



A Data Guide to AWS on oblako



Almostly

Denis Burakov



oblako

A Data Guide to AWS on oblako

Denis Burakov

FIRST EDITION, 2026

A Data Guide to AWS on oblako

Copyright © 2026 Denis Burakov. All rights reserved.

Published by Almostly.

oblako is open-source software developed by Almostly and released under the Apache License, Version 2.0. The book is independent of AWS; AWS and the names of its services are trademarks of Amazon.com, Inc.

Cover: “Cumulostratus”, a lithograph by E. M. Williams from Luke Howard’s *Essay on the Modifications of Clouds*, third edition, London, 1865; public domain.



`almostly.ai`

Contents

Introduction	1
Part I oblako	4
1 What is oblako?	5
2 Running oblako	16
Part II The data services	25
3 S3	26
4 S3 Tables, Iceberg and S3 Vectors	40
5 The Glue Data Catalog and Athena	51
6 Glue jobs	62
7 Redshift	71
8 RDS and Aurora	84
9 Replicating PostgreSQL	96
10 DynamoDB	115
11 Streaming with Kinesis and Firehose	129
Part III Data engineering tools	145
12 Spark with Iceberg	146

13	A multi-node Redshift	161
14	Feature Store with Feast	174
15	Orchestration with MWAA	185
Part IV Practice		196
16	Testing pipelines in CI	197
17	Infrastructure as code	210
18	Moving to AWS	223
19	Conclusion	229
Appendices		231
A	Other services	231

Introduction

This book is a guide to the AWS data services as they run on oblako, which runs those services on your own machine. S3 calls reach a real object store, Redshift SQL runs in a real database, and Athena queries run on a real query engine that reads the files. Your code receives the responses these engines produce, and nothing is billed.

You can create a bucket, a table, a warehouse schema and a streaming pipeline in an afternoon, break them and start again, without an AWS account or a credit card and with nothing billable left to clean up. When the code works, you point it at AWS by changing an endpoint and a set of credentials, and the code itself stays the same.

Who this book is for

The book is for data engineers, analytics engineers and data scientists who work with AWS or plan to. It assumes you write Python, know SQL and have used a terminal, but it assumes no prior knowledge of AWS, since each chapter explains its service before it uses it. If you already run these services in production, you can use the book to develop and test against them locally, with more confidence than a mock gives you.

A disclosure

I wrote oblako, which is open source under the Apache 2.0 license with its code public. This book is therefore a companion to software I maintain, and you should read it with that in mind. I have tried to state exactly where oblako behaves like AWS and where it differs. Each chapter on a service or tool has a section on the differences, and every listing that shows something working was run on oblako before it went into the text.

How the book is organized

Part I covers oblako itself: what it is, how a request travels from your code to a real engine, and how to install and run it. Read it first, because the rest of the book assumes you know which engine sits behind which port.

Part II covers the data services one at a time: S3; S3 Tables, Iceberg and S3 Vectors; the Glue Data Catalog and Athena; Glue jobs; Redshift; RDS and Aurora; replicating PostgreSQL; DynamoDB; and streaming with Kinesis and Firehose.

Part III covers the data engineering tools that run on those services: Spark writing Iceberg tables to S3 Tables, a multi-node Redshift built on Citus, a feature store with Feast over Redshift and DynamoDB, and Airflow pipelines on MWAA.

Part IV covers practice: testing a pipeline in CI, describing its infrastructure as code with SAM, Terraform and Pulumi, and running the same code and tests on AWS.

Appendix A covers services outside the book's data focus that oblako also runs: Lambda, Step Functions, EventBridge, Bedrock, ECS and EKS, with one working example each.

The examples

Instead of a running example, each chapter uses small, self-contained data chosen for its own topic, such as a few files for S3, a partitioned table for Athena or a handful of rows for Redshift. Each chapter's code runs on its own, and its listings create their data or a short script next to them does.

Conventions

Code is plain boto3 or the service's own driver, such as `redshift-connector` for Redshift, and listings never call oblako-specific APIs. The only oblako commands in the book are calls to the `oblako` command-line tool in setup steps, so the same code runs on AWS.

Python listings look like this:

```
import boto3

s3 = boto3.client("s3")
print(s3.list_buckets()["Buckets"])
```

and shell commands like this:

```
oblako up s3
```

The listings were run against oblako 0.1.0 from its main branch, most recently at commit 3117e6f. oblako changes quickly, and where a later release behaves differently from the book, its release notes say so.

PART I

oblako



1

What is oblako?

How oblako runs the AWS data services on your machine, and where it differs from AWS.

The name is Russian: *oblako* (облако), stressed on its first syllable, is the word for a cloud.¹ oblako puts a small one on your machine, with the AWS data services running inside it.

1.1 Developing against the cloud

Code that uses AWS is awkward to develop because every experiment touches remote resources, such as a bucket that has to exist, a warehouse that has to be running and a role that has to allow the call. Each try is a deployment, which makes the loop slow, and some services bill from the moment they start, whether you query them or not. A team also usually shares one development account, so your half-finished table sits next to a colleague's.

The usual alternative is a mock, a library that intercepts the call your code makes and returns a response written in advance. Mocks suit unit tests because they are fast, need nothing installed and make a test independent of the network. A mock returns only what its author wrote, though: a mocked S3 keeps your bytes in a Python dictionary, and a

¹The word comes from Old Church Slavonic облакъ (*oblakŭ*). By its parts, об- and волок-, it is related to волочить, "to drag". Wiktionary, "облако", <https://en.wiktionary.org/wiki/облако>.

mocked Redshift accepts `CREATE TABLE` and runs nothing. When your code depends on what the service does, such as how a query engine reads partitioned Parquet or whether your SQL is valid, only a real engine can answer the question.

1.2 Real behavior, simulated topology

oblako's design follows from one principle, *real behavior, simulated topology*.

Every AWS service has two parts. The **engine** does the work of storing the bytes, running the SQL or reading the files. The **topology** is the AWS shape around the engine, which includes the endpoint you call, the cluster or bucket it belongs to, the account and region in its ARN, and the control plane that creates and describes it.

oblako runs a real engine for each service wherever one exists and simulates only the topology around it. S3 is served by S3Proxy, an open-source implementation of the S3 API that writes real files. Redshift is a PostgreSQL 16 database that accepts Redshift's SQL, Athena is the Trino query engine, and DynamoDB is Amazon's own DynamoDB Local. oblako adds the parts that make these engines look like AWS to your code: the endpoints, the cluster metadata, and the APIs that create and describe resources.

oblako sits between a mock and the cloud. Like a mock, it runs on your machine, costs nothing and resets in seconds, and like the cloud, it does the work, so a wrong query fails and a correct query returns the rows.

1.3 What oblako covers

oblako does not try to run all of AWS. Its core is the data toolkit in the frame of Figure 1.1: the storage, catalog, query, warehouse, database, streaming and orchestration services. Each of them is backed by a real engine and is the subject of a chapter in Parts II and III. oblako also runs the compute, integration and AI services that data pipelines call, such as Lambda, Step Functions, EventBridge and Bedrock. moto answers the common services that most code touches, such as IAM, SQS, SNS, Secrets Manager and KMS.

The rest of AWS is outside its scope. A call to one of those services goes to moto, which records it if it implements that service's control plane, with no engine behind it, and rejects it otherwise.

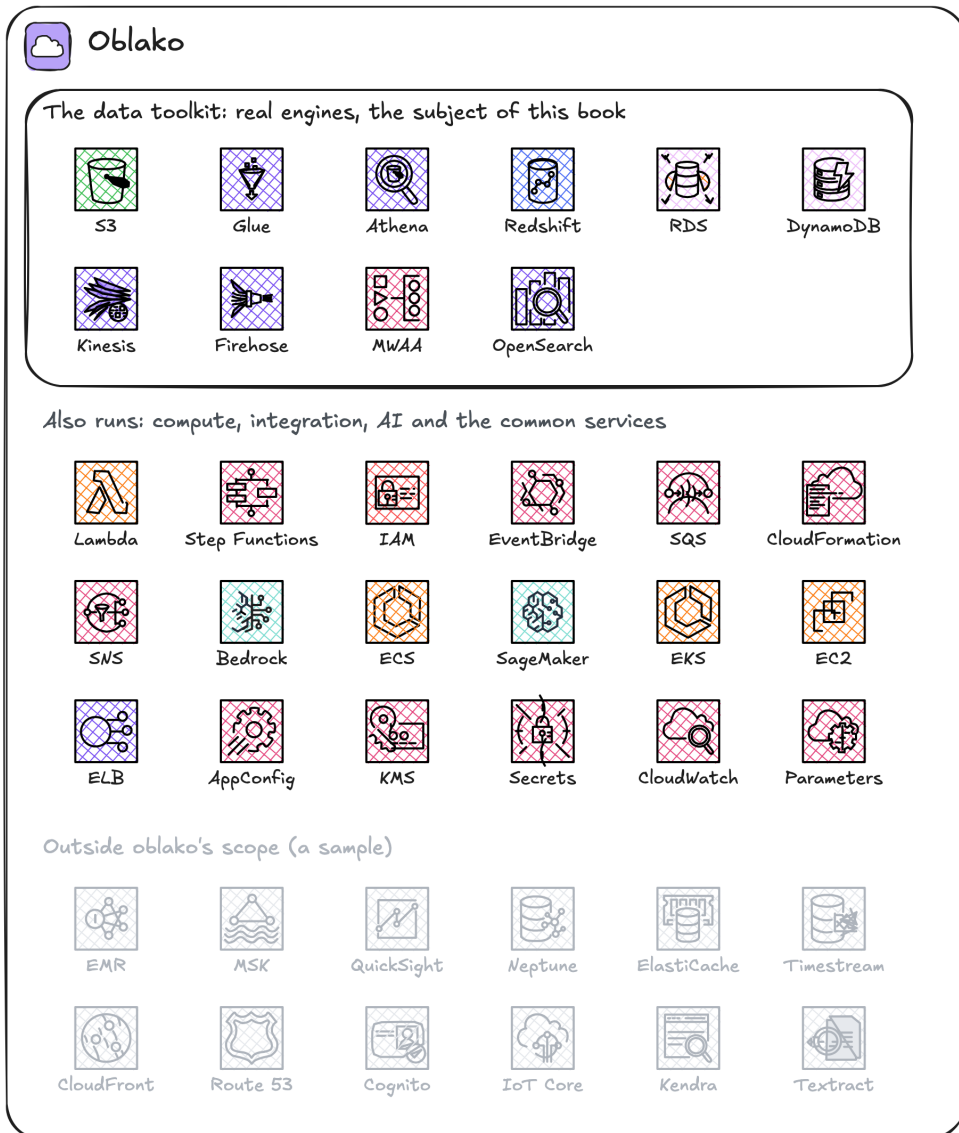


Figure 1.1: What oblako runs. The data toolkit in the frame is the subject of this book. The services below it run too, several through moto's control plane. The faded services are a sample of AWS outside oblako's scope.

1.4 Ports

Each service listens on its own fixed port on localhost, and your code sends its calls there instead of to AWS. The AWS SDKs and the AWS CLI can read a separate endpoint for each service from a profile in their configuration file. oblako writes such a profile, named oblako, with an endpoint for every service it runs and local credentials, and Chapter 2 creates it. Selecting the profile is the only change, and the code itself stays the same:

```
export AWS_PROFILE=oblako
```

With the profile selected, this ordinary boto3 code runs against oblako's S3:

```
import boto3

s3 = boto3.client("s3")
s3.create_bucket(Bucket="raw-data")
s3.put_object(
    Bucket="raw-data", Key="hello.txt", Body=b"hello from oblako"
)

obj = s3.get_object(Bucket="raw-data", Key="hello.txt")
body = obj["Body"].read()
print(body.decode())
```

```
hello from oblako
```

Select one of your AWS profiles instead, and the same script talks to AWS.

Because the ports are fixed, the profile, notebooks and CI jobs can all use the same map. Table 1.1 lists the ports for the services this book uses.

Table 1.1: The local port for each service used in this book.

Service	Port
S3	9000
S3 Tables	8013
S3 Vectors	8012
Glue Data Catalog	8486
Athena	8009

Service	Port
Redshift (SQL)	5439
Redshift Data API	8002
RDS PostgreSQL (shared)	5432
RDS API (instances, replicas)	8014
RDS Data API	8006
DynamoDB	8007
DynamoDB Streams	8001
Kinesis Data Streams	4567
Firehose	8008
Redshift API (clusters)	8015
Control planes (IAM, Lambda, EC2)	5500

1.5 Request paths

Figure 1.2 shows the path a call takes. Your code sends the call, unchanged, to a port. Behind each port is one of four things: the engine itself, a thin front that adds what the engine lacks, a Python engine that oblako provides, or the control plane.

Direct

In the direct case the client talks to the engine with nothing in between. Kinesis Data Streams is kinesalite on port 4567, which your boto3 Kinesis client calls directly, and DynamoDB Streams is served by DynamoDB Local itself, on port 8001. When an engine already speaks the AWS API, oblako's only job is to start it on the right port.

Fronted engines

Some engines implement only part of the AWS API. S3Proxy stores and serves objects but lacks object tagging, S3 Inventory and event notifications, so port 9000 is a stock nginx in front of S3Proxy. nginx passes ordinary requests to S3Proxy on port 9001, sends tagging and Inventory requests to a small oblako engine on port 8020, and logs completed writes so that notifications can fire. Your client sees one S3 endpoint.

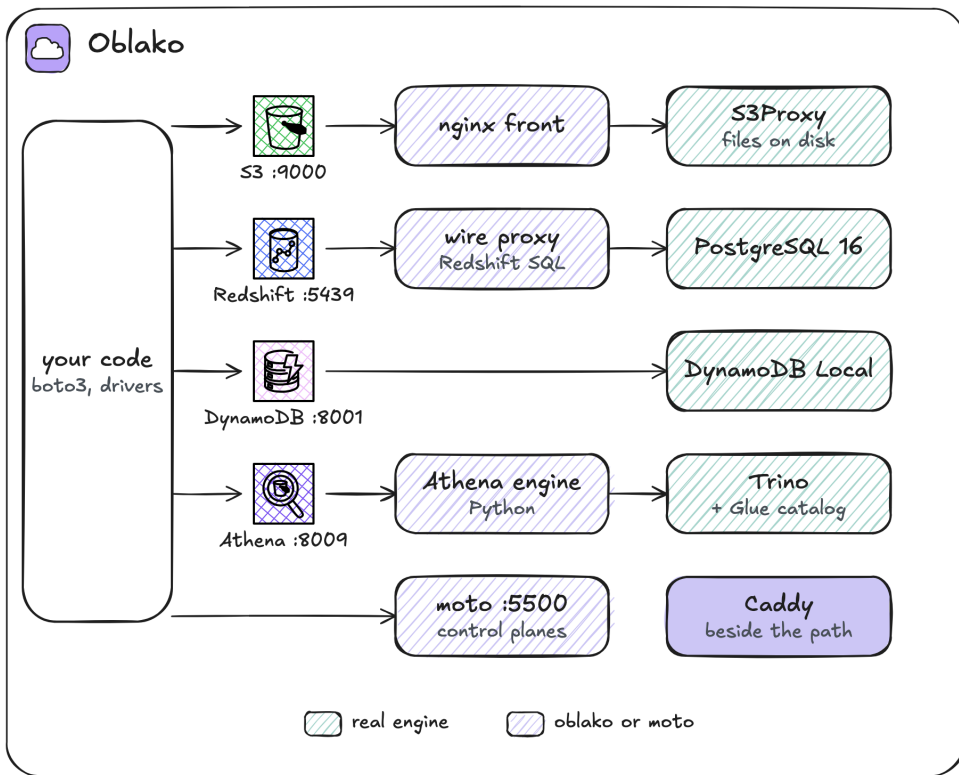


Figure 1.2: How a call travels from your code to an engine. Each icon marks the AWS service a port plays; teal boxes are real engines, and lavender boxes are oblako's own code or moto, which give the engines their AWS shape.

DynamoDB has a front of the same kind. Port 8007 is an oblako proxy that forwards every call to DynamoDB Local on port 8001 and adds the operations DynamoDB Local lacks: vector indexes, SearchVectors, and tags.

Redshift needs a different kind of front, because PostgreSQL rejects Redshift-only syntax such as `DISTKEY` and `SORTKEY` while it parses a statement, before any code inside the database runs. The Redshift image therefore carries a wire proxy that listens on Redshift's port, 5439, terminates TLS, rewrites Redshift SQL into SQL PostgreSQL accepts and passes everything else through byte for byte. The proxy also turns `COPY` and `UNLOAD` into reads and writes against oblako's S3, and your driver connects to it as it would to a single Redshift endpoint:

```

import redshift_connector

conn = redshift_connector.connect(
    host="localhost",
    port=5439,
    database="oblako",
    user="oblako",
    password="oblako",
    ssl=True,
    sslmode="verify-ca",
)
conn.autocommit = True
cur = conn.cursor()
cur.execute(
    "CREATE TABLE IF NOT EXISTS stock"
    " (sku varchar(16), on_hand int)"
    " DISTKEY(sku) SORTKEY(sku)"
)
cur.execute("SELECT split_part(version(), ' on ', 1)")
print(cur.fetchone()[0])

```

PostgreSQL 16.15 (Debian 16.15-1.pgdg13+2)

The `CREATE TABLE` with its distribution and sort keys succeeds because the proxy removed the Redshift-only clauses before PostgreSQL parsed the statement. The `version()` call reports the PostgreSQL underneath, which is the one place where the simulation is visible on purpose.²

oblako engines

For some services, Athena among them, no open-source engine implements the AWS API. Trino can run Athena's queries, but it lacks `StartQueryExecution` and writes its results differently from Athena. For these services oblako provides a small Python server that implements the AWS API and passes the work to a real engine. The Athena engine on port 8009 accepts the athena API, runs each query on Trino and writes the results to S3, where Athena would put them. Trino reads table definitions from oblako's Glue Data Catalog, another Python engine, on port 8486.

²oblako also has a multi-node variant, built on the Citus extension, in which `DISTKEY` shards a table across worker nodes. It is intended for self-hosting; for development, use the single-node engine.

S3 Tables, S3 Vectors, Firehose, EventBridge and the Data APIs for Redshift and RDS work the same way. `oblako up` starts each of these engines as a background process on its port, outside any container:

```
oblako up athena
```

The control plane

Every service also has calls that create and describe resources, such as `create_cluster`, `describe_db_instances` and `create_role`. `oblako` answers them with `moto`, a widely used open-source library that models AWS control planes, served on port 5500. For some services a thin `oblako` proxy sits in front of `moto` and points the resources at real engines; Redshift's is on port 8015:

```
import boto3

redshift = boto3.client("redshift")
redshift.create_cluster(
    ClusterIdentifier="analytics",
    NodeType="ra3.large",
    MasterUsername="oblako",
    MasterUserPassword="Oblako123",
)
resp = redshift.describe_clusters(ClusterIdentifier="analytics")
cluster = resp["Clusters"][0]
print(cluster["ClusterStatus"])
print(cluster["Endpoint"]["Address"], cluster["Endpoint"]["Port"])
```

```
available
localhost 5439
```

With `AWS_PROFILE=oblako`, the `redshift` client calls this proxy on port 8015. `moto` records the cluster, so code that creates, describes or waits on a cluster works. The proxy then reports the endpoint of the engine that serves it, which for a single-node cluster is the Redshift engine of the previous section, on port 5439. A multi-node cluster runs on its own servers (Chapter 13). `oblako` keeps the mocked surface small on purpose and uses `moto` only where there is nothing real to run.

Caddy

Caddy, a web server, sits beside the request path and has two jobs in oblako. Each Application Load Balancer you create is its own Caddy container, which spreads requests across the load balancer's targets. Caddy also gives some services AWS-shaped hostnames; a local MLflow tracking server, for example, gets a URL of the same form as a SageMaker-managed one. Ordinary service traffic goes straight to the ports described above and bypasses Caddy.

1.6 Real and simulated parts

Table 1.2 summarizes, for each data service in this book, which part is real and which part oblako simulates.

Table 1.2: The real and the simulated part of each data service.

Service	Real	Simulated
S3	objects, on disk (S3Proxy)	tags, Inventory and notifications (oblako engine)
S3 Tables	Iceberg tables in a REST catalog	table buckets and namespaces
Glue Data Catalog	table metadata Trino reads	the Glue API (oblako engine)
Athena	queries (Trino)	the Athena API, workgroups
Redshift	SQL (PostgreSQL 16)	clusters and nodes (moto)
RDS	SQL (PostgreSQL), a server per instance, streaming replicas	Aurora clusters and their members (moto)
DynamoDB	tables, items and streams (DynamoDB Local)	vector search and tags (oblako proxy)
Kinesis	streams (kinesalite)	nothing
Firehose	delivery to S3 and Redshift	the Firehose API (oblako engine)

1.7 Where oblako differs from AWS

These are the differences that matter for the data services in this book, and each later chapter repeats the ones that apply to it.

- **Permissions are not enforced.** Any access key works, and bucket policies are stored and returned without being evaluated, so test IAM on AWS.
- **S3 checksums.** S3Proxy does not support the flexible checksums newer SDKs send by default, so oblako sets the SDK to calculate checksums only when an operation requires them.
- **Redshift is a row store.** It runs your SQL correctly without reproducing Redshift's columnar performance, and `version()` reports PostgreSQL. Python user-defined functions still run, though Redshift no longer supports them (Chapter 7).
- **Athena speaks Trino's SQL.** Athena's own engine is based on Trino too, but some functions and DDL differ.
- **Glue has no crawlers.** You define tables yourself, or let a tool such as `awswrangler` or `PyIceberg` define them.
- **RDS runs PostgreSQL 16** whichever engine you request. oblako can run MySQL instead, but only when you start it from its Python API.
- **Vector search is exact.** S3 Vectors and DynamoDB vector search compare the query with every vector, and S3 Vectors keeps its vectors in memory, so they are lost on a restart.
- **Scale is your machine's.** Everything runs on one computer, so oblako shows whether your code is correct but gives no prediction of its performance on AWS.

When a chapter needs something oblako lacks, the book says so and reports the gap to the oblako project instead of working around it in the example.

1.8 Summary

oblako runs real engines (S3Proxy, PostgreSQL, Trino, DynamoDB Local) behind fixed local ports and simulates the AWS topology around them, meaning the endpoints, the control planes and the APIs no engine implements. Your code reaches oblako through the

oblako profile, which gives every service its local endpoint, and is the same on oblako and AWS. oblako differs from AWS in permissions, in scale and in a short list of service-specific details, and each chapter states the details that apply to it.

The next chapter installs oblako, starts it, and makes the first calls against it.

2

Running oblako

Installing oblako, starting its services, and pointing code and the AWS CLI at them.

2.1 What you need

oblako runs on macOS and Linux and needs the following:

- **A container runtime.** Docker is the default. Podman, Colima, a Kubernetes cluster and Apple's container also work; Section 2.7 covers them.
- **Python 3.12** and **uv**, the Python package manager the book uses to install tools and run scripts.
- **Disk space.** The images oblako pulls take a few gigabytes, and the largest of them for the data services is the Redshift engine, at about 1 GB.

The AWS CLI is optional, since everything this chapter does with `boto3` can also be done with `aws`.

2.2 Installing oblako

Install oblako from its source repository. `uv tool install` puts the oblako command on your path, isolated from your projects' dependencies:

```
git clone https://github.com/almostly/oblako.git
cd oblako
uv tool install .
```

Check that the command is installed:

```
oblako --help
```

```
usage: oblako [-h]
              {up,down,status,logs,pull,models,test,
              test-integration,dashboard,notebook,
              redshift-data,trust,bedrock-runtime,rds-data,
              cloudformation,agentcore} ...
```

2.3 Starting the services

`oblako up` starts the services that run in containers:

```
oblako up
```

It pulls missing images, starts the containers and waits until each one answers. The first run takes a few minutes because of the downloads, and later runs take seconds. The containers it starts are the S3 front and S3Proxy, Redshift, RDS, DynamoDB, Step Functions, OpenSearch, Ollama for Bedrock, and moto for the control planes.

A bare `oblako up` leaves out two groups, which you start by name when a chapter needs them. The first is the query stack, made up of Trino (the engine behind Athena) and the Iceberg catalog behind S3 Tables; starting Trino also starts the Glue Data Catalog it reads from:

```
oblako up trino
oblako up iceberg
```

The second group is the Python engines described in Chapter 1, such as Athena, S3 Tables, S3 Vectors, Firehose and the Data APIs. Each runs outside any container as a background process on its port and writes its log under `~/ .oblako/logs`:

```
oblako up athena
oblako up s3tables
```

Each chapter in Part II lists the commands it needs at its start.

The same names work for a single container service, and for down and logs:

```
oblako up redshift      # start just Redshift
oblako logs redshift    # show its log
oblako down redshift    # stop it
```

`oblako status` lists every service and engine with its state. On the machine this book was written on, with the data services running, it printed:

```
bedrock: stopped
opensearch: stopped
redshift: running
rds: stopped
s3proxy: running
dynamodb: running
stepfunctions: stopped
sagemaker: stopped
s3vectors: running
s3tables: running
athena: running
...
glue: running
s3-ext: running
```

`s3proxy` is S3, and `s3-ext` is the add-ons engine behind the S3 front. `moto`, which serves the control planes, is missing from the list but starts with everything else.

2.4 Endpoints

Your code reaches `oblako` through an endpoint per service. The AWS SDKs and the AWS CLI read endpoints, like credentials and the region, from a profile in the shared configuration file, `~/ .aws/config`, and `oblako` writes such a profile for you:

```
oblako configure
```

This adds a profile named `oblako` to `~/.aws/config` and leaves your other profiles as they are. Trimmed to a few services, it reads:

```
[profile oblako]
region = us-east-1
services = oblako
request_checksum_calculation = when_required
response_checksum_validation = when_required
s3 =
    addressing_style = path

[services oblako]
dynamodb =
    endpoint_url = http://localhost:8007
redshift_data =
    endpoint_url = http://localhost:8002
s3 =
    endpoint_url = http://localhost:9000
```

The `services` line names a section that gives each service its endpoint; the full section lists every service `oblako` runs, with the ports from Chapter 1. The checksum settings are needed because `S3Proxy` does not support the checksums newer SDKs add to every upload by default; with them set, the SDK adds a checksum only where an operation requires one. Path-style addressing puts the bucket name in the path of each S3 request instead of in the host name, which `S3Proxy` requires. The command also writes keys for the profile to `~/.aws/credentials`. `oblako` accepts any key, but the SDK signs a request only when it has one, so the keys are generated once for your machine and kept on later runs.

Select the profile in your working shell, and every AWS tool started from that shell then talks to `oblako`:

```
export AWS_PROFILE=oblako
```

Setting `AWS_PROFILE` to one of your AWS profiles instead points the same tools at AWS, as switching a `kubectl` context points `kubectl` at another cluster (Figure 2.1). Chapter 18 builds on this to move the same code to AWS. A few tools outside the AWS SDKs do not read profiles: Spark and PyIceberg ask `boto3` for the endpoint the profile gives (Chapters 4 and 12), and PostgreSQL clients use their own variables (Chapter 8).

This script makes a first call to three services:

```
import boto3

s3 = boto3.client("s3")
s3.create_bucket(Bucket="first-bucket")
s3.put_object(
    Bucket="first-bucket",
    Key="notes/today.txt",
    Body=b"oblako is up",
)
for obj in s3.list_objects_v2(Bucket="first-bucket")["Contents"]:
    print("s3:", obj["Key"], obj["Size"])

dynamodb = boto3.client("dynamodb")
dynamodb.create_table(
    TableName="first-table",
    KeySchema=[{"AttributeName": "id", "KeyType": "HASH"}],
    AttributeDefinitions=[
        {"AttributeName": "id", "AttributeType": "S"}
    ],
    BillingMode="PAY_PER_REQUEST",
)
dynamodb.put_item(
    TableName="first-table",
    Item={"id": {"S": "1"}, "note": {"S": "hello"}},
)
got = dynamodb.get_item(
    TableName="first-table", Key={"id": {"S": "1"}}
)
item = got["Item"]
print("dynamodb:", item["note"]["S"])

identity = boto3.client("sts").get_caller_identity()
print("sts:", identity["Account"], identity["Arn"])
```

```
s3: notes/today.txt 12
dynamodb: hello
sts: 123456789012 arn:aws:sts::123456789012:user/moto
```

moto answers `GetCallerIdentity` with a fixed account number, 123456789012, which also appears in the ARNs that moto and oblako's own engines create. Kinesis and DynamoDB are the exceptions: kinesalite and DynamoDB Local put account 000000000000 in their ARNs.

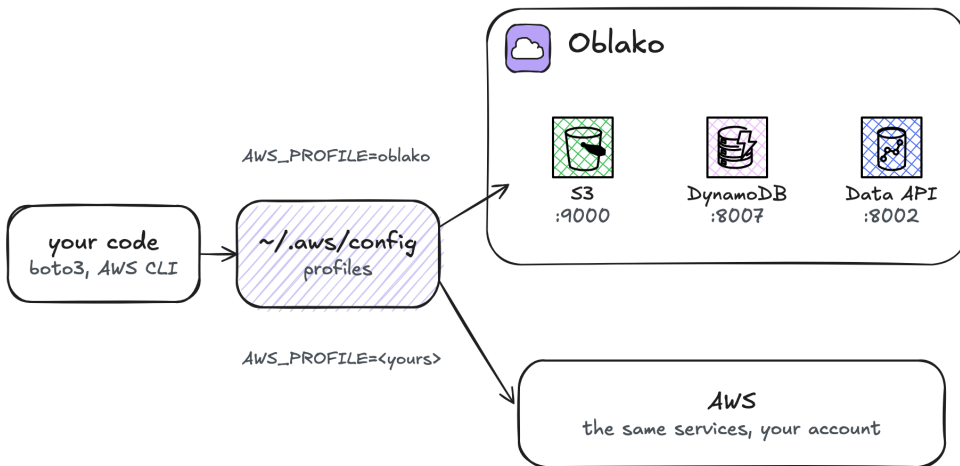


Figure 2.1: One program, two profiles. With `AWS_PROFILE=oblako` the SDK sends each call to oblako's port for that service; with one of your AWS profiles the same code calls AWS.

The AWS CLI reads the same profile, so it sees the same bucket:

```
aws s3 ls s3://first-bucket --recursive
```

```
2026-10-03 17:20:47
```

```
12 notes/today.txt
```

Check which cloud you are talking to

With `AWS_PROFILE=oblako`, every AWS command in that shell goes to oblako, and with another profile, commands go to AWS. Before running anything that creates or deletes resources, run `aws sts get-caller-identity`; the account `123456789012` means you are talking to oblako.

2.5 Connecting to Redshift

Redshift is the one service in this book that your code reaches through its own driver instead of boto3. The engine encrypts connections with a certificate it generates, and

`redshift-connector` checks certificates against a bundle it ships with, so `oblako trust` adds oblako's certificate to that bundle in the Python environment you name:

```
oblako trust --python .venv/bin/python
```

Then connect with full certificate checking, as you would to AWS:

```
import redshift_connector

conn = redshift_connector.connect(
    host="localhost",
    port=5439,
    database="oblako",
    user="oblako",
    password="oblako",
    ssl=True,
    sslmode="verify-ca",
)
```

Reinstalling `redshift-connector` replaces the bundle, so run `oblako trust` again afterwards.

2.6 Data and resets

Each stateful service keeps its data in a named Docker volume: `oblako-s3-data`, `oblako-redshift-data`, `oblako-dynamodb-data` and `oblako-rds-data`. `oblako down` stops the containers and keeps the volumes, so your buckets and tables are there after the next `oblako up`.

To reset all data, stop the services and remove the volumes:

```
oblako down
docker volume rm oblako-s3-data oblako-redshift-data \
    oblako-dynamodb-data oblako-rds-data
```

Some engines keep their data only in memory. S3 Vectors, for example, holds its vectors in memory, so they are lost when it stops, as the chapter on S3 Vectors explains.

2.7 Other container runtimes

oblako starts containers through a pluggable backend, chosen with one variable that you set before `oblako up`. Your code stays the same whichever backend you choose, because each backend makes the services reachable on the same ports on `localhost`; the Kubernetes backend does it by port-forwarding.

Table 2.1: The container runtimes oblako can use.

OBLAKO_CONTAINER_BACKEND	Runtime
<code>docker</code> (default)	Docker Engine
<code>podman</code>	Podman
<code>colima</code>	Colima
<code>kubernetes</code>	any Kubernetes cluster, reached through <code>kubectrl port-forward</code>
<code>apple</code>	Apple's container, on macOS 26 and later

```
export OBLAKO_CONTAINER_BACKEND=podman
oblako up
```

Two limits apply. The S3 front, and with it tagging, Inventory and event notifications, needs a runtime that speaks the Docker API (Docker, Podman or Colima); on Kubernetes or Apple's container, port 9000 is S3Proxy alone. The services that start a container per task, namely Lambda, Glue jobs and SageMaker, need a Docker daemon whichever backend runs the rest.

2.8 The dashboard and notebooks

Two more commands help with local development, though no chapter depends on them. `oblako dashboard` starts a web interface on port 8000, built with Cloudscape, the open-source design system of the AWS console. It shows which services are running, lets you browse buckets and DynamoDB tables, and runs SQL against Redshift. `oblako notebook` starts JupyterLab on port 8888 with the endpoints already set, so `boto3` in a notebook reaches oblako without a profile.

Both are for your own machine. Neither asks for a login, so do not open their ports to other machines or deploy them for other people.

2.9 Summary

Install `oblako` with `uv tool install` from its repository, start the containers with `oblako up` and the Python engines by name, and run `oblako configure` once, so that `AWS_PROFILE=oblako` points the SDKs and the AWS CLI at them. Before connecting to Redshift, run `oblako trust` once per Python environment. Data survives `oblako down`, and removing the volumes resets it.

Part II starts with S3, which the other data services build on.

PART II

The data services



3

S3

Objects, keys, tags, Parquet datasets, bucket policies and event notifications in S3.

3.1 Before you start

Start S3 and source the environment file from Chapter 2. `oblako up s3` starts S3Proxy together with the front that adds tags, policies and notifications:

```
oblako up s3
export AWS_PROFILE=oblako
```

The listings use `boto3`, and the Parquet section adds `pandas` with its S3 support:

```
uv add boto3 pandas pyarrow
```

The examples store sign-ups and page clicks from a small website in a bucket called `events`, one JSON object per event.

3.2 Buckets, objects and keys

S3 has three basic concepts. A **bucket** is a named container that belongs to one account and one region. An **object** is a sequence of bytes plus a little metadata. A **key** is the object's name inside the bucket, any string up to 1,024 bytes long.

Keys are flat: a key such as `raw/2026-09-01/signup-0001.json` is a single string, which S3 stores as written, and its slashes take on meaning only when you list the bucket. A list request can ask for the keys that start with a prefix and group the rest of each key at a delimiter, which is how tools that browse a bucket show it as a file tree. Key design is therefore the main design decision in S3.

S3 writes objects whole, so changing an object, whether by appending to it or by changing a byte in the middle, means replacing it. Since December 2020, S3 has been strongly consistent: once a write returns, every read and every listing sees it.

3.3 What oblako runs

On port 9000, oblako runs a stock nginx in front of S3Proxy, an open-source server that implements the S3 API over a directory of files. nginx passes object reads and writes to S3Proxy and sends the parts of the API that S3Proxy lacks (object tags and S3 Inventory) to a small oblako engine. It also records every completed write so that event notifications can fire. Figure 3.1 shows the arrangement.

The objects live in a Docker volume, `oblako-s3-data`, so they survive oblako down.

