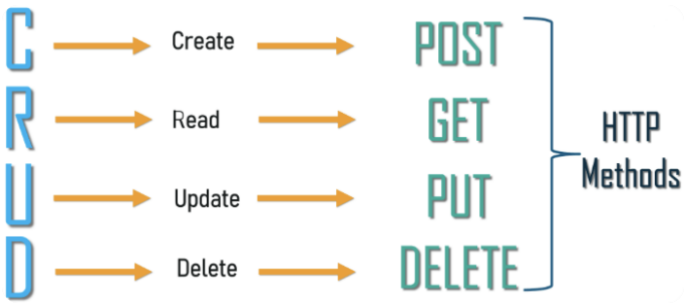
Generic Back-End Development - SIRIUS

Problem Statement

Backend development - for exposing business flows has been historically CRUD. This basic structure never changed.



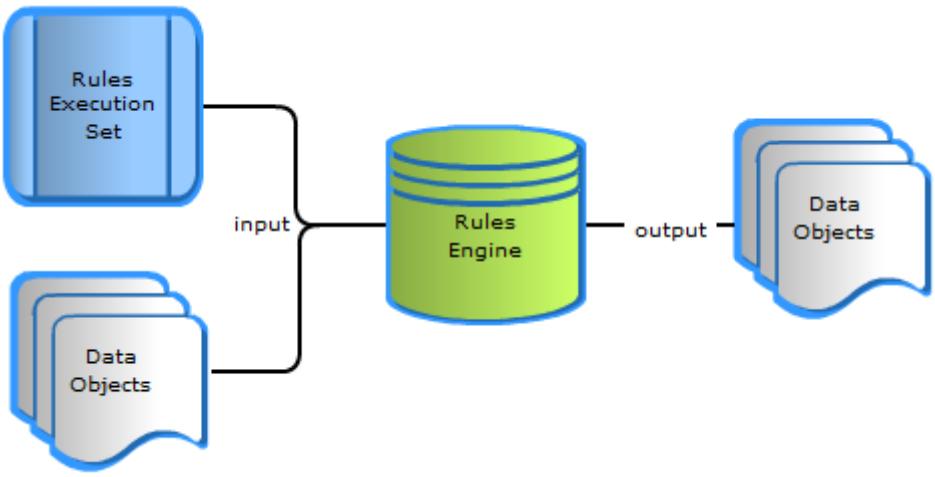
Definition of transactionality though remained the same - the "visible" behaviour changed over time (take SAGA for example).

Thus any typical API does any or both of these :

1. Read/Write data (more often than not from/to a data store - by simply wrapping around) - atoms of operations
2. Mix and match different APIs - data transformation & aggregation

Hence we ask the question - is it justifiable to keep on creating thousands of API's with hundreds of teams for trivial business processes? This question from cost perspective has very clear answer - while from the cash rich firms perspective has different answer.

Some argue that the "business logic" should not even allow for such a generalization - while some others argue that the "business logic" should be never put inside anything "compiled" in the first place - of sorts - these counter points are also proponents of Rule Engines.



Immaterial of that what can be generally argued that one system can be built upon which "trivial" end point creations should be trivial as drag and drop. Given Engineering is applied science to garner maximal ROI within a strict timeframe - hence the objective of the project was to enusre we can do more (create and maintain more API end points) with less (resource, including people). This was the motivation for SIRIUS.

Historical Foundation

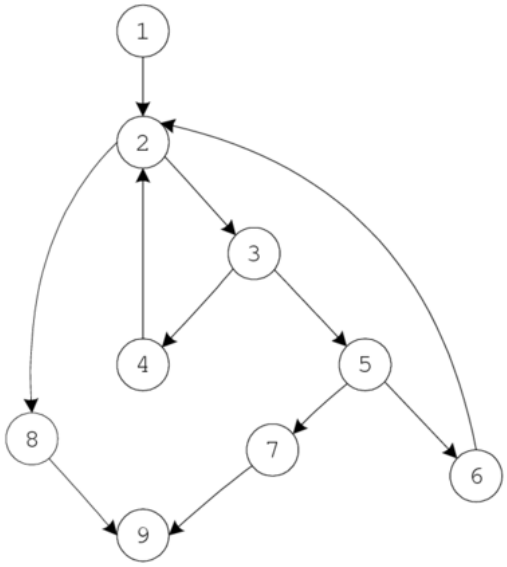
In 1960s before the advent of "specialized data stores" COBOL let people read and write from files and let people be transaction oriented. Only in 1970s the idea of a generic component as "database" was begining to be formalized - and there was a DSL that defines the structure of computation over data - declaratively SQL. By 1980s that became standard.

