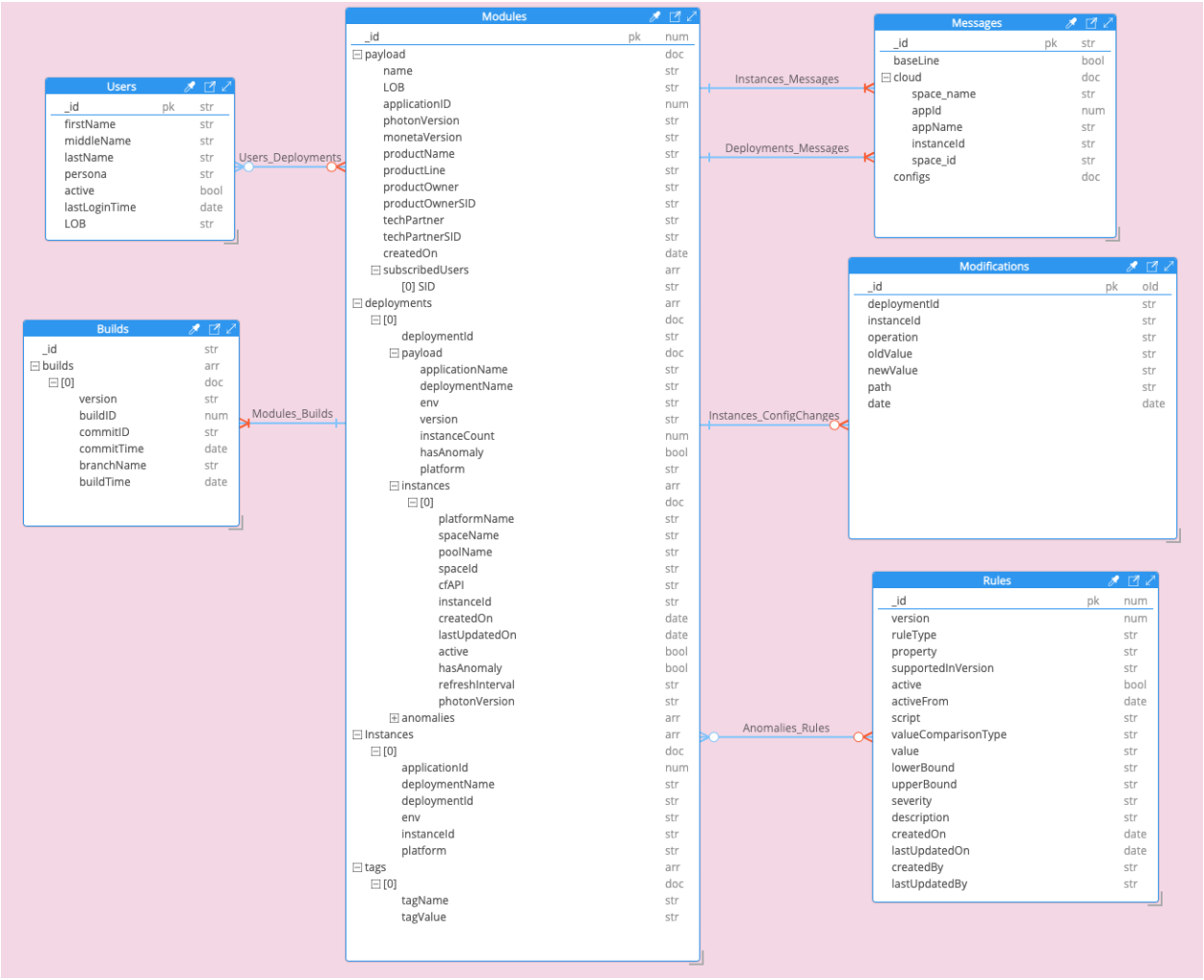

This is the model for Photon Insights I think would make most sense given the information we have at the moment. The main thing I've done is to embed deployment, instance and anomaly information within the module documents. This replaces the module, deployment, module_tag, user_subscription, anomaly and the three environment specific deployment tables in your relational model. I gave a bit of thought about whether it really made sense to go with this level of embedding. The main factors that led me to decide it was the right way to go were:

- The resulting documents look like they will still be pretty small - the test data I generated had an average document size of under 5KB. This was without any simulated anomaly data, but it leaves a reasonable amount of room for that to be added whilst still remaining within an efficient document size (<200KB)
- There's a relatively small number of documents needed in this collection. My test data simulated 4500 modules each with between 1 and 9 deployments, each deployment having between 1 and 3 instances. There were ~56,000 distinct instances in the test set. Even with a ten-fold increase in that data and assuming an average document size of 200KB once anomalies are added, that would still be well under 1GB before storage compression.

- I've assumed individual documents will be updated relatively infrequently i.e. most phone home messages received will be from an already known instance and will not generate any anomalies.

The _id of this collection corresponds to module_id in your relational model (module_id and application_id appear to be used interchangeably in the relational model and there may be instances in the ERD above where the same occurs).

To populate the initial application list page, I've added a "payload" section to the module document that contains the fields needed to fill the entries in the table. This field should be included in a projection clause when retrieving the initial list of modules to avoid returning an excessive amount of data that's not needed to populate the initial view:

```
db.Modules.find({},{"payload": 1, "_id": 0})
```

It looks as though when module documents are initially created, some of the fields in the payload section might be derived from the product and/or seal tables in your relational model rather than from the received phone-home messages. If that's the case, a product and seal collection should be added to be referenced whenever a message is first received from an instance in a new module.

Retrieving the payload section on population of the landing page works well enough for a data set of 4500 modules, but you may wish to revisit returning the entire set if the number of modules grows significantly. A paged approach might be more appropriate in those circumstances, perhaps ordered alphabetically by application name. Another alternative might be to prompt users to do an upfront search to restrict the initial list of modules returned. One other option to consider is to maintain the list of module documents in-memory in your application server and use change-streams to update the list whenever an update occurs on the underlying collection.

Creating a payload section is a good approach for handling fields that need to be returned but on which no actual processing is carried out as it can shield you from code changes if the set of fields required changes. In those circumstances, you would modify the list of fields included in the payload, but as your code is just returning the payload section, it doesn't need any corresponding changes to handle the modifications. There are similar sections in the deployment sub-documents used in a similar manner when the user elects to drill down to a specific module.

The modules document includes a top-level "instances" array listing identifying details of all instances associated with a module - this is used when processing newly received messages to identify if the instance associated with the message has previously been seen.

The deployment sub-documents list the deploymentId, payload fields used to populate the UI when a user drills down into a module, a list of associated instance documents, and a list of associated anomaly documents. The instance sub-documents can be varied according to the deployment platform (GAP, GKP, LOCAL) with the only requirement being that each contains a unique instanceId field. I elected to list anomalies at the deployment level rather than the instance level as I'm assuming a given anomaly is likely to impact all instances in a deployment in most cases. The anomaly sub-documents include an array of the instanceId of each affected instance.


_id: 403
payload: Object
name: "application_403"
LOB: "CCB"
applicationID: 403
photonVersion: "2.6.3-SNAPSHOT"
monetaVersion: "2.6.4"
productName: "Business Banking - SB Lending"
productLine: "SB Lending"
productOwner: "Henry Smith"
productOwnerSID: "HS5678"
techPartner: "Martin Baker"
techPartnerSID: "MB1234"
createdOn: 2022-04-27T19:23:20.000+00:00
403
instances: Array (19)
0: "403:GKP:PROD:deployment_2:instance_1"
1: "403:GKP:UAT:deployment_3:instance_2"
2: "403:LOCAL:PROD:deployment_1:instance_2"
3: "403:GKP:DEV:deployment_1:instance_2"
4: "403:LOCAL:DEV:deployment_1:instance_3"
5: "403:LOCAL:PROD:deployment_1:instance_1"
6: "403:GKP:DEV:deployment_1:instance_3"
7: "403:GKP:UAT:deployment_2:instance_3"
8: "403:LOCAL:PROD:deployment_3:instance_3"
9: "403:LOCAL:UAT:deployment_2:instance_1"
10: "403:GAP:PROD:deployment_2:instance_3"
11: "403:GKP:DEV:deployment_3:instance_2"
12: "403:LOCAL:PROD:deployment_2:instance_3"
13: "403:GAP:PROD:deployment_3:instance_2"
14: "403:LOCAL:UAT:deployment_3:instance_1"
15: "403:LOCAL:PROD:deployment_1:instance_3"
16: "403:GAP:UAT:deployment_3:instance_1"
17: "403:GKP:UAT:deployment_1:instance_1"
18: "403:LOCAL:PROD:deployment_2:instance_1"
deployments: Array (15)
0: Object
deploymentId: "96244901-205b-4b31-a7f3-05922950a2a9"
payload: Object
deploymentName: "deployment_2"
env: "PROD"
platform: "GKP"
version: "2.6.3-SNAPSHOT"
instanceCount: 1
hasAnomoly: false
instances: Array (1)
0: Object
instanceid: "403:GKP:PROD:deployment_2:instance_1"
1: Object
2: Object
3: Object
4: Object
5: Object
6: Object
7: Object