3.4 Objects and metadata

Create the bucket and write one event:

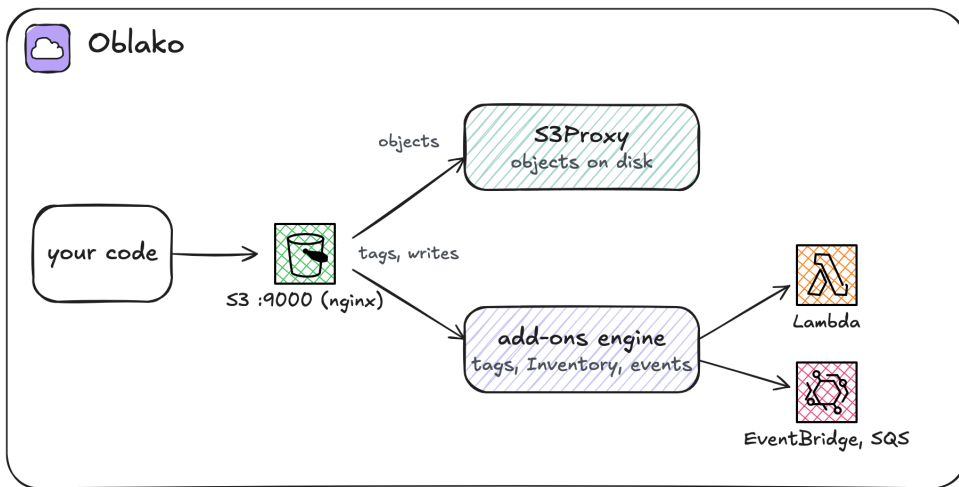


Figure 3.1: S3 on oblako. Objects go to S3Proxy; nginx sends what S3Proxy lacks to oblako's add-ons engine, which also delivers notifications.

```

import json

import boto3

s3 = boto3.client("s3")
s3.create_bucket(Bucket="events")

event = {
    "user": "u-17",
    "action": "signup",
    "at": "2026-09-01T08:15:00Z",
}

s3.put_object(
    Bucket="events",
    Key="raw/2026-09-01/signup-0001.json",
    Body=json.dumps(event).encode(),
    ContentType="application/json",
    Metadata={"source": "web"},
)

key = "raw/2026-09-01/signup-0001.json"
head = s3.head_object(Bucket="events", Key=key)
print(
    head["ContentLength"], head["ContentType"], head["Metadata"]
)

obj = s3.get_object(Bucket="events", Key=key)
print(json.loads(obj["Body"].read()))

```

```
66 application/json {'source': 'web'}
{'user': 'u-17', 'action': 'signup', 'at': '2026-09-01T08:15:00Z'}
```

An object carries two kinds of metadata. **System metadata**, such as the size and Content-Type, is set by S3. **User metadata** is the Metadata dictionary you set, which is stored with the object and returned on every `head_object` and `get_object`. When you need only the metadata, use `head_object`, which returns it without transferring the body.

User metadata is fixed once the object is written, and changing it means copying the object onto itself with new metadata, which rewrites the whole object. Labels that need to change without a rewrite belong in tags, described below.

3.5 Prefixes and listing

Write a few more events, one prefix per day:

```
import boto3

s3 = boto3.client("s3")
for day in ["2026-09-01", "2026-09-02", "2026-09-03"]:
    for n in range(3):
        key = f"raw/{day}/click-{n:04d}.json"
        s3.put_object(Bucket="events", Key=key, Body=b"{}")

resp = s3.list_objects_v2(
    Bucket="events", Prefix="raw/", Delimiter="/"
)
for prefix in resp["CommonPrefixes"]:
    print(prefix["Prefix"])

resp = s3.list_objects_v2(
    Bucket="events", Prefix="raw/2026-09-02/"
)
print([obj["Key"] for obj in resp["Contents"]])
```

```
raw/2026-09-01/
raw/2026-09-02/
raw/2026-09-03/
['raw/2026-09-02/click-0000.json',
 'raw/2026-09-02/click-0001.json',
 'raw/2026-09-02/click-0002.json']
```

The first call asks for the keys under `raw/` and groups everything after the next slash, so it returns three `CommonPrefixes` in place of nine keys; these are what a console shows as folders. The second call lists the keys for one day.

Design keys around how you will read them. If you usually read one day at a time, put the day early in the key, as here, so that the data one read needs sits under one prefix.

Listing more than 1,000 keys

A single `list_objects_v2` call returns at most 1,000 keys, with a continuation token for the rest. A paginator follows the tokens:

```
import boto3

s3 = boto3.client("s3")
for n in range(1200):
    key = f"many/day=2026-09-04/part-{n:05d}.json"
    s3.put_object(Bucket="events", Key=key, Body=b"{}")

paginator = s3.get_paginator("list_objects_v2")
pages = paginator.paginate(Bucket="events", Prefix="many/")
sizes = [len(page["Contents"]) for page in pages]
print(sizes, sum(sizes))
```

```
[1000, 200] 1200
```

Use a paginator in any code that may list a large prefix. Code that reads only the first response works on small prefixes during development and then, in production, silently misses every key after the first 1,000.

3.6 Tags

Tags are up to ten key-value pairs stored on an object separately from its data and its metadata, so you can change them without rewriting the object. That makes them the place for labels that change over an object's life, such as a processing stage, and on AWS, lifecycle rules and access policies can act on them.

```
import boto3

s3 = boto3.client("s3")
key = "raw/2026-09-01/signup-0001.json"

s3.put_object_tagging(
    Bucket="events",
    Key=key,
    Tagging={
        "TagSet": [
            {"Key": "stage", "Value": "raw"},
            {"Key": "pii", "Value": "no"},
        ]
    },
)
print(s3.get_object_tagging(Bucket="events", Key=key)["TagSet"])

# A second put replaces the whole set; it does not merge.
s3.put_object_tagging(
    Bucket="events",
    Key=key,
    Tagging={"TagSet": [{"Key": "stage", "Value": "clean"}]},
)
print(s3.get_object_tagging(Bucket="events", Key=key)["TagSet"])
```

```
[{'Key': 'stage', 'Value': 'raw'}, {'Key': 'pii', 'Value': 'no'}]
[{'Key': 'stage', 'Value': 'clean'}]
```

The second output shows that `put_object_tagging` replaces the whole tag set, so to change one tag you read the set, change it and write it back.

3.7 Datasets in Parquet

A query over JSON objects reads every byte of every file. Data you will query is better written as Parquet, a columnar format that stores each column together and compressed, so that a query reads only the columns it uses. Splitting the data into partitions, one prefix per value of a column you filter on, also lets a query read only the partitions it needs.

pandas writes a partitioned dataset in one call:

```

import boto3
import pandas as pd
from pyarrow import fs

# pandas reads and writes S3 through pyarrow, whose S3 client
# ignores the profile's endpoints, so it gets the one boto3 uses
endpoint = boto3.client("s3").meta.endpoint_url
s3_fs = fs.S3FileSystem(endpoint_override=endpoint)

clicks = pd.DataFrame(
    {
        "day": [
            "2026-09-01",
            "2026-09-01",
            "2026-09-02",
            "2026-09-03",
        ],
        "user": ["u-1", "u-2", "u-1", "u-3"],
        "page": ["/", "/pricing", "/docs", "/"],
    }
)

clicks.to_parquet(
    "events/clicks/",
    partition_cols=["day"],
    filesystem=s3_fs,
)

one_day = pd.read_parquet(
    "events/clicks/",
    filters=[("day", "=", "2026-09-02")],
    filesystem=s3_fs,
)
print(one_day)

```

```

      user  page      day
0  u-1  /docs  2026-09-02

```

pandas writes one prefix per day:

```

clicks/day=2026-09-01/<file>.parquet
clicks/day=2026-09-02/<file>.parquet
clicks/day=2026-09-03/<file>.parquet

```

The `column=value` form is the Hive convention, which Athena, Glue, Spark and Redshift Spectrum read as a partition column. pandas uses it to apply the `filters` argument, listing only the `day=2026-09-02` prefix and opening only the files under it.

Not every tool reads the profile's endpoints

boto3, the AWS CLI, awswrangler and s3fs read the endpoints of the oblako profile, but tools with their own S3 client may not. pandas reads and writes Parquet on S3 through pyarrow's S3 client, which takes the credentials and region from the profile but not its endpoints, and DuckDB has its own client too. The listing therefore gives pyarrow the endpoint that boto3 uses, `boto3.client("s3").meta.endpoint_url`, which on AWS is AWS's own. Without it, pyarrow goes to AWS, which rejects oblako's credentials.

The Glue and Athena chapter gives this kind of dataset a table definition and queries it with SQL.

3.8 Bucket policies

A bucket policy is a JSON document attached to a bucket that allows or denies actions to principals. This one denies everyone permission to delete objects:

```

import json

import boto3

s3 = boto3.client("s3")
policy = {
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DenyDelete",
            "Effect": "Deny",
            "Principal": "*",
            "Action": "s3:DeleteObject",
            "Resource": "arn:aws:s3:::events/*",
        }
    ],
}
s3.put_bucket_policy(Bucket="events", Policy=json.dumps(policy))
resp = s3.get_bucket_policy(Bucket="events")
stored = json.loads(resp["Policy"])
print(stored["Statement"][0]["Sid"])

# On AWS this delete is refused with AccessDenied. oblako stores
# the policy but does not evaluate it, so the delete goes through.
s3.delete_object(
    Bucket="events", Key="raw/2026-09-03/click-0002.json"
)
print("deleted")

```

```

DenyDelete
deleted

```

This is the main limit of S3 on oblako, which stores and returns policies without enforcing them. Code and infrastructure templates that set policies run unchanged, but what a policy allows and denies has to be tested on AWS.

3.9 Event notifications

When an object is created or deleted, S3 can send a message describing the event to an SQS queue, an SNS topic, a Lambda function or EventBridge. A pipeline uses these notifications to process data as it arrives, without polling.

This listing sends an event to an SQS queue whenever an object is created under `incoming/`. moto serves SQS, and the `oblako` profile points the SDK's SQS endpoint at it.

```

import json
import time

import boto3

s3 = boto3.client("s3")
sqs = boto3.client("sqs")

queue_url = sqs.create_queue(QueueName="new-events")["QueueUrl"]
attrs = sqs.get_queue_attributes(
    QueueUrl=queue_url, AttributeNames=["QueueArn"]
)
queue_arn = attrs["Attributes"]["QueueArn"]

prefix_rule = {"Name": "prefix", "Value": "incoming/"}
s3.put_bucket_notification_configuration(
    Bucket="events",
    NotificationConfiguration={
        "QueueConfigurations": [
            {
                "QueueArn": queue_arn,
                "Events": ["s3:ObjectCreated:*"],
                "Filter": {"Key": {"FilterRules": [prefix_rule]}},
            }
        ]
    },
)

s3.put_object(
    Bucket="events", Key="incoming/batch-001.json", Body=b"{"
)

for _ in range(10):
    resp = sqs.receive_message(
        QueueUrl=queue_url, WaitTimeSeconds=1
    )
    messages = resp.get("Messages", [])
    if messages:
        record = json.loads(messages[0]["Body"])[0]
        print(record["eventName"], record["s3"]["object"]["key"])
        break
    time.sleep(1)

```

```
ObjectCreated:Put incoming/batch-001.json
```

The message body is the same JSON document S3 sends on AWS, with a `Records` list, the event name and the bucket and key, so the same consumer code runs on both.

Two details apply on AWS and on oblako alike. Notifications are delivered at least once, so a consumer must handle the same event arriving twice. They usually arrive within seconds of the write rather than at once, which is why the listing polls for a few seconds.

3.10 Where oblako differs from AWS

- **Policies are not enforced**, and neither are IAM permissions on S3 calls.
- **Tags have no effect**. oblako stores and returns them, but it has no lifecycle rules or tag-based access control to act on them.
- **Checksums**. S3Proxy does not support the flexible checksums newer SDKs send by default; the oblako profile sets the SDK to calculate checksums only when an operation requires them.
- **Batch deletes send no events**. `delete_objects` removes the objects, but its keys travel in the request body, which the front does not log, so no deletion notifications fire.
- **The front is Docker-only**. Tags, policies and notifications need the nginx front, which runs on Docker, Podman and Colima; on Kubernetes or Apple's container, those calls return `NotImplemented`.
- **Throughput**. Large downloads through the front run at several hundred megabytes per second, which is slower than S3Proxy alone and well below S3's aggregate throughput. Small requests run at the same speed with or without the front.

3.11 Running it on AWS

To run the listings against S3, unset the oblako variables (or open a new shell) and use your AWS credentials. AWS then requires three changes:

- **Bucket names are global**. All AWS accounts share one bucket namespace, and events is taken, so use a name with something unique in it, such as your account ID.

- **Regions other than us-east-1** need the region named when you create the bucket, through `CreateBucketConfiguration`, as shown after this list.
- **The queue must allow S3 to write to it.** On AWS, `put_bucket_notification_configuration` checks that S3 is allowed to send to the queue and fails if it is not. Before configuring the notification, attach a queue policy that allows the `s3.amazonaws.com` service principal to call `sqs:SendMessage` for this bucket. oblako skips this check, which makes this the one listing that needs an extra step on AWS.

Creating a bucket outside us-east-1:

```
s3.create_bucket(  
    Bucket="events-123456789012",  
    CreateBucketConfiguration={"LocationConstraint": "eu-west-1"},  
)
```

Delete what you created when you finish: the objects, then the bucket, then the queue.

3.12 Summary

S3 stores whole objects under keys in buckets, and listing with a prefix and a delimiter makes the keys look like folders; code that lists a large prefix needs a paginator. Tags can be changed after a write, while user metadata is fixed at write time. Data you query is best written as partitioned Parquet in the `column=value` key layout. On oblako, S3Proxy stores the objects and an nginx front adds tags, policies and notifications; policies are stored but not enforced, so test permissions on AWS.

4

S3 Tables, Iceberg and S3 Vectors

Iceberg tables in S3 Tables, and vector search in S3 Vectors.

4.1 Before you start

Start the Iceberg catalog and the two engines, and select the oblako profile:

```
oblako up iceberg
oblako up s3tables
oblako up s3vectors
export AWS_PROFILE=oblako
```

The table listings use PyIceberg, the Python library for Iceberg:

```
uv add boto3 "pyiceberg[pyarrow,pandas]"
```

The examples keep daily page-view counts for a website, and a few pages of its documentation as vectors.

4.2 Table formats

The partitioned Parquet dataset at the end of Chapter 3 is a set of files, which works while one program writes it and others only read. It leaves three gaps: if two jobs write at once, a reader can see half of one job's files; if you add a column, nothing records which files have it; and once a file is overwritten, the old version is gone.

Apache Iceberg is an open table format that closes these gaps with metadata. An Iceberg table is still Parquet files in object storage, plus a tree of small files that says which data files make up the table at each moment (Figure 4.1):

- a **catalog** entry points at the table's current **metadata file**;
- the metadata file holds the schema, the partitioning and a list of **snapshots**, one per change;
- each snapshot points at **manifests**, which list **data files** with statistics such as each column's minimum and maximum.

A write adds data files, then new manifests and a new metadata file. Finally it swaps the catalog's pointer to that metadata file in one atomic step, so a reader sees the table either as it was before the swap or as it is after it. When two writers commit at the same time, the catalog accepts only a swap made from the current metadata file; the other writer finds the table has moved on and retries its commit on the new state. An append commits on retry unless the retries run out under heavy contention; a change that conflicts with the one just committed, such as a delete of files the other writer rewrote, fails instead of overwriting it. Old snapshots remain until they are expired, so earlier versions of the table stay readable.

Because every engine reads the same metadata, an Iceberg table written from Python can be queried by Athena, Spark or Trino without copying it.

4.3 S3 Tables

S3 Tables is S3's managed Iceberg service. You create a **table bucket**, a bucket that holds only Iceberg tables, and inside it **namespaces**, which group tables the way a database groups schemas. S3 Tables runs the Iceberg catalog for those tables and maintains them in the background: it compacts small files and expires old snapshots.

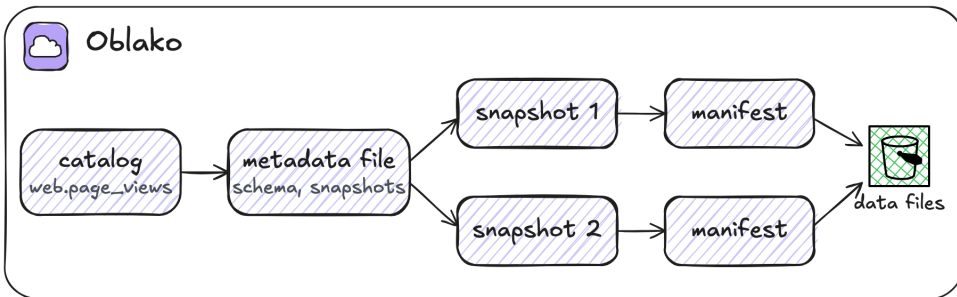


Figure 4.1: An Iceberg table. The catalog points at the current metadata file; each snapshot lists the data files it contains through manifests.

You work with S3 Tables through two interfaces: the `s3tables` API manages buckets, namespaces and tables, and the data goes through the **Iceberg REST endpoint**, which speaks the standard Iceberg catalog protocol that any Iceberg client supports.

What oblako runs

oblako runs one Iceberg REST catalog, `tabulario/iceberg-rest`, which keeps its files in oblako's S3. In front of it, the S3 Tables engine on port 8013 serves the `s3tables` API and, under `/iceberg`, the same Iceberg endpoint AWS serves, addressed as on AWS by the table bucket's ARN. Each table bucket becomes the first level of a namespace in the shared catalog, so the tables created here are also visible to Trino, and through Trino to Athena in the next chapter.

4.4 Creating a table bucket

The `s3tables` API creates the bucket and a namespace in it:

```
import boto3

s3tables = boto3.client("s3tables")
bucket_arn = s3tables.create_table_bucket(name="analytics")["arn"]
s3tables.create_namespace(
    tableBucketARN=bucket_arn, namespace=["web"]
)

print(bucket_arn)
namespaces = s3tables.list_namespaces(tableBucketARN=bucket_arn)
print(namespaces["namespaces"][0]["namespace"])
```

```
arn:aws:s3tables:us-east-1:123456789012:bucket/analytics
['web']
```

The rest of the chapter identifies the bucket by this ARN, whose account number is the simulated account from Chapter 2.

4.5 Connecting PyIceberg

PyIceberg connects to S3 Tables as a REST catalog, with the table bucket's ARN as the warehouse and requests signed with your AWS credentials. The connection lives in a small module, `catalog.py`, which the later listings share:

```

import boto3
from pyiceberg.catalog import load_catalog

def open_catalog():
    s3tables = boto3.client("s3tables")
    buckets = s3tables.list_table_buckets(prefix="analytics")
    return load_catalog(
        "s3tables",
        **{
            "type": "rest",
            # the profile's S3 Tables endpoint, oblako or AWS
            "uri": s3tables.meta.endpoint_url + "/iceberg",
            "warehouse": buckets["tableBuckets"][0]["arn"],
            "rest.sigv4-enabled": "true",
            "rest.signing-name": "s3tables",
            "rest.signing-region": s3tables.meta.region_name,
        },
    )

```

PyIceberg does not read the endpoints in an AWS profile, so `catalog.py` takes the catalog's URI from the boto3 client: the S3 Tables endpoint the selected profile gives, followed by `/iceberg`. Under the `oblako` profile that is `oblako's` catalog, `http://localhost:8013/iceberg`; under an AWS profile it is the S3 Tables endpoint for your region, and `catalog.py` does not change.

4.6 Creating and appending to a table

PyIceberg creates a table from an Arrow schema and appends Arrow data to it:

```

import pyarrow as pa
from catalog import open_catalog

catalog = open_catalog()
views = pa.table(
    {
        "day": ["2026-09-01", "2026-09-01", "2026-09-02"],
        "page": ["/", "/pricing", "/docs"],
        "views": [120, 31, 45],
    }
)
schema = views.schema
table = catalog.create_table("web.page_views", schema=schema)
table.append(views)

print(catalog.list_tables("web"))
print(table.scan(row_filter="views > 40").to_pandas())

```

```

[('web', 'page_views')]
   day  page  views
0  2026-09-01    /   120
1  2026-09-02 /docs    45

```

append wrote a Parquet file, a manifest, a snapshot and a new metadata file, then committed them in one step. scan with a `row_filter` uses the column statistics in the manifests to skip files that cannot match before it reads any data.

4.7 Snapshots and time travel

Every commit creates a snapshot, and earlier snapshots remain readable:

```

import pyarrow as pa
from catalog import open_catalog

table = open_catalog().load_table("web.page_views")
first = table.current_snapshot().snapshot_id

more = pa.table(
    {"day": ["2026-09-03"], "page": ["/"], "views": [98]}
)
table.append(more)

for entry in table.history():
    print(entry.snapshot_id, entry.timestamp_ms)

print("now:", table.scan().to_arrow().num_rows, "rows")
then = table.scan(snapshot_id=first).to_arrow()
print("then:", then.num_rows, "rows")

```

```

7052313727309517191 1790942355646
953361004551372475 1790942356454
now: 4 rows
then: 3 rows

```

The table has two snapshots, and a scan pinned to the first one returns the table as it was before the second append. You can use this to reproduce an earlier report or to check what a bad write changed, without keeping copies.

4.8 Changing the schema

Iceberg tracks each column by an ID that is independent of its name and position, so adding, renaming or dropping a column changes only the metadata and leaves the data files as they are:

```

import pyarrow as pa
from catalog import open_catalog
from pyiceberg.types import StringType

table = open_catalog().load_table("web.page_views")
with table.update_schema() as update:
    update.add_column("country", StringType())

new = pa.table(
    {
        "day": ["2026-09-04"],
        "page": ["/docs"],
        "views": [52],
        "country": ["NL"],
    }
)
table.append(new)
rows = table.scan().to_pandas().sort_values("day")
print(rows.tail(3))

```

	day	page	views	country
4	2026-09-02	/docs	45	None
1	2026-09-03	/	98	None
0	2026-09-04	/docs	52	NL

Rows written before the change read the new column as empty; rows written after carry it.

4.9 S3 Vectors

A vector here is a list of numbers that stands for a piece of content, such as a page of text, so that similar content has nearby vectors. An embedding model produces it, such as one of those Bedrock offers. A search by meaning finds the vectors nearest to the vector of a question.

S3 Vectors stores vectors in S3 and searches them. A **vector bucket** holds **indexes**. Each index fixes the number of dimensions and the distance measure, and holds vectors that each carry a key and optional metadata you can filter on.

The listing uses three-dimensional vectors written by hand, which stand in for the embeddings of three documentation pages:

```

import boto3

s3vectors = boto3.client("s3vectors")
s3vectors.create_vector_bucket(vectorBucketName="docs-vectors")
s3vectors.create_index(
    vectorBucketName="docs-vectors",
    indexName="pages",
    dataType="float32",
    dimension=3,
    distanceMetric="cosine",
)

def page(key, vector, section):
    return {
        "key": key,
        "data": {"float32": vector},
        "metadata": {"section": section},
    }

s3vectors.put_vectors(
    vectorBucketName="docs-vectors",
    indexName="pages",
    vectors=[
        page("pricing", [0.9, 0.1, 0.0], "sales"),
        page("quickstart", [0.1, 0.9, 0.2], "docs"),
        page("api", [0.0, 0.8, 0.6], "docs"),
    ],
)

resp = s3vectors.query_vectors(
    vectorBucketName="docs-vectors",
    indexName="pages",
    queryVector={"float32": [0.1, 1.0, 0.3]},
    topK=2,
    filter={"section": "docs"},
    returnDistance=True,
    returnMetadata=True,
)

for hit in resp["vectors"]:
    print(hit["key"], round(hit["distance"], 4), hit["metadata"])

```

```
quickstart 0.0027 {'section': 'docs'}
api 0.0656 {'section': 'docs'}
```

The query asks for the two nearest vectors in the docs section. Cosine distance is one minus the cosine of the angle between two vectors, so 0 means the two vectors point the same way, and quickstart, at 0.0027, is almost parallel to the query. pricing is further away, and the metadata filter excludes it as well.

4.10 Where oblako differs from AWS

- **One catalog.** oblako keeps every table bucket in one Iceberg catalog, as the first level of a namespace, where AWS makes each table bucket its own catalog. Your code behaves the same, and Trino and Glue see oblako's table buckets as namespaces.
- **No maintenance.** S3 Tables compacts files and expires snapshots on its own, and oblako does neither, so snapshots and small files accumulate for as long as the table exists.
- **S3 Vectors searches by brute force** and keeps vectors in memory, so its results are exact (AWS does not promise exact results on large indexes) and the vectors are lost when the engine stops.
- **Permissions are not enforced**, for tables or vectors.

4.11 Running it on AWS

The listings run on AWS without code changes once you select an AWS profile; catalog.py then connects PyIceberg to the S3 Tables endpoint for your region. Your role needs the s3tables permissions for the bucket, namespace and table operations, which PyIceberg calls through the endpoint. Check that S3 Tables and S3 Vectors are available in your region. When you finish, delete the table, the namespace and the table bucket, and the vector index and vector bucket.

4.12 Summary

Iceberg adds atomic commits, snapshots and schema changes to files in object storage, through metadata that lists which files make up each version of a table. S3 Tables is a managed Iceberg service in which the `s3tables` API manages buckets and namespaces and any Iceberg client reads and writes through the REST endpoint, addressed by the table bucket's ARN. S3 Vectors stores vectors with metadata and finds the ones nearest to a query. On oblako, the same code runs against one shared Iceberg catalog and a brute-force vector engine.

5

The Glue Data Catalog and Athena

Describing data in S3 as tables with the Glue Data Catalog, and querying it with Athena.

5.1 Before you start

Athena on oblako needs four services: S3 for the data, the Iceberg catalog, Trino (which also starts the Glue Data Catalog), and the Athena engine itself:

```
oblako up s3
oblako up iceberg
oblako up trino
oblako up athena
export AWS_PROFILE=oblako
```

```
uv add boto3 pandas pyarrow awswrangler
```

The examples analyze page clicks: which pages were opened, by whom, and how long they took to load.

5.2 The Data Catalog

A table in the Glue Data Catalog is a description of data that lives elsewhere, in S3. It records a database and table name, the columns and their types, the file format, the S3 location of the files, and the columns the files are partitioned by. Each partition is a further entry that maps a value, such as `day=2026-09-02`, to a prefix.

Any engine that reads the catalog (Athena, Spark, Redshift Spectrum, Trino) sees the same tables over the same files. The files belong to none of them: a change to a table's description is visible to every engine, and deleting a table leaves its files in place.

What oblako runs

oblako's Glue Data Catalog is a Python engine on port 8486. It keeps Parquet, CSV and JSON tables, with their partitions, in a small database of its own, and Iceberg tables in the shared Iceberg catalog from Chapter 4. Athena's engine on port 8009 accepts the athena API and runs each query on Trino, whose `awsdatacatalog` catalog reads table definitions from the Glue engine. Figure 5.1 shows the path of a query.

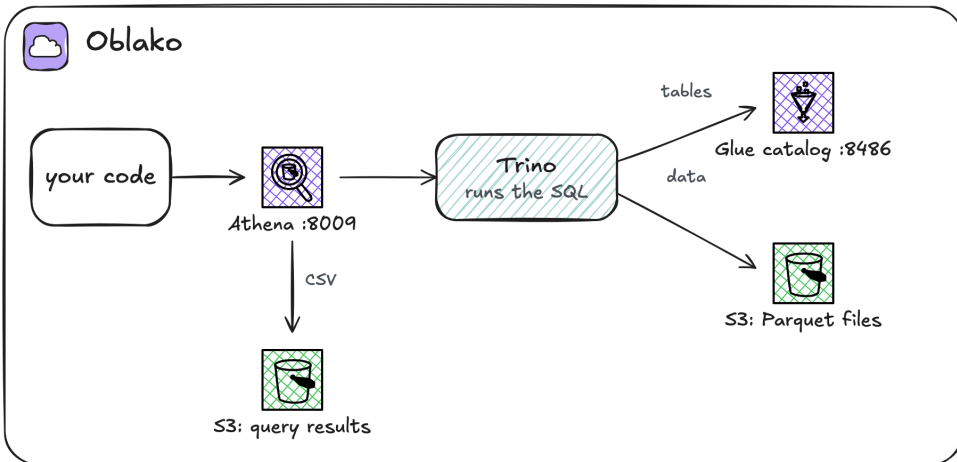


Figure 5.1: An Athena query on oblako. Trino reads table definitions from the Glue engine and data from S3; the Athena engine writes the results to S3, as Athena does.

5.3 Sample data

Write a small partitioned Parquet dataset, as in Chapter 3:

```
import boto3
import pandas as pd
from pyarrow import fs

s3 = boto3.client("s3")
s3.create_bucket(Bucket="lake-data")
# pandas reads and writes S3 through pyarrow, whose S3 client
# ignores the profile's endpoints, so it gets the one boto3 uses
s3_fs = fs.S3FileSystem(endpoint_override=s3.meta.endpoint_url)
clicks = pd.DataFrame(
    {
        "day": ["2026-09-01"] * 3
        + ["2026-09-02"] * 2
        + ["2026-09-03"] * 3,
        "user": [
            "u-1",
            "u-2",
            "u-3",
            "u-1",
            "u-4",
            "u-2",
            "u-3",
            "u-5",
        ],
        "page": [
            "/",
            "/pricing",
            "/",
            "/docs",
            "/",
            "/docs",
            "/",
            "/pricing",
        ],
        "ms": [820, 1430, 610, 2210, 450, 1980, 700, 1250],
    }
)
clicks.to_parquet(
    "lake-data/clicks/",
    partition_cols=["day"],
    filesystem=s3_fs,
```

```
)
print(len(clicks), "rows")
```

```
8 rows
```

5.4 Defining a table

A table definition names the columns, the format and the location. For Parquet, the format is three Hive class names, which every engine recognizes. The partition column, day, is listed separately from the other columns because its values come from the prefixes rather than from the files:

```
import boto3

glue = boto3.client("glue")
glue.create_database(DatabaseInput={"Name": "web"})

hive = "org.apache.hadoop.hive ql.io.parquet"
parquet = {
    "InputFormat": f"{hive}.MapredParquetInputFormat",
    "OutputFormat": f"{hive}.MapredParquetOutputFormat",
    "SerdeInfo": {
        "SerializationLibrary": f"{hive}.serde.ParquetHiveSerDe"
    },
}

glue.create_table(
    DatabaseName="web",
    TableInput={
        "Name": "clicks",
        "TableType": "EXTERNAL_TABLE",
        "PartitionKeys": [{"Name": "day", "Type": "string"}],
        "StorageDescriptor": {
            "Columns": [
                {"Name": "user", "Type": "string"},
                {"Name": "page", "Type": "string"},
                {"Name": "ms", "Type": "bigint"},
            ],
            "Location": "s3://lake-data/clicks/",
            **parquet,
        },
    },
)
```

```
    },
)
```

The table now exists, but a query finds rows only in registered partitions, so register one partition per prefix:

```
days = ["2026-09-01", "2026-09-02", "2026-09-03"]
glue.batch_create_partition(
    DatabaseName="web",
    TableName="clicks",
    PartitionInputList=[
        {
            "Values": [day],
            "StorageDescriptor": {
                "Location": f"s3://lake-data/clicks/day={day}/",
                **parquet,
            },
        },
        for day in days
    ],
)
parts = glue.get_partitions(
    DatabaseName="web", TableName="clicks"
)
print(sorted(p["Values"][0] for p in parts["Partitions"]))
```

```
['2026-09-01', '2026-09-02', '2026-09-03']
```

On AWS, a Glue crawler can discover tables and partitions by scanning S3, and Athena can add partitions with `MSCK REPAIR TABLE`. oblako has neither, so on oblako you define tables explicitly, as here or with a library.

5.5 Querying with Athena

Athena's API is asynchronous: you start a query, wait for it to finish, then fetch the result. The helper in `athena_query.py` does all three steps:

```

import time

import boto3

athena = boto3.client("athena")
DONE = ("SUCCEEDED", "FAILED", "CANCELLED")

def run_query(sql, database="web"):
    query_id = athena.start_query_execution(
        QueryString=sql,
        QueryExecutionContext={"Database": database},
        WorkGroup="primary",
    )["QueryExecutionId"]
    while True:
        execution = athena.get_query_execution(
            QueryExecutionId=query_id
        )
        status = execution["QueryExecution"]["Status"]
        if status["State"] in DONE:
            break
        time.sleep(0.5)
    if status["State"] != "SUCCEEDED":
        raise RuntimeError(status.get("StateChangeReason"))
    results = athena.get_query_results(QueryExecutionId=query_id)
    rows = results["ResultSet"]["Rows"]
    return [
        [col.get("VarCharValue") for col in row["Data"]]
        for row in rows
    ]

```

With the helper, a query is one call:

```

from athena_query import run_query

for row in run_query(
    "SELECT page, count(*) AS clicks, avg(ms) AS avg_ms"
    " FROM clicks WHERE day >= '2026-09-02'"
    " GROUP BY page ORDER BY clicks DESC, page"
):
    print(row)

```

```
['page', 'clicks', 'avg_ms']
['/', '2', '575.0']
['/docs', '2', '2095.0']
['/pricing', '1', '1250.0']
```

The first row is the header, and every value comes back as a string, as Athena returns them. The `WHERE day >= '2026-09-02'` filter is on the partition column, so the engine reads only two of the three prefixes, which on AWS lowers the cost because Athena charges by the bytes it scans.

Query results

Athena writes every result to S3 as a CSV file, in the location the workgroup names. A **workgroup** groups queries and holds their settings, and every account has one called `primary`:

```
import boto3

athena = boto3.client("athena")
group = athena.get_work_group(WorkGroup="primary")["WorkGroup"]
results = group["Configuration"]["ResultConfiguration"]
print(results["OutputLocation"])
```

```
s3://oblako-athena-results/
```

On `oblako`, `primary` comes configured with that bucket, while on AWS you set the output location on the workgroup yourself, once.

5.6 awswrangler

`awswrangler`, the AWS SDK for pandas, wraps the steps above: `wr.s3.to_parquet` writes a partitioned dataset, creates the table and registers its partitions, and `wr.athena.read_sql_query` runs a query and returns a `DataFrame`:

```

import awswrangler as wr
import pandas as pd

signups = pd.DataFrame(
    {
        "day": ["2026-09-01", "2026-09-01", "2026-09-02"],
        "user": ["u-1", "u-2", "u-4"],
        "plan": ["free", "team", "free"],
    }
)

wr.s3.to_parquet(
    signups,
    path="s3://lake-data/signups/",
    dataset=True,
    database="web",
    table="signups",
    partition_cols=["day"],
)

df = wr.athena.read_sql_query(
    "SELECT plan, count(*) AS n FROM signups"
    " GROUP BY plan ORDER BY plan",
    database="web",
)
print(df)

```

```

      plan  n
0  free    2
1  team    1

```

By default `to_parquet` with `dataset=True` appends, so running the listing twice doubles the rows; pass `mode="overwrite"` to replace them. `read_sql_query` wraps the query in a `CREATE TABLE AS SELECT`, which writes Parquet, and then reads those files, which for large results is much faster than fetching rows through the API.

5.7 CREATE TABLE AS SELECT

`CREATE TABLE AS SELECT`, or CTAS, runs a query and saves its result as a new table, in the format and location you choose. Use it to build a smaller, faster table from a larger one:

```

from athena_query import run_query

run_query(
    "CREATE TABLE daily_pages"
    " WITH (format = 'PARQUET',"
    " external_location = 's3://lake-data/daily_pages/')"
    " AS SELECT day, page, count(*) AS clicks"
    " FROM clicks GROUP BY day, page"
)
rows = run_query("SELECT * FROM daily_pages ORDER BY day, page")
for row in rows:
    print(row)

```

```

['day', 'page', 'clicks']
['2026-09-01', '/', '2']
['2026-09-01', '/pricing', '1']
['2026-09-02', '/', '1']
['2026-09-02', '/docs', '1']
['2026-09-03', '/', '1']
['2026-09-03', '/docs', '1']
['2026-09-03', '/pricing', '1']

```

The new table is in the Glue Data Catalog and its Parquet files are under `external_location`, so any engine can read it.

5.8 Querying S3 Tables

Athena reaches S3 Tables through a catalog per table bucket, named `s3tablescatalog/<bucket>`, in which each namespace is a database. Chapter 4's table is `page_views` in namespace `web` of bucket `analytics`:

```

from athena_query import run_query

sql = (
    "SELECT day, sum(views) AS views"
    " FROM "s3tablescatalog/analytics"."web"."page_views"
    " GROUP BY day ORDER BY day"
)
for row in run_query(sql):
    print(row)

```

```
[ 'day', 'views' ]
[ '2026-09-01', '151' ]
[ '2026-09-02', '45' ]
[ '2026-09-03', '98' ]
[ '2026-09-04', '52' ]
```

These are the rows PyIceberg wrote in Chapter 4, which Athena reads from the same table without a copy. You can also leave the catalog out of the SQL and pass it in the query context, as `QueryExecutionContext={"Catalog": "s3tablescatalog/analytics", "Database": "web"}`.

5.9 Where oblako differs from AWS

- **Trino's SQL.** Athena's engine is based on Trino, and oblako runs Trino itself, so most queries behave the same. Athena's Hive DDL (`CREATE EXTERNAL TABLE ... ROW FORMAT, MSCK REPAIR TABLE`) and `UNLOAD` are not translated, so define tables through Glue or a library.
- **No crawlers,** connections, Lake Formation permissions or table versions in the Glue Data Catalog.
- **No scan statistics.** Query results do not report bytes scanned, so you cannot measure locally what a query would cost.
- **Scale.** Trino runs as a single node on your machine.

