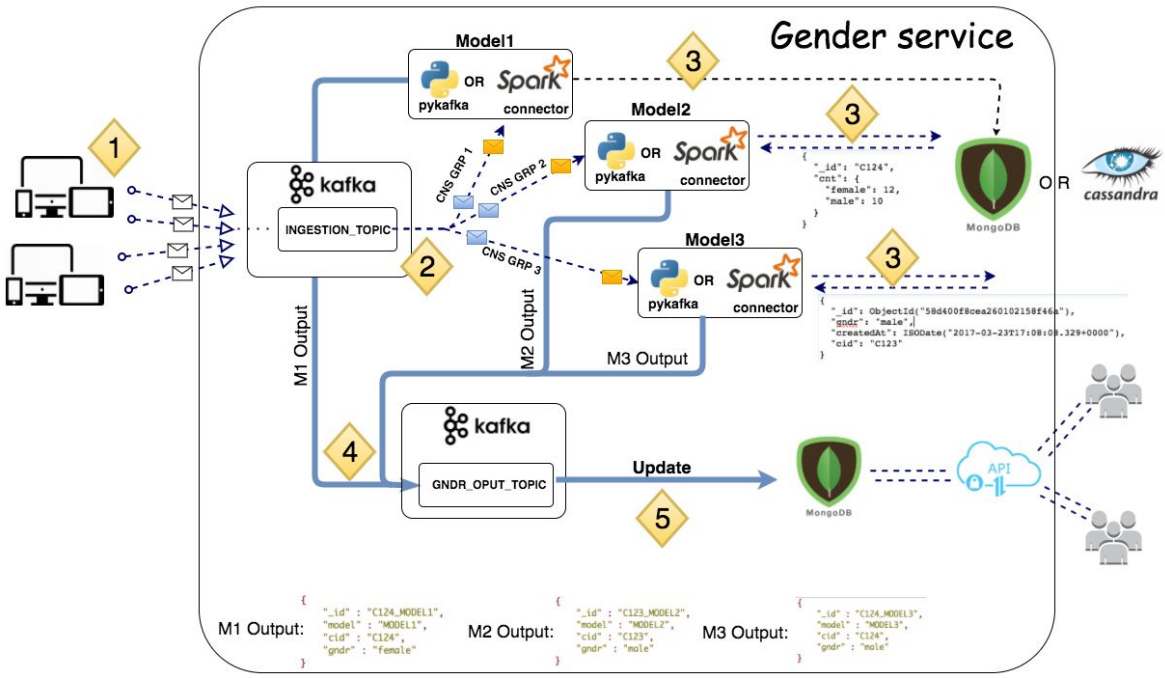
Architecture:

Following is an overview of my design solution for gender service.



Technologies:

Apache Kafka:

kafka used to two times in above architecture

1. To ingest input metrics so that each model can read the data separately. Also new model can read from the first offset(depending on retention policy) in kafka using a new consumer group.
2. To collect all the output results from each model.
3. Defining multiple partitions for ingested topics different copies of a model can run to increase parallelism.

pyKafka wrapper/Spark-kafka connector: I have written a wrapper around pykafka to simplify push(produce) and pull(consume) from kafka. It also serialize and deserialize the dictionary in python. That can also be replaced by spark streaming-kafka connector. Using spark parallelize processing the ingested metric and improve scalability. Using also the my wrapper it is just the matter of launching a copy of a model to process the metrics in parallel.

Mongodb: A database is used to persist the data. I have used Mongdb because I am using that in my current company, and it was easy to test my code with that. But I would definitely consider Cassnadra as the workload is mostly write intensive.

Flask: Flask web service is used to handle the API calls from the end user of the service by querying the outputs from mongo and send back the reply.

Python: Project code is written in python which is my comfortable language. Recently interested in Go. Also love to code more in JAVA. :)

Advantages:

- Kafka is massively scalable. There is no limitation on the volume of metrics.
- Metrics can be pushed from anywhere/any device Regardless of which programing language is used.
- Using different consumer group each new model can start consuming json metrics from kafka
- Easy to test a new model by defining new consumer group and consume the metrics
- Any new model can be added by a data scientist by implementing identify_gender function and using a new consumer group name.

Solution(Code):

Note: to know about the algorithm please read the description of each method

1. Base Class:

- **Class Name: Model**
- **Location: ops/modelbase.py**
- **Description:**
  - **i.** A parent class is defined with an abstract function identify_gender to be implemented by child class.
  - **ii.** Dispatch function in the base class uses the template design pattern to hide the steps. Steps contains consuming messages from kafka, call identify_gender function and produce the output to output topic in kafka.
  - **iii.** The class design helps data scientist to test any new method by just implementing function identify_gender, test and ship it for test.

