AIP-XX - Extendable parsing controls

Motivation

From the very beginning Airflow relied on DAG parsing loop continuously re-parsing and maintaining the state of DAGs in Airflow deployment. This simple approach served Airflow for a long time, but ultimately made a number of basic use cases hard or impossible to implement:

- No way to synchronously update a DAG in a fast & reliable manner. To deploy a new version of the DAG, the user needs to deliver a new version of the code (1) to components and wait until the DAG processor eventually picks up a newer version (2). For larger Airflow deployments the delay between (1) and (2) can be significant. There are workarounds (like bumping the priority),but they either don't achieve a full sync parsing process or have huge latency overhead (like `airflow dags reserialize -S <subdir>`).
- A safer rollout of new DAG code is limited by the fact that any import error removes the previous non-broken version of the DAG and disrupts the execution if it was running. This came from the fact that DAG removal and absence of DAG on next parsing iteration are equivalent in the current implementation.
- Many companies opt-in for building simpler (and more restrictive) workflow orchestration solutions on top of Airflow (for ex. have all DAGs preserved as YAMLs). All these solutions have to hack around Airflow, because the entrypoint in Python files is the only way to define DAG, despite the generation of DAG itself from external combination being trivial.
- Most of the parsing iterations are throw-away CPU cycles. While dynamically generated DAGs (like the one depending on Variables) are supported, most of the DAGs do not require it and are immutable. If they are baked into the container image (a fairly common case) it would've been sufficient to parse them once. But to satisfy a relatively small percentage of dynamic DAGs there is no option to save up resources by avoiding extra re-parsing iterations (partially solved by AIP-66 manifest options).

Solution

Allow parsers to deviate from the out-of-the-box default of:

```
None
# from DagFileProcessor.process_file(file_path).
 1. Execute the file and look for DAG objects in the namespace.
 2. Serialize the DAGs and save it to DB (or update existing record in the DB).
 3. Mark any DAGs which are no longer present as inactive
```

```
4. Record any errors importing the file into ORM
```

Instead the DAG processor is supposed to apply arbitrary *pluggable* strategy to the process of ingestion and deactivating DAGs within a defined bundle.

Path to implementation

1. Refactor DagBag into the DAG store

DagBag historically was the internal interface to have the DAG loaded, all the way to the point where almost any interaction with DAG required it to be parsed. In modern Airflow 2 there are 2 different (and mutually exclusive) use cases for DagBag that are still bound together in a single class:

- `DagBag(read_dags_from_db=True)` - read DAG(s) at the last parsed version from DB
- `DagBag(collect_dags=True)` - parse DAGs in the given subpath to either:
    - Run local CLI command (for ex. `airflow dags …` command). Most of them would work just fine with (read_dags_from_db)
    - Save parsed DAGs to DB

Airflow 3.0 gives us an opportunity to improve the interface and separate it into a separate instances:

- `DagStore`
    - Consists of:
        - read access to DAGs from DB
        - ingest a new DAG(s) version (`submit_dags(dags: Iterable[Dag])`)
        - deactivate no longer active DAGs (`deactivate_dags(dag_ids: Iterable[str])`, `deactivate_subpaths_dags(filepaths: Iterable[str])`)
    - single point responsible for DAG serialization/new version submission (that will use internal API of AIP-72 underneath)
- `DagLoader` - a mechanism to parse DAGs from the bundle backend.
- [depends on AIP-72 implementation, might be skipped] `TaskLoader` - loader of the “task” dependencies, for task executions

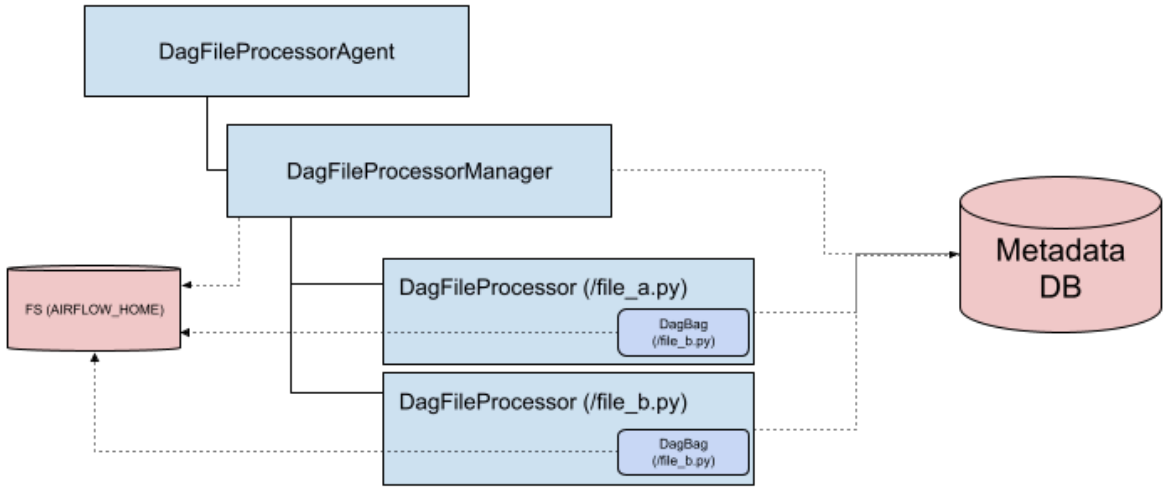
2. Move Callbacks to workers

This is already planned to be *eventually* implemented as part of AIP-72 or follow-up to it (with potential special treatment of notifying callbacks). The bottom-line is that callback executions are fully separated from the DAG parsing process