5.10 Running it on AWS

Unset the oblako variables, give the primary workgroup an output location in a bucket you own (or create a workgroup of your own), and use a globally unique name instead of `lake-data`. Your role needs the Glue permissions for the database and tables, the S3 permissions for the data and the results bucket, and the Athena permissions to run queries. Querying S3 Tables also needs the table buckets integrated with AWS analytics services, an integration the S3 console offers when you create the first table bucket.

Delete what you created when you finish: the tables and the database in Glue, which leaves the files, and then the files.

5.11 Summary

A Glue table is metadata (columns, format, location and partitions) over files in S3, and Athena runs SQL over those tables and writes the results to the workgroup's S3 location. Filtering on partition columns reduces the bytes a query scans, and so its cost. `awsdatacatalog` creates tables and runs queries from pandas in one call each, and CTAS saves a query's result as a new table. On `oblako`, the Glue engine holds the catalog, Trino runs the queries, and S3 Tables are reachable as `s3tables.catalog/<bucket>`.

6

Glue jobs

A PySpark job on Glue, from script to Parquet output.

6.1 Before you start

Glue jobs need S3, the Glue engine, and Docker, because each run starts a container:

```
oblako up s3
oblako up glue
export AWS_PROFILE=oblako
```

```
uv add boto3 pandas pyarrow
```

The first run pulls `amazon/aws-glue-libraries:5`, the image AWS publishes for developing Glue 5 jobs locally. It is about 5 GB, so the first run takes a while; later runs start in seconds.

6.2 Glue jobs in brief

Besides the Data Catalog, Glue runs serverless Spark. A **job** is a definition made up of a Python script in S3, the arguments to pass it, the Glue version and the number of **workers**

to run it on. A **job run** is one execution of that definition. You pay for worker time while a run is in progress and nothing between runs.

The script is ordinary PySpark plus Glue's own library, `awsGlue`, whose main type is the **DynamicFrame**, Glue's version of a Spark DataFrame. Where a DataFrame needs one schema up front, a DynamicFrame works out each record's schema as it reads. A field that is a number in some files and a string in others becomes a *choice* between the two types, which you resolve explicitly; a DataFrame would fail partway through the run on the same data. This makes DynamicFrames suited to messy raw data, and `toDF()` and `DynamicFrame.fromDF()` convert between the two at any point.

What oblako runs

oblako's Glue engine, on port 8486, serves Glue's job API next to the Data Catalog. When you start a run, the engine fetches the script from S3 and runs it with `spark-submit` in a fresh container of AWS's Glue 5 image, which has the same Spark and `awsGlue` versions as Glue 5.0. Inside the container, `s3://` paths and the AWS SDKs point at oblako, so the script needs no settings of its own. The run's output goes to CloudWatch Logs, served by moto, in the log groups Glue uses. Figure 6.1 shows the arrangement.

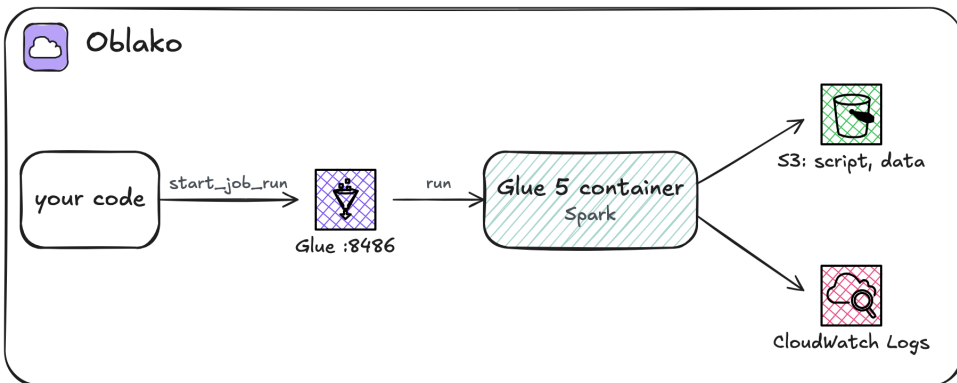


Figure 6.1: A Glue job run on oblako. The engine runs the script from S3 in a Glue 5 container; the container reads and writes oblako's S3 and logs to CloudWatch Logs.

6.3 Raw data

The job cleans page-click events that arrive as JSON lines, one event per line:

```
import json

import boto3

s3 = boto3.client("s3")
s3.create_bucket(Bucket="etl-demo")

clicks = [
    ("2026-09-01", "u-1", "/", 820),
    ("2026-09-01", "u-2", "/pricing", 1430),
    ("2026-09-01", "u-3", "/", 2650),
    ("2026-09-02", "u-1", "/docs", 2210),
    ("2026-09-02", "u-4", "/", 450),
    ("2026-09-03", "u-2", "/docs", 1980),
]
lines = [
    json.dumps({"day": d, "user": u, "page": p, "ms": ms})
    for d, u, p, ms in clicks
]
s3.put_object(
    Bucket="etl-demo",
    Key="raw/clicks/batch-001.json",
    Body="\n".join(lines).encode(),
)
print(len(lines), "events")
```

6 events

6.4 The job script

The complete job is saved as `clean_clicks.py`. It reads the raw events, gives each field its type and a clearer name, keeps only page loads under two seconds, and writes the result as Parquet partitioned by day:

```

import sys

from awsglue.context import GlueContext
from awsglue.job import Job
from awsglue.transforms import ApplyMapping, Filter
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext

args = getResolvedOptions(
    sys.argv, ["JOB_NAME", "source", "target"]
)
glue = GlueContext(SparkContext())
job = Job(glue)
job.init(args["JOB_NAME"], args)

raw = glue.create_dynamic_frame.from_options(
    "s3", {"paths": [args["source"]]}, format="json"
)
typed = ApplyMapping.apply(
    frame=raw,
    mappings=[
        ("day", "string", "day", "string"),
        ("user", "string", "user", "string"),
        ("page", "string", "page", "string"),
        ("ms", "long", "load_ms", "long"),
    ],
)
fast = Filter.apply(
    frame=typed, f=lambda row: row["load_ms"] < 2000
)
glue.write_dynamic_frame.from_options(
    fast,
    "s3",
    {"path": args["target"], "partitionKeys": ["day"]},
    format="parquet",
)
print("rows in:", raw.count(), "rows out:", fast.count())
job.commit()

```

The script does the following.

- `getResolvedOptions` reads the job's arguments. Glue passes every run `--JOB_NAME` and the arguments you define, here `--source` and `--target`, so the same script runs against any paths.

- `GlueContext` wraps Spark with Glue's readers and writers, and `Job` marks the start and end of the run.
- `from_options` reads every file under the source prefix into a `DynamicFrame`. `from_catalog(database=..., table_name=...)` reads a table defined in the Data Catalog instead, as in Chapter 5.
- `ApplyMapping` selects, renames and types fields in one step: `ms`, a long, becomes `load_ms`.
- `Filter` keeps the rows for which the function returns true.
- `write_dynamic_frame` writes Parquet, one prefix per value of `day`, the layout Athena reads as partitions.

The script leaves endpoints and credentials to the environment it runs in, so the same file runs on Glue.

6.5 Creating the job

Upload the script and define the job around it:

```

import boto3

s3 = boto3.client("s3")
glue = boto3.client("glue")

s3.upload_file(
    "clean_clicks.py", "etl-demo", "scripts/clean_clicks.py"
)
glue.create_job(
    Name="clean-clicks",
    Role="arn:aws:iam::123456789012:role/glue-etl",
    Command={
        "Name": "glueetl",
        "ScriptLocation": "s3://etl-demo/scripts/clean_clicks.py",
        "PythonVersion": "3",
    },
    DefaultArguments={
        "--source": "s3://etl-demo/raw/clicks/",
        "--target": "s3://etl-demo/clean/clicks/",
    },
    GlueVersion="5.0",
    WorkerType="G.1X",
    NumberOfWorkers=2,
)
print(glue.list_jobs()["JobNames"])

```

```
['clean-clicks']
```

glueetl is the Spark job type, and DefaultArguments are passed to every run unless the run overrides them. On AWS the job runs as the given role, which must allow it to read and write its S3 paths and write its logs; oblako records the role and runs the job without checking its permissions.

6.6 Running the job

A run is asynchronous, like an Athena query, so the listing starts a run and polls its state until it finishes:

```

import time

import boto3

glue = boto3.client("glue")
DONE = ("SUCCEEDED", "FAILED", "TIMEOUT", "STOPPED", "ERROR")

run_id = glue.start_job_run(JobName="clean-clicks")["JobRunId"]
while True:
    run = glue.get_job_run(JobName="clean-clicks", RunId=run_id)
    state = run["JobRun"]["JobRunState"]
    if state in DONE:
        break
    time.sleep(10)
print(state)
if state != "SUCCEEDED":
    print(run["JobRun"]["ErrorMessage"])

logs = boto3.client("logs")
events = logs.get_log_events(
    logGroupName="/aws-glue/jobs/output", logStreamName=run_id
)["events"]
for event in events:
    print(event["message"])

```

```

SUCCEEDED
rows in: 6 rows out: 4

```

What the script prints goes to the CloudWatch Logs group `/aws-glue/jobs/output`, in a stream named after the run; Spark's own logging goes to `/aws-glue/jobs/error`. Glue on AWS uses the same log groups, so the same code finds a run's logs there. When a run fails, `ErrorMessage` holds the error, for example `GlueArgumentError`: the following arguments are required: `--target` when an argument is missing.

6.7 Output

The output is a partitioned Parquet dataset, which pandas reads directly:

```
import boto3
import pandas as pd
from pyarrow import fs

# pandas reads and writes S3 through pyarrow, whose S3 client
# ignores the profile's endpoints, so it gets the one boto3 uses
endpoint = boto3.client("s3").meta.endpoint_url
s3_fs = fs.S3FileSystem(endpoint_override=endpoint)

clean = pd.read_parquet(
    "etl-demo/clean/clicks/", filesystem=s3_fs
)
print(clean.sort_values(["day", "user"]).to_string(index=False))
```

```
load_ms user      page      day
   820  u-1        / 2026-09-01
  1430  u-2 /pricing 2026-09-01
   450  u-4        / 2026-09-02
  1980  u-2    /docs 2026-09-03
```

Two events took two seconds or more and were filtered out. To query this dataset with Athena, define a table over it as in Chapter 5.

6.8 Where oblako differs from AWS

- **One container per run.** oblako records `WorkerType` and `NumberOfWorkers` but runs every job as one container running Spark locally. Results match Glue's, while speed and memory depend on your machine.
- **Job bookmarks are not modeled.** On Glue, a bookmark lets a job process only the files that arrived since its last run; on oblako every run reads all of its input.
- **Unsupported features.** Triggers, crawlers, streaming jobs and interactive sessions are unavailable, so you start runs yourself with `StartJobRun`.
- **Directory markers.** The open-source S3 client that Spark uses locally writes empty objects such as `clean/clicks/day=2026-09-01/` next to the data, whereas Glue on AWS uses its own S3 client. Readers skip these objects, though a listing shows them.
- **The role is not checked,** so a job that runs locally can still fail on AWS for lack of a permission.

6.9 Running it on AWS

Unset the `oblako` variables and use a bucket name of your own. Create an IAM role for the job that Glue can assume, with permission to read the script and the raw data, write the output prefix, and write to CloudWatch Logs. Use its ARN as `Role`, and the listings run as they are. On AWS a run of this size usually takes a minute or two, most of it starting workers, and is billed per worker-second with a one-minute minimum. Delete the job and the bucket when you finish.

6.10 Summary

A Glue job is a PySpark script in S3 with arguments and a worker count, and a job run executes it. `DynamicFrames` work out the schema of messy data as they read it, and `ApplyMapping`, `Filter` and `write_dynamic_frame` cover most cleaning jobs. You create and run jobs with `boto3`, and read their output from CloudWatch Logs. On `oblako` each run is AWS's Glue 5 image in a local container, with S3, the Data Catalog and logs pointing at `oblako`, so the same script runs on AWS unchanged.

7

Redshift

Tables, loading, SQL, JSON and unloading in Redshift, through a driver and the Data API.

7.1 Before you start

Start Redshift, its Data API and S3, and trust the engine's certificate in your project's Python environment, as in Chapter 2:

```
oblako up redshift
oblako up redshift-data
oblako up s3
oblako trust --python .venv/bin/python
export AWS_PROFILE=oblako
```

```
uv add redshift-connector boto3 pandas pyarrow
```

The examples analyze page views: who opened which page, on which day, and how long it took to load.

7.2 Redshift in brief

Redshift stores each column separately and compressed, so a query that reads three columns of a wide table reads only those three. It splits every table across the **slices** of its compute nodes and runs each query on all slices in parallel, with a leader node planning the work and combining the results. You run it as a provisioned cluster of a chosen size, or as Redshift Serverless, which sizes itself and bills per use.

Two choices in a table's DDL decide how well that parallelism works. The **distribution key** (`DISTKEY`) decides which slice each row lives on: rows with the same key share a slice, so joins and aggregations on that key need no data moved between nodes. The **sort key** (`SORTKEY`) decides the order rows are stored in, so a filter on it can skip whole blocks. Data is loaded and unloaded in bulk through S3, with `COPY` and `UNLOAD`.

What oblako runs

oblako's Redshift is PostgreSQL 16 with Redshift's surface added: the startup handshake Redshift's driver expects, Redshift's system views and functions, and the `SUPER` type. In front of it, on Redshift's port, 5439, a wire proxy terminates TLS, rewrites Redshift-only SQL that PostgreSQL would reject, and carries out `COPY` and `UNLOAD` against oblako's S3. The Data API is a separate engine on port 8002 that runs its statements through the same proxy, and moto holds the cluster's metadata. Figure 7.1 shows the pieces.

Redshift SQL therefore runs and returns Redshift's results, while the data itself sits in PostgreSQL's row-based storage, in a single engine on one node.

7.3 Connecting

Keep the connection in a small module, `warehouse.py`, that the listings share:

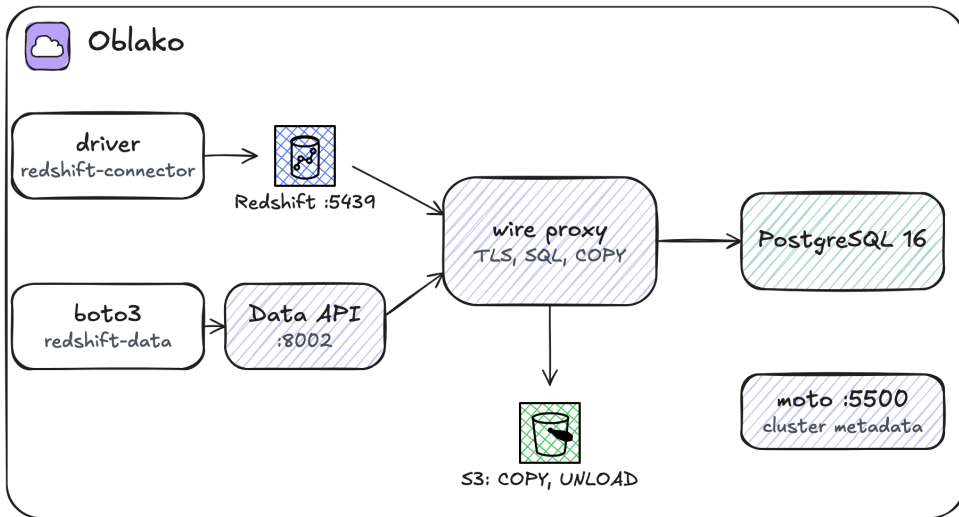


Figure 7.1: Redshift on oblako. Both the driver and the Data API reach PostgreSQL through the wire proxy, which also reads and writes S3 for COPY and UNLOAD.

```
import redshift_connector

def connect():
    conn = redshift_connector.connect(
        host="localhost",
        port=5439,
        database="oblako",
        user="oblako",
        password="oblako",
        ssl=True,
        sslmode="verify-ca",
    )
    conn.autocommit = True
    return conn

def query(sql):
    with connect().cursor() as cur:
        cur.execute(sql)
        return cur.fetchall() if cur.description else []
```

`redshift-connector` is Amazon's Python driver for Redshift; the same module, with your cluster's endpoint and credentials, connects to AWS.

7.4 Creating tables

This table uses Redshift's physical options: a compression encoding, `varchar(max)`, a distribution key on the user, so each user's views share a slice, and a sort key on the day, so date filters skip blocks:

```
from warehouse import query

query("DROP TABLE IF EXISTS page_views")
query(
    "CREATE TABLE page_views ("
    " day date ENCODE az64,"
    " user_id varchar(16),"
    " page varchar(max),"
    " load_ms integer"
    ") DISTKEY(user_id) SORTKEY(day)"
)
print(
    query(
        'SELECT "column", type FROM pg_table_def'
        " WHERE tablename = 'page_views'"
    )
)
```

```
(['day', 'date'], ['user_id', 'character varying(16)'], ['page', 'text'], ['load_ms', 'integer'])
```

The statement runs unchanged, and so does the DDL that tools such as `dbt`, `awswrangler` and `Alembic` write for Redshift, within the limits listed at the end of the chapter. `pg_table_def`, Redshift's view of a table's columns, shows what was stored: `varchar(max)` became PostgreSQL's `text`, and the encoding and keys were accepted and ignored. A single PostgreSQL has no slices to distribute over and no column blocks to sort.¹

¹`oblako` also has a multi-node variant, built on the Citus extension, in which `DISTKEY` shards a table across worker nodes. It is intended for self-hosting; for development, use the single-node engine.

7.5 Loading with COPY

COPY loads data into Redshift by reading files from S3 in parallel on every slice. First, stage a small Parquet file in S3:

```

import boto3
import pandas as pd
from pyarrow import fs

s3 = boto3.client("s3")
s3.create_bucket(Bucket="wh-staging")
# pandas reads and writes S3 through pyarrow, whose S3 client
# ignores the profile's endpoints, so it gets the one boto3 uses
s3_fs = fs.S3FileSystem(endpoint_override=s3.meta.endpoint_url)
days = [
    "2026-09-01",
    "2026-09-01",
    "2026-09-02",
    "2026-09-02",
    "2026-09-03",
    "2026-09-03",
]
views = pd.DataFrame(
    {
        "day": pd.to_datetime(days).date,
        "user_id": ["u-1", "u-2", "u-1", "u-3", "u-2", "u-3"],
        "page": [
            "/",
            "/pricing",
            "/docs",
            "/",
            "/docs",
            "/pricing",
        ],
        "load_ms": [821, 1430, 2210, 450, 1980, 700],
    }
)
views.to_parquet(
    "wh-staging/page_views/part-0.parquet",
    index=False,
    filesystem=s3_fs,
)
print(len(views), "rows staged")

```

6 rows staged

Then load every file under the prefix:

```

from warehouse import query

query(
    "COPY page_views FROM 's3://wh-staging/page_views/'
    " IAM_ROLE default FORMAT AS PARQUET"
)
print(
    query("SELECT count(*), min(day), max(day) FROM page_views")
)

```

```

([6, datetime.date(2026, 9, 1), datetime.date(2026, 9, 3)],)

```

`IAM_ROLE default` tells Redshift to read S3 with the role attached to the cluster as its default. On obloko the wire proxy reads the files itself without checking the role, while on AWS that role must allow `s3:GetObject` on the prefix.

7.6 Redshift's SQL

Redshift's SQL is derived from PostgreSQL 8 and adds its own functions. These two queries use three of them: `LISTAGG`, which joins a group's values into one string, and `DATEDIFF` and `DATEADD`, which take an unquoted date part such as day or month:

```

from warehouse import query

rows = query(
    "SELECT user_id,"
    " LISTAGG(page, ', ' ) WITHIN GROUP (ORDER BY day) AS path,"
    " DATEDIFF(day, min(day), max(day)) AS days_active"
    " FROM page_views GROUP BY user_id ORDER BY user_id"
)
for row in rows:
    print(row)

print(query("SELECT DATEADD(month, 1, '2026-09-30'::date)"))

```

```

['u-1', '/', '/docs', 1]
['u-2', '/pricing, /docs', 2]
['u-3', '/', '/pricing', 1]
([datetime.datetime(2026, 10, 30, 0, 0)],)

```

oblako's engine and proxy supply these functions, which PostgreSQL lacks in this form, and the queries return Redshift's results.

7.7 JSON with SUPER

SUPER holds semi-structured values: objects, arrays and scalars. JSON_PARSE turns a JSON string into one, and the dot notation of PartiQL, the query language Redshift uses for semi-structured data, reaches inside it:

```
from warehouse import query

query("DROP TABLE IF EXISTS raw_events")
query("CREATE TABLE raw_events (payload SUPER)")
query(
    "INSERT INTO raw_events VALUES"
    ' (JSON_PARSE(\'{"page": "/", '
    ' "device": {"os": "ios"}}\')), '
    ' (JSON_PARSE(\'{"page": "/docs", '
    ' "device": {"os": "linux"}}\'))'
)
print(
    query(
        "SELECT payload.page::varchar, payload.device.os::varchar"
        " FROM raw_events ORDER BY 1"
    )
)
```

```
(['/', 'ios'], ['/docs', 'linux'])
```

Each field is cast to varchar. Without the cast, Redshift returns a SUPER value to the driver as JSON text, so the page would read `"/"` with its quotation marks, while oblako returns the plain string either way. With the cast, both return the rows above.

Storing raw events as SUPER and extracting fields in SQL lets you load first and decide the schema later.

7.8 Unloading to S3

UNLOAD writes a query's result to S3 in parallel, as files other engines can read:

```
import boto3
import pandas as pd
from pyarrow import fs

from warehouse import query

query(
    "UNLOAD ('SELECT day, count(*) AS views"
    " FROM page_views GROUP BY day')"
    " TO 's3://wh-staging/exports/daily_' IAM_ROLE default"
    " FORMAT AS PARQUET ALLOWOVERWRITE"
)
# pandas reads and writes S3 through pyarrow, whose S3 client
# ignores the profile's endpoints, so it gets the one boto3 uses
endpoint = boto3.client("s3").meta.endpoint_url
s3_fs = fs.S3FileSystem(endpoint_override=endpoint)
daily = pd.read_parquet("wh-staging/exports/", filesystem=s3_fs)
print(daily.sort_values("day").to_string(index=False))
```

day	views
2026-09-01	2
2026-09-02	2
2026-09-03	2

The TO location is a prefix, under which Redshift names the files itself. Redshift refuses to write over files it finds there, so a second run of the same UNLOAD fails unless it says ALLOWOVERWRITE, as this one does. Unloading to Parquet is the usual way to hand warehouse data to a lake, where Athena or Spark can read it.

7.9 The Data API

The Data API runs SQL over HTTPS without a driver or an open connection, which suits Lambda functions and scripts. Like Athena, it is asynchronous, so the listing submits a statement, polls until it finishes, and raises the error if it failed:

```

import time

import boto3

rsd = boto3.client("redshift-data")
statement = rsd.execute_statement(
    ClusterIdentifier="analytics",
    Database="oblako",
    DbUser="oblako",
    Sql="SELECT page, avg(load_ms) AS avg_ms,"
    " avg(load_ms::float) AS exact_ms FROM page_views"
    " GROUP BY page ORDER BY avg_ms DESC",
)
while True:
    desc = rsd.describe_statement(Id=statement["Id"])
    if desc["Status"] in ("FINISHED", "FAILED", "ABORTED"):
        break
    time.sleep(0.5)
if desc["Status"] != "FINISHED":
    raise RuntimeError(desc.get("Error", desc["Status"]))

result = rsd.get_statement_result(Id=statement["Id"])
print([col["name"] for col in result["ColumnMetadata"]])
for record in result["Records"]:
    print([list(field.values())[0] for field in record])

['page', 'avg_ms', 'exact_ms']
['/docs', 2095, 2095.0]
['/pricing', 1065, 1065.0]
['/', 635, 635.5]

```

Each field in a record is a one-key dictionary naming its type, such as `{"longValue": 6}`, which is why the listing takes the single value out of each.

The two load times of / add up to 1271 milliseconds, so their average is 635.5, yet `avg_ms` reports 635. `load_ms` is an integer column, and on Redshift `AVG` of an integer returns an integer, `BIGINT`, dropping the fraction. Where the fraction matters, cast first, as `exact_ms` does with `avg(load_ms::float)`.

7.10 SQLAlchemy and pandas

Most Python applications reach a database through SQLAlchemy, and so do pandas, Alembic and many BI tools. For Redshift, Amazon publishes the `sqlalchemy-redshift` dialect, which uses `redshift-connector` underneath:

```
uv add sqlalchemy sqlalchemy-redshift
```

The engine URL names the dialect and the driver. This listing reflects the table's columns from the catalog, builds a query with SQLAlchemy's expression language, and hands the engine to pandas:

```
import pandas as pd
import sqlalchemy as sa

url = "redshift+redshift_connector://oblako:oblako@localhost:5439"
engine = sa.create_engine(
    f"{url}/oblako",
    connect_args={"ssl": True, "sslmode": "verify-ca"},
)

columns = sa.inspect(engine).get_columns("page_views")
print([(c["name"], str(c["type"])) for c in columns])

page_views = sa.Table(
    "page_views", sa.MetaData(), autoload_with=engine
)
query = (
    sa.select(page_views.c.page, sa.func.count().label("views"))
    .group_by(page_views.c.page)
    .order_by(sa.desc("views"), page_views.c.page)
)
with engine.connect() as conn:
    for row in conn.execute(query):
        print(row)

daily = pd.read_sql(
    "SELECT day, avg(load_ms) AS avg_ms"
    " FROM page_views GROUP BY day",
    engine,
)
print(daily.sort_values("day").to_string(index=False))
```

```
[('day', 'DATE'), ('user_id', 'VARCHAR(16)'), ('page', 'TEXT'), ('load_ms', '
    INTEGER')]
('/', 2)
('/docs', 2)
('/pricing', 2)
    day avg_ms
2026-09-01    1125
2026-09-02    1330
2026-09-03    1340
```

To reflect a table, the dialect reads catalog columns that only Redshift has, such as each table's distribution style and each column's encoding, and oblako's proxy answers those queries. Reflection therefore works, and so does Alembic's autogenerate, which compares your models with the live schema to write migrations. The page column reflects as TEXT, as `pg_table_def` showed earlier, and the daily averages are whole numbers, for the reason the Data API section gave.

7.11 Where oblako differs from AWS

- **A row store on one node.** Results match Redshift's, while storage and speed are PostgreSQL's. Distribution keys, sort keys and encodings are accepted and have no effect, so performance tuning has to wait for a real cluster.
- **A few results differ.** `SELECT version()` reports PostgreSQL. SUPER fields read with dot notation and no cast come back as plain text, where Redshift returns JSON text with quotation marks around strings; cast them, as the listing does, and both agree. `varchar(max)` is stored as text, where Redshift stores `varchar(65535)`, so a schema reflected from oblako, by Alembic's autogenerate for example, shows a different type for those columns than one reflected from AWS.
- **COPY and UNLOAD** handle Parquet, CSV and delimited text, and run without checking the IAM role. UNLOAD also writes over existing files without `ALLOWOVERWRITE`, where Redshift fails.
- **Unsupported features.** Late-binding views, which dbt creates for views configured with `bind: false`, VARBYTE, workload management and concurrency scaling are unavailable.

- **Python UDFs still run**, on Python 3, but Redshift stopped supporting them after June 30, 2026, in favor of Lambda UDFs, so code that relies on them works only on oblako.²
- **The cluster is metadata.** `describe_clusters` answers from moto, and the same engine serves whatever node type you ask for.

7.12 Running it on AWS

Point `warehouse.py` at your cluster or Serverless workgroup endpoint, with a database user and password, or with IAM authentication, which `redshift-connector` supports. Attach an IAM role to the cluster that can read and write the staging bucket, and make it the default role, or name it in `IAM_ROLE`. For the Data API, pass `WorkgroupName` for Serverless, or `ClusterIdentifier` with `DbUser` or a Secrets Manager `SecretArn` for a provisioned cluster. A provisioned cluster bills from the moment it starts, whether you query it or not, so pause or delete it when you finish.

7.13 Summary

Redshift is a columnar, parallel warehouse: distribution and sort keys decide how rows are spread and ordered, and `COPY` and `UNLOAD` move data in bulk through S3. Its SQL adds functions such as `LISTAGG` and `DATEADD`, the `SUPER` type for JSON, and the Data API for SQL over HTTPS. On oblako, PostgreSQL behind a wire proxy runs that SQL and returns Redshift's results, with a short list of differences; storage and performance are PostgreSQL's.

²Redshift's list of behavior changes, <https://docs.aws.amazon.com/redshift/latest/mgmt/behavior-changes.html>, ends the creation of new Python UDFs after October 30, 2025, and its UDF documentation, such as <https://docs.aws.amazon.com/redshift/latest/dg/udf-example-uses.html>, ends their use after June 30, 2026.



RDS and Aurora

Aurora PostgreSQL: connecting, writing, pgvector and the Data API.

8.1 Before you start

```
oblako up rds
oblako up rds-data
export AWS_PROFILE=oblako
```

```
uv add boto3 "psycopg[binary]"
```

The examples keep a small shop's product catalog and the embeddings of a few documentation pages.

8.2 RDS and Aurora in brief

RDS gives you a database **instance**: one server running PostgreSQL, MySQL, MariaDB, SQL Server or Oracle, which AWS installs, patches, backs up and fails over. You choose its size, and pay for it while it runs.

Aurora is a **cluster**: one writer instance and up to fifteen readers, all on a single storage volume that AWS replicates across three availability zones. Because readers share the writer's storage, adding one copies no data, and a reader can take over as writer in seconds. A cluster has two endpoints: the **writer endpoint** always reaches the writer, and the **reader endpoint** spreads connections over the readers. Aurora PostgreSQL speaks the PostgreSQL wire protocol, so PostgreSQL drivers and tools such as `psycpg` and `psql` connect to it as they would to PostgreSQL.

In both cases your application connects to a PostgreSQL server. AWS adds the control plane that creates and resizes it and, for Aurora, the **Data API**, which takes SQL over HTTPS.

What oblako runs

oblako runs one real PostgreSQL 16 with the `pgvector` extension, the same engine RDS and Aurora PostgreSQL run, on port 5432. The RDS API on port 8014 keeps its metadata in moto, so the Aurora clusters and cluster instances you create exist as metadata, and all of them lead to this one database. (A standalone PostgreSQL instance, created without a cluster, gets a server of its own; Chapter 9 uses those.) The Data API is an oblako engine on port 8006 that runs its statements on the same PostgreSQL. Figure 8.1 shows the three paths.

8.3 Creating an Aurora cluster

On AWS you create a database through the control plane. Create a cluster, then the writer instance in it:

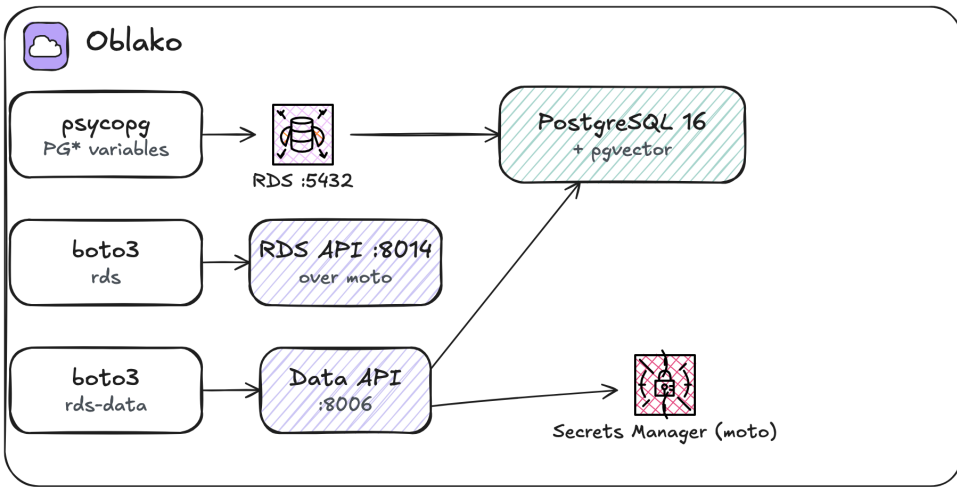


Figure 8.1: RDS and Aurora on oblako. A driver reaches PostgreSQL directly, the RDS API keeps the cluster metadata in moto, and the Data API engine runs its SQL on the same database, with credentials from Secrets Manager in moto.

```
import boto3

rds = boto3.client("rds")
rds.create_db_cluster(
    DBClusterIdentifier="catalog",
    Engine="aurora-postgresql",
    EngineVersion="16.4",
    MasterUsername="oblako",
    MasterUserPassword="oblako-local-1",
    DatabaseName="oblako",
    EnableHttpEndpoint=True,
)
rds.create_db_instance(
    DBInstanceIdentifier="catalog-writer",
    DBClusterIdentifier="catalog",
    DBInstanceClass="db.r6g.large",
    Engine="aurora-postgresql",
)
cluster = rds.describe_db_clusters(DBClusterIdentifier="catalog")
c = cluster["DBClusters"][0]
print(c["Status"], c["Engine"], c["EngineVersion"])
print("writer", c["Endpoint"])
print("reader", c["ReaderEndpoint"])
members = c["DBClusterMembers"]
print("members:", [m["DBInstanceIdentifier"] for m in members])
```

```

available aurora-postgresql 16.4
writer catalog.cluster-jpzuh2ls12s9.us-east-1.rds.amazonaws.com
reader catalog.cluster-ro-jpzuh2ls12s9.us-east-1.rds.amazonaws.com
members: ['catalog-writer']

```

`EnableHttpEndpoint` turns on the Data API for the cluster. The endpoints have AWS's format but exist only as moto metadata, so they do not resolve. On oblako the database is always at `localhost:5432`, while on AWS you connect to the writer endpoint. The connection module in the next section keeps this difference out of the code.

8.4 Connecting

PostgreSQL drivers built on libpq, which includes psycopg, read their connection settings from standard environment variables when the code gives none: `PGHOST`, `PGPORT`, `PGDATABASE`, `PGUSER` and `PGPASSWORD`. They are not AWS settings, so the oblako profile does not set them; export them for oblako:

```

export PGHOST=localhost PGPORT=5432 PGDATABASE=oblako
export PGUSER=oblako PGPASSWORD=oblako

```

If another PostgreSQL already listens on port 5432 on your machine, `oblako up rds` stops with an error that names the port; stop that server first.

psycopg reads every connection setting from these variables, so the listings call `psycopg.connect()` with no settings of their own, and they run against AWS once the variables hold the writer endpoint and your credentials.

8.5 Writing data

Aurora PostgreSQL is PostgreSQL, so this listing is ordinary PostgreSQL. It creates a table, inserts with `RETURNING` to get the stored row back, updates an existing row with an upsert, and writes two rows in one transaction:

```

import psycopg

with psycopg.connect(autocommit=True) as conn:
    conn.execute("DROP TABLE IF EXISTS products")
    conn.execute(
        "CREATE TABLE products ("
        " sku text PRIMARY KEY,"
        " name text NOT NULL,"
        " price numeric(10, 2) NOT NULL)"
    )
    row = conn.execute(
        "INSERT INTO products VALUES (%s, %s, %s)"
        " RETURNING sku, price",
        ("TEA-01", "Green tea", 4.50),
    ).fetchone()
    print("inserted:", row)

    upsert = (
        "INSERT INTO products VALUES (%s, %s, %s)"
        " ON CONFLICT (sku) DO UPDATE SET price = EXCLUDED.price"
    )
    conn.execute(upsert, ("TEA-01", "Green tea", 4.95))

    with conn.transaction():
        conn.execute(upsert, ("MUG-01", "Mug", 12.00))
        conn.execute(upsert, ("POT-01", "Teapot", 29.00))

    rows = conn.execute(
        "SELECT sku, price FROM products ORDER BY sku"
    )
    for row in rows:
        print(row)

```

```

inserted: ('TEA-01', Decimal('4.50'))
('MUG-01', Decimal('12.00'))
('POT-01', Decimal('29.00'))
('TEA-01', Decimal('4.95'))

```

`ON CONFLICT ... DO UPDATE` makes the write idempotent: running it twice leaves one row with the latest price. `conn.transaction()` commits both rows together, or rolls both back if anything inside the block raises an exception.