The Messages collection contains the latest raw phone-home message for each instance, with just a handful of modifications; the _id field of the documents are set to the instanceId in the messages, and if a received message is the first seen from a new module, a second copy if the message is save with a “baseLine” field added and set to boolean true (otherwise baseLine is set to false). For the baseLine documents, the literal string ‘baseLine’ is suffixed to the _id value to avoid a unique key constraint violation.

In my test environment, the ‘cloud’ section of messages is significant in being used to identify the module, deployment and instance to which a message belongs. Otherwise the fields in messages can vary - in the ERD I only show a single ‘configs’ field, but this would actually include all the various sections in the received messages.

The sample message we were sent was around 32KB in size and if this is representative, it is a good size for efficient storage and retrieval in MongoDB. Given that, I didn’t see any need to apply any special patterns or optimizations to this collection. After the initial message for an instance is received, my suggestion is to only apply modifications as needed when changes are detected, rather than save the entire new message each time as, I’m assuming, changes between messages will actually be fairly uncommon.


```
  _id: "403:GKP:PROD:deployment_2:instance_1:baseLine"
▸ monetaBoot: Object
▸ applicationConfig: Object
▾ cloud: Object
    appId: 403
    appName: "application_403#deployment_2"
    instanceId: "403:GKP:PROD:deployment_2:instance_1"
    space_id: "GKP"
    space_name: "PROD"
▸ dataSources: Array (1)
▸ features: Array (6)
▸ git: Object
  startTime: "2022-04-27T19:23:20+0000"
▸ unit: Object
▸ adfs_clients: Object
▸ kafka: Object
  schemas: "jpmc.encryption.SecuredData"
▸ application: Object
▸ build: Object
  baseLine: true
```


The Modifications collection is a time-series collection containing details of changes identified between messages received from instances. The date of the message containing the changed data is the date used by the time-series collection to organize storage of the documents, with

instanceId identified as a meta field to further bucket the data. Time series collections efficiently store data for retrieval of a range of documents between two dates in temporal order.

```
date: 2023-11-13T21:21:12.804+00:00
instanceId: "654:GKP:PROD:deployment_2:instance_3"
oldValue: "INFO"
newValue: "WARN"
path: "/applicationConfig/logging.level.root"
operation: "replace"
_id: ObjectId('6552934853c6a0bf94d50cd4')
```

The rules collection is a straight copy of the rules table from your relational model. When applying rules, I would recommend taking each identified modification and running a search against the rules to see if any are triggered by the modification, rather than apply every rule to every value in every message received. The exceptions to this would be on receipt of the first message from a given instance, and whenever a new rule is added or an existing rule modified.

I included the Builds and Users collections as these may be needed by the UI in some circumstances although I didn't see immediately where the data they contained would be used. They may also be needed during the initial creation of a module or deployment. You can review and decide if these are actually necessary.

New Message Processing

When a new message is received, the process I envisioned for handling it is as follows:

1. Parse the JSON and extract the module ID, module name, instance ID, deployment ID, platform and environment from it. Use this data to determine if this is a new module, deployment or instance (I maintained an application side cache of previously seen instances, deployments and modules and only queried the DB if the instance was not found in the local cache.

2. If the message comes from a previously seen instance, retrieve the prior message for that instance from the messages collection:
   `db.Messages.find({_id: <instanceId>})`
   a. Compare the previous message with the current one to get a list of changes (I used the Golang jsondiff library for this - I'm sure there will be equivalent libraries in most languages).
   b. If any changes are identified, apply them to the current message document in the Messages collection. This is made slightly complicated by the fact some of the phone-home message field name include dots, which is generally not

recommended in MongoDB as dots are used to define path levels and it's not possible to escape dots in field names in update statements. However, the $setField expression can be used in an aggregation pipeline to get around this as the following example shows:

```
[
  {
    $match: {
      baseLine: false,
      _id: "instance_1",
    },
  },
  {
    $set: {
      applicationConfig: {
        $setField: {
          field: "java.version",
          input: "$applicationConfig",
          value: "11.0.9",
        },
      },
    },
  },
  {
    $set: {
      "kafka.producers.kafka": {
        $setField: {
          field:
            "sasl.oauthbearer.expected.issuer",
          input: "$kafka.producers.kafka",
          value: "$$REMOVE",
        },
      },
    },
  },
  {
    $merge: {
      into: "PhotonInsights.Messages",
      on: "_id",
      whenMatched: "merge",
    },
  },
]
```

In this example, the required message document is identified in the $match stage, then the “applicationConfig” section of the document is updated first with a copy of the section where the value of field “java.version” has been changed to “11.0.9”, and then section kafka.producers.kafka is replaced with a copy of the section where field “sasl.oauthbearer.expected.issuer” has been removed. Finally, the $merge section writes the updated document back to the database.

c. Once the Message document has been updated, write the individual changes to the Modifications collection.

3. If the message does not come from a previously seen instance, write the message directly to the Messages collection, setting the _id value to the message’s instanceId, and adding a baseLine field set to Boolean false.

4. If the message came from a previously unseen module, create a new module document and add it to the Modules collection.

5. If the message came from a new deployment for an existing module, create a new deployment sub-document for the deployment and push it to the deployments array within the module document, and also add the instanceId to the module’s array of instances. This can be done using an UpdateOne command. My Golang code to do this looks like:

```
if newDeployment {
       filter := bson.M{"_id": appId}
       update := bson.M{
           "$push": bson.M{"deployments": dep, "instances": instanceId},
       }
       res, err := ModulesColl.UpdateOne(context.TODO(), filter, update)
       if err != nil {
           log.Fatal(err)
       }
       if res.ModifiedCount == 1 {
           log.Printf("Updated module document with id: %d", appId)
       } else {
           log.Printf("Failed to update module document with id: %d",
appId)
       }
   }
```

6. If the message came from a new instance for an existing deployment, create a new instance sub-document and add it to the corresponding payload sub-document, also

incrementing the deployment's instance counter and adding the instanceId to the module's array of instances. This is slightly tricky because of the nested arrays, but you can use the array filter options to the UpdateOne command to achieve what's needed. My Go code to do this is:

```
if newInstance {
       filter := bson.M{"_id": appId}
       update := bson.M{
           "$push": bson.M{"deployments.$[elem].instances": inst, "instances": instanceId},
           "$inc":  bson.M{"deployments.$[elem].payload.instanceCount": 1},
       }
       arrayFilters := options.ArrayFilters{
           Filters: bson.A{
               bson.M{"$and": bson.A{
                   bson.M{"elem.payload.deploymentName": deploymentName},
                   bson.M{"elem.payload.env": env},
                   bson.M{"elem.payload.platform": platform},
               }},
           },
       }
       updateOptions := options.Update().SetArrayFilters(arrayFilters)
       res, err := ModulesColl.UpdateOne(context.TODO(), filter, update, updateOptions)
       if err != nil {
           log.Fatal(err)
       }
       if res.ModifiedCount == 1 {
           log.Printf("Updated module document with id: %d", appId)
       } else {
           log.Printf("Failed to update module document with id: %d", appId)
       }
   }
```

7. Process the updates to check for and record anomalies. I'm not exactly sure what your current process for this is, but any anomalies detected should be stored and the deployment level and include a list of impacted instances within that deployment

Searching Message Documents for Configuration Keys With Specific Values

One of the requirements discussed during our previous meeting was the ability to search for deployments and instances with a particular value. Given the variable and dynamic nature of the phone-home messages, this is going to be best achieved using Atlas Search, as traditional indexing to support querying for fields selected on an ad-hoc basis would be difficult if not impossible to implement without breaking the message documents down into key/value tuples.

Atlas search will allow the message documents to be stored in their original form without any significant processing needed to refactor them.

I would recommend creating a search index on the Messages collection. A search for instances with a specific config value would then involve an aggregation pipeline that would look something like this:

```
[
  {
    $search:
      {
        index: "message_search",
        phrase: {
          query: "55.0",
          path: "applicationConfig.java.class.version",
        },
      },
  },
  {
    $project:
      {
        cloud: 1,
        _id: 0,
      },
  },
]
```

Atlas Search is able to handle field names with dots in them. In the example above, the field “java.class.version” in the applicationConfig sub-document in messages results in a path specification of “applicationConfig.java.class.version” when searched via Atlas Search.

The $project stage restricts the returned data to only the ‘cloud’ section of each matched message document, as this contains the data needed to identify the corresponding instance / deployment / application.