# Bayesian Dropout Approximation (BDA) Engine design

## Overview

The BDA Engine acts as a back end for modeling and analysis of data stored in the CESMII Smart Manufacturing Platform. It will exist as a cloud service, served from a Kubernetes cluster instantiated on Microsoft Azure and made available via the MS Azure Marketplace. An instance of the Engine and the K8 cluster on which it runs (together called the BDA Service) is expected to be single-tenant and created (via the Azure Resource Manager's REST API) by an individual instance of a third-party app (TPA) that wishes to make use of the Service's modeling capabilities and which already interacts with the CESMII SM Platform. This instance of the Service will receive and fulfill requests through a RESTful interface.

In response to requests, the Engine will operate in two modes: training mode and inference mode.

In training mode, the Engine will receive and authenticate requests from the TPA to train a model based on a specified set of data from the SM Platform. The Engine will then fetch the data and use it to train a model, and return an encapsulated data package called a Model Kernel Object (MKO), which represents all the information necessary to compute predictions with the model and is expected to be retained by the TPA. Local caching or remote storage of the MKO is a possible additional feature.

In inference mode, the Engine will receive and authenticate requests from the TPA to use a provided (or possibly cached) MKO and provided model inputs to compute samples of possible model outputs. These samples can then be returned to the TPA or used in a variety of ways by the Engine's "post-processing" toolset, with the resulting analysis being returned to the TPA. The tools in this toolset will include capabilities to create histogram images, form probability distribution functions (PDFs), create statistical summaries, integrate certain quantities over PDFs, and visualize model predictions.

## Design Considerations

- TPAs may have high latency or intermittent connections to their instance of the BDA Service.
- The SM Platform service is highly available, low latency, and GraphQL based.
- Instances of the BDA Service are likely not persistent for the useful lifetime of a trained model.
- A typical training dataset from the SM Platform service may be large, $10^5$ - $10^6$.
- The typical number of degrees of freedom in an MKO is order of magnitude $10^3$ - $10^4$.

- Post training, internal data structures in the Engine will be a few hundred MB at the outside, typically much smaller.
- Training task times will be from seconds to multiple hours.
- Inference task times will be from seconds to multiple days of processor time, with typical values being in tens of minutes.
- MKOs should embed provenance and metadata describing what system it represents and which data it is derived from.

## Approach

We propose a hybrid microservices architecture within a Kubernetes cloud service, employing shared memory libraries for highly replicated functionality that involves sharing large or complex data structures.

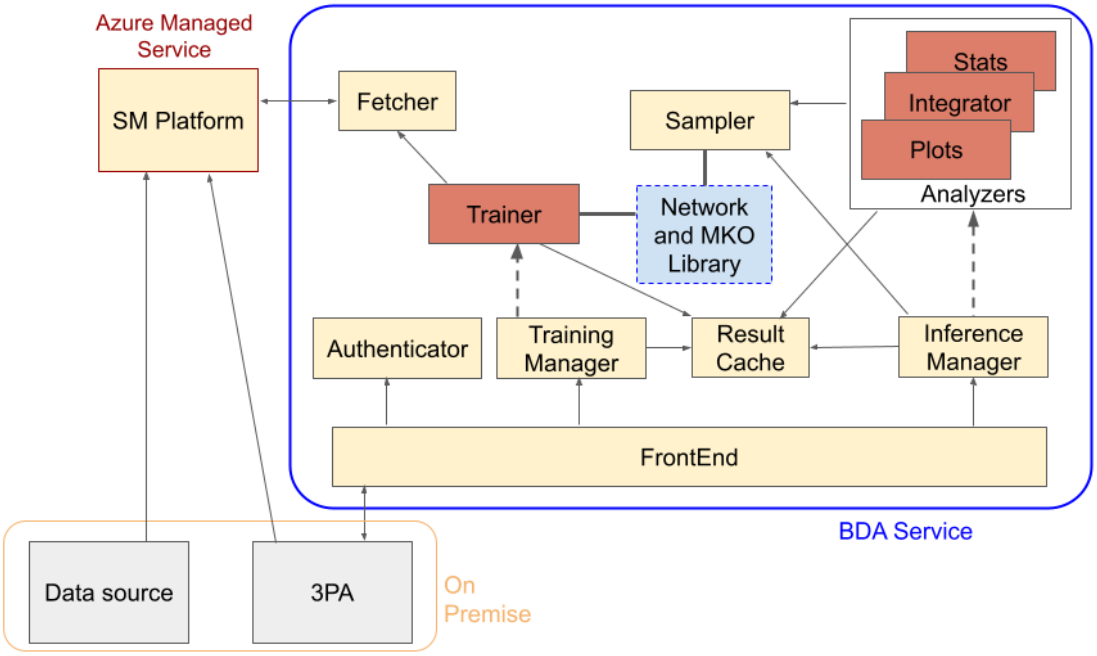


*Figure 1: Architecture of BDA Service and external points of contact. Microservices within the BDA Service may be scaled out to meet demand. Solid arrows indicate RESTful web APIs, dotted arrows indicate separate executables created via system call, and solid thick lines indicate posix function calls to libraries.*

### External communications

Requests to the BDA Service will be through a FrontEnd via REST API, with java web tokens (JWTs) used to manage ongoing authentication and service continuity. Many types of requests will require long-running operations to return results. In these cases, a token, an expected wait

time, and an expiration date will be returned, which can be used to retrieve the forthcoming result.

Communication with the SM Platform will be via its GraphQL interface. Authentication will be via a JWT provided in the TPA's request.

## Internal communications

Internal communications will be via REST APIs, with two major exceptions. First, Trainers and Analyzers will be executed via system call and given control parameters via standard input. Second, as the neural network functionality will be provided by the TensorFlow 2.x framework, TensorFlow functions will be called directly by the Trainer service and by the Sampler service. Subsequently, the TensorFlow data structures will be encoded into and decoded from MKOs directly in memory.

## Individual Services

**FrontEnd** - REST API. The FrontEnd service will listen on an exposed port and take HTTPS requests. It will call the Authenticator service for users' authentication. FrontEnd will call either the Training Manager or the Inference Manager to handle a request.

**Training Manager** - REST API. The Training Manager service sanity checks requests for training a model, determines model hyperparameters, makes estimates for time to complete, and reports errors. It creates instances of Trainers and handles returning MKOs for both short and long operation results. It is responsible for retrieving long-operation MKOs from the Results Cache and returning them to the TPA.

**Trainer** - POSIX executable. Instances are created by Training Manager to instantiate a TensorFlow network and train the network parameters. Employs Fetcher to get training data. Posts MKOs to the Results Cache service and exits.

**Fetcher** - REST API. Fills requests to retrieve data from SM Platform.

**Result Cache** - REST API. Accepts various results from the Trainer and the Generators and holds them until retrieved by Training Manager or Inference Manager or until the expiration date.

**Inference Manager** - Rest API. The Inference Manager service sanity checks requests for inference with a model, makes estimates for time to complete, and reports errors. It creates an instance of the appropriate Generator via system call, passing it an array of MKOs and control parameters, and handles returning inference results for both short and long operation results. It is responsible for retrieving long-operation inferences from the Result Cache and returning them to the TPA.

**Generators** - Posix executables. A set of analysis tools that all operate on model samples. Samples are obtained by passing an MKO and control parameters to Sampler. There may be multiple calls to multiple Samplers. Responsibly written Generators release their Samplers when they are done.

**Sampler** - REST API. Fills requests for obtaining samples, by creating and initializing a network to the state represented by an MKO.  Session based, so that a network can be built and persist across multiple requests for samples.

## Assumptions and Risks

Communication of data specification - There must be a data specification, a path to the piece of equipment and the appropriate time series of data, available to the TPA and communicable to Fetcher.

Authorization is not a bottleneck - The usage of AuthServer must be low enough that a single instance can keep up. If this proves otherwise, a separate database service will have to be created or subscribed to.

Low latency and availability from the TPA cannot be assumed. The overhead associated with the Results Cache is in response to the lack of a good connection to the TPA.

# BDA AuthServer API Design

## Overview

The AuthServer microservice is responsible for authentication and authorization within the BDA engine. Authorization for external resources must be handled by those resources. I'm looking at you, Platform. Only the FrontEnd talks to the AuthServer.

## AuthServer Services

AuthServer has three groups of endpoints: user endpoints, admin endpoints, and FE endpoints. The User and Admin groups are accessible via the User and Admin resources on the FrontEnd. The FE endpoints are only available to the FrontEnd.

| Endpoint Name | Arguments | Required permission | Description |
|---|---|---|---|
| /User/login | username, password | - | Log user in for a session, returns authToken, sets http_only cookie |
| /User/logout | None | User | Log user out of a session |
| /User/set_passwrd | new_password, current_password | User | Change a user password |
| /User/refresh_token | None | user | Issues a new authToken and cookie |
| /Admin/Init | server_secret, username, password | - | Used to create initial admin user at server start up. See Starting A Server for details. |
| /Admin/create_user | username, password | Admin | Creates user with username and password |
| /Admin/delete_user | username | Admin | Removes user |
| /Admin/set_access_list | username, access_list | Admin | Replaces access_list of username |
| /Admin/set_password | username, password | Admin | Replaces password of username |
| /FE/get_permissions | username | FE | Used by FrontEnd to check permission |

# BDA FrontEnd API Design

## Overview

The FrontEnd microservice is the only part of the BDA engine exposed to the external network. It is written in Python (3.6 compatible) and is based on a GNunicorn web server. The job of the FrontEnd is to create modeling sessions, to interact directly with a user, and to serve as an interface with the AuthServer, the Training Manager, and the Inference Manager services. The FrontEnd service can be scaled out in deployment to meet load, although under anticipated usage, a single instance of the microservice should be sufficient.

## FrontEnd Services

FrontEnd endpoints are grouped into *resources*, each under an appropriate top level path component. Resources are *Train, Infer, Admin, User, Logging*, and *Analyze.* Endpoints under each resource are passed directly to their handlers after passing sanity and permission checks.

| Resource Name | Handler | Required permission | Enpoints |
|---|---|---|---|
| /User | AuthServer | User | login, logout, set_password |
| /Admin | AuthServer | admin | create_user, delete_user, set_access_list, set_password |
| /Train | Training Manager | model | See Training Manager API document |
| /Infer | Inference Manager | infer | See Inference Manager API document |
| /Analyze | Inference Manager | analyze | See Inference Manager API document |
| /Logging | FrontEnd | logs | view_logs |

# BDA Inference Manager API

## Overview

The Inference Manager microservice exists to direct the sampling of models represented by MKOs, direct the construction of analyses based on MKOs, and to deliver results back to authorized requestors. Inference Manager is only available through the *Infer and Analyze* resources of the FrontEnd, which performs authorization checks and then passes requests through to Inference Manager.

## Inference Manager endpoints

Endpoints within the *Analyze* resource launch analysis tools from a toolbox, with much of that toolbox to be built upon request by partners. The *sample* endpoint in the *Infer* resource exists as a special case, designed to allow fast, repeated requests for predictions of a trained model and accesses Sampler directly.

| Name | Method | Input Parameters | Required access | Description |
|---|---|---|---|---|
| /Infer/sample | POST | model_MKO, sample_num, data_array | Infer | Request sample_num number of samples, using parameterized MKO model_MKO, with network inputs data_array |
| /Analyze/stat | POST | model_MKO, data_array | Analyze | Return mean and std dev of resulting distribution predicted using parameterized MKO model_MKO, with network inputs data_array |
| /Analyze/Integrator | POST | model_MKO, data_array, function_table | Analyze | Return integral of *function_table* multiplied by PDF predicted at data_array, based on parameterized MKO model_MKO. May also return a claim check for later pickup of results. |
| /Analyze/[matplotlib] | POST | model_MKO, data_array_list, plot_control | Analyze | These endpoints mimic matplotlib functionality, using matplotlib plot_control parameters. They will be implemented in later tasks, as guided by partners. May return a claim check for later pickup of results. |

# BDA Training Manager API Design

## Overview

The Training Manager microservice exists to construct valid MKOs, translate those MKOs into valid input files for the Trainer microservice, and to deliver MKOs back to authorized requestors. Training Manager is only available through the *Train* resource of the FrontEnd, which performs authorization checks and then passes requests through to Training Manager.

Training Manager is the only service capable of creating new MKOs. See *MKO description* for details of the contents and structure of MKOs.

## Training Manager endpoints

| Name | Method | Input Parameters | Required access | Description |
|---|---|---|---|---|
| /create_MKO | POST | model_name, | model_mod | Initialize a neural network with name *model_name*. Returns initialized MKO. |
| /add_data | POST | model_MKO, data_descriptor | model_mod | takes model_MKO and adds data_descriptor (see MKO description), returning updated augmented MKO |
| /add_topo | POST | model_MKO, topo_spec | model_mod | Takes model_MKO and updates topology to topo_spec, retaining any attached data. Returns updated MKO. |
| /delete_param | POST | model_MKO | model_mod | Takes model_MKO and returns updated MKO with parameterization removed, retaining any attached data. |
| /train | POST | model_MKO | model_mod | Takes an augmented, topological MKO and sends it to Trainer to be instantiated and trained. Returns an estimated time to completion and a claim check. |

# BDA Fetcher API Design

## Overview

The Fetcher microservice would be the interface between the BDA Engine and the CESMII Smart Manufacturing Platform. As its name suggests, it provides the API through which client data can be queried and retrieved from the platform via HTTP requests. Although the SM Platform deploys a GraphQL API, the fetcher's API would be a REST API deployed in Python's Flask library. Since this microservice would be a subset of the Engine, it would not be responsible for any of the kubernetes container management and/ or deployment.  However, it would be a contained web instance within this container.

Additionally, as a design consideration, the fetcher in-time would have bi-directional support, allowing the Engine to retrieve and transfer data to and from the platform. Below is a layout of the API structure for this microservice.

## Fetcher query parameters

This list serves as a reference to the possible input parameters that can be passed to the fetcher's APIs.

| Name | Type | Description |
|---|---|---|
| auth_json | JSON object | Containing the required authentication data |
| query_json | JSON object | Containing the required query data |
| filters_json | JSON object | Containing the optional query filter options |
| tag_ids | array | Array of tag_id strings |
| attr_ids | array | Array of attribute_id strings |
| start_time | Date (ISO) | Data retrieval start time |
| end_time | Date (ISO) | End time for data retrieval |
| max_samples | Int | Number of samples to be set aside |
| order | string | Order by id by default |
| first | int | Return the first n values |
| offset | int | Skip  the returned attributes by the offset value |

| | | |
|---|---|---|
| /getcreateddate/ | equipment_id, mko_id | Query the created date for the specified equipment or MKO |
| /getlastupdated/ | equipment_id, mko_id, or any_id | This should return the last date the underlying entity was updated or changed |

The fetcher gets the URI plus other input parameters and parses it into a json string to be added to the graphql query, via the lightweight Flask library. The  response from the SM platform would be a json object to be mangled into the appropriate REST format by the fecher. The Flask web server would only process 1 request; i.e. it is not persistent and wouldn't use sessions.

The Fetcher should be able to match the API/ endpoints to the existing GraphQL API/ endpoints within the Platform. The fetcher's main job is to receive and mangle the request correctly to the appropriate graphql query and similar for the responses.

The unit tests suite for the fetcher is currently being implemented.

Next Step: Implementing the service to check for matching time and matching intervals... the time series data step sizes must match up across all similar dataset. THis might involve fetching the data as is first, then performing some wranginling, filtering out etc within a persistent storage. I am guessing this might be done within the trainer since the fetcher isn't designed for data persistence... look at GetRawHistoryDataWithSamplingRecordFilter!

## auth_json variables

This list represents the snippet of required variables for the fetcher to access the SM Platform. Most of these variables are only required once when establishing the connection to the platform and are a part of the auth_json received from the trainer

| Name | Type | Description |
|---|---|---|
| instanceGraphQLEndpoint | string | Cesmii instance's endpoint url |
| clientId | string | Graphql authenticator username |
| clientSecret | string | Client secret that can be randomly set |
| userName | string | ThinkIQ login username |
| role | string | Clientid associated role within the authenticator |
| fetcher_port | integer | Port address on which the fetcher is running |
| fetcher_port | integer | Port on which the fetcher is running - docker |

## Must-Have-endpoints

| Name | Input Parameters | Description |
|---|---|---|
| /gettimeseries/ | equipment_id, attributes, start_date, end_date | This should return all saved data for a particular equipment, from start date to end date. Attributes and dates are optional |

## Good-To-Have-endpoints

| Name | Input Parameters | Description |
|---|---|---|
| /getcrosssectional/ | equipments, attributes, attribute, date, time | This should return all data from multiple equipment of the same type at a given date, and possibly time. Attributes is optional |
| /getcolumnunit/ | equipments, attributes | Returns an array of matching measurements_units for each attribute presented. It defaults to all units for all equipment specified |
| /getcolumnlength/ | equipment_id, attribute_id | Get the number of saved entries for this measured attribute within the platform |

# BDA MKO Description

## Overview

MKO stands for Model Kernel Object and an MKO exists as a portable container for the necessary information to instantiate a particular neural network in a desired state. MKOs contain descriptions of network structure including topology and activation functions, weight and bias values, and algorithm hyper parameter values.  They are expressible as JSON objects.

## MKO Schema

MKOs are JSON compatible objects. They are valid only if they contain a *modelname* entity. An MKO with a model_structure entity is described as “topographic”. An MKO with a *modelSaved* entity must be topographic to be valid and is described as “parameterized”. An MKO with a *data_descriptor* entity is described as “augmented”. Augmented MKOs may be topographic or not.

Augmented, topographic MKOs may be trained. Parameterized MKOs may be used to make inferences.

```
{
  modelname: "MODELNAME",
  Model_structure (optional): {}
  modelSaved (optional): "encode64 of keras hdf5 model save format"
  data_descriptor (optional): {
    serverName: "",
    serverJWT: "",
    serverURI: ""
    tag_ids: [],
    start_time: [],
    end_time: [],
    max_samples: [],
    offset: [],
  }
  Training_controls (optional): {
    new_iterations: INT,

}
```

| Name | Type | Description |
|---|---|---|
| modelname | string | a user-provided string identifier of 3-64 alphanumeric characters in length |
| modelStructure | JSON | A JSON description of the model topology |
| modelSaved | string | a 64-bit encoded version of an output of Keras’s model.save(save_format=”hd5”) function |

| | | |
|---|---|---|
| serverName | string | The fully qualified domain name of a SMIP instance |
| serverJWT | string | JWT providing access to the described data |
| tag_ids | array | Array of tag_id strings |
| attr_ids | array | Array of attribute_id strings |
| start_time | Date (ISO) | Data retrieval start time |
| end_time | Date (ISO) | End time for data retrieval |
| max_samples | Int | Number of samples to be set aside |
| offset | int | Skip  the returned attributes by the offset value |