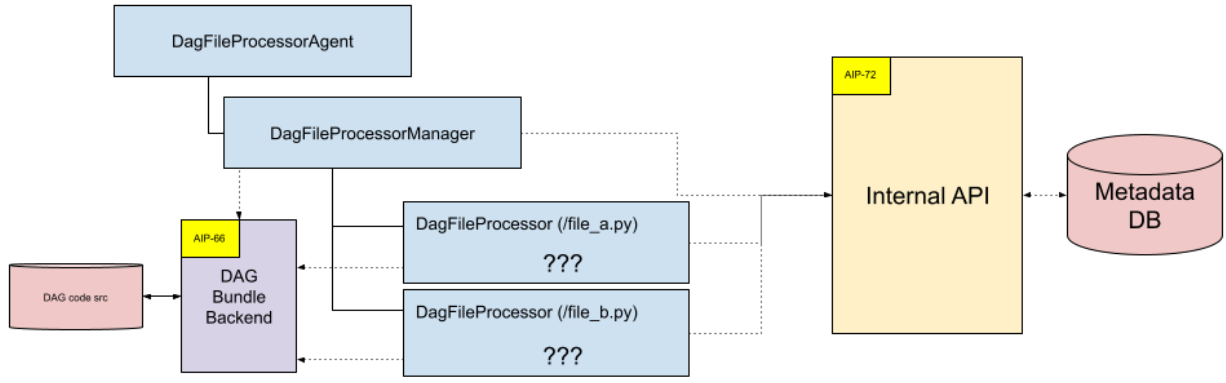
3. Implementation of BundleParser

Bundle parser sits in between the AIP-66s bundle backends, bundle backend implementations and AIP-72 DAG parsing changes.

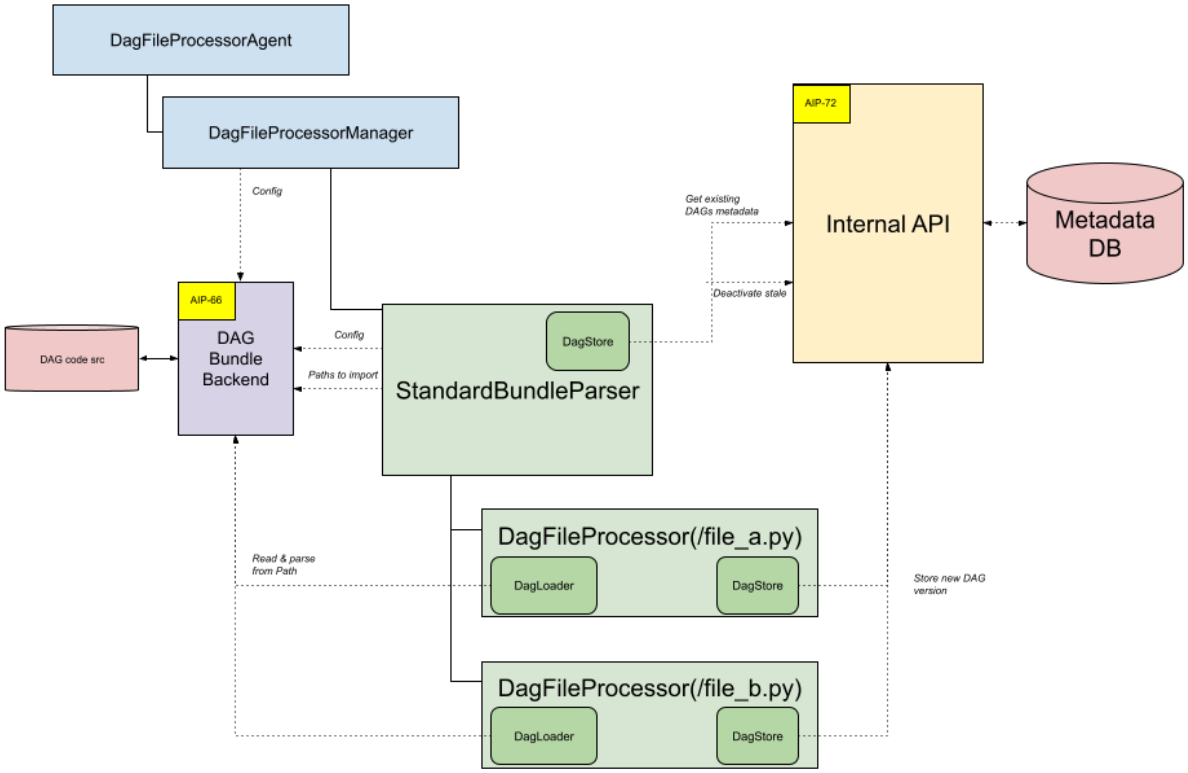
If in Airflow 2 DAG parsing from the high-level looks like