8.6 Embeddings with pgvector

RDS and Aurora PostgreSQL include **pgvector**, an extension that adds a vector column type, distance operators and indexes for nearest-neighbor search. It lets you keep embeddings next to the rows they describe and search them with SQL. This listing stores the same three hand-made vectors as the S3 Vectors listing in Chapter 4:

```
import psycopgpg

pages = [
    ("pricing", "Plans and prices", "[0.9, 0.1, 0.0]"),
    ("quickstart", "Install and first run", "[0.1, 0.9, 0.2]"),
    ("api", "API reference", "[0.0, 0.8, 0.6]"),
]

with psycopgpg.connect(autocommit=True) as conn:
    conn.execute("CREATE EXTENSION IF NOT EXISTS vector")
    conn.execute("DROP TABLE IF EXISTS doc_pages")
    conn.execute(
        "CREATE TABLE doc_pages ("
        " slug text PRIMARY KEY, title text, embedding vector(3))"
    )
    with conn.cursor() as cur:
        cur.executemany(
            "INSERT INTO doc_pages VALUES (%s, %s, %s)", pages
        )
    conn.execute(
        "CREATE INDEX ON doc_pages"
        " USING hnsw (embedding vector_cosine_ops)"
    )

    query = "[0.1, 1.0, 0.3]"
    nearest = conn.execute(
        "SELECT slug,"
        " round((embedding <=> %s::vector)::numeric, 4)"
        " FROM doc_pages"
        " ORDER BY embedding <=> %s::vector LIMIT 2",
        (query, query),
    ).fetchall()
    print(nearest)
```

```
[('quickstart', Decimal('0.0027')), ('api', Decimal('0.0656'))]
```

`<=>` is cosine distance, and the **HNSW** (hierarchical navigable small world) index, a graph-based index, answers an `ORDER BY` on it without comparing the query with every row.

The distances match S3 Vectors' to four places, since the vectors and the measure are the same. pgvector keeps vectors in the same transaction as your other data and lets you join and filter them with SQL. S3 Vectors, by contrast, stores many more vectors at lower cost and has no server to run.

8.7 SQLAlchemy

SQLAlchemy works with RDS and Aurora PostgreSQL as with any PostgreSQL. Its psycopg dialect passes only what the URL contains, so a URL with no host or credentials takes them from the PG* variables. pgvector ships a SQLAlchemy column type, `Vector`, with the distance functions as methods:

```
uv add sqlalchemy pgvector
```

This listing maps the two tables from the previous sections as ORM classes, upserts a product with PostgreSQL's `INSERT ... ON CONFLICT`, queries through the ORM, and runs the nearest-neighbor search as an ORM query:

```

from decimal import Decimal

import sqlalchemy as sa
from pgvector.sqlalchemy import Vector
from sqlalchemy.dialects.postgresql import insert
from sqlalchemy.orm import (
    DeclarativeBase,
    Mapped,
    Session,
    mapped_column,
)

# no host, port or credentials: psycopg reads the PG* variables
engine = sa.create_engine("postgresql+psycopg://")

class Base(DeclarativeBase):
    pass

class Product(Base):
    __tablename__ = "products"
    sku: Mapped[str] = mapped_column(primary_key=True)
    name: Mapped[str]
    price: Mapped[Decimal] = mapped_column(sa.Numeric(10, 2))

class DocPage(Base):
    __tablename__ = "doc_pages"
    slug: Mapped[str] = mapped_column(primary_key=True)
    title: Mapped[str]
    embedding = mapped_column(Vector(3))

with Session(engine) as session:
    upsert = insert(Product).values(
        sku="CUP-01", name="Cup", price=Decimal("6.50")
    )
    session.execute(
        upsert.on_conflict_do_update(
            index_elements=["sku"],
            set_={"price": upsert.excluded.price},
        )
    )
    session.commit()

```

```

cheap = session.scalars(
    sa.select(Product)
    .where(Product.price < 10)
    .order_by(Product.sku)
)
print([(p.sku, p.price) for p in cheap])

query = [0.1, 1.0, 0.3]
nearest = session.scalars(
    sa.select(DocPage)
    .order_by(DocPage.embedding.cosine_distance(query))
    .limit(2)
)
print([page.slug for page in nearest])

```

```

[('CUP-01', Decimal('6.50')), ('TEA-01', Decimal('4.95'))]
['quickstart', 'api']

```

`cosine_distance` compiles to the same `<=>` operator as before, so the HNSW index serves the ORM query too.

8.8 The Data API

The Data API runs SQL against an Aurora cluster over HTTPS, without a driver or an open connection, which suits Lambda functions, where connections are expensive to hold. Each call names the cluster by its ARN and authenticates with a Secrets Manager secret that holds the database credentials:

```

import json

import boto3

secrets = boto3.client("secretsmanager")
secret_arn = secrets.create_secret(
    Name="catalog/oblako",
    SecretString=json.dumps(
        {"username": "oblako", "password": "oblako"}
    ),
)["ARN"]
rds = boto3.client("rds")
cluster = rds.describe_db_clusters(DBClusterIdentifier="catalog")
cluster_arn = cluster["DBClusters"][0]["DBClusterArn"]

rds_data = boto3.client("rds-data")
target = {
    "resourceArn": cluster_arn,
    "secretArn": secret_arn,
    "database": "oblako",
}

tx = rds_data.begin_transaction(**target)["transactionId"]
rds_data.execute_statement(
    **target,
    transactionId=tx,
    sql="UPDATE products SET price = price * 1.1"
    " WHERE sku = :sku",
    parameters=[
        {"name": "sku", "value": {"stringValue": "MUG-01"}}
    ],
)
done = rds_data.commit_transaction(
    resourceArn=cluster_arn,
    secretArn=secret_arn,
    transactionId=tx,
)
print(done["transactionStatus"])

result = rds_data.execute_statement(
    **target,
    sql="SELECT sku, price FROM products ORDER BY sku",
    formatRecordsAs="JSON",
)
print(json.loads(result["formattedRecords"]))

```

```
Transaction Committed
[{'sku': 'MUG-01', 'price': '13.20'},
 {'sku': 'POT-01', 'price': '29.00'},
 {'sku': 'TEA-01', 'price': '4.95'}]
```

Parameters are named, as in `:sku`, and typed, as in `stringValue`, which keeps values out of the SQL text. A transaction spans calls: `begin_transaction` returns an ID that later statements pass until `commit_transaction` or `rollback_transaction` ends it. `formatRecordsAs="JSON"` returns the rows as one JSON string instead of typed fields, and the prices in it are strings, because the Data API returns `DECIMAL` as a string so that no precision is lost. To get numbers, pass:

```
resultSetOptions = {"decimalReturnType": "DOUBLE_OR_LONG"}
```

8.9 Where oblako differs from AWS

- **One database behind every cluster.** Clusters, their instances, readers and endpoints are metadata, and any two clusters you create share one PostgreSQL, so use separate databases or schemas if you need to keep them apart.
- **Always PostgreSQL 16**, whatever Engine and EngineVersion you request. oblako can run MySQL instead, but only when started from its Python API, `RdsService(engine="mysql")`.
- **Aurora readers, failover and backups are metadata.** The calls succeed and are recorded, but the database stays as it was. Read replicas and parameter groups of standalone instances are real (Chapter 9).
- **Credentials are not checked.** The control plane accepts any credentials, and the Data API runs statements without reading the secret it is given.
- **Extensions:** `pgvector` is installed; other extensions RDS offers may not be.

8.10 Running it on AWS

On AWS the control-plane listing creates a real Aurora cluster, which bills by the hour per instance, so delete the instance and the cluster afterwards. Point the PG* variables at

the cluster's writer endpoint and your credentials. Your machine reaches the cluster only if its instances are publicly accessible and their security group allows your address; otherwise, run the listings from inside the VPC. Store the secret for the Data API with `create_secret` as shown, and grant your role `rds-data:ExecuteStatement` and `secrets-manager:GetSecretValue` on it. `pgvector` is enabled per database with the same `CREATE EXTENSION vector`.

8.11 Summary

RDS runs a database instance for you; Aurora runs a cluster of instances over shared storage, with writer and reader endpoints. Your code speaks PostgreSQL to either, and with the `PG*` variables set it leaves the host to the environment. `pgvector` adds vector columns and nearest-neighbor indexes inside the database, and the Data API runs SQL over HTTPS with named parameters, transactions and JSON results. On `oblako`, all of it is one PostgreSQL 16 with `pgvector`, under `moto`'s cluster metadata.

9

Replicating PostgreSQL

Read replicas and logical replication between RDS for PostgreSQL instances.

9.1 Before you start

```
oblako up rds
export AWS_PROFILE=oblako
```

```
uv add boto3 "psycopg[binary]"
```

The examples use a table of shop orders that the listings create. Because each listing creates or uses RDS instances by name, run the listings in order; the last section removes everything they create.

9.2 Physical and logical replication

PostgreSQL writes every change to its **write-ahead log** (WAL) before it changes the data files, and both kinds of replication in this chapter start from that log.

Physical replication sends the WAL itself to a standby server, which replays it into an exact copy of the primary's data files: every database, table, index and role, at the same PostgreSQL major version. A standby that serves read-only queries while it replays is a **hot standby**, and it can be **promoted** to a primary of its own. An RDS read replica is a physical standby fed by streaming replication. The stream is asynchronous, which means the primary commits without waiting for the replica, so the replica can lag behind.

Logical replication decodes the WAL into row changes for the tables in a **publication**, the set of tables a server makes available to others. A **subscription** on another server receives those changes and applies them to its own tables. The subscriber is an ordinary, writable database that can hold other tables and indexes and can run a newer major version. That makes logical replication the usual way to upgrade with little downtime or to feed an analytics database. It copies rows only, so the tables must exist on both sides, and CREATE and ALTER TABLE statements are made on each side separately.

On RDS you create a read replica with a control-plane call, a request to the RDS API, whereas logical replication is SQL you run yourself once a parameter group sets `rds.logical_replication`.

What oblako runs

Each PostgreSQL instance you create through the RDS API on oblako is its own PostgreSQL 16 container with its own data, built from the same image as Chapter 8. The RDS API on port 8014 forwards every call to moto, which keeps the metadata, and then starts, restarts, promotes and removes the containers to match. An instance's endpoint is `<id>.<region>.rds.localhost` on a port of its own. Your operating system resolves any name under `localhost` to your own machine. The containers share a Docker network on which the same name reaches the same instance, so one endpoint works both from your code and between instances (Figure 9.1). Aurora clusters keep the shared database of Chapter 8.

9.3 The primary

The primary is a standalone instance, so it needs no cluster. The examples connect to instances by name and look up the endpoint each time; save that lookup as `db.py`:

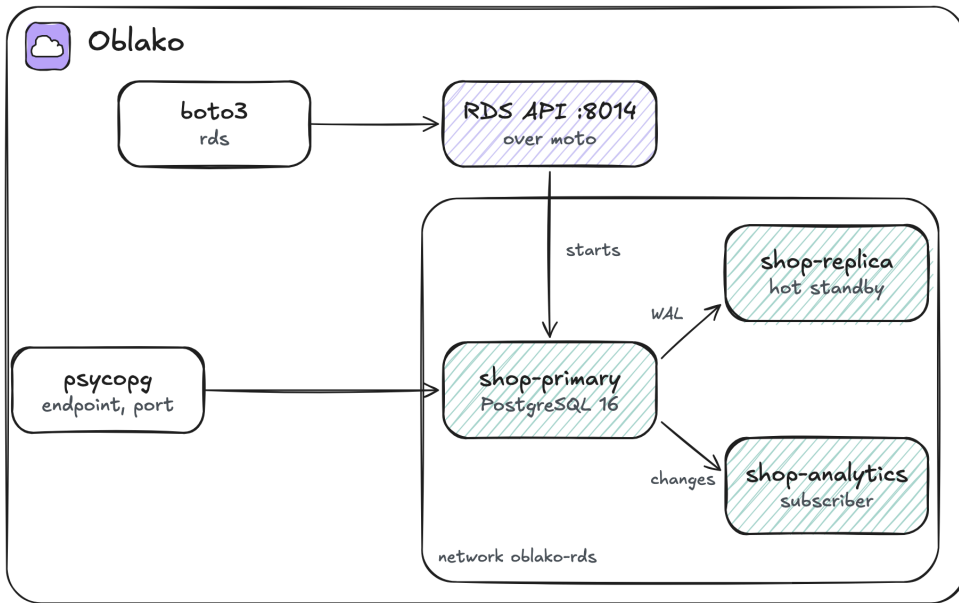


Figure 9.1: Replication on oblako. The RDS API starts one PostgreSQL container per instance, and each is reached at its own endpoint. The replica streams the primary's WAL; the subscriber receives decoded changes.

```

import boto3
import psycopg

rds = boto3.client("rds")
USER, PASSWORD = "shop", "shop-local-1"

def describe(instance_id):
    found = rds.describe_db_instances(
        DBInstanceIdentifier=instance_id
    )
    return found["DBInstances"][0]

def wait(instance_id):
    waiter = rds.get_waiter("db_instance_available")
    waiter.wait(DBInstanceIdentifier=instance_id)
    return describe(instance_id)

```

```

def endpoint(instance_id):
    return describe(instance_id)["Endpoint"]

def connect(instance_id):
    e = endpoint(instance_id)
    return psycopg2.connect(
        host=e["Address"],
        port=e["Port"],
        dbname="shop",
        user=USER,
        password=PASSWORD,
        autocommit=True,
    )

def one(conn, sql, params=None):
    # the first column of the first row
    return conn.execute(sql, params).fetchone()[0]

```

Create the primary and wait for it:

```

from db import PASSWORD, USER, rds, wait

rds.create_db_instance(
    DBInstanceIdentifier="shop-primary",
    Engine="postgres",
    EngineVersion="16",
    DBInstanceClass="db.t4g.micro",
    AllocatedStorage=20,
    MasterUsername=USER,
    MasterUserPassword=PASSWORD,
    DBName="shop",
    BackupRetentionPeriod=1,
)
db = wait("shop-primary")
print(db["DBInstanceStatus"], db["Engine"], db["EngineVersion"])
print(db["Endpoint"]["Address"])

```

```

available postgres 16.15
shop-primary.us-east-1.rds.amazonaws.com

```

EngineVersion="16" asks for the newest PostgreSQL 16 release, and RDS reports the one it chose. BackupRetentionPeriod must be above zero on an instance you want to replicate, because RDS builds a read replica from the source's backups and refuses to create one without them. The waiter polls DescribeDBInstances until the status is available, which takes several minutes on AWS and a few seconds on oblako.

Load a thousand orders:

```
from db import connect

with connect("shop-primary") as conn:
    conn.execute("""
        CREATE TABLE orders (
            id          bigint PRIMARY KEY,
            customer    text NOT NULL,
            total       numeric(10, 2) NOT NULL,
            placed_at   timestamptz NOT NULL DEFAULT now()
        )""")
    conn.execute("""
        INSERT INTO orders (id, customer, total)
        SELECT g, 'customer-' || (g % 50),
               round((g % 97) * 1.25, 2)
        FROM generate_series(1, 1000) AS g""")
    totals = "SELECT count(*), sum(total) FROM orders"
    print(conn.execute(totals).fetchone())
```

```
(1000, Decimal('58781.25'))
```

9.4 Read replicas

One call creates a replica:

```

from db import rds, wait

rds.create_db_instance_read_replica(
    DBInstanceIdentifier="shop-replica",
    SourceDBInstanceIdentifier="shop-primary",
)
db = wait("shop-replica")
source = db["ReadReplicaSourceDBInstanceIdentifier"]
print(db["DBInstanceStatus"], "replica of", source)
print(db["Endpoint"]["Address"])

```

```

available replica of shop-primary
shop-replica.us-east-1.rds.amazonaws.com

```

The replica has its own endpoint. The primary lists it in `pg_stat_replication`, which has one row per connected standby, and the replica reports that it is a standby:

```

import psycopg

from db import connect, one

RECOVERY = "SELECT pg_is_in_recovery()"

with connect("shop-primary") as primary:
    standbys = primary.execute(
        "SELECT state, sync_state FROM pg_stat_replication"
    )
    print("primary sends to:", standbys.fetchall())

with connect("shop-replica") as replica:
    print("in recovery:", one(replica, RECOVERY))
    print("orders:", one(replica, "SELECT count(*) FROM orders"))
    try:
        replica.execute("DELETE FROM orders")
    except psycopg.errors.ReadOnlySqlTransaction as e:
        print("refused:", e)

```

```

primary sends to: [('streaming', 'async')]
in recovery: True
orders: 1000
refused: cannot execute DELETE in a read-only transaction

```

`pg_is_in_recovery()` returns true on a standby, because the server is still replaying WAL. All 1,000 orders are there, copied when the replica was created, and the replica refuses any write.

9.5 Read-your-writes consistency

Because the stream is asynchronous, a row you have just committed on the primary may not be on the replica yet. An application that writes and then reads from a replica, say to show a customer the order they placed, can get an empty result. The fix uses the position in the WAL. After a write, ask the primary for its current **log sequence number** (LSN), and before reading, wait until the replica has replayed past that position:

```
import time

from db import connect, one

with connect("shop-primary") as primary:
    primary.execute(
        "INSERT INTO orders (id, customer, total)"
        " VALUES (1001, 'customer-7', 42.00)"
    )
    lsn = one(primary, "SELECT pg_current_wal_lsn()")

caught_up = "SELECT pg_last_wal_replay_lsn() >= %s::pg_lsn"
with connect("shop-replica") as replica:
    polls = 0
    while not one(replica, caught_up, (lsn,)):
        time.sleep(0.01)
        polls += 1
    row = replica.execute(
        "SELECT customer, total FROM orders WHERE id = 1001"
    ).fetchone()
    print("replayed", lsn, "after", polls, "polls:", row)
```

```
replayed 0/303DA58 after 0 polls: ('customer-7', Decimal('42.00'))
```

On oblako the replica had caught up before the first poll; on AWS, expect a few milliseconds of polling within a region, and more under heavy writes.

9.6 Replication lag

Both servers report lag:

```
from db import connect, one

with connect("shop-primary") as primary:
    bytes_behind, replay_lag = primary.execute("""
        SELECT pg_wal_lsn_diff(pg_current_wal_lsn(), replay_lsn),
               replay_lag
        FROM pg_stat_replication""").fetchone()
    print("bytes behind:", bytes_behind)
    print("replay lag:", replay_lag)

with connect("shop-replica") as replica:
    since = "SELECT now() - pg_last_xact_replay_timestamp()"
    print("last replayed commit:", one(replica, since), "ago")
```

```
bytes behind: 0
replay lag: 0:00:00.000472
last replayed commit: 0:00:00.295249 ago
```

The primary reports how many bytes of WAL the replica has yet to replay and, as `replay_lag`, how long recent WAL took to be replayed there. The replica reports how long ago the last transaction it replayed was committed, a number that keeps growing on an idle primary even when the replica is fully caught up, because there is nothing new to replay. RDS computes its `ReplicaLag` CloudWatch metric the same way, and its documentation warns that an idle source can show up to five minutes of lag. The byte count is therefore the reliable measure of how far behind a replica is.

9.7 Promotion

Promotion turns a replica into a standalone instance that accepts writes, either to recover from losing the primary or to split off a copy of a database. Promotion is permanent:

```

from db import connect, one, rds, wait

rds.promote_read_replica(DBInstanceIdentifier="shop-replica")
db = wait("shop-replica")
source = db.get("ReadReplicaSourceDBInstanceIdentifier")
print(db["DBInstanceStatus"], "source:", source)

COUNT = "SELECT count(*) FROM orders"
RECOVERY = "SELECT pg_is_in_recovery()"
with connect("shop-replica") as promoted:
    print("in recovery:", one(promoted, RECOVERY))
    promoted.execute("DELETE FROM orders WHERE id = 1001")
    print("orders:", one(promoted, COUNT))

with connect("shop-primary") as primary:
    print("primary still has:", one(primary, COUNT))

```

```

available source: None
in recovery: False
orders: 1000
primary still has: 1001

```

The promoted instance is out of recovery and has no source. From the moment of promotion the two databases diverge, so the order deleted on the promoted instance is still on the primary.

9.8 Enabling logical replication

Logical decoding needs `wal_level = logical`, which RDS lets you change only through its own parameter. Set `rds.logical_replication = 1` in a parameter group, attach the group to the instance and reboot it, since the setting is read only at startup:

```

from db import connect, one, rds, wait

rds.create_db_parameter_group(
    DBParameterGroupName="shop-logical",
    DBParameterGroupFamily="postgres16",
    Description="PostgreSQL 16 with logical replication",
)
rds.modify_db_parameter_group(
    DBParameterGroupName="shop-logical",
    Parameters=[
        {
            "ParameterName": "rds.logical_replication",
            "ParameterValue": "1",
            "ApplyMethod": "pending-reboot",
        }
    ],
)
rds.modify_db_instance(
    DBInstanceIdentifier="shop-primary",
    DBParameterGroupName="shop-logical",
    ApplyImmediately=True,
)
wait("shop-primary")
rds.reboot_db_instance(DBInstanceIdentifier="shop-primary")
wait("shop-primary")

with connect("shop-primary") as primary:
    print("wal_level:", one(primary, "SHOW wal_level"))

```

```
wal_level: logical
```

The first wait is needed on AWS, where an instance must finish being modified before it can be rebooted. A higher `wal_level` writes more WAL, so turn it on only on instances that publish.

9.9 Subscriptions

The subscriber is a second, ordinary instance:

```
from db import PASSWORD, USER, rds, wait

rds.create_db_instance(
    DBInstanceIdentifier="shop-analytics",
    Engine="postgres",
    EngineVersion="16",
    DBInstanceClass="db.t4g.micro",
    AllocatedStorage=20,
    MasterUsername=USER,
    MasterUserPassword=PASSWORD,
    DBName="shop",
)
db = wait("shop-analytics")
print(db["DBInstanceIdentifier"], "is", db["DBInstanceStatus"])
```

```
shop-analytics is available
```

The next listing publishes the orders table on the primary, then creates the same table on the subscriber and subscribes to the publication, with the primary's endpoint as the connection string:

```

import time

from db import PASSWORD, USER, connect, endpoint, one

COUNT = "SELECT count(*) FROM orders"
ORDERS = """
    CREATE TABLE orders (
        id          bigint PRIMARY KEY,
        customer    text NOT NULL,
        total       numeric(10, 2) NOT NULL,
        placed_at   timestampz NOT NULL DEFAULT now()
    )"""

with connect("shop-primary") as primary:
    primary.execute(
        "CREATE PUBLICATION shop_orders FOR TABLE orders"
    )

source = endpoint("shop-primary")
dsn = (
    f"host={source['Address']} port={source['Port']} "
    f"dbname=shop user={USER} password={PASSWORD}"
)
copying = """
    SELECT bool_or(srsubstate <> 'r') FROM pg_subscription_rel"""
with connect("shop-analytics") as analytics:
    analytics.execute(ORDERS)
    analytics.execute(
        f"CREATE SUBSCRIPTION shop_orders CONNECTION '{dsn}' "
        " PUBLICATION shop_orders"
    )
    while one(analytics, copying):
        time.sleep(0.2)
    print("copied:", one(analytics, COUNT))

```

```
copied: 1001
```

CREATE SUBSCRIPTION creates a **replication slot** on the publisher, which records how far the subscriber has received changes, then copies the table's current rows and streams every later change. pg_subscription_rel tracks each table's state, and r (ready) means the initial copy is done. All 1,001 orders arrived, including the one deleted from the promoted instance, because that delete was made on a separate database.

After the initial copy, changes replicate as they commit. The next listing inserts, updates and deletes rows on the primary, waits until the slot confirms that the subscriber has received the changes, and then reads the subscriber:

```

import time

from db import connect, one

confirmed = """
    SELECT confirmed_flush_lsn >= %s::pg_lsn
    FROM pg_replication_slots WHERE slot_name = 'shop_orders'"""

with connect("shop-primary") as primary:
    primary.execute(
        "INSERT INTO orders (id, customer, total)"
        " VALUES (2001, 'customer-3', 19.99)"
    )
    primary.execute(
        "UPDATE orders SET total = total * 2 WHERE id = 1"
    )
    primary.execute("DELETE FROM orders WHERE id = 2")
    lsn = one(primary, "SELECT pg_current_wal_lsn()")
    while not one(primary, confirmed, (lsn,)):
        time.sleep(0.1)

with connect("shop-analytics") as analytics:
    rows = analytics.execute(
        "SELECT id, total FROM orders"
        " WHERE id IN (1, 2, 2001) ORDER BY id"
    )
    print(rows.fetchall())
    count = one(analytics, "SELECT count(*) FROM orders")
    print("orders:", count)

```

```

[(1, Decimal('2.50')), (2001, Decimal('19.99'))]
orders: 1001

```

Order 1 doubled to 2.50, order 2 is gone and order 2001 arrived. On the subscriber, updates and deletes find their row by its **replica identity**, the columns that identify a row in the change stream, which here is the table's primary key. A published table without a primary key needs a replica identity set by hand before the primary accepts updates and deletes on it.

9.10 Adding tables

Because logical replication copies only rows, a new table is created on both sides, added to the publication, and picked up by refreshing the subscription:

```

import time

from db import connect, one

CUSTOMERS = """
    CREATE TABLE customers (
        name      text PRIMARY KEY,
        segment text NOT NULL
    )"""

with connect("shop-primary") as primary:
    primary.execute(CUSTOMERS)
    primary.execute("""
        INSERT INTO customers
        SELECT 'customer-' || g,
               CASE WHEN g % 5 = 0 THEN 'wholesale'
                   ELSE 'retail' END
        FROM generate_series(0, 49) AS g""")
    primary.execute(
        "ALTER PUBLICATION shop_orders ADD TABLE customers"
    )

copying = """
    SELECT bool_or(srsubstate <> 'r') FROM pg_subscription_rel"""
with connect("shop-analytics") as analytics:
    analytics.execute(CUSTOMERS)
    analytics.execute(
        "ALTER SUBSCRIPTION shop_orders REFRESH PUBLICATION"
    )
    while one(analytics, copying):
        time.sleep(0.2)
    segments = analytics.execute("""
        SELECT segment, count(*) FROM customers
        GROUP BY segment ORDER BY segment""")
    print(segments.fetchall())

```

```
[('retail', 40), ('wholesale', 10)]
```

The refresh copies the table's existing rows, and from then on its changes stream with the orders. To add a column, add it on the subscriber first and then on the publisher, so that the stream only ever carries columns the subscriber already has.

9.11 Replication slots

A replication slot makes the publisher keep every WAL segment the subscriber has not yet confirmed. This protects the subscriber from missing changes. It is also the commonest way to fill a database's disk, because a subscriber that goes away without dropping its subscription leaves the slot behind and the publisher keeps WAL for it indefinitely, as AWS's documentation warns. Check the slots, and drop subscriptions you no longer need:

```
from db import connect

SLOTS = """
    SELECT slot_name, slot_type, active
    FROM pg_replication_slots"""

with connect("shop-primary") as primary:
    print("before:", primary.execute(SLOTS).fetchall())

with connect("shop-analytics") as analytics:
    analytics.execute("DROP SUBSCRIPTION shop_orders")

with connect("shop-primary") as primary:
    print("after:", primary.execute(SLOTS).fetchall())
    primary.execute("DROP PUBLICATION shop_orders")
```

```
before: [('shop_orders', 'logical', True)]
after: []
```

`DROP SUBSCRIPTION` connected to the publisher and dropped the slot with it, so the drop fails when the publisher cannot be reached. In that case, detach the slot first with `ALTER SUBSCRIPTION ... SET (slot_name = NONE)`, drop the subscription, and remove the slot on the publisher later with `pg_drop_replication_slot`. Investigate any slot whose `active` column stays false for a long time.

9.12 Cleaning up

```
from db import rds

for name in ("shop-analytics", "shop-replica", "shop-primary"):
    try:
        rds.delete_db_instance(
            DBInstanceIdentifier=name, SkipFinalSnapshot=True
        )
    except rds.exceptions.DBInstanceNotFoundFault:
        pass

try:
    rds.delete_db_parameter_group(
        DBParameterGroupName="shop-logical"
    )
except rds.exceptions.DBParameterGroupNotFoundFault:
    pass
```

On oblako, deleting an instance removes its container and its data.

9.13 Where oblako differs from AWS

- **Instances are containers on your machine.** Instance classes, storage size, IOPS and Multi-AZ are recorded and ignored, and every instance runs PostgreSQL 16, whatever version you request. Other engines, and the members of Aurora clusters, stay metadata on the shared database of Chapter 8.
- **The master user is a PostgreSQL superuser.** On RDS it is a member of `rds_superuser`, which includes `rds_replication` and `pg_monitor`, so the listings here work on both. A statement that needs true superuser rights works on oblako and fails on RDS.
- **All instances share one network.** Every instance can reach every other one, whereas on AWS the subscriber reaches the publisher only if a security group allows it.
- **Replicas copy from the primary directly** with `pg_basebackup`, whereas RDS builds them from a backup snapshot. Because oblako keeps no backups, it checks `BackupRetentionPeriod` but does not act on it.

- **Lag is measured in SQL**, as above, because oblako publishes no CloudWatch metrics such as `ReplicaLag`.
- **State lives in moto's memory**. If moto restarts, the containers stay but the RDS API forgets the instances.

9.14 Running it on AWS

The listings run on AWS unchanged, against three `db.t4g.micro` instances that bill by the hour; delete them with the cleanup listing when you finish. Your machine must reach the instances: add `PubliclyAccessible=True` to the create calls and use a security group that admits your address on port 5432, or run the listings from inside the VPC. The subscriber connects to the publisher over the VPC, so the publisher's security group must also admit the subscriber. Creating the replica takes longer than on oblako, because RDS snapshots the source first, and the reboot that applies the parameter group is a short outage for the primary.

9.15 Summary

A read replica replays the primary's WAL into an exact, read-only copy that can lag behind. To read your own writes from it, wait until it has replayed past the primary's LSN, and measure its lag in bytes. Promotion makes the copy a permanently independent database. Logical replication publishes chosen tables as row changes to any writable PostgreSQL database. On RDS it needs `rds.logical_replication` and a reboot, the tables must exist on both sides, and every subscription holds a slot that keeps WAL on the publisher until the subscription is dropped. On oblako each instance is a real PostgreSQL container with its own endpoint, and all of the listings run on your laptop.

10

DynamoDB

Table design, conditional writes, DynamoDB Streams and vector search.

10.1 Before you start

```
oblako up dynamodb  
export AWS_PROFILE=oblako
```

```
uv add "boto3>=1.43.64"
```

Vector search requires boto3 1.43.64 or later. The examples use a table of shop orders and a table of products with embeddings, and the last listing removes both tables.

10.2 DynamoDB in brief

A DynamoDB **table** holds **items**, each a set of attributes up to 400 KB. The only fixed part of an item is its **primary key**: a **partition key**, which decides where the item is stored, and optionally a **sort key**, which orders items that share a partition key. Every other attribute is free-form and can differ from item to item.

The key is also the only efficient way to read items. `GetItem` reads one item by its full key, and `Query` reads the items of one partition key in sort-key order, optionally narrowed by a condition on the sort key. `Scan` reads the whole table, which makes it the operation to avoid on large tables. A DynamoDB design therefore starts from the access patterns, the questions your application asks of the data, and chooses keys so that each question is one `Query`. A question the table's key cannot answer gets a **global secondary index** (GSI), a copy of the table under another key that DynamoDB keeps up to date.

In **on-demand** mode you pay per request and set no capacity in advance, whereas in provisioned mode you reserve read and write capacity. **DynamoDB Streams** is a log of every change to a table's items, kept in order for each item, and **vector indexes** search the embeddings stored on items by similarity.

What oblako runs

oblako runs DynamoDB Local, the downloadable version of DynamoDB that AWS publishes, with its data on a volume. In front of it, an oblako proxy on port 8007 forwards every call and adds two features that DynamoDB Local lacks: vector indexes with `SearchVectors`, and tags. The oblako profile points the dynamodb client at the proxy, while DynamoDB Streams have their own API and endpoint, which the profile points at DynamoDB Local directly on port 8001 (Figure 10.1).

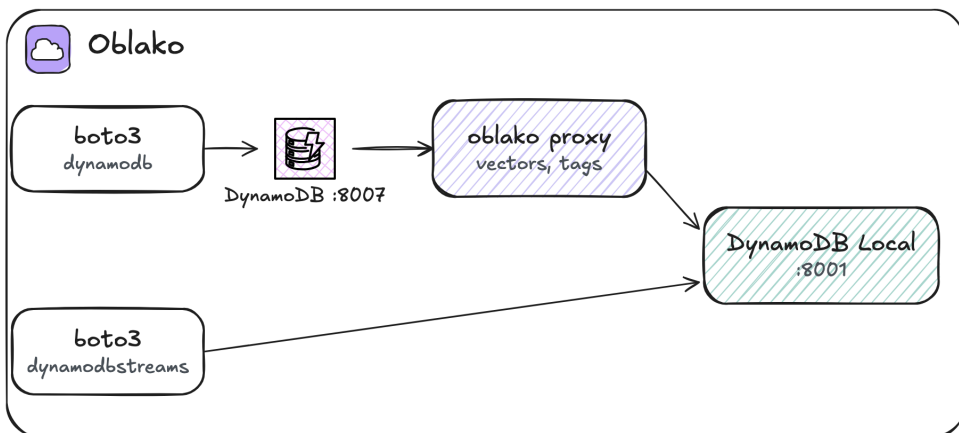


Figure 10.1: DynamoDB on oblako. Table calls go through the proxy, which answers vector searches itself; stream reads go straight to DynamoDB Local.

10.3 Table design

The shop asks two questions of its orders: *what has this customer ordered, newest last?* and *which orders are waiting to ship?* The first is answered by the table's key: partition key `customer`, sort key `order_id`, which starts with the order date so that orders sort by time. The second needs a GSI keyed by `status`:

```

import boto3

dynamodb = boto3.client("dynamodb")
dynamodb.create_table(
    TableName="orders",
    AttributeDefinitions=[
        {"AttributeName": "customer", "AttributeType": "S"},
        {"AttributeName": "order_id", "AttributeType": "S"},
        {"AttributeName": "status", "AttributeType": "S"},
    ],
    KeySchema=[
        {"AttributeName": "customer", "KeyType": "HASH"},
        {"AttributeName": "order_id", "KeyType": "RANGE"},
    ],
    GlobalSecondaryIndexes=[
        {
            "IndexName": "by-status",
            "KeySchema": [
                {"AttributeName": "status", "KeyType": "HASH"},
                {"AttributeName": "order_id", "KeyType": "RANGE"},
            ],
            "Projection": {"ProjectionType": "ALL"},
        }
    ],
    BillingMode="PAY_PER_REQUEST",
    StreamSpecification={
        "StreamEnabled": True,
        "StreamViewType": "NEW_AND_OLD_IMAGES",
    },
)
dynamodb.get_waiter("table_exists").wait(TableName="orders")
table = dynamodb.describe_table(TableName="orders")["Table"]
billing = table["BillingModeSummary"]["BillingMode"]
print(table["TableStatus"], billing)
print("stream:", table["StreamSpecification"]["StreamViewType"])

```

```

ACTIVE PAY_PER_REQUEST
stream: NEW_AND_OLD_IMAGES

```

AttributeDefinitions declares only the attributes used in a key, here three; total and anything else stay undeclared. StreamSpecification turns on the table's stream.

10.4 Reads and writes

The boto3 resource API reads and writes plain Python values, whereas the client API needs every value tagged with its type (`{"S": "..."}`). `batch_writer` groups puts into `BatchWriteItem` calls of up to 25 items and retries any the service returns unprocessed:

```
from decimal import Decimal

import boto3

orders = boto3.resource("dynamodb").Table("orders")
with orders.batch_writer() as batch:
    for n in range(1, 13):
        batch.put_item(
            Item={
                "customer": f"customer-{n % 3}",
                "order_id": f"2026-10-{n:02d}#{n:04d}",
                "total": Decimal(n * 125) / 10,
                "status": "shipped" if n % 4 else "placed",
            }
        )
print("items:", orders.scan(Select="COUNT")["Count"])
```

```
items: 12
```

DynamoDB numbers are exact decimals and the resource API refuses Python floats, so totals are `Decimal` values. A scan is acceptable here only because the table holds twelve items.

The next listing runs both queries and reads one item by its key:

```

import boto3
from boto3.dynamodb.conditions import Key

orders = boto3.resource("dynamodb").Table("orders")

key = {"customer": "customer-1", "order_id": "2026-10-04#0004"}
order = orders.get_item(Key=key)["Item"]
print("get:", order["status"], order["total"])

early = orders.query(
    KeyConditionExpression=Key("customer").eq("customer-1")
    & Key("order_id").begins_with("2026-10-0"),
)
for o in early["Items"]:
    print("customer-1:", o["order_id"], o["status"])

placed = orders.query(
    IndexName="by-status",
    KeyConditionExpression=Key("status").eq("placed"),
)
print("placed:", [o["customer"] for o in placed["Items"]])

```

```

get: placed 50
customer-1: 2026-10-01#0001 shipped
customer-1: 2026-10-04#0004 placed
customer-1: 2026-10-07#0007 shipped
placed: ['customer-1', 'customer-2', 'customer-0']

```

The Query on the table reads one customer's orders in sort-key order, and `begins_with` on the sort key keeps the first nine days of October, which leaves out the same customer's order 0010. The Query on the GSI answers the second question without a scan. Reads from a GSI are eventually consistent, meaning that a write to the table reaches the index a moment later.

