Machine Learning Platform: Deploying and Managing Models in the CERN Control System

Jean-Baptiste de Martel - Nico Madysa, Roman Gorbonosov, Verena Kain

01/07/2022 – BE Seminar

Summary

- **System overview**
  - Motivation
  - Objectives
  - Workflow
  - Continuous retraining
- **Standalone deployment prototype**
- **MLP in practice**

ML for accelerator controls - motivation

- **Some accelerators / processes difficult to model with means available in CCC -> no online analytical model available**
  - Space charge dominated dynamics in LINACs
  - Transmission optimization during transition crossing
  - Collimator alignment, septa with many degrees of freedom
  - Multi-turn injection with accumulated intensity diagnostics only
- **Usually solved by trial & error, (semi-)manual scans**
  - Time consuming
  - Depends on experience of operator
- **ML techniques outperform previous techniques and open new doors to automation**
- **But the approach today is ad hoc and heterogeneous**

Finding a compromise

Volatile world of ML

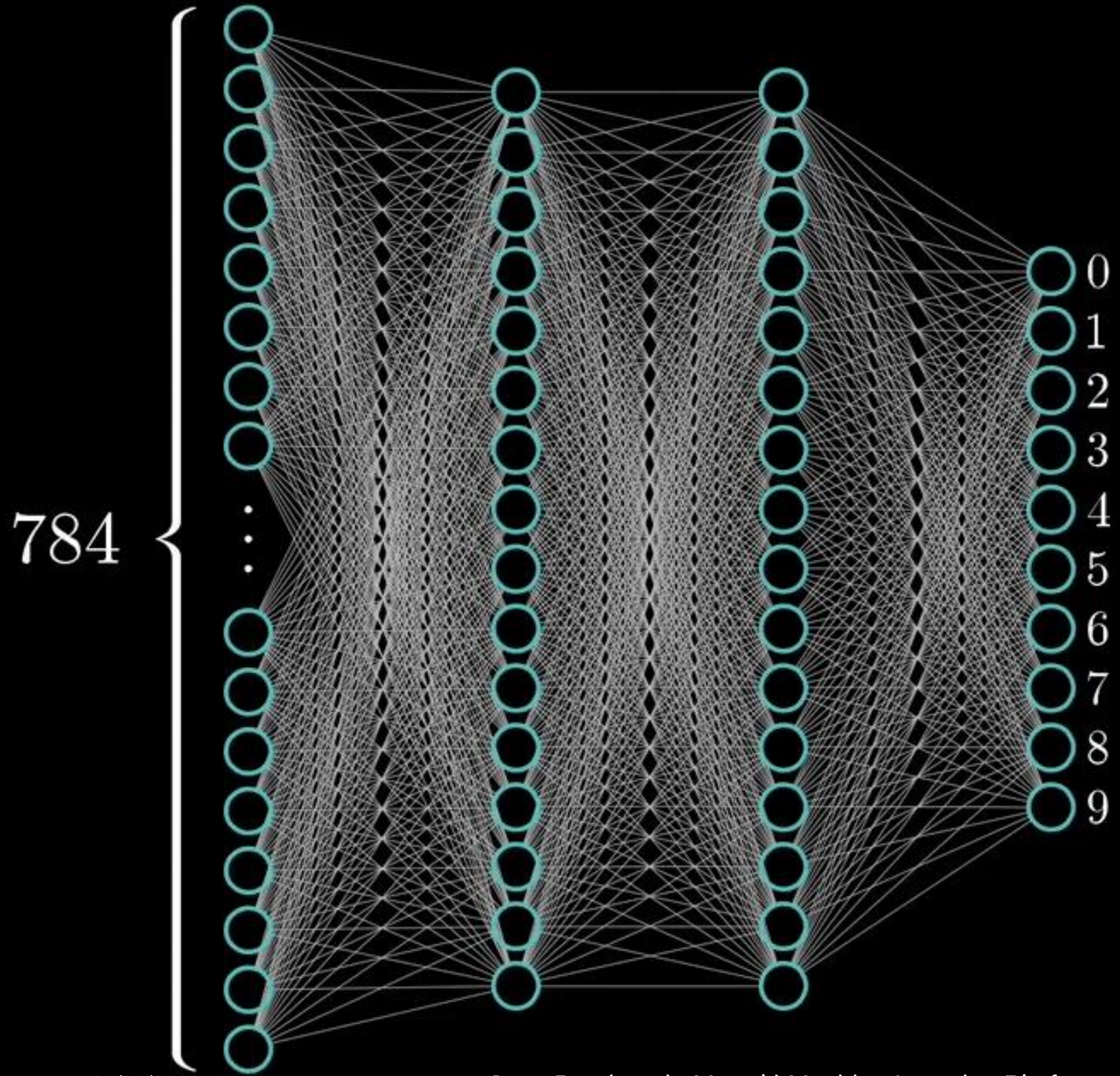
- Code needs to run once
- Bleeding edge technology
- Used to own tools and comfort, cloud services
- Maintainability is not always the main concern

Reliable world of accelerator controls

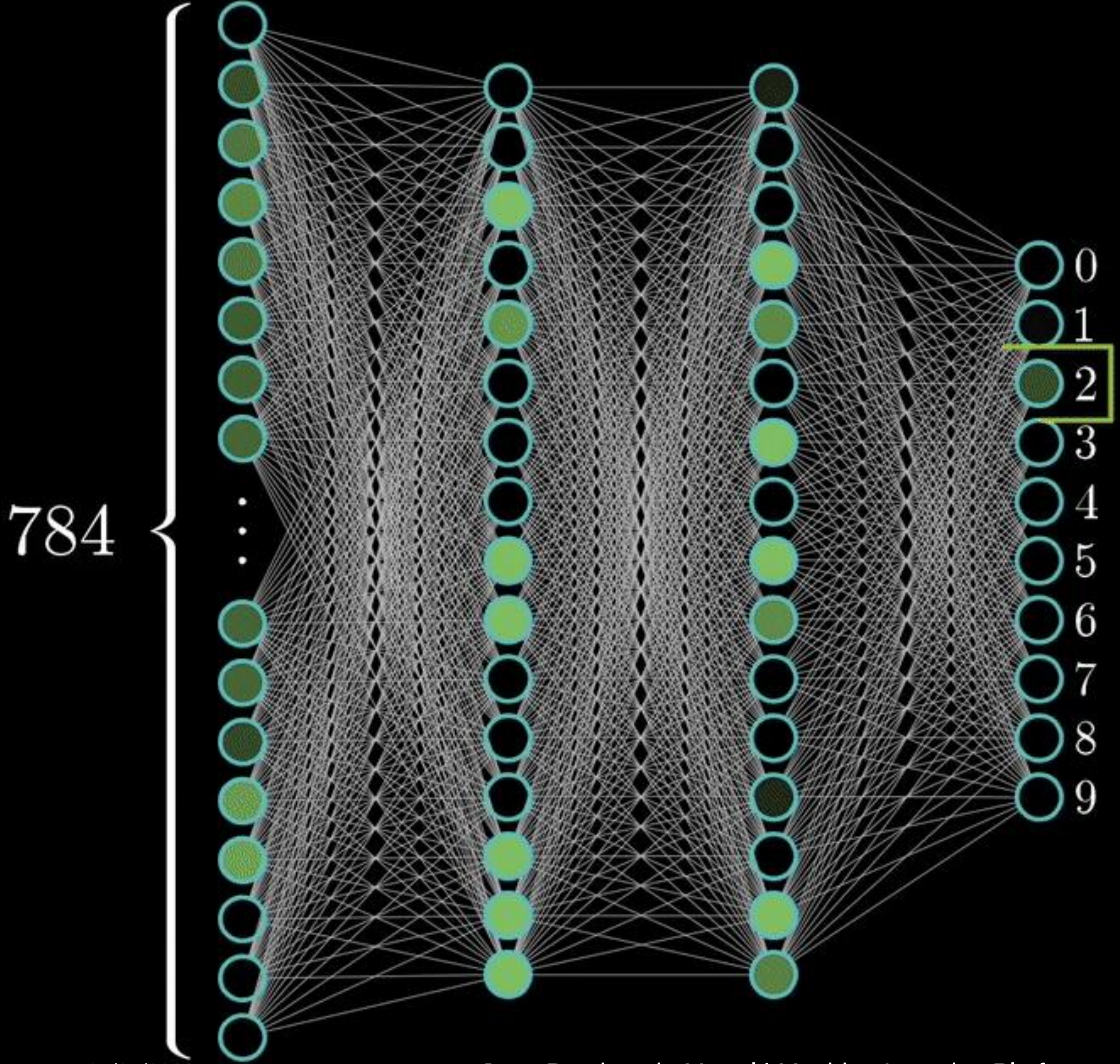
- Need to run reliably 24/7/365: need reproducibility, robustness, traceability
- Use highly reliable, battle-tested tools
- Constraints of the technical network: no internet access, restricted tooling, security precautions
- Standardize and unify to minimize maintenance

Objectives

- **Provide a common approach to storage, versioning, deployment and usage of models**
- **Accelerate and simplify the model lifecycle by abstracting infrastructural concerns**
- **Fulfill the specific needs of the accelerator control system**
  - Reliability
  - Traceability
  - Security
  - Standardization
- **Stay out of the user's way**
  - Minimize constraints on model developer's workflow
  - Avoid constraints on choice of tools



Source: https://www.youtube.com/watch?v=aircAruvnKk

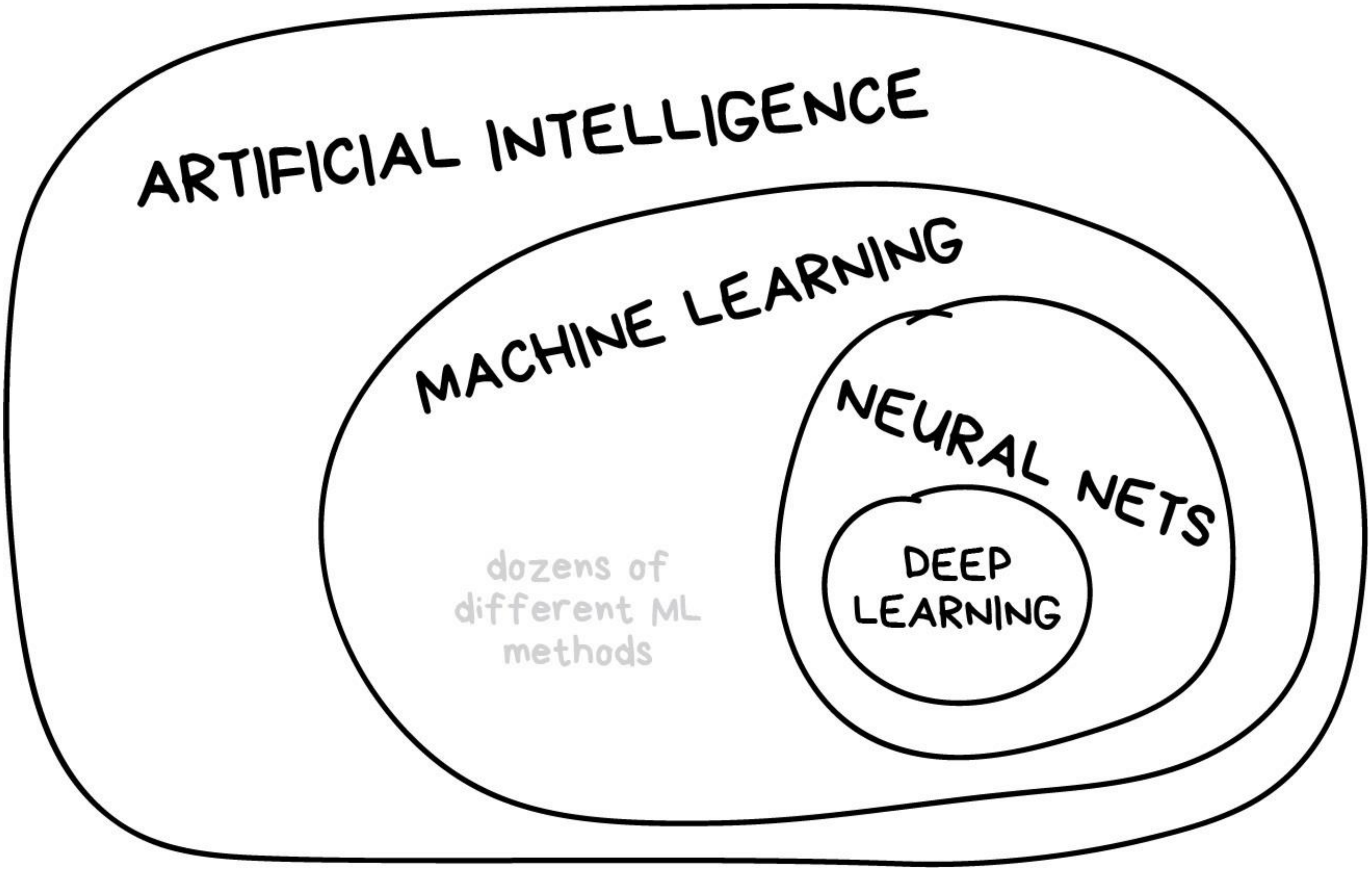


Model parameters:
"Trained weights"
- values assigned to the neurons and connections after training

Model type:
Layout/architecture of the neural network –
i.e., number of neurons, how they are connected, etc…

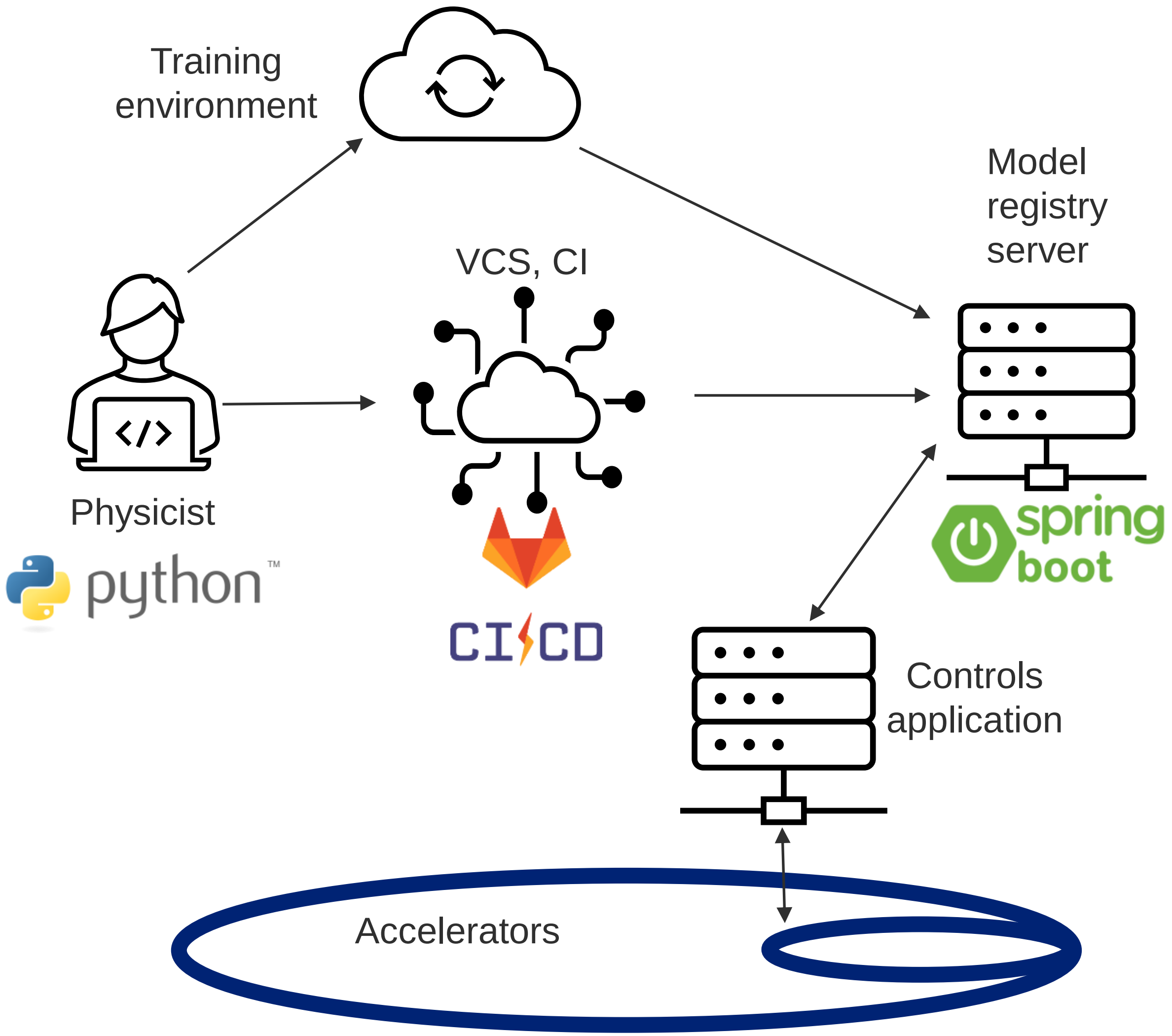
Model:
Combination of a model type and model parameters

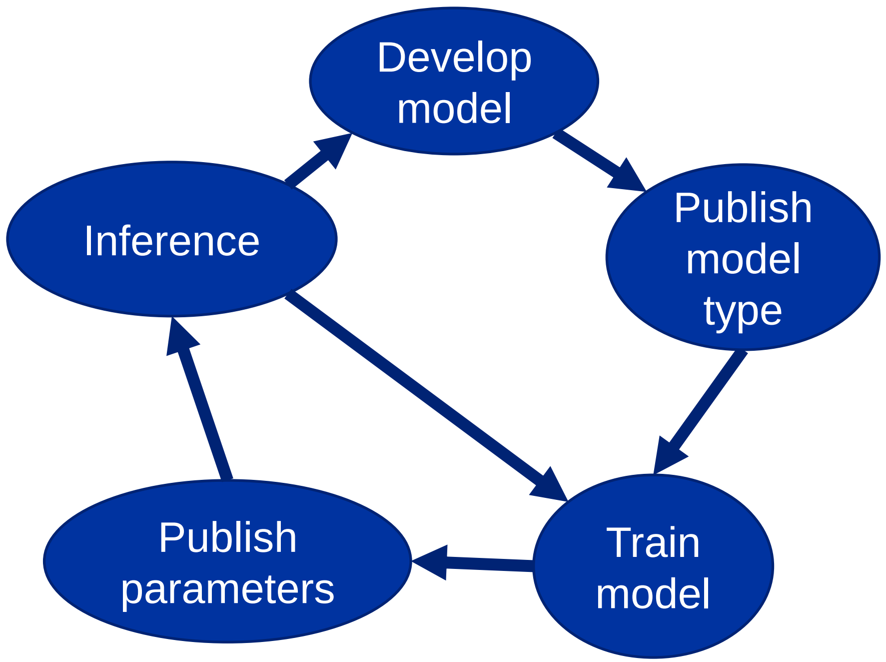
Source: https://www.youtube.com/watch?v=aircAruvnKk



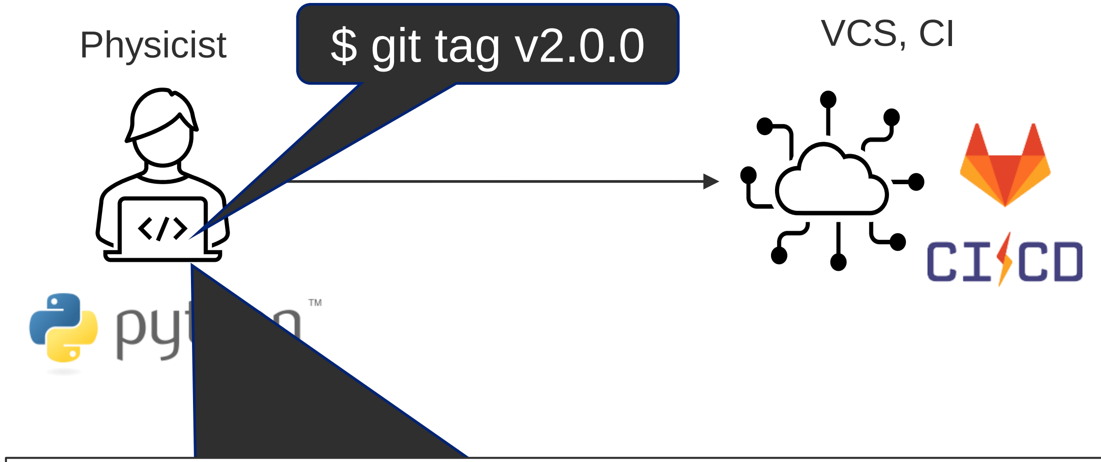
Source: https://vas3k.com/blog/machine_learning/

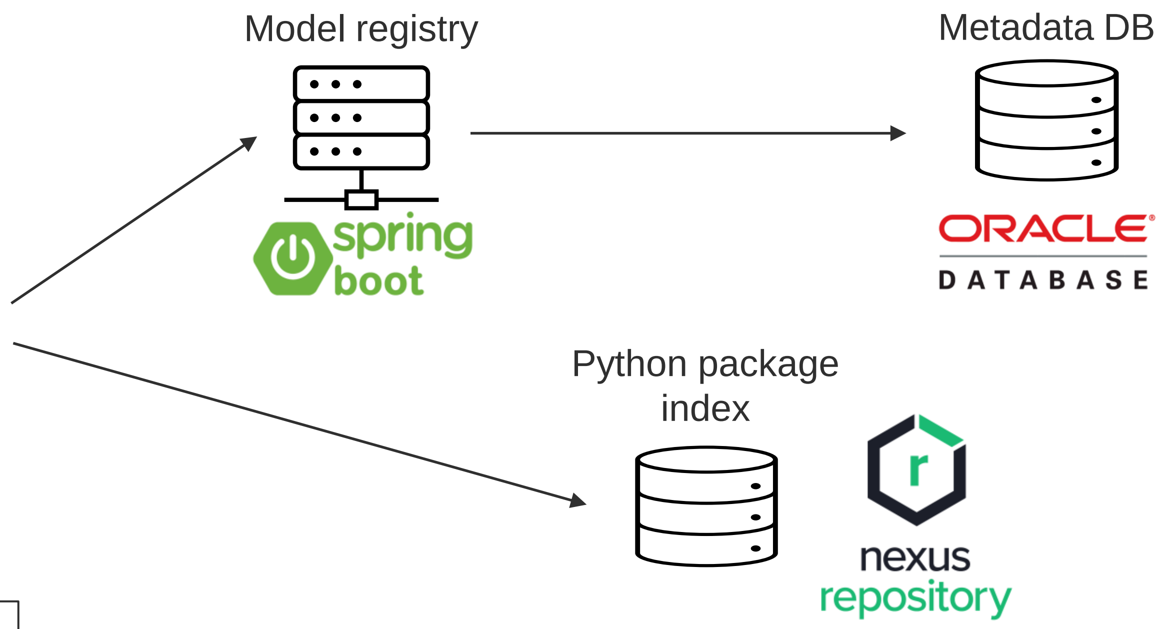
Development workflow

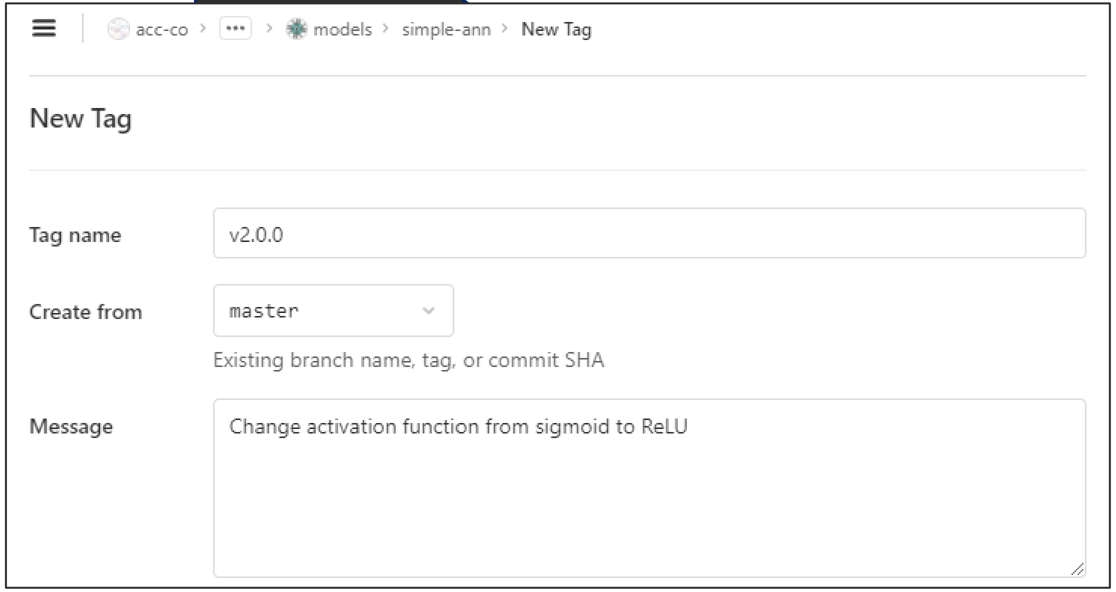




Publishing model types



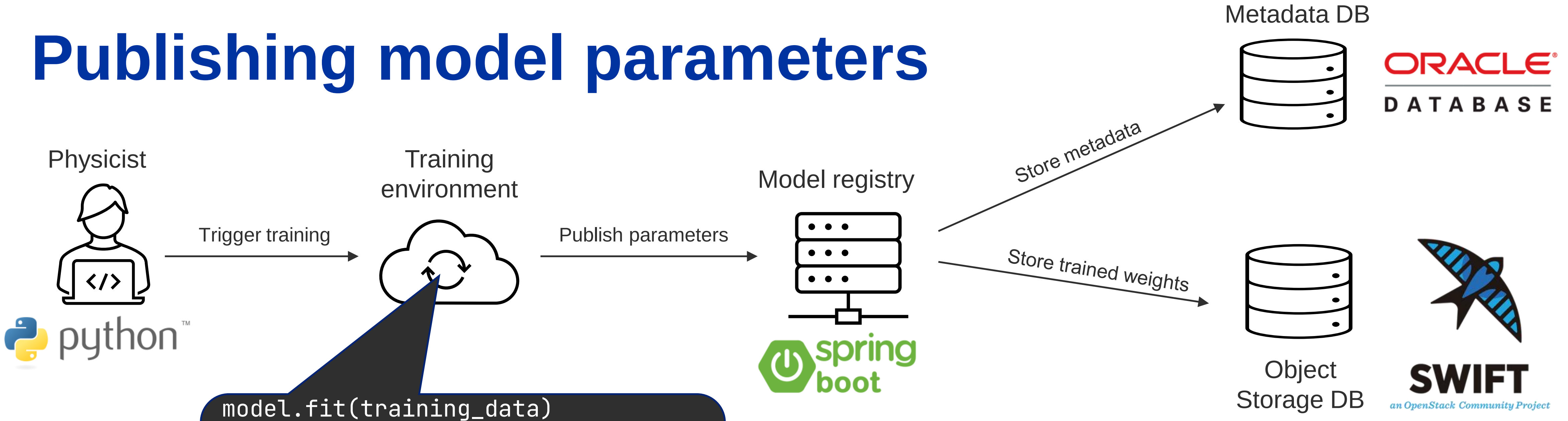




Advantages

- **Access control and traceability for model types**
- **Quick & easy, no need to learn new tools, complexity is hidden**

Publishing model parameters




```
model.fit(training_data)
client.publish_parameters_version(
    model,
    name = “proton_beam_config”,
    version = “2.0.3”
)
```

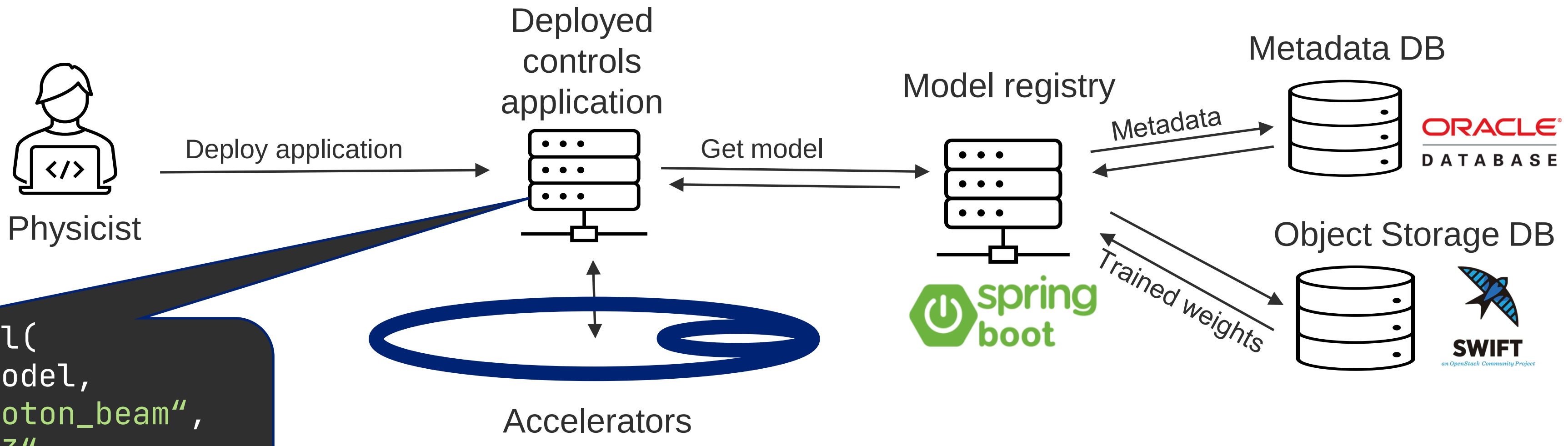
Usage

- **Choose parameters name and version**
- **Use the client library to publish**

Advantages

- **All parameters stored centrally and reliably**
- **Compatibility is fully managed**

Inference




```
model = client.create_model(
    model_type = BeamLineModel,
    model_parameters = "proton_beam",
    params_version = "2.0.3"
)
result = model.predict(input)
```

Usage

- **Use the MLP client library to instantiate the model**
- **Provide model type, parameters name and version**

Advantages

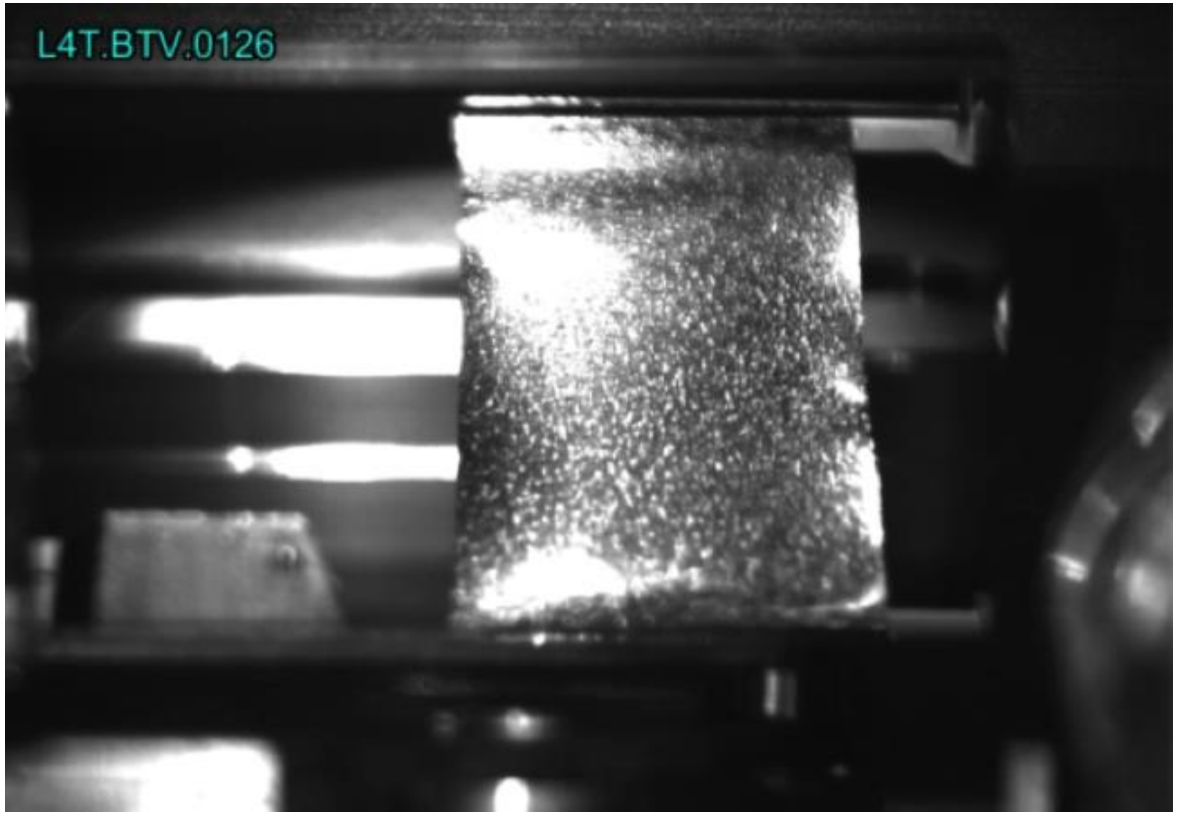
- **Parameters retrieved and loaded transparently**
- **Parameter traceability**

Continuous retraining - motivation

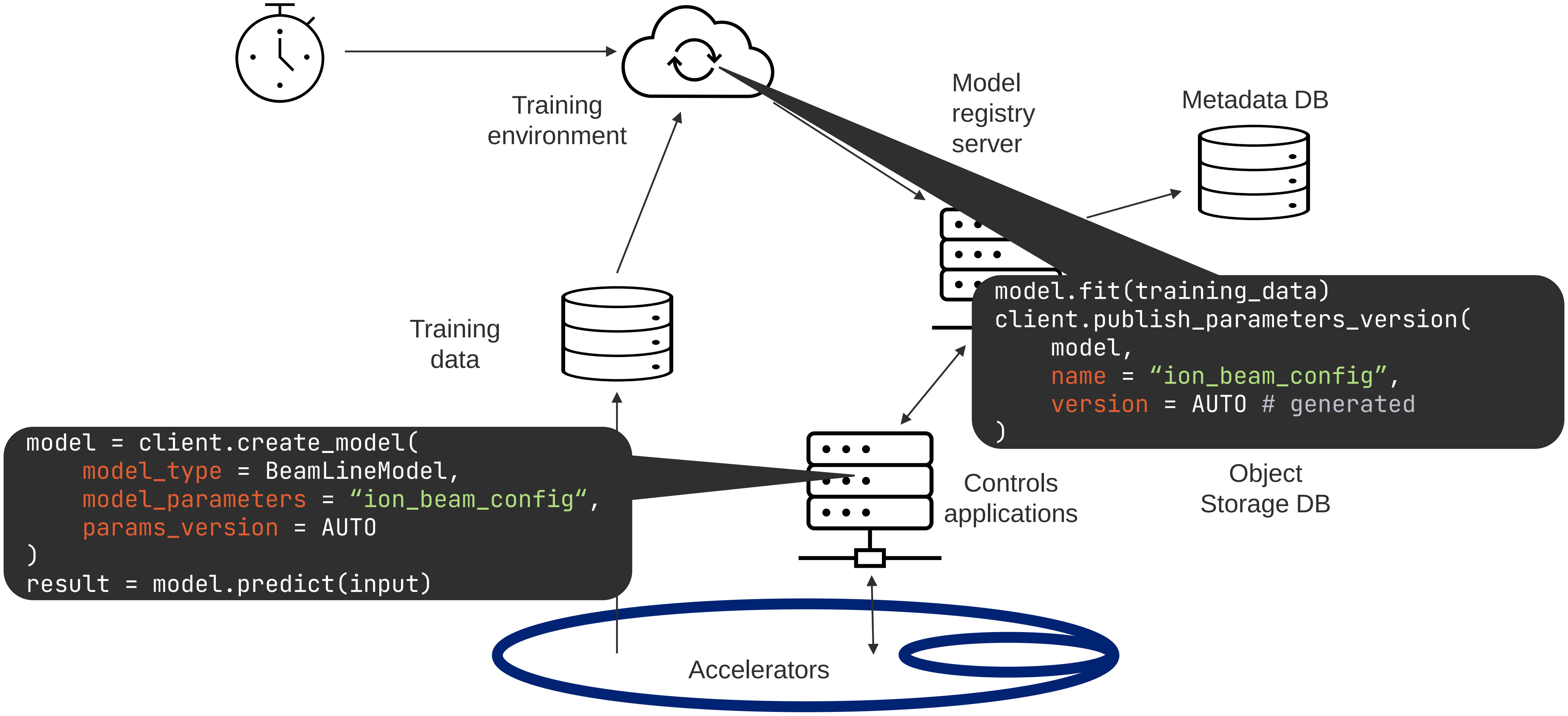
Example: stripper foil degradation

- **The stripper foil is an essential component of our linacs**
- **It degrades over time and is replaced regularly**
- **Beam characteristics vary**
- **Machine parameters need to adapt**

-> need to <u>re-train</u> model continuously to keep it up to date



Continuous retraining - implementation

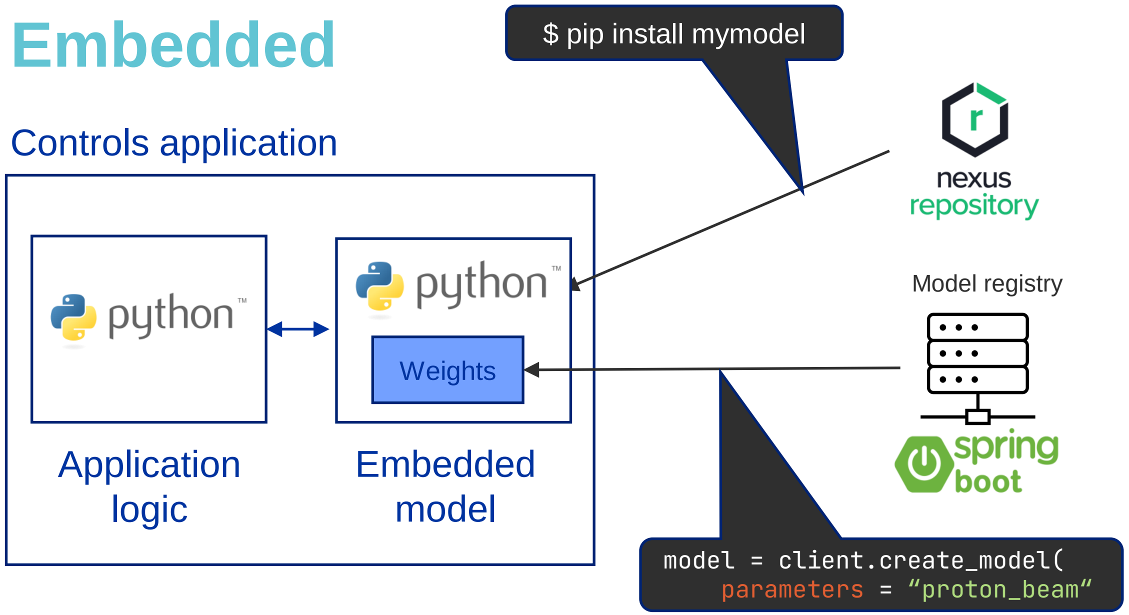


Standalone deployment prototype

Calling models remotely from any language

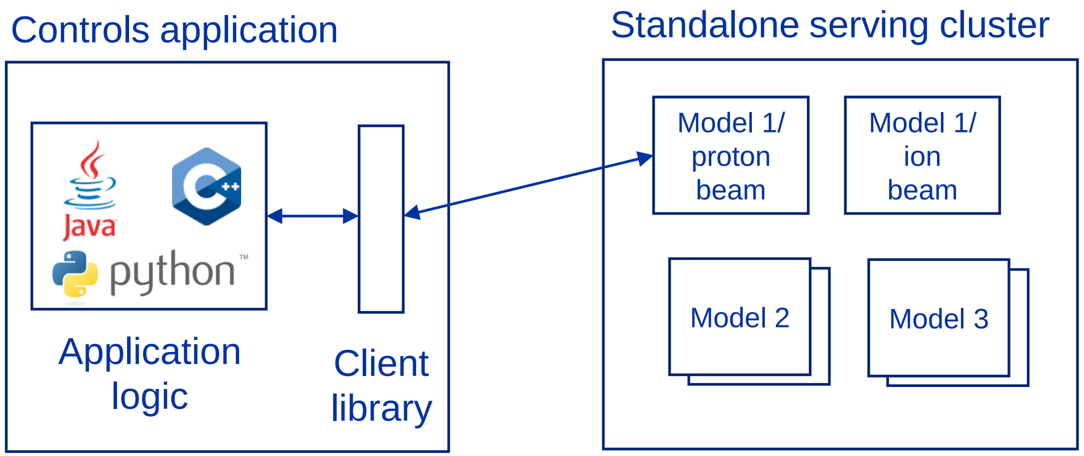
Inference - Embedded vs standalone

Embedded



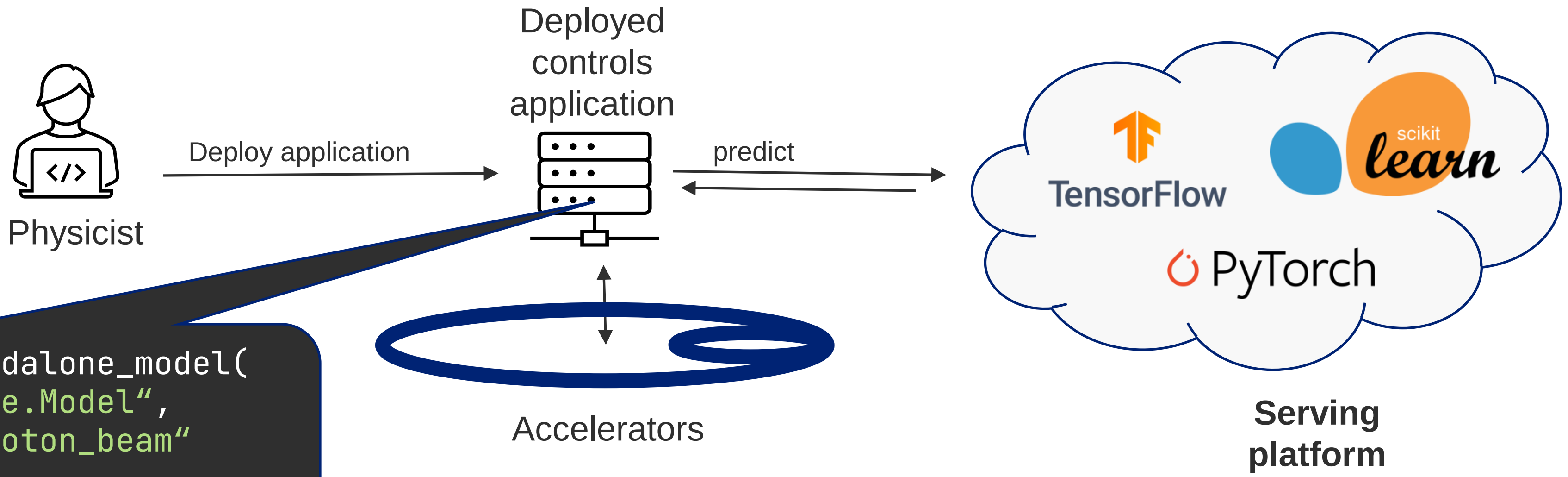
- **Python only**
- **Model type must be installed**
- **Parameters retrieved then stored locally**

Standalone



- **Language-agnostic approach**
- **No local model installation needed**
- **Everything happens remotely**

Inference
(Standalone)




```
model = client.create_standalone_model(
    model_type = "beam_line.Model",
    model_parameters = "proton_beam"
)
result = model.predict(input)
```

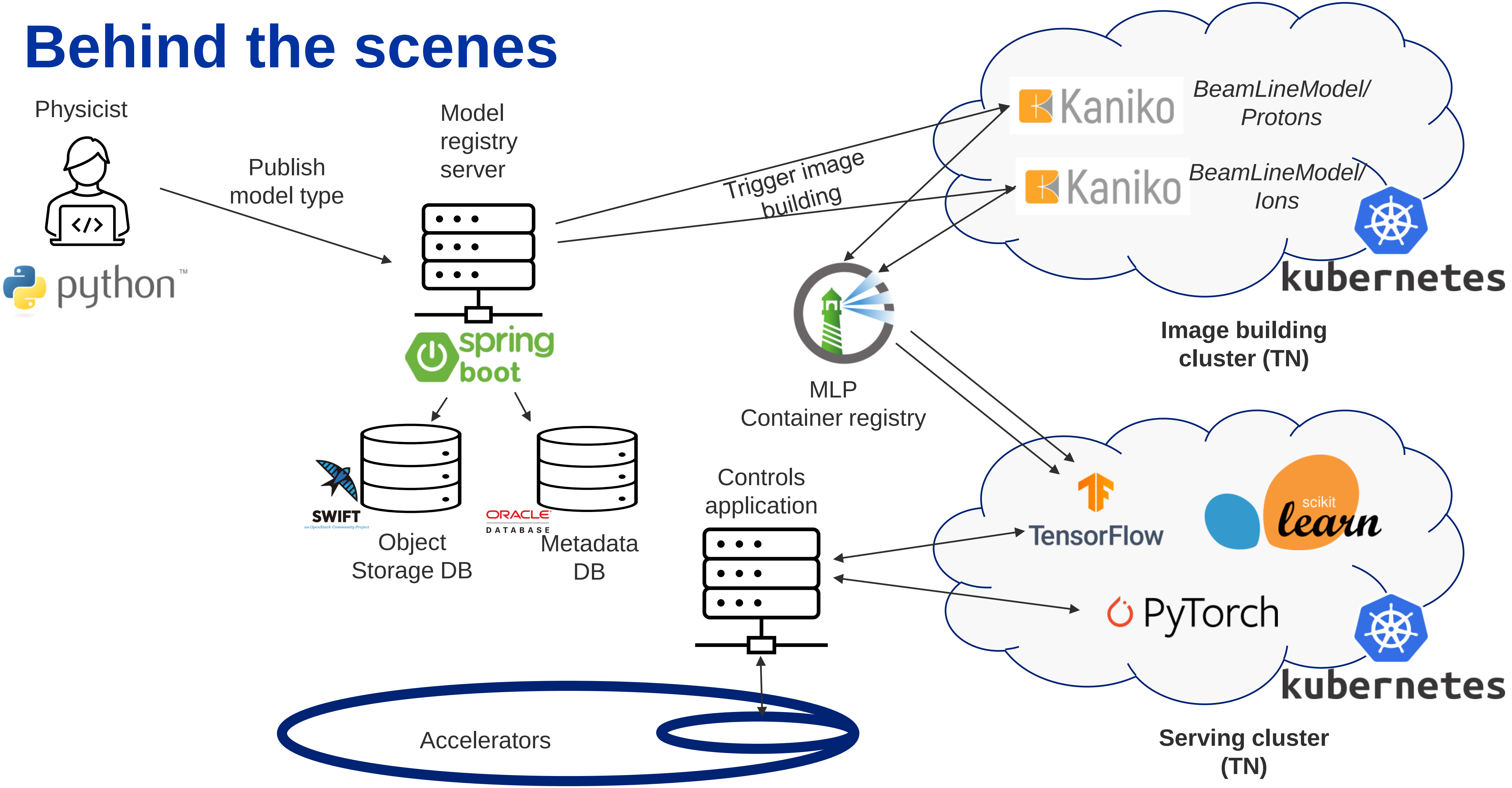
Usage

- Use the MLP client library to instantiate the model
- Provide model type and parameters name

Advantages

- Call models from any language
- Seamless model updates

Behind the scenes



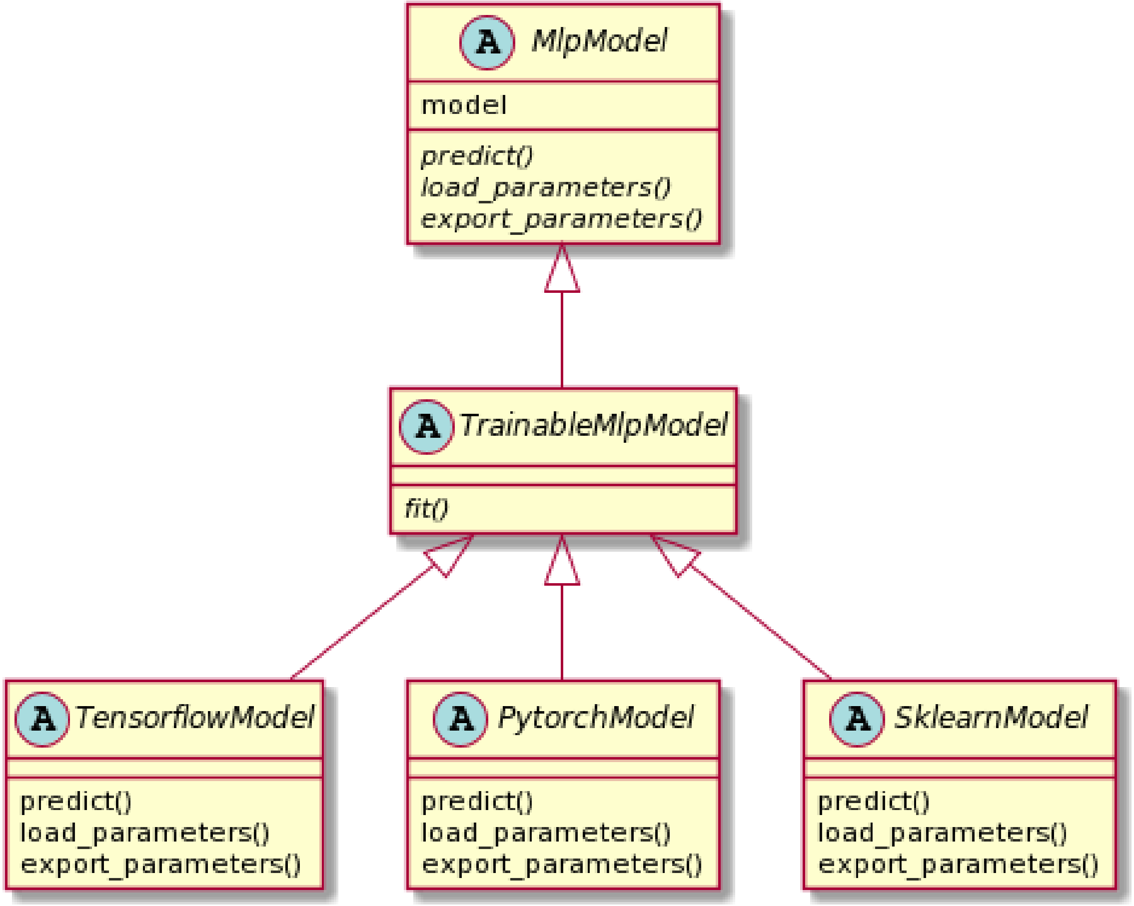
MLP in practice

MLP in Practice

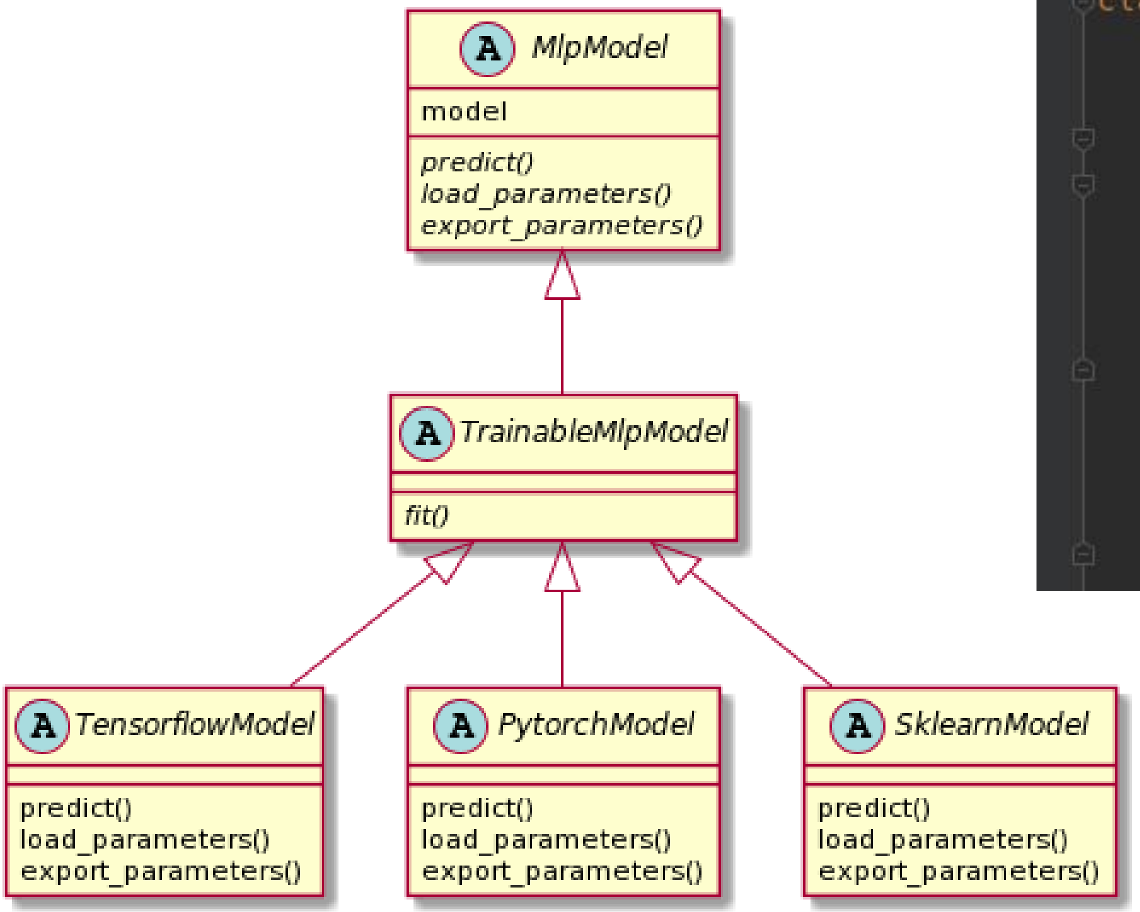
- **Adapting a model**
  - Implementing the model interface
  - Model declaration
  - CI template
- **Using a model**
  - Publishing trained parameters
  - Loading parameters
  - Standalone prediction

MLP in practice: Implementing a model

- **We define a common API for all controls models**
  - shared abstraction layer
- **Interface defines 4 methods:**
  - *Fit* – train the model on the provided data
  - *Export parameters* – extract current values of all model parameters
  - *Load parameters* – configure the model using the provided parameters
  - *Predict* – return a prediction from the input data
- **Default extensible implementations for common frameworks**

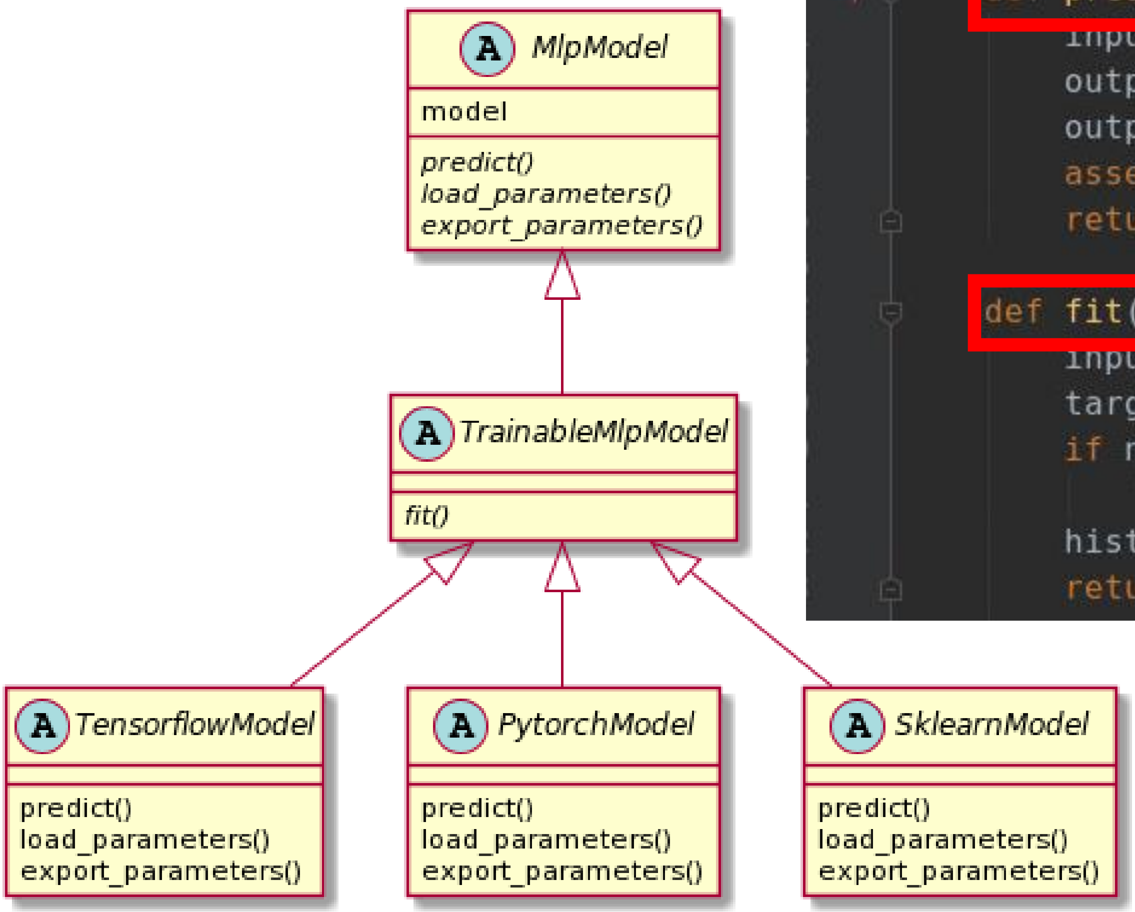


MLP in practice: Implementing a model




```python
class Model(mlptf.TensorFlowModel):
    """ANN that fits a sine function."""

    def __init__(self):
        model: tf.keras.Model = tf.keras.models.Sequential([
            tf.keras.layers.Dense(16, activation="relu"),
            tf.keras.layers.Dense(16, activation="relu"),
            tf.keras.layers.Dense(1)
        ])
        model.compile(loss='mean_squared_error',
                      optimizer=tf.keras.optimizers.Adam(0.01))
        model.build(input_shape=self.input_shape)
        super().__init__(model)
```

MLP in practice: Implementing a model




```python
def predict(self, input_data):
    inputs = np.asarray(input_data[api.INPUTS])
    outputs = self.model.predict(inputs[:, np.newaxis])
    outputs = outputs[:, 0]
    assert np.ndim(outputs) == 1
    return {api.OUTPUTS: outputs}

def fit(self, input_data) -> api.FitHistory:
    inputs = np.asarray(input_data[api.INPUTS])
    targets = np.asarray(input_data[api.TARGETS])
    if np.ndim(targets) == 2:
        targets = np.squeeze(targets, 1)
    history = self.model.fit(inputs[:, np.newaxis], targets, epochs=500)
    return mlptf.convert_to_common_history(history)
```

MLP in practice: Implementing a model

Model declaration
- mlp-models.toml

```
mlp-tensorflow-model-demo
  mlp_tensorflow_model_demo
    tests
    __init__.py
    __main__.py
    _version.py
    mlp-models.toml
```

```toml
[[model]]
name = "mlp_tensorflow_model_demo:Model"
standalone = false

[[model]]
name = "mlp_tensorflow_model_demo:SecondModel"
standalone = false
```

CI configuration
- gitlab-ci.yml

```yaml
include:
  - project: acc-co/machine-learning-platform/mlp-ci
    file: pipeline-templates/mlp-model-gitlab-ci-template.yml

variables:
  project_name: mlp_tensorflow_model_demo

#
# --- Custom CI jobs ---
#

# Full installation, tested with pytest.
test_install:
  extends: .acc_py_full_test


# Development installation, tested with pytest.
test_dev:
  extends: .acc_py_dev_test
```

MLP in practice: Using a model

Publish trained parameters

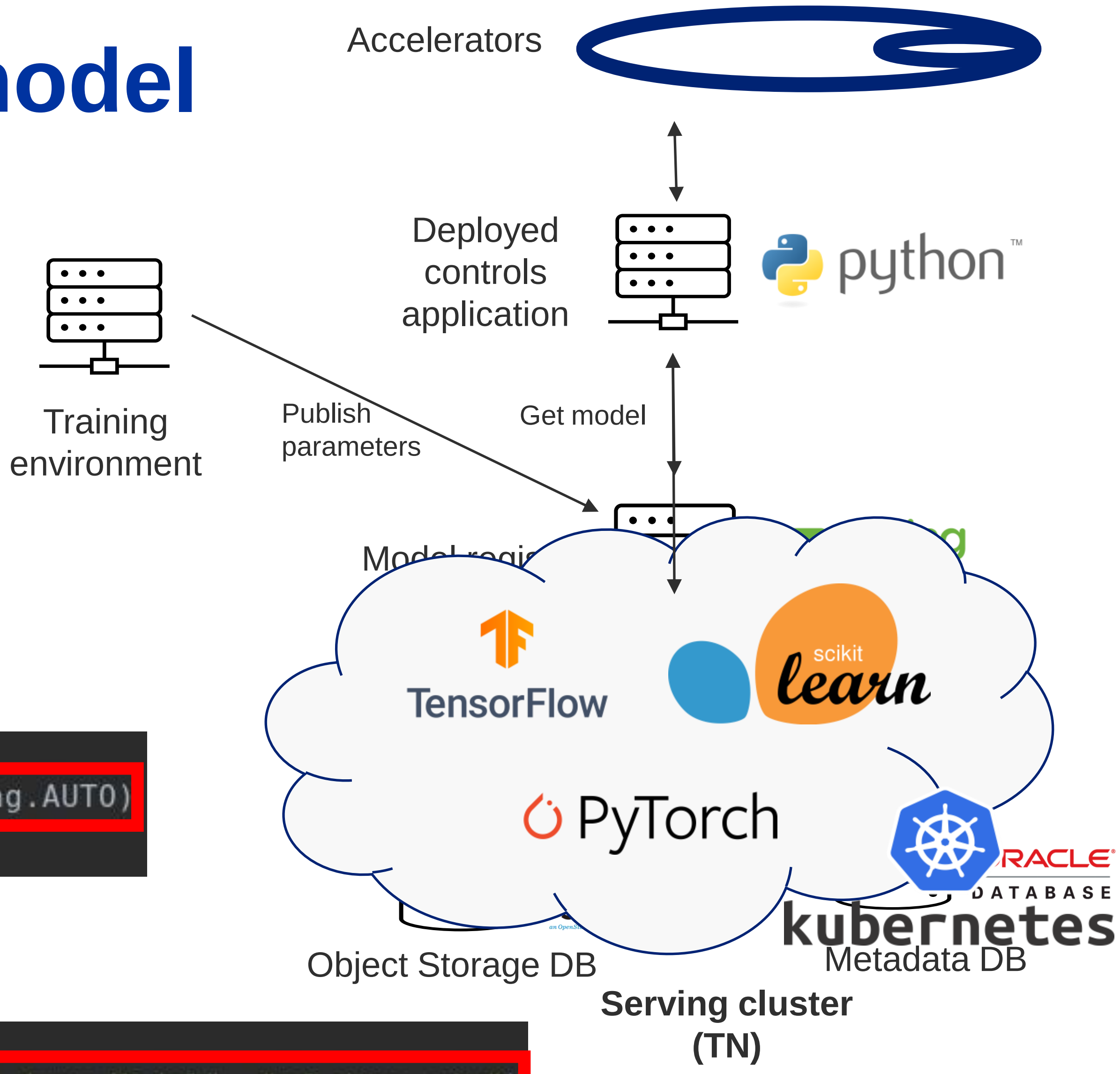
```
model = Model()
history = model.fit(dict(inputs=inputs, targets=targets))

client = Client(Profile.DEV)
mpv = client.publish_model_parameters_version(
    model, name="tf_demo.tanh", version=AUTO)
```

Load remote parameters

```
client = Client(Profile.DEV)
model = client.create_model(Model, "tf_demo.sin", VersionFlag.AUTO)
preds = model.predict(dict(inputs=inputs))["outputs"]
```

Call standalone model (prototype)

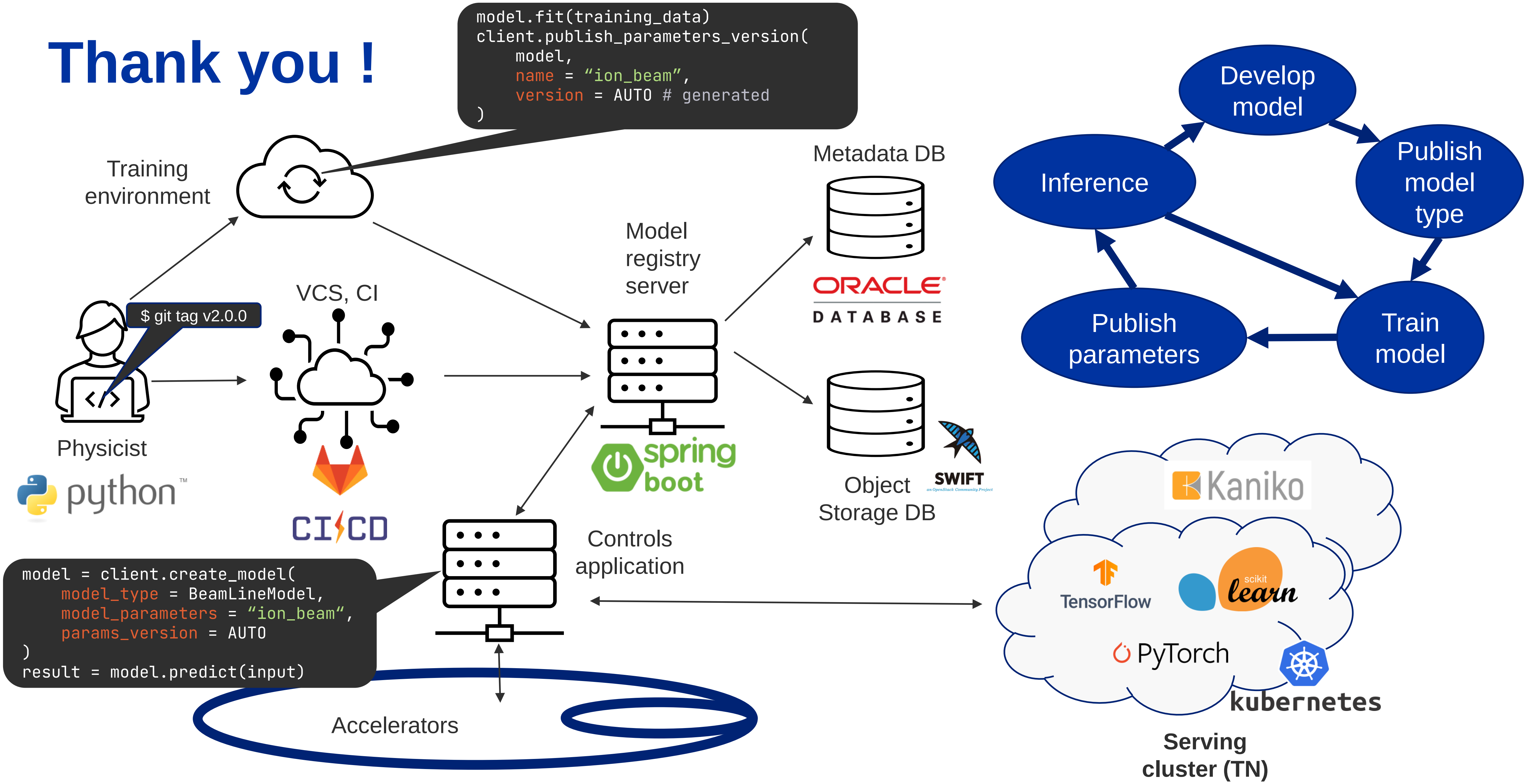
```
client = Client(Profile.DEV)
model = client.create_standalone_model("mlp_tensorflow_model_demo:Model", "tf_demo.sin")
preds = model.predict(dict(inputs=inputs))["outputs"]
```



Conclusion

- **We want to help physicists develop models faster and unburden them from infrastructural concerns while minimizing constraints**
- **We also want to apply software engineering best practices to ensure reliability and maintainability of the control system**
- **MLP provides a basis to achieve these goals and is ready for production (standalone deployment in beta)**
- **Could not cover everything, simplified a lot – please contact us offline!**
  - MLP Wikis - ICALEPCS paper – machine-learning-platform-support@cern.ch
  - jean-baptiste.de.martel@cern.ch
  - nico.madysa@cern.ch
  - roman.gorbonosov@cern.ch

Thank you !



model.fit(training_data)
client.publish_parameters_version(
model,
name = “ion_beam”,
version = AUTO # generated
)
Training environment
Metadata DB
ORACLE DATABASE
Model registry server
VCS, CI
$ git tag v2.0.0
Physicist
python
CI/CD
spring boot
Object Storage DB
SWIFT
Controls application
model = client.create_model(
model_type = BeamLineModel,
model_parameters = “ion_beam“,
params_version = AUTO
)
result = model.predict(input)
Accelerators
Develop model
Publish model type
Inference
Train model
Publish parameters
Kaniko
TensorFlow
scikit learn
PyTorch
kubernetes
Serving cluster (TN)

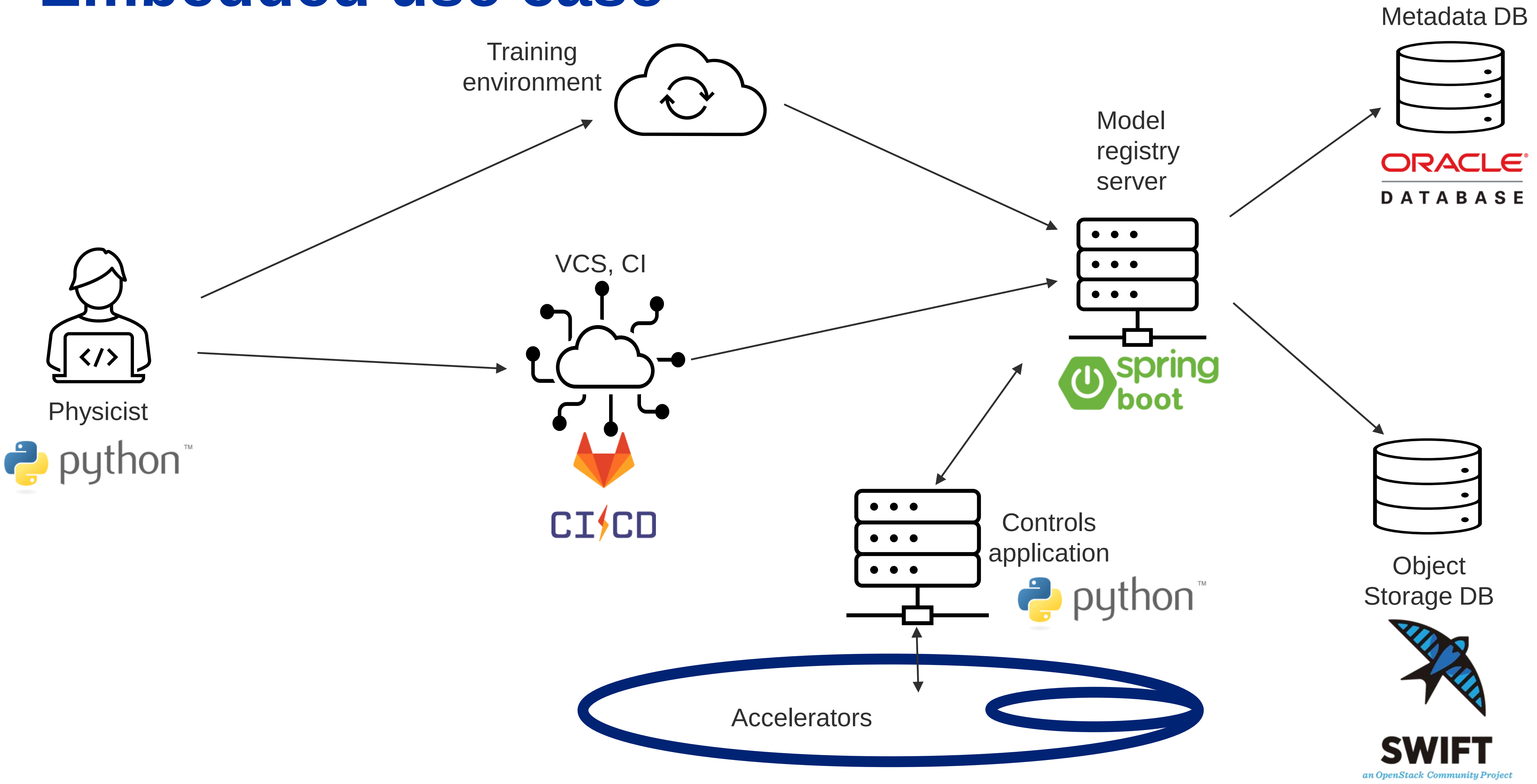
CERN

Reserve slides

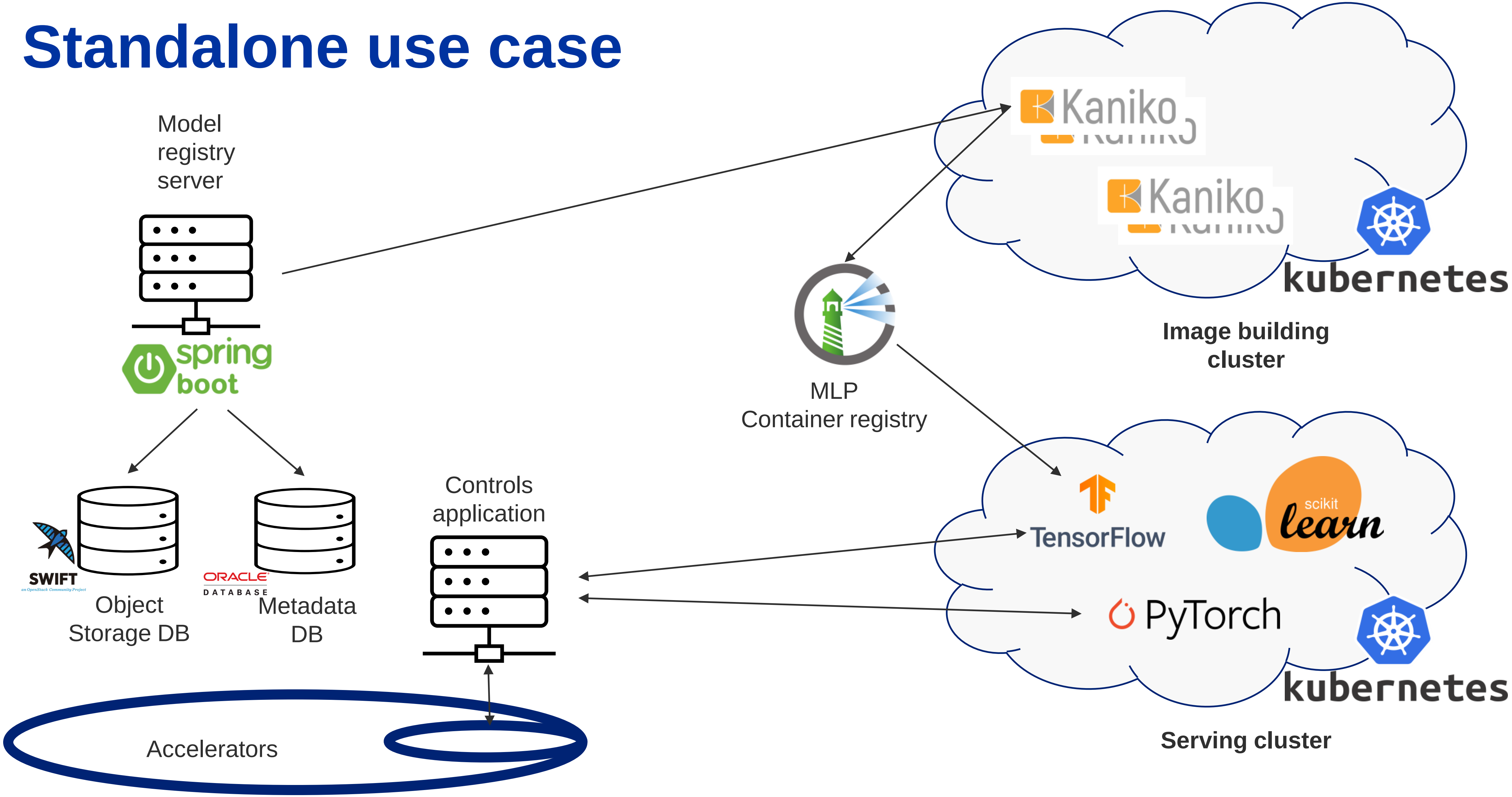
Use cases

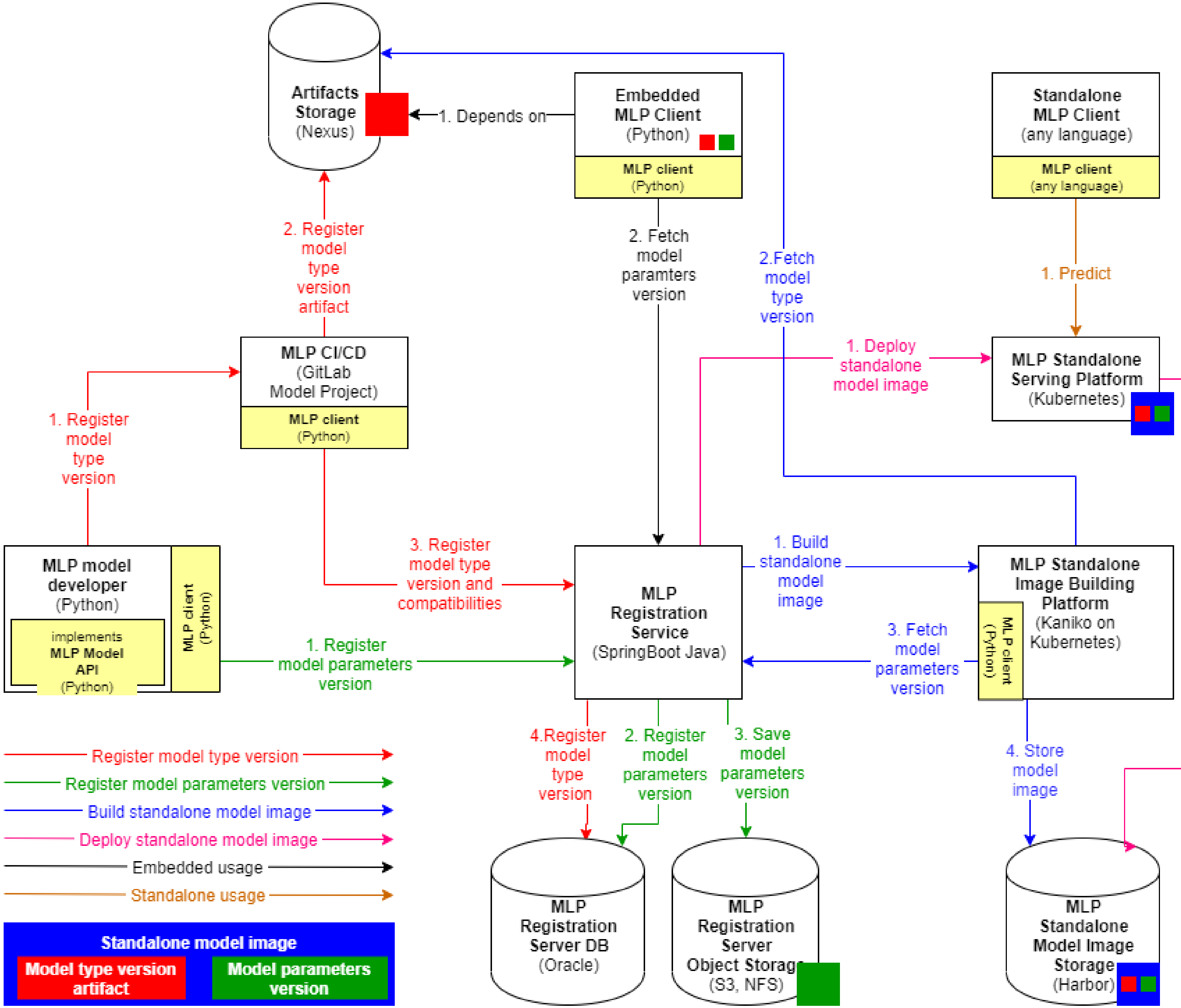
- SPS eddy current effect – V. Kain
- SPS deep hysteresis compensation – N. Madysa
- LEIR Schottky computer vision – N. Madysa
- SPS MKD dump pattern analysis – F. M. Velotti
- Awake auto-matching - F. M. Velotti

Embedded use case

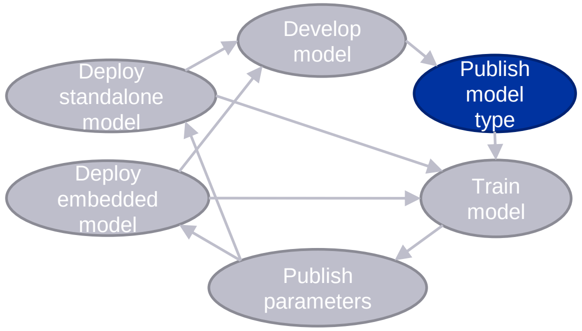


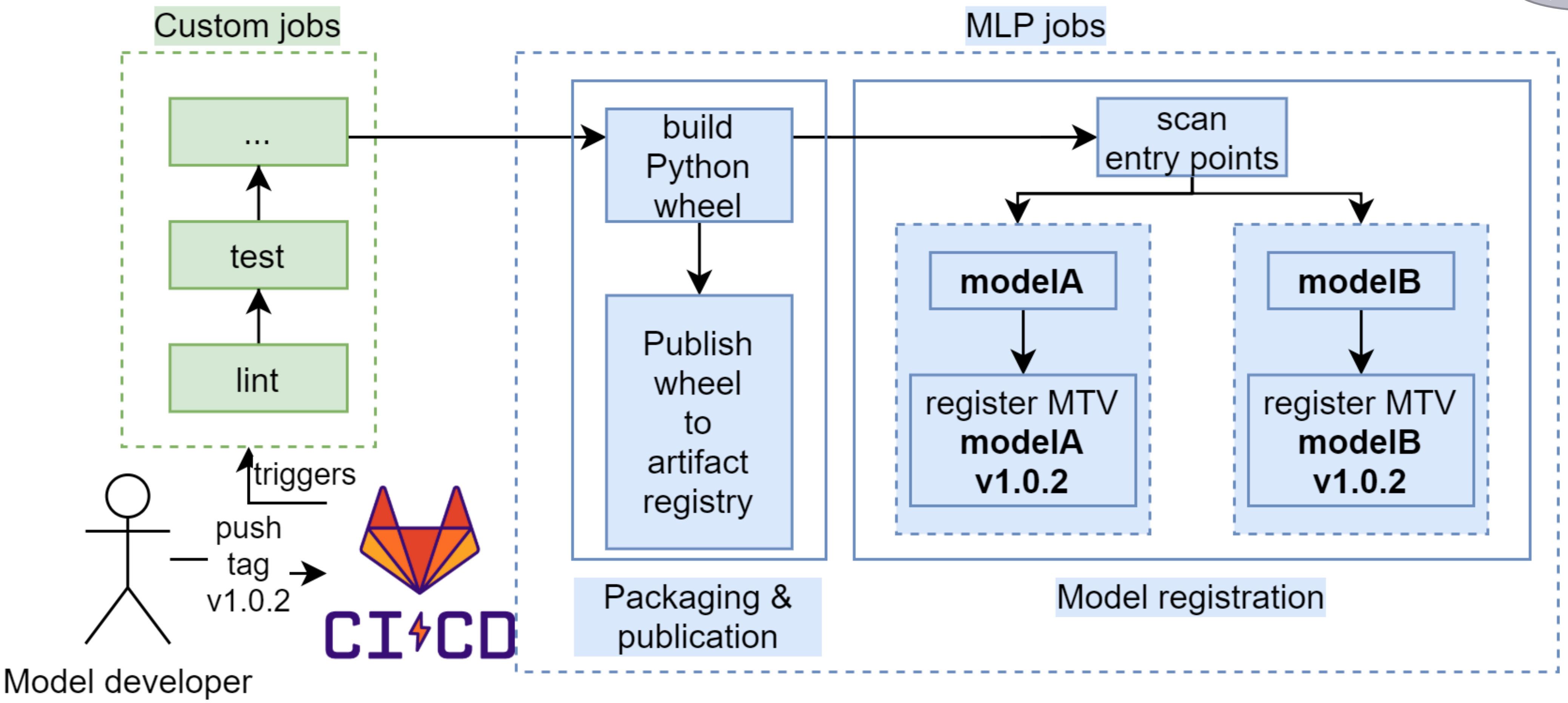
Standalone use case




Artifacts Storage (Nexus)
1. Depends on
Embedded MLP Client (Python)
MLP client (Python)
Standalone MLP Client (any language)
MLP client (any language)
2. Register model type version artifact
2. Fetch model paramters version
2.Fetch model type version
1. Predict
MLP CI/CD (GitLab Model Project)
MLP client (Python)
1. Deploy standalone model image
MLP Standalone Serving Platform (Kubernetes)
1. Register model type version
2. Fetch model image to deploy
MLP model developer (Python)
implements MLP Model API (Python)
MLP client (Python)
3. Register model type version and compatibilities
MLP Registration Service (SpringBoot Java)
1. Build standalone model image
MLP Standalone Image Building Platform (Kaniko on Kubernetes)
MLP client (Python)
1. Register model parameters version
3. Fetch model parameters version
4.Register model type version
2. Register model parameters version
3. Save model parameters version
4. Store model image
Register model type version
Register model parameters version
Build standalone model image
Deploy standalone model image
Embedded usage
Standalone usage
MLP Registration Server DB (Oracle)
MLP Registration Server Object Storage (S3, NFS)
MLP Standalone Model Image Storage (Harbor)
Standalone model image
Model type version artifact
Model parameters version

Publishing model types

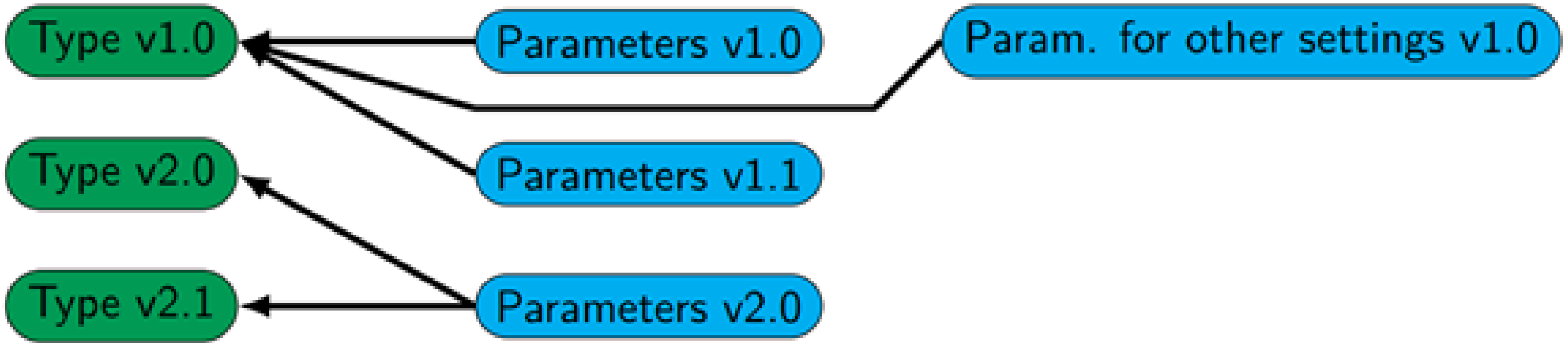




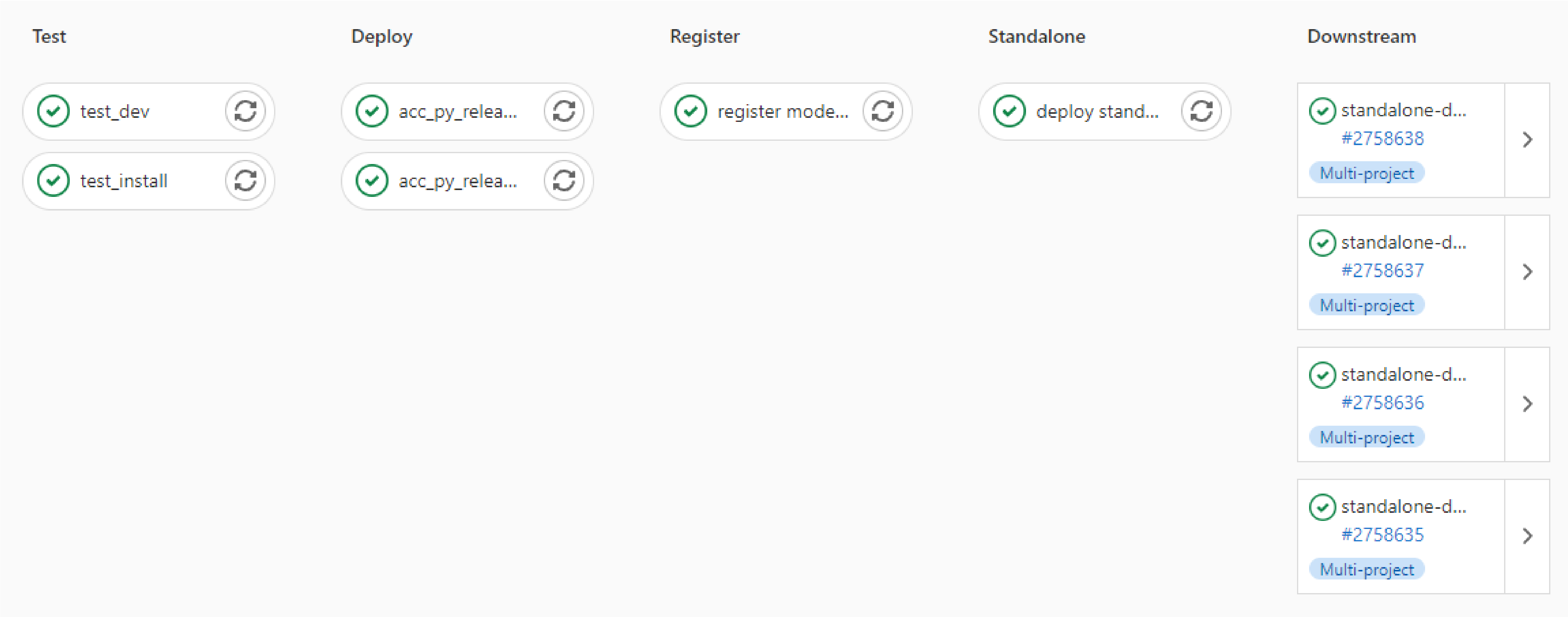
Model parameters version number generation

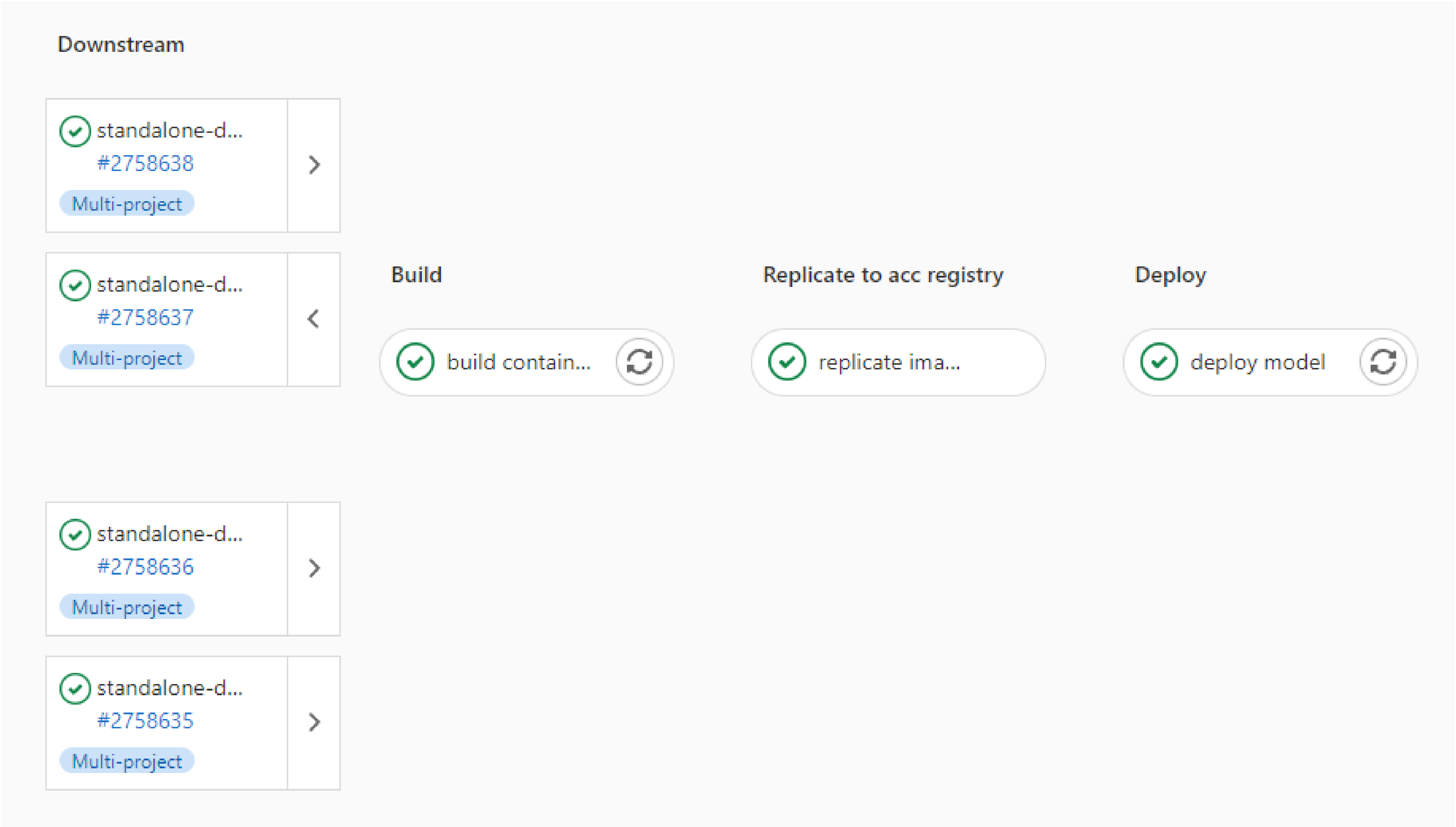
Model type version	Highest existing parameters version	->	Generated parameters version
1.0.0	None exist yet	->	1.0
1.0.0	1.0	->	1.1
1.6.0	1.1	->	1.2
2.0.0	1.2	->	2.0
3.3.0	4.0 (no 3.x)	->	ambiguity
3.3.0	4.0 (3.3 exists)	->	3.4

Compatibility



Standalone deployment CI




Downstream
standalone-d...
#2758638
Multi-project
standalone-d...
#2758637
Multi-project
standalone-d...
#2758636
Multi-project
standalone-d...
#2758635
Multi-project
Build
build contain...
Replicate to acc registry
replicate ima...
Deploy
deploy model

CERN