency, etc. There is a broad range of work in this area, including [7, 8, 11, 16, 22]. However, Eudoxia is generally uninterested in how power consumption limits resources availability and workload runtime, as blob storage and VM services offered by cloud vendors abstract away the power consumption needs for cloud infrastructure.

6 CONCLUSION

In this paper, we described a composable lakehouse, *Bauplan*, from the perspective of its programming and execution model, which reduced the main use cases to scheduling functions with different priorities. While the developer experience gets simplified, we face greater optimization challenges compared to general purpose FaaS systems due to our target workload, which prevented us from reusing existing FaaS schedulers. As simulations are a cost-effective ways to test cloud distributed systems, we introduced our simulator, Eudoxia , to enable robust but cheap evaluation of the effect of different scheduling algorithms. Through concrete examples, we described Eudoxia design principles and developer experience, and we provided a preliminary quantitative validation running standard OLAP benchmarks in production and in the simulator.

Eudoxia is easy to extend and applicable beyond its initial simulation scope. For example, plugging real-world scaling functions estimated from traces is trivial and may be useful for other use cases. In the same vein, while most of our simulations are single pool because of Bauplan's design, benchmarking functions across capacity pools is a useful feature in distributed systems. We release the code to the community with a permissive license as we believe our abstractions and lessons – if not Eudoxia itself – to be of use to the broader composable data system community.

REFERENCES

[1] 2024. AWS Lambda. https://docs.aws.amazon.com/lambda/latest/api/API_Invoke.html

[2] 2024. Azure Functions. https://azure.microsoft.com/en-us/products/functions/.

[3] 2024. Open Whisk. https://github.com/apache/openwhisk.

[4] 2025. Arroyo. 2023. Arroyo - Serverless Stream Processing. https://www.arroyo.dev/.

[5] 2025. InfluxDB: open source time series, metrics, and analytics database. https://influxdata.com/.

[6] Apache Arrow Flight [n.d.]. Arrow Flight RPC — Apache Arrow v20.0.0. https://arrow.apache.org/docs/format/Flight.html

[7] Peter E. Bailey, Aniruddha Marathe, David K. Lowenthal, Barry Rountree, and Martin Schulz. 2015. Finding the limits of power-constrained application performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 79, 12 pages. https://doi.org/10.1145/2807591.2807637

[8] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N. Calheiros. 2009. Modeling and simulation of scalable Cloud computing environments and the CloudSim toolkit: Challenges and opportunities. In *2009 International Conference on High Performance Computing and Simulation*. 1–11. https://doi.org/10.1109/HPCSIM.2009.5192685

[9] Carlo Curino, Evan P.C. Jones, Samuel Madden, and Hari Balakrishnan. 2011. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 313–324. https://doi.org/10.1145/1989323.1989357

[10] Darren Dao, Jeannie Albrecht, Charles Killian, and Amin Vahdat. 2009. Live Debugging of Distributed Systems. In *Compiler Construction*, Oege de Moor and Michael I. Schwartzbach (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 94–108.

[11] Neha Gholkar, Frank Mueller, and Barry Rountree. 2016. Power Tuning HPC Jobs on Power-Constrained Systems. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (Haifa, Israel) *(PACT '16)*. Association for Computing Machinery, New York, NY, USA, 179–191. https://doi.org/10.1145/2967938.2967961

[12] Tao Hai, Jincheng Zhou, Dayang Jawawi, Dan Wang, Uzoma Oduah, Cresantus Biamba, and Sanjiv Kumar Jain. 2023. Task scheduling in cloud environment: optimization, security prioritization and processor selection schemes. *Journal of Cloud Computing* 12, 1 (2023), 15.

[13] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 295–308.

[14] Amandeep Khurana and Julien Le Dem. 2018. The Modern Data Architecture The Deconstructed Database. (2018).

[15] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 5–17. https://doi.org/10.1145/3626246.3653368

[16] Weiwei Lin, Siyao Xu, Ligang He, and Jin Li. 2017. Multi-resource scheduling and power simulation for cloud computing. *Information Sciences* 397-398 (2017), 168–186. https://doi.org/10.1016/j.ins.2017.02.054

[17] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M Frans Kaashoek, and Zheng Zhang. 2008. D3S: Debugging deployed distributed systems. In *NSDI*.