10.5 Conditional writes

A **condition expression** makes a write check the item's state atomically, as part of the write itself. The next listing uses one so that two workers that both see order 0004 as placed cannot both ship it:

```

import boto3

orders = boto3.resource("dynamodb").Table("orders")
errors = orders.meta.client.exceptions
key = {"customer": "customer-1", "order_id": "2026-10-04#0004"}

def ship():
    orders.update_item(
        Key=key,
        UpdateExpression="SET #s = :shipped",
        ConditionExpression="#s = :placed",
        ExpressionAttributeNames={"#s": "status"},
        ExpressionAttributeValues={
            ":placed": "placed",
            ":shipped": "shipped",
        },
    )

ship()
print("shipped once")
try:
    ship()
except errors.ConditionalCheckFailedException:
    print("second ship refused: the order is no longer placed")

```

```

shipped once
second ship refused: the order is no longer placed

```

The second update finds the order already shipped. Its condition fails, so DynamoDB leaves the item as the first update wrote it, and boto3 raises `ConditionalCheckFailedException`. The listing refers to status through the `#s` placeholder because status is a reserved word in DynamoDB expressions. The same pattern with a version number checked on every write gives **optimistic locking**, in which a write succeeds only if the item is unchanged since it was read.

10.6 DynamoDB Streams

The stream keeps a record of every change to an item for 24 hours, in the order the changes were made to that item. With `NEW_AND_OLD_IMAGES`, each record carries the item

as it was before and after the change. Like a Kinesis data stream, the stream is split into shards, partitions of the stream that are read separately, and this listing reads every shard from its oldest record:

```
from collections import Counter

import boto3

dynamodb = boto3.client("dynamodb")
streams = boto3.client("dynamodbstreams")

table = dynamodb.describe_table(TableName="orders")["Table"]
arn = table["LatestStreamArn"]
described = streams.describe_stream(StreamArn=arn)
records = []
for shard in described["StreamDescription"]["Shards"]:
    iterator = streams.get_shard_iterator(
        StreamArn=arn,
        ShardId=shard["ShardId"],
        ShardIteratorType="TRIM_HORIZON",
    )["ShardIterator"]
    batch = streams.get_records(ShardIterator=iterator)
    records += batch["Records"]

print(Counter(r["eventName"] for r in records))
for record in records:
    if record["eventName"] == "MODIFY":
        change = record["dynamodb"]
        old = change["OldImage"]["status"]["S"]
        new = change["NewImage"]["status"]["S"]
        print(change["Keys"]["order_id"]["S"], old, "->", new)
```

```
Counter({'INSERT': 12, 'MODIFY': 1})
2026-10-04#0004 placed -> shipped
```

The stream holds twelve inserts from the batch write and the one update that succeeded, with the status before and after it; the refused update changed nothing and so produced no record. A consumer of the stream can copy changes to S3 for Athena, keep a search index in step, or trigger a Lambda function on each change. A production consumer stores the sequence number of the last record it processed and resumes after it with the `AFTER_SEQUENCE_NUMBER` iterator type.

10.7 Vector search

A vector index stores the embedding attribute of each item and answers SearchVectors requests with the items whose vectors are nearest to a query vector. As in chapters 4 and 8, the examples use short hand-written vectors in place of a model's embeddings.

The index can carry a **search schema**. A HASH element is the **vector index partition key**: every search names one value of it and examines only the items with that value, which keeps searches fast as the index grows. INLINE_FILTER attributes are stored next to each vector, so a search can filter on them as it runs. Both are declared in AttributeDefinitions, as keys are:

```
import boto3

dynamodb = boto3.client("dynamodb")
dynamodb.create_table(
    TableName="products",
    AttributeDefinitions=[
        {"AttributeName": "sku", "AttributeType": "S"},
        {"AttributeName": "category", "AttributeType": "S"},
        {"AttributeName": "brand", "AttributeType": "S"},
    ],
    KeySchema=[{"AttributeName": "sku", "KeyType": "HASH"}],
    BillingMode="PAY_PER_REQUEST",
    VectorIndexes=[
        {
            "IndexName": "similar",
            "VectorAttribute": {"AttributeName": "embedding"},
            "SearchSchema": [
                {
                    "AttributeName": "category",
                    "SearchSchemaElementType": "HASH",
                },
                {
                    "AttributeName": "brand",
                    "SearchSchemaElementType": "INLINE_FILTER",
                },
            ],
            "Projection": {"ProjectionType": "ALL"},
            "Dimensions": 3,
            "DistanceFunction": "COSINE",
        },
    ],
)
```

```

)
dynamodb.get_waiter("table_exists").wait(TableName="products")
table = dynamodb.describe_table(TableName="products")["Table"]
for index in table["VectorIndexes"]:
    print(index["IndexName"], index["IndexStatus"])

```

```
similar ACTIVE
```

Vector indexes need on-demand mode. The table's key is still sku, and category partitions only the vector index. Items are written as usual, with the embedding as a list of numbers:

```

import boto3

dynamodb = boto3.client("dynamodb")

def vector(*values):
    return {"L": [{"N": str(v)} for v in values]}

products = [ # sku, category, brand, title, embedding
    ("k-100", "kitchen", "acme", "Chef's knife", (0.9, 0.1, 0)),
    ("k-101", "kitchen", "acme", "Paring knife", (0.8, 0.2, 0.1)),
    ("k-102", "kitchen", "forge", "Bread knife", (0.7, 0.3, 0)),
    ("k-200", "kitchen", "forge", "Breadboard", (0.2, 0.9, 0.1)),
    ("g-300", "garden", "acme", "Pruning shears", (0.9, 0, 0.2)),
]
for sku, category, brand, title, embedding in products:
    dynamodb.put_item(
        TableName="products",
        Item={
            "sku": {"S": sku},
            "category": {"S": category},
            "brand": {"S": brand},
            "title": {"S": title},
            "embedding": vector(*embedding),
        },
    )
print(len(products), "products")

```

```
5 products
```

The next listing searches the kitchen partition for the products nearest a query vector, then searches only the kitchen products of one brand:

```
import boto3

dynamodb = boto3.client("dynamodb")
query = [{"N": "1.0"}, {"N": "0.1"}, {"N": "0.0"}]

def search(condition, values):
    found = dynamodb.search_vectors(
        TableName="products",
        IndexName="similar",
        SearchVector=query,
        TopK=3,
        SearchConditionExpression=condition,
        ExpressionAttributeValues=values,
    )
    for hit in found["SearchResults"]:
        title = hit["Item"]["title"]["S"]
        print(f"{hit['Score']:.4f} {title}")

search("category = :c", {"c": {"S": "kitchen"}})
print("--")
search(
    "category = :c AND brand = :b",
    {"c": {"S": "kitchen"}, "b": {"S": "forge"}},
)

0.0001 Chef's knife
0.0177 Paring knife
0.0462 Bread knife
--
0.0462 Bread knife
0.6888 Breadboard
```

With COSINE the score is a distance, in which 0 means the same direction and a lower score means closer, so results come nearest first. The pruning shears point almost the same way as the chef's knife, but they are in the garden partition, so the kitchen search leaves them out. The second search keeps only one brand. SearchVectors returns the vector itself only when a ProjectionExpression names it, because embeddings are large and searches are billed by the bytes they process.

DynamoDB enforces the index's rules on searches and on writes:

```
import boto3

dynamodb = boto3.client("dynamodb")
errors = dynamodb.exceptions

try: # the vector index partition key must be in every search
    dynamodb.search_vectors(
        TableName="products",
        IndexName="similar",
        SearchVector=[{"N": "1"}, {"N": "0"}, {"N": "0"}],
        TopK=3,
    )
except errors.ClientError as e:
    print(e.response["Error"]["Code"], "on search")

try: # a vector must have the index's dimensions
    dynamodb.put_item(
        TableName="products",
        Item={
            "sku": {"S": "k-999"},
            "category": {"S": "kitchen"},
            "embedding": {"L": [{"N": "1"}, {"N": "0"}]},
        },
    )
except errors.ClientError as e:
    print(e.response["Error"]["Code"], "on write")
```

```
ValidationException on search
ValidationException on write
```

A search must name the vector index partition key, and a vector must have the index's dimensions. Inline filters are optional in a search, and `SearchConditionExpression` currently supports only equality. An item written without the partition key attribute is stored in the table without an error but is left out of the index, so it never appears in a search.

DynamoDB is also used as an **online feature store**, the low-latency side of a feature store such as Feast, which Part III sets up over Redshift and DynamoDB.

10.8 Cleaning up

```
import boto3

dynamodb = boto3.client("dynamodb")
for name in ("orders", "products"):
    try:
        dynamodb.delete_table(TableName=name)
    except dynamodb.exceptions.ResourceNotFoundException:
        pass
```

10.9 Where oblako differs from AWS

- **One local instance.** DynamoDB Local has no partitions, throughput limits, throttling or global tables, so every request runs at the speed of your machine.
- **Vector search is exact.** The proxy compares the query with every indexed vector, so results on oblako are the true nearest neighbors, whereas AWS uses an approximate index. An index on oblako is searchable as soon as it exists, while AWS builds the index first and makes new writes searchable a moment later. A scan of the whole table per search suits test data but does not scale to millions of items.
- **Some write checks are partial.** Vector dimensions are checked on PutItem, BatchWriteItem and transactional puts but not on UpdateItem, and searches report no ConsumedCapacity.
- **ARNs are DynamoDB Local's,** with region ddblocal and account 000000000000; use the stream ARN the API returns and do not build one.
- **No Streams triggers.** You can read the stream, but oblako invokes no Lambda function from it.

10.10 Running it on AWS

The listings run unchanged against DynamoDB in your account. On-demand tables this small cost close to nothing, and the cleanup listing deletes them. Your role needs the dynamodb actions the listings call, dynamodb:SearchVectors on the vector index, and the

`dynamodb:GetRecords` family on the stream. A vector index is created with the table and built asynchronously, so wait until `DescribeTable` reports it `ACTIVE` without `Backfilling`, and treat a `ValidationException` on the first searches as a reason to retry, as AWS's documentation advises. Newly written items become searchable a moment after the write.

10.11 Summary

A DynamoDB table is designed from its access patterns, with partition and sort keys that answer the main question with a `Query` and a GSI for another. Condition expressions make writes check state atomically, and the stream records every change, in order for each item, for downstream consumers. Vector indexes add similarity search over embeddings stored on items, scoped by a vector index partition key and narrowed by inline filters. On oblako, DynamoDB Local holds the data and a proxy adds vector search and tags, so the same `boto3` code runs on oblako and on AWS.

11

Streaming with Kinesis and Firehose

Kinesis Data Streams, and Firehose delivery to S3 and Redshift.

11.1 Before you start

```
oblako up kinesis
oblako up firehose
oblako up trino
oblako up athena
oblako up redshift
export AWS_PROFILE=oblako
```

```
uv add boto3 redshift-connector
```

The chapter reuses the events bucket and the web database of Chapter 5, the Athena helper `athena_query.py` from the same chapter, and `warehouse.py` from Chapter 7; copy both helpers next to this chapter's listings. Two listings each wait about a minute for Firehose to deliver, and the last listing removes what the chapter creates.

11.2 Kinesis and Firehose in brief

A **Kinesis data stream** is an append-only log split into **shards**, each of which stores part of the stream. Each record carries a **partition key**, which Kinesis hashes with MD5 to a 128-bit number, and the shard whose hash key range holds the result stores the record. Records with the same partition key therefore land in one shard, in the order they were written, and each gets a **sequence number** there. A stream keeps records for 24 hours by default and for up to a year. A record stays in the stream after it is read, so any number of applications can read the same stream, each at its own position. A shard takes up to 1 MB or 1,000 records a second of writes in provisioned mode; on-demand streams add shards as traffic grows.

Amazon Data Firehose is a managed consumer that delivers data to storage. A **Firehose stream** (formerly *delivery stream*, the name the API still uses) takes records directly or from a Kinesis stream and buffers them until a size or time limit. It then writes each batch to its destination: S3, Redshift, OpenSearch, Apache Iceberg tables, Splunk, Snowflake or an HTTP endpoint, among others. On the way it can convert JSON to Parquet or ORC and compress the output. Firehose takes the place of the consumer code you would otherwise write, including its tracking of how far it has read.

What oblako runs

On oblako, Kinesis Data Streams is served by *kinesalite*, an open-source implementation of the Kinesis API, on port 4567, and Firehose by an oblako engine on port 8008. The engine buffers, names and writes S3 objects as Firehose does, and converts records to Parquet with the column types of a Glue table. It loads Redshift with the `COPY` statement Firehose issues, which oblako's Redshift (Chapter 7) runs against S3 (Figure 11.1).

11.3 Shards

Create a stream with two shards:

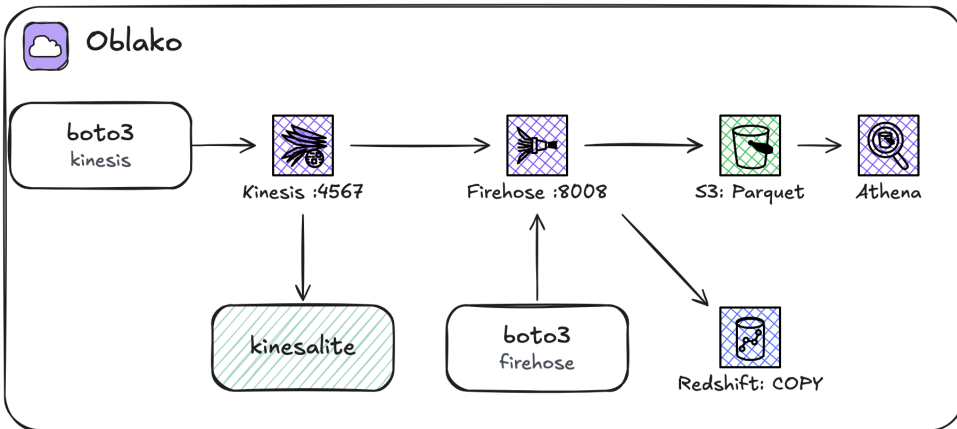


Figure 11.1: Streaming on oblako. Kinesis is kinesalite; the Firehose engine reads the stream and writes Parquet to S3 for Athena, or stages to S3 and runs COPY on Redshift.

```

import boto3

kinesis = boto3.client("kinesis")
kinesis.create_stream(StreamName="clicks", ShardCount=2)
kinesis.get_waiter("stream_exists").wait(StreamName="clicks")

stream = kinesis.describe_stream(StreamName="clicks")[
    "StreamDescription"
]
print(
    stream["StreamStatus"],
    stream["RetentionPeriodHours"],
    "hours",
)
for shard in stream["Shards"]:
    hashes = shard["HashKeyRange"]
    start = int(hashes["StartingHashKey"])
    end = int(hashes["EndingHashKey"])
    print(shard["ShardId"], f"{start:.3e}", f"{end:.3e}")

```

```

ACTIVE 24 hours
shardId-000000000000 0.000e+00 1.701e+38
shardId-000000000001 1.701e+38 3.403e+38

```

The two shards split the hash key space, 0 to $2^{128} - 1$, in half. The clicks come from a small generator, `clicks.py`, which the following listings import; its `as_records` turns each click into a Kinesis record keyed by its user:

```
import datetime
import json
import random

PAGES = ["/", "/pricing", "/docs", "/signup"]

def clicks(n, seed):
    rng = random.Random(seed)
    now = datetime.datetime.now(datetime.timezone.utc)
    for _ in range(n):
        yield {
            "user": f"user-{rng.randint(1, 6)}",
            "page": rng.choice(PAGES),
            "at": now.isoformat(timespec="seconds"),
        }

def as_records(events):
    return [
        {
            "Data": json.dumps(e).encode(),
            "PartitionKey": e["user"],
        }
        for e in events
    ]
```

11.4 Producers

`PutRecords` writes up to 500 records in one call, and the next listing uses it to write forty clicks:

```

from collections import Counter

import boto3

from clicks import as_records, clicks

kinesis = boto3.client("kinesis")
result = kinesis.put_records(
    StreamName="clicks", Records=as_records(clicks(40, seed=1))
)
print("failed:", result["FailedRecordCount"])
print(Counter(r["ShardId"] for r in result["Records"]))

```

```

failed: 0
Counter({'shardId-000000000000': 36, 'shardId-000000000001': 4})

```

Always check `FailedRecordCount`. `PutRecords` succeeds as a call even when some of its records fail, typically because a shard is over its limit, and the failed records must be sent again. The clicks split 36 to 4 between the shards because the six partition keys hash to six points, five of which fell in the first shard's half. Since the partition key decides both ordering and load, choose a key with many distinct values; with few, one shard can become *hot*, taking most of the writes while the others idle.

11.5 Consumers

A reader asks for a **shard iterator** at a position, then pages through records with `GetRecords`. `TRIM_HORIZON` starts at the oldest record still kept:

```

import json
from collections import defaultdict

import boto3

kinesis = boto3.client("kinesis")
stream = kinesis.describe_stream(StreamName="clicks")[
    "StreamDescription"
]

for shard in stream["Shards"]:
    iterator = kinesis.get_shard_iterator(
        StreamName="clicks",
        ShardId=shard["ShardId"],
        ShardIteratorType="TRIM_HORIZON",
    )["ShardIterator"]
    records = kinesis.get_records(ShardIterator=iterator)[
        "Records"
    ]
    users = defaultdict(int)
    for record in records:
        users[json.loads(record["Data"])["user"]] += 1
    print(shard["ShardId"], len(records), "records")
    for user, count in sorted(users.items()):
        print(f"  {user}: {count}")

```

```

shardId-000000000000 36 records
  user-2: 4
  user-3: 7
  user-4: 10
  user-5: 9
  user-6: 6
shardId-000000000001 4 records
  user-1: 4

```

Each user's clicks are in one shard, in the order they were written. Kinesis guarantees order within a partition key; records with different partition keys have no defined order relative to each other.

A reader that restarts must resume where it stopped, so it records the sequence number of the last record it processed, called its **checkpoint**, and resumes after that record:

```

import boto3

from clicks import as_records, clicks

kinesis = boto3.client("kinesis")
shard = kinesis.describe_stream(StreamName="clicks")
shard_id = shard["StreamDescription"]["Shards"][0]["ShardId"]

def read(**position):
    iterator = kinesis.get_shard_iterator(
        StreamName="clicks", ShardId=shard_id, **position
    )["ShardIterator"]
    return kinesis.get_records(ShardIterator=iterator)["Records"]

seen = read(ShardIteratorType="TRIM_HORIZON")
checkpoint = seen[-1][
    "SequenceNumber"
] # store this after processing
print("processed", len(seen), "up to", checkpoint[-6:])

kinesis.put_records(
    StreamName="clicks", Records=as_records(clicks(10, 2))
)
new = read(
    ShardIteratorType="AFTER_SEQUENCE_NUMBER",
    StartingSequenceNumber=checkpoint,
)
print("new since the checkpoint:", len(new))

```

```

processed 36 up to 001346
new since the checkpoint: 8

```

Eight of the ten new clicks landed in this shard, and the reader sees only those eight. In production this loop is rarely written by hand. The Kinesis Client Library balances shards across workers and keeps checkpoints in a DynamoDB table, a stream can trigger a Lambda function, and Firehose can read a stream and deliver it to S3 or Redshift.

11.6 Firehose to S3

Firehose can convert JSON records to Parquet as it writes them, which makes the data much cheaper to query. It takes the columns and their types from a table in the Glue Data Catalog, so define that table first, partitioned by day:

```
import boto3

glue = boto3.client("glue")
try:
    glue.create_database(DatabaseInput={"Name": "web"})
except glue.exceptions.AlreadyExistsException:
    pass

HIVE = "org.apache.hadoop.hive ql.io.parquet"
STORAGE = {
    "InputFormat": f"{HIVE}.MapredParquetInputFormat",
    "OutputFormat": f"{HIVE}.MapredParquetOutputFormat",
    "SerdeInfo": {
        "SerializationLibrary": f"{HIVE}.serde.ParquetHiveSerDe"
    },
}

glue.create_table(
    DatabaseName="web",
    TableInput={
        "Name": "stream_clicks",
        "TableType": "EXTERNAL_TABLE",
        "PartitionKeys": [{"Name": "dt", "Type": "string"}],
        "StorageDescriptor": {
            "Columns": [
                {"Name": "user", "Type": "string"},
                {"Name": "page", "Type": "string"},
                {"Name": "at", "Type": "timestamp"},
            ],
            "Location": "s3://events/stream-clicks/",
            **STORAGE,
        },
    },
)
print("table web.stream_clicks")
```

```
table web.stream_clicks
```

Then create the Firehose stream, reading the Kinesis stream:

```

import boto3

ROLE = "arn:aws:iam::123456789012:role/firehose-delivery"
ERRORS = "stream-errors/!{firehose:error-output-type}/"
kinesis = boto3.client("kinesis")
firehose = boto3.client("firehose")

source = kinesis.describe_stream_summary(StreamName="clicks")
firehose.create_delivery_stream(
    DeliveryStreamName="clicks-to-s3",
    DeliveryStreamType="KinesisStreamAsSource",
    KinesisStreamSourceConfiguration={
        "KinesisStreamARN": source["StreamDescriptionSummary"][
            "StreamARN"
        ],
        "RoleARN": ROLE,
    },
    ExtendedS3DestinationConfiguration={
        "RoleARN": ROLE,
        "BucketARN": "arn:aws:s3:::events",
        "Prefix": "stream-clicks/dt=!{timestamp:yyyy-MM-dd}/",
        "ErrorOutputPrefix": ERRORS,
        "BufferingHints": {
            "IntervalInSeconds": 60,
            "SizeInMBs": 64,
        },
        "DataFormatConversionConfiguration": {
            "SchemaConfiguration": {
                "DatabaseName": "web",
                "TableName": "stream_clicks",
                "RoleARN": ROLE,
            },
            "InputFormatConfiguration": {
                "Deserializer": {"OpenXJsonSerDe": {}}
            },
            "OutputFormatConfiguration": {
                "Serializer": {"ParquetSerDe": {}}
            },
        },
    },
)
description = firehose.describe_delivery_stream(
    DeliveryStreamName="clicks-to-s3"
)["DeliveryStreamDescription"]
print(

```

```

        description["DeliveryStreamStatus"],
        description["DeliveryStreamType"],
    )

```

ACTIVE KinesisStreamAsSource

The Prefix sets where objects go. Firehose evaluates `!{timestamp:yyyy-MM-dd}` with the arrival time of the oldest record in each object, so objects land in Hive-style `dt=` folders that match the table's partition. A prefix with expressions requires an `ErrorOutputPrefix`, under which Firehose writes the records that fail to convert, with the reason. Format conversion requires a buffer size of at least 64 MB, so here the 60-second interval decides when a batch is written. A Firehose stream reads its source from the `LATEST` position and so delivers only clicks written after it was created. The next listing sends new clicks and waits:

```

import time

import boto3

from clicks import as_records, clicks

kinesis = boto3.client("kinesis")
s3 = boto3.client("s3")

kinesis.put_records(
    StreamName="clicks", Records=as_records(clicks(200, seed=3))
)
started = time.monotonic()
objects = []
while not objects:
    time.sleep(5)
    listing = s3.list_objects_v2(
        Bucket="events", Prefix="stream-clicks/"
    )
    objects = listing.get("Contents", [])
waited = time.monotonic() - started
for obj in objects:
    folder, name = obj["Key"].rsplit("/", 1)
    print(folder + "/" + name)
    print(" " + name[:34] + "..." + name[-8:])
print(f"delivered after about {waited:.0f} s")

```

```
stream-clicks/dt=2026-10-02/
clicks-to-s3-1-2026-10-02-17-07-46....parquet
delivered after about 65 s
```

An object's name is its evaluated prefix, then the stream's name and version, the date and time, and a random identifier, with `.parquet` added by the conversion. Athena reads a partition only once it is in the catalog. On AWS, a Glue crawler or Athena's `MSCK REPAIR TABLE` can find new partitions, but with a known layout it is simpler to register the partition yourself:

```
import boto3

glue = boto3.client("glue")
s3 = boto3.client("s3")

table = glue.get_table(DatabaseName="web", Name="stream_clicks")["Table"]
listing = s3.list_objects_v2(
    Bucket="events", Prefix="stream-clicks/", Delimiter="/"
)
for common in listing.get("CommonPrefixes", []):
    folder = common["Prefix"] # stream-clicks/dt=2026-10-02/
    day = folder.rstrip("/").split("dt=")[1]
    storage = dict(table["StorageDescriptor"])
    storage["Location"] = f"s3://events/{folder}"
    try:
        glue.create_partition(
            DatabaseName="web",
            TableName="stream_clicks",
            PartitionInput={
                "Values": [day],
                "StorageDescriptor": storage,
            },
        )
        print("added partition dt =", day)
    except glue.exceptions.AlreadyExistsException:
        print("partition dt =", day, "exists")
```

```
added partition dt = 2026-10-02
```

Query the clicks with Athena:

```

from athena_query import run_query

rows = run_query("""
    SELECT page, count(*) AS clicks,
           count(DISTINCT "user") AS users
    FROM stream_clicks
    GROUP BY page
    ORDER BY clicks DESC""")
for row in rows:
    print(row)

```

```

['page', 'clicks', 'users']
['/docs', '55', '6']
['/signup', '54', '6']
['/', '53', '6']
['/pricing', '38', '6']

```

All 200 clicks arrived. The usual pattern is one partition per day, registered once a day, and Athena's *partition projection*, which computes partitions from the table's properties, removes the need to register them.

11.7 Firehose to Redshift

For Redshift, Firehose stages each batch in S3 and then runs a COPY on the cluster from the staged files, with the credentials of its role and the CopyOptions you give it. Most failures come from these options; for example, without a format option, COPY expects pipe-delimited text, and JSON records fail to load. This stream sends JSON, gzipped in staging:

```

import boto3

from warehouse import query

ROLE = "arn:aws:iam::123456789012:role/firehose-delivery"
query("DROP TABLE IF EXISTS page_views")
query("""
    CREATE TABLE page_views (
        "user" varchar(32),
        page   varchar(64),
        at     timestamp
    )""")

firehose = boto3.client("firehose")
firehose.create_delivery_stream(
    DeliveryStreamName="views-to-redshift",
    DeliveryStreamType="DirectPut",
    RedshiftDestinationConfiguration={
        "RoleARN": ROLE,
        "ClusterJDBCURL": "jdbc:redshift://localhost:5439/oblako",
        "Username": "oblako",
        "Password": "oblako",
        "CopyCommand": {
            "DataTableName": "page_views",
            "CopyOptions": "JSON 'auto' GZIP TIMEFORMAT 'auto'",
        },
        "S3Configuration": {
            "RoleARN": ROLE,
            "BucketARN": "arn:aws:s3:::events",
            "Prefix": "staging/page-views/",
            "CompressionFormat": "GZIP",
            "BufferingHints": {
                "IntervalInSeconds": 60,
                "SizeInMBs": 5,
            },
        },
    },
)
print("views-to-redshift created")

```

```
views-to-redshift created
```

JSON 'auto' matches the records' keys to column names, GZIP matches the staging compression, and TIMEFORMAT 'auto' parses the ISO timestamps. This stream takes records directly (DirectPut) from the application:

```
import json
import time

import boto3

from clicks import clicks
from warehouse import query

firehose = boto3.client("firehose")
events = list(clicks(50, seed=4))
firehose.put_record_batch(
    DeliveryStreamName="views-to-redshift",
    Records=[{"Data": json.dumps(e).encode()} for e in events],
)

count = 0
while count < len(events):
    time.sleep(5)
    count = query("SELECT count(*) FROM page_views")[0][0]
    print("rows in page_views:", count)
    print(
        query("""
        SELECT "user", count(*) FROM page_views
        GROUP BY 1 ORDER BY 2 DESC LIMIT 3""")
    )
```

```
rows in page_views: 50
(['user-1', 14], ['user-3', 9], ['user-2', 8])
```

11.8 Cleaning up

```

import boto3

from warehouse import query

firehose = boto3.client("firehose")
for name in ("clicks-to-s3", "views-to-redshift"):
    try:
        firehose.delete_delivery_stream(DeliveryStreamName=name)
    except firehose.exceptions.ResourceNotFoundException:
        pass

kinesis = boto3.client("kinesis")
try:
    kinesis.delete_stream(StreamName="clicks")
except kinesis.exceptions.ResourceNotFoundException:
    pass

glue = boto3.client("glue")
try:
    glue.delete_table(DatabaseName="web", Name="stream_clicks")
except glue.exceptions.EntityNotFoundException:
    pass

s3 = boto3.client("s3")
for prefix in ("stream-clicks/", "stream-errors/", "staging/"):
    listing = s3.list_objects_v2(Bucket="events", Prefix=prefix)
    for obj in listing.get("Contents", []):
        s3.delete_object(Bucket="events", Key=obj["Key"])
query("DROP TABLE IF EXISTS page_views")

```

11.9 Where oblako differs from AWS

- **No limits or scaling.** kinesislite enforces no shard throughput limits, so nothing is throttled, and it lacks on-demand mode, resharding and enhanced fan-out. Its ARNs and errors name account 000000000000.
- **Firehose destinations are S3 and Redshift.** The engine lacks Lambda transformation, dynamic partitioning, ORC output, and ZIP and Snappy compression. It also lacks UpdateDestination, so the stream version in object names is always 1.
- **Buffering is as configured.** oblako flushes on whatever interval you set, which shortens tests, whereas AWS has minimum intervals for some destinations.

Records still in the buffer when the Firehose engine stops are lost, but the Firehose streams themselves are kept.

- **Failures are not retried.** The engine logs a failed Redshift COPY with oblako's own error text and leaves the staged object in S3. AWS instead retries for a configurable time, records the reason in `STL_LOAD_ERRORS`, and writes a manifest of the failed files to S3.
- **Glue partitions are registered by hand** here, as on AWS without a crawler, because oblako has no crawler and no `MSCK REPAIR TABLE`.

11.10 Running it on AWS

The listings run unchanged in your account. Firehose needs an IAM role it can assume, so in place of the placeholder `ROLE`, create one that `firehose.amazonaws.com` trusts, with read access to the Kinesis stream, write access to the bucket, and `glue:GetTable` for format conversion. For Redshift, Firehose connects from AWS's network, so the cluster must accept connections from Firehose's address range for your Region and the role must be able to read the staging bucket, since the COPY uses it. Kinesis streams bill per shard-hour and Firehose per gigabyte, so run the cleanup listing when you finish.

11.11 Summary

A Kinesis data stream keeps records in shards for a day or more, ordered per partition key, and any number of readers can read it, each tracking its own sequence number. Partition keys with many values spread the load across shards. Firehose is a managed reader that delivers a stream to storage: it buffers records, names objects with time-based prefixes, converts JSON to Parquet with a Glue table's schema, and loads Redshift with a COPY whose options you supply. On oblako, kinesalite holds the stream and oblako's Firehose engine delivers it, so you can query a stream with Athena or Redshift on your laptop.

PART III

Data engineering tools



12

Spark with Iceberg

Building and maintaining an Iceberg table in S3 Tables with Apache Spark.

12.1 Before you start

Spark runs on the Java virtual machine, so it needs a Java runtime. Spark 4 requires Java 17 or later; on a Mac, Homebrew installs it:

```
brew install openjdk@17
export JAVA_HOME=/opt/homebrew/opt/openjdk@17
```

The listings use PySpark 4.0.1, the Spark release that Iceberg 1.10 is built for, together with boto3 and PyIceberg:

```
uv add "pyspark==4.0.1" boto3 "pyiceberg[pyarrow]"
oblako up s3tables
oblako up trino
oblako up athena
export AWS_PROFILE=oblako
```

The first Spark session downloads two Iceberg libraries from Maven Central, about 110 MB, and keeps them in Ivy's cache under your home directory for later runs. The chap-

ter also reuses `catalog.py` from Chapter 4 and `athena_query.py` from Chapter 5; copy both next to this chapter's listings and change the prefix in `catalog.py` to `lakehouse`.

12.2 Spark and Iceberg

Apache Spark is a distributed engine for batch and streaming jobs, and it is the most common engine for writing Iceberg tables. Iceberg ships a Spark runtime library that adds Iceberg catalogs, table DDL, `MERGE INTO` and `DELETE`, and stored procedures for maintenance. Chapter 4 created an Iceberg table with PyIceberg, which suits small writes from Python; Spark is the tool for rewriting large tables, merging changes and maintaining files.

Spark reaches S3 Tables through its Iceberg REST endpoint, the same endpoint PyIceberg used. The catalog is configured with the table bucket's ARN as the warehouse, and every request to it is signed with SigV4 for the `s3tables` service. Data files are read and written through Iceberg's own S3 client, `S3FileIO`.

What oblako runs

oblako serves the S3 Tables Iceberg REST endpoint at `http://localhost:8013/iceberg` and stores the data files in its S3 on port 9000. Spark itself runs in your Python process, in local mode, with the Iceberg runtime it downloads; oblako only provides the endpoints (Figure 12.1).

12.3 A table bucket and a session

Create a table bucket for the chapter:

```
import boto3

s3tables = boto3.client("s3tables")
bucket = s3tables.create_table_bucket(name="lakehouse")
print(bucket["arn"])
```

```
arn:aws:s3tables:us-east-1:123456789012:bucket/lakehouse
```

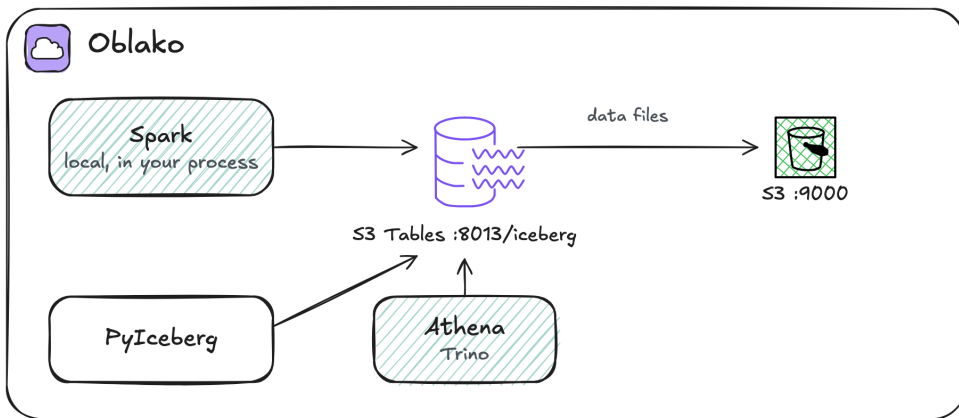


Figure 12.1: Spark on oblako. Spark runs in your process; it, PyIceberg and Athena find the table through the S3 Tables Iceberg REST endpoint, and the table's data files are in oblako's S3.

Every listing that uses Spark starts its session from one module, `lake.py`. It names the catalog `lake`, so tables are addressed as `lake.<namespace>.<table>`:

```

import boto3
from pyspark.sql import SparkSession

ICEBERG = "1.10.0"
PACKAGES = [
    f"org.apache.iceberg:{name}:{ICEBERG}"
    for name in (
        "iceberg-spark-runtime-4.0_2.13",
        "iceberg-aws-bundle",
    )
]
S3_FILE_IO = "org.apache.iceberg.aws.s3.S3FileIO"

def bucket_arn(name="lakehouse"):
    s3tables = boto3.client("s3tables")
    found = s3tables.list_table_buckets(prefix=name)
    return found["tableBuckets"][0]["arn"]

def session():
    # the JVM does not read the profile's endpoints, so ask boto3
    aws = boto3.Session()
    region = aws.region_name
    tables = aws.client("s3tables").meta.endpoint_url
    s3 = aws.client("s3")
    lake = "spark.sql.catalog.lake"
    builder = (
        SparkSession.builder.appName("lakehouse")
        .config("spark.jars.packages", ",".join(PACKAGES))
        .config("spark.sql.session.timeZone", "UTC")
        .config("spark.ui.showConsoleProgress", "false")
        .config(
            "spark.sql.extensions",
            "org.apache.iceberg.spark.extensions"
            ".IcebergSparkSessionExtensions",
        )
        .config(lake, "org.apache.iceberg.spark.SparkCatalog")
        .config(f"{lake}.type", "rest")
        .config(f"{lake}.uri", f"{tables}/iceberg")
        .config(f"{lake}.warehouse", bucket_arn())
        .config(f"{lake}.rest.sigv4-enabled", "true")
        .config(f"{lake}.rest.signing-name", "s3tables")
        .config(f"{lake}.rest.signing-region", region)
        .config(f"{lake}.io-impl", S3_FILE_IO)
    )

```

```

)
endpoint, s3_config = s3.meta.endpoint_url, s3.meta.config.s3
builder = builder.config(f"{lake}.s3.endpoint", endpoint)
if s3_config.get("addressing_style") == "path":
    path_style = f"{lake}.s3.path-style-access"
    builder = builder.config(path_style, "true")
spark = builder.getOrCreate()
spark.sparkContext.setLogLevel("ERROR")
return spark

```

Spark runs in the JVM, which does not read the endpoints in an AWS profile, so the module asks boto3 for them: the S3 Tables endpoint, to which the catalog adds /iceberg, and the S3 endpoint for S3FileIO. Under the oblako profile these are oblako's, and the profile's path-style setting switches on path-style addressing, which oblako's S3 requires; under an AWS profile they are the regional AWS endpoints. The session runs in UTC, so that timestamps display in the same time zone that Iceberg uses to compute day partitions.

12.4 Creating a partitioned table

The table holds orders, partitioned by the day each order was placed. `days(placed_at)` is a **partition transform**: Iceberg derives the day from the timestamp, so queries filter on `placed_at` itself and Iceberg skips the days outside the filter, without a separate partition column. The listing creates the table and fills it with 7,000 orders, 1,000 a day for a week:

```

from lake import session

spark = session()
spark.sql("CREATE NAMESPACE IF NOT EXISTS lake.sales")
spark.sql("""
    CREATE TABLE lake.sales.orders (
        order_id  bigint,
        customer  string,
        total     decimal(10, 2),
        placed_at timestamp)
    USING iceberg
    PARTITIONED BY (days(placed_at))""")

# 7,000 orders, 1,000 a day from 1 October 2026
spark.sql("""
    INSERT INTO lake.sales.orders
    SELECT id,
        concat('customer-', id % 50),
        CAST((id % 97) * 1.25 AS decimal(10, 2)),
        timestamp'2026-10-01 00:00:00'
            + make_interval(0, 0, 0, CAST(id DIV 1000 AS int),
                           0, 0, CAST(id % 1000 * 60 AS int))
    FROM range(7000)""")
spark.sql("""
    SELECT date(placed_at) AS day, count(*) AS orders
    FROM lake.sales.orders GROUP BY 1 ORDER BY 1""").show()

```

```

+-----+-----+
|      day|orders|
+-----+-----+
|2026-10-01| 1000|
|2026-10-02| 1000|
|2026-10-03| 1000|
|2026-10-04| 1000|
|2026-10-05| 1000|
|2026-10-06| 1000|
|2026-10-07| 1000|
+-----+-----+

```

Iceberg records the partitions and files in metadata tables, which Spark queries like ordinary tables by appending their names to the table's:

```

from lake import session

spark = session()
spark.sql("""
    SELECT partition.placed_at_day AS day,
           record_count, file_count
    FROM lake.sales.orders.partitions
    ORDER BY day""").show()

```

```

+-----+-----+-----+
|      day|record_count|file_count|
+-----+-----+-----+
|2026-10-01|        1000|         1|
|2026-10-02|        1000|         1|
|2026-10-03|        1000|         1|
|2026-10-04|        1000|         1|
|2026-10-05|        1000|         1|
|2026-10-06|        1000|         1|
|2026-10-07|        1000|         1|
+-----+-----+-----+

```

Each day is one partition with one data file. Iceberg's Spark writer distributes rows by partition before writing a partitioned table, so each day's rows reach a single task.

12.5 Upserts with MERGE INTO

Late and corrected data is the usual reason to change a table in place. `MERGE INTO` matches incoming rows against the table by a key and, in one commit, updates the rows that match and inserts the rest. Here 100 orders get corrected totals and 50 late orders arrive:

```

from lake import session

spark = session()
# corrected totals for 100 existing orders, and 50 late orders
spark.sql("""
    SELECT id AS order_id,
           concat('customer-', id % 50) AS customer,
           CAST(id % 97 * 1.25 + 5 AS decimal(10, 2)) AS total,
           timestamp'2026-10-07 12:00:00' AS placed_at
    FROM range(6900, 7050)""").createOrReplaceTempView("updates")
spark.sql("""
    MERGE INTO lake.sales.orders AS o
    USING updates AS u
    ON o.order_id = u.order_id
    WHEN MATCHED THEN UPDATE SET o.total = u.total
    WHEN NOT MATCHED THEN INSERT *""")
spark.sql("""
    SELECT count(*) AS orders, sum(total) AS revenue
    FROM lake.sales.orders""").show()

```

```

+-----+-----+
|orders| revenue|
+-----+-----+
|  7050|422471.25|
+-----+-----+

```

The table has 7,050 orders: the 50 new ones were inserted and the 100 corrected ones replaced in place.

12.6 Deleting by partition

A DELETE whose filter covers whole partitions removes those partitions' files from the table without reading or rewriting any data. This one drops the first day:

```

from lake import session

spark = session()
spark.sql("""
    DELETE FROM lake.sales.orders
    WHERE placed_at < timestamp'2026-10-02 00:00:00'""")
spark.sql("""
    SELECT operation,
           summary['deleted-data-files'] AS files_removed,
           summary['added-data-files'] AS files_added
    FROM lake.sales.orders.snapshots
    ORDER BY committed_at""").show()

```

```

+-----+-----+-----+
|operation|files_removed|files_added|
+-----+-----+-----+
|  append|          NULL|          7|
|overwrite|           1|          1|
|  delete|           1|         NULL|
+-----+-----+-----+

```

The snapshots metadata table lists every commit so far. The append wrote the seven daily files. The overwrite is the `MERGE INTO`: by default Iceberg's Spark writes are **copy-on-write**, so the merge replaced the one file holding the changed rows with a rewritten copy. The delete removed the first day's file and wrote nothing.

12.7 Time travel

Each snapshot is a complete version of the table, and a query can read any snapshot the table still keeps:

```

from lake import session

spark = session()
first = spark.sql("""
    SELECT snapshot_id FROM lake.sales.orders.snapshots
    ORDER BY committed_at LIMIT 1""").first()[0]
count = "SELECT count(*) FROM lake.sales.orders"
print("now", spark.sql(count).first()[0])
then = spark.sql(f"{count} VERSION AS OF {first}").first()[0]
print("first snapshot", then)

```

```

now 6050
first snapshot 7000

```

The first snapshot still has the 7,000 orders it was created with, before the merge added 50 and the delete removed 1,000. `VERSION AS OF` selects a snapshot by time in the same way.

12.8 Changing the schema

Iceberg tracks columns by ID, so adding a column changes only the metadata, and existing files read the new column as null:

```

from lake import session

spark = session()
spark.sql(
    "ALTER TABLE lake.sales.orders ADD COLUMN channel string"
)
spark.sql("""
    INSERT INTO lake.sales.orders
    VALUES (9001, 'customer-1', 20.00,
            timestamp'2026-10-07 18:00:00', 'web')""")
spark.sql("""
    SELECT channel, count(*) AS orders
    FROM lake.sales.orders GROUP BY channel""").show()

```

```
+-----+-----+
|channel|orders|
+-----+-----+
|  NULL|  6050|
|   web|    1|
+-----+-----+
```

12.9 Maintenance

Every commit adds files and a snapshot, and nothing is removed until it is maintained. Small files slow down reads, and old snapshots keep data files that the current table no longer uses. Iceberg's Spark procedures handle both: `rewrite_data_files` compacts small files into larger ones, and `expire_snapshots` removes old snapshots and deletes the files only they referenced:

```
from lake import session

spark = session()

def files_and_snapshots():
    counts = [
        spark.sql(f"SELECT count(*) FROM lake.sales.orders.{t}")
        for t in ("files", "snapshots")
    ]
    return tuple(c.first()[0] for c in counts)

print("before (files, snapshots):", files_and_snapshots())
compacted = spark.sql("""
    CALL lake.system.rewrite_data_files(
        table => 'sales.orders',
        options => map('min-input-files', '2'))""").first()
rewritten = compacted.rewritten_data_files_count
added = compacted.added_data_files_count
print("rewrote", rewritten, "files into", added)
expired = spark.sql("""
    CALL lake.system.expire_snapshots(
        table => 'sales.orders',
        older_than => current_timestamp(),
        retain_last => 1)""").first()
```

```
deleted = expired.deleted_data_files_count
print("deleted", deleted, "unreferenced files")
print("after (files, snapshots):", files_and_snapshots())
```

```
before (files, snapshots): (7, 4)
rewrote 2 files into 1
deleted 4 unreferenced files
after (files, snapshots): (6, 1)
```

The last day's partition held two files, the merge's rewritten file and the one-row insert from the schema listing, and compaction rewrote them into one. Expiring every snapshot except the latest deleted four data files that no snapshot still used: the first day's file, the day-seven file replaced by the merge, and the two files just compacted. Time travel now reaches only the current snapshot.

On AWS, S3 Tables runs these jobs itself: compaction, snapshot expiration and unreferenced file removal are maintenance settings on the table bucket and on each table. Table buckets also refuse direct deletion of objects, so `expire_snapshots` as written here is likely to fail against a real table bucket; this is untested. Run the procedures yourself for Iceberg tables in general-purpose S3 buckets, and on oblako, which runs no maintenance.

12.10 Reading from other engines

The table is plain Iceberg, so other engines read it through the same catalog. PyIceberg, configured as in Chapter 4, reads the new column with a filter:

```
from catalog import open_catalog

table = open_catalog().load_table("sales.orders")
orders = table.scan(row_filter="channel == 'web']").to_arrow()
print(orders.num_rows, "web orders")
print(orders.select(["order_id", "total"]).to_pylist())
```

```
1 web orders
[{'order_id': 9001, 'total': Decimal('20.00')}]
```

Athena reads it through the `s3tablescatalog` catalog of Chapter 5:

```
from athena_query import run_query

sql = (
    "SELECT date(placed_at) AS day, count(*) AS orders"
    ' FROM "s3tablescatalog/lakehouse"."sales"."orders"'
    " GROUP BY 1 ORDER BY 1"
)
for row in run_query(sql):
    print(row)
```

```
['day', 'orders']
['2026-10-02', '1000']
['2026-10-03', '1000']
['2026-10-04', '1000']
['2026-10-05', '1000']
['2026-10-06', '1000']
['2026-10-07', '1051']
```

The first day is gone, and the last day has its 1,000 orders plus the 50 late ones and the order with a channel.

12.11 Cleaning up

```
import boto3
from lake import bucket_arn

s3tables = boto3.client("s3tables")
arn = bucket_arn()
for table in s3tables.list_tables(tableBucketARN=arn)["tables"]:
    s3tables.delete_table(
        tableBucketARN=arn,
        namespace=table["namespace"][0],
        name=table["name"],
    )
spaces = s3tables.list_namespaces(tableBucketARN=arn)[
    "namespaces"
]
for space in spaces:
    s3tables.delete_namespace(
        tableBucketARN=arn, namespace=space["namespace"][0]
    )
s3tables.delete_table_bucket(tableBucketARN=arn)
```

12.12 Where oblako differs from AWS

- **No managed maintenance.** oblako's table buckets run no compaction, snapshot expiration or unreferenced file removal, and they allow the deletes that `expire_snapshots` makes.
- **Spark runs locally.** On AWS the same session code runs in Glue, EMR or EMR Serverless, which manage the cluster and supply the Iceberg libraries; this chapter runs Spark in local mode in your process.
- **One catalog per table bucket,** as on AWS, but oblako's REST endpoint implements the operations Spark, PyIceberg and Athena use in this book, which is a subset of the Iceberg REST specification.

12.13 Running it on AWS

With an AWS profile selected, `lake.py` connects to the regional S3 Tables endpoint and S3, and the listings run against your account; your credentials need the `s3tables` actions on the table bucket. Leave out the maintenance listing, or point it at an Iceberg table in a general-purpose bucket, and set the table bucket's maintenance configuration instead. For production jobs, the same Spark SQL runs in AWS Glue 5 or Amazon EMR, which can also access S3 Tables.

12.14 Summary

Spark writes Iceberg tables to S3 Tables through the Iceberg REST endpoint, with SigV4 signing and the table bucket as the warehouse. Partition transforms such as `days()` partition a table by a derived value, `MERGE INTO` applies late and corrected data in one commit, and a `DELETE` over whole partitions drops files without rewriting them. Every commit is a snapshot that queries can read until it is expired. Compaction and snapshot expiration keep a table fast and its storage in check; S3 Tables runs them for you on AWS, and on oblako you run them with Iceberg's Spark procedures. PyIceberg and Athena read the same table without copying it.

13

A multi-node Redshift

Distribution styles, skew and co-located joins on a Redshift cluster with several compute nodes.

13.1 Before you start

```
oblako up redshift
oblako up redshift-data
export AWS_PROFILE=oblako
```

```
uv add boto3 redshift-connector pyarrow
```

`oblako trust` from Chapter 2 must have run in this environment, because the listings connect with certificate verification. The first cluster you create pulls oblako's cluster image, or builds it from oblako's sources when it cannot be pulled, which takes a few minutes; later clusters start in seconds.

13.2 Nodes, slices and distribution

A multi-node Redshift cluster has one **leader node** and several **compute nodes**. Clients connect to the leader, which plans each query, sends the work to the compute nodes and combines their results. The compute nodes store the tables, and each is divided into **slices**, each with its own share of the node's memory and disk. A query runs on all slices at once, so it is as fast as the slice with the most work.

Where each row lives is decided by the table's **distribution style**. **KEY** distribution hashes a chosen column, the **distribution key**, so rows with the same key value land on the same slice. **ALL** keeps a full copy of the table on every node, which suits small tables that many queries join to. **EVEN** spreads rows round-robin, and **AUTO**, the default, lets Redshift choose and change the style as the table grows. A **sort key** orders the rows within each slice, so that range filters on it can skip blocks.

Two things follow from this. A table distributed on a column with few values, or with a few very common values, puts more rows on some slices than others, and that **skew** makes every query on the table wait for its busiest slice. And when two tables are distributed on the column they are joined on, matching rows are already on the same slice, so the join runs without moving rows between nodes; such a join is **co-located**.

What oblako runs

Chapter 7 ran one Redshift engine on port 5439. A cluster created through the Redshift API with `ClusterType='multi-node'` runs as several containers instead: oblako's Redshift engine with Citus, an open-source PostgreSQL extension that shards tables across servers. The leader is the Citus coordinator and holds no table data, and each compute node is a Citus worker (Figure 13.1). The Redshift API on port 8015 is a proxy in front of moto that starts, reports and removes these containers; the cluster's endpoint is its own name and port. Citus is licensed under the AGPL-3.0, which matters if you distribute or host a service built on the cluster image.

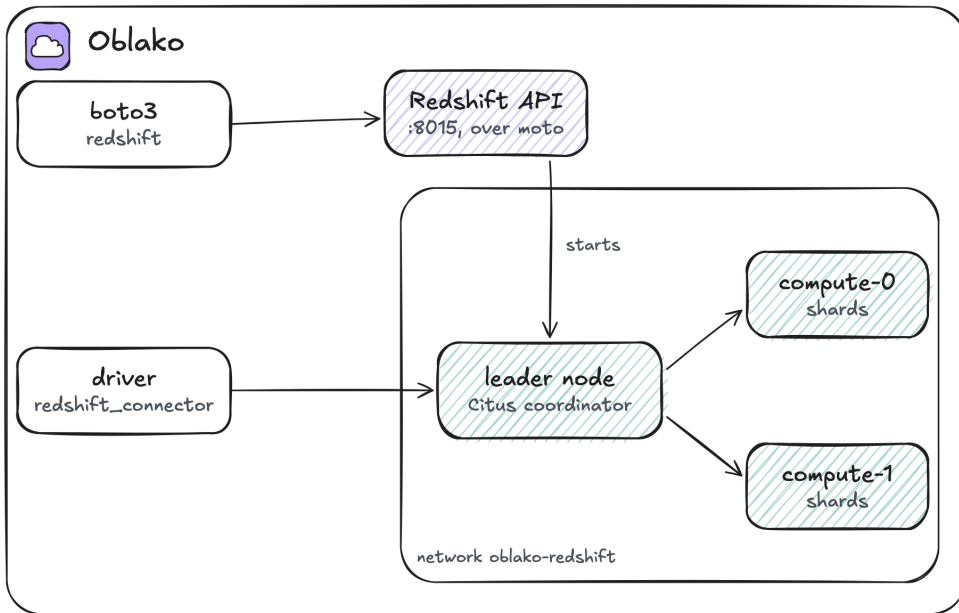


Figure 13.1: A two-node Redshift cluster on oblako. The Redshift API starts a leader and two compute nodes on their own network; drivers connect to the leader's endpoint, and the leader runs each query on the compute nodes that hold the table's shards.

13.3 Creating a cluster

```
import boto3

redshift = boto3.client("redshift")
redshift.create_cluster(
    ClusterIdentifier="analytics",
    ClusterType="multi-node",
    NodeType="ra3.large",
    NumberOfNodes=2,
    MasterUsername="admin",
    MasterUserPassword="Analytics-local-1",
    DBName="dev",
)
waiter = redshift.get_waiter("cluster_available")
waiter.wait(ClusterIdentifier="analytics")

found = redshift.describe_clusters(ClusterIdentifier="analytics")
cluster = found["Clusters"][0]
print(cluster["ClusterStatus"], cluster["NumberOfNodes"], "nodes")
print(cluster["Endpoint"]["Address"])
for node in cluster["ClusterNodes"]:
    print(node["NodeRole"], node["PrivateIPAddress"])
```

```
available 2 nodes
analytics.us-east-1.redshift.localhost
LEADER 172.28.0.4
COMPUTE-0 172.28.0.2
COMPUTE-1 172.28.0.3
```

`NumberOfNodes` counts compute nodes; the leader is extra. `ClusterNodes` lists every node with its role, and the endpoint is the leader's. On AWS the cluster takes several minutes to become available, and the waiter covers both cases.

The listings connect through one helper, which looks the endpoint up by cluster identifier:

```

import boto3
import redshift_connector

redshift = boto3.client("redshift")

def connect(cluster_id="analytics"):
    found = redshift.describe_clusters(
        ClusterIdentifier=cluster_id
    )
    endpoint = found["Clusters"][0]["Endpoint"]
    conn = redshift_connector.connect(
        host=endpoint["Address"],
        port=endpoint["Port"],
        database="dev",
        user="admin",
        password="Analytics-local-1",
        ssl=True,
        sslmode="verify-ca",
    )
    conn.autocommit = True
    return conn

def query(sql):
    with connect().cursor() as cur:
        cur.execute(sql)
        return cur.fetchall() if cur.description else []

```

13.4 Loading the tables

The data is a shop's orders, its customers and the regions they belong to, written as Parquet to S3 for COPY:

```

import datetime
import io
import random

import boto3
import pyarrow as pa
import pyarrow.parquet as pq

s3 = boto3.client("s3")
s3.create_bucket(Bucket="warehouse-staging")
rng = random.Random(7)
N = 20000 # orders
day0 = datetime.date(2026, 9, 1)
days = [day0 + datetime.timedelta(d) for d in range(30)]
names = "North South East West Coast Hills Lakes Plains".split()

def amount():
    return round(rng.uniform(5, 250), 2)

tables = {
    "customers": {
        "customer_id": list(range(1, 2001)),
        "region": [f"r{i % 8}" for i in range(1, 2001)],
    },
    "orders": {
        "order_id": list(range(1, N + 1)),
        "customer_id": [rng.randint(1, 2000) for _ in range(N)],
        "order_date": [days[i % 30] for i in range(N)],
        "total": [amount() for _ in range(N)],
    },
    "regions": {
        "region": [f"r{i}" for i in range(8)],
        "name": names,
    },
}

for name, columns in tables.items():
    table = pa.table(columns)
    buffer = io.BytesIO()
    pq.write_table(table, buffer)
    key = f"{name}/part-0.parquet"
    body = buffer.getvalue()
    s3.put_object(Bucket="warehouse-staging", Key=key, Body=body)
    print(name, table.num_rows, "rows")

```

```
customers 2000 rows
orders 20000 rows
regions 8 rows
```

The orders and customers are distributed on `customer_id`, the column they are joined on, and the eight regions are copied to every node. Each table is then loaded with `COPY`, which reads the Parquet files on the compute nodes' behalf:

```
from cluster import query

ROLE = "arn:aws:iam::123456789012:role/redshift-copy"
query("""
    CREATE TABLE customers (
        customer_id integer,
        region        varchar(4))
    DISTKEY(customer_id) SORTKEY(customer_id)""")
query("""
    CREATE TABLE orders (
        order_id      integer,
        customer_id integer,
        order_date    date,
        total          decimal(10, 2))
    DISTKEY(customer_id) SORTKEY(order_date)""")
query("""
    CREATE TABLE regions (
        region varchar(4),
        name   varchar(16))
    DISTSTYLE ALL""")
for table in ("customers", "orders", "regions"):
    query(f"""
        COPY {table} FROM 's3://warehouse-staging/{table}/'
        IAM_ROLE '{ROLE}' FORMAT AS PARQUET""")
print(query("SELECT count(*) FROM orders"))
```

```
([20000],)
```

13.5 Checking the distribution

`SVV_TABLE_INFO` reports each table's distribution style, its first sort key column, its row count and its skew:

```

from cluster import query

rows = query("""
    SELECT "table", diststyle, sortkey1, tbl_rows, skew_rows
    FROM svv_table_info
    WHERE "table" IN ('customers', 'orders', 'regions')
    ORDER BY "table" """)
for row in rows:
    print(*row)

```

```

customers KEY(customer_id) customer_id 2000 1.03
orders KEY(customer_id) order_date 20000 1.04
regions ALL None 8 None

```

skew_rows is the number of rows on the fullest slice divided by the number on the emptiest, so 1.00 is a perfectly even spread. With 2,000 distinct customer IDs hashed over the nodes, the orders are within 4% of even. The regions table has no skew value because every node holds all of it.

13.6 Choosing the distribution key

A distribution key with few distinct values spreads rows less evenly. This listing copies the orders into a table distributed on the order date, which has 30 values:

```

from cluster import query

query("""
    CREATE TABLE orders_by_day DISTKEY(order_date) AS
    SELECT * FROM orders""")
rows = query("""
    SELECT "table", diststyle, skew_rows
    FROM svv_table_info
    WHERE "table" IN ('orders', 'orders_by_day')
    ORDER BY "table" """)
for row in rows:
    print(*row)

```

```

orders KEY(customer_id) 1.04
orders_by_day KEY(order_date) 1.31

```

The fullest node now holds 1.31 times the rows of the emptiest, against 1.04 for the customer key. With a key such as a country or a status code, a few values can hold most of the rows, and the skew is far larger. A good distribution key has many distinct values, spreads rows evenly, and is the column the largest tables are joined on. `CREATE TABLE ... AS SELECT` with a new key, as here, is also how you change the distribution of a table that already holds data.

13.7 Co-located joins

The revenue by region joins all three tables:

```
from cluster import query

rows = query("""
    SELECT r.name, count(*) AS orders, sum(o.total) AS revenue
    FROM orders o
    JOIN customers c ON c.customer_id = o.customer_id
    JOIN regions r ON r.region = c.region
    GROUP BY r.name
    ORDER BY revenue DESC
    LIMIT 3""")
for row in rows:
    print(*row)
```

```
East 2591 331616.85
Coast 2567 329946.10
South 2508 327164.23
```

Each order and its customer share a `customer_id`, so they are on the same slice, and every slice has a copy of the regions. The join therefore runs on each slice with no rows moved between nodes. Had the orders been distributed on the order date, Redshift would have redistributed one of the two tables across the network for every run of this query. On Redshift, `EXPLAIN` shows this as `DS_DIST_NONE` for a co-located join and `DS_BCAST_INNER` or `DS_DIST_BOTH` when rows have to move.

13.8 The Data API

The Redshift Data API reaches a provisioned cluster by its identifier. With DbUser, AWS issues temporary credentials for that database user:

```
import time

import boto3

data = boto3.client("redshift-data")
statement = data.execute_statement(
    ClusterIdentifier="analytics",
    Database="dev",
    DbUser="admin",
    Sql="SELECT count(DISTINCT customer_id) FROM orders",
)
while True:
    status = data.describe_statement(Id=statement["Id"])["Status"]
    if status in ("FINISHED", "FAILED", "ABORTED"):
        break
    time.sleep(0.5)
result = data.get_statement_result(Id=statement["Id"])
print(status, result["Records"])
```

```
FINISHED [[{'longValue': 2000}]]
```

13.9 Adding nodes

On one laptop, more compute nodes share the same processor cores, so a larger local cluster is not faster; it shows where rows go, not how fast they get there. The speed-up comes from adding machines. On Amazon EC2, with native amd64 nodes of 8 vCPUs each, the author timed one heavy GROUP BY over 40 million rows on the same engine with Citus underneath:

Table 13.1: One aggregation query over 40 million rows on EC2, by number of Citus workers. Speed-up is relative to the single PostgreSQL server.

Configuration	Time (ms)	Speed-up
Single PostgreSQL, 8 vCPUs	1,425	1.00

Configuration	Time (ms)	Speed-up
Citus, 1 worker (8 vCPUs)	1,493	0.95
Citus, 2 workers (16 vCPUs)	829	1.72
Citus, 4 workers (32 vCPUs)	474	3.01

With one worker the cluster is slightly slower than a single server with the same cores, because the coordinator adds a step. Each doubling of the workers then makes the query 1.75 to 1.8 times faster, a little short of twice because the coordinator still merges and sorts the grouped results. These are measurements of one query shape, not a benchmark of Redshift, whose columnar storage and compiled queries behave differently.

13.10 Cleaning up

```
import boto3

redshift = boto3.client("redshift")
try:
    redshift.delete_cluster(
        ClusterIdentifier="analytics",
        SkipFinalClusterSnapshot=True,
    )
except redshift.exceptions.ClusterNotFoundFault:
    pass

s3 = boto3.client("s3")
try:
    listing = s3.list_objects_v2(Bucket="warehouse-staging")
except s3.exceptions.NoSuchBucket:
    listing = None
if listing is not None:
    for obj in listing.get("Contents", []):
        s3.delete_object(
            Bucket="warehouse-staging", Key=obj["Key"]
        )
    s3.delete_bucket(Bucket="warehouse-staging")
```

Deleting the cluster removes its containers and their data.

13.11 Where oblako differs from AWS

- **A compute node is the unit of distribution.** Redshift divides each compute node into slices; oblako hashes rows into shards that are spread over the compute nodes, and `skew_rows` compares rows per compute node. When a node holds no rows of a table, oblako reports no skew value.
- **EVEN and AUTO tables stay on the leader.** Only KEY and ALL tables are distributed. Queries on the other tables return the same results but do not run in parallel.
- **EXPLAIN shows the Citus plan,** with tasks per shard, and has none of Redshift's `DS_DIST` labels. `ALTER TABLE ... ALTER DISTKEY` fails with a syntax error; recreate the table as in the skew listing.
- **The Data API uses the master user's credentials** for a multi-node cluster and accepts `DbUser` without issuing temporary credentials.
- **At most eight compute nodes,** all on one machine, so a larger cluster is not faster. Resizing or pausing a cluster leaves its running nodes as they are.

13.12 Running it on AWS

The listings run unchanged against a real cluster, which bills by the hour per node, so run the cleanup listing as soon as you finish. The cluster needs an IAM role that can read the staging bucket, attached at creation with `IamRoles=[...]` and named in the `COPY`, in place of the made-up role. Your machine reaches the cluster only if it is publicly accessible and its security group allows your address. `redshift_connector` verifies AWS's certificate with the bundle it ships with, which `oblako trust` extended without replacing anything.

13.13 Summary

A multi-node cluster stores each table on its compute nodes according to its distribution style: KEY by a hashed column, ALL as a copy on every node. A distribution key with many distinct values keeps skew low, and distributing the tables you join on their join column keeps the join on each slice. `SVV_TABLE_INFO` shows the style, the sort key and the skew of

every table. On oblako the same API call starts a real cluster of leader and compute nodes, so these choices can be tested on your machine before they reach AWS.

14

Feature Store with Feast

A feature store with Redshift as its offline store and DynamoDB as its online store.

14.1 Before you start

```
oblako up s3
oblako up redshift
oblako up redshift-data
oblako up dynamodb
export AWS_PROFILE=oblako
```

```
uv add "feast[aws]" pyarrow pandas
```

The listings use Feast 0.66 and the Redshift cluster analytics and database oblako of Chapter 7. The last listing removes everything they create.

14.2 Feature stores

A **feature** is a value a model takes as input, such as a customer's number of orders in the last seven days. The same features are needed twice: as history, to build a training set, and

as the latest value, when the model scores one customer in production. Computing them in two places invites the two to drift apart, and a training set built carelessly from history can leak information from after the moment it describes.

A **feature store** keeps both sides consistent. Features are defined once, in code. An **offline store**, usually a warehouse, holds their history; a **point-in-time join** reads, for each training example, the values that were current at that example's timestamp and none later. An **online store**, a key-value database, holds the latest value of each feature for each entity, and **materialization** copies those values from the offline store into it. Feast is an open-source feature store that works with either store from AWS: Redshift offline and DynamoDB online.

Feast's Redshift offline store uses only the Redshift Data API and S3. It uploads the training examples to S3 and loads them into Redshift with `COPY`, runs the point-in-time join as SQL, and unloads the result to S3 with `UNLOAD`. Its DynamoDB online store keeps one table per feature view. A **registry**, a file that Feast keeps in S3 here, records the definitions, so every process that opens the store, from a notebook to a scheduled job, sees the same ones.

What oblako runs

Every part of that path is an oblako engine from earlier chapters. The Redshift Data API (Chapter 7) runs `COPY` and `UNLOAD` against oblako's S3, and DynamoDB sits behind oblako's proxy (Chapter 10), which provides the table tags Feast reads and writes. Feast runs in your Python process (Figure 14.1).

14.3 Feature data in Redshift

The features are daily statistics for 50 customers over 14 days: orders and spend over the previous seven days. The listings run their SQL through the Data API, as Feast does, with one helper:

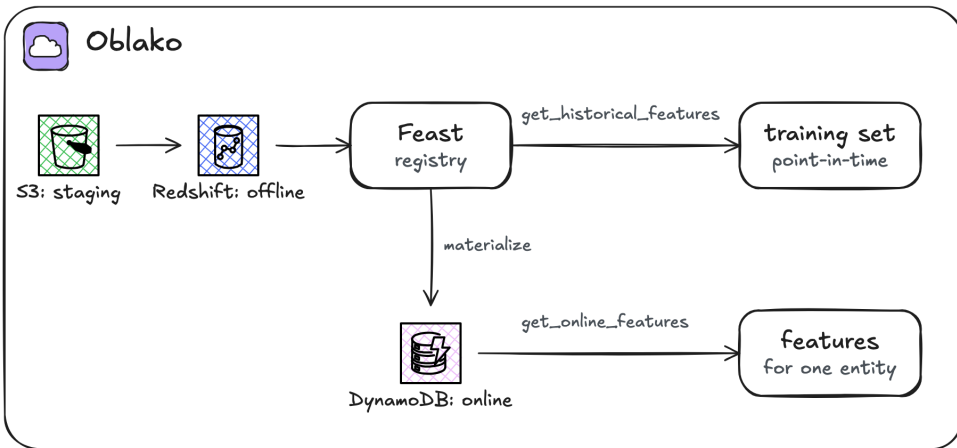


Figure 14.1: Feast on oblako. Redshift holds the feature history, staged through S3; Feast builds training sets from it and materializes the latest values into DynamoDB for online lookups.

```

import time

import boto3

data = boto3.client("redshift-data")

def run(sql):
    statement = data.execute_statement(
        ClusterIdentifier="analytics",
        Database="oblako",
        DbUser="oblako",
        Sql=sql,
    )
    while True:
        desc = data.describe_statement(Id=statement["Id"])
        if desc["Status"] in ("FINISHED", "FAILED", "ABORTED"):
            break
        time.sleep(0.2)
    if desc["Status"] != "FINISHED":
        raise RuntimeError(desc.get("Error"))
    return statement["Id"]

```

The statistics are written to S3 as Parquet and loaded with COPY:

```

import datetime
import io
import random

import boto3
import pyarrow as pa
import pyarrow.parquet as pq
from warehouse import run

s3 = boto3.client("s3")
s3.create_bucket(Bucket="feature-staging")
rng = random.Random(3)
day0 = datetime.datetime(2026, 9, 1)

rows = []
for customer in range(1, 51):
    for day in range(14):
        orders = rng.randint(0, 9)
        spend = round(orders * rng.uniform(10, 60), 2)
        at = day0 + datetime.timedelta(days=day)
        rows.append(
            {
                "customer_id": customer,
                "event_timestamp": at,
                "orders_7d": orders,
                "spend_7d": spend,
                "created": at + datetime.timedelta(hours=1),
            }
        )
buffer = io.BytesIO()
pq.write_table(pa.Table.from_pylist(rows), buffer)
s3.put_object(
    Bucket="feature-staging",
    Key="customer_stats/part-0.parquet",
    Body=buffer.getvalue(),
)

run("""
CREATE TABLE customer_stats (
    customer_id      integer,
    event_timestamp  timestamp,
    orders_7d        bigint,
    spend_7d         double precision,
    created           timestamp)
DISTKEY(customer_id) SORTKEY(event_timestamp)""")

```

```
run("""
    COPY customer_stats
    FROM 's3://feature-staging/customer_stats/'
    IAM_ROLE 'arn:aws:iam::123456789012:role/redshift-copy'
    FORMAT AS PARQUET""")
print(len(rows), "rows in customer_stats")
```

```
700 rows in customer_stats
```

event_timestamp is the time each value describes. created is the time the row was written; when two rows for the same customer have the same event time, Feast uses the later created.

14.4 Defining features

A Feast definition names an **entity**, the thing features describe, and a **feature view**, a set of features read from one source. The module also holds the store's configuration, so the listings share it:

```

from datetime import timedelta

from feast import (
    Entity,
    FeatureStore,
    FeatureView,
    Field,
    RepoConfig,
    ValueType,
)
from feast.infra.offline_stores.redshift import (
    RedshiftOfflineStoreConfig,
)
from feast.infra.offline_stores.redshift_source import (
    RedshiftSource,
)
from feast.infra.online_stores.dynamodb import (
    DynamoDBOnlineStoreConfig,
)
from feast.types import Float64, Int64

FEATURES = ["customer_stats:orders_7d", "customer_stats:spend_7d"]

customer = Entity(
    name="customer",
    join_keys=["customer_id"],
    value_type=ValueType.INT64,
)

customer_stats = FeatureView(
    name="customer_stats",
    entities=[customer],
    ttl=timedelta(days=2),
    schema=[
        Field(name="orders_7d", dtype=Int64),
        Field(name="spend_7d", dtype=Float64),
    ],
    source=RedshiftSource(
        name="customer_stats_source",
        table="customer_stats",
        timestamp_field="event_timestamp",
        created_timestamp_column="created",
    ),
    online=True,
)

```

```

config = RepoConfig(
    project="shop",
    provider="aws",
    registry="s3://feature-staging/feast/registry.db",
    offline_store=RedshiftOfflineStoreConfig(
        cluster_id="analytics",
        region="us-east-1",
        database="oblako",
        user="oblako",
        s3_staging_location="s3://feature-staging/feast",
        iam_role="arn:aws:iam::123456789012:role/redshift-copy",
    ),
    online_store=DynamoDBOnlineStoreConfig(region="us-east-1"),
    entity_key_serialization_version=3,
)

def open_store():
    return FeatureStore(config=config)

```

The feature view's ttl of two days limits how old a value may be: a training example more than two days after a customer's latest statistics gets no values. The configuration names no endpoints: Feast creates boto3 clients for DynamoDB, the Redshift Data API and S3, including the one that reads and writes the registry, and each takes its endpoint from the selected profile. `apply` registers the definitions and creates the online store's DynamoDB table:

```

from store import customer, customer_stats, open_store

store = open_store()
store.apply([customer, customer_stats])
for view in store.list_feature_views():
    fields = [field.name for field in view.features]
    print(view.name, view.ttl, fields)

```

```
customer_stats 2 days, 0:00:00 ['orders_7d', 'spend_7d']
```

Running `apply` again after changing a definition updates the registry and the table; it is safe to run on every deployment.

