cfa_azure Overview

Description

`cfa_azure` is a python package created by the Peraton team in support of CFA. It eases the setup of infrastructure and the execution of tasks in Azure, so that the focus of Azure Batch workload development can be on the science and modeling rather than the complex code needed to use Azure Batch. Individuals across CFA may need to implement similar workflows related to Azure Batch, making it necessary to standardize these sorts of Azure interactions with a package.

`cfa_azure` simplifies many repeated workflows when interacting with Azure, Blob Storage, Batch, Container Registry and more. For example, programmatically creating a pool in Azure takes multiple credentials and several clients to complete. With `cfa_azure`, creating a pool is reduced to a single function with only a few parameters.

The repo is available at https://github.com/CDCgov/cfa_azure and can be pip installed from GitHub.

Components of `cfa_azure`

The `cfa_azure` module is composed into two main submodules: `clients` and `helpers`. The module `clients` contains the `AzureClient`, which combines the multiple Azure clients needed to interact with Azure and consolidates them into a single client for object-oriented programming. The module `helpers` contains more fine-grained, low-level functions which are used within the `clients` module or can be used independently for more control and flexibility while working with Azure.

Most use cases will make use of the `AzureClient`. It can be initialized using a configuration toml file containing Azure authentication information in the following way:

```
client = AzureClient("./configuration.toml")
```

From there, `client` can be used to upload/download files to/from Blob storage, package and upload Docker containers to Azure Container Registry, create pools in Azure Batch, mount Blob Storage to Batch pools, create jobs and execute tasks in Batch, and monitor Batch jobs.

Benefits

Some benefits of `cfa_azure` have already been discussed above, but the following list is more comprehensive:

- standardized way to interact with Azure
- speeds up time to spin up Azure infrastructure such as Batch pools
- reduces the number of lines of code significantly
- object oriented programming for easier workflows
- low level functions for flexibility while still providing ease of use

Examples

Uploading Docker Container to Azure Container Registry:

```
client.package_and_upload_dockerfile(
    registry_name="test_registry", repo_name="repo1", tag="latest"
)
```

Create a Blob container and provide a relative path to use within Azure Batch:

```
client.create_blob_container("containername", "/path")
```

Upload Files in a Folder to Blob Storage:

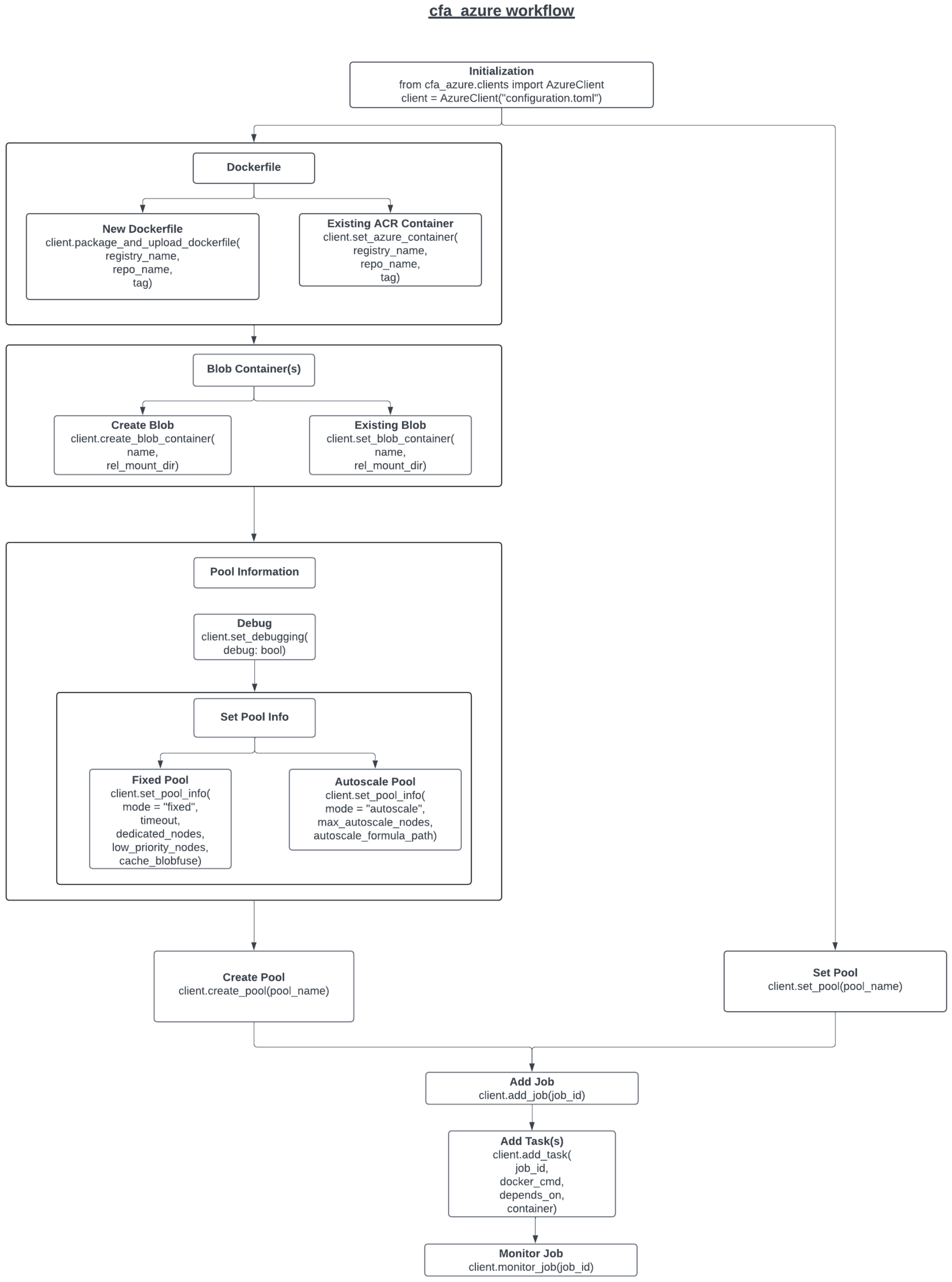
```
client.upload_files_in_folder(
    ["data", "input"],
    blob_container = "containername")
```

Add a Task to Job in Azure Batch:

```
client.add_task(job_id="example_job", docker_cmd="python3 main.py")
```

Workflow

The diagram below illustrates the flow of using the `AzureClient` depending on use case.


cfa_azure workflow
Initialization
from cfa_azure.clients import AzureClient
client = AzureClient("configuration.toml")
Dockerfile
New Dockerfile
client.package_and_upload_dockerfile(
registry_name,
repo_name,
tag)
Existing ACR Container
client.set_azure_container(
registry_name,
repo_name,
tag)
Blob Container(s)
Create Blob
client.create_blob_container(
name,
rel_mount_dir)
Existing Blob
client.set_blob_container(
name,
rel_mount_dir)
Pool Information
Debug
client.set_debugging(
debug: bool)
Set Pool Info
Fixed Pool
client.set_pool_info(
mode = "fixed",
timeout,
dedicated_nodes,
low_priority_nodes,
cache_blobfuse)
Autoscale Pool
client.set_pool_info(
mode = "autoscale",
max_autoscale_nodes,
autoscale_formula_path)
Create Pool
client.create_pool(pool_name)
Set Pool
client.set_pool(pool_name)
Add Job
client.add_job(job_id)
Add Task(s)
client.add_task(
job_id,
docker_cmd,
depends_on,
container)
Monitor Job
client.monitor_job(job_id)