Imperative paradigm is the closest to the bare metal under Von Neumann Arch. The problem at hand was not new. Any computation - can be taken as a data or control flow graph - which is then a directed graph. If one ignores the cycles (axioms later) the problem of a computation becomes a trivial problem of traversing the graph and just parellize in case multiple processors are available.

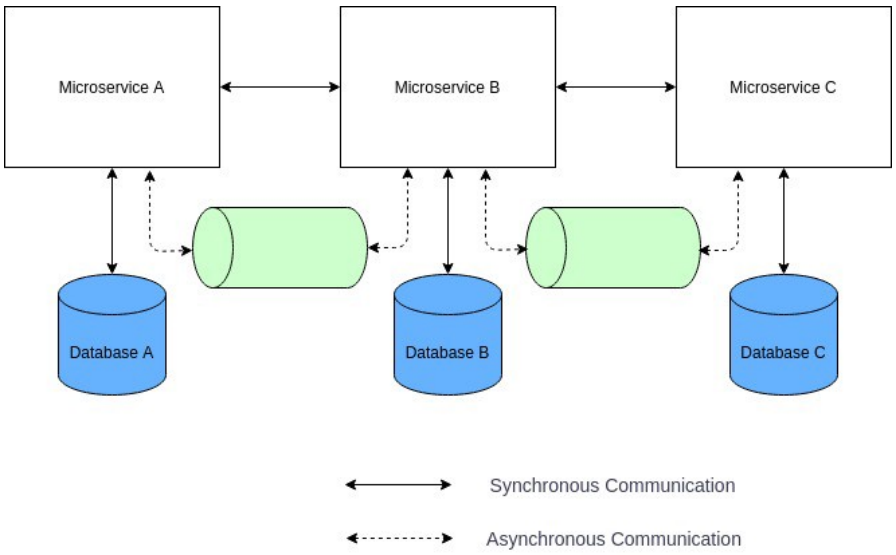
Source Program:

```
int binsearch(int x, int v[], int n)
{
    |  int low, high, mid;
  1 |  low = 0;
    |  high = n - 1;
    |  while (low <= high) | 2
    |  {
    |      3 |  mid = (low + high)/2;
    |        |  if (x < v[mid])
    |        |        high = mid - 1;  | 4
    |      5 |  else if (x > v[mid])
    |        |        low = mid + 1;   | 6
    |      7 |  else return mid;
    |  }
    |  return -1; | 8
} | 9
```

CFG:

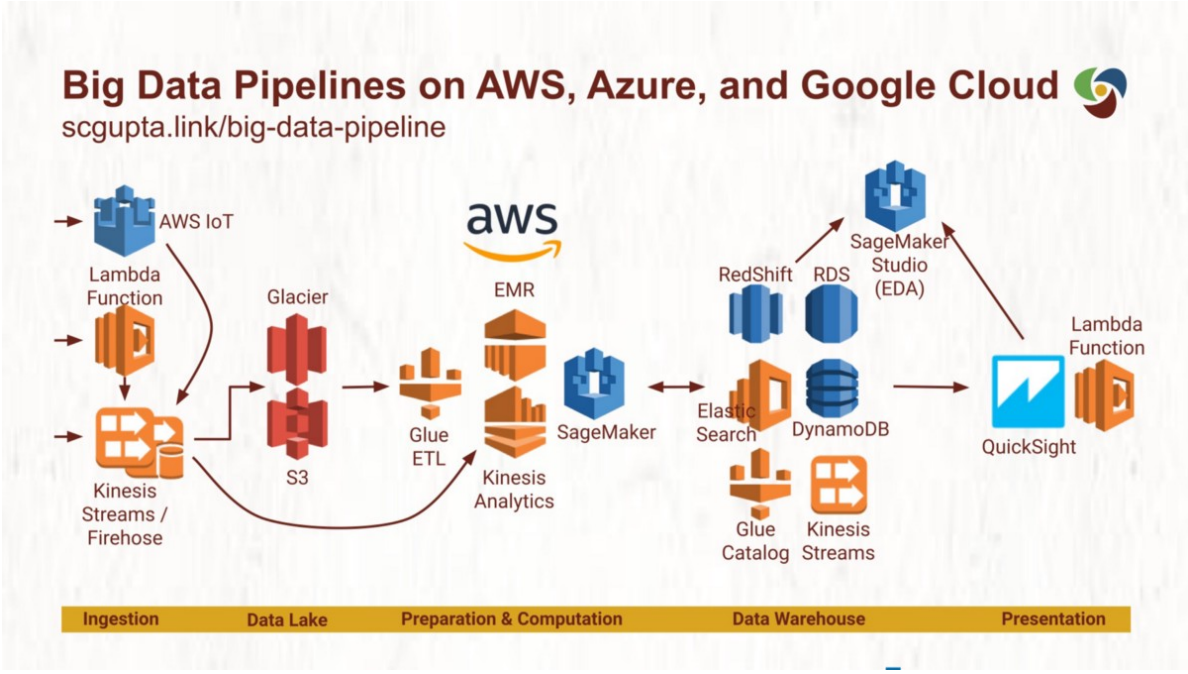


In a virtual level order traversal for the graph - the "above" nodes are the "atom" nodes which triggers the compute, and the lower level nodes take data from the higher up nodes and continue computation till the end bottom nodes reached.



Advent of microservice pattern demonstrated the advantage of abstraction using a singular isolated datastore for the endpoint. Hence, by definition they qualify as the top level atomic nodes to call. Rest of the workflow then can be just connecting nodes which transforms and aggregates data.

The whole workflow, then, as a dependency graph, can be represented in a custom DSL form - in the spirit of Maven or Gradle - or even just as Kubernetes yaml files. Again, this is not new. Data Pipeline has been a popular trend in the last 10 years:



Instead of running non real time, the project simply ensures the run in real time. While in the "slow" spectrum there is AirFlow, "not slow" Cadence - and "right now" would be SIRIUS. Many features of SIRIUS overlaps with GraphQL - and that is not coincidence. GraphQL was build for aggregation (sans JOIN) while SIRIUS was build to not only solve the aggregation problem but also the whole real time data pipeline problem.

A whole analysis about graphQL vs SIRIUS is out of scope but can be created out of this document and references.

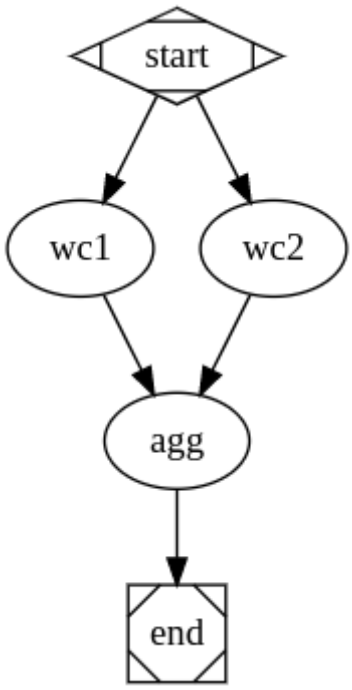
Algorithm

```
name : "some workflow"
timeout : 500 # in ms
nodes:
  agg :
    depends: [ wc1, wc2 ]
    expr : "select wc1.id, wc1.a, wc2.b from inner join wc2 on wc1.id = wc2.id"
  wc1:
    timeout : 200
    depends : []
    url : "get@/foo/wc1?q=${ARG.wc1_param}"
  wc2:
    timeout : 200
    depends : []
    url : "get@/foo/wc2?q=${ARG.wc2_param}"
```

This depicts a very simple aggregation workflow which ends with the node "agg" which depends on two web calls.

Compute Graph

As described before, the abstraction for any computation starts with the compute graph. The nodes for such a graph defines atomic compution of sorts. The relation between the nodes are "depends on" relation. Thus the compute graph is a dependency graph just like Maven or Gradle.



Nodes

A node can be of 2 types. They have their own retry strategies and timeouts. Nodes are equipped with Object mappers - which declaratively can transform any object to another object. A node must disclose all nodes it depends upon by name.

Data

These nodes define either an API call or to connect to a data store to load data into the computational graph. `wc1` and `wc2` are data nodes.

Compute

These define the data transformation or aggregation capapabilities. `agg` is a compute node.

Workflows

Workflows are a strict subset of a compute graph. Typically a single SIRIUS API call starts with the "desired exit node" and then inverts the dependency and continues the computation till the desired node is being reached. A workflow must disclose all workflows it depends upon by name. In the demonstrated cases desired exit node can be set to any, but for any practical purpse it has to be `agg`.

Memory Model

SIRIUS works with MIMD - model. There are many potential processors picking up a node to execute. Each node has it's dedicated memory, stored in a shared map. A particulars node's memory is nothing but another map keyed by the name of the node. For example we are gathering the virtual table of `wc1` by giving it the name `wc1` in the join statement.

Initial parameter passed are available to all nodes - as read only. That is how we are getting `ARG.wc1_param`.

It is possible to access other nodes memory, but it is not recomended. After a node is done running the corresponding memory is sealed off - that is becomes readonly, and then it can be accessed by any node further down the line, if need be.

Expressions

Data transformation is done via data binding and it's own expression engine. It supports guarding and adds Turing completeness to the system. The syntax `${expr}` essetially tells the underlying expression evaluator to evaluate the expression based on the memory model.

SQL engine for the in memory mysql engine was also modified to permit Turing Completeness in the SQL itself - specifically adding FIND operation.

Schema

Takes JSON input and outputs in the guise of maps. There are optional typing capabilities to/from typed binary input / output (thrift / protobuf/ avro). This is the job of the type converter. In the custom generic schema notation the following would suffice:

```
wc1:
  in:
    param: string
  out:
    id : string
    a : int
    #...  more here
wc2
  in:
    param: string
  out:
    id : string
    b : string
    #... more here
agg:
  in:
    p1 : wc1.out
    p2 : wc2.out
  out:
    id : string
    a : int
    b : string
```

Basic Execution

A SIRIUS virtual API is defined as follows:

1. A configuration file listing out nodes
2. An exit node that the execution must reach to
3. Parameters to be passed to the execution

Then the execution is as follows:

1. Engine finds out by reverse traversing from the target (exit) node - the subgraph that is to be run.
2. Once the subgraph is identified, it isolates the nodes which can be run via the information already available.
    1. A node can be run if all dependencies already have run
    2. All run nodes gets into a visited set.
3. Repeat [2] till the exit node is reached.
4. The output of the exit node is the output of the computation.

In the context of the given workflow it would start with the `agg` node and will figure out it can not run it. Then it would reach to `wc1` and `wc2` which both can be run parallely if there are enough free processors available. It runs them, and then finally runs `agg` to produce the output.

Functionally, each node is a function. The whole graph of execution is a functional composition.

