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 (key, value) NoSQL **FileStore** to store the raw document data (MongoDB) and a **Transactional Database** (SQL) for storing file info, computational results and transaction status for each document.

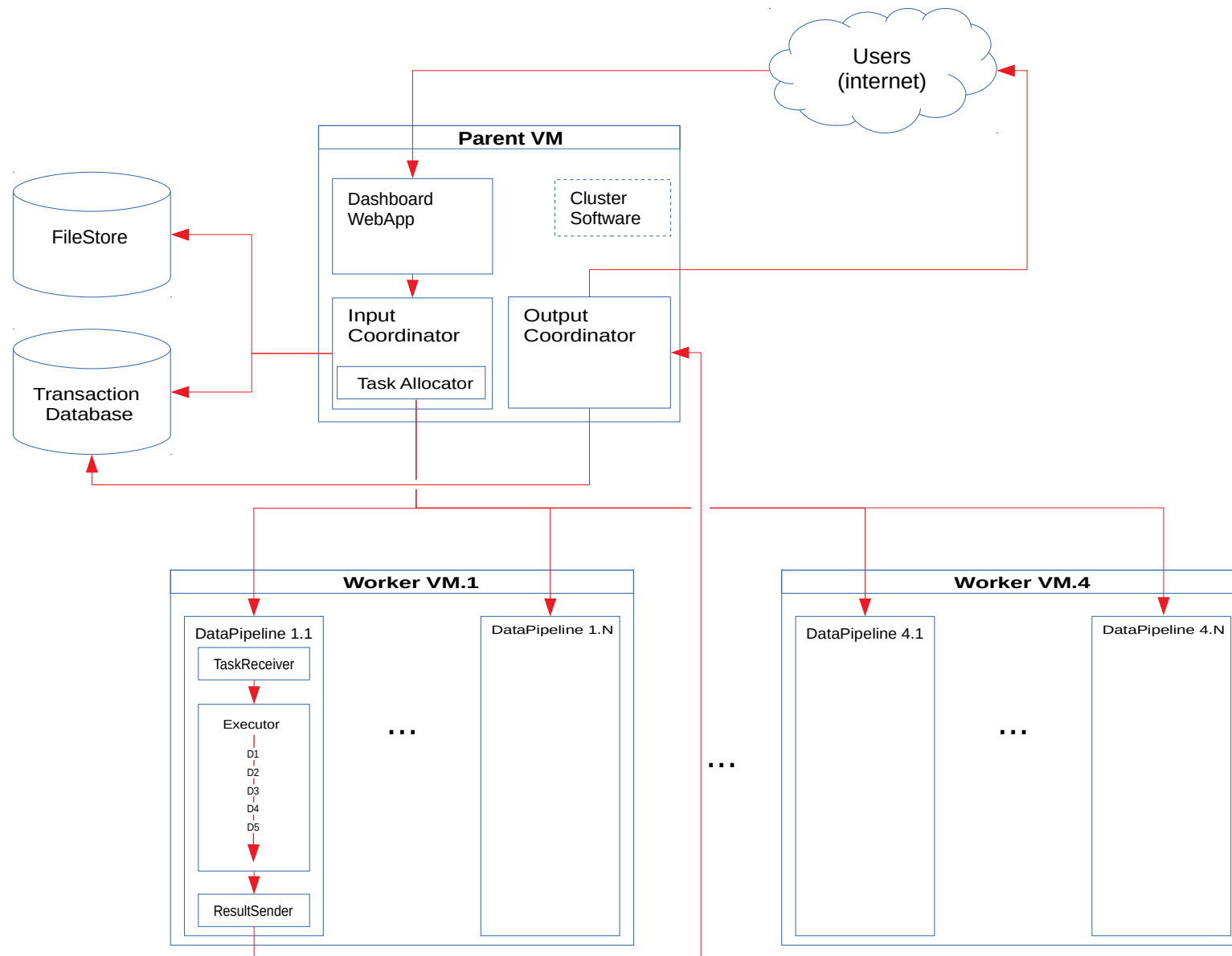The **parent VM** has the following responsibilities:

- host the container that provides the web interface to the users to upload their files

- store the uploaded documents to a **FileStore** and record the corresponding file information 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 task (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

Users (internet)

Parent VM

Dashboard WebApp

Cluster Software

Input Coordinator

Task Allocator

Output Coordinator

FileStore

Transaction Database

Worker VM.1

DataPipeline 1.1

TaskReceiver

Executor

D1
D2
D3
D4
D5

ResultSender

DataPipeline 1.N

. . .

Worker VM.4

DataPipeline 4.1

DataPipeline 4.N

. . .

. . .

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 a document a *taskID* and contains basic info on the file (*FileID*, name, size, type, date uploaded).

    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 *Transactional 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 process.