APAC Tech Summit
MXNet Gluon Workshop
Machine Learning on Edge

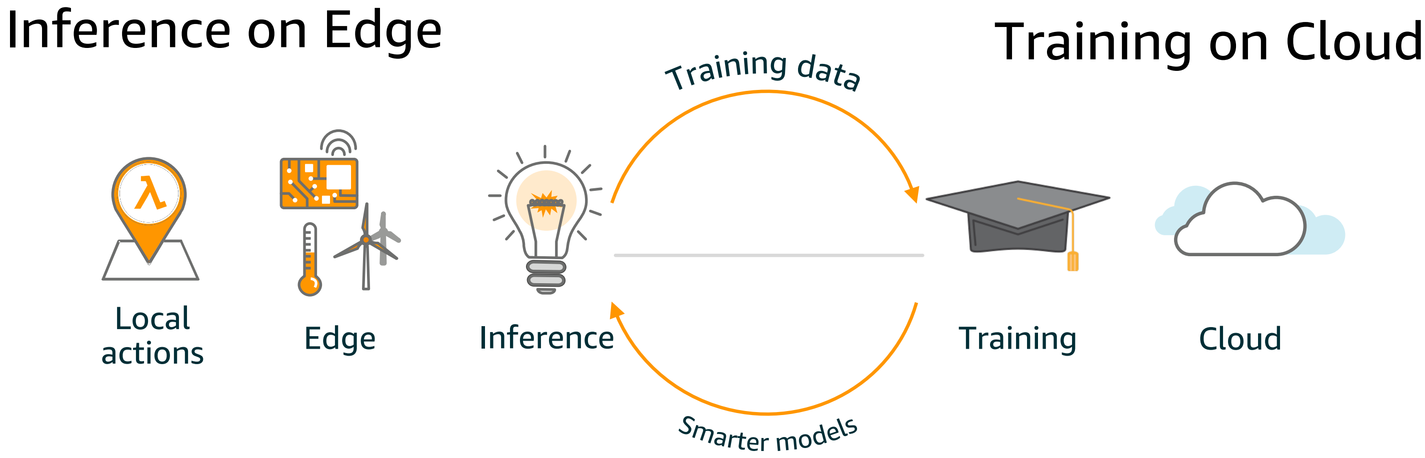
Amazon Web Services Japan

Agenda

- About Edge + ML
- Architecture design considerations
- Bring Your Own Script Overview
- Efficient DL models for edge
- Demo
- Summary

About Edge + ML

What is Edge + ML ?



Benefits of Edge + ML

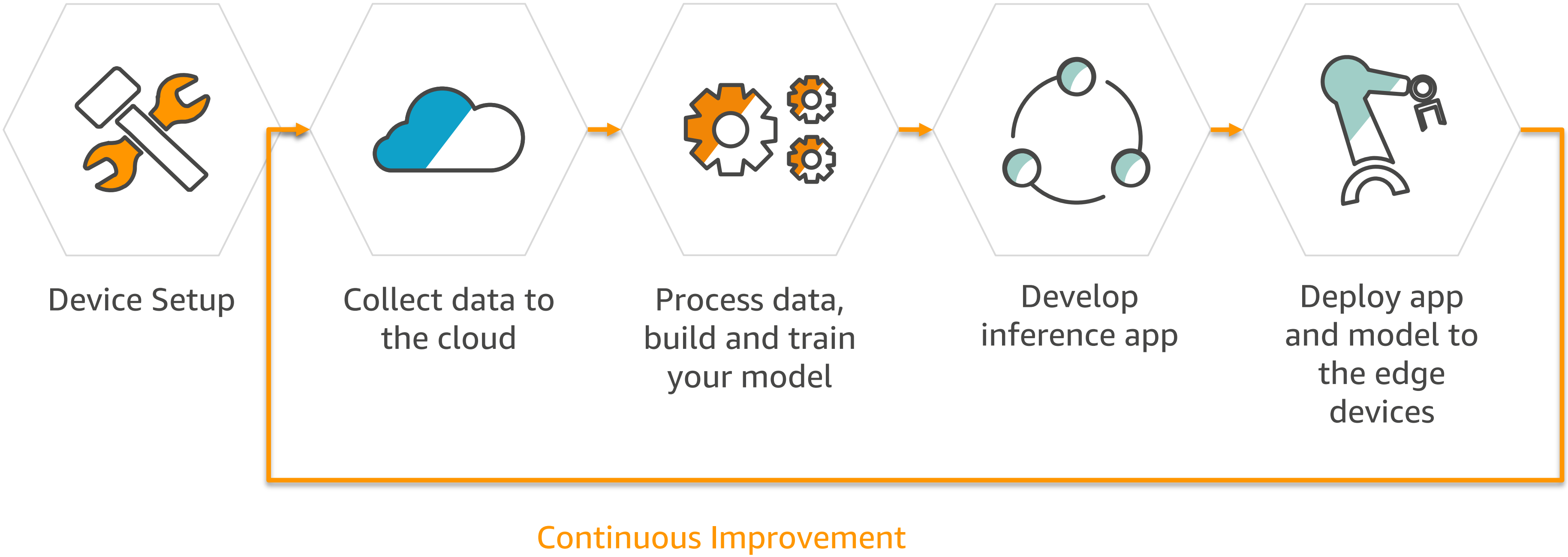
Latency

Bandwidth

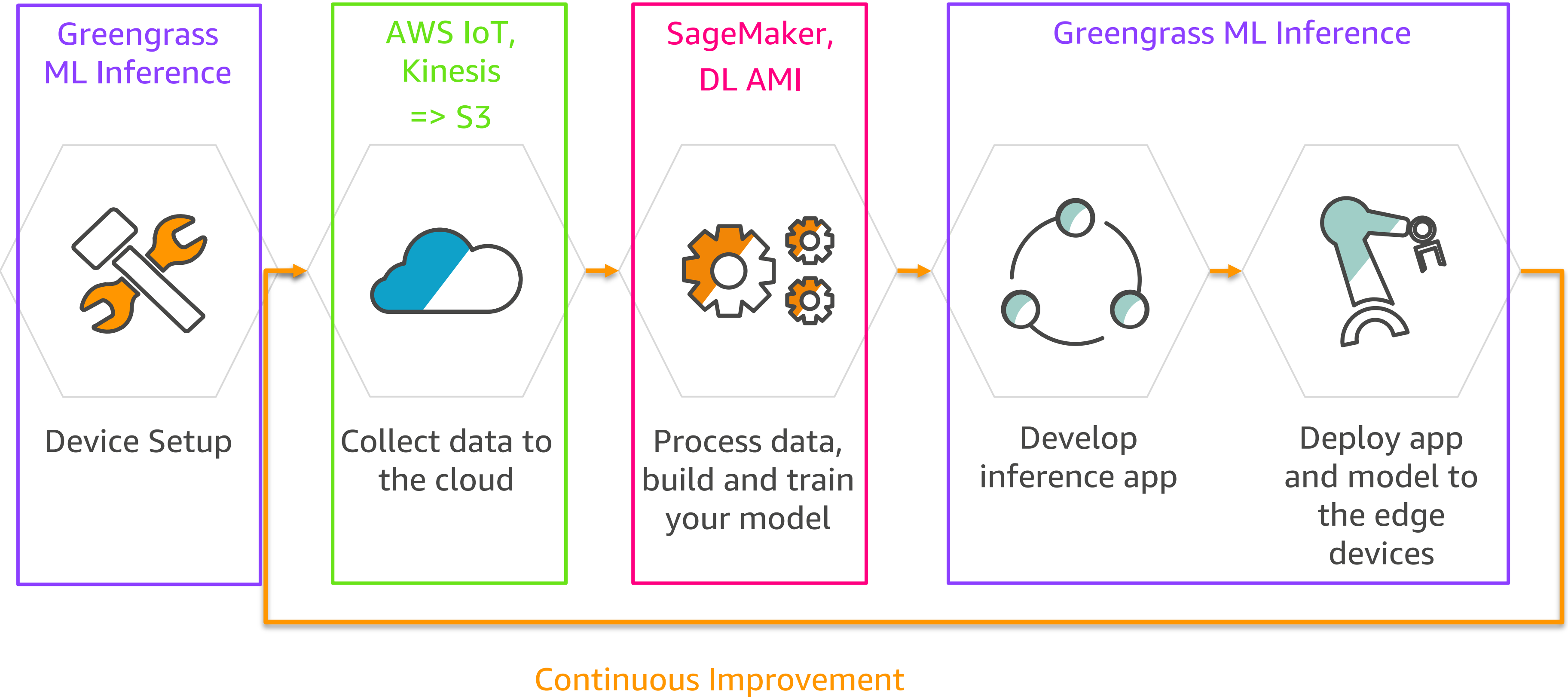
Availability

Privacy

Edge + ML Developing Cycle



Related Services



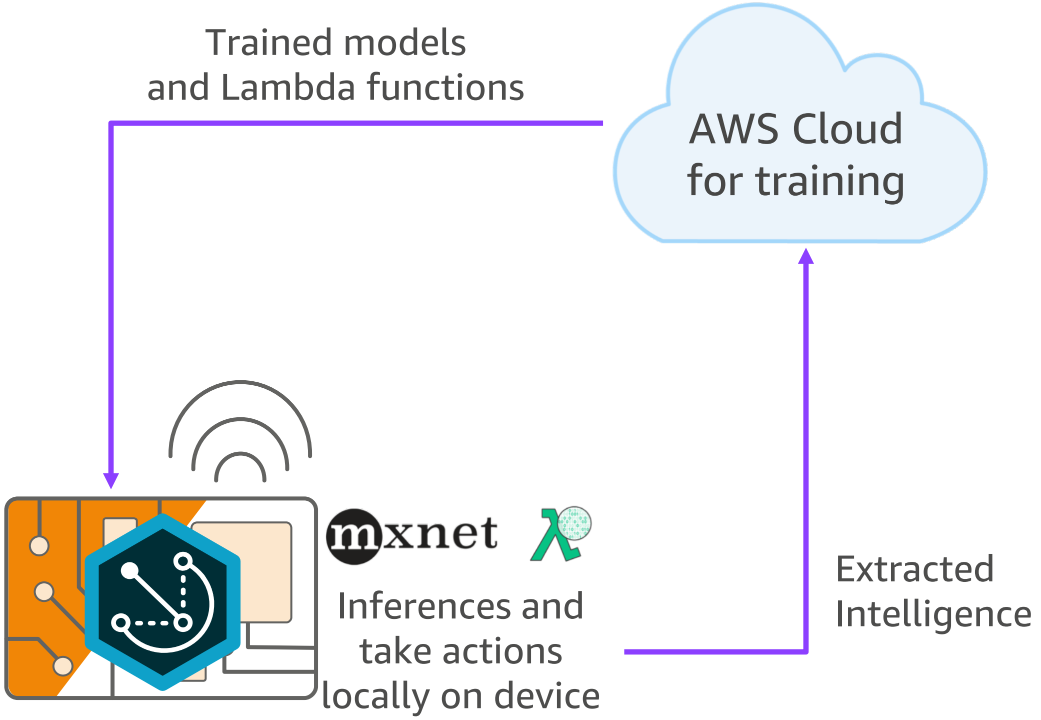
Greengrass ML Inferences

Train in the cloud

- Massive computing power
- Large repository of data

Inference at the edge

- Low latency
- bandwidth saving
- regulation/privacy
- reliability



Amazon SageMaker

Pre-built notebooks for common problems

Built-in, high performance algorithms

One-click training

Hyperparameter optimization

One-click deployment

Fully managed hosting with auto-scaling

BUILD

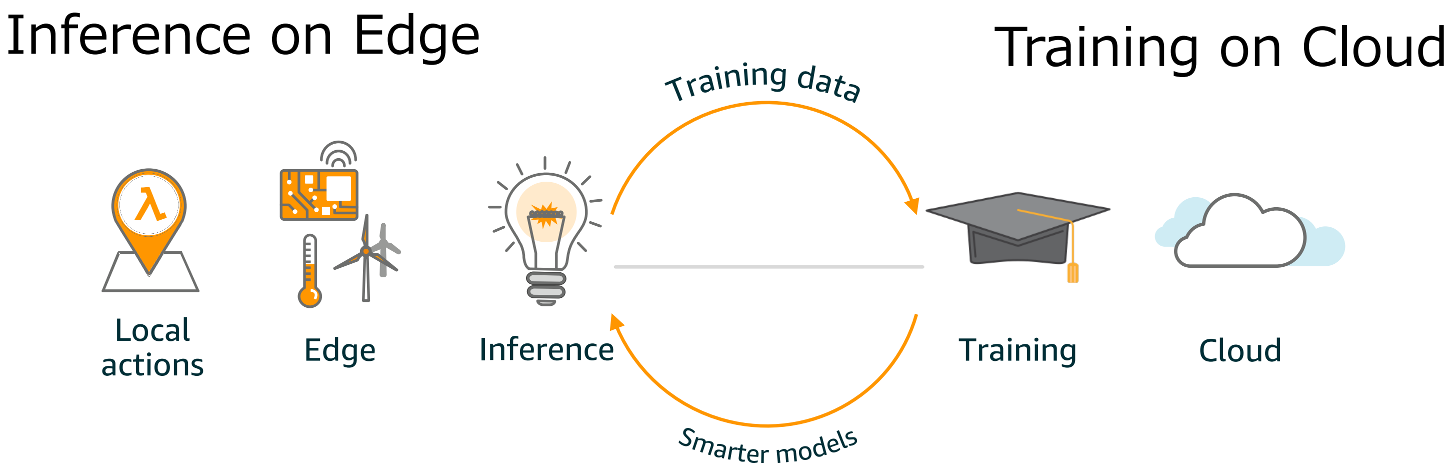
TRAIN

DEPLOY

Edge + ML Use Cases

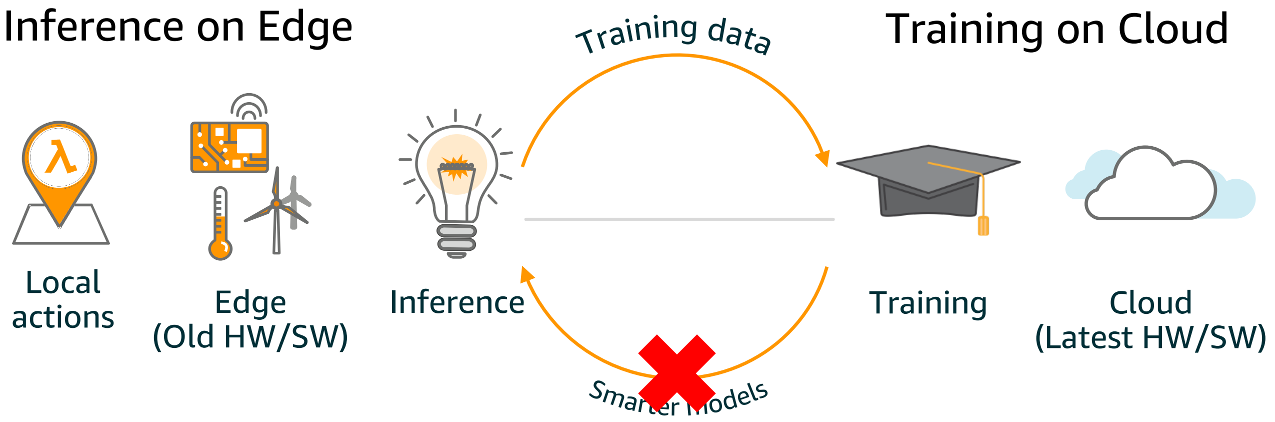
Architecture Design Considerations

Basic Edge + ML Architecture



Edge Considerations

- Device lifecycle
  - The lifecycle of device tends to be long.(e.g. 5 years)
    The latest technology may not be available.(e.g. OS/SW)
    ⇒Enable feature enhancement on Cloud



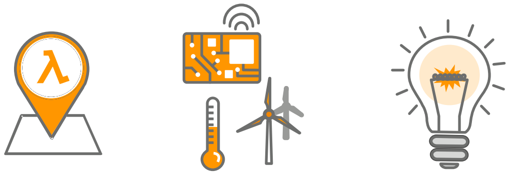
New generation ML model may not be executable on Edge

Edge Considerations

- HW Spec
  - High spec HW may not be available.
    Because of the HW cost/may need to use existing HW
    ⇒ Enable feature enhancement on Cloud
    Consider using lightweight ML model

PoC Environment

(limited number of devices)

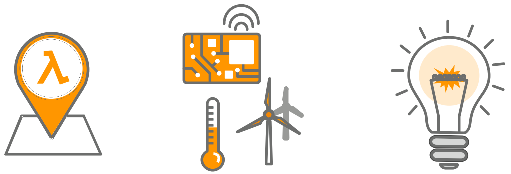


Local actions | Edge (High Spec HW) | Inference

Poor inference performance (e.g. below required throughput)

Production Environment

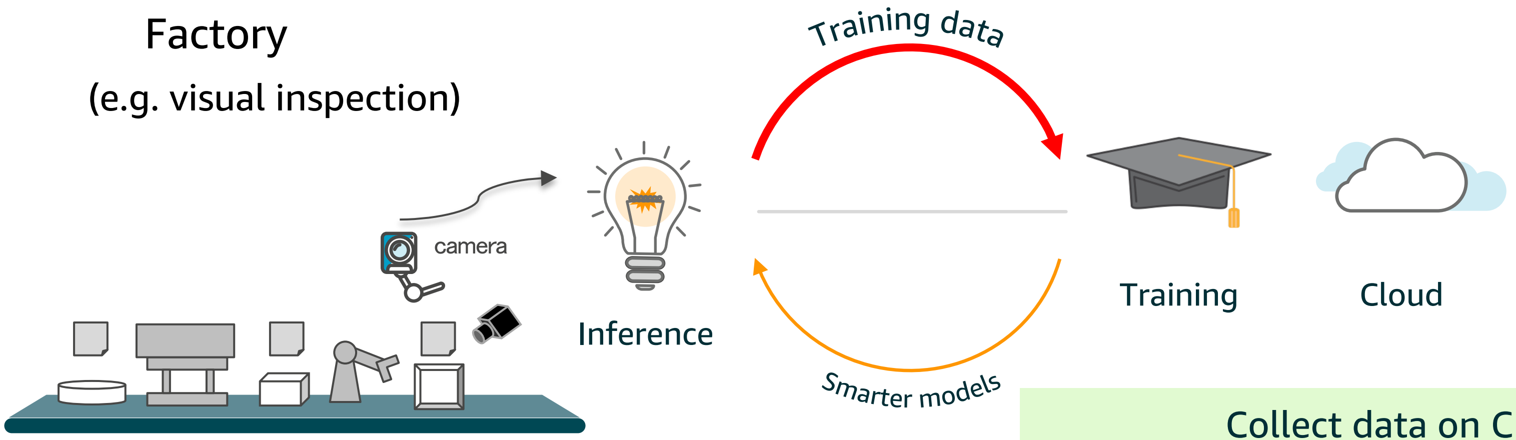
(e.g. 10,000 devices)



Local actions | Edge (Low Spec HW) | Inference

Model Lifecycle Considerations

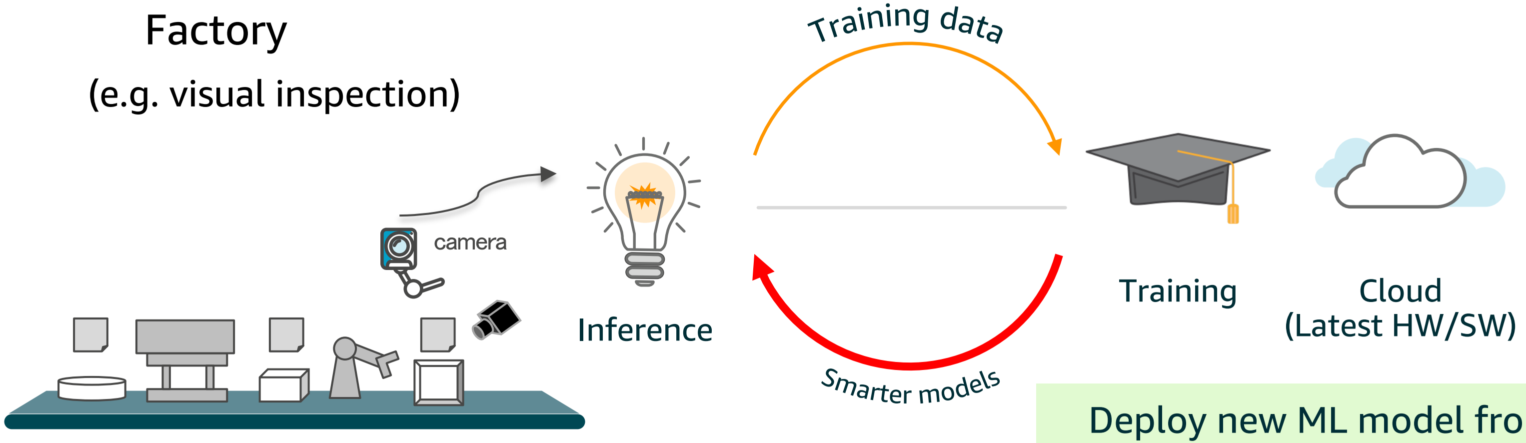
- ML model performance monitoring
  - Inference on Edge is not always correct.
    ML model performance monitoring is necessary.
    ⇒ Collect data and inference result on Cloud to monitor model performance



Collect data on Cloud
Evaluate ML model performance regularly

Model Lifecycle Considerations

- ML model deployment on Edge
  - ML model updates needed.
    e.g. additional classification class, performance improvement
    ⇒ Need Cloud -> Edge model deployment mechanism.



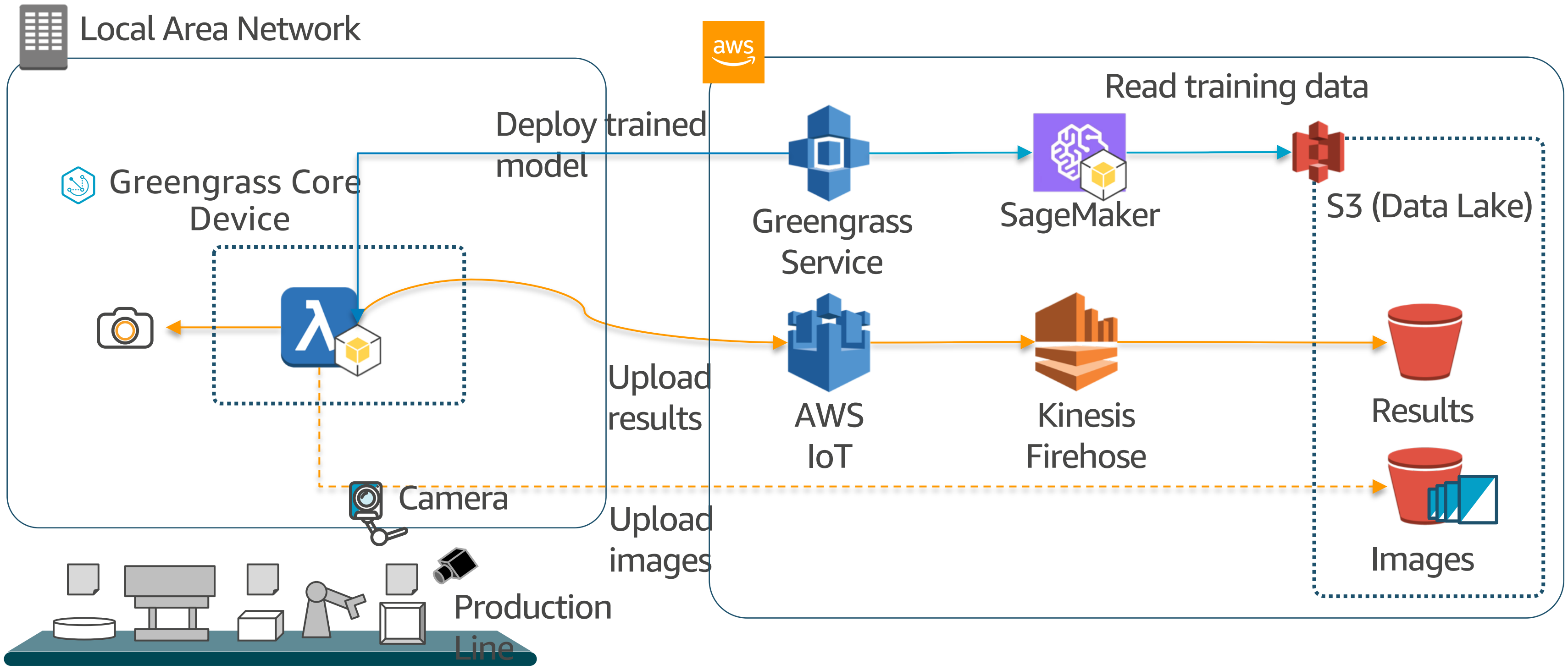
Deploy new ML model from Cloud to Edge

Architecture Design Considerations Summary

- Edge
    - Consider HW lifecycle and production HW spec to design the architecture. Enable feature enhancement on Cloud. Consider using lightweight ML model.
- Model lifecycle
    - ML model performance monitoring is always needed even when inference task is executed on Edge.
    - ML model updates mechanism needed

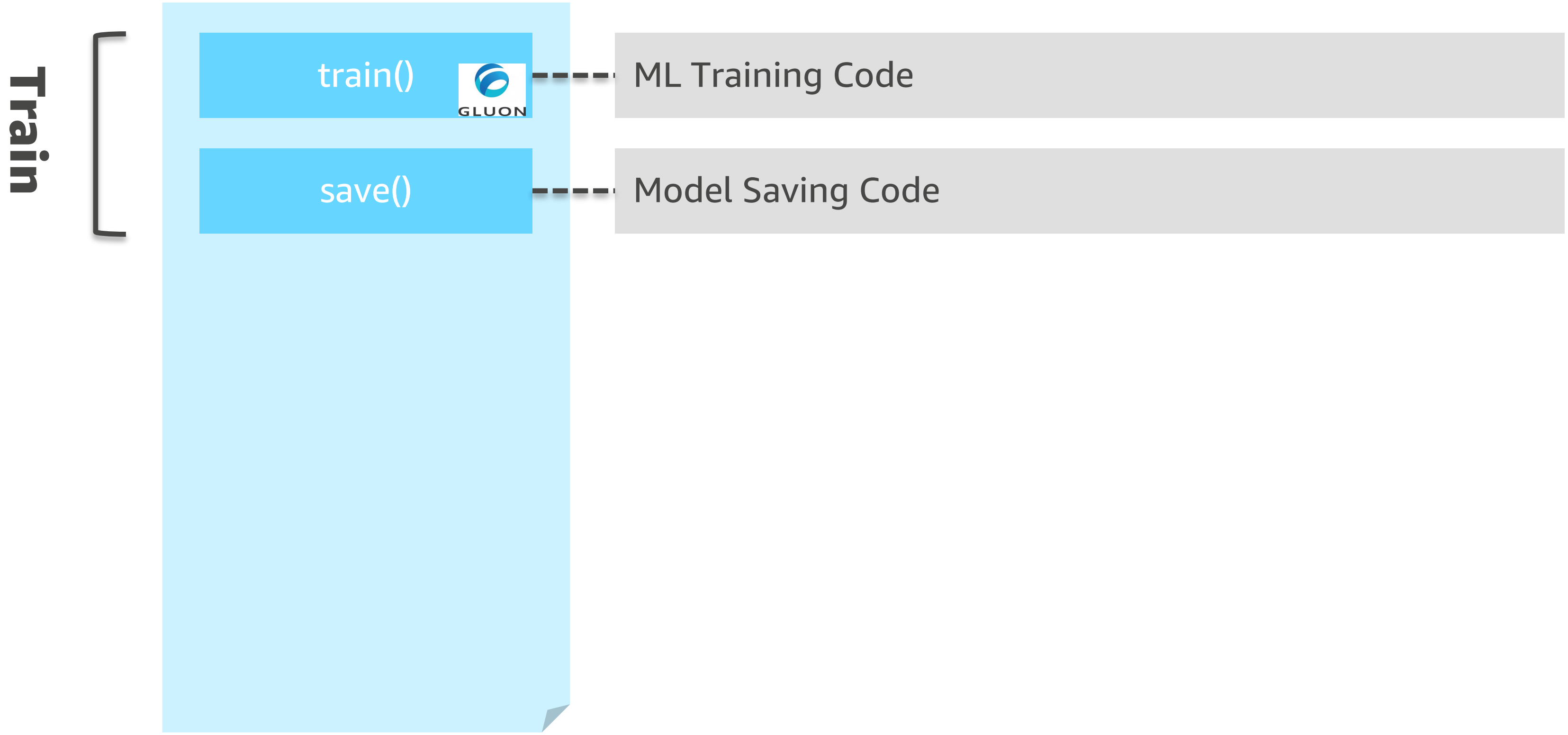
Architecture Example

e.g. Factory visual inspection



SageMaker Bring Your Own Script Overview MXNet

MXNet – Functions in Your Script



MXNet – Functions in Your Script

Train

train()	ML Training Code
save()	Model Saving Code

Inference

model_fn()	Model Loading Code
transform_fn()	Transform input data into prediction result

MXNet – `train` function

- Put your code for ML training into `train()` function
- SageMaker provides information about training environment
- Returns trained MXNet Module API `module` object or Gluon API `net` object

```
def train(hyperparameters, input_data_config, channel_input_dirs, output_data_dir,
          num_gpus, num_cpus, hosts, current_host):
    pass
```

```
# Only work with hyperparameters and num_gpus, ignore all other hyperparameters
def train(hyperparameters, num_gpus, **kwargs):
    pass
```

MXNet – `train` function Parameters

- **hyperparameters (dict[string,string]):** Hyperparameter for ML Training

- **input_data_config (dict[string,dict]):** SageMaker TrainingJob InputDataConfig object

- **channel_input_dirs (dict[string,string]):** Directories of training data
- **output_data_dir (str):** Directory to put checkpoint files

- **num_gpus (int):** # of GPUs
- **num_cpus (int):** # of CPUs

- **hosts (list[str]):** List of host names for distributed training
- **current_host (str):** Name of the training instance

MXNet – `train` function Example

```python
def train(channel_input_dirs, hyperparameters, hosts, num_gpus, **kwargs):
    # SageMaker passes num_cpus, num_gpus and other args we can use to tailor training to
    # the current container environment, but here we just use simple cpu context.
    ctx = mx.cpu()

    # retrieve the hyperparameters we set in notebook (with some defaults)
    batch_size = hyperparameters.get('batch_size', 100)
    epochs = hyperparameters.get('epochs', 10)
    learning_rate = hyperparameters.get('learning_rate', 0.1)
    momentum = hyperparameters.get('momentum', 0.9)
    log_interval = hyperparameters.get('log_interval', 100)

    # load training and validation data
    # we use the gluon.data.vision.MNIST class because of its built in mnist pre-processing logic,
    # but point it at the location where SageMaker placed the data files, so it doesn't download them again.
    training_dir = channel_input_dirs['training']
    train_data = get_train_data(training_dir + '/train', batch_size)
    val_data = get_val_data(training_dir + '/test', batch_size)

    # define the network
    net = define_network()

    # Collect all parameters from net and its children, then initialize them.
    net.initialize(mx.init.Xavier(magnitude=2.24), ctx=ctx)
    # Trainer is for updating parameters with gradient.

    if len(hosts) == 1:
        kvstore = 'device' if num_gpus > 0 else 'local'
    else:
        kvstore = 'dist_device_sync' if num_gpus > 0 else 'dist_sync'
```

MXNet – `save` function for **Module** API

- SageMaker will invoke `save` function with return-value of `train`
- Default implementation works with MXNet Module API `Module` object
- You should define model serialization logic if
  - `train` function returns Gluon API net object or
  - special processing is needed

```
def save(model, model_dir)
```

  - model_dir : directory to save model

MXNet – **save** function for **Gluon** API

- Default save function is <u>NOT</u> compatible with Gluon API net object
- net **object's** save_params() **method does serialization of parameters**

```
num_hidden = 256
num_outputs = 1
net = gluon.nn.Sequential()
```

-
```
filename = "checkpoints/testnet.params"
net.save_params(filename)
```

```
def save(net, model_dir):
    # save the model
    net, vocab = net
    y = net(mx.sym.var('data'))
    y.save('%s/model.json' % model_dir)
    net.collect_params().save('%s/model.params' % model_dir)
    vocab_to_json(vocab, '%s/vocab.json' % model_dir)
```

MXNet – `model_fn` function

- Loading trained model
- Default `model_fn` function provided for MXNet Module API model
- You should write your `model_fn` for Gluon API model

- **model_dir** : directory where your model files and sub-directories, saved by `save`, have been mounted

```python
def model_fn(model_dir):
    """
    Load the gluon model. Called once when hosting service starts.

    :param: model_dir The directory where model files are stored.
    :return: a model (in this case a Gluon network)
    """
    symbol = mx.sym.load('%s/model.json' % model_dir)
    outputs = mx.symbol.softmax(data=symbol, name='softmax_label')
    inputs = mx.sym.var('data')
    param_dict = gluon.ParameterDict('model_')
    net = gluon.SymbolBlock(outputs, inputs, param_dict)
    net.load_params('%s/model.params' % model_dir, ctx=mx.cpu())
    return net

```

```python
def model_fn(model_dir):
    """
    Load the gluon model. Called once when hosting service starts.

    :param: model_dir The directory where model files are stored.
    :return: a model (in this case a Gluon network)
    """

    net = models.get_model('resnet34_v2', ctx=mx.cpu(), pretrained=False, classes=10)
    net.load_params('%s/model.params' % model_dir, ctx=mx.cpu())
    return net
```

MXNet – transform_fn

- Transforming input data into a prediction result
- Default transform_fn implementations that work withGluon and Module models
- If you provide transform_fn in your hosting script, it will be used to handle the entire request. You don't need to provide any other request handling functions (input_fn, predict_fn, or output_fn). If you do provide them, they will be ignored.

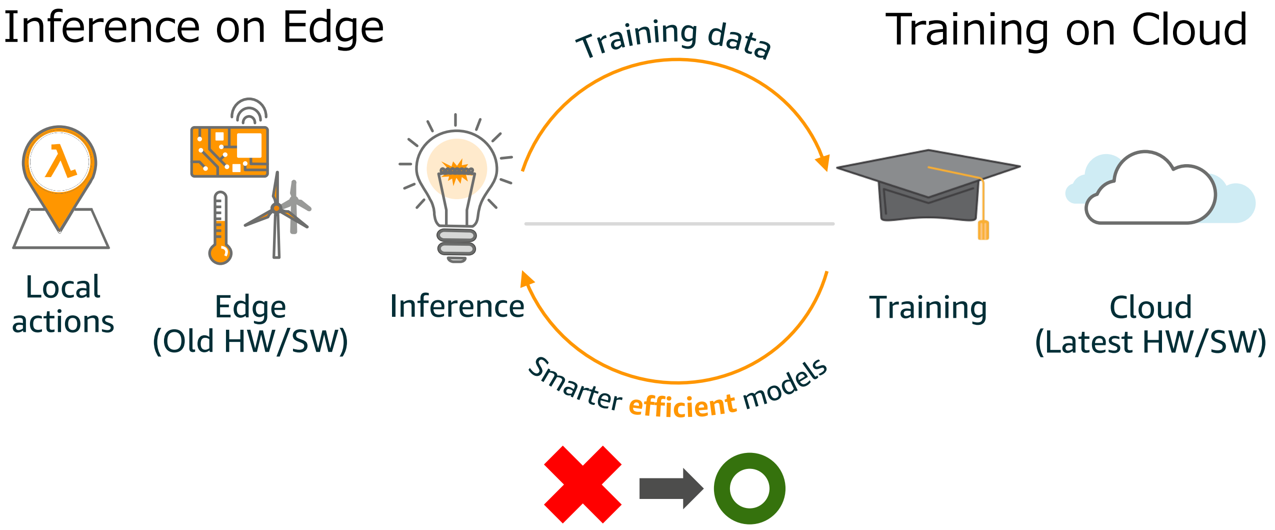
```
def transform_fn(model, input_data, content_type, accept):
    """
    Args:
        - input_data: The input data from the payload of the
          InvokeEndpoint request.
        - content_type: The content type of the request.
        - accept: The content type from the request's Accept header.

    Returns:
        - (object, string): A tuple containing the transformed result
          and its content type
    """
```

Efficient deep learning model for edge

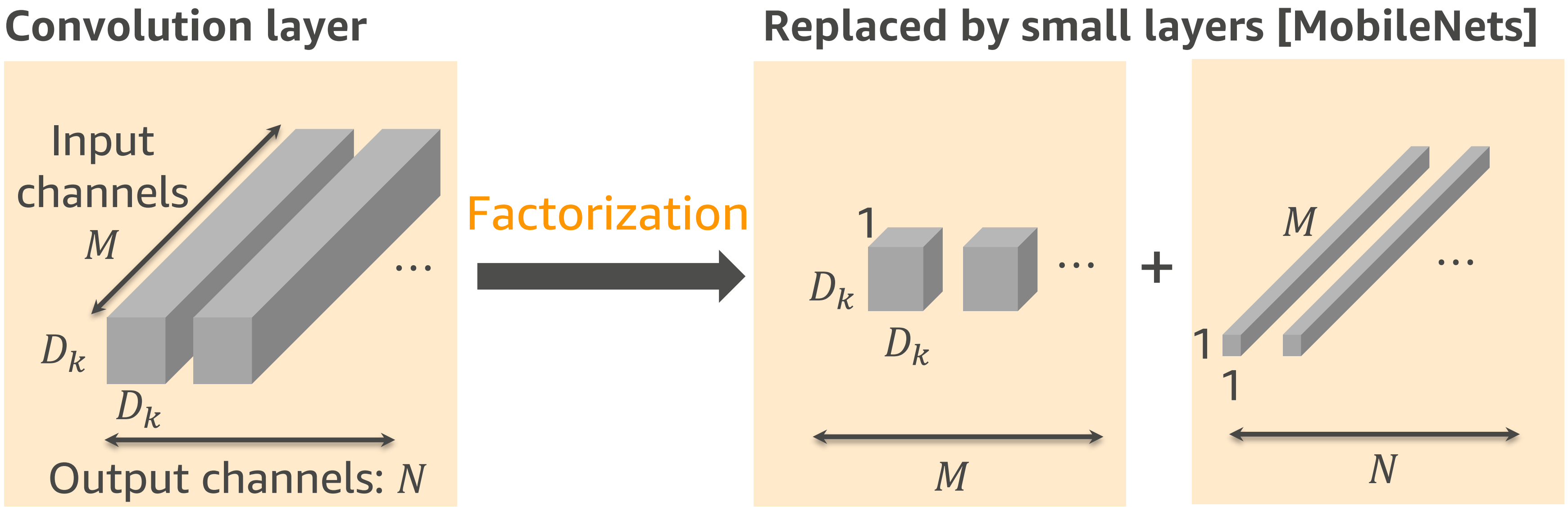
Motivation of efficient DL models

While modern huge DL models cannot work at an edge, state-of-the-art efficient DL models enable to run at edge with high accuracy.



Trend of efficient DL models

Factorization of convolution layers that are computationally demanding [SqueezeNet, 16'], [MobileNets, 17'], [ShuffleNet, 17'].

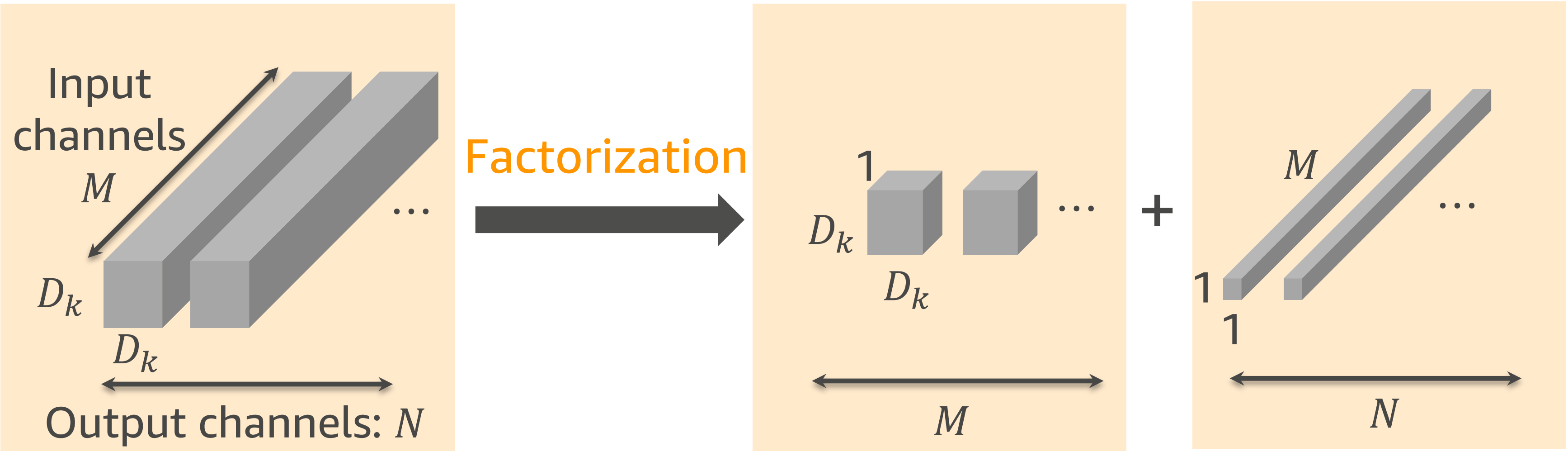


Computational cost for $D_F \times D_F$ image

$$D_k D_k M N D_F D_F \xrightarrow{\frac{1}{N} + \frac{1}{D_K^2}} D_k D_k M D_F D_F + M N D_F D_F$$

Convolution layer

Replaced by small layers [MobileNets]



Framework supporting MobileNets

Models with and without pretraining are available.

- **MXNet** example is provided on github official repository.
- **Gluon/GluonCV** supports in model_zoo.
- **Tensorflow** supports in TF-slim library.
- **Keras** supports in keras.applications.mobilenet.MobileNet

(We can find source codes for Pytorch and Chainer on github.)

Demo (10 min.)

Summary

- For machine learning at edge, we need to care device lifecycle, HW performance, and model monitoring/updating.
- MobileNets is efficient and supported by DL frameworks, which is of use for machine learning at edge.

Thank you!