Iraya Use Case Design Document

This document contains a proposed design for the use case in Neil Ongkingco's Iraya Backend Engineer Interview Test.

General Overview

The system is designed to be a cluster of 5 nodes (1 parent VM and 4 worker VMs) using the Kubernetes and Docker cluster and container frameworks. Each node will host several software containers (a self-contained application and runtime environment that is isolated from other containers) that will provide the functionality required to fulfill the responsibilities of the node.

In addition to the 5 nodes, the system uses a keyed **FileStore** to store the raw file conents and a **Transaction Database** (SQL) for storing basic information, task status and compute results for each file.

The **parent VM** has the following responsibilities:

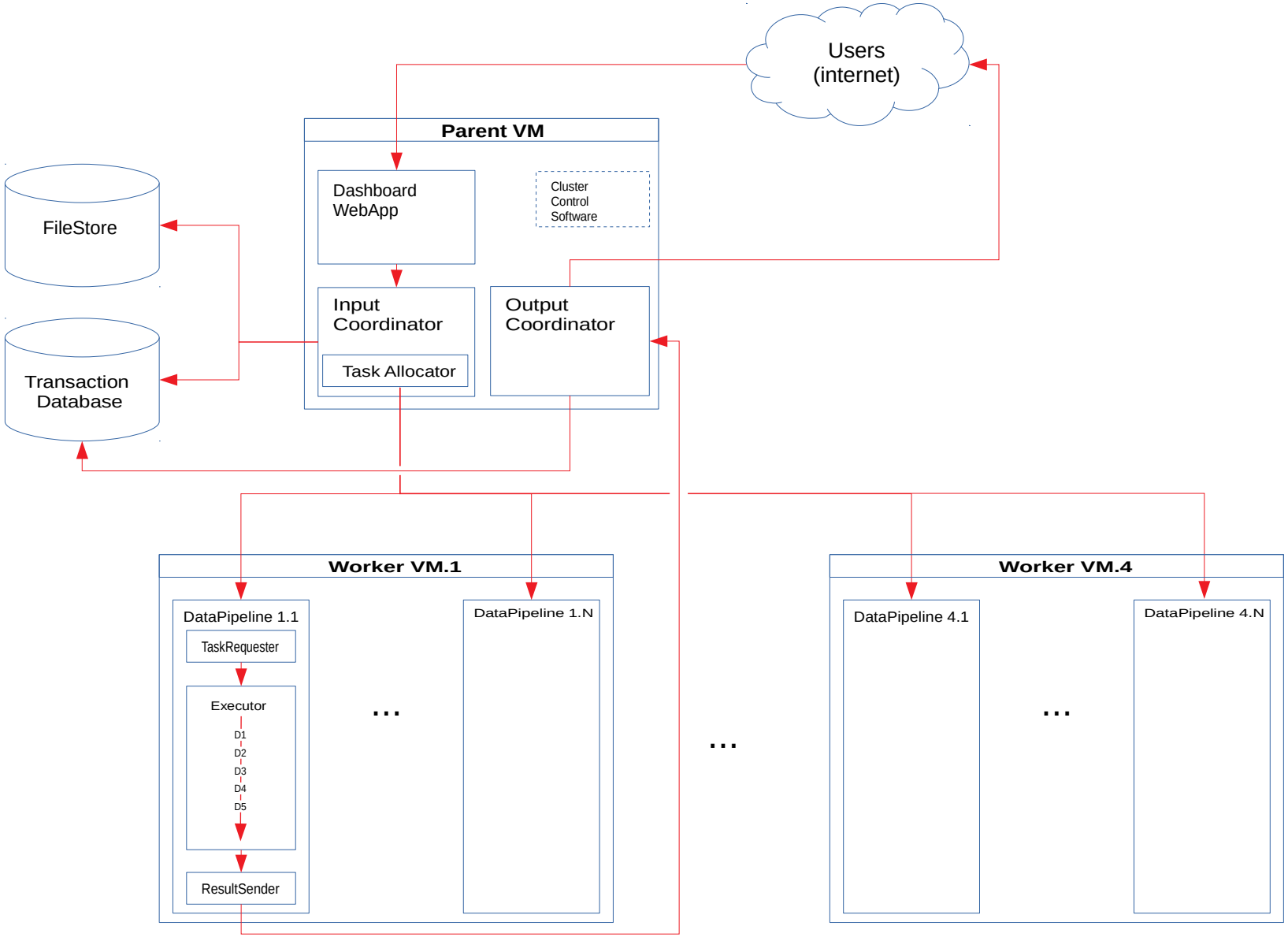
- host the container that provides the web interface to the users to upload their files
- store the uploaded documents to a **FileStore.** At the same time, it will create a *Task* record for the file and record relevant file info and task state in the **Transaction Database**
- dispatch document processing tasks to the DataPipeline containers in the worker nodes
- receive and store the results computed by the workers into the transaction database
- provide notifications to the users on the progress of the document processing tasks (total progress, duplicate document and processing failure emails)

In addition the parent VM will also act as the master of the cluster and host the Kubernetes control plane.

The **worker VMs** will host multiple **DataPipeline** containers that will actually execute the steps of the data pipeline on the documents. The goal here is to parallelize document processing to maximize throughput. These containers have the following responsibilities:

- Request and process document processing tasks from the master node
- Send each document through the 5 steps (D1 to D5) of the data processing pipeline
- Package the results upon completion of the 5 step pipeline and send them to the parent VM for processing and storage
- Handle any failures in data processing by retrying or upon too many failures notify the master node of the failure of the task

General Architecture



The figure above shows the general architecture of the proposed design, and details the major components of the system as well as the high-level data-flows between the components within the system and the outside world.

The system consists of the following major components:

- **Parent Virtual Machine (Parent VM)**: The virtual machine that hosts the containers that coordinate the functions of the components in the worker nodes, as well as the WebApp container that runs the Web Dashboard that users will use to uploading documents. The Parent VM will also be the master node that hosts the Kubernetes control plane that manages the cluster. The parent node will host the following containers:
    - ***Dashboard WebApp***: runs the dashboard web application that the users will access to upload files
    - ***Input Coordinator***: receives the uploaded file from the Dashboard WebApp and performs the following:
        - create a unique *FileID* for the document (by computing the SHA256 hash on the file contents)
        - detect if the document is a duplicate and notify the Output Coordinator if it is so
        - store the file contents in the *FileStore* keyed with *FileID*
        - create a *task entry* in the Transaction database the will be used to keep track and route the document through the system. A task will be uniquely identified by a TaskID and will contain other information needed for routing the file through the system (*FileID*, name, size, type, task state among others)

      The *Input Coordinator* will also have a *Task Allocator* sub-component that dispatches tasks to the DataPipelines that will process them. It does this by listening to requests from DataPipeline components and serving tasks if any are available in a first-in-first-out order.
    - ***Output Coordinator***: receives the computed results for the document from the *DataPipeline* component on the worker VMs, stores the results in the *Transaction Database* using the corresponding transaction ID, as well as handling any errors caused by processing failures in the worker nodes. This includes notifying users of any documents that are duplicates or that have failed to process after the appropriate number of retries.
- **Worker Virtual Machines 1 to 4 (Worker VM.1 to VM.4)**: These virtual machines will contain multiple instances of *DataPipeline* containers which will actually perform the data processing tasks on a document. The goal is to have as many *DataPipeline* containers running on a worker VM to parallelize the processing of the documents and maximize overall throughput.

- The ***DataPipeline*** containers have the following sub-components:
  - ***TaskRequester***: Constantly requests tasks from the Input Coordinator as long as the *Executor* is not currently busy on a task. Upon receiving a task, it retrieves the corresponding file contents from the FileStore and passes it to the *Executor* for processing.
  - ***Executor***: Runs the steps D1 to D5 in succession on the file contents and collects the results. The executor is also responsible for retrying a step (up to 3 times) in case of failure. The results (or failure) are then passed to the *ResultSender*.
  - ***ResultSender***: packages the results (or failure details) with the corresponding taskID and sends them to the *Output Coordinator* on the *Parent VM* for final processing.

Technology Review

The two major cloud technologies that the proposed system will use are **Docker** for software containers and **Kubernetes** for cluster management.