Components

Configuration Storage

Stores the following :

1. Workflow Yamls (versioned)
2. Schema definitions for nodes (versioned)
3. Meta informations (author, reviewer, ..)

The underlying data gets stored in the following way:

Metadata

```
{
  "id" : "id of the workflow",
  "version" : "version of the workflow",
  "author" : "author name",
  "reviewers" : [ "r1", "r2" ],
  "bin_data"  : "binary data for the yaml file",
  /* later impromvement */
  "data_url" : "location of the actual revision"
}
```

Master Data

```
{
    "id" : "id of the workflow",
    "team" : "team identifier",
    "owner" : "owner for the team",
    "lversion" : "latest version"
}
```

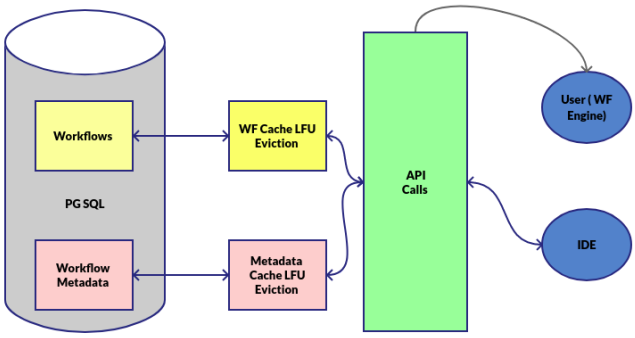
Exposes API to get versioned information for the same given name of a workflow. W/o any version gives the latest.

```
method : "get"
url : /workflow/<id>/[version]
out : {
    base 64 encoding of the workflow file

}
```

```
method : "get"
url : /workflow/<id>
body : {
    "data" : base 64 encoding of the workflow file
    "user_tok" : "auth"
}
out : {
    "status" : "ok|fail",
    "version" : "created version"
}
```

PGSQL was used to store the data along with versions, although a better compressed storage for storing contextual data was underway.



Workflow Engine

This is the computation machinary that runs the nodes to run the workflows. It comprise of node executors. These rely upon 3 independent execution (thread) pools:

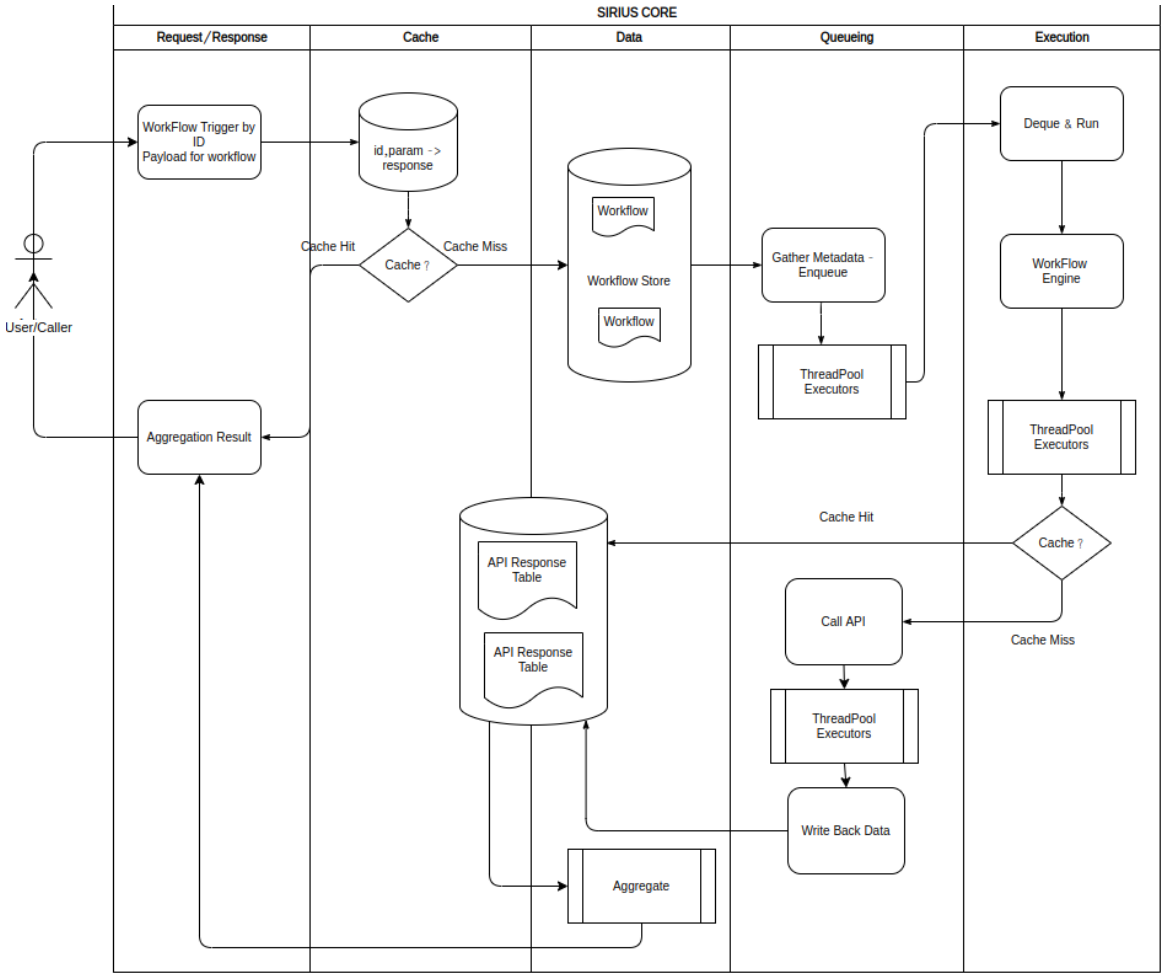
1. Data pool
    1. API calls
    2. Data Store calls
2. Compute Pool - runs the executions of expression

This disjunction allows the system to put retries and timeouts on each "atom" of operations - including data transformations.

L1 cache is the engine memory stored in a map.

Internally it uses an in-memory mysql engine to stage data - and this is the L2 cache. This is why every SQL operation in DML is naturally supported via the engine.

```
method : "POST"
url : /workflow/<id>/<node>/[version]
body: {
    user_token : "auth",
    params : {
        /* depends on what we need */
    }
}
out : {
    /* the target node resp */
}
```



Expression Engine

This is the engine that does the data binding, evaluates expressions in every situation. Every exec pool runs it's own copy of engine, it is thread safe. This is a fork of GOJA, and then customised to incorporate multitude of functionalties which ensures most succint form of expression can be expressive.

The system has the following properties:

1. Sandboxing of scripts
2. Thread safe execution
3. Arbitrary precision arithmetic

4. Virtual Machine based execution
5. Object Mapper - declarative transformation of object of one type to another

Cache

There are two different cache is in place. Redis is being used for both of them.

1. One between the configuration storage and the workflow engine - for caching the most frequently used configurations to run
2. Another in the output where we store the most frequently used
    1. schemas (most frequent)
    2. scripts (expression engine expressions / most frequent)
    3. Optional output caching (ttl)

Auth

Auth Engine had RBAC support. Given multiple API would have different access levels for different users, one could in principle deduce the auth for an user for a particular workflow automatically. Once done an automated request would be sent to the underlying API teams to grant access the respective users for the same.

The algorithm for the same runs as follows. From the target node traverse the "depends on" nodes to find nodes for which a particular user has no access. Add that node to the set of "auth required for the user" nodes, and continue.

It can also be done with any roles, instead of users.

Once such a set is prepared, it is trivial to raise tickets asking for auth for the user/role for the nodes data sources.

Compute nodes are pass through, given data anyone can compute anything.

Logging

Logging system was in place as a plug-in that acted as server with clients like datadog who could gather the final data. The logging system can "pause" execution because it was part of every component. The plan was to create a time series db to store much more interesting behaviour of time series data (shown in th pic).

Alerts and monitoring were in place to put the system into "zones of stability". My own work in my startup was useful to create better models of stability than the rectangle based model.