14.5 Training sets

A training set starts from **labels**: entity keys with the time each example refers to. Here customer 7 has two examples, a week apart, and customer 21 one, at the end of the month:

```
import pandas as pd
from store import FEATURES, open_store

store = open_store()
times = [
    "2026-09-05 18:00",
    "2026-09-12 09:30",
    "2026-09-30 12:00",
]
labels = pd.DataFrame(
    {
        "customer_id": [7, 7, 21],
        "event_timestamp": pd.to_datetime(times),
    }
)
training = store.get_historical_features(
    entity_df=labels,
    features=FEATURES,
).to_df()
print(
    training.sort_values(
        ["customer_id", "event_timestamp"]
    ).to_string(index=False)
)
```

customer_id	event_timestamp	orders_7d	spend_7d
7	2026-09-05 18:00:00+00:00	3.0	162.73
7	2026-09-12 09:30:00+00:00	5.0	197.37
21	2026-09-30 12:00:00+00:00	NaN	NaN

Each row has the values that were current at its own timestamp. Customer 7's first example, at 18:00 on 5 September, gets the statistics dated that midnight; the second gets those of 12 September. Customer 21's example on 30 September is 16 days after the last statistics, beyond the two-day TTL, so its values are missing rather than stale values presented as current. That missing value is also why `orders_7d` prints as 3.0 and 5.0 although the feature is an `Int64`: pandas stores a column with a missing value as floating point. Feast

did the work in Redshift: it loaded the labels with COPY, joined them to customer_stats in SQL, and read the result back through S3.

14.6 Materializing

materialize copies the latest value of each feature, for each customer, within a time range, from Redshift to DynamoDB:

```
from datetime import datetime

import boto3
from store import open_store

store = open_store()
store.materialize(
    start_date=datetime(2026, 9, 1),
    end_date=datetime(2026, 9, 15),
)
dynamodb = boto3.client("dynamodb")
tables = dynamodb.list_tables()["TableNames"]
table = [t for t in tables if t.startswith("shop.")][0]
items = dynamodb.scan(TableName=table, Select="COUNT")["Count"]
print(table, items, "items")
```

Materializing 1 feature views from 2026-09-01 00:00:00+00:00 to 2026-09-15 00:00:00+00:00 into the dynamodb online store.

```
customer_stats:
shop.customer_stats 50 items
```

The table holds one item per customer. In production a scheduled job runs `materialize_incremental`, which continues from where the last run stopped.

14.7 Online features

A service that scores a customer reads the latest features from DynamoDB by key:

```

from store import FEATURES, open_store

store = open_store()
features = store.get_online_features(
    features=FEATURES,
    entity_rows=[{"customer_id": 7}, {"customer_id": 21}],
).to_dict()
for key in ("customer_id", "orders_7d", "spend_7d"):
    print(key, features[key])

```

```

customer_id [7, 21]
orders_7d [4, 3]
spend_7d [132.94, 59.02]

```

These are the statistics of 14 September, the last day materialized, for each customer.

Customer 21's values are the same 16-day-old statistics that the training set left empty. Feast, in version 0.66, does not apply the feature view's `tTL` to online reads: `get_online_features` returns the last values materialized, however old they are. A model would therefore be served values that its training data treated as missing, which is the skew between training and serving that a feature store is meant to prevent. Online freshness depends on materializing often: run `materialize_incremental` on a schedule shorter than the `tTL`, and alert when it fails.

14.8 Cleaning up

```

import boto3
from store import open_store
from warehouse import run

open_store().teardown()
run("DROP TABLE IF EXISTS customer_stats")
s3 = boto3.client("s3")
listing = s3.list_objects_v2(Bucket="feature-staging")
for obj in listing.get("Contents", []):
    s3.delete_object(Bucket="feature-staging", Key=obj["Key"])
s3.delete_bucket(Bucket="feature-staging")

```

`teardown` deletes the online store's tables; the `listing` then drops the source table and deletes the staging bucket, with the registry in it.

14.9 Where oblako differs from AWS

- **Credentials are not checked:** the IAM role in the configuration is a name only, and the Data API uses the engine's user.
- **Speed is your machine's.** The point-in-time join runs on oblako's single Redshift engine; on a cluster (Chapter 13) distributing the source table on the entity key keeps the join on each slice.

14.10 Running it on AWS

With an AWS profile selected, the same code uses Redshift, S3 and DynamoDB in your account. The IAM role named in the configuration must be attached to the cluster and allowed to read and write the staging location, because Feast's COPY and UNLOAD use it. Your credentials need the Data API actions and `redshift:GetClusterCredentials` for the database user. The Data API calls it to get temporary credentials for that user, and it is usually the first permission missing. They also need S3 access to the staging location and the registry, and DynamoDB permissions to create, write and read the online tables. DynamoDB tables bill per request in on-demand mode; `teardown` removes them.

14.11 Summary

Feast keeps training and serving consistent, as long as materialization keeps up, by defining features once and reading them from two stores. Redshift holds the history, and `get_historical_features` builds training sets with a point-in-time join that returns, for each example, only the values current at its timestamp, within the feature view's TTL. `materialize` copies the latest values to DynamoDB, where `get_online_features` reads them by key. On oblako the whole path runs on the Redshift Data API, S3 and DynamoDB of earlier chapters, with the same code as on AWS.

15

Orchestration with MWAA

A daily Airflow pipeline on Amazon MWAA, run on oblako with the same images AWS uses.

15.1 Before you start

```
oblako up s3
oblako up redshift
oblako up redshift-data
oblako up mwaa
export AWS_PROFILE=oblako
```

The first environment you create builds AWS's Airflow image from its sources, which takes about six minutes and 4.7 GB of disk; later environments start in under a minute. The listings use Airflow 3.3.1 and the Redshift cluster `analytics` and database `oblako` of Chapter 7. The last listing removes everything they create.

15.2 Airflow and MWAA

Apache Airflow runs **DAGs**, directed acyclic graphs of tasks written in Python, on a schedule or on demand. A **scheduler** decides which tasks are due, **workers** run them, a **web-server** serves the user interface and the REST API, and a **metadata database** records every run and task. With the Celery executor, which MWAA uses, the scheduler hands tasks to the workers through a queue.

Amazon Managed Workflows for Apache Airflow (MWAA) runs all of these for you as an **environment**. You give it an S3 bucket, the **source bucket**, and a folder in it for DAG files; MWAA copies new and changed files to its scheduler and workers, so deploying a DAG is an upload to S3. A `requirements.txt`, a plugins zip and a startup script in the same bucket are applied when the environment is created or updated. Tasks run with the environment's **execution role**, and the `InvokeRestApi` operation calls Airflow's REST API with your AWS credentials, without logging in to the webserver.

AWS's Airflow images

Since Airflow 2.9, AWS publishes the images MWAA runs as source code, in the GitHub repository `aws/amazon-mwaa-docker-images`, under the Apache-2.0 license. The repository has one folder per Airflow version that MWAA supports, from 2.9.2 to 3.3.1 at the time of writing. Each folder holds the Dockerfiles for a base image and the images built on it, MWAA's own entrypoint in Python, and a Docker Compose file that runs an environment on one machine. The Compose file starts PostgreSQL for the metadata database, ElasticMQ in place of SQS for the Celery queue, and then the webserver, scheduler and worker. It exists so that you can reproduce MWAA's Airflow, with its configuration and installed packages, outside AWS. It replaces `aws-mwaa-local-runner`, an earlier tool for the same purpose, which AWS has archived and which stops at Airflow 2.

What oblako runs

oblako's MWAA engine serves the MWAA API on port 8016. `CreateEnvironment` builds AWS's image for the requested Airflow version, from a fixed commit of that repository, and runs the same containers as its Compose file on a network of their own (Figure 15.1). The engine copies DAG files from the source bucket into the environment every ten seconds, and `InvokeRestApi` reaches the webserver's REST API. Tasks reach oblako's ser-

vices: the containers get the same endpoint settings as the oblako profile, so a DAG's AWS calls go to oblako unchanged.

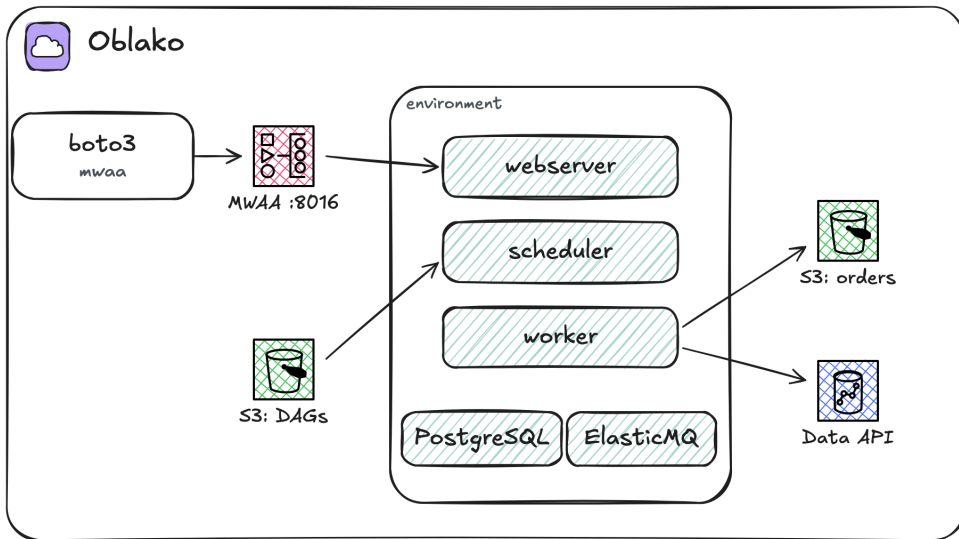


Figure 15.1: An MWAA environment on oblako. The MWAA API starts AWS's Airflow containers; DAGs arrive from the source bucket, and the worker's tasks call oblako's S3 and Redshift Data API.

15.3 The pipeline

The pipeline loads one day of orders into Redshift and computes that day's revenue. An upstream system writes each day's orders as a CSV file under a prefix per day and, when the day is complete, an empty `_SUCCESS` file next to it. The DAG waits for that marker, loads the file with `COPY` and replaces the day's row in a summary table:

```

import datetime

from airflow.providers.amazon.aws.operators.redshift_data import (
    RedshiftDataOperator,
)
from airflow.providers.amazon.aws.sensors.s3 import S3KeySensor
from airflow.sdk import dag

BUCKET = "shop-raw"
DATA_API = {
    "cluster_identifier": "analytics",
    "database": "oblako",
    "db_user": "oblako",
    "wait_for_completion": True,
}

@dag(
    schedule="@daily",
    start_date=datetime.datetime(2026, 9, 1),
    catchup=False,
    tags=["shop"],
)
def daily_orders():
    landed = S3KeySensor(
        task_id="orders_landed",
        bucket_name=BUCKET,
        bucket_key="orders/date={{ ds }}/_SUCCESS",
        poke_interval=10,
        timeout=600,
    )
    load = RedshiftDataOperator(
        task_id="load_orders",
        sql=[
            "DELETE FROM orders WHERE day = '{{ ds }}'",
            "COPY orders FROM "
            "'s3://shop-raw/orders/date={{ ds }}/orders.csv' "
            "IAM_ROLE 'arn:aws:iam::123456789012:role/copy' "
            "CSV IGNOREHEADER 1",
        ],
        **DATA_API,
    )
    summarise = RedshiftDataOperator(
        task_id="daily_revenue",
        sql=[

```

```

        "DELETE FROM daily_revenue WHERE day = '{{ ds }}'",
        "INSERT INTO daily_revenue "
        "SELECT day, count(*), sum(amount) FROM orders "
        "WHERE day = '{{ ds }}' GROUP BY day",
    ],
    **DATA_API,
)
landed >> load >> summarise

```

```
daily_orders()
```

The three tasks are operators from Airflow's Amazon provider, which AWS's image includes. `S3KeySensor` checks for the marker every ten seconds and fails after ten minutes. `RedshiftDataOperator` runs its statements through the Redshift Data API and waits for them to finish. The `{{ ds }}` template is the run's date, so each run touches only its own day, and deleting the day's rows before loading them makes a rerun safe, as in Chapter 16. The schedule is daily; `catchup=False` stops Airflow from scheduling runs for the days between the start date and today.

The SQL runs through the Data API with a helper you have seen in Chapter 14:

```
import time

import boto3

data = boto3.client("redshift-data")

def run(sql):
    statement = data.execute_statement(
        ClusterIdentifier="analytics",
        Database="oblako",
        DbUser="oblako",
        Sql=sql,
    )
    while True:
        desc = data.describe_statement(Id=statement["Id"])
        if desc["Status"] in ("FINISHED", "FAILED", "ABORTED"):
            break
        time.sleep(0.2)
    if desc["Status"] != "FINISHED":
        raise RuntimeError(desc.get("Error"))
    return statement["Id"]
```

The next listing creates the two buckets and the two tables, writes one day of orders with its marker, and uploads the DAG to the source bucket:

```

import boto3
from warehouse import run

s3 = boto3.client("s3")
for bucket in ("shop-raw", "shop-dags"):
    s3.create_bucket(Bucket=bucket)

run("""CREATE TABLE orders (
    day date, order_id integer, customer integer,
    amount decimal(10, 2))""")
run("""CREATE TABLE daily_revenue (
    day date, orders bigint, revenue decimal(12, 2))""")

rows = ["day,order_id,customer,amount"]
rows += [
    f"2026-09-30,{n},{n % 7},{10 + n}.50" for n in range(1, 41)
]
prefix = "orders/date=2026-09-30/"
s3.put_object(
    Bucket="shop-raw",
    Key=prefix + "orders.csv",
    Body="\n".join(rows).encode(),
)
s3.put_object(
    Bucket="shop-raw", Key=prefix + "_SUCCESS", Body=b""
)
s3.upload_file(
    "dags/daily_orders.py", "shop-dags", "dags/daily_orders.py"
)
print("staged", len(rows) - 1, "orders and the DAG")

```

staged 40 orders and the DAG

15.4 Creating the environment

```
import time

import boto3

mwaa = boto3.client("mwaa")
mwaa.create_environment(
    Name="shop",
    AirflowVersion="3.3.1",
    EnvironmentClass="mw1.micro",
    SourceBucketArn="arn:aws:s3:::shop-dags",
    DagS3Path="dags",
    ExecutionRoleArn="arn:aws:iam::123456789012:role/mwaa",
    NetworkConfiguration={
        "SubnetIds": ["subnet-a", "subnet-b"],
        "SecurityGroupIds": ["sg-airflow"],
    },
    WebserverAccessMode="PUBLIC_ONLY",
)
while True:
    env = mwaa.get_environment(Name="shop")["Environment"]
    if env["Status"] != "CREATING":
        break
    time.sleep(15)
print(env["Name"], env["Status"], env["AirflowVersion"])
```

```
shop AVAILABLE 3.3.1
```

`mw1.micro` is MWAA's smallest environment class, with one scheduler, one worker and one webserver. The network settings are required by the API; on AWS they name two private subnets and a security group, and oblako records them. The environment on oblako was available after about 20 seconds; on AWS, the same call took about 40 minutes.

15.5 Running the DAG

The next listing waits until the scheduler has parsed the DAG, unpauses it, starts a run for 30 September and waits for it to finish, all through `InvokeRestApi`:

```

import time

import boto3

mwaa = boto3.client("mwaa")

def rest(path, method="GET", body=None):
    extra = {"Body": body} if body is not None else {}
    resp = mwaa.invoke_rest_api(
        Name="shop", Path=path, Method=method, **extra
    )
    return resp["RestApiResponse"]

while "daily_orders" not in [
    d["dag_id"] for d in rest("/dags")["dags"]
]:
    time.sleep(15) # the DAG processor has not parsed it yet
    rest("/dags/daily_orders", "PATCH", {"is_paused": False})
    run = rest(
        "/dags/daily_orders/dagRuns",
        "POST",
        {"logical_date": "2026-09-30T00:00:00Z"},
    )
    path = f"/dags/daily_orders/dagRuns/{run['dag_run_id']}"
    while rest(path)["state"] not in ("success", "failed"):
        time.sleep(10)
    tasks = rest(path + "/taskInstances")["task_instances"]
    for t in sorted(tasks, key=lambda t: t["start_date"]):
        print(t["task_id"], t["state"])

```

```

orders_landed success
load_orders success
daily_revenue success

```

A new DAG file appears in Airflow only after the scheduler next scans its DAG folder, which Airflow does every five minutes by default; on oblako the DAG appeared after five minutes, and on AWS after about three. `InvokeRestApi` returns Airflow's own response, with fields whose value is null left out. When Airflow answers with an error, boto3 raises `RestApiClientException` or `RestApiServerException`, with Airflow's status code and message in the exception's response.

The summary table holds the day's orders and revenue:

```
import boto3
from warehouse import run

data = boto3.client("redshift-data")
statement = run("SELECT day, orders, revenue FROM daily_revenue")
for row in data.get_statement_result(Id=statement)["Records"]:
    print([list(value.values())[0] for value in row])
```

```
['2026-09-30', 40, '1240.00']
```

15.6 Cleaning up

```
import boto3
from warehouse import run

boto3.client("mwaa").delete_environment(Name="shop")
run("DROP TABLE IF EXISTS orders")
run("DROP TABLE IF EXISTS daily_revenue")
s3 = boto3.client("s3")
for bucket in ("shop-raw", "shop-dags"):
    for obj in s3.list_objects_v2(Bucket=bucket).get(
        "Contents", []
    ):
        s3.delete_object(Bucket=bucket, Key=obj["Key"])
    s3.delete_bucket(Bucket=bucket)
```

Deleting the environment removes its containers, its metadata database and the files synced to it.

15.7 Where oblako differs from AWS

- **Anyone can log in to the webserver.** oblako runs Airflow with AWS's testing authentication, in which every user is an administrator. `CreateCliToken` and `CreateWebLoginToken` are not available; use `InvokeRestApi`, or open `WebserverUrl` in a browser.

- **Logs stay in the containers.** The environment reports MWAA's default logging configuration, with task logs sent to CloudWatch, but oblako does not send them; the webserver shows each task's log.
- **Capacity and networking are recorded only.** The environment class, worker and scheduler counts, subnets and security groups are reported as given; an environment has one worker and the resources of your machine.
- **The execution role is not enforced,** so a task can call any oblako service.

15.8 Running it on AWS

An MWAA environment needs a VPC with two private subnets in different availability zones, with a route to the internet through a NAT gateway or VPC endpoints for the services Airflow uses. The source bucket must have versioning enabled and public access blocked. The execution role must be allowed to read the source bucket and to call what the DAG calls: here `s3:GetObject` and `s3:ListBucket` on the orders bucket, the Data API actions, and `redshift:GetClusterCredentials` for the `db_user`. Your own credentials need the `airflow` actions, including `airflow:InvokeRestApi`.

The environment bills by the hour from creation to deletion, and creating or deleting one takes tens of minutes, so describe it, with its network, in CloudFormation and delete the stack when you finish, as Chapter 17 shows. Deleting the stack leaves behind the log group MWAA created for task logs, named `airflow-` followed by the environment's name and `-Task`; delete it as well. The listings then run unchanged with an AWS profile selected.

15.9 Summary

MWAA runs Airflow's scheduler, workers, webserver and metadata database as an environment that reads its DAGs from an S3 bucket, and `InvokeRestApi` calls Airflow's REST API with AWS credentials. AWS publishes the images MWAA runs, and oblako's MWAA engine runs them per environment. A DAG built from the Amazon provider's operators therefore runs on oblako as written, against oblako's S3 and Redshift, and the same calls create and drive an environment on AWS.

PART IV

Practice



16

Testing pipelines in CI

Unit and integration tests for a small S3-to-Redshift pipeline, and a GitHub Actions workflow that runs them against oblako.

16.1 Before you start

```
oblako up s3
oblako up redshift
```

The chapter's code is a small project with its own `pyproject.toml`, a `tests` folder, and a workflow under `.github/workflows`. Create its environment and trust oblako's Redshift certificate in it, as in Chapter 2:

```
uv sync
oblako trust --python .venv/bin/python
```

The tests create everything they use and remove it when they finish.

16.2 The pipeline

The pipeline under test reads one day of order events from S3, totals each customer's orders and revenue, and loads the totals into a Redshift table. The events are JSON lines under a prefix per day, `raw/orders/date=2026-10-01/`, written by some other system in one or more files. A refund subtracts its amount from the customer's revenue and does not count as an order.

```
import io
import json
from collections import defaultdict
from decimal import Decimal

import boto3
import pyarrow as pa
import pyarrow.parquet as pq
import redshift_connector

def summarise(events):
    """Total each customer's orders; refunds subtract."""
    totals = defaultdict(lambda: [0, Decimal("0")])
    for event in events:
        amount = Decimal(event["amount"])
        if event["type"] == "refund":
            amount = -amount
        else:
            totals[event["customer"]][0] += 1
            totals[event["customer"]][1] += amount
    return [
        {"customer": c, "orders": n, "revenue": r}
        for c, (n, r) in sorted(totals.items())
    ]

def read_events(bucket, day):
    s3 = boto3.client("s3")
    pages = s3.get_paginator("list_objects_v2").paginate(
        Bucket=bucket, Prefix=f"raw/orders/date={day}/"
    )
    for page in pages:
        for obj in page.get("Contents", []):
            body = s3.get_object(Bucket=bucket, Key=obj["Key"])[
                "Body"
            ]
```

```

    ]
    for line in body.iter_lines():
        yield json.loads(line)

def load(conn, bucket, day, rows, schema, role):
    table = pa.Table.from_pylist(
        [{"day": day, **row} for row in rows],
        schema=pa.schema(
            [
                ("day", pa.string()),
                ("customer", pa.string()),
                ("orders", pa.int64()),
                ("revenue", pa.decimal128(12, 2)),
            ]
        ),
    )
    buffer = io.BytesIO()
    pq.write_table(table, buffer)
    key = f"staging/daily_totals/date={day}/part-0.parquet"
    boto3.client("s3").put_object(
        Bucket=bucket, Key=key, Body=buffer.getvalue()
    )
    cur = conn.cursor()
    cur.execute(
        f"DELETE FROM {schema}.daily_totals WHERE day = %s",
        (day,)
    )
    cur.execute(
        f"COPY {schema}.daily_totals FROM 's3://{bucket}/{key}'"
        f" IAM_ROLE '{role}' FORMAT AS PARQUET"
    )
    conn.commit()

def run(day, *, bucket, schema, role, connect):
    rows = summarise(read_events(bucket, day))
    with connect() as conn:
        load(conn, bucket, day, rows, schema, role)
    return len(rows)

def connect_redshift(host, port, database, user, password):
    return lambda: redshift_connector.connect(
        host=host,

```

```

    port=port,
    database=database,
    user=user,
    password=password,
    ssl=True,
    sslmode="verify-ca",
)

```

The work is split so that each part can be tested on its own. `summarise` is a plain function of a list of events. `read_events` and `load` talk to S3 and Redshift. `run` joins them, and takes the bucket, the schema, the IAM role and a connection factory as arguments, so a test can point it at resources it created. `load` writes the totals to S3 as Parquet and loads them with `COPY`, as Chapter 7 did.

A daily job is often run twice for the same day, after a failure or when late events arrive, so `load` deletes the day's rows before copying the new ones. The `DELETE` and the `COPY` are committed together: if the `COPY` fails, the connection closes without a commit and the day's earlier totals remain.

The table is created once, outside the pipeline:

```

CREATE TABLE {schema}.daily_totals (
    day      varchar(10),
    customer varchar(32),
    orders   bigint,
    revenue  decimal(12, 2)
) DISTKEY(customer) SORTKEY(day);

```

16.3 Two kinds of tests

A **unit test** checks a function on its own, with its inputs written in the test. It needs no services and runs in milliseconds, so it can cover many cases: here, the arithmetic of orders and refunds. An **integration test** runs the code against the services it uses in production. It catches what a unit test cannot: a wrong S3 prefix, a column type that `COPY` rejects, SQL that Redshift does not accept, a load that is not safe to repeat. Without a local stack, integration tests either run against a shared AWS account, with its costs and the risk of tests interfering with each other, or replace the services with mocks that accept whatever the code sends. With `oblako` they run against real engines on the machine that runs the tests.

The project marks integration tests so that the two kinds can be run separately:

```
[project]
name = "daily-totals"
version = "0.1.0"
requires-python = ">=3.12"
dependencies = ["boto3", "pyarrow", "redshift-connector"]

[dependency-groups]
dev = ["pytest"]

[tool.pytest.ini_options]
markers = ["integration: needs oblako (or AWS) to be running"]
# redshift_connector builds its SSL context in a deprecated way
filterwarnings = ["ignore::DeprecationWarning:redshift_connector"]
```

16.4 Unit tests

```
from decimal import Decimal as D

from pipeline import summarise

def event(customer, type_, amount):
    return {"customer": customer, "type": type_, "amount": amount}

def test_orders_are_counted_and_summed():
    events = [
        event("c1", "order", "10.50"),
        event("c1", "order", "4.50"),
        event("c2", "order", "7.00"),
    ]
    assert summarise(events) == [
        {"customer": "c1", "orders": 2, "revenue": D("15.00")},
        {"customer": "c2", "orders": 1, "revenue": D("7.00")},
    ]

def test_refunds_are_subtracted():
    events = [
        event("c1", "order", "20.00"),
```

```

        event("c1", "refund", "5.00"),
        event("c3", "refund", "3.00"),
    ]
    assert summarise(events) == [
        {"customer": "c1", "orders": 1, "revenue": D("15.00")},
        {"customer": "c3", "orders": 0, "revenue": D("-3.00")},
    ]

def test_no_events_no_rows():
    assert summarise([]) == []

```

```
uv run pytest -m "not integration" -v
```

```

tests/test_summarise.py::test_orders_are_counted_and_summed PASSED
tests/test_summarise.py::test_refunds_are_subtracted PASSED
tests/test_summarise.py::test_no_events_no_rows PASSED

===== 3 passed, 2 deselected in 0.01s =====

```

16.5 Integration tests

Integration tests need a bucket and a Redshift table, and each test run should have its own, so that two runs on the same machine, or a run that failed halfway, never see each other's data. `pytest` fixtures provide them:

```

import os
import uuid
from pathlib import Path

import boto3
import pytest

from pipeline import connect_redshift

REDSHIFT = {
    "host": os.environ.get("REDSHIFT_HOST", "localhost"),
    "port": int(os.environ.get("REDSHIFT_PORT", "5439")),
    "database": os.environ.get("REDSHIFT_DATABASE", "oblako"),
    "user": os.environ.get("REDSHIFT_USER", "oblako"),
    "password": os.environ.get("REDSHIFT_PASSWORD", "oblako"),
}

ROLE = os.environ.get(
    "REDSHIFT_COPY_ROLE",
    "arn:aws:iam::123456789012:role/redshift-copy",
)

RUN = uuid.uuid4().hex[:8] # one id per test session

@pytest.fixture(scope="session")
def connect():
    return connect_redshift(**REDSHIFT)

@pytest.fixture(scope="session")
def bucket():
    s3 = boto3.client("s3")
    name = f"daily-totals-test-{RUN}"
    s3.create_bucket(Bucket=name)
    yield name
    for page in s3.get_paginator("list_objects_v2").paginate(
        Bucket=name
    ):
        for obj in page.get("Contents", []):
            s3.delete_object(Bucket=name, Key=obj["Key"])
    s3.delete_bucket(Bucket=name)

@pytest.fixture(scope="session")
def schema(connect):
    name = f"test_{RUN}"

```

```
ddl = Path(__file__).parent.parent / "schema.sql"
with connect() as conn:
    cur = conn.cursor()
    cur.execute(f"CREATE SCHEMA {name}")
    cur.execute(ddl.read_text().format(schema=name))
    conn.commit()
yield name
with connect() as conn:
    conn.cursor().execute(f"DROP SCHEMA {name} CASCADE")
    conn.commit()
```

Each test session draws a random identifier and creates a bucket and a schema named after it, with the table inside the schema. The fixtures are session-scoped, so they are created once for all the integration tests, and the code after each `yield` removes them when the session ends, whether the tests passed or failed. The connection settings come from environment variables with `oblako`'s values as defaults, so the same tests can run against a Redshift cluster on AWS by setting the variables, as Chapter 18 does.

The tests write events to S3, run the pipeline and read the table back:

```

import json
from decimal import Decimal as D

import boto3
import pytest

from pipeline import run
from tests.conftest import ROLE
from tests.test_summarise import event

pytestmark = pytest.mark.integration

def put_events(bucket, day, part, events):
    body = "\n".join(json.dumps(e) for e in events).encode()
    key = f"raw/orders/date={day}/part-{part}.jsonl"
    s3 = boto3.client("s3")
    s3.put_object(Bucket=bucket, Key=key, Body=body)

def totals(connect, schema, day):
    sql = (
        "SELECT customer, orders, revenue"
        f" FROM {schema}.daily_totals"
        " WHERE day = %s ORDER BY customer"
    )
    with connect() as conn:
        cur = conn.cursor()
        cur.execute(sql, (day,))
        return [tuple(row) for row in cur.fetchall()]

def test_loads_a_day_from_two_files(bucket, schema, connect):
    day = "2026-10-01"
    put_events(
        bucket,
        day,
        0,
        [
            event("c1", "order", "10.00"),
            event("c2", "order", "5.00"),
        ],
    )
    put_events(bucket, day, 1, [event("c1", "refund", "2.50")])
    loaded = run(

```

```

        day,
        bucket=bucket,
        schema=schema,
        role=ROLE,
        connect=connect,
    )
    assert loaded == 2
    assert totals(connect, schema, day) == [
        ("c1", 1, D("7.50")),
        ("c2", 1, D("5.00")),
    ]

def test_rerunning_a_day_replaces_it(bucket, schema, connect):
    day = "2026-10-02"
    put_events(bucket, day, 0, [event("c1", "order", "3.00")])
    for _ in range(2):
        run(
            day,
            bucket=bucket,
            schema=schema,
            role=ROLE,
            connect=connect,
        )
    assert totals(connect, schema, day) == [("c1", 1, D("3.00"))]
    # the other day's rows are untouched
    assert len(totals(connect, schema, "2026-10-01")) == 2

```

```

export AWS_PROFILE=oblako
uv run pytest -m integration -v

```

```

tests/test_pipeline.py::test_loads_a_day_from_two_files PASSED
tests/test_pipeline.py::test_rerunning_a_day_replaces_it PASSED

===== 2 passed, 3 deselected in 1.27s =====

```

The first test checks that the pipeline reads every file under the day's prefix and that a refund in one file reduces the revenue of an order in another. The second runs the same day twice. Without the `DELETE` in `load`, it fails: the second run appends a second copy of the day's rows, and the assertion reports the extra row.

16.6 Running the tests in CI

GitHub Actions runs a workflow on a fresh virtual machine for each push and pull request. GitHub's hosted Linux runners have Docker installed, which is all oblako needs, so the integration tests run there as they do on your machine:

```
name: tests

on: [push, pull_request]

jobs:
  unit:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: astral-sh/setup-uv@v6
      - run: uv run pytest -m "not integration"

  integration:
    runs-on: ubuntu-latest
    env:
      AWS_PROFILE: oblako
    steps:
      - uses: actions/checkout@v4
      - uses: astral-sh/setup-uv@v6

      - name: Install oblako
        run: >
          uv tool install
          git+https://github.com/almostly/oblako.git

      - name: Start S3 and Redshift
        run: |
          oblako up s3
          oblako up redshift
          oblako configure

      - name: Trust the Redshift certificate
        run: |
          uv sync
          oblako trust --python .venv/bin/python

      - name: Integration tests
        run: uv run pytest -m integration
```

The workflow has two jobs, which run in parallel (Figure 16.1). The unit job needs nothing but Python. The integration job installs `oblako` as a tool, starts S3 and Redshift, writes the `oblako` profile, trusts the Redshift certificate in the project's environment and runs the integration tests with the profile selected for the whole job.

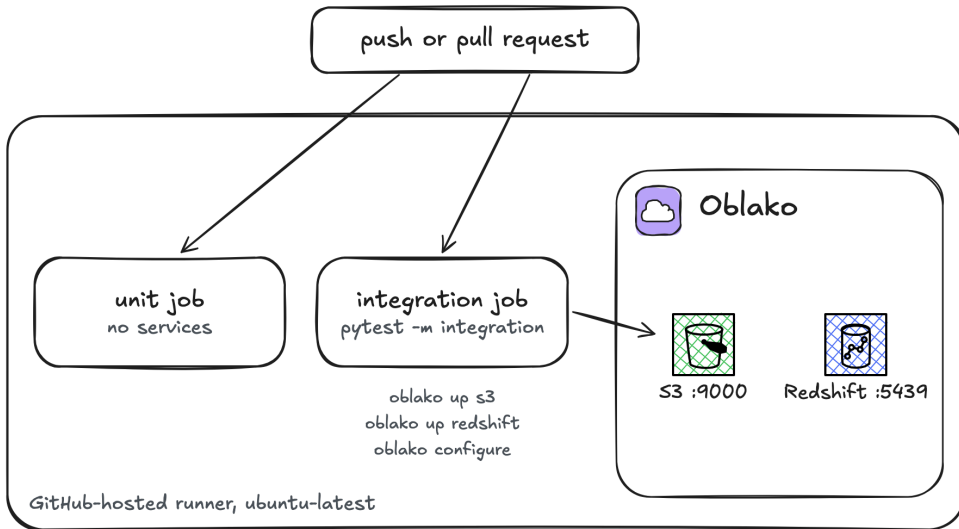


Figure 16.1: The tests in GitHub Actions. Both jobs run on a fresh runner for each push and pull request; the integration job starts `oblako`'s S3 and Redshift on the runner's Docker and runs the tests against them.

`oblako up` waits for each service it starts to accept requests, up to 120 seconds by default, and fails the step when a service is not ready in time; `--timeout` changes the limit. The first start on a runner downloads the service images, so the step takes longer there than on a machine that already has them. A test step that starts before the services are ready fails with connection errors that look like bugs in the code, so failing at the start step makes the cause clear.

The trust step must run after `uv sync`, because it patches the certificate bundle of the installed `redshift-connector`. Without it, every Redshift connection fails:

```

redshift_connector.error.InterfaceError: ('communication error',
SSLCertVerificationError(1, '[SSL: CERTIFICATE_VERIFY_FAILED]
certificate verify failed: self-signed certificate (_ssl.c:1081)'))
  
```

The install step takes oblako's latest code. For a build that does not change under you, pin a commit by appending @ and the commit's hash to the URL.

Which services to start

Start only the services the tests use. Each takes memory and adds to the start-up time, and a runner has less of both than most laptops. A pipeline that also reads from DynamoDB adds `oblako up dynamodb`; one that queries through the Redshift Data API adds `oblako up redshift-data`, which starts a process on the runner instead of a container.

16.7 Where oblako differs from AWS

- **Permissions are not checked.** The tests pass whatever the IAM role and the credentials allow, so a missing permission shows up only on AWS.
- **The cluster is single-node** unless the tests create a multi-node one (Chapter 13). Distribution styles and sort keys are accepted, and on a single node they do not change where rows are stored.
- **Speed is the runner's.** Timings in CI say nothing about the same load on Redshift.

16.8 Summary

Unit tests cover a pipeline's logic with inputs written in the test; integration tests run the pipeline against S3 and Redshift and catch what only the services can reject. Fixtures that create a uniquely named bucket and schema for each test session, and remove them afterwards, keep runs from interfering with each other. In GitHub Actions, the integration job installs oblako, starts the services the tests need, writes the oblako profile and trusts the Redshift certificate. It then runs the tests with that profile selected, on the Docker that every hosted Linux runner has.

17

Infrastructure as code

The same stack of a bucket, an IAM role and Redshift Serverless, written for SAM, Terraform and Pulumi and deployed to oblako.

17.1 Before you start

```
oblako up s3
oblako up redshift
oblako up cloudformation
oblako configure
export AWS_PROFILE=oblako
```

The listings take the admin password from `ADMIN_PASSWORD` and the address allowed to connect from `ALLOWED_CIDR`; on oblako any address will do, such as `203.0.113.7/32`. The chapter uses SAM CLI 1.166, Terraform 1.16 with the AWS provider 6.67, and Pulumi 3.267 with `pulumi-aws` 7.48. Each section deploys the stack and removes it again, so the three can run in any order.

17.2 Describing infrastructure in code

The pipeline of Chapter 16 needs a bucket to stage files in, an IAM role that Redshift assumes to read them with COPY, and a Redshift database to load them into. Creating these by hand, in the console or with one-off scripts, leaves no record of how they were configured and no reliable way to create the same set again. **Infrastructure as code** describes them in files kept with the pipeline's code: a tool compares the description with what exists, creates or changes what differs, and can remove everything it created.

