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 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 document data and a **Transaction Database** (SQL) for storing file info, computational results and task status for each document.

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

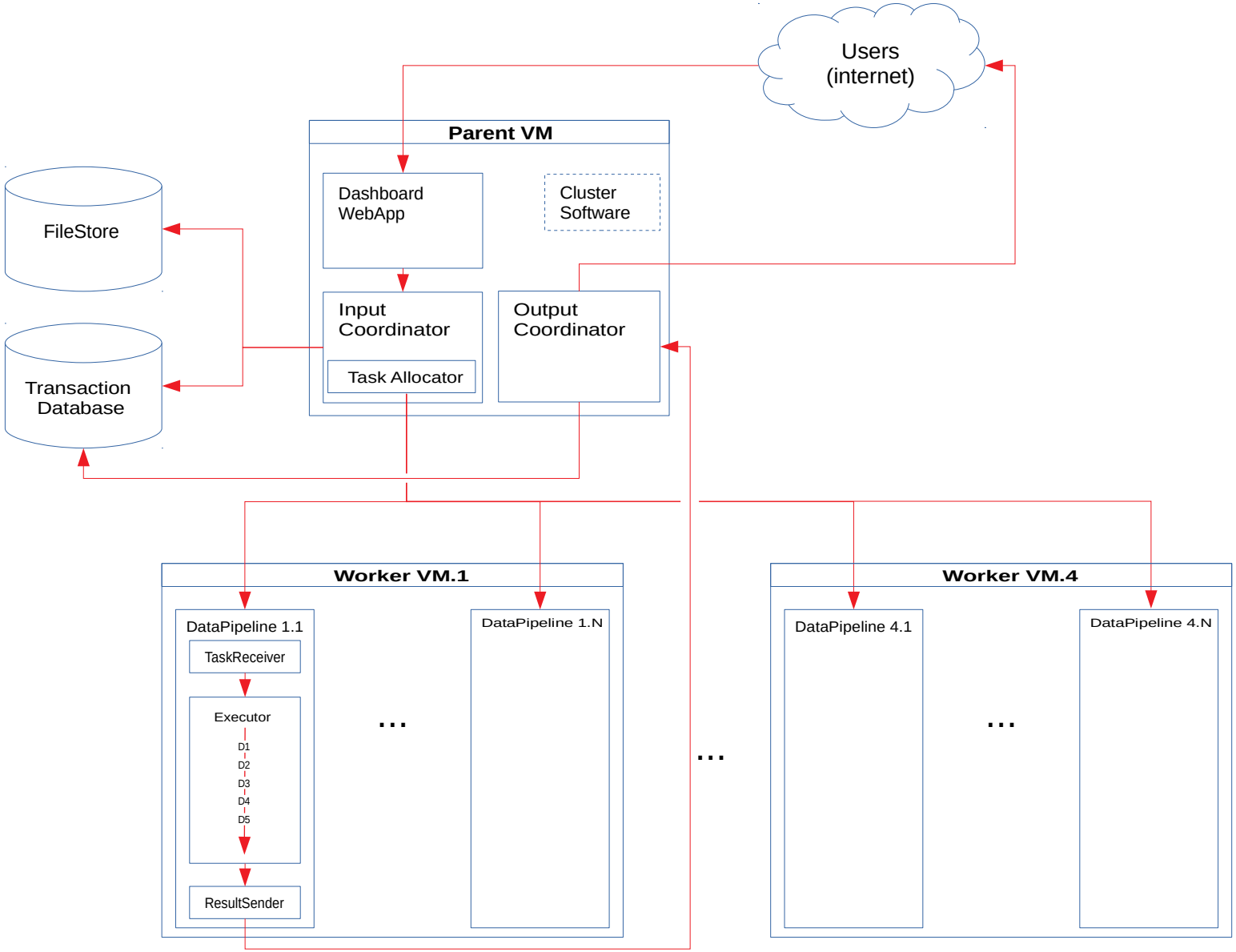
- host the container that provides the web interface to the users to upload their files
- store the uploaded documents to a **FileStore.** At the same time, it will create a *Task* record for the file and record relevant file info and task state in the **Transaction Database**
- package and dispatch 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 nodes** 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 of 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 document contents)
    - detect if the document is a duplicate and send the necessary email notifications to the users while aborting storage of the document data
    - store the document contents in the *FileStore* keyed with *FileID*
    - create a *task entry* in the Transaction database that gives assigns a document a *taskID* and contains basic info on the task (*FileID*, name, size, type, date uploaded, task state).

    The *Input Coordinator* will also have a *Task Allocator* sub-component that packages the document into *tasks* (*taskID* + document contents) and orders them into a queue. It then listens to requests from DataPipeline components and serves tasks if any are available in the queue.
  - ***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 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 overall throughput.
  - The ***DataPipeline*** containers have the following sub-components:
    - *Task Receiver*: Constantly requests tasks from the Input Coordinator as long as the *Executor* is not currently busy on a task. Upon receiving a task, it extracts the document data and passes it to the *Executor* for processing.
    - ***Executor***: Runs the steps D1 to D5 in succession on the document data 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 through the 5 step data pipeline.

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 were 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 (by enforcing idempotence and atomicity of all write operations to the *FileStore* and *Transaction Database*). Kubnertes allows for the restarting of failed nodes and containers and these can just 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 systems 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 document contents has been explored but from preliminary review it seems that the most popular NoSQL solutions for storing large files 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 document 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 ($\sim 10^{-60}$) for identical hash values for 1 billion random documents) it should be a sufficient choice for use as the *FileID*. Alternatively, the *FileID* could be a combination of the document's SHA256 hash and its filename for increased human readability (e.g. `[SHA256Hash]-docname.pdf`).

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>)
```

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 DocumentInfo(
    TaskID: int (foreign key to Task)
    FileID: string (identical to 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)`).