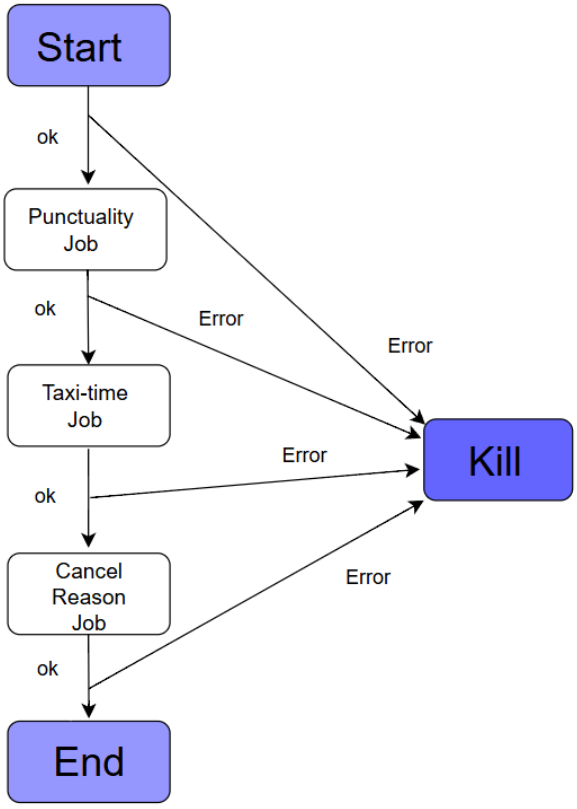
a. A diagram that shows the structure of your Oozie workflow



b. A detailed description of the algorithm you designed to solve each of the problems.

1. To calculate 3 airlines with highest and lowest probability for being on schedule I have used driver, mapper and reducer.

   Driver takes in the input directory containing the csv files. The input directory is specified in job.properties and it is passed into the workflow of this job. It runs the mapper and reducer codes and writes the output in the output directory.
   Mapper function: Mapper function first skips the header, extracts airline code and on time status (which is in minutes) checks if the time is less than 10 minutes, then it sets the value to 1 (on-time), 0 otherwise.
   On time status is calculated using scheduled arrival time and arrival delay.

Mapper function generates a key,value pair of airline code and on time status.
Reducer code: Counts the total number of flights and number of on-time flights. It calculates the on-time probability which is $\frac{onTimeFlights}{totalFlights}$

This probability is saved along with the airline, for sorting later.
Sorting is done such that the highest on-time probability is first, and pick the top three airlines and the bottom three airlines and print the airline along with its probability into the output directory.

2. The 3 airports with the longest and shortest average taxi time per flight (both in and out).

   The driver takes the input directory, output path and number of years (number of csv files it needs to process).
   Mapper: It ignores the headers of the csv files and extracts “origin airport”, “destination airport”, “taxiOut” and “taxiIn” time for each record. There were many “NA” so it skips those records and it skips the records where the origin was the same as destination airports.
   For each valid record, Key is “origin-destination” , Value is total taxi time (taxiOut+taxiIn).
   (There were no valid records from 1987-1994)
   Reducer: It collects the key, value pairs and for each pair it calculates the average taxi-time and prints the pair with the shortest and longest average taxi-time and then sorts airports with average taxi-times and prints top 3 airports with longest and shortest taxi-times.

3. The most common reason for flight cancellations

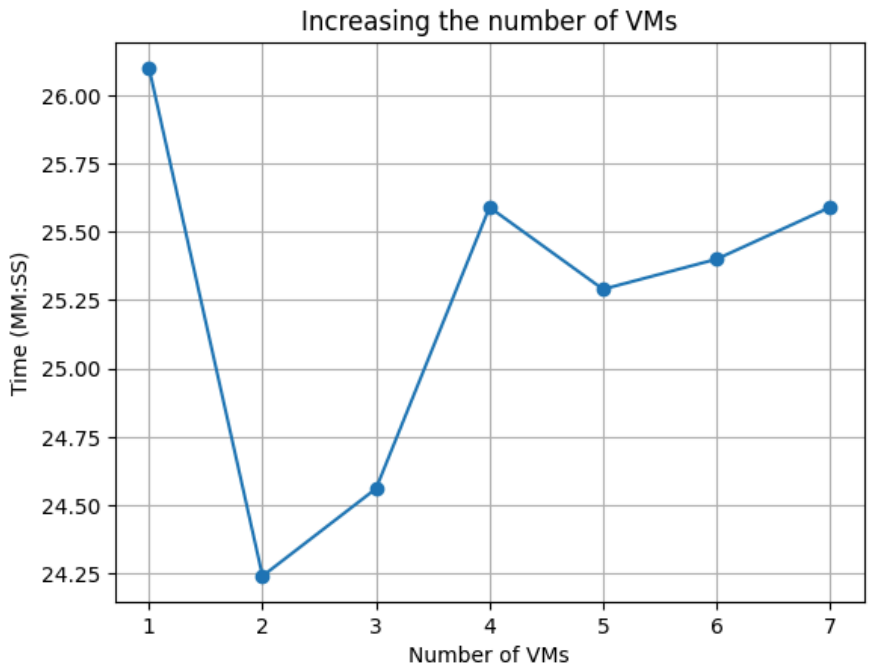
   For this problem I was tasked to find the most common cancellation flight based on input. I split up this task into 3 java files: a Mapper, Reducer, and Driver File.

The Mapper class looks at the csv file and skips the header row by checking if the line contains the value “Year”. It then parses each line by commas and stores it into a fields array. We then check this fields if it has more than 22 values in it to know that the cancellation column exists. From here we get the if the flight was cancelled ann the code/reason for cancellation from columns 21 and 22 respectively. Finally we check if the cancel value is “1” if true that means the flight was cancelled, and then we check if the cancellation reason is a valid code and if both conditions are true we emit a count 1 for every valid cancellation code when a flight is marked as cancelled.

The Reducer class, for each unique key, sums all its counts and stores them into a Map called counts. After all reduce() is done for all incoming data, it then automatically calls the close method, which first checks if the counts map is empty, if true it outputs no reasons given. Otherwise it finds the top 2 most common reasons for cancellations and then outputs the top reason for cancellation if it is not NA, if the top cancellation reason is NA then it outputs the second most common reason for cancellation.

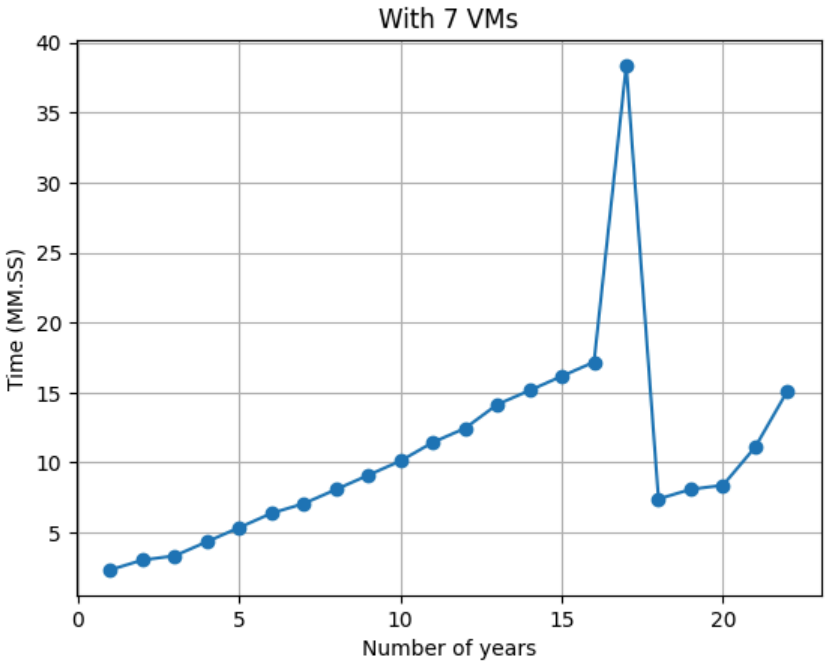
Finally we get to the Driver that has the job to configure the job which it does by limiting the number of input files, takes in dynamic input and output directories and launches map reduce jobs with configurations. Expanding on this abstract, we see that our logic reads input/output and optional file limit from configuration and uses FileSystem.listStatus() to get input files and limit them by user or workflow defined limit. The logic then sets all relevant classes and all relevant JobConf. Finally it submits the job to the JobClient using the method .runJob().

c. A performance measurement plot that compares the workflow execution time in response to an increasing number of VMs used for processing the entire data set (22 years) and an in-depth discussion on the observed performance comparison results.



Shown above we see that at first with one VM we get a time that is greater than all other numbers of VMs. This is expected behavior for one VM because all work is done in a singular node. Once we add another VM making the total count of VMs 2, we notice a drastic change in running time of the workflow of about two minutes. We would expect as the number of VMs increased that our time would follow suit, but we noticed that this wasn't the case.The more we increased our VMs, we noticed our time to complete workflow gradually increased and back to a similar time as running on one VM. This was a peculiar behavior observed but was later explained by closely examining the overhead and extra management that more VMs introduced when running the workflow. As more VMs were added the data sharing and processing became slower as more communication was needed among nodes, as well as some nodes not being utilized at all. Clearly, we see that adding VMs speeds up processing, on the other hand, we cannot deny the fact that at a certain point in this power increase we start to see diminishing returns as shown in the graph.

d. A performance measurement plot that compares the workflow execution time in response to an increasing data size (from 1 year to 22 years) and an in-depth discussion on the observed performance comparison results.



We noticed that as expected the more data we fed into the seven VMs, our time to complete workflow increased. Increased times is something to be expected but as shown in the above there is an irregularity in the graph at around when 16 years of data are fed into the workflow. At first glance one might become confused at this substantial increase in time to completion. However, this is explained not only by the different specifications of one's system but also by how consecutively someone runs these workflows on 1 through 22 years of data. To expand on this idea, we noticed that once a machine had run the workflow for an x amount of years our timing to execute such workflows would increase. To combat this increase in time we would either clear caches and reboot our instances completely to create a more than fair environment to execute all workflow years. Even with all the precautions, the instance environment is an unpredictable one, and anomalies occur as observed in the graph at around year 16. This would also explain the differences in times from running all years at once on different numbers of VMs and

running all years on 7 VMs. Unexpected things occur such as environments being overloaded with RAM taken up, memory issues, or even connectivity issues, all causing unstable environments.