Three tools are common on AWS. **CloudFormation** is AWS's own service: you send it a template, and it creates the resources as a **stack** and keeps the stack's state on its side. **SAM**, the Serverless Application Model, is an extension of CloudFormation templates with a command-line tool, `sam`, that deploys and deletes stacks. **Terraform** and **Pulumi** run on your machine instead: they call each service's API directly and keep their own record of what they created, called **state**. Terraform reads its own configuration language, HCL; Pulumi runs a program in a general-purpose language, here Python, that declares the resources.

What oblako runs

oblako runs a CloudFormation engine on port 8017, which creates a template's resources in oblako's engines and keeps each stack's state across restarts. Terraform and Pulumi need nothing of their own. Both use the AWS SDK for Go, which reads the per-service endpoints in the `oblako` profile from Chapter 2, so with `AWS_PROFILE=oblako` their calls reach oblako's S3, IAM and Redshift Serverless APIs (Figure 17.1). The Go SDK addresses buckets by host name, as `daily-totals-123456789012.localhost:9000`, and reads a bucket's tags through the S3 Control API, and oblako answers both.

17.3 The stack

The three versions describe the same resources:

- a bucket named after the account, `daily-totals-` followed by the account ID, because bucket names are global on AWS;

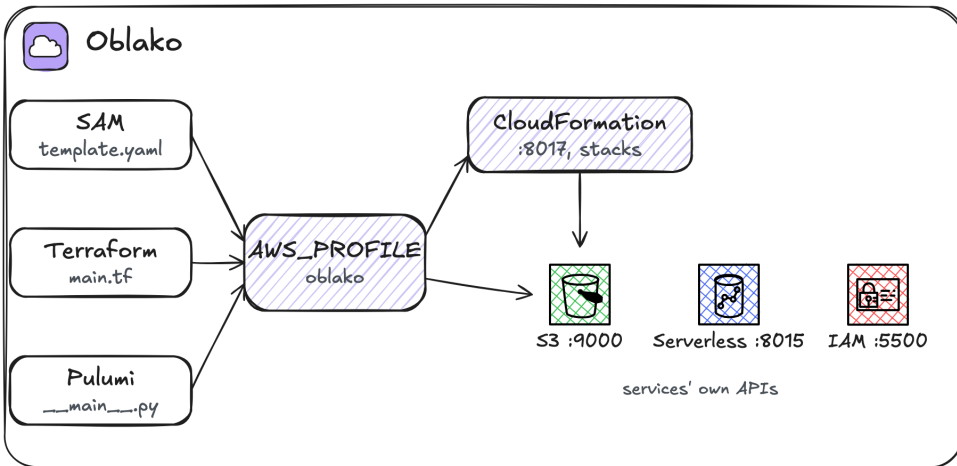


Figure 17.1: Three tools, one profile. SAM sends its template to oblako's CloudFormation engine, which creates the resources; Terraform and Pulumi call the services' own APIs. All three find oblako through `AWS_PROFILE=oblako`.

- an IAM role, `daily-totals-copy`, that Redshift may assume and that may read buckets named `daily-totals-*` in the same account: the staging bucket and those the tests of Chapter 16 create;
- a Redshift Serverless namespace, `daily-totals`, with an admin user and a database named `dev`, and the role attached so that COPY can use it;
- a security group that admits connections to port 5439 from one address, given as a parameter;
- a workgroup, `daily-totals`, with 8 RPU's of base capacity, reachable from outside its VPC through that security group.

Each version outputs the bucket's name, the role's ARN and the workgroup's host name, which the pipeline's tests take as settings. The admin password and the allowed address are parameters, not part of the code.

17.4 SAM

```

AWSTemplateFormatVersion: "2010-09-09"
Transform: AWS::Serverless-2016-10-31
Description: Staging bucket, COPY role and Redshift Serverless

Parameters:
  AdminPassword:
    Type: String
    NoEcho: true
  AllowedCidr:
    Type: String
    Description: The address that may connect to Redshift

Resources:
  Staging:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: !Sub "daily-totals-${AWS::AccountId}"

  CopyRole:
    Type: AWS::IAM::Role
    Properties:
      RoleName: daily-totals-copy
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
          - Effect: Allow
            Principal:
              Service: redshift.amazonaws.com
            Action: sts:AssumeRole

  Policies:
    - PolicyName: read-staging
      PolicyDocument:
        Version: "2012-10-17"
        Statement:
          - Effect: Allow
            Action: [s3:GetObject, s3:ListBucket]
            # the staging bucket and the tests' buckets; the
            # condition keeps them to this account, as bucket
            # ARNs carry no account ID
            Resource:
              - "arn:aws:s3:::daily-totals-*"
              - "arn:aws:s3:::daily-totals-*/*"
            Condition:
              StringEquals:
                aws:ResourceAccount: !Ref AWS::AccountId

```

```

Namespace:
  Type: AWS::RedshiftServerless::Namespace
  Properties:
    NamespaceName: daily-totals
    AdminUsername: pipeline_admin
    AdminUserPassword: !Ref AdminPassword
    DbName: dev
    IamRoles: [!GetAtt CopyRole.Arn]

RedshiftAccess:
  Type: AWS::EC2::SecurityGroup
  Properties:
    GroupDescription: Redshift from one address
    SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 5439
        ToPort: 5439
        CidrIp: !Ref AllowedCidr

Workgroup:
  Type: AWS::RedshiftServerless::Workgroup
  Properties:
    WorkgroupName: daily-totals
    NamespaceName: !Ref Namespace
    BaseCapacity: 8
    PubliclyAccessible: true
    SecurityGroupIds: [!GetAtt RedshiftAccess.GroupId]

Outputs:
  Bucket:
    Value: !Ref Staging
  CopyRoleArn:
    Value: !GetAtt CopyRole.Arn
  RedshiftHost:
    Value: !GetAtt Workgroup.Workgroup.Endpoint.Address

```

!Sub builds a string from parameters and attributes, here the account ID; !GetAtt reads an attribute of another resource, such as the role's ARN or the workgroup's endpoint. sam deploy sends the template as a change set and waits for the stack:

```
sam deploy --stack-name daily-totals \  
  --template-file template.yaml \  
  --capabilities CAPABILITY_NAMED_IAM \  
  --parameter-overrides AdminPassword=$ADMIN_PASSWORD \  
    AllowedCidr=$ALLOWED_CIDR \  
  --no-confirm-changeset
```

```
Key    Bucket  
Value  daily-totals-123456789012  
Key    CopyRoleArn  
Value  arn:aws:iam::123456789012:role/daily-totals-copy  
Key    RedshiftHost  
Value  localhost  
Successfully created/updated stack - daily-totals in None
```

CAPABILITY_NAMED_IAM acknowledges that the stack creates an IAM role with a fixed name. Deleting the stack removes every resource in it:

```
sam delete --stack-name daily-totals --no-prompts
```

17.5 Terraform

```

terraform {
  required_providers {
    aws = { source = "hashicorp/aws", version = "~> 6.0" }
  }
}

provider "aws" {}

variable "admin_password" {
  type      = string
  sensitive = true
}

# the address that may connect to Redshift, such as 203.0.113.7/32
variable "allowed_cidr" {
  type = string
}

data "aws_caller_identity" "me" {}

locals {
  account = data.aws_caller_identity.me.account_id
}

resource "aws_s3_bucket" "staging" {
  bucket = "daily-totals-${local.account}"
}

resource "aws_iam_role" "copy" {
  name = "daily-totals-copy"
  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Effect      = "Allow"
      Principal = { Service = "redshift.amazonaws.com" }
      Action      = "sts:AssumeRole"
    }]
  })
}

resource "aws_iam_role_policy" "read_staging" {
  name = "read-staging"
  role = aws_iam_role.copy.id
  policy = jsonencode({
    Version = "2012-10-17"
  })
}

```

```

Statement = [{
  Effect = "Allow"
  Action = ["s3:GetObject", "s3:ListBucket"]
  # the staging bucket and the tests' daily-totals-* buckets,
  # in this account only: bucket ARNs carry no account ID
  Resource = [
    "arn:aws:s3:::daily-totals-*",
    "arn:aws:s3:::daily-totals-*/*",
  ]
  Condition = {
    StringEquals = { "aws:ResourceAccount" = local.account }
  }
}]
})
}

resource "aws_redshiftserverless_namespace" "ns" {
  namespace_name      = "daily-totals"
  admin_username      = "pipeline_admin"
  admin_user_password = var.admin_password
  db_name             = "dev"
  iam_roles            = [aws_iam_role.copy.arn]
}

resource "aws_security_group" "redshift" {
  name      = "daily-totals-redshift"
  description = "Redshift from one address"
  ingress {
    from_port = 5439
    to_port   = 5439
    protocol  = "tcp"
    cidr_blocks = [var.allowed_cidr]
  }
}

resource "aws_redshiftserverless_workgroup" "wg" {
  workgroup_name      = "daily-totals"
  namespace_name      = aws_redshiftserverless_namespace.ns.id
  base_capacity       = 8
  publicly_accessible = true
  security_group_ids  = [aws_security_group.redshift.id]
}

output "bucket" {
  value = aws_s3_bucket.staging.bucket
}

```

```

}

output "copy_role_arn" {
  value = aws_iam_role.copy.arn
}

output "redshift_host" {
  value = aws_redshiftserverless_workgroup.wg.endpoint[0].address
}

```

The provider `"aws"` block is empty: the region, the credentials and, under the `oblako` profile, the endpoints all come from the profile. `terraform apply` creates the resources and writes what it created to `terraform.tfstate` in the working directory:

```

terraform init
export TF_VAR_admin_password=$ADMIN_PASSWORD
export TF_VAR_allowed_cidr=$ALLOWED_CIDR
terraform apply

```

```

Apply complete! Resources: 6 added, 0 changed, 0 destroyed.
bucket = "daily-totals-123456789012"
copy_role_arn = "arn:aws:iam::123456789012:role/daily-totals-copy"
redshift_host = "localhost"

```

The role policy is its own resource here, so Terraform counts six. Running `terraform plan` afterwards compares the state with what the services report; on `oblako` it found no differences, which shows that `oblako` answers the provider's reads with what it was given. `terraform destroy` removes the six resources. The state file holds the admin password in plain text, like every value Terraform manages, so keep it out of version control or in an encrypted remote backend.

17.6 Pulumi

```

import json

import pulumi
import pulumi_aws as aws

config = pulumi.Config()
me = aws.get_caller_identity()

staging = aws.s3.Bucket(
    "staging", bucket=f"daily-totals-{me.account_id}"
)
copy_role = aws.iam.Role(
    "copy",
    name="daily-totals-copy",
    assume_role_policy=json.dumps(
        {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Principal": {
                        "Service": "redshift.amazonaws.com"
                    },
                    "Action": "sts:AssumeRole",
                }
            ],
        }
    ),
)
aws.iam.RolePolicy(
    "read-staging",
    role=copy_role.id,
    # the staging bucket and the tests' daily-totals-* buckets, in
    # this account only: bucket ARNs carry no account ID
    policy=json.dumps(
        {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Action": ["s3:GetObject", "s3:ListBucket"],
                    "Resource": [
                        "arn:aws:s3:::daily-totals-*",
                        "arn:aws:s3:::daily-totals-*/*",
                    ],
                }
            ],
        }
    ),
)

```

```

        "Condition": {
            "StringEquals": {
                "aws:ResourceAccount": me.account_id,
            },
        },
    ],
},
),
)
access = aws.ec2.SecurityGroup(
    "redshift",
    name="daily-totals-redshift",
    description="Redshift from one address",
    ingress=[
        {
            "protocol": "tcp",
            "from_port": 5439,
            "to_port": 5439,
            "cidr_blocks": [config.require("allowedCidr")],
        },
    ],
)
namespace = aws.redshiftserverless.Namespace(
    "namespace",
    namespace_name="daily-totals",
    admin_username="pipeline_admin",
    admin_user_password=config.require_secret("adminPassword"),
    db_name="dev",
    iam_roles=[copy_role.arn],
)
workgroup = aws.redshiftserverless.Workgroup(
    "workgroup",
    workgroup_name="daily-totals",
    namespace_name=namespace.namespace_name,
    base_capacity=8,
    publicly_accessible=True,
    security_group_ids=[access.id],
)

pulumi.export("bucket", staging.bucket)
pulumi.export("copy_role_arn", copy_role.arn)
pulumi.export("redshift_host", workgroup.endpoints[0].address)

```

Resources are Python objects, and an attribute that is known only after creation, such as the bucket's ARN in the role policy, is an **output** that Pulumi resolves when it creates the

resource. Pulumi keeps its state in a backend; for the book it is a folder in the project, and the passphrase encrypts the secrets stored there:

```
export PULUMI_BACKEND_URL=file://$PWD/.state
export PULUMI_CONFIG_PASSPHRASE=choose-one
pulumi stack init dev
pulumi config set aws:region us-east-1
pulumi config set --secret adminPassword $ADMIN_PASSWORD
pulumi config set allowedCidr $ALLOWED_CIDR
pulumi up --yes
```

```
+ aws:iam:Role copy created (0.04s)
+ aws:iam:RolePolicy read-staging created (0.02s)
+ aws:s3:Bucket staging created (0.08s)
+ aws:redshiftserverless:Namespace namespace created (0.20s)
+ aws:ec2:SecurityGroup redshift created (0.28s)
+ aws:redshiftserverless:Workgroup workgroup created (0.02s)
+ pulumi:pulumi:Stack daily-totals-dev created (0.51s)
Outputs:
bucket      : "daily-totals-123456789012"
copy_role_arn: "arn:aws:iam::123456789012:role/daily-totals-copy"
redshift_host: "localhost"
Resources:
+ 7 created
```

Pulumi counts the stack itself as a seventh resource. `pulumi preview --refresh` reported all seven unchanged, and `pulumi destroy` removes them.

17.7 Choosing between them

The three produce the same resources, and the choice is mostly about where state lives and what language a team prefers to write. CloudFormation keeps state in the service, so there is nothing to store or lock, and SAM adds conveniences for Lambda functions and APIs; it covers AWS only. Terraform's state is a file you manage, usually in an S3 bucket with locking, and its providers cover many clouds and services beyond AWS. Pulumi has the same model as Terraform, with programs in Python, TypeScript, Go or other languages, which suits teams that would rather write loops and functions than a configuration language.

On oblako the difference that matters is coverage. oblako's CloudFormation engine supports a subset of resource types, listed in its documentation, and refuses a template with others. Terraform and Pulumi have no such list: any resource whose API oblako serves can be created through them.

17.8 Where oblako differs from AWS

- **IAM policies are stored, not enforced**, so a role that lacks a permission still works on oblako.
- **A workgroup is oblako's single Redshift engine**. Base capacity, VPC settings and public access are recorded and reported as given.
- **CloudFormation supports a subset of resource types**, and a template that uses another type fails to deploy, with the stack rolled back as on AWS.

17.9 Running it on AWS

With an AWS profile selected, each version creates the same resources in your account. The workgroup bills for its RPU's while it runs queries, and storage for the namespace's data. `PubliclyAccessible` gives the workgroup a public address, and the stack's security group admits your address, given as the allowed CIDR: your public IP address followed by `/32`. Without it, the VPC's default security group would allow connections only from inside the group. When you finish, run `sam delete`, `terraform destroy` or `pulumi destroy`.

17.10 Summary

Infrastructure as code describes resources in files that a tool applies and can remove again. SAM sends a template to CloudFormation, which keeps the state; Terraform and Pulumi call the services' APIs and keep their own state, in HCL and in a general-purpose language respectively. On oblako, SAM uses oblako's CloudFormation engine and the other two reach oblako through the `oblako` profile, so the same description deploys locally and, with another profile, to AWS.

18

Moving to AWS

The pipeline and tests of @sec-testing and the infrastructure of @sec-iac, run in an AWS account with only their settings changed.

18.1 Before you start

You need an AWS account and a profile for it in `~/.aws/config`, created with `aws configure sso` for single sign-on or `aws configure` for access keys. The examples call it `my-account`. Your identity needs permission to create the stack's resources: S3 buckets, an IAM role and its policy, a security group, and a Redshift Serverless namespace and workgroup.

The run below took a few minutes of Redshift Serverless compute, which bills per second for the RPU's in use. It also created two things that bill while idle: the namespace's storage, and the public IPv4 address that public access allocates, which AWS bills by the hour like every public IPv4 address. Delete the stack as soon as you finish. In the outputs, the account ID is replaced by `111122223333`, AWS's documentation placeholder.

18.2 What changes

Nothing in the code changes. The pipeline and its tests take their connection settings from environment variables with oblako's values as defaults, and the Terraform configuration takes the password and the allowed address as variables. Moving to AWS changes four kinds of settings (Figure 18.1):

- **The profile**, which selects the credentials, the region and the endpoints: oblako for oblako, my-account for AWS.
- **The Terraform workspace**, which keeps a separate state for each target, so that the resources on oblako and on AWS are tracked apart.
- **The names the stack outputs**: the bucket and the role carry the account ID, and the workgroup's host name is AWS's.
- **The secrets and the network**: the admin password, and the address that may connect to Redshift.

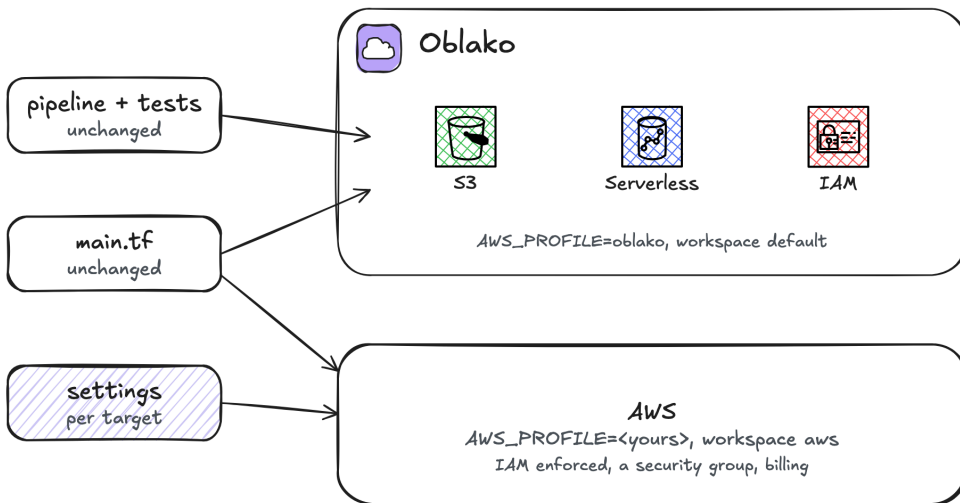


Figure 18.1: One pipeline, one Terraform configuration, two targets. The profile and the Terraform workspace select oblako or AWS; on AWS, IAM is enforced, the workgroup is behind a security group, and the resources bill.

18.3 Deploying the stack

From the Terraform configuration of Chapter 17, select a workspace for AWS, which Terraform creates on first use, and apply with the AWS profile:

```
export AWS_PROFILE=my-account
terraform workspace select -or-create aws
export TF_VAR_admin_password=$ADMIN_PASSWORD
export TF_VAR_allowed_cidr=$(curl -s https://checkip.amazonaws.com)/32
terraform apply
terraform output
```

```
bucket = "daily-totals-11112223333"
copy_role_arn = "arn:aws:iam::11112223333:role/daily-totals-copy"
redshift_host = "daily-totals.11112223333.us-east-1.redshift-serverless.
amazonaws.com"
```

checkip.amazonaws.com returns the address from which your requests reach AWS. As Chapter 17 noted, the admin password ends up in the Terraform state in plain text; on AWS the namespace can instead have Secrets Manager create and hold it, with the provider's manage_admin_password argument. The apply took about a minute. The workgroup reported AVAILABLE at the end of it, but its public endpoint refused connections for about three minutes more, so wait before concluding that the security group is wrong. Running nc -vz with the host name and port 5439 shows when it accepts them.

18.4 Running the tests

The integration tests of Chapter 16 read their settings from the environment. On AWS they are the stack's outputs, the admin user and its password:

```
export REDSHIFT_HOST=$(terraform output -raw redshift_host)
export REDSHIFT_DATABASE=dev
export REDSHIFT_USER=pipeline_admin
export REDSHIFT_PASSWORD=$ADMIN_PASSWORD
export REDSHIFT_COPY_ROLE=$(terraform output -raw copy_role_arn)
uv run pytest -m integration -v
```

```
tests/test_pipeline.py::test_loads_a_day_from_two_files PASSED
tests/test_pipeline.py::test_rerunning_a_day_replaces_it PASSED
===== 2 passed, 3 deselected in 26.37s =====
```

The tests did what they do on oblako. They created a bucket and a schema of their own, wrote events to S3 and ran the pipeline, which loaded the totals with COPY through the role. Then they read the totals back and removed what they created. They took 26.4 seconds, against 1.2 on oblako; the first run after the workgroup was created took 65.5.

18.5 What AWS adds

The run passed because the stack already accounted for what only AWS checks. The most common of these is IAM.

A policy that is too narrow

COPY reads S3 as the role attached to the namespace, so the role's policy must cover every bucket the pipeline loads from. The tests create their own bucket, `daily-totals-test-` followed by a random suffix, which is why the policy allows `daily-totals-*` and not only the staging bucket. To see what the narrower policy does, I replaced the role's policy with one for the staging bucket alone and ran the tests again:

```
FAILED tests/test_pipeline.py::test_loads_a_day_from_two_files
FAILED tests/test_pipeline.py::test_rerunning_a_day_replaces_it

error:    S3ServiceException: User: arn:aws:sts::111122223333:
         assumed-role/daily-totals-copy/RedshiftIamRoleSession
         is not authorized to perform: s3:ListBucket on
         resource: "arn:aws:s3:::daily-totals-test-29ae037a"
         because no identity-based policy allows the s3:ListB
code:     8001
context:  Listing bucket=daily-totals-test-29ae037a prefix=
         staging/daily_totals/date=2026-10-01/part-0.parquet
```

COPY assumed the role and was refused `s3:ListBucket` on the test's bucket; Redshift cuts the message off where it is shown. The same tests pass on oblako with either policy, because oblako stores IAM policies without enforcing them. Running `terraform apply`

again found the policy changed outside Terraform, put the stack's version back, and the tests passed.

Bucket names are global, and a bucket's ARN carries no account ID, so `daily-totals-*` on its own would also match buckets of that name in other accounts. The policy's condition on `aws:ResourceAccount` limits it to the stack's own account.

Other checks

- **The network is closed by default.** A publicly accessible workgroup still accepts only what its security group admits, so the stack opens port 5439 to one address.
- **Certificates are verified.** `redshift-connector` verifies AWS's certificate with the bundle it ships with; `oblako trust` in Chapter 2 added `oblako`'s certificate to that bundle and removed nothing.
- **Resources take time and bill.** The workgroup took about a minute to create, its endpoint some minutes more to accept connections, and its deletion about two and a half minutes.

18.6 Running it from CI

The workflow of Chapter 16 could run the same tests against AWS by selecting an AWS profile in its integration job instead of starting `oblako`. On GitHub Actions the usual way to get credentials is an IAM role that trusts GitHub's OpenID Connect provider, assumed by the `aws-actions/configure-aws-credentials` action, so that no long-lived keys are stored in the repository. The book does not show that workflow, because it was not run for the book; a common arrangement runs the tests against `oblako` on every pull request and against AWS before a release.

18.7 Tearing down

```
terraform destroy
terraform workspace select default
```

```
Destroy complete! Resources: 6 destroyed.
```

Destroying the stack also releases the Elastic IP address that public access allocated. Afterwards no workgroup, namespace, bucket, Elastic IP, security group or role from the run remained in the account; `aws redshift-serverless list-workgroups` and `aws s3 ls` are quick checks.

18.8 Summary

The pipeline, its tests and its Terraform configuration moved from oblako to AWS with only settings changed: the profile, the Terraform workspace, the stack's outputs, the password and the allowed address. On AWS the same tests passed, because the stack already handled what only AWS enforces: the role's access to every bucket COPY reads, and a security group for the workgroup. Running the same code against both, oblako for every change and AWS before release, keeps the differences few and visible.

19

Conclusion

What the book has covered, how to keep oblako's differences from AWS in view as both change, and where to find more.

19.1 What the book covered

The book started from one claim about oblako, that it gives you real behavior in a simulated topology. Each chapter tested that claim on one service or tool by running code against oblako and stating where the result differs from AWS. Part II ran the data services one at a time: S3 and its table and vector buckets, the Glue Data Catalog with Athena, Glue jobs, Redshift, RDS and its replication, DynamoDB, and streaming with Kinesis and Firehose. Part III built on them with Spark and Iceberg, a multi-node Redshift, a feature store with Feast, and Airflow on MWAA. Part IV tested a pipeline in CI, described its infrastructure as code, and ran the same tests on AWS.

Redshift carried much of the weight, because it is where a local copy is hardest to come by. The same engine answered `redshift-connector`, the Data API, `COPY` and `UNLOAD` through S3, SQLAlchemy with reflection and Alembic, `awswrangler` and Feast. Code written against any of them runs on oblako and on AWS without a branch for either.

19.2 Keeping the differences in view

Each chapter on a service or tool had a section on where oblako differs from AWS, and the differences fall into a few kinds. Some checks are not enforced, IAM policies above all, so code with a missing permission passes on oblako. Capacity and topology are recorded but not applied: node counts, worker counts, RPU and VPC settings are reported as you set them while the work runs on your machine. A few features are absent, such as Glue triggers and crawlers, and those chapters said so.

The habit from Chapter 18 keeps these differences small and visible: run the tests against oblako on every change, and against AWS before a release. A test that passes on oblako and fails on AWS usually points to one of the differences above, most often a permission, and it is worth a comment in the code where it was fixed.

oblako also changes, so treat it as a dependency like any other. The book was tested against the version named in the introduction; pin the version you test with, and read the release notes when you move to a newer one.

19.3 Where to find more

oblako's documentation, at <https://oblako-sdk.almostly.ai/>, is the reference for what this book did not cover: every service oblako runs, its port, the operations it supports and its known limits. It changes with each release, so where it and this book disagree, the documentation describes the current version.

The differences the chapters list are known ones: some are by design, such as permissions that are stored but not enforced, and others are features oblako does not have yet. Any other difference, where oblako does something AWS does not, is a bug in oblako, and AWS's behavior is the one to match. Report it, or a missing feature you need, as an issue in oblako's repository, <https://github.com/almostly/oblako>, with the code that shows it and what AWS does with the same code. For AWS's own behavior, the AWS documentation remains the authority.



Other services

Six services outside the book's data focus that oblako also runs, each with one call that works on it.

A.1 Before you start

```
oblako up moto
oblako up stepfunctions
oblako up bedrock-runtime
oblako up ecs
export AWS_PROFILE=oblako
```

Lambda and ECS run containers on Docker, and Bedrock answers from a local model in Ollama, which oblako expects on its default port with `qwen2.5:0.5b` pulled (`oblako pull`). The last listing removes everything the others create.

A.2 Lambda

oblako keeps Lambda's control plane in moto and runs each invocation in a container from a Lambda runtime image, so the handler runs as it would on AWS:

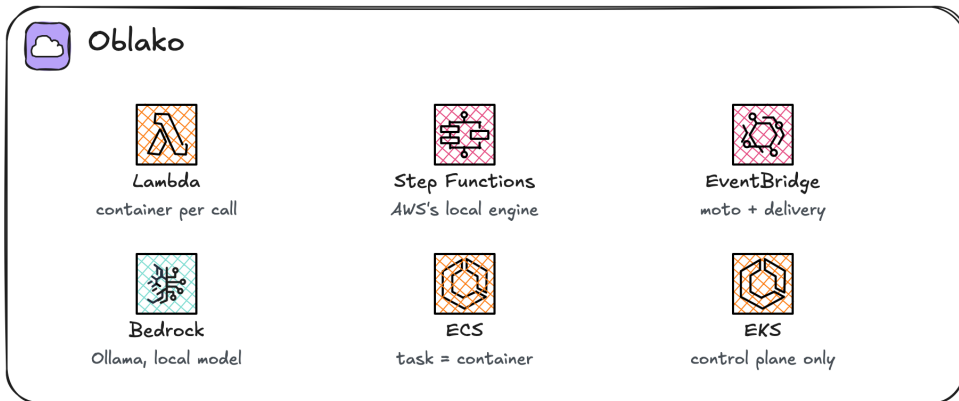


Figure A.1: What runs behind each service. Lambda, Step Functions, Bedrock and ECS run real engines or containers; EventBridge is moto with oblako's delivery added; EKS is a control plane only.

```
import io
import json
import zipfile

import boto3

HANDLER = """
def handler(event, context):
    total = sum(item["amount"] for item in event["orders"])
    return {"orders": len(event["orders"]), "total": total}
"""

package = io.BytesIO()
with zipfile.ZipFile(package, "w") as archive:
    archive.writestr("app.py", HANDLER)

# Lambda needs a role it may assume, on oblako as on AWS
role = boto3.client("iam").create_role(
    RoleName="order-total",
    AssumeRolePolicyDocument=json.dumps(
        {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Principal": {
```

```

        "Service": "lambda.amazonaws.com"
    },
    "Action": "sts:AssumeRole",
}
],
}
),
)["Role"]["Arn"]

lam = boto3.client("lambda")
lam.create_function(
    FunctionName="order-total",
    Runtime="python3.12",
    Role=role,
    Handler="app.handler",
    Code={"ZipFile": package.getvalue()},
)
lam.get_waiter("function_active_v2").wait(
    FunctionName="order-total"
)
event = {"orders": [{"amount": 12.5}, {"amount": 7.5}]}
resp = lam.invoke(
    FunctionName="order-total", Payload=json.dumps(event)
)
print(resp["StatusCode"], json.loads(resp["Payload"]).read())

```

```
200 {'orders': 2, 'total': 20.0}
```

As on AWS, the function's role must exist and trust `lambda.amazonaws.com`, or `Create-Function` fails. The runtime image is `x86_64` with Python 3.12, which Docker emulates on an ARM machine; the first invocation also pulls the image.

A.3 Step Functions

oblako runs `aws-stepfunctions-local`, AWS's own engine for running Step Functions on one machine, which executes the states of a definition:

```

import json
import time

import boto3

definition = {
    "StartAt": "Check",
    "States": {
        "Check": {
            "Type": "Choice",
            "Choices": [
                {
                    "Variable": "$.rows",
                    "NumericGreaterThan": 0,
                    "Next": "Load",
                }
            ],
            "Default": "Skip",
        },
        "Load": {"Type": "Pass", "Result": "loaded", "End": True},
        "Skip": {
            "Type": "Pass",
            "Result": "skipped",
            "End": True,
        },
    },
}

sfn = boto3.client("stepfunctions")
machine = sfn.create_state_machine(
    name="daily-load",
    definition=json.dumps(definition),
    roleArn="arn:aws:iam::123456789012:role/sfn",
)["stateMachineArn"]
for rows in (40, 0):
    run = sfn.start_execution(
        stateMachineArn=machine, input=json.dumps({"rows": rows})
    )["executionArn"]
    while (desc := sfn.describe_execution(executionArn=run))["status"]
        == "RUNNING":
        time.sleep(0.2)
    print(rows, desc["status"], desc["output"])

```

```
40 SUCCEEDED "loaded"
0 SUCCEEDED "skipped"
```

A Task state that calls a Lambda function needs that function reachable from the engine; Pass, Choice, Wait and the other flow states need nothing more.

A.4 EventBridge

moto delivers matching events to SQS, SNS and Lambda targets; oblako adds delivery to Redshift Data API targets and fires scheduled rules:

```
import json

import boto3

sqs = boto3.client("sqs")
events = boto3.client("events")
queue = sqs.create_queue(QueueName="file-arrivals")["QueueUrl"]
queue_arn = sqs.get_queue_attributes(
    QueueUrl=queue, AttributeNames=["QueueArn"]
)["Attributes"]["QueueArn"]

events.put_rule(
    Name="orders-landed",
    EventPattern=json.dumps(
        {
            "source": ["shop.ingest"],
            "detail-type": ["File landed"],
        }
    ),
)
events.put_targets(
    Rule="orders-landed",
    Targets=[{"Id": "queue", "Arn": queue_arn}],
)
events.put_events(
    Entries=[
        {
            "Source": "shop.ingest",
            "DetailType": "File landed",
            "Detail": json.dumps(
                {"key": "orders/date=2026-09-30/"}
```

```

        ),
    }
]
)
msg = sqs.receive_message(QueueUrl=queue, WaitTimeSeconds=5)
body = json.loads(msg["Messages"][0]["Body"])
print(body["detail-type"], body["detail"])

```

```
File landed {'key': 'orders/date=2026-09-30/'}
```

A.5 Bedrock

The Bedrock runtime API on port 8004 translates `Converse` and `InvokeModel` to a local model in Ollama; a Bedrock model ID such as Claude 3 Haiku's is mapped to the local model:

```

import boto3

bedrock = boto3.client("bedrock-runtime")
resp = bedrock.converse(
    modelId="anthropic.claude-3-haiku-20240307-v1:0",
    messages=[
        {
            "role": "user",
            "content": [{"text": "Capital of France, one word?"}],
        }
    ],
    inferenceConfig={"maxTokens": 10, "temperature": 0},
)
print(resp["output"]["message"]["content"][0]["text"])
print(resp["stopReason"], resp["usage"]["inputTokens"] > 0)

```

```

Paris.
end_turn True

```

The answer comes from a half-billion-parameter local model, so the API, the request and the response shape match Bedrock's while the quality does not. For better answers, oblako can send the calls to hosted models through OpenRouter, with your own key.

A.6 ECS

`oblako up ecs` serves the ECS API in front of `moto` and runs each task's containers on Docker; the task's exit code comes back in `DescribeTasks`:

```
import boto3

ecs = boto3.client("ecs")
ecs.create_cluster(clusterName="batch")
ecs.register_task_definition(
    family="count-rows",
    requiresCompatibilities=["FARGATE"],
    networkMode="awsvpc",
    cpu="256",
    memory="512",
    containerDefinitions=[
        {
            "name": "job",
            "image": "public.ecr.aws/docker/library/busybox:1.36",
            "command": ["sh", "-c", "echo 3 rows; exit 0"],
            "essential": True,
        }
    ],
)
task = ecs.run_task(
    cluster="batch",
    taskDefinition="count-rows",
    launchType="FARGATE",
    networkConfiguration={
        "awsvpcConfiguration": {
            "subnets": ["subnet-a"],
            "assignPublicIp": "ENABLED",
        }
    },
)["tasks"][0]
ecs.get_waiter("tasks_stopped").wait(
    cluster="batch", tasks=[task["taskArn"]]
)
done = ecs.describe_tasks(
    cluster="batch", tasks=[task["taskArn"]]
)
for t in done["tasks"]:
    container = t["containers"][0]
    print(
        t["lastStatus"], container["name"], container["exitCode"]
    )
```

```
)
```

```
STOPPED job 0
```

Networking and capacity settings are recorded: an `awsipc` task gets a network interface in its description, while its container runs on Docker's bridge network.

A.7 EKS

EKS is a control plane only: oblako records clusters in moto, with no Kubernetes behind them:

```
import boto3

eks = boto3.client("eks")
eks.create_cluster(
    name="analytics",
    roleArn="arn:aws:iam::123456789012:role/eks",
    resourcesVpcConfig={"subnetIds": ["subnet-a", "subnet-b"]},
)
cluster = eks.describe_cluster(name="analytics")["cluster"]
print(cluster["name"], cluster["status"], cluster["version"])
```

```
analytics ACTIVE 1.19
```

The version is moto's default, an old one, which is a reminder that the cluster is a record and not a cluster; for Kubernetes itself, oblako can run its containers on a local Kubernetes cluster instead of Docker.

A.8 Cleaning up

```

import boto3

boto3.client("lambda").delete_function(FunctionName="order-total")
boto3.client("iam").delete_role(RoleName="order-total")

sfn = boto3.client("stepfunctions")
for machine in sfn.list_state_machines()["stateMachines"]:
    if machine["name"] == "daily-load":
        sfn.delete_state_machine(
            stateMachineArn=machine["stateMachineArn"]
        )

events = boto3.client("events")
events.remove_targets(Rule="orders-landed", Ids=["queue"])
events.delete_rule(Name="orders-landed")
sqs = boto3.client("sqs")
sqs.delete_queue(
    QueueUrl=sqs.get_queue_url(QueueName="file-arrivals")[
        "QueueUrl"
    ]
)

boto3.client("eks").delete_cluster(name="analytics")

ecs = boto3.client("ecs")
for arn in ecs.list_tasks(cluster="batch")["taskArns"]:
    ecs.stop_task(cluster="batch", task=arn)
for arn in ecs.list_task_definitions(familyPrefix="count-rows")[
    "taskDefinitionArns"
]:
    ecs.deregister_task_definition(taskDefinition=arn)
ecs.delete_cluster(cluster="batch")

```