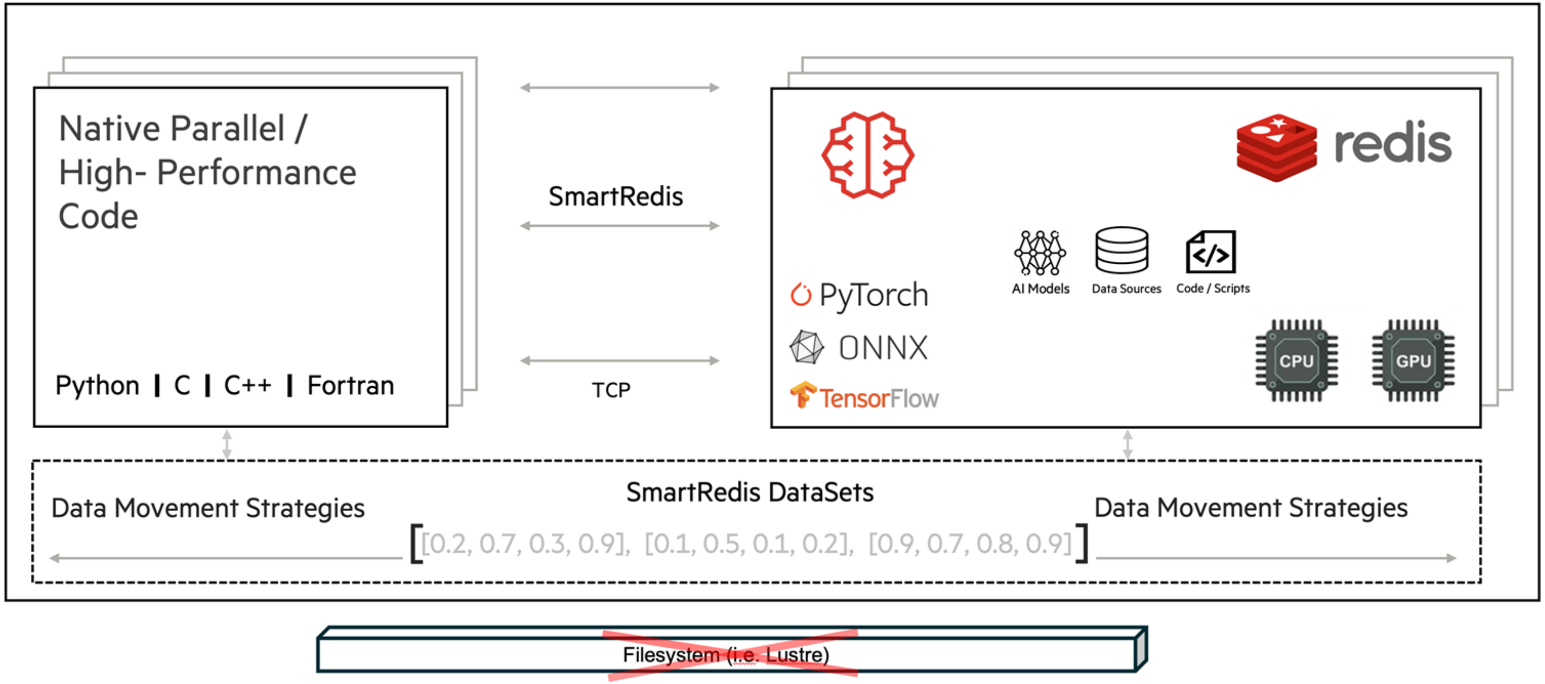
# SmartSim Introduction

# Contents

- What is SmartSim
- SmartSim Components
- Example

# SmartSim

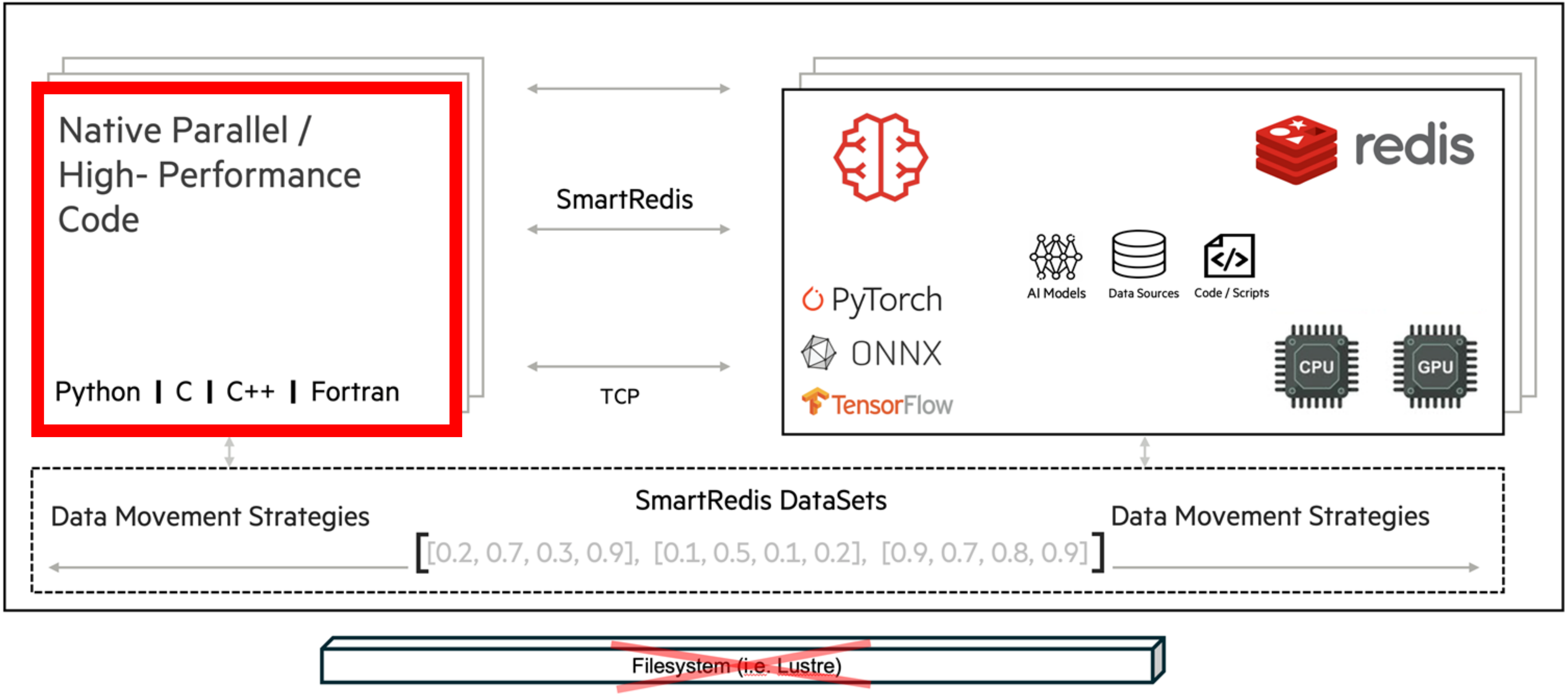
SmartSim enables scientists to utilize machine learning inside traditional HPC workloads




SmartSim provides this capability by:

- Automating the deployment of HPC workloads and distributed, in-memory storage (Redis).
- Making TensorFlow, Pytorch, and ONNX callable from Fortran, C, and C++ simulations.
- Providing flexible data communication and formats for hierarchical data, enabling online analysis, visualization, and processing of simulation data.

# SmartSim



Native Parallel / High- Performance Code
Python | C | C++ | Fortran
SmartRedis
TCP
PyTorch
ONNX
TensorFlow
redis
AI Models
Data Sources
Code / Scripts
CPU
GPU
Data Movement Strategies
SmartRedis DataSets
[[0.2, 0.7, 0.3, 0.9], [0.1, 0.5, 0.1, 0.2], [0.9, 0.7, 0.8, 0.9]]
Data Movement Strategies
Filesystem (i.e. Lustre)

# SmartSim

The main goal of SmartSim is to provide scientists a flexible, easy to use method for interacting at runtime with the data generated by simulation. The type of interaction is completely up to the user:

- Embed calls to machine learning models inside a simulation
- Create hooks to manually or programmatically steer a simulation
- Visualize the progression of a simulation integration from a Jupyter notebook

# SmartSim (Infrastructure Library)

