Data Lake Architecture -

A Comprehensive Design Document

Medical Data Processing Company

# Tracker

## Revision, Sign off Sheet and Key Contacts

## Change Record

| Date | Author | Version | Change Reference |
| --- | --- | --- | --- |
| 01/26/2022  02/09/2022 | Joseph Corrado  Joseph Corrado | 0.1  0.2 | Initial draft  First Revision |

## Reviewers / Approval

| Name | Version Approved | Position | Date |
| --- | --- | --- | --- |
| FirstName LastName | 1.0 | Udacity Reviewer  Enterprise Data Lake Architect |  |

## Key Contacts

| Name | Role | Team | email |
| --- | --- | --- | --- |
| Joseph Corrado | Data Architect | Medical Data Processing | student@email.com |

# Purpose

The purpose of this document is to explain and illustrate the components of my design for the data architecture for the Medical Data Processing Company. In this document, there is a deep dive in the technical implementation of the various components that make up the design. In addition, there is discussion into the rationale behind the decisions and assumptions that were made. Finally, an illustration of the design is included for clarity.

I am creating this document so that the concepts and key architectural decisions can be more easily communicated to the key stake holders of the business. The target audience of this document are the more technically oriented architects, software developers and tech leads at the company. The items that are in scope for this document are the various components of my architectural design, arguments defending those decisions, as well as explanations of the requirements that are driving those decisions. Some elements that are out of scope for this document are the frameworks we use for the analytics/dashboard layers, the implementation of the data warehouses for the analytics layer, and the application code for the applications supporting our clients.

# Requirements

The problem is that the previous data architecture resulted in situations where the components of the system failed due to lack of capacity and a higher influx of data.

In the current design, the relational database system backs all of the ETL, analytics and application systems. This represents a major bottleneck in terms of capacity, availability, and resiliency.

We’d like to create a design that has a few desirable properties. Firstly, we have a situation where an increase in load on the ETL process caused our database system to go down which, since many other systems depended on it, caused many other systems to go down for quite some time. Even after that, recovering the system was arduous and took a long time. This is an example of how the previous system was not resilient nor available.

In addition, we have some requirements in terms of the use cases we have to support. We must support building machine learning models, building near-real time dashboards and the generation of various reporting scripts.

In the existing technical environment, we essentially have databases and application servers. To improve our design, we will introduce a few components that decouple, and take some of the strain off these systems.

Another requirement is that we must support all manner of file types, in addition to compressed files. We have to support CSV, TXT and XML files, and those files can be zip or gzip compressed.

Through the use of a data lake, we are able to support all of the required file types and compression algorithms. Since we’ll be utilizing an event-driven design for our processing, we’ll be able to alter our processing, based on the file type and compression used.

The current data volume, according to the technical statement, is 77000 zip files per day, at approximately 1 MB per zip file, which means 77 GB per day in zip files.

This represents the initial ingestion requirement of our system. But, since we’re using S3, which can store an unlimited amount of data, with the constraint of 5 TB per object, we satisfy the initial ingestion requirement.

I found these requirements from the company profile, with the statements of current problems, the current deployment, and then the few use cases that need to be supported by this data system.

# Data Lake Architecture design principles

Some design principles for data lake architecture:

* store all of the data in its initial form
* store every version of the data in each step of ETL in separate locations
* store data files that are consistent and in the same format in the same folder
* correspond a folder to a table for data files that will be ingested into another system
* make sure to set lifecycle policies on files that either need to be kept for regulatory reasons for a specific length of time
* make sure to document all data sets on the data lake
* make sure to set access policies on the various data sets to follow the path of least privilege

We need to store the data in its initial form, as well as the data after each step of our transformations so that we can have visibility into what’s happening at each step, and so that we can retry any processing that doesn’t pass data validation.

Many systems that integrate closely with data lake infrastructure have the capability to recursively load files that are located under a particular logical folder in the data lake. Because of this, it is most convenient to place files that have the same structure and can be considered to be part of the same “table” in the same folder. Then, your load statements usually simplify to just pointing at that logical folder.

For medical data, there are the HIPAA guidelines, and even more stringent requirements on the way in which you store and access ePHI (electronic protected health information). To be legally compliant, it is necessary to make sure that the appropriate life cycle policies and access policies are set on specific data sets containing ePHI.

Finally, it’s very important to make sure to document the processes, and catalog the data sets that are present in the data lake. This is conducive to creating a self service workflow with the data in the data lake, and any potential developers or applications who’d like to consume that data.

# Assumptions

Some assumptions I’ve made are that we’d prefer to use an event driven architecture over a batch architecture, and that we will require application databases to back the applications that our clients use.

