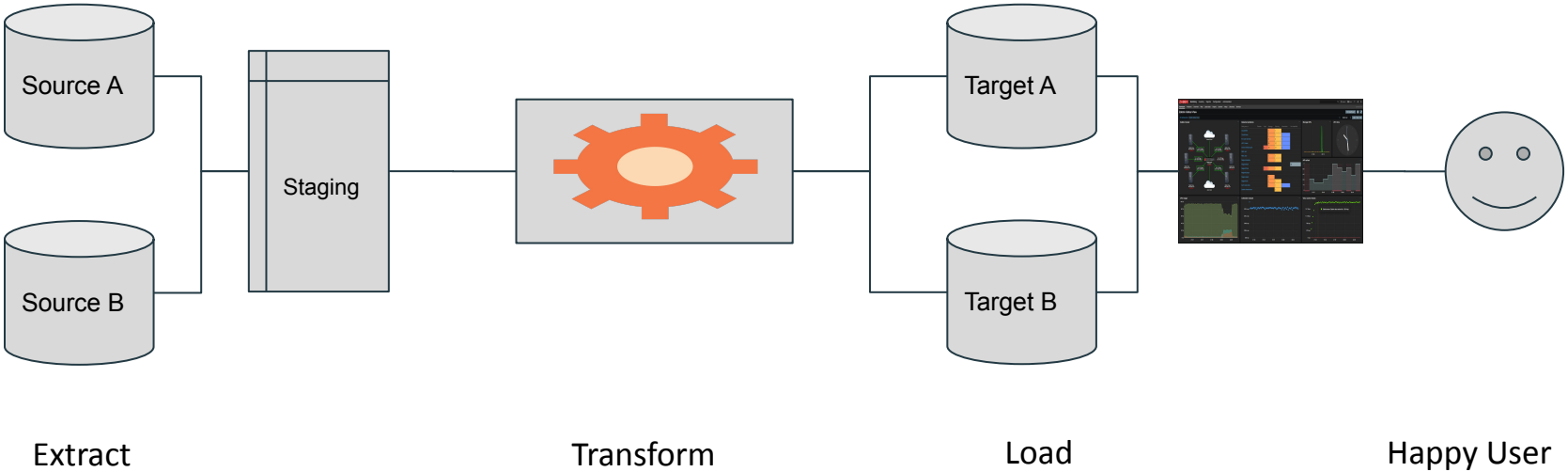
Addressing Challenges of ETL Using Apache Spark

About us

- We are a leading ride hailing service provider in the middle east touching millions of lives everyday.
- To improve customer satisfaction, we have enhanced data driven decision making.
- Our in-house ETL solution makes heavy use of Apache Spark to handle complex real world problems.
- I’m one of the lead contributors of our in-house solution to address the needs and demands of the ETL.

ETL: From Exosphere

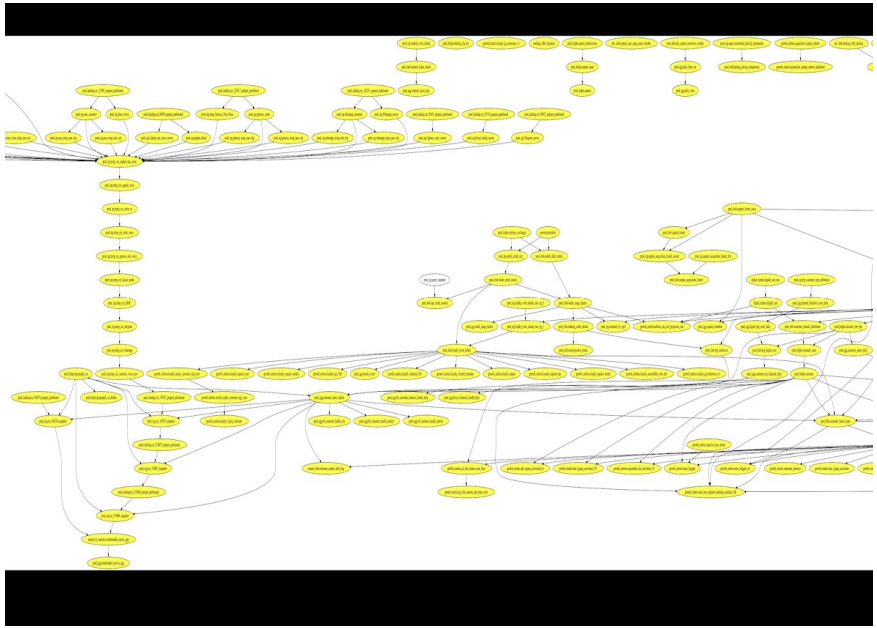


ETL: In practice

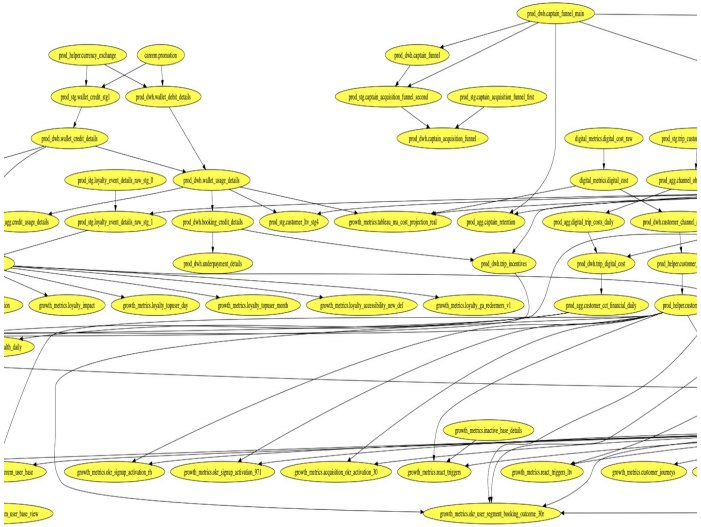
1. Any process can fail
    a. Trying to load into a table or read from a location which does not exist
    b. Resources not enough for the process
    c. Network issue or any unpredictable error
2. On failure a process can get manually triggered multiple times
3. In addition of running different process on different queues, different process can also be running over different clusters
4. Components can get added/removed/modified
    a. New source/target added or existing removed
    b. Sql file updated to support new KPI or column change
    c. New mechanism of alerting/monitoring added
5. Not all sources are extracted at the same frequency, some involve batch processing while some need real time treatment
6. Security is needed at every stage
7. Maintaining data quality

ETL: In practice

Complex Dependencies in Processing/Transformation



Little Zoom



Challenges of creating an ETL Solution

1. Supporting batch and streaming use cases
2. Early detection of failure or delay
3. Less manual intervention (Ideally zero)
4. Extensible in supporting new sources/targets/processing
5. Easy to be used by the end users
6. Quick in adding new dependency and handling complex dependencies without affecting others
7. Intelligent in resource consumption
8. Scale to support large data processing

Possible Solution

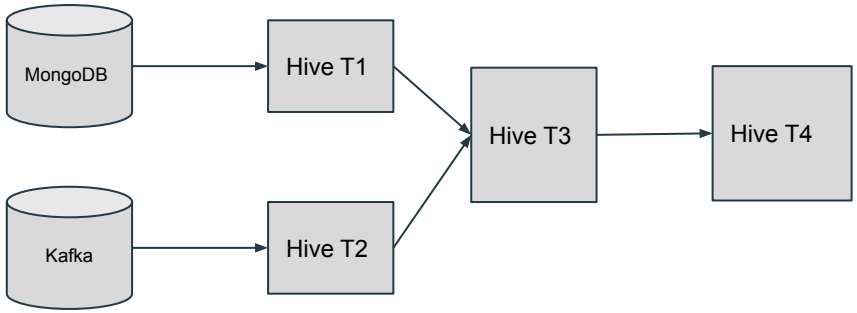
Using a scheduler like Airflow with SubDAGs

- Pros
  i. Handles dependencies with ease
  ii. Supports task retries to reduce failures due to temporary issues
  iii. With Subdags, logically same entity can be grouped together
- Cons
  i. Not intuitive for running streaming jobs continuously
  ii. Need to build application for each
    1. Source
    2. Target
    3. Processing logic
  iii. High code redundancy when dealing with event sourcing, monitoring, alerting and any other feature common to all types of applications
  iv. Need different service to keep track of the freshness of loaded data (ie. how recent was it successfully loaded)

Shakti: Our In-House ETL PowerBooster

- Shakti means power
- Shakti powers our ETLs, handling complex dependencies of different types of user tasks at ease and at scale
- Spark-Scala based ETL boiler plate to:
    - Support variety of user tasks (Eg: Hive Task, MongoDB Task, Redshift Task etc.)
    - Provide optimisations across all the tasks and dependencies
    - Leverage the power of Spark and added customizations just by passing:
        - a json file containing parameters
        - optionally a configuration file (containing spark configurations)
        - a sql file(if it is a spark sql application)
    - Support various stages of ETL Pipeline: Ingestion till Presentation

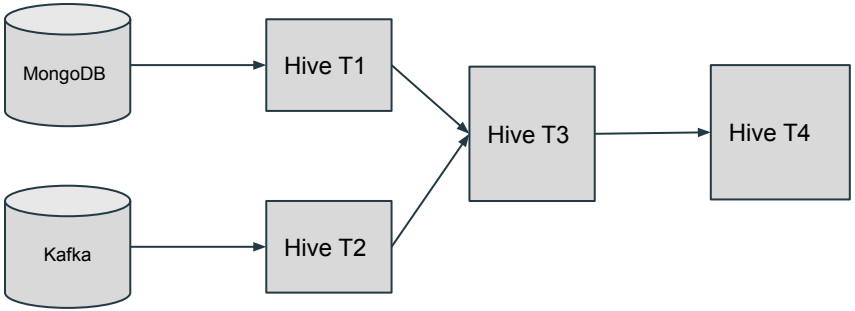
Shakti: How it works



What information is needed:

1. The location and name of MongoDB table/collection
2. The location and name of Kafka topic
3. The names of the hive tables T1 and T2
4. The query to load T3 from T1 and T2
5. The query to load T4 from T3
6. The dependency that: T3 depends on T1 and T2. Also, T4 depends on T3

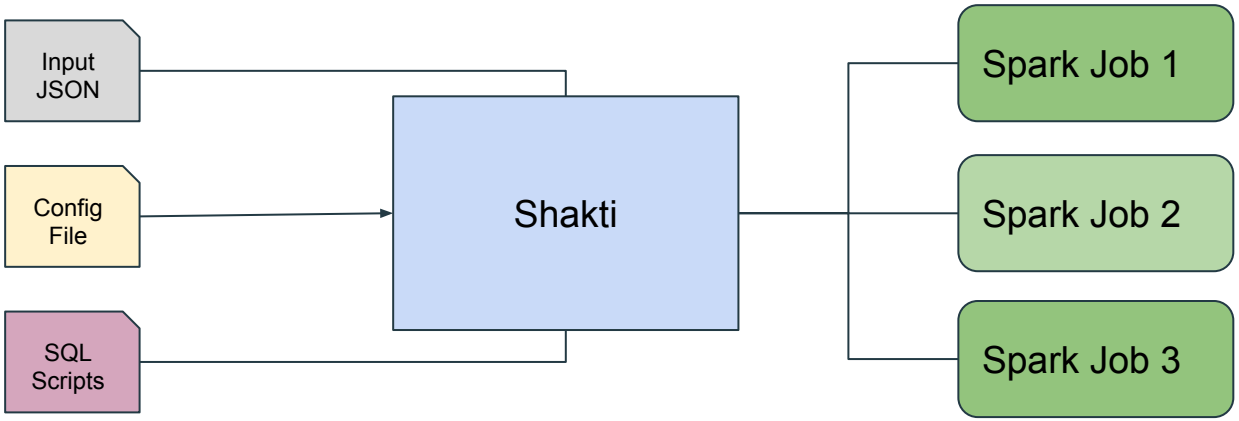
Shakti: How it works



How to create this pipeline with Shakti:

1. Give the details in the JSON.
2. Some tasks are related ingestion, while some are transformations. Shakti takes care of all
3. Some tasks are batch and some are streaming, Shakti runs them all
4. The dependencies are optimised at run time
5. For hive tables, the partitions are swapped (since we use S3)
6. Fresh tasks do not execute
7. Parallelize the tasks, if they can be
8. No duplicate tasks run together
9. Dependencies can be across multiple JSONs
10. Blacklisted tasks and its children are skipped

Shakti: How it works



Spark jobs are launched in parallel decided by user-defined config and following the dependencies in the JSON.

Shakti: Tasks

- Tasks are extensible.
- Just extend the **Executable** trait and start using **SparkSession** inside.
- Override **exec()** method to provide what task should do
- Some of the existing tasks are:
    - Hive Task
    - MongoDb Task
    - Parquet Writer Task
    - Redshift Task
    - Compaction Task
    - Merge Task
    - External User Task

Shakti: How JSON looks like

```
{
      "task_id": "current_task",
      .
      .
      .
      "dependenciesWithMeta": {
            "parent_task": {
                  "freshnessCriteriaInHours": 1}},
      .
      "sql": "location of sql file",
      "sla": 1800,
      "task_type": "hive",
      "post_sql": "sql for Data Quality",
     "freshnessCriteria": 1,
      .
      .
      .
}
```

For the current_task to run, parent_task must have succeeded in the last 1 hour

If that is not the case, current_task will:

- Either, wait on the execution of parent_task (if the parent_task is defined in the same json)
- Or, time out (if the parent_task is defined in a different json)

If the current_task has succeeded in the last 1 hour, it will be considered fresh and will not execute in the current run

Shakti: How application conf looks like

```
execution {
  number-of-threads = 2 ; Controls the number of spark jobs to run in parallel
  slack-notification-endpoint = "https://hooks.slack.com/services/TXXXXX/XXXXX/XXXXX"
}

task {
  kafka = {
    brokers = "kafka101:9093,kafka102:9093,kafka201:9093,kafka202:9093,kafka301:9093,kafka302:9093"
    producer = { ... } ; Configs to apply on kafka producer
  }
  zookeeper = {
    "url" = "zookeeper100:2181,zookeeper200:2181,zookeeper300:2181,zookeeper400:2181,zookeeper500:2181"
  }
}

spark {
  application.name = XXXX
  config { ... ; configs to be contained by the SparkContext
    spark.dynamicAllocation.initialExecutors = 4 ; One example
    ...
  }
}
```

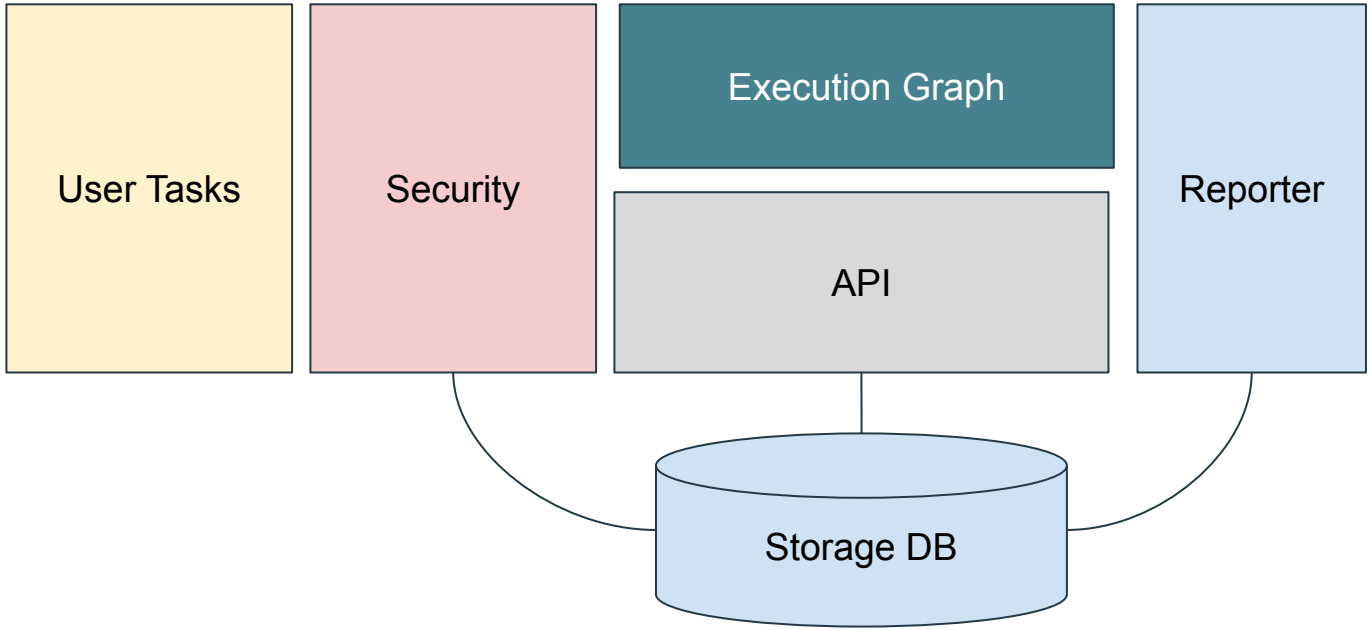
Shakti: How to run

```
#!/usr/bin/env bash

application_config="path_to_application_config"
tasks_json="path_to_tasks_json"
specific_tasks="specific_tasks_to_run"

spark-submit \
--packages ... \
--class shakti.spark_sql_etl.Main \
--jars ... \
--driver-memory ... \
--num-executors ... \
--executor-memory ... \
--executor-cores 4 \
--conf ... \
--files ... \
--driver-java-options="-Dconfig.file=${application_config}" \
shakti-assembly*.jar ${tasks_json} ${specific_tasks}
```

Shakti: Components



Shakti: Why choose Spark

- Fast and performant in-memory processing
- Support to cache frequent data
- Powerful and elegant data unification, composing streaming, sql and RDD operations together in the same application
- Support in multiple languages, we use Scala
- Rapid development, fits we

Shakti: Web API

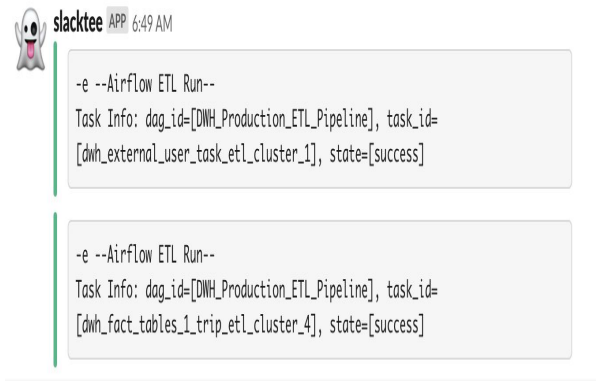
shakti-backend.engineering.com:8899/shakti/metrics/v1/runtime?optimus_task_id=prod_dwh.trip

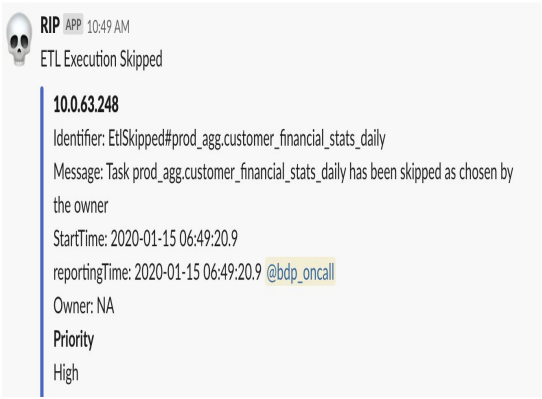

```
{
    "etlType": "HIVE",
    "duration": 365,
    "startTime": "2020-01-16 02:34:26.0",
    "entry_id": 152096,
    "optimusTaskId": "prod_dwh.trip",
    "endTime": "2020-01-16 02:40:31.0",
    "status": "SUCCESS",
    "targetName": "prod_dwh.trip",
    "message": "NONE",
    "clusterName": [redacted],
    "authorName": [redacted],
    "updateTime": "2020-01-16 02:40:31.0"
}
```


shakti-backend.careem-engineering.com:8899/shakti/speculation/v1/status/blacklisted

```
cli_test_1
prod_agg.customer_financial_stats_daily
```

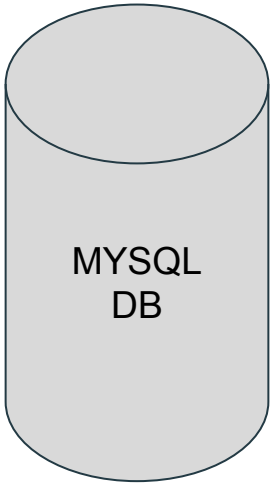
Shakti: Operations Alerting





Shakti: Operations State Management

etl_meta_entries	
etl_id	int(11)
task_id	varchar(512)
etl_type	varchar(64)
start_time	timestamp
end_time	timestamp
status	varchar(64)

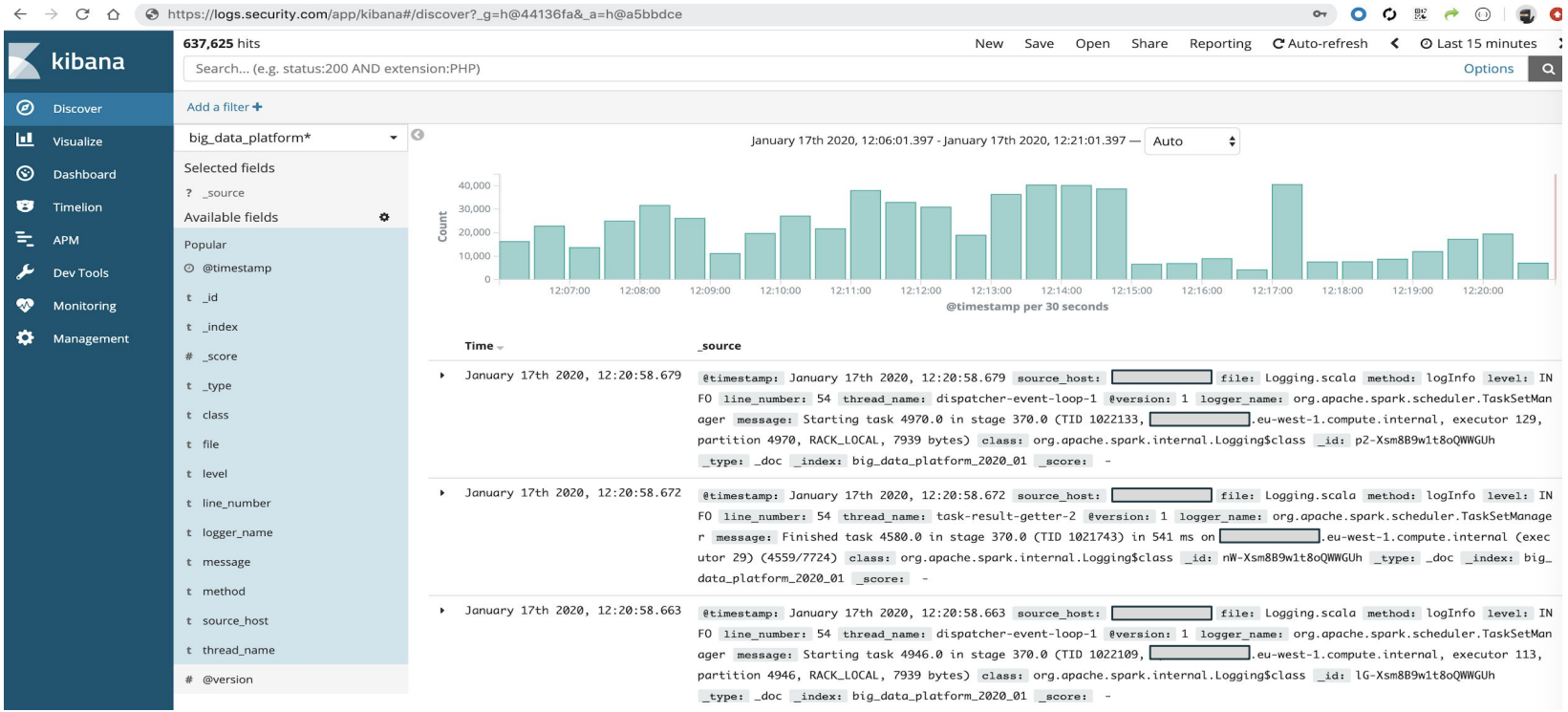


etl_speculation	
etl_id	int(11)
task_id	varchar(512)
requested_by	varchar(64)
status	varchar(20)
creation_date	timestamp
expiration_date	timestamp

Shakti: Operations
Event Sourcing


```
{
 "clusterName": "ip-10-0-62-120",
 "appName": "nrt_15.json",
 "executionId": "c90e4706-15eb-46d0-8ac1-84051a282b4b",
 "taskId": "nrt_avail_hr",
 "status": "succeeded",
 "message": "task succeeded",
 "ts": 1578563293559
}
```

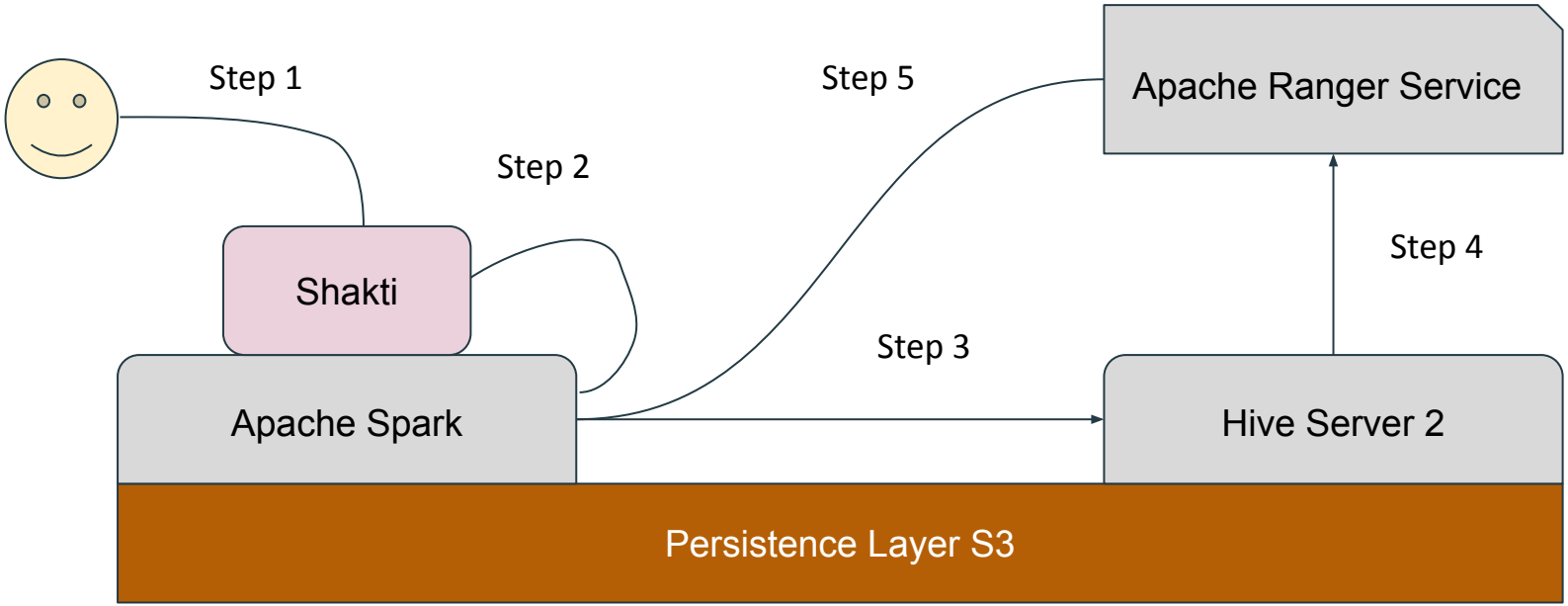
Shakti: Operations
Central Logging



Shakti: Operations
Monitoring Resource Utilisation

- Use NewRelic to monitor the resources requested by the application and the resources actually used by it.
- Graphana can also be used along with a time series DB like prometheus/graphite.
- Make use of a JMX exporter jar
    - -javaagent:/home/hadoop/newrelic/newrelic.jar

Shakti: User Authentication and Authorization



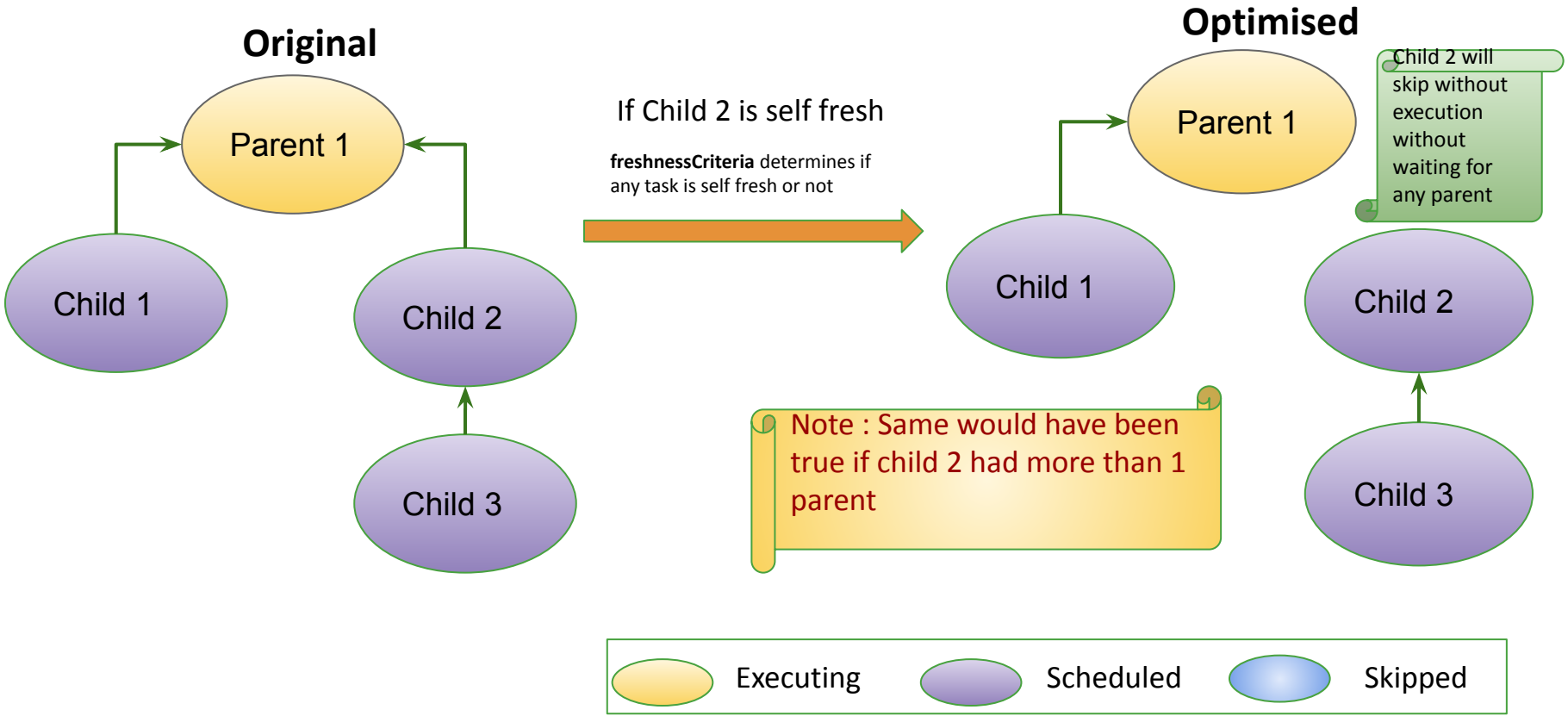
Shakti: User Authentication and Authorization

1. User submits the query.
2. User query gets intercepted based by spark extension manager, this allow us to add custom logic to dispatch SQL query operation to hive using HiveAuthImpl.
3. HiveAuthorizer is invoked on the proposed LogicalPlan of SparkSQL transformation.
4. HiveAuthorizer then invokes RangerHiveAuthorizerFactory, this is an entry point of Out of the box Ranger support for Hive.
5. Authorization is performed and based upon the grant access user operation either succeeds or access-denied error is thrown with detailed message.

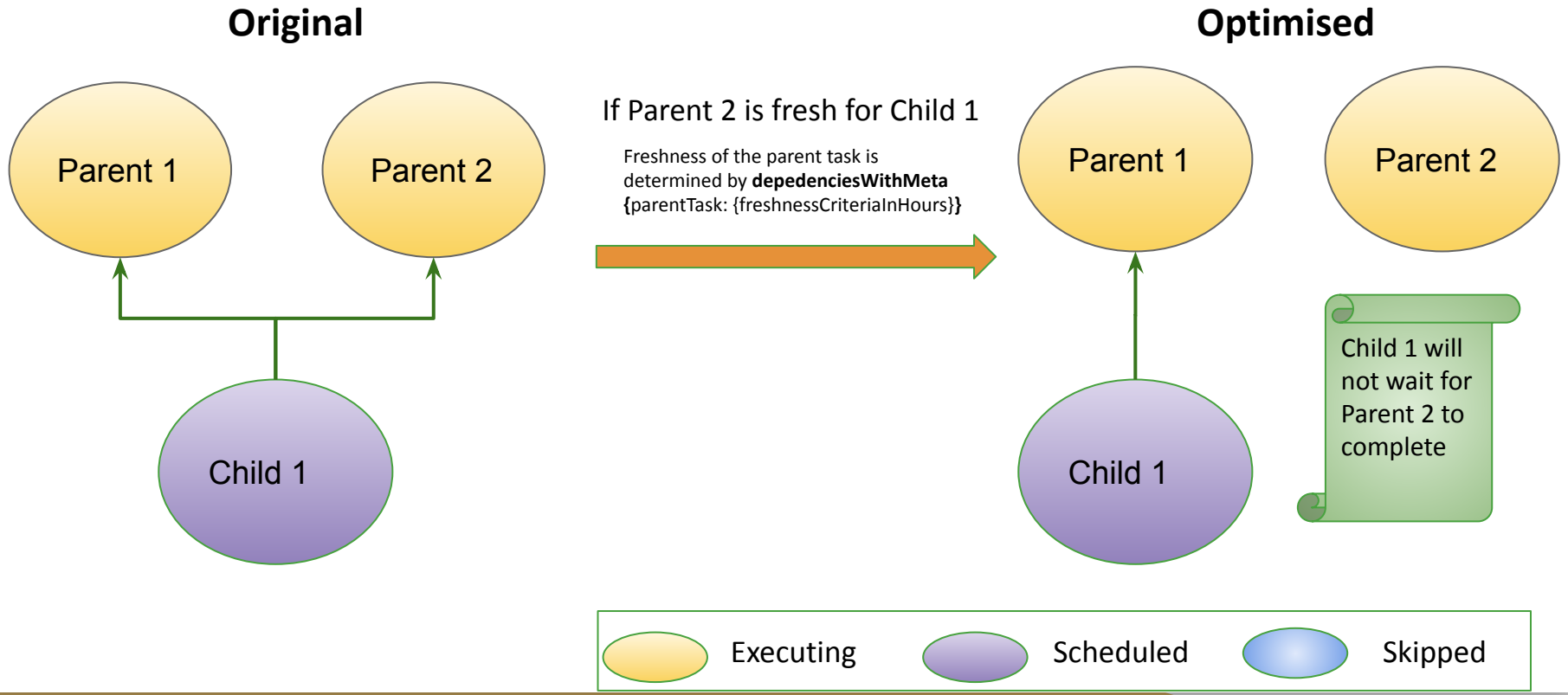
Shakti: Optimisations

- Use of dynamic allocation of executors
  - spark.dynamicAllocation.enabled = true
  - spark.shuffle.service.enabled = true
- Tuning parameters based on the requirements of the spark job
  - --driver-memory
  - --num-executors
  - --executor-memory
  - --executor-cores
- Making use of adaptive execution
  - We are currently using Spark 2.4.0
- Using different version of fileoutputcommitter to write to S3
  - mapreduce.fileoutputcommitter.algorithm.version = 2
- Creating a swap directory in S3 for writing current data
  - Point the hive table location or the partition location to the swap location and prune it later
- Considering data freshness
- Restructure DAG

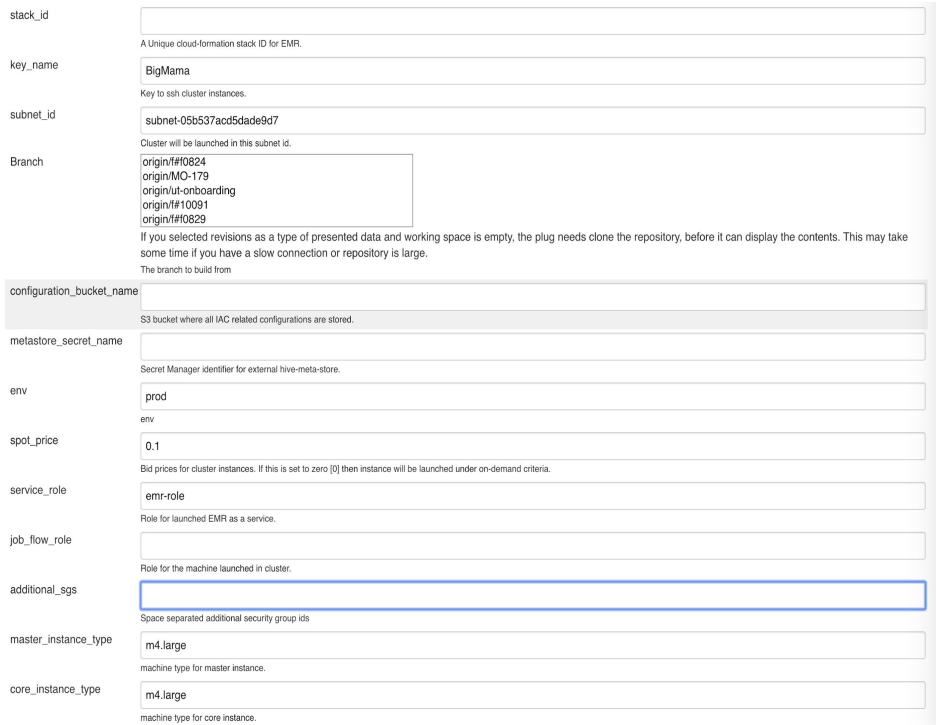
Shakti: Optimisations

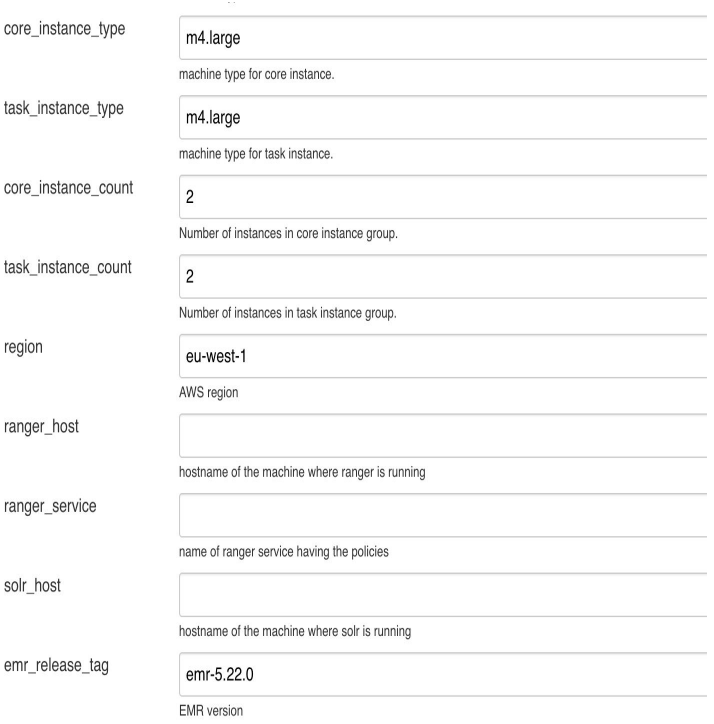


Shakti: Optimisations

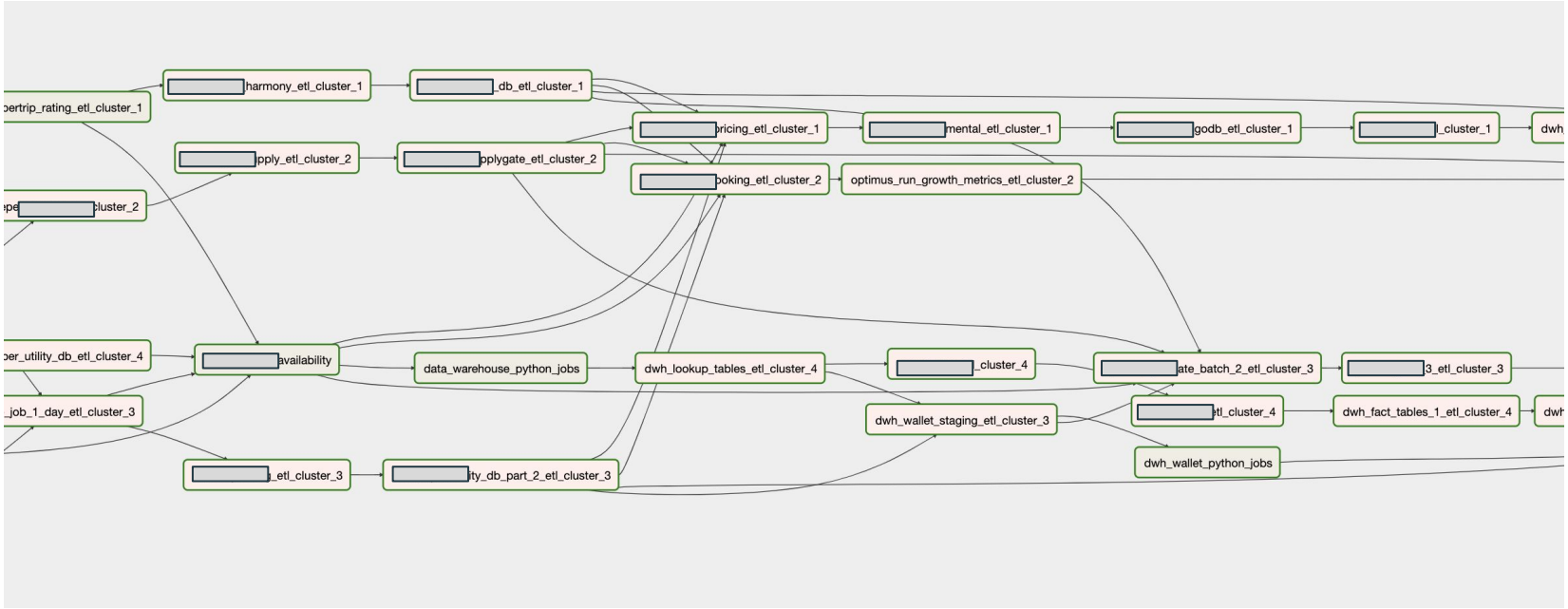


Shakti: IAC on Cloud





Shakti: With Airflow



Shakti: Usage Report

- **>150** big jobs running every hour **powering all of organization's data team.**
- Daily hands-on support to all verticals, including Warehouse, Data Science and BI teams.
- **2TB of data** processed per hour.
- Near Real time Data lake in production with critical use cases.
- **> $250K cost reduction** of the platform cost.
- Massive work on **tech compliance**.
- Users **running ~20K queries** per day through presentation layer.

Shakti: Road Ahead

- Query cost computation: With feedback, decide at run time the resources to be allocated
- Maintaining current hot data in HDFS and background sync to S3
- Analyzing DeltaLake and EMRFS
- Web-based self service data platform, for enhanced UX
- Open source the project