Debugger

The above behaviour was used to create the debugger infra - where one can potentially pause any workflow in between any "atom" of operation. The state inspection was trivial because they are in:

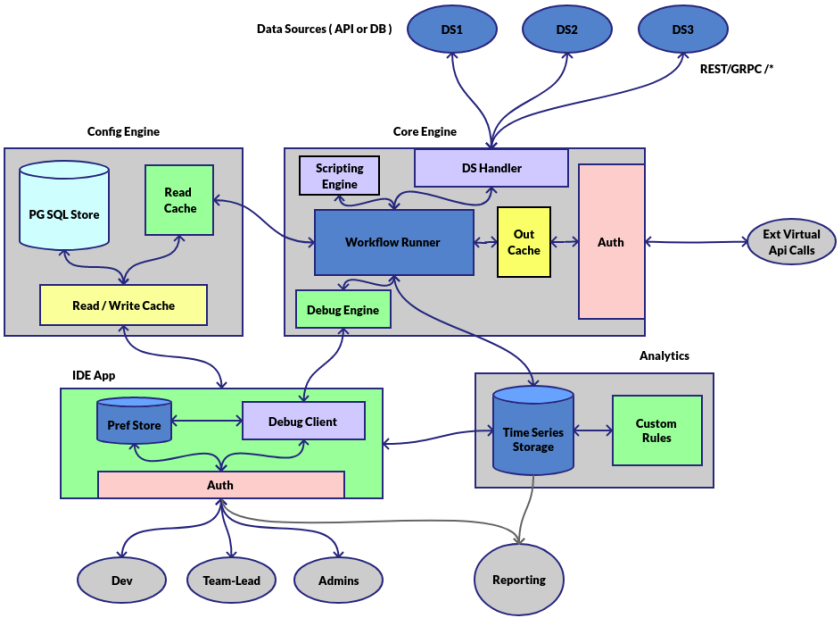
1. In memory SQL server
2. In main Memory for the Workflow Engine

Debugging expressions are easy because there is already debug support in GOJA.

IDE

An IDE (front end) was setup which let's one visually create / update workflows and let it run from the UI. With that creating back end system was a matter of drag and drops. It has integrated debugger with it to debug any existing workflows.

Simplified System Diagram



Usage & Results

Business Adoption

All Mobile API read calls were done via this system. Given the end user can chose what exactly to get from the end point, the payloads were massively reduced, along with the automatic parallisation - the end points were at least 50% faster than their non mobile counterparts in median.

Error handling became much better because of the visual representation in which end point any fault really lie.

Moreover, because of the logging one could early detect in a single source - what are the back end data sources whic are regressing or showing symptoms of fault.

Performance Metric

For 99.999% of the time the system adds 1 msec more time to add to the already existing calls and data processing. This comparison was done against old services which were "pure" that is "manually developed". 99% of the cases the delay is within nanosec ranges.

References

1. Query Languages:
    1. https://en.wikipedia.org/wiki/Yahoo!_Query_Language
    2. https://en.wikipedia.org/wiki/GraphQL
    3. https://en.wikipedia.org/wiki/SQL
    4. https://en.wikipedia.org/wiki/Apache_Pig
2. Workflow Aggregation
    1. https://en.wikipedia.org/wiki/Apache_Pig
    2. https://en.wikipedia.org/wiki/Apache_Airflow
    3. https://github.com/uber/cadence
    4. https://azkaban.github.io
3. Back End for Front End Systems:
    1. https://docs.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends
    2. https://www.mobilelive.ca/blog/why-backend-for-frontend-application-architecture
    3. https://medium.com/mobilepeople/backend-for-frontend-pattern-why-you-need-to-know-it-46f94ce420b0
4. Issues:
    1. Join in GraphQL :
        1. https://stackoverflow.com/questions/51805890/how-to-do-a-simple-join-in-graphql
        2. https://hasura.io/blog/introducing-graphql-joins-for-federating-data-across-graphql-services/
    2. Performance: https://shinesolutions.com/2022/01/04/alleviating-graphql-performance-anxiety/
5. DSLs for Business Processes :
    1. Understanding : https://arxiv.org/abs/1604.05903
    2. Google Path Query : https://arxiv.org/abs/2106.09799
    3. GoJA Engine : https://github.com/nmondal/goja
    4. In Memory MYSQL : https://github.com/dolthub/go-mysql-server
    5. A Prototypic Workflow Engine with Tests : https://github.com/nmondal/flower