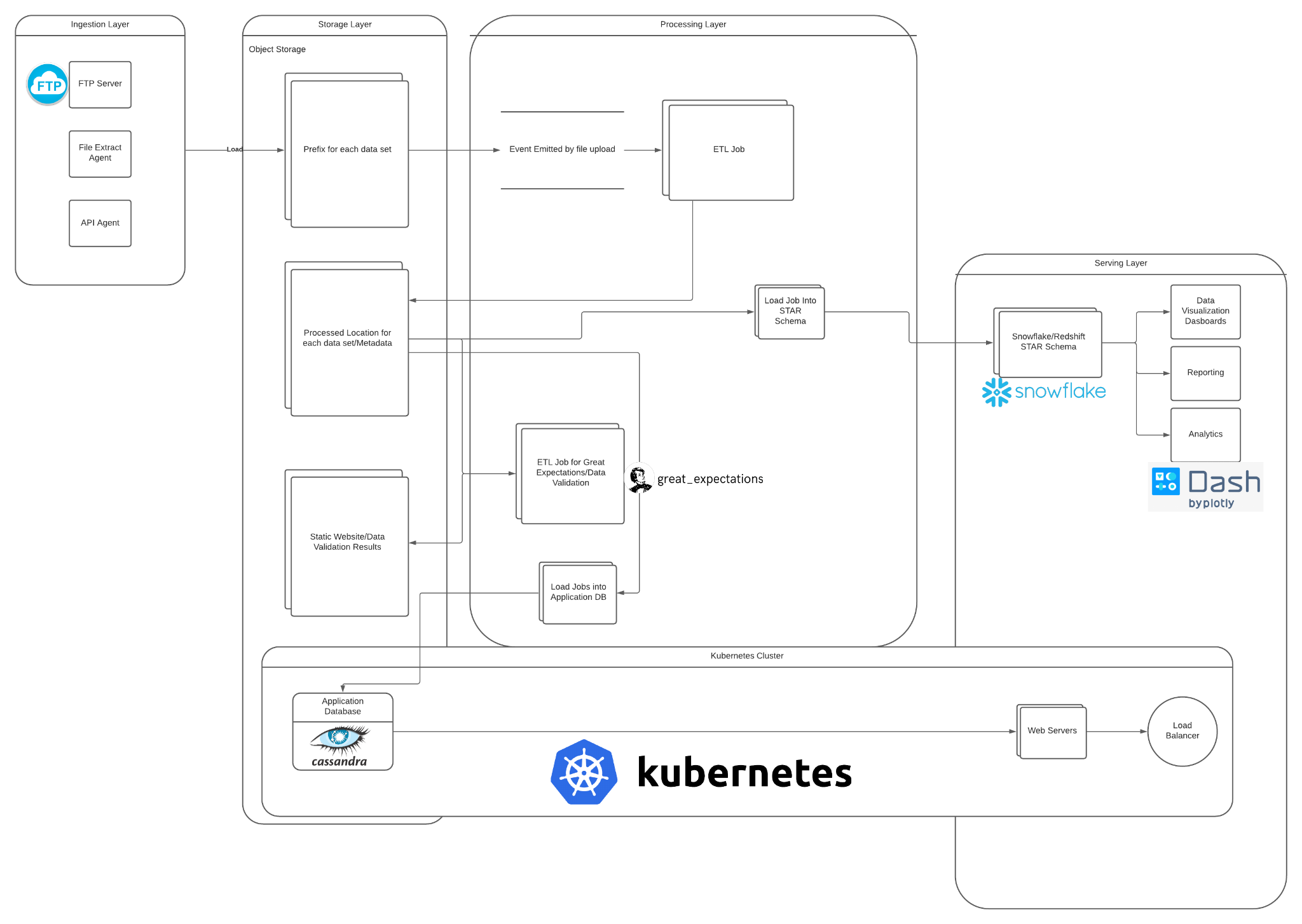
Some questions that I had while designing the architecture was what kind of cloud expertise is present on the tech teams. There are many services on AWS and GCP that can be bespoke, but offer great returns if utilized correctly. Not to mention that many of those services integrate natively with our data lake architecture. For our applications and various analytic use cases, using some of these nuanced services can make our lives much easier, and drastically reduce development time.

In the problem statement, there isn’t any information about the general requirements around data quality as it’s coming in. In addition, there isn’t information as to the delimiters of the flat files that come in. I know the problem statement says CSV, but in my experience a lot of times other kinds of delimiters are used, or FWF files are used in the health sector. If there are FWF files coming in, then we need to do discovery and create more information about the schemas of the incoming data files. Also, it is possible that we have pipe delimited files, instead of comma delimited flat files coming in. Because of our reliance on a data lake, we don’t actually care about any of these considerations, since we can save any object down wholesale at the outset.

I think that along with a data lake and event driven architecture, we’re increasing the level of operational risk in order to meet a greater number of complex business requirements.

Because of the flexibility afforded by this kind of design and architecture, we have to do pay more care to what different parties can access, what metadata we store, and how we handle logs. Because of the increased reliance on metadata, logs, and the overall complexity of the design, we are increasing operational risk, as it might become more difficult to debug the architecture, and we require a higher level of competency from our technical staff.

# Data Lake Architecture for Medical Data Processing Company



# Design Considerations and Rationale

## Ingestion Layer

In terms of ingestion, there are some constraints in terms of how the data is initially exposed to our systems. Because of our decision to use a data lake, we can leave all of the ingestion systems we had previously (FTP, File Extract Agent, API Agent), and we can simply redirect the load of that data to files in our data lake. The ability to load data into a common area without having to care about the data format is significant, because it means we can choose how to organize our data in a uniform manner across all of our data sources.

So, whether the data is of different formats, or zipped, doesn’t really matter to our solution, because we’re using object storage in our cloud to back the data lake.

For data coming from databases, FTP servers or APIs, we could just use an ETL job with the necessary connections enabled. If there was a custom setup that needed special attention, we could just use a function as a service offering to execute an arbitrary bit of code.

Since we’re using a data lake technology, we can typically leverage message-passing services to enable an event driven workflow with our compute services.

## Storage Layer

To store all of our data, we will simply utilize an infinitely scaling, highly available, and highly reliable object store. This will cause there to be no issues about storing the data.

Then, for the analytical use cases, we’ll use a massively parallel data warehouse that is designed to store and operate on petabytes of data. Finally, for serving our application, we’ll use dedicated application databases that only store the recent data necessary for the applications to function.

For the year over year data growth rate, there would be no change to any of the components, since each component would simply scale serverlessly to meet the change in data velocity. The data lake would see no difference due to the increase in data, since we would just create new logical folders in the data lake. The data warehouse would see no difference since modern data warehouses are designed to scale out automatically to support more historical data being stored. Finally, the only question mark is the application databases. But, since we’ll have an architecture where we keep only the necessary data on the databases, then move that data out into the data warehouse, and clean out the database, we’ll always have a fresh database that isn’t running at capacity. In this way, there’ll be low load on the databases, and it’ll be much harder to take them down. In addition, since we’ll be running in a cloud environment, our database instances will also be much more robust, due to the able to make automated backups, read replicas, and multi-availability zone deployments.

For back up and recovery, we’ll simply use the high availability deployments of each component, and we’ll continuously back up the data from each system in multiple regions. Even though modern data lake stores have high reliability, for disaster recovery, we can also use cross-region replication, and back up our data in multiple geographic regions.

For the databases, we have automated back up and serverless recovery options afforded to us by the use of fully managed database services. On my diagram, you’ll see I’ve noted that we’ll be serving our applications and application databases from a self-healing, fully managed kubernetes cluster. Utilizing kubernetes will allow our application to maintain 100% uptime, and it will allow us to leverage the self-healing properties of kubernetes. When one application goes down, containerization technology can be used to spin up new infrastructure with another copy of our application on it. Because we’ve been using automated backups and a data lake to store all of our data, we can simply spin new database instances up and down, restoring from those images.

For metadata information, we’ll also store that in our data lake, but in a different logical folder. One option is to create a static website that holds the information on metadata, and data validation that is hosted by the data lake. The great expectations section listed on my diagram is a framework that uses python (or spark) to create “expectations” that are executed against files in a data store, databases, or data warehouses. When those expectations are run, they are then compiled into a static website that can be served from a website that’s backed by and integrated with our data lake.

The metadata should hold information about the schemas, data types, data source information, data size and frequency of load.

To secure the data, we’ll use role-based access and access control lists to manage the access to data in the data lake. In addition, we’ll make sure to always use functional users/roles when performing ETL/data loading or any other access by our application.

This will help us to enforce the path of least privilege, and it will keep humans away from touching and changing data in a way that’s not scalable.

Many of the tools I’ve suggested in my design are 3rd party/open source tools, but I’ve designed the architecture such that those tools all utilize the infrastructure of one of the big 3 cloud providers. I do this so that we can utilize the good points of those tools, but not have to worry about infrastructural concerns, and we can leverage the economies of scale and expertise of major cloud providers. For example, even though Kubernetes is open source, I use it in my design, insofar as we use a managed version of the service to ensure uptime for our application. Kubernetes is inherently elastic and will take care of load balancing for us.

In addition, I’ve also suggested the use of great expectations. This is a relatively new framework that I think solves a problem that every major data company has to solve one way or another. It allows us to essentially “test” our data - we assert what “ought” to be true about the data, and run computations to check if it is indeed true. Furthermore, you can leverage both python or spark to run the checks, so we can run these checks on the entirety of our data without having to worry about scale. We can leverage a fully managed spark cluster to run great expectations on the data in our data lake. Those results are then compiled into a static website, and hosted in the same environment.

## Processing Layer

There are multiple options for processing our data at each step of the way. In the cloud, ETL tools are typically compatible with a variety of compute frameworks, such as Python and Spark.

We can satisfy different processing needs by using different kinds of components where needed. For example, we can handle batch transformations by utilizing ETL jobs or distributed compute cluster jobs on a schedule. For realtime processing, we can stick a Kafka cluster inside our diagram to decouple the data producers from the consumers that would handle and process real time data. Many public cloud providers are offering managed Kafka cluster services, which we can use and integrate with our data lake. In the future, the utilization of a Kafka cluster would be extremely valuable as a message broker between the real time components of our infrastructure, and the analytical consumers.

For ad hoc querying capabilities, we can leverage one of several frameworks, given that our data is processed, and sitting in a flat file format in our data lake. If analysts need to run ad-hoc queries, then the concept of using a data warehouse with a dimensional schema also satisfies that requirement. When you perform the dimensional modeling needed for a STAR schema on a data warehouse, those designs typically satisfy any ad hoc querying requirements, and supports all manner of user side queries.

For processing, we’re several different frameworks. The important part here is that we’re using an architecture that permits flexibility in using different processing frameworks, depending on the requirements of the task.

In terms of processing services with our data lake, we’ll typically expect there to be a way to run scheduled Python jobs on elastically scaling hardware, function as a service jobs, and distributed compute jobs. In addition, these frameworks should be compatible with event driven message passing components, as well as running on a schedule.

In terms of consideration for 3rd party tools that were **excluded**, there are not many. This is because my architecture is flexible enough to permit many different tools to be utilized for processing purposes. Most cloud providers will allow you to, at the very worst, use your own deployment of whatever processing frameworks you want to use. For the major cloud providers, you can seamlessly provision distributed compute clusters, on which you can install the major distributed compute frameworks, such as Hive, Spark, Hadoop MapReduce, Pig, etc.

All of these components can be run in an event-driven manner, on a schedule, or after batching several events together. Since we’re using a much more modular architecture, thanks to our usage of the cloud, we have a lot more flexibility to run individual processing jobs on as small of a resolution as we desire. You can utilize almost any 3rd party tools you want, since all major cloud providers provide infrastructure and platform as a service tools. In the worst case scenario, you could manage deployments of these tools yourself.

My design attempts to satisfy all of our computing and scalability constraints, while not consigning us to one tool or another, but rather a whole host of options at once. Because every component is open to several flavors, we can have dynamic choices be made about what platform is used, based on the incoming data, or the transformations to be applied. If you utilize functions as a service technology, and common open source cloud provider CLI’s, you can change the infrastructure or platforms used dynamically to respond to changing data requirements.

## Serving Layer

The serving layer is made up of all of the frameworks that are used to pull data from the processed locations of our design, and then serve those results to clients. This consists of both our analytical serving layer, in which we use a data warehouse with a dimensional model to allow analysts to generate ad hoc reports, as well as our application layer, where users can authenticate themselves and retrieve data from our operational data stores.

We will store our day to day operational data in the application databases, and then we’ll push that data into our analytics platforms and data warehouse. This will result in many tables in both of these systems, as we have to support all of our datasets. In addition, we will probably have some schedule of loading and cleaning out the data that resides on the application data bases, as they’re not designed to hold so much data at once. Only the data that is required by the applications should be held on those databases.

The data in the serving layer could be used by our clients to visualize their data, and it could also be used for business intelligence. Not only would we be able to support the use case of supporting our applications, but we’ll also be able to run statistical studies of the processes that underlie the data creation process. We can use this information to change the way we handle the data, or think of new novel use cases for the data we have.

# 8. Conclusion

In this technical document, I outlined the proposal to create an event driven architecture, centered around a data lake. Not only will this design facilitate all of our use cases and business requirements, but it will also open up the path to new projects in the future.

The next steps are to work with senior cloud architects and IT personnel to figure out which cloud provider to move forward with on this initiative. Then, we can list the set of services we’ll need to be whitelisted and approved for use in our environment.