In Airflow 3.0 with currently accepted PoCs the DAG parsing flow would be



After this AIP implementation, it should roughly look like



The definition of the BundleParser can be part of the bundle manifest, making the bundle parser the “consumer” of its configuration.
The fully abstract base of the BundlerParser (AbstractBundleParser) implementation doesn’t define a strict ingestion pipeline.
It only provides:

- Bundle backend to use
- DagLoader as the mechanism to load the DAGs from path
- DagStore as the source of the existing DAGs metadata (like last parse time) and to serialize & store constructed DAGs

The rest is left to the pluggable BundleParser implementation. The following implementations are targeted to be included out-of-the-box:

1. *[default]* **StandardBundleParser** - backwards-compatible implementation. Fully replicates the current flow of DAG management (except for the callbacks processing).
    - It will allow for customization via parameters that will cover the customizations highlighted in AIP-66 (parse on-modification only, different parse frequency, etc.).
    - Additional customization option - do not deactivate DAG if its new version failed to import at its filepath.
2. **SinglePassBundleParser** - the parser that performs a single pass over the bundle, updates re-parsed DAGs, deactivates stale and terminates parsing. Good fit for static

DAGs (e.g. if they are baked into container images) to be run as the last step of a new version rollout.

3. **EventBasedBundleParser** - the parser that only parses and applies changes on the explicit request from the user. Effectively it'd expose the API (protocol & authn/authz to be decided):
    - /ParsePath(subpath, dry_run, deactivate_removed)
    - /DeactivateDag(dag_id)
    - /DeactivateDag(subpath)

    This will allow for both exposing an interactive "dev" environment with synchronous reparsing and wiring-in arbitrary event streaming sources without the loss in performance.

In addition to that, plugins/providers can define their own custom BundleParsers. That can be a natural base to build visual DAG authoring for those, who'd prefer 0-code workflows construction, providing a stable point of integration. Maybe, Airflow will provide its own implementation out-of-the-box one day.

Configuration format

If the bundle doesn't contain a manifest (see AIP-66), **StandardBundleParser** is used (with default parameters).

Otherwise manifest is expected to define configuration for bundle parser:

```
None
version: v1beta
bundle_parser: StandardBundleParser
bundle_fetch_interval: 5 minutes
bundle_config:
  - directories_to_scan:
      - path: /glacier/
        scan_interval: 1 day
        min_file_parse_interval: 1 day
      ...
```

Other changes

- CLI - `airflow dags …` CLI should be ported to DagStore/DagLoader interfaces
- Task execution - one of the points of AIP-72 is the removal of necessity to perform a DAG parsing at the moment of task execution. The exact details are still to be implemented, so there might be a certain overlap (and potential migration to DagStore interface). To be aligned at execution time.

Considerations/Blockers

Performance

DAG parsing in Airflow 2 has years of optimizations behind it. It will give us a strict baseline to target with the default implementation after code restructuring. The following dimensions will be measured as part of the performance tests:

- Number of DAG files
- Number of DAGs in single file
- Number of accesses to other parse-time parameters

Given that the AIP-72 is expected to introduce some additional overhead (e.g. for identity provisioning), some small performance regression is anticipated, however, as acceptance criteria should not allow for more than 5-10% in comparison to Airflow 2.
At the same time alternative parsers will provide the performance improvement in themselves (e.g. by avoiding unnecessary parsing) which can overweight any anticipated regression.

Compatibility

From the UX perspective, there is no significant deviation for the “default” Airflow setup. All Airflow configuration properties are either supported by StandardBundleParser or introduced as breaking one in AIP-66.
Given the timeline of Airflow 3.0 delivery, it is unlikely that the full scope of this AIP will be delivered for 3.0 launch with a target for full code completeness in 3.1. At the same time, certain aspects of the proposals (like refactoring of serialization and DagBag code or establishment of bundle parsing format) would significantly smooth out the delivery of full functionality.

Maintainability

Given the thin interface and overall simplification of flows into clear steps of changing the version and DAG state, the overhead for the maintenance of the pluggable parser interface is expected to be low, so long as it is extensively tested. It might be worth migrating some portion of the end-to-end tests that don’t rely on “dynamic” DAGs to make use of SinglePassBundleParser as it will be an overall simplification of the test flow.

Execution dependencies

The two key dependencies, which this AIP builds upon are AIP-66 and AIP-72 (both in the accepted state). We will work closely with project owners to align the requirements and accommodate for the changes introduced in Airflow architecture during the execution.

Are there any downsides to this change?

While one of the goals of this AIP is to clean up the DAG parsing logic, spread across multiple units, this change does introduce an additional abstraction layer which can be viewed as a

contribution to the complexity. We believe that with the right implementation, the comprehensibility of DAG parsing in Airflow can be improved, despite the introduction of a new concept.