Docker is a platform that allows the packaging of software services (i.e. web servers, ML compute tasks) into a *container* that has all the dependencies needed by the service and be able to run it in a lightweight, standalone execution environment. Multiple containers can be hosted in a single virtual machine, therefore decoupling software services from specific hosts and allowing for the deployment of multiple instances of the same service in a single machine for improved reliability, increased throughput and higher resource utilization. For our purposes, each Docker container can be treated as a separate host, and can communicate (with some configuration and some help from Kubernetes) to other containers using standard web messaging interfaces (HTTP requests).

Using Docker will allow for easier development of the features required by the system as they can be coded and tested in isolation on developer machines with a reasonable guarantee that with the correct deployment configuration they will also work correctly on the cloud. The ability to create multiple instances of the same container will also be essential to allow for the creation of as many *DataPipeline* components as the worker VMs can handle to parallelize the processing of the documents.

Kubernetes allows the setup and management of clusters: a set of nodes (i.e. virtual machines) that host multiple containers. Kubernetes has several features that will significantly affect the ease of implementation and deployment of the proposed system.

The first feature of Kubernetes that is relevant to the design system is its ability to deploy multiple instances of a component over all worker nodes with ease (you only need to specify the component image to use and how many replicas you want to deploy and it can

be done in a single configuration file without having to reference the explicit nodes where the components will be deployed). This relieves a lot the coordination required from the master node when initializing the *DataPipeline* components, and will also allow for changing the number and resource allocations of the deployed containers on the fly to maximize document throughput.

The second relevant feature is Kubernetes' ability to allow components to directly connect to each other without going through the container's node first. Kubernetes' network is designed so a container is identifiable by its assigned IP address *across all nodes*. This virtually allows the system to treat each component as a separate host, using standard web messaging techniques (HTTP requests) to transfer data. This will significantly ease the development of the logic required by the various containers required by the system.

A tertiary feature that will be relevant when the system is deployed in a real environment is Kubnertes' ability to recover from system failures (both node and container crashes). With careful design of the logic in the coordinator and *DataPipeline* containers this will allow the system to recover from such failures without loss of documents. Kubernetes allows for the restarting of failed nodes and containers and with a recovery procedure these can look at the state of the tasks in the Transaction Database and continue execution of any unfinished tasks.

The technologies to be used for the system's main data storage components (*FileStore* and *Transaction Database*) are also worth discussing.

The *FileStore* only requires that the documents' contents be stored in a way that they can be retrieved using a key that uniquely identifies that document. Using a cryptographic hash (such as SHA256) over the document's contents should be sufficient to produce this key. As for the storage technology to use, a cloud BLOB storage that has atomic reads and writes should be sufficient for the purpose, again using the files SHA256 hash as the key.

The use of NoSQL databases for storing file contents has been explored but from preliminary review it seems that the most popular NoSQL solutions for storing large files either have limits that could be exceeded by the expected size of the documents, or have write performance that degrades as document sizes become larger (MongoDB). Hence the decision to use a simple keyed BLOB store.

The *Transaction Database* will use a traditional RDBMS (SQL). This satisfies the requirements of the system for task management and result storage (atomicity and the ability to quickly perform join and filter queries).

Designer's note: These reviews are based on a relatively light overview of the documentation and beginner presentations for Docker and Kubernetes. Due to tight time constraints and limited experience, I have not been able to do proof of concept implementations to properly try out their features (particularly for Kubernetes). There could be hidden caveats in these technologies that would require changes to the design.

Data Stores and Schemas

First we start with a description of the data stores used by the system and the structure of the data stored within. This will provide a foundation for all the processes and dataflows that will occur in the system. The system uses two major data stores: the **FileStore** for storing the raw file content, and the **Transaction Database** for storing file, task and results data.

FileStore

The *FileStore* is a simple BLOB (binary large object) store which has the following properties:

- Allows storage and retrieval of a document's contents based on a *FileID* unique to that document
- Atomicity (all or nothing) of read and write operations

A simple cloud blob storage solution (like Azure blob storage) would satisfy these requirements with the appropriate choice of *FileID*.

The *FileID* itself could simply be the SHA256 hash of the contents of the file. Given the properties of the SHA256 cryptographic function (which turns any size file into a 256-bit binary string with negligible chances (~10^{-60} for identical hash values for 1 billion random documents) it should be a sufficient choice for use as the *FileID*.

Transaction Database

The transaction database is intended to store the following information:

- A record of a document's basic information (*FileID*, name, type, size, owner/uploader among others)
- A record of the task that will process the document (this maps a task to the *FileID* of the document that it is processing as well has having the current status of the task)
- The results of running a document through the pipeline mapped to the document's *FileID*
- A log of the task's status changes and the times when they happened, which will be useful for reporting, diagnostics and error recovery

The rationale for this structure is the creation of a logical *Task* entity that will contain all the state required by the system to process and keep track of a document's progress as it passes through the system. As a high-level object, a task can be thought of as

```
(TaskID, FileID, FileInfo, TaskStatus, <results>)
```

This Task entity contains all the information that the system needs to correctly route documents through the system.

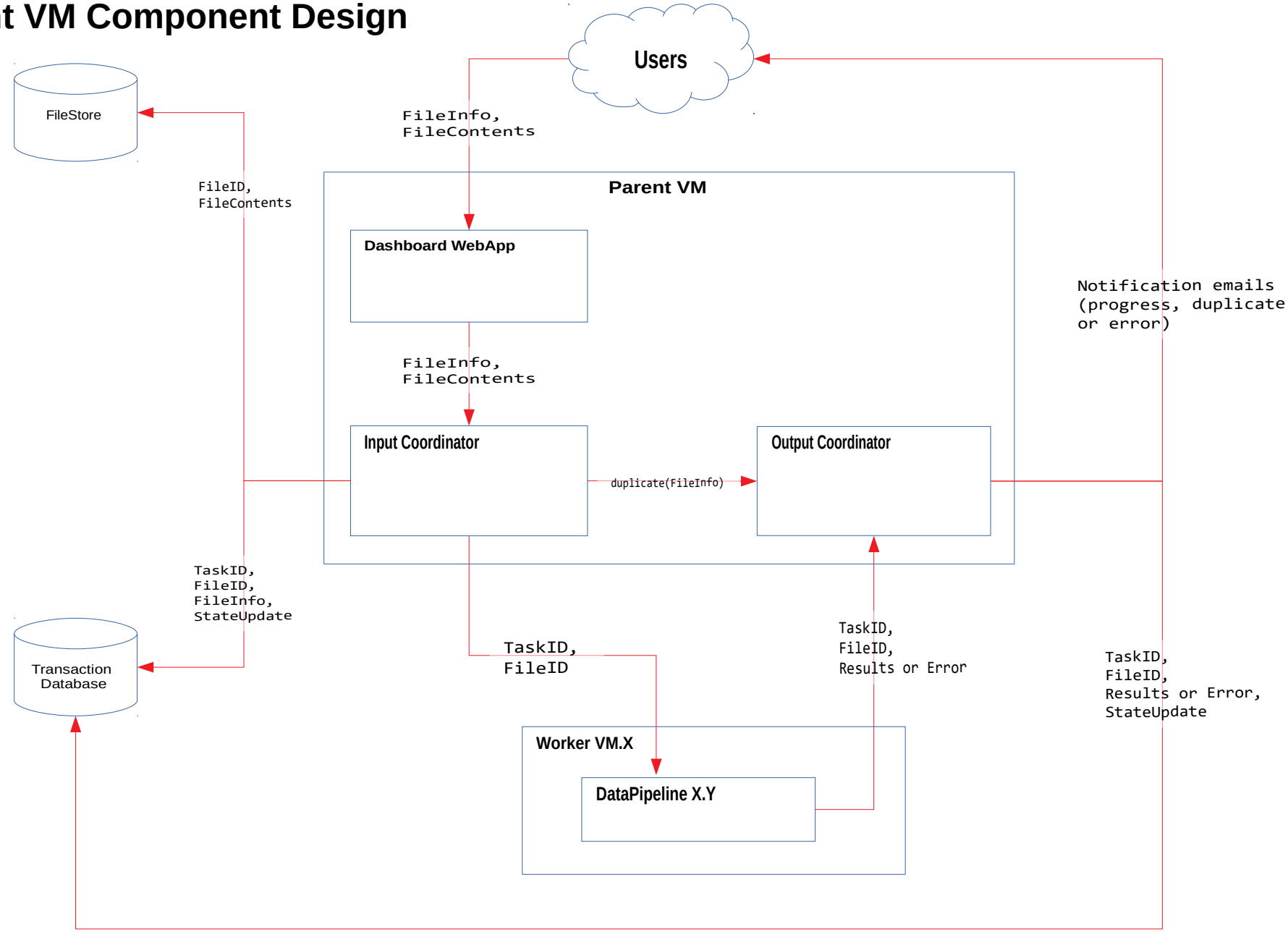
The schema of the database is an implementation of this high level object as relational database tables. A more detailed description of the Transaction Database's schema (the structure of the tables in the database) is shown below in pseudo-SQL format. Comments are in light green.

```
table Task(
     TaskID: int identity
     TaskStatus: string (one of UPLOADED|PROCESSING|DONE|ERROR)
)
table FileInfo(
     TaskID: int (foreign key to Task)
     FileID: string (identical to the FileID used in FileStore)
     Name: string
     Size: numeric
     Type: string
     Owner: string (the user that uploaded the file)
     (other fields as needed)
)
table Results(
     TaskID: int (foreign key to Task)
     FileID: string
     <results>: varies
)
table TaskLog(
     TaskID: int (foreign key to Task)
     StatusChange: string (same values as TaskStatus)
     ChangeTime: timestamp (time when the status change occurred)
     Message: string (any info/diagnostic message attached to the
     status change)
)
```

Note that the `Results` table is just a generalized abstraction for how the results would be stored in the database. It is likely that different types of results will actually be stored in different tables on the database (e.g. table `MapData(TaskID, FileID, latitude, longitude)`).

Continues on next page

Parent VM Component Design



The diagram above shows a more detailed view of the dataflows to and from the components of the Parent VM containers.

Definition of Data Labels

Before going into a detailed description of how the components work, we define the different kinds of data used by the system.

- **FileInfo** : a collective label for all the basic information about a file (name, size, type, owner, etc)
- **FileContents** : raw binary contents of a file
- **FileID** : a key that uniquely identifies a file
- **TaskID** : a key that uniquely identifies a task. This is generated as a new task is written to the database
- **StateUpdate** : a collective label for the changes required to update the TaskStatus and TaskLog entries of a Task to reflect a new state
- **Results** : a collective label for the computed results after the FileContents have been processed by the 5 step data pipeline
- **Error** : the details of an error that occurred when trying to process the data at any point in the system

Parent VM Component Descriptions

The Parent VM has 3 major components: the **Dashboard WebApp**, the **Input Coordinator** and the **Output Coordinator**. The following sections will describe the functionality of each using pseudocode followed by a short section on the rationale for the choices made in their design.

Dashboard WebApp

Pseudocode:

```
while true:
     wait for upload from user
     send FileInfo, FileContents to InputCoordinator
```

From the standpoint of the system, the dashboard app is a simple pass-through that just accept uploads from a user and passes it on to the Input Coordinator.

Input Coordinator

Pseudocode:

```
(UploadHandler: handles uploads from WebApp)
while true:
      wait for FileInfo, FileContents from WebApp
      compute FileID from FileContents
      if FileID already in FileStore:
            send dup_file(FileInfo) event to OutputCoordinator
      else:
            store FileID, FileContents in FileStore
            add Task in DB from FileID, FileContents with state UPLOADED

(TaskAllocator: handles task requests from DataPipeline)
while true:
      wait for TaskRequest from DataPilelines
      if exists a Task with state == UPLOADED:
            find oldest Task with state == UPLOADED
            reply with TaskID, FileID of the Task
            apply a StateUpdate to change the Task to the PROCESSING state
      else:
            reply with No Task Available
```

The input coordinator has 2 sub-components running at the same time: the **`UploadHandler`** which accepts uploads from the WebApp, and the **`TaskAllocator`** which handles TaskRequests from the system's DataPipeline containers. In implementation these will just be HTTP request handlers listening on 2 different ports.

The UploadHandler is pretty straightforward, it just receives the upload, checks if it is a duplicate and if it is not stores the file and updates the Transaction Database with the appropriate information.

The `TaskAllocator` deals with the other side: it listens for TaskRequests from the system's DataPipelines and allocates any Tasks in the UPLOADED state to these requests in a first-in-first-out order.

A pull architecture (DataPipelines sending requests to the Input Coordinator instead of the coordinator pushing tasks out to the pipelines) was chosen to minimize the amount of state that the coordinator has to store, simplifying its implementation while also providing better error recovery in case either the Input Coordinator container or Parent VM crashes. This is also in line with the general principles of the design of Kubernetes, which uses a similar architecture. This should create less framework friction (i.e. trying to do what the framework doesn't want you to do) during implementation.

Output Coordinator

Pseudocode:

```
while true:
      listen to events from InputCoordinator and DataPipelines
      if event == dup_file(FileInfo) from Input Coordinator:
            send duplicate document email to Users
      elseif event == (TaskID, FileID, results) from DataPipeline:
            store (TaskID, FileID, results) in DB
            perform StateUpdate on Task to set state to DONE
            if (number of DONE Tasks % 1000 == 0):
                  send progress email to Users
      elseif event == (TaskID, FileID, error) from DataPipeline:
            perform StateUpdate on Task to set state to ERROR
            send document error email to Users
```

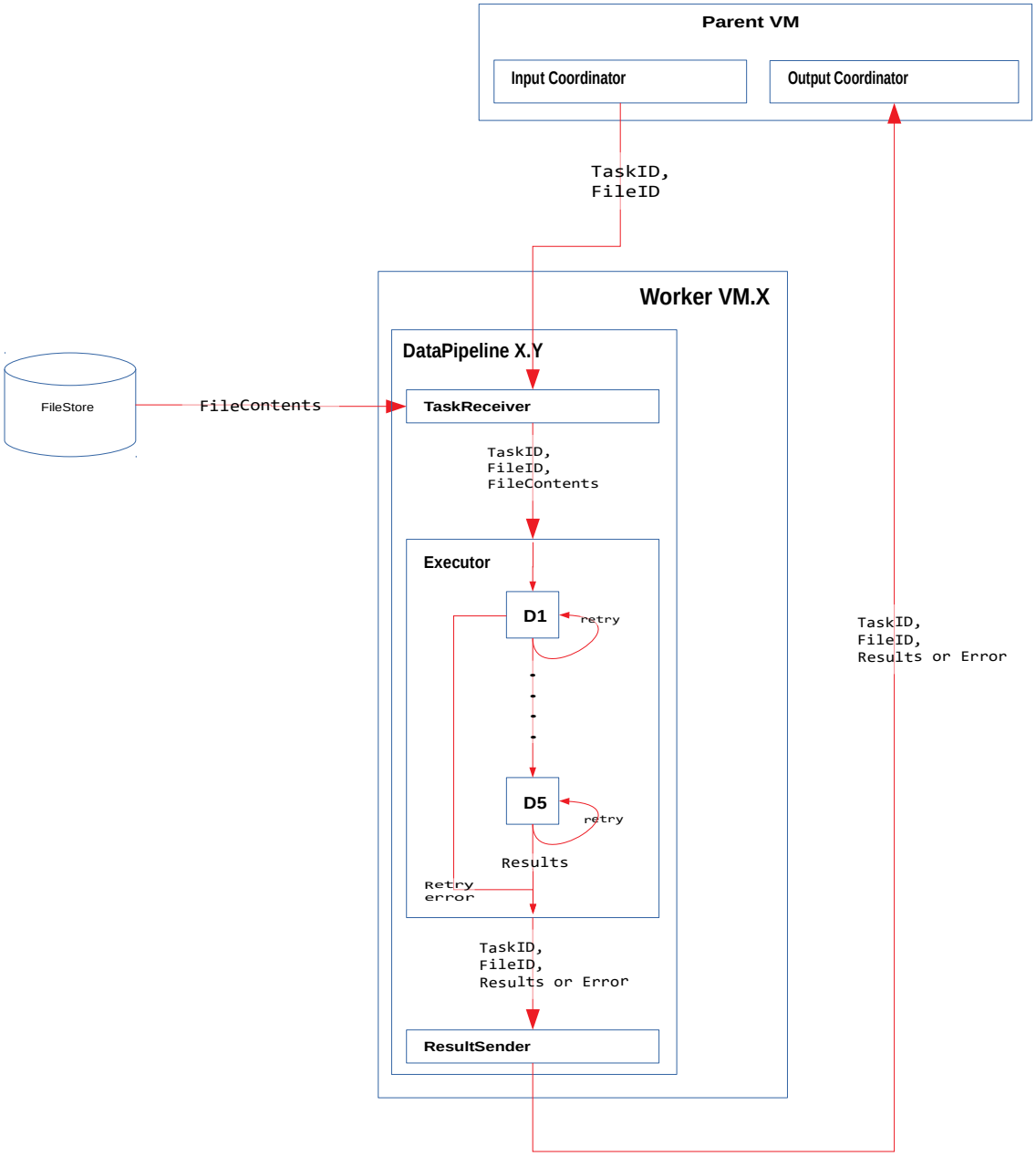
The Output Coordinator is a simple event handler that listens for events from DataPipelines and the Input Coordinator and performs the corresponding action. In implementation this is just an HTTP request handler listening at a given port.

Additional Considerations

Since the Parent VM hosts both the user-facing dashboard webapp and the coordination infrastructure for the system, it is best to overprovision its containers to ensure that there are no resource shortages that could affect either the user experience or bring down coordination tasks that will have system-wide effects. Given the small number of users and an expected low number of messages per second to process, CPU utilization is expected to be pretty low, while memory and intermediate storage use could potentially be high if file sizes get too large.

However, if actual usage shows that there is enough unused resources on the Parent VM, Kubernetes should make it possible to run a few DataPipeline containers on the parent VM to increase utilization and throughput. However, care must be taken to set the appropriate quotas to make sure the coordination and user facing containers have enough resources for smooth operation.

Worker VM Component Design



The diagram above is a more detailed view of the dataflows of the worker VM components. It only has one container type, the **DataPipeline**.

DataPipeline

Pseudocode:

```
(TaskRequester)
while true:
      wait(POLL_INTERVAL) (POLL_INTERVAL is around 30 seconds)
      if Executor is READY:
            send TaskRequest to InputCoordinator
            if Task(TaskID, FileID) received
                  fetch FileContents from FileStore using FileID
                  send (TaskID, FileID, FileContents) to Executor

(Executor)
while true:
      wait for (TaskID, FileID, FileContents) from TaskRequester
      set state to BUSY, clear Errors
      stepInput = fileContents
      for dataStep in [D1...D5]:
            retries = 0
            while retries <= 3:
                  stepResult = dataStep(stepInput)
                  if no error:
                        stepInput = stepResult
                        continue to next dataStep
            set RetryError (maximum retries reached)
            break (get out for loop, maximum retries)
      if no error:
            Results = stepInput
            send (TaskID, FileID, Results) to ResultSender
      else:
            send (TaskID, FileID, RetryError) to ResultSender
      set state to READY

(ResultSender)
while true:
      wait for (TaskID, FileID, Results or Error) from Executor
      if Results:
            send (TaskID, FileID, Results) to OutputCoordinator
      elseif Error:
            send (TaskID, FileID, Error) to OutputCoordinator
```

The DataPipeline has 3 subcomponents: the **TaskRequester** that periodically requests tasks from the InputCoordinator if the Executor is not busy, the **Executor** that runs the data steps on the FileContents, handling retries when needed, and the **ResultSender** that handles the results or errors generated by the **Executor** and sends them to the OutputCoordinator.

In terms of implementation, though all these components can be implemented in a single process (with the TaskRequester being the outer loop that calls Executor and then ResultSender), there is an argument to be made for implementing them as separate processes as the Executor is expected to consume a lot of resources and can cause process-ending faults like Out of Memory errors. If they are run as separate processes, the TaskRequester and ResultSender would still be running in case of Executor failure and take the appropriate action (in this case the ResultSender would send a document error message to the Output Coordinator and then restart the Executor).

Additional System Design Notes

Number of DataPipeline Containers

On initial evaluation, the best number of DataPipeline containers to use in the system seems to be the greatest number that could fit given memory and storage space constraints on the Worker VMs. This would allow for the greatest amount of parallelization of the tasks while making the most use of the resources available. However, there could be other considerations that would argue for a more conservative number of DataPipelines per VM. Communication and coordination overhead cost in Kubernetes could become an issue if there are too many containers running, and limitations like the number of cores and threads supported by the underlying CPU could lead to degraded performance if there are too many containers running at the same time. The specific numbers would have to be determined closer to production when there is a more concrete idea of the loads and resources the system will have.

Fortunately Kubernetes allows for dynamically changing the number of DataPipelines containers in deployment, so this process could be done with relative ease.

Detailed Logging

While the values of the *TaskStatus* field are sufficient for routing Tasks through the system, the addition of more granular status data for the Tasks could be useful for error diagnosis and performance tuning. For example, the PROCESSING state could be further broken down into the individual D1 to D5 steps, while also logging the amount of resources used by the Executor at each stem for a given file.

To implement this, we could add an *ExecutorMonitor* sub-component in the DataPipeline that would be notified by the Executor at appropriate points in the data pipeline. The ExecutorMonitor would then apply the necessary *StateUpdates* to the Transaction Database to reflect the change in the Task's state.

System Failure Recovery

Though the system can already handle failures at the data processing level (errors that happen when executing steps D1 to D5), there has been no description yet of how it may recover from system failures, such as a VM or container crash.

The goal of failure recovery in this case is to produce results that are least surprising to the user. The users expect any unrecoverable errors to be reported to them, and for recoverable errors to be quietly handled and not lead to loss of uploaded documents.

The default behavior for a container or VM crash is to bring an instance of the same entity back up again. That should at least turn the expected services back on. However, the crash itself could have left the system in an inconsistent state that requires some special processing to correctly recover.

To begin recovering from these types of failures, we list the various possible system failures and the error state that they leave the system below:

WebApp Crash

ErrorState A: User can’t upload file

InputCoordinator Crash

ErrorState B: WebApp can’t send (FileInfo, FileContents) to InputCoordinator

ErrorState C: Coordinator updated task to PROCESSING, but failed to send the corresponding TaskID, FileID to the requesting DataPipeline

OutputCoordinator Crash

ErrorState D: Coordinator received results from DataPipeline but was unable to write the results to the DB

ErrorState E: Coordinator received dup_file(FileID) from InputCoordinator but was unable to send notification email

DataPipeline Crash

ErrorState F: Pipeline received (TaskID, FileID) from InputCoordinator but was unable to finish processing the file

The system then needs to specify recovery behavior for each of the ErrorStates.

ErrorState A is immediately known by the user because they would not be able to use the WebApp. This is solved by just restarting the WebApp container which would return access to the dashboard app.

ErrorState B can be detected by the Dashboard WebApp as it is trying to send the file to the InputCoordinator. On failure, it could retry on the expectation that the InputCoordinator will be restarted and if a sufficient number of retry failures are reached it could inform the user that the upload had failed and that they should try again.

ErrorState E is a bit of a special case in that it is not stored in the database or filestore, and therefore the loss of the duplicate file notification would mean that a duplicate file email would not be sent. However, the consequences of this failure would not be as dire because an identical copy of the document would already exist in the system and would produce the results that the user expected. If we wish to properly handle this, a table for email notifications could be added in the Transaction Database to ensure that the duplicate notification email could be resent even if it is lost due to an output coordinator crash.

ErrorStates C,D and *F* are similar in that they have the same inconsistent state: there is a task marked PROCESSING in the database, but there is no actual DataPipeline will process the file and produce results (hence the task will never transition to the DONE or ERROR state without additional action). To recover from this failure, we use the following procedure:

- Periodically scan the Transaction DB to find Tasks that have been in the PROCESSING state for 3x (or more) the expected run time by comparing the latest TaskLog timestamp with the current system time
- Perform a StateUpdate on these tasks to return them to the UPLOADED state (and adding a new TaskLog timestamp to move them to the back of the queue)
- Do task processing as usual

The Task will then again be part of the queue of tasks scheduled for processing and will eventually be assigned to a DataPipeline.

For this recovery procedure to work, any updates to the database concerning Tasks would have to satisfy the following properties:

Atomicity: all updates to the DB for a particular task should be all-or-nothing (no partially stored results)

Idempotence: applying the same DB update for a given Task even multiple times should lead to the same DB state (e.g. updating the DB with the results of the same Task multiple times should not lead to multiple copies of the results occurring in the database)

Atomicity ensures that as long as the data is written to the database, we can be sure that the it is complete and consistent, and idempotence ensures that even if the recovery procedure re-processes a file that has already been processed, it would not create duplicate entries in the database.

Atomicity can be achieved by simply putting all DB updates caused by a Task in a single transaction (BEGIN/COMMIT transaction in SQL), while idempotence can be achieved by checking if the database already has an identical entry for the Task before inserting new data (IF NOT EXISTS ... INSERT in SQL).