2. Ingesting Module:

- **Module name: ingest**
- **Location: ops/ingest.py**
- **Description:** This part will ingest the raw metrics(contains cilent_id, gender type visit) from farfetch app or internal webservices to kafka cluster. This can be written in any language. I wrote a simple python module to push(produce) the test random metrics(client id, gender page) to kafka INGEST_TOPIC.

3. SubClass for method1 Implementation:

- **Class Name: Model1**
- **Location: ops/model.py**
- **Description:** The data input which contains (cid, gender) consumed from kafka broker using a consumer group assigned to this model. Since the solution is based on last gender visit, we just need to ingest(produce) the output json(contains gender, cid, model name) to kafka output topic that collects the final result.

4. SubClass for method2 Implementation:

- **Class Name: Model2**
- **Location: ops/model.py**
- **Description:** To be able to get the top gender we need to calculate the gender count over the time. To be able to do that each time a new gender consumed from the kafka broker, we send a request to database and increment the gender. Note that we use function find_one_and_update to ***atomically increment*** the value to avoid race conditions. The counts are persisted in database and this function returns the value of each gender in database. Using that we also *avoid one more call to database*.

{
"_id": male,
"tot": "5",
}

Having the count of each gender returned, we pick the largest one and push the gender output json(contains gender, cid, model name) to kafka output topic that collects the final result.

5. **SubClass for method3 Implementation:**
   - **Class Name: Model3**
   - **Location: ops/model.py**
   - **Description:** In most of the modern data store user can define a *TTL time* for records added to database so that after a certain amount of time, record gets expired and deleted. Using that feature it always makes sure that data is available from the past 7 days in database.

     D1: {
     "_id": ObjectId("58d400f8cea260102158f46a"),
     "gndr": "male",
     "createdAt": ISODate("2017-03-23T17:08:08.329+0000"),
     "cid": "C123"
     }

     Therefore in get_gendr inside identify_gender function, sum of the number of each gender is aggregated and the maximum sum will specify the gender.

     {
     "_id": male,
     "tot": "5",
     }
     {
     "_id": female,
     "tot": "8",
     }

     Finally the gender output json(contains gender, cid, model name) will be produced to kafka output topic.

6. **Output module :**
   - **Module name: dispatch_output**
   - **Location: ops/outputs.py**
   - **Description:** Output module has a *dispatch_output* function. This function consumes the output messages produced by each above models. The output message contains client id, gender and module name. For each client and model we just need to have one record. To achieve that we define a unique identifier using cid and model(cid_model). In *Mongodb* I assigned that to *_id* which is a unique identifier for each record. In other key value databases like *Cassandra* this can be defined as key. therefore on receiving of any new data this module will update the gender according to id module key.
   - Output:

     {
     "_id" : "C124_MODEL3",
     "model" : "MODEL3",
     "cid" : "C124",
     "gndr" : "male"
     }

7. **Webservice:**
    - **Module name: webserive**
    - **Location: webservice.py**
    - **Description:** This will be a simple flask Webservice that receives clientid and model, then sends a database query and gets the gender for that clientid. We can also define a default model to pass to webservice in the case the model has not provided.

Improvement on Current Architecture:

Above architecture is proposed as a base architecture. Following points came straight to my mind that can help to improve the throughputs output of the above service.

- The solution for each method tide to database performance. One way of reducing the latency of each operation that involves database is the use an in memory databases like redis as one layer of caching. For example in method 3 since we need to do a query and aggregate the results it will significantly help.
- Another improvement on output module is to keep the last output gender in memory and update the database only if there is a change in gender for a client. In that case a lot of calls to database can be avoided.

How to run it:

- **Requirements:**
    - Fill database connection info in configuration.py
    - Fill Kafka broker IP in configuration.py
- **Run it:**

1) Run each model
    a) Python model.py --model MODEL1
    b) Python model.py --model MODEL2
    c) Python model.py --model MODEL3
2) Ingest some random data:
    a) Python ingest.py
3) Process outputs:
    a) Python output.py

Deployment:

To deploy, each model and output module will be containerize and pushed to docker hub. On deployed node those containers will be pulled and run inside their own container.

Alternative Architecture:

Alternative solution that came to my mind based on my past experience was relying entirely on AWS infrastructure and service. Amazon have a service called kinesis which is equivalent to Apache Kafka. That service with lambda can resemble the same functionality as first architecture.

Advantages:

- AWS will manage the infrastructure. Less administration hassle for different applications like kafka, spark and Mongodb/Cassandra

- Lambda function is elastically scalable. Based on the number of times the function is called amazon will launch more resources. The higher the requests the more function calls by lambda service.
- Easier deployment: One the function uploaded in lambda and firehose start listening.
- Lambda Function gets triggered automatically as soon as a new metric Ingested into kinesis firehose.
- Following is my proposed alternative architecture: