Initial User API Design and System Architecture

This is the design for the user facing API (i.e., this is the API users are going to use to build an Anacostia Pipeline). I want the user to interact with an API that allows them to build a DAG representing their pipeline.

We can think of the pipeline as a DAG.

* Each node in the DAG represents a stage of the pipeline.
  + Nodes are responsible for executing the code for a particular stage of the pipeline (e.g., loading data, transforming data, training models, saving models, deploying models, etc.)
  + A node listens to either:
    - Signals emitted from other nodes.
    - A resource (i.e., monitor when the state of that resource changes); examples of a node listening to a resource include (but not limited to):
      * Listening to an HTTP endpoint for when it produces new data (like if the user is acquiring data through a 3rd party API, the node will keep pinging the API to see if any new data is produced by the API).
      * Watching a user’s filesystem directory for when new files are added to the directory.
      * Watching for when new rows are inputted into a user’s database.
      * Watching our own metadata storage for when retraining runs produce data about a new model.
      * Watching our own model registry for when a new model trained and saved into the registry.
      * Watching our own feature store for new feature vectors.
      * Watching the Microsoft Outlook API for when a senior data scientist clicks “approve” in an email.
  + A node can ONLY emit a signal when the following conditions are met:
    - Trigger function returns true.
    - The node has received a signal from all the nodes its listening to.
  + A node can write to a resource by acquiring the mutex lock to access the resource.
* Each edge in the DAG represents:
  + The path signals take from one node to another (basically shows which node is outputting signals and which nodes are listening to those signals).
  + The label of the path represents how two nodes are connected (e.g., a node listening to the output signal of another node will have an edge with the label “signal”; whereas a node listening to a resource (say an S3 bucket) will have a label of “s3://path/to/folder”
* A DAG can be spread over multiple servers, e.g., you can have a subgraph running on the main server which sends an output signal in the form of a POST request once the model has finished training. From there, you can have a subgraph running on mobile/edge devices responsible for optimizing the model for on a specific platform (like iOS) and one of the leaf nodes in the subgraph listens to an API endpoint for updated models from the main server.
* The nodes of a subgraph can be bundled into a single node and used as part of a larger graph. This allows us to standardize certain workflows into a node and create “guardrails” for users.
* We can enforce correctness of a pipeline by requiring certain nodes take certain input signals. E.g.,
  + Require data preprocessing nodes to get a signal from a resource node.
  + Require a training node to get a signal from a data preprocessing node and a model loading node.
  + Require a model evaluation node to get a signal from a training node and a node responsible for loading a benchmark model.

Class Node:

* Setup:
  + Responsible for downloading and connecting tools to pipeline.
  + Tools installed during setup are the tools needed for the running the trigger function.
  + These tools can be installed and configured using docker python.
  + Responsible for spinning up the node:
    - This is basically like \_\_init\_\_ but I want to do the initialization based on the order determined by a topological sorting of the DAG not when the user creates the Node object.
    - Responsibilities for initializing the node includes:
      * Creating a new process (or thread) to run the trigger function.
      * Starting the infinite loop which listens for signals and runs the trigger function.
* Subscriptions list:
  + List of nodes to list to.
  + I’m thinking we can implement this via having the node subscribe to another node’s Kafka topic.
* Trigger function:
  + Trigger function describes the action that is supposed to take place when the node executes. Examples include:
    - Training loop
    - Cleaning data
    - Creating python packages and docker images for deployment
  + Trigger will only execute ONLY when it received signals from all the nodes in the subscriptions list.

Class DAG:

* Keeps track of relevant information about the DAG resources:
  + IP addresses, ports, docker volumes, filepaths, URIs, account info, etc.
  + The file used to track the relevant information can be used to recreate the DAG in another environment.
* Uses topological sort to determine the order to which node’s setup() function needs to be called.
  + Note: nodes that are not listening to a trigger are leaf nodes and thus get initialized first.
  + Leaf nodes must be resource nodes (node that listen to a resource).
  + Note: the topological order of the graph can also be used to know which tools need to be set up first (i.e., which docker containers need to be set up first)
* Responsible for cleaning up resources if a node is removed from the DAG.

Nodes we should create right now:

* Resource nodes:
  + Nodes responsible for reading from and writing to a SQL database.
  + Nodes responsible for loading/saving a model from/to a model registry.
  + Nodes responsible for loading/saving vectors from/to a feature store.
  + Nodes responsible for loading/saving data from/to an artifact storage.
* AND nodes:
  + This node emits a signal ONLY when it receives a signal from all nodes it is subscribed to.
* OR nodes:
  + This node emits a signal when it receives a signal from one of the nodes it is subscribed to.
* Example nodes: we build these just to use as an example.
  + Nodes for processing data.
  + Nodes for training models.
  + Nodes for evaluating models.
  + Nodes for deploying models.

Key thing to note: this DAG approach is very similar to the approach used by [Dataiku](https://knowledge.dataiku.com/latest/getting-started/dataiku-ui/concept-flow.html) but there are some key differences:

* We are also managing resources (i.e., spinning up docker containers, configuring the containers to connect to the pipeline, deleting containers and images when necessary)
* I think the Dataiku Data Science Studio (DSS) does a little too much with their GUI. Eventually, I do want to build a GUI for this but only for visualizing the flow of the pipeline (just to see which stage is still executing).
* I have yet to see Dataiku be used in edge and mobile environments.
* Eventually, our focus should be on creating our own components that provide state-of-the-art security, privacy, and transparency (e.g., verifying data with ZKP, deploying MLOps pipelines as blockchain smart contracts, TEE-based inference servers, etc).
* I don’t want everyone to think we are trying to mimic Dataiku, we are taking inspiration from them. But the problems we are trying to solve is fundamentally different from Dataiku.