Introduction

This document outlines the requirements that are required to enable multi-tenancy in Airflow. This means make it so that a single Airflow cluster can launch jobs for multiple users securely. Many of the subprojects have value in-and-of-themselves.

Next Steps

- Find owners for overarching design for each of the projects below, including an owner for the top-level multi-tenancy project which is just the coordination of the other projects.
- Each owner should come up with an AIP for their project, AIP for each section and an aggregate one, and a roadmap. Figure out how this will play with airflow 2.0/backwards compatibility (especially for Dag Isolation), maybe do this at a level of each project and then bring it together.

Projects

DAG Definition Serialization

Description

Instead of Airflow components (scheduler, websesrver, workers) reading DAG Python files, they would instead read from serialized representations of the DAGs (e.g. JSON files). Instead of DAG owners pushing their DAG files to the Airflow components via varying mechanisms (e.g. git), they would instead call an Airflow CLI to push the serialized DAG representations to some datastore.

Problems Being Solved

- Necessary for proper security/multi-tenancy. Otherwise since a DAG is arbitrary python code, it could change the user it is running as dynamically which would allow a DAG to impersonate another one. E.g. the DAG python file could retrieve the user to run as from a database, which the DAG may or not may have permission to impersonate.
- Enables DAG versioning in the UI. For example, currently if a task is removed from a DAG, all runs of the task will be removed from the UI, but if the definition is serialized you can visualize changes to a DAG's topology over time, i.e. not lose task instances in the UI, not have empty white squares for old versions of tasks in the tree view, etc. You could also remove the "removed" task instance state which would simplify Airflow.
- Enables DAG consistency, i.e. if a DAG definition is changed midway through a dagrun, it will not affect/potentially break the dagrun, since the dagrun is associated to a specific

version of the DAG definition. This also helps give users confidence that the version of a task that is run by a worker is the same one that was scheduled by the scheduler, and the same version they recently uploaded.
- Enables auditing, there will now be a history of who changed what DAG at what time to what definition.
- Makes the scheduler more lightweight since it will no longer be running python files, but just parsing some config. This is also a good stepping stone to making dag updates event based rather than polling based which can greatly decrease the time it takes for DAG updates to propagate.
- Pushes the responsibility of updating DAGs to clients, they get to control the frequency of parsing, the dependencies required to parse them, etc.

Work Required

- Decide on serialization format and data store: strawman is json blob in mysql. Mysql should be the default data store, but there should be an with abstraction to allow storage in other places besides mysql. This should essentially build on the SiimpleDagbag idea in the scheduler and making it a first class citizen.
- Make all services support this serialized format (webserver, worker, scheduler, airflow CLI).
- Write CLI command to take a DAG, serialize it, and then upload it to the data store.
- Figure out how to do versioning between operators in this serialized format (e.g. how to handle the case when operator params change for an operator). Also need to figure out how to support the ad-hoc experimentation use-cases with this new model instead of only production pipelines running on a scheduler.

DAG Isolation

Description

Currently DAG execution dependencies are shared between DAGs, e.g. if two DAGs both depend on the same python library, it’s not possible to change the version of this library in one of the DAGs without changing it in the other, which makes it harder to make changes to DAGs without potentially affecting other production-critical DAGs. This change applies to file dependencies as well (e.g. two DAGs depend on some python file that exists in their git repo). Not only should DAGs be isolated from each other, but they should also be isolated from Airflow dependencies, such that a DAG could use a different python library version than the version of this library used by Airflow.

Problems Being Solved

- Necessary for security/multi-tenancy, otherwise a DAG-owner could modify a shared dependency (e.g. pip package) and it would affect all DAGs, even those that they do not have permissions for.
- Stops updates to shared dependencies from affecting/breaking other DAGs on a cluster. Can safely deploy any changes to a DAG including dependency changes without affecting other DAGs or the Airflow cluster itself.
- Enables experimentation/ad-hoc use cases; users will be able to run multiple versions of the same DAG without worrying about affecting the dependencies of in-flight dagruns.
- If DAG dependencies are decoupled from Airflow it would drastically reduce the size and startup overhead of each DAG, and it would also allow for using different versions of dependencies

Work Required

- Decide on abstraction for DAG container, e.g. do we map DAGs to docker images, some more general abstraction?
- Make workers "dumb", i.e. ideally they don't even depend on Airflow code, and the interface to e.g. pass airflow context and/or/xcom to tasks is via environment variables or a similar mechanism. Have to figure out what other things we would have to solve e.g. how log forwarding would work without airflow code. As part of this change logic like setting the state of tasks to "up for retry" would need to be moved to the scheduler, which is a change that should be made regardless.

Security/Multi-tenancy

Description

Makes it so that a single Airflow cluster can launch jobs for multiple users securely.

Problems Being Solved

- Give the Scheduler a secure source-of-truth for which DAGs are allowed to run as which users.
- Prevents DAGs/users from accessing or affecting other DAGs

Work Required

- We need a way to map accounts/users DAGs are running as to DAGs. This needs to be done in a data store that only the Airflow administrators have access to (the Airflow DB by default is accessible by all users).
- This project depends on the rest of the projects described in this document.

Features Broken By Projects/Workarounds Needed

The following features will be incompatible with the projects listed below. We need to come up with ways to solve the compatibility problems, or simply deprecate some of them.

- **XCom/Airflow Context** - If the Airflow scheduler launches containers that do not run Airflow or have Airflow bundled, then we need some way for them to pull down xcom args from the DB
- **Jinja templates** - Can't serialize as config
- **Python callbacks** - Can't serialize as config
- **Python operator** - Can't serialize as config
- **SubDAGs** - Potentially possible to keep these, but they will make serialization very complicated, they need to be replaced anyways with something more natural, see this for ideas
- **Scalability** - Both for the serialized DAG definitions that need to be pulled down by the scheduler/webserver, and also the containers (e.g. Docker for KubernetesExecutor) that need to be pulled down with low latency even on cold start on the workers.
- **Compatibility with LocalExecutor/StandardTaskRunner** - LocalExecutor would need to support docker and the new serialized DAG definition format/storage backend instead of just parsing DAGs from python files.