[18] Semih Salihoğlu Luca Bigon, Jacopo Tagliabue. 2025. DAG lakehouse planning with an ephemeral and embedded graph database. *Proceedings of Workshops at the 51th International Conference on Very Large Data Bases* (2025).

[19] Dipankar Mazumdar, Jason Hughes, and JB Onofre. 2023. The Data Lakehouse: Data Warehousing and More. arXiv:2310.08697 [cs.DB] https://arxiv.org/abs/2310.08697

[20] Frank McSherry, Michael Isard, and Derek Gordon Murray. 2015. Scalability! But at what COST?. In *USENIX Workshop on Hot Topics in Operating Systems*.

[21] Our World in Data. 2024. Historical price of computer memory and storage. https://ourworldindata.org/grapher/historical-cost-of-computer-memory-and-storage?time=2010..latest&facet=metric.

[22] Tapasya Patki, Zachary Frye, Harsh Bhatia, Francesco Di Natale, James Glosli, Helgi Ingolfsson, and Barry Rountree. 2019. Comparing GPU Power and Frequency Capping: A Case Study with the MuMMI Workflow. In *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. 31–39. https://doi.org/10.1109/WORKS49585.2019.00009

[23] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *Proc. VLDB Endow.* 16, 10 (June 2023), 2679–2685. https://doi.org/10.14778/3603581.3603604

[24] Jacopo Tagliabue Ryan Curtin. 2025. The Deconstructed Warehouse: An Ephemeral Query Engine Design for Apache Iceberg. *Proceedings of Workshops at the 51th International Conference on Very Large Data Bases* (2025).

[25] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, Andrii Golovei, Pradeep Venkat, Andrew Mcfague, Dimitrios Skarlatos, Vipul Patel, Ravinder Thind, Ernesto Gonzalez, Yun Jin, and Chunqiang Tang. 2023. XFaaS: Hyperscale and Low Cost Serverless Functions at Meta. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 231–246. https://doi.org/10.1145/3600006.3613155

[26] Jacopo Tagliabue. 2021. You Do Not Need a Bigger Boat: Recommendations at Reasonable Scale in a (Mostly) Serverless and Open Stack *(RecSys '21)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3460231.3474604

[27] 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).

[28] Jacopo Tagliabue, Tyler Caraza-Harter, and Ciro Greco. 2024. Bauplan: Zero-copy, Scale-up FaaS for Data Pipelines. In *Proceedings of the 10th International Workshop on Serverless Computing* (Hong Kong, Hong Kong) *(WoSC10 '24)*. Association for Computing Machinery, New York, NY, USA, 31–36. https://doi.org/10.1145/3702634.3702955

[29] Jacopo Tagliabue, Ryan Curtin, and Ciro Greco. 2024. FaaS and Furious: abstractions and differential caching for efficient data pre-processing . In *2024 IEEE International Conference on Big Data (BigData)*. IEEE Computer Society, Los Alamitos, CA, USA, 3562–3567. https://doi.org/10.1109/BigData62323.2024.10825377

[30] 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

[31] 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

[32] Shanjiang Tang, Bingsheng He, Ce Yu, Yusen Li, and Kun Li. 2022. A Survey on Spark Ecosystem: Big Data Processing Infrastructure, Machine Learning, and Applications. *IEEE Transactions on Knowledge and Data Engineering* 34, 1 (2022), 71–91. https://doi.org/10.1109/TKDE.2020.2975652

[33] TPC-H [n.d.]. TPC-H Homepage. https://www.tpc.org/tpch/

[34] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil 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. *Proc. VLDB Endow.* 17, 11 (July 2024), 3694–3706. https://doi.org/10.14778/3681954.3682031

[35] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 18, 17 pages. https://doi.org/10.1145/2741948.2741964

[36] Zehao Wang. 2021. Understanding the Challenges and Assisting Developers with Developing Spark Applications. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (2021), 132–134.

[37] Michael Whittaker, Cristina Teodoropol, Peter Alvaro, and Joseph M. Hellerstein. 2018. Debugging Distributed Systems with Why-Across-Time Provenance. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 333–346. https://doi.org/10.1145/3267809.3267839

[38] 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

[39] 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*.