| iMBP – Architecture & Design |
| --- |

Charles Cai

### Objective

* Build a platform in distributed microserives architecture
* Build an application that is dynamically configurable
* Build a platform that is expandable and scalable
* Build an application with asynchronous requests and fault-tolerance

### **i**MBP Platform Components

* API gateway
* Domain & Microservice apps
* Service Discovery & Configuration system
* Logging system
* Monitoring system
* Data storage units

Architecture Diagram

Admin

C\*

UI

Redis

MongoDb

Redis

C\*

Redis

Prometheus

Kibana

Consul

Consul

Consul

Config &

Discovery system

Kafka

SEC log

SEC log

Routing

Routing

UI/Admin

Consul Agent

ETL

Consul Agent

Routing

Consul Agent

Retrieval

Monitor

system

API Gateway

LB

Consul Agent

Logging

system

Gafana

Elastic

Search

Log

Stach

| iMBP – Architecture & Design |
| --- |

| iMBP – Architecture & Design |
| --- |

**API Gateway**

* Direct client requests to corresponding domains, projects and microservices, the primary domains for iMBP has ETL, Retrieval(RT), UI(may need to break down to be more specific), Admin and more as the project moves on. Each URL must include domain (ETL, RT) and project (AOI, SMT) to ensure distinguishes of the requests’ objective.
* Load balance to underline application servers dynamically based on service discovery and availability.
* Pan out and manage aggregated service calls if it’s necessary.
* Manage authentication and authorization requests.

**Domain & Microservice**

* Each domain and project should have its own dedicated microservice server group to accommodate or tailor to its own specific needs or requirements. In following section, will focus **only** on ETL domain process.
* ETL
  + API gateway is application’s first entry point of all domains, each ETL request is load balanced through round robin to asynchronously send to downstream ETL servers. Upon receiving the request, the ETL server checks the cached system metadata, then route to corresponding handler to save the data asynchronously to its designated storage unit Cassandra.

| iMBP – Architecture & Design |
| --- |

* + Each server should have approximated same load which requires overall requests is proximate same payload on average is the application’s first line of defense to prevent server hotspot/overload.
  + Any image has size over the 1M bytes should not send to server without compression at origin to avoid network bandwidth and storage space costs. If compressed data still remain over M bytes footprint, a separated url and servers will be dedicated to process the data that is our second line of defense to avoid server hotspot. (If iMBP needs to do compression, a POC needs to search best compression technology. If none found, will think of other way).
  + Any request passing over predefined process time will be cut off to avoid resource occupation and server hanging that is our third line of defense against server hotspot. The payload of short circuited request will be committed to SEC log (**S**hort circuit and **E**xception **C**ommit log) in disk for future process, second copy will send to logging system for backup or inspection, same to any exception generated or failure during ETL process, will utilize two-copy paradigm method. A configurable SEC log size threshold should be setup to alert administrator for inspection and an action follows.
  + Actions on SEC log will be handled through Admin UI. Admin and ETL servers will provide several APIs to accommodate select, update, purge (single or all) and replay SEC log records. The deletion also includes second copy in ElasticSearch
* RT (Retrieve)
  + Retrieving data can categorized primarily in four ways to serve different objectives and usages: pagination, data feeding and streaming, file downloading, at last analysis and summarization of data (part of feeding).

| iMBP – Architecture & Design |
| --- |

* + Detail for each retrieving category:
    - Rendering data to Web UI by pagination, There are two ways to do pagination, one that is bounded by defining range from start to end, one has start time, but without ending time.
    - Constantly feeding data to third party, the end user can either be web browser which usually the data is used in graphic drawing or statistical analysis, or to servers or apps such as Spark, AI.
    - Periodically retrieving large data to a designated location.
  + Technical analysis of three categories
    - Pagination: it is the easiest implementation amongst three, data in Cassandra are sorted by timestamp, by giving the last retrieved data with limitation of each page in each request, data can be retrieved without technical complexity in implementation, therefore no implementation will be provided for now, and another way to do is to ask sorted index in Redis first, then call Cassandra with primary keys.
    - Feeding: there are several channels to conduct data feeding, from http to browser, from kafka to other apps or from peer-to-peer communication. Which ways to choose, is very much depended on specified needs and demands. Current iteration only provided http to browser implementation which uses websocket and event stream technology to delivery request data continuously to browser. The expected input has a format with date like: aoi:date:20181101 or aoi:date:20181101:20181103, with time range like: aoi:dateTime:20181101112100 or optional with end time, those formats are subject to change with specific requirements.

| iMBP – Architecture & Design |
| --- |

* + - File: In a distributed environment, each app server is obligated to participate in data retrieving to achieve speed and efficient, one server will act as a master to manage and monitor tasks, the other servers act as a slave to process task assigned by the master. The procedures depict as following:
      * When the file request comes to gateway server, the server selects one of RT servers in a round robin fashion. The selected server acts as a master and begins initialization of processes as a master. A file name returns to user and save to MongoDb with all servers information such as filename, server ids, ips, data ranges, status etc.
      * The first thing the master does is to find an available port in a defined range 4000 - 4100, and figure out available slave servers in RT group, then calculate input date or date time range, assign one day to each server in round robin fashion until out of tasks.
      * Other than group id, each slave server has its own id assigned and saved to memory and Mongo for purpose of tracking and monitoring.
      * After the master finishes the primary tasks gathering, it starts to create an asynchronous server and distribute task to slave servers in the group.
      * Once slave servers received the task from the master, they immediately return status of starting and kick off client process.
      * The first thing the client does is to initialize multi-threaded processes and make connections to master server, follow by retrieving data from Cassandra. There are two approaches to send data back to master, it outlines as below:

| iMBP – Architecture & Design |
| --- |

* + - * + Constantly send data back to master while slaves extract data from Cassandra. This approach is faster in term of performance. The major problem is when a slave server is dead or experiencing an issue, there is no way for other to pick it up from where the issue starts, because of all slave servers are appending data to a file in master.
        + Save data to memory or local first, then master server conducts an aggregation operation after all slave servers have reportedly completed extracting data from Cassandra. This approach is slower but it is easier to pick up where it left when a slave server is dead. During the extraction from Casandra in both approaches, slave servers will send status of processing in a designated interval, let’s say every ten seconds, if one of slave server is dead, the first approach has to start all over again, because the continuously file appending operation from all slave servers left master without knowing which part of file belongs to which slave server. However, for second approach, master can reassign the task to other server to complete the task, and even during the aggregation, if one of slave servers is dead, master can pick up where it left by marking end of file when each slave server finish transmission as the master iterates slave servers sequentially, also, this approach suits to client requirements of ordered data.

| iMBP – Architecture & Design |
| --- |

* + - * + To be considered: What happen to master if it is dead in the middle of transaction? Therefore, data in master must be replicated across all participated servers, the first server in master metadata list should inform the rest of servers that it is in charge now, and then the new master starts aggregation operation from that point on. When master finishes the job before closing out itself, it should send status to all slaves, informs them that the group task is done and cleanup begins. Master also broadcasts status periodically during aggregation.
      * At end of each slave task, it sends a status of completion. When all tasks have been completed, master closes the server, the file download is finished.
* Admin
  + Consul server has no login mechanism to safely guard an untended mistake, it is widely open to everyone, therefore, the Consul cluster should locate behind the firewall and no public access except calls are initialized by Admin server which will act as a proxy, manage login and render consul requests such as configuration deletion or addition.
  + Also, Admin server has obligation to manage authentication and authorization process. (Upcoming detail of an innovative approach to authenticate and authorize a user without middleware or central repository involved such as mysql, mongoDb rtc.)
  + Admin server is responsible for APIs calls for SEC log verification and inspection.

| iMBP – Architecture & Design |
| --- |

* UI
  + The primarily usage is for user to manage its own task or result.
  + Also this is main communication channel between web UI and server. Browser should never directly talk to ETL or RT servers.

**Service discovery & Configuration**

* Consul is our primary service discovery, configuration and metadata management system. Consul requires minimum three servers cluster in production, each ETL server has pre-installed consul agent. Whenever an ETL app starts, the consul agent will send a service call to consul cluster server for registration. Join or remove a service is transparent to API Gateway’s load balancer
* Configuration parameters defined in the application also being loaded from consul server kv store during startup, any configuration change in consul cluster server will reflect to the application immediately without restart of app servers, the flexibility and dynamic configurability of consul greatly extends the control of app behave and flow
* Consul also provides api for configuration and service read, update and delete. The metadata will be rendered to UI from mongo, managed through consul kv store and eventually pushed to all ETL app serves
* Consul comes health checking of each app server with configurable interval

**Logging**

| iMBP – Architecture & Design |
| --- |

* iMBP logging system employs Kafka, Logstash, ElasticSearch and Kibana technology stack. Kafka passes data to Logstash which parses the data to desired format, then it

pushes the data to ElasticSearch for storage, at the end, Kibana is used to render the data visually.

* Logging level will be dynamically configured through consul kv store
* The format of logged data will have basic information such as domain, project, serviceName, timestamp, ipAddress, type, errorMessage, rawData etc.
* Logging system will have a dedicated topic in Kafka such as Error

**Monitoring**

* Spring boot provides library allowing app to pass server related statistics such memory, CPU etc. to Prometheus. Gafana will be used as graphic interface for statistic display
* Domain and project related statistics such as # of request, size of SEC log etc. need to be defined as the project progresses

**Data Storage**

* **MongoDB** will be used to store administration, service related information such as file name etc. Consul configuration data also will be saved in the database as backup.
* **Redis** primarily be used to store indexes generated during ETL, it is project specified approach like AOI
* **Cassandra Database Design**

| iMBP – Architecture & Design |
| --- |

* + The objective of big data database allows user to read and write a large quantity of data in a scalable way. Unlike traditional database, it aims to fit all scenarios in one basket, big data is tailored to specific needs and query-oriented approach. To design a big data schema, the first step is to know your data, and follow by schema design, at last tune and optimize Cassandra cluster server.
  + AOI is an image based project, there are tens and thousands of images generated each day, each image has size in range of 55k to 150k, the largest volume generated by single device from sample data has more than 300 megabytes, 3589 images in total (20181105, SOP8-50-150). The largest volume in a day is 20181103 which has more than 4.4GB in size, and 368,123 images in total.
  + Below is AOI entity with all fields

|  |  |  |
| --- | --- | --- |
| * + - created\_day | * + - text |  |
| * + - device\_type | * + - text |  |
| * + - label | * + - text |  |
| * + - created\_time | * + - timestamp |  |
| * + - board\_id | * + - text |  |
| * + - board\_loc | * + - text |  |
| * + - extension | * + - text |  |
| * + - file\_name | * + - text |  |
| * + - product\_type | * + - text |  |
| * + - image | * + - blob |  |

* + - The first thing to design a schema is to figure out what is partition key, by previous data analysis and visually inspection, using created\_day and device\_type in case of 20181105 and SOP8-50-150 are well above Cassandra suggested 100M threshold, therefore, the key needs to break up

| iMBP – Architecture & Design |
| --- |

further with an additional basket, since created\_day is date object has 24 hour time span, if break it to hourly, there are 3600 seconds in an hour, in a worst scenario, every second there is an image generated for the device, in average of 100k per image, the partition size of hourly data would be 3600 x 100k = 350M, it is still more than 100M, what about break it to minute, each minute has 60 seconds, 60 x 100k = 6M, bingo! It is well below the threshold of 100M, we can live with it.

* + - The label and created\_time will be used as cluster key to guarantee the uniqueness and inequality search attribute with descending order of created\_time.
    - The final schema looks like: (more in configuration section)

|  |  |  |
| --- | --- | --- |
| * + - created\_day | * + - text | * + - k |
| * + - device\_type | * + - text | * + - k |
| * + - hour | * + - int | * + - k |
| * + - minute | * + - int | * + - k |
| * + - second | * + - int | * + - k (optinal) |
| * + - label | * + - text | * + - c |
| * + - created\_time | * + - timestamp | * + - c desc |
| * + - board\_id | * + - text |  |
| * + - board\_loc | * + - text |  |
| * + - extension | * + - text |  |
| * + - file\_name | * + - text |  |
| * + - product\_type | * + - text |  |
| * + - image | * + - blob |  |

| iMBP – Architecture & Design |
| --- |

* + - AOI schema is time based, each month should have its own table, it is better and easier for data archive and storage further down the road, also reduce and free up Cassandra disk occupation.
    - By the time of writing, don’t know what are queries and application requests would be, to overcome the uncertainty hanging over the head and no further delay in the project, I came up with a strategy which uses secondary index outside Cassandra server may serve the purpose and greatly improve Cassandra access efficiencies. There are a few benefits associated with the secondary index, which are flexibility, less memory footprint in cluster and efficient in performance.
      * During ETL process, a primary index is built with ordered in timestamp, the index is saved in Redis, with help of redis sorted set, the index looks like this, (system+”:”+created\_day, partitionKey+”#”+clusterKey, millisecond), the composite partition key and cluster key is separated by #, if a new query comes in such as retrieving all images for a label, Cassandra won’t allow you to do such query without knowing the partition key first, traditional approach is to construct second table, duplicate data with label as partition key, that is not only waster disk storage, but also it is time consuming of data copy to new table across servers if petabytes of data already set in existing table. With primary index in Redis, the second index can be constructed based on primary index and using label as key, the value are same as primary index.

| iMBP – Architecture & Design |
| --- |

* + - * With secondary index in Redis, retrieving data becomes much easier and faster. Let’s see if a query needs all data from 20181103 which has over 4.4Gb in size, traditionally, Cassandra will load all data in memory first, during the period, if request time exceeds predefined time, no data will retrieve, even the request time is enlarged bigger enough to accommodate the operation, what happen if the available memory is less than 1Gb, the request may max out with 4G request size. With secondary index in place, Cassandra is able to retrieve data in a small scale, which vastly reduce server memory footprint pressure, and overcome the issue of request time constraint.
      * Tested index in my local setup, three VM Cassandra server cluster, 5G memory and 2 CPU cores in each Cassandra server, one VM Redis with 2G memory, one app server, Ubuntu 18.04 and with data replication of 3. To retrieve 448M data of 20181112, it takes about 9 seconds, 1G data of 20181109, takes 19 seconds, over 4.4G data of 20181103, takes 81 seconds.
    - To be further consideration: if the schema to be designed with even more granularity, break partition key down to second. What could be the performance impact between minute and second? Theoretically, break partition key down further to second, Cassandra would contains less values in each partition size, results in quicker time to grab and retrieve, also small footprint in memory allows quicker garbage collection time and shorter pause.

| iMBP – Architecture & Design |
| --- |

* + - Second to be considered: if access pattern is monthly, or slightly across over the other month, should the month table be loaded into memory for faster access?
  + Server configuration and optimization
    - At least 5 servers in cluster, 32 core CPUs, 250G memory and 64G for heap size, disk space should be able to save one year data (need more accurate estimation), but with table schema constructed on monthly base, data can move in or out without affect others.
    - Java should use G1GC instead of default CMS, G1GC first looks up the region that contains the most of garbage objects, then compact it on the fly, unlike CMS is a stop-the-world approach, G1GC has lower latency than CMS and tunable, and is easier to configure.
    - num\_tokens: value of 16 has best performance based on the site <https://www.instaclustr.com/cassandra-vnodes-how-many-should-i-use/> under condition all tables must has same replication factor and allocate\_tokens\_for\_keyspace must be specified.
    - hinted\_handoff\_enabled: it is primarily used by write operation, can be turned off when monthly data collection is complete. Adjust max\_hint\_window\_ms as same as gc\_grace\_seconds which indicates the hint expiration time.
    - Enable LZ4 compression if hint is enabled.
    - Separate commitlog and data directories in different HDD.
    - key\_cache\_size\_mb: set to disable, no much performance gain.

| iMBP – Architecture & Design |
| --- |

* + - Enable OHCProvider: it is off-heap row cache than partial off-heap cache of SerializingCacheProvider. Disable if no repeatable queries.
    - row\_cache\_size\_in\_mb: set to 1/64th of heap size.
    - row\_cache\_save\_period: depended on queries repeatable frequency.
    - counter\_cache\_size\_in\_mb: set to 0.
    - concurrent\_counter\_writes: 0
    - concurrent\_reads: 16\*number of drives.
    - concurrent\_writes: 8\*number of cpu cores.
    - concurrent\_materialized\_view\_writes: 0
    - file\_cache\_size\_in\_mb: set 2G opposed to 512M default
    - buffer\_pool\_use\_heap\_if\_exhausted: true
    - memtable\_heap\_space\_mb: the default is ¼ of the heap, too large for 64 suggested heap size, and if the Cassandra hosts multiple column families, so the heap consumption per node would be memtable\_heap\_space\_mb \* number of column families, therefore, 4G is good enough to accommodate one day data
    - memtable\_offheap\_space\_in\_mb: 4096
    - rpc\_address and listen\_address: set both to local ip address
    - concurrent\_materialized\_view\_builders: 0
    - column\_index\_size\_in\_kb: using default 64k may incur two indexes for each column has minimum image size of 55k, since the largest image size is 150k plus fields and overhead of column index, 250 probably is good estimation for all size.

| iMBP – Architecture & Design |
| --- |

* + - counter\_write\_request\_timeout\_in\_ms: 15000
    - range\_request\_timeout\_in\_ms: 30000
    - read\_request\_timeout\_in\_ms: 15000
    - request\_timeout\_in\_ms: 30000
    - truncate\_request\_timeout\_in\_ms: 180000
    - write\_request\_timeout\_in\_ms: 6000
    - cas\_contention\_timeout\_in\_ms: 3000
    - slow\_query\_log\_timeout\_in\_ms: 1500
    - endpoint\_snitch: SimpleSnitch if one datacenter, change to GossipingPropertyFileSnitch for multi-datacenter.
    - Cassandra comes with default SizeTieredCompactionStrategy (STCS), by testing with LeveledCompactionStrategy (LCS), yielded no difference since AOI project is read only, no modification operation associated. By Cassandra document, STCS merge process doesn’t group data by row which yields slightly disadvantage on read, also compaction requires large amount of disk space may outstrip a node disk capacity. In other hand, LCS is grouped by level, 90% read can be satisfied in one SSTable, and therefore, I choose LCS with default setting.
    - Enable node to node and app to node encryption may not be necessary, it adds unnecessary overhead, but all servers should be behind firewall and only allowable ports be opened. If still requires, follow the link of how to setup. <https://www.linode.com/docs/databases/cassandra/set-up-a-cassandra-node-cluster-on-ubuntu-and-centos/>

| iMBP – Architecture & Design |
| --- |

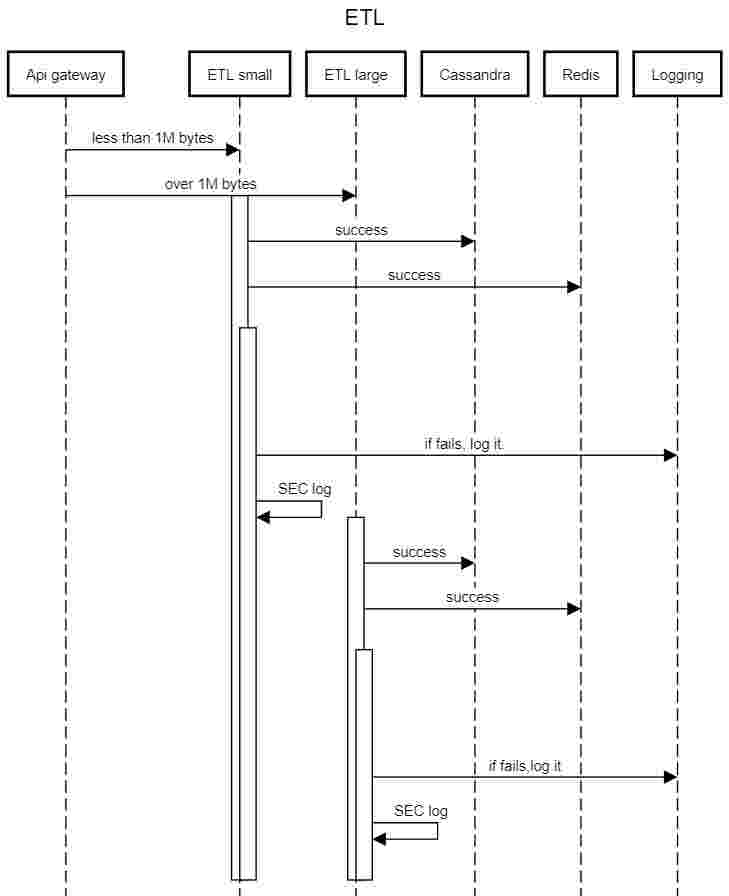
* + - Needs improvement in Cassandra: allow specific rows or keys to be cached in configuration file.

**Security**

**– An innovative approach to Authentication & Authorization (Upcoming)**

**ETL Sequence Flow**

| iMBP – Architecture & Design |
| --- |



| iMBP – Architecture & Design |
| --- |

**SEC log Algorithm**

* SEC log algorithm is based on simplified LSM algorithm with variation of update and save operations. The SEC log system always operates on appending data at end of a file without modifying and deleting single data for fast write performance, also SEC log system provides index access which is stored in both disk and memory for durability and fast random read access.
* Save: index id is generated using input payload with Murmur3 hash function for both SEC log and iMBP logging system. To prevent duplicated entry, the id is checked against index file, app will do nothing if index exists and doesn’t mark as a tombstone, otherwise, app will bring tombstone to a live data.
* Update: generate a new index id and old one will be marked as a tombstone, payload will be appended to data file.
* Purge: to purge a single data, the index of the data will be marked as a tombstone, the data will not be removed from data file. To purge all of the data, both index and data sec files will be cleaned up.
* Re-run: after saving single file to Cassandra, index will be marked as a tombstone. After running all of the data, index and data files will be emptied out.
* APIs: <http://10.208.136.33:8080/browse/AOISEM-49>