|  |  |
| --- | --- |
| Acquisition  There are multiple data sources to acquire data from; .csv, .xml, .json. To handle these scenarios I would build out a common interface from which I would build classes to inherit from and override the abstract methods. These classes would be built out with the specific processing logic to handle loading the specific data format. These classes would instantiate my data access objects. I would also build an interface to create an uniform data transfer object. The classes which inherit from this interface and override the abstract methods with specific logic to handle transforming the different formats of loaded data. These objects would output a uniform format such as a data frame to be transferred to the processing step. To handle instantiating the correct data access object, I would build some form of configuration to specify whether to instantiate the json DAO, csv DAO, etc. From the DAOs, they would call the corresponding DTO classes. How this configuration takes form would be dependent upon the architectural decisions such as passing configuration through an SNS message, or building internal configuration to call correct class based upon the extension. In a production scenario, I would research proven best practice for configuration solutions based upon architectural choices. Please see flow diagram section one for acquisition overview. |  |
| Processing  In this step, I am focused on building out the structure of the data to be ingested into storage. The structure of this data lends itself well to be ingested into a RDBS. Making use of an ORM to create an independent solution to ingesting the data into various databases would be the best choice in this scenario. This would involve building out data models to tie the processed data to the corresponding attribute fields to load into the database of our choosing. The output of this section would be a collection of the data models to be ingested. A crucial part to processing this data would be to apply transformation logic to it. In this case, to prepare the data to be ingested into a RDBS, you have to think about cleaning the data to ensure the records follow a uniform format. For example, I would handle common clerical errors such as stripping leading and trailing white spaces, handling for word separation, ensuring that all characters are of same case. Depending upon unique keys, you would have to handle duplicate records and delete all duplicate records if use case calls for it. After handling all common scenarios, depending upon requirements of story I would apply custom transformation logic to records. Once, the cleaning and transformation is finished, the data models are ready to be built and passed on to the storage section. See section 2 of flow diagram |  |
| Storage  Depending on the scale of the data, and team resources I would make a choice between Amazon RDS database options. In this scenario, with data volume ranging from low to medium I would tend to go towards Aurora. Aurora is a low cost, fully managed, and auto-scaling up to 128 TB database solution. We wouldn’t need to worry about adjusting our provisions should the data volume expand beyond our expected capacity. Furthermore, aurora is fault tolerant continuously backing up data to an s3 bucket to recover should a failure occur. At this point in the solution, we are ready to make a connection to the database and being the process of loading our data to aurora db. See section 3 of flow diagram. |  |

The pipeline is built into lambda functions, allowing the solution to scale should we receive multiple files to process. This also helps ensure that should a file fail processing, it does not compromise the system. Should a file fail, other files can continue processing and an error log should populate within Cloud watch. This allows the team to review the reason for the error and recover by finding root cause.