Permify authorization system

Mahmoudreza Kheyrati Fard
Yasin Rezaei
Arsalan Jabbari

Supervisor: Prof.Khunjush
Faculty of Computer Engineering, Shiraz University

Contents

Tools

ZIPKIN

PERMIFY

fn

GO

Apache Airflow

Google zanzibar review

The google zanzibar paper presents a global system designed for google's authorization needs.
Zanzibar provides a uniform data model and configuration language for expressing a wide range of access control policies from several google applications.
It efficiently handles hierarchical namespaces, often a complex aspect of authorizationsystems and can also handle various types of relationships between resources and principal (users,services,etc).
The system is designed for high scalability and low latency, critical for a large-scale company like Google.

Key Concepts:

Namespace: In Zanzibar, a "namespace" defines a type of resource that requires authorization. Each namespace has its own set of rules and policies.

Relations: The system also models various relationships between principals (users, services) and resources. These can be straightforward like 'owner' or 'reader,' or more complex custom relations.

ACL (Access Control Lists): Traditional ACLs are a part of the Zanzibar system but are extended to accommodate more complex, expressive rules.

RBack (Role-Based Access Control): This is one of the most widely used access control models. In RBack, permissions are not assigned to specific users but rather to specific roles. Users are then assigned roles, and through those roles, they inherit permissions. This makes managing permissions easier, especially in large organizations where many users may require the same set of permissions. Roles can also be hierarchical, where higher-level roles inherit permissions from lower-level roles.

ReBAC (Relationship-Based Access Control): This model extends traditional access control paradigms by considering relationships between entities to make access decisions. ReBAC is highly dynamic and can accommodate complex access control requirements based on relationships, not just roles or attributes.

Zookies: Zanzibar uses these short-lived authorization tokens to cache decisions, thereby reducing the need for repeated computation and enhancing performance.

Architecture:

Storage Layer: A globally distributed, strongly consistent database that holds the authorization state, including relationships and policies.

Logic Layer: This is where the rules are evaluated. The system pulls the necessary data from the storage layer and evaluates if an access request should be allowed or denied.

Caching: Given the scale of Google's operations, caching is vital for performance. Zanzibar makes intelligent use of caching to minimize latency and maximize throughput.

Features:

Consistency: Zanzibar ensures that authorization is consistently applied, irrespective of the service making the request.

Expressiveness: The system is designed to model complex relationships and access control scenarios that go beyond simple permission checks.

Scalability: It can handle millions of QPS (queries per second) with low-latency, a necessity for Google's massive, global infrastructure.

Auditability: Because all access control policies are centrally managed, auditing and compliance become more straightforward.

Permify cluster

Permify is an relationship based authorization system for creating and maintaining fine-grained user permissions while ensuring least privilege across your organization.

Key Features

Production ready authorization API that serve as gRPC and REST

Domain Specific Authorization Language - Permify Schema - to easily model your authorization

Database Configuration to store your permissions in house with high availability

Perform access control checks and get answers down to 10ms with parallel graph engine

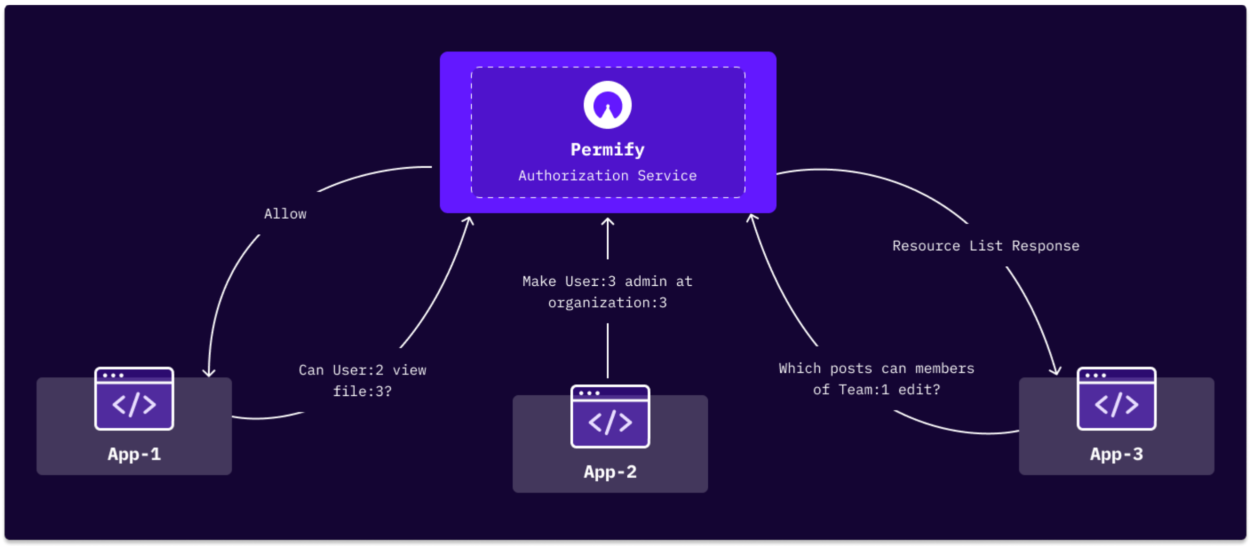
Battle tested, robust authorization architecture and data model based on Google Zanzibar

Create custom permissions for your tenants, and manage them in single place with Multi Tenancy

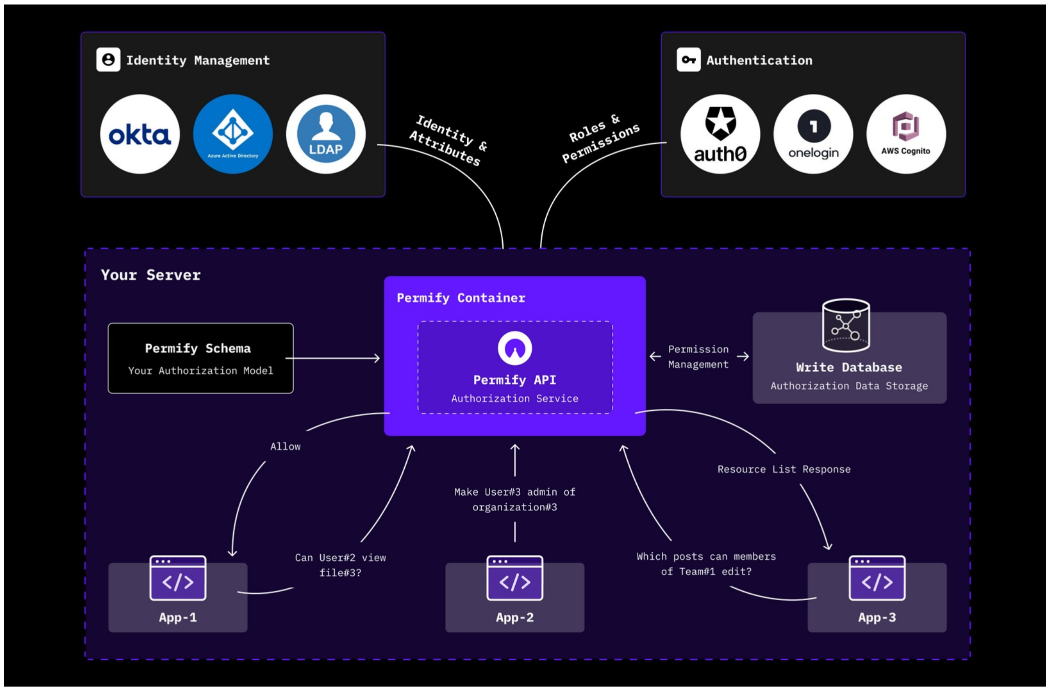
Analyze performance and behavior of your authorization with tracing tools jaeger, signoz or zipkin

Authorization service big flow:

The picture below shows the connections between applications and permify service.



Permify Architecture:

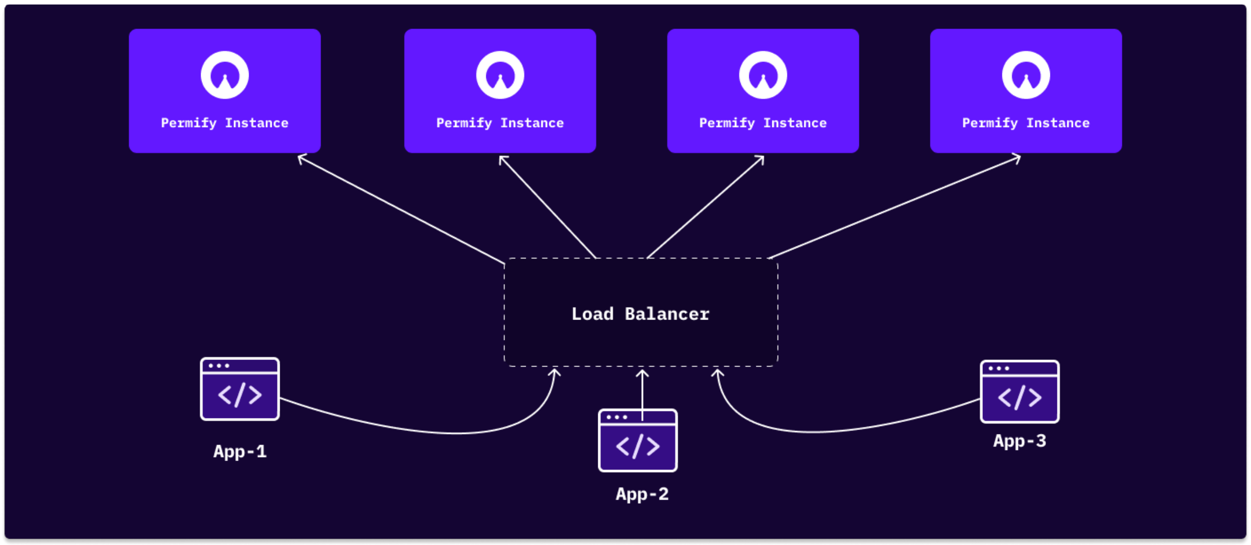


Permify uses a datastore, write db, to store schema, tenants, and permissions.
Permify can connect to various identity management and authentication services.

Permify Deployment Patterns:

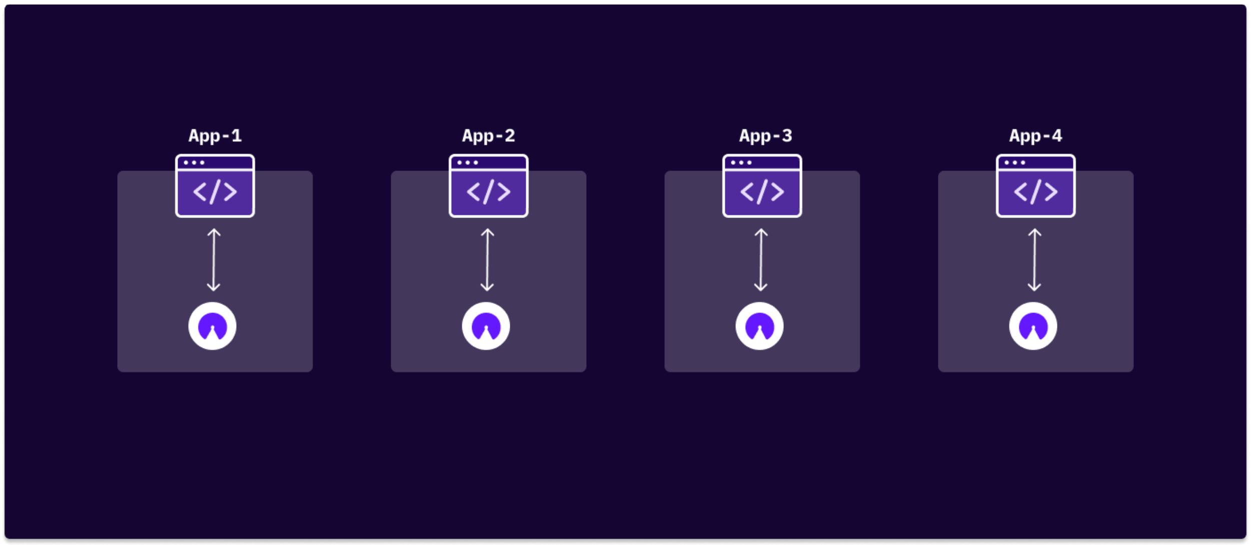
Permify can be deployed as a sole service that abstracts authorization logic from core applications and behaves as a single source of truth for authorization. Gathering authorization logic in a central place offers important advantages over maintaining separate access control mechanisms for individual applications. See the [What is Authorization Service]() Section for a detailed explanation of those advantages.

Permify uses consistent hashing algorithms for cluster management. It uses hashicorp implementation of the consistent hashing algorithm in golang.



SideCar deployment:

Permify can be used as a sidecar as well. In this deployment model, each application uses its own Permify instance and manages its own specific authorization.



Although unified authorization offers many advantages, using the sidecar model ensures high performance and availability plus avoids the risk of a single point of failure of the centered authorization mechanism.

How to use permify?

First we should create tenant which specifies the application domain. Then schemas are created to define entities and relationships between them and after that the permissions cerated. And finally we can send our queries to the permify to figure out the user(or entity) has permission to a resource or not.

Consistent hashing

Consistent hashing is an algorithm used in distributed systems to efficiently distribute data across multiple nodes (such as servers) in a way that minimizes re-distribution of data when the number of nodes changes. The basic idea is to map both the data items and the nodes onto a consistent "hash ring" using a hash function. In this hash ring, each node is responsible for a range of hash values, and each data item's hash value determines which node will store it. When a request comes in for a particular data item, the system hashes the item's key and locates the nearest node in the hash ring to handle the request, either by storing the new data item or retrieving an existing one.

One of the main advantages of consistent hashing is its ability to handle node additions or removals with minimal impact on the system. In traditional hashing, adding or removing a node usually requires rehashing most of the data items, which is computationally expensive and can cause service interruptions. In contrast, with consistent hashing, when a node is added or removed, only a small subset of data items needs to be moved to new nodes. This makes the system more scalable and robust, particularly useful for services like distributed caches, databases, and content delivery networks.

Gossip protocol:

The Gossip Protocol is a decentralized communication protocol used in distributed systems for disseminating information among a network of nodes (like computers or servers) in a scalable and fault-tolerant manner. In a system that employs gossip protocols, each node periodically exchanges information with a randomly selected subset of other nodes in the network. The "gossip" could be any data, such as updates to a dataset, the health status of nodes, or any other kind of state information. Because each node only communicates with a small subset of the network at each round, the protocol is highly scalable, allowing for the quick spread of information throughout the system, much like how gossip spreads in a social network.

One of the key advantages of gossip protocols is their robustness in the face of network failures or partitions. Since there is no centralized coordinator, the system can continue to operate even if some nodes become unreachable. Information eventually propagates to all reachable nodes, assuming that the network is mostly connected over time. This makes gossip protocols particularly useful for large, dynamic, and unreliable environments, such as peer-to-peer networks, cloud computing clusters, and sensor networks. They're often used for tasks like data dissemination, membership tracking, and failure detection.

Serf

Serf relies on an efficient and lightweight gossip protocol to communicate with nodes. The Serf agents periodically exchange messages with each other in much the same way that a zombie apocalypse would occur: it starts with one zombie but soon infects everyone. In practice, the gossip is very fast and extremely efficient.

Permify uses Serf as an implementation of gossip protocol for cluster management.

Add support for NewSql

- Cockroach distributed postgres - xid8 and transactionId + permify issue

NewSql

NewSQL is a category of modern relational database management systems that aim to provide the same scalable performance of NoSQL systems for OLAP (Online Analytical Processing) and OLTP (Online Transaction Processing) workloads, while still maintaining the ACID guarantees (Atomicity, Consistency, Isolation, Durability) and SQL query capabilities of traditional relational databases. These systems often use distributed architectures, sharding, and other advanced techniques to achieve high throughput and low latency for large-scale, distributed applications, offering a blend of the best features from both traditional SQL databases and NoSQL databases.

Cockroachdb

CockroachDB is a distributed, NewSQL database that aims to bring together the best features of traditional relational databases and modern NoSQL architectures. Designed for horizontal scalability and strong consistency, CockroachDB offers ACID-compliant transactions, support for SQL queries, and a self-healing architecture that ensures data availability and resilience against failures. Built with a focus on cloud-native deployment, it enables seamless scaling and rebalancing of data across multiple nodes, making it well-suited for global, distributed applications that require high availability and fault tolerance.

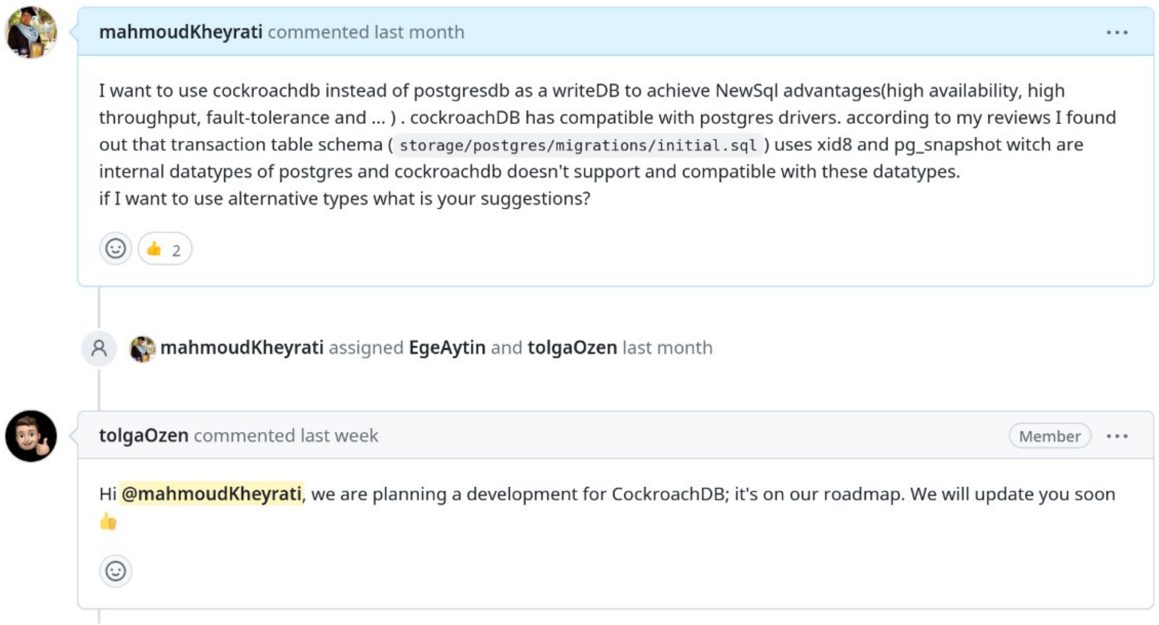
Add support for CockroachDB

Cockroachdb is fully compatible with postgres api and language, but it has its own internal datatypes. The internal datatypes of postgres and cockroachdb can be different and not compatible.

Permify uses xid8 in its database schemas so it’s not compatible with cockroach.

We run cockroachDb cluster and tries int, uint datatype but it stil not works.

We opens an issue in the github and describe the problem



mahmoudKheyrati commented last month

I want to use cockroachdb instead of postgresdb as a writeDB to achieve NewSql advantages(high availability, high throughput, fault-tolerance and ...) . cockroachDB has compatible with postgres drivers. according to my reviews I found out that transaction table schema (`storage/postgres/migrations/initial.sql`) uses xid8 and pg_snapshot witch are internal datatypes of postgres and cockroachdb doesn't support and compatible with these datatypes.
if I want to use alternative types what is your suggestions?

👍 2

mahmoudKheyrati assigned **EgeAytin** and **tolgaOzen** last month

tolgaOzen commented last week — Member

Hi @mahmoudKheyrati, we are planning a development for CockroachDB; it's on our roadmap. We will update you soon 👍

one of the owner said it’s on the roadmap of the permify to integrate with cockroachdb.

But its very time consuming task and it needs to change the structure of the whole project and more abstraction at WriteDB layer.

Permify docker deployment:

Permify single node deployment using postgres as writeDB and zipkin as tracing system.

```yaml
version: '3'

services:
 database:
   image: postgres:13-alpine
   environment:
     POSTGRES_USER: permify
     POSTGRES_PASSWORD: permify
     POSTGRES_DB: permify
   ports:
     - "5432:5432"
   volumes:
     - permify-data:/var/lib/postgresql/data
   networks:
     - permify_net

 zipkin:
   hostname: zipkin
   image: openzipkin/zipkin
   ports:
     - "9411:9411"
   networks:
     - permify_net

 permify:
   image: permify/permify
   ports:
     - "3476:3476"
     - "3478:3478"
   command: "serve --database-auto-migrate=true --database-engine=postgres --database-uri=postgres://root:root@cockroachdb:26257/permify --tracer-exporter=zipkin --tracer-endpoint=http://zipkin:9411/api/v2/spans --tracer-enabled=true "
   depends_on:
     - database
     - zipkin
   networks:
     - permify_net
networks:
 permify_net:

volumes:
 permify-data:
```

Permify can store permissions in memory but in the failures data-loss happens.

Serf deployment:

Serf node needed for cluster managment (gossip protocol). Docker.compose.yaml are shown below:

```
version: '3'
services:
 serf:
   image: dweomer/serf:latest
   command: agent -rpc-addr=0.0.0.0:7373 -node=agent_one -
bind=0.0.0.0:7946
   ports:
     - "7947:7946"
     - "7373:7373"
```

Permify local deployment:

Permify doesn't support multi-nodes on the same machine. Due to our observation This issue is caused by gossip protocol configuration. The serf framework uses a bind port to establish connection and share cluster data and each node in the cluster should have a unique name. This configuration is not configurable in the Permify so we changed the code. The patch available in the root path of the github repo.

To start cluster in single machine we should use the commands below:

```
./permify1 serve --http-port 3476 --grpc-port 3478 --distributed-
enabled=true --distributed-node=localhost:7947 --distributed-
node-na
me=permify2 --distributed-protocol=serf

./permify2 serve --http-port 3476 --grpc-port 3479 --distributed-
enabled=true --distributed-node=localhost:7947 --distributed-
node-na
me=permify2 --distributed-protocol=serf
```

Permify Profiling:

Profiler available in the http://localhost:9898/debug/pprof/ . performance related metrics can be seen in that page that includes (allocs. Goroutines , mutex, …) .

Permify integrate with smart contract:

In our endeavor to enhance the security, auditability, and data transparency of Permify, an open-source implementation of Google Zanzibar, we initially considered integrating smart contract functionalities into the system. The idea was to leverage blockchain technology for immutable and transparent access control logs. However, upon further investigation, we realized that enforcing permissions on the client-side via Web3 would be considered a bad security practice, as client-side environments are inherently less secure and more susceptible to tampering. Moreover, integrating smart contract checks in the backend would not offer any unique advantages in terms of access control, as the system already utilizes robust databases like PostgreSQL and in-memory data structures for this purpose.

Consequently, we decided not to proceed with the smart contract integration, as it would neither improve security nor provide additional functional value, while introducing added complexity and overhead.

Python simple example:

At the beginning of the work, we used a python script to start working with permify and how to work with its APIs and create relations and schemas, and we tested the whole process.

All the details related to the design of schemas and relations and etc, can be accessed and implemented through the main document of permify.

In our Python script, test schemas and relationships were created and finally the accesses were checked randomly.

Serverless deployment using Fn

The Fn Project provides a set of tools and SDKs to build serverless functions. It's container-native, which means each function is bundled as a Docker container.This enables greater portability across different environments.

Core Features:

Multi-Language Support: Write functions in languages like Java, Go, Node.js, and more.

Hot Functions: These are long-lived and can maintain state between invocations for improved performance.

Fn Flow: Allows for composing multiple functions into a workflow.

Local Development: You can test functions locally before deploying.

Fn Server: The server component that manages the deployment and scaling of your functions.

How It Works:

Fn CLI: A command-line tool for creating, deploying, and managing functions.

Function Development Kit (FDK): Libraries that help you bootstrap function development in various languages.

FnServer: The runtime environment where your deployed functions execute.

Flexibility and Portability: One of the standout features is that you're not tied to a specific cloud provider. You can run your Fn functions anywhere Docker containers can run, making it incredibly flexible and avoiding vendor lock-in**.**

The process of implementing the fn project and using it is easily accessible through its main document

Deploying a python function to the Fn project involves several steps, below is a simplified walkthrough:

- Install Fn CLI and start Fn server locally using docker
- Initialize and write functions

  Create new directory for out python and initialize the function
- Deploy function and invoke

Create workflow using Apache Airflow

We use apache airflow for create a workflow that invoke our fn functions:

First, we installed Apache Airflow with docker compose and then start the Airflow web server and scheduler.
Then, we create a new Python script, usually within the `dags` folder. This script will define our DAG and its tasks.

In the Python script, the necessary modules, such as `DAG` and the operators like `BashOperator` or `PythonOperator` imported and then we define a dictionary of default arguments using the `default_args` parameter to set common properties for all tasks in the DAG. These might include the owner, start date, and dependencies on past runs.

After setting up our default arguments, we can instantiate our DAG using the `DAG` class. Within this DAG instance. Each task will have its unique `task_id` and any specific arguments it needs.

Once our tasks are defined, the next step is to set their dependencies. This dictates the order in which tasks will run. we can use methods like `set_upstream` and `set_downstream` or bitshift operators `>>` and `<<` to establish these relationships.

Before deploying the DAG, we can test it using Airflow's Command Line Interface (CLI) to make sure it behaves as expected. Once we've verified that it works, move our Python script into Airflow's DAG folder to make it visible in the Airflow web UI. Finally, we can monitor our DAG runs, view logs, and even modify schedules directly through the web interface.

Airflow Architecture

An Airflow installation generally consists of the following components:

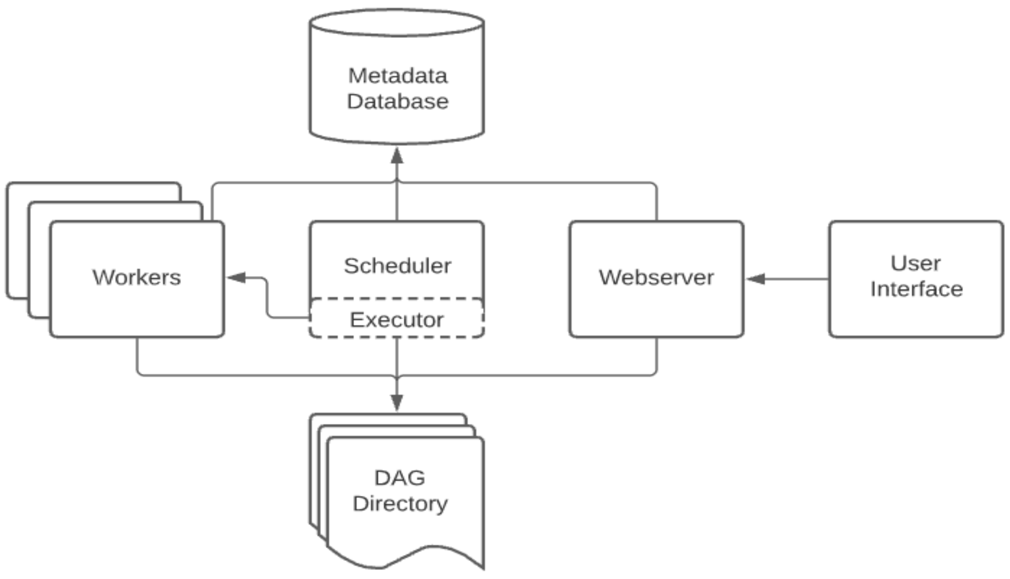
A **scheduler**, which handles both triggering scheduled workflows, and submitting Tasks to the executor to run.

An **executor**, which handles running tasks. In the default Airflow installation, this runs everything inside the scheduler, but most production-suitable executors actually push task execution out to workers.

A **webserver**, which presents a handy user interface to inspect, trigger and debug the behaviour of DAGs and tasks.

A **folder of DAG files**, read by the scheduler and executor (and any workers the executor has)

A **metadata database**, used by the scheduler, executor and webserver to store state.

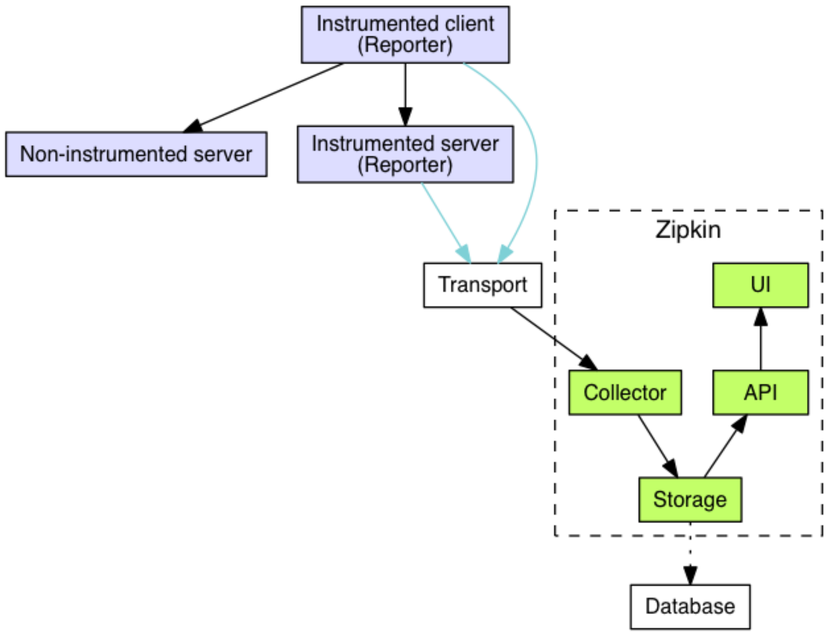


Monitoring with Zipkin

Tracers live in your applications and record timing and metadata about operations that took place. They often instrument libraries, so that their use is transparent to users.

Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in service architectures.

The component in an instrumented app that sends data to Zipkin is called a Reporter. Reporters send trace data via one of several transports to Zipkin collectors, which persist trace data to storage. Later, storage is queried by the API to provide data to the UI.


Instrumented client (Reporter)
Non-instrumented server
Instrumented server (Reporter)
Zipkin
Transport
UI
Collector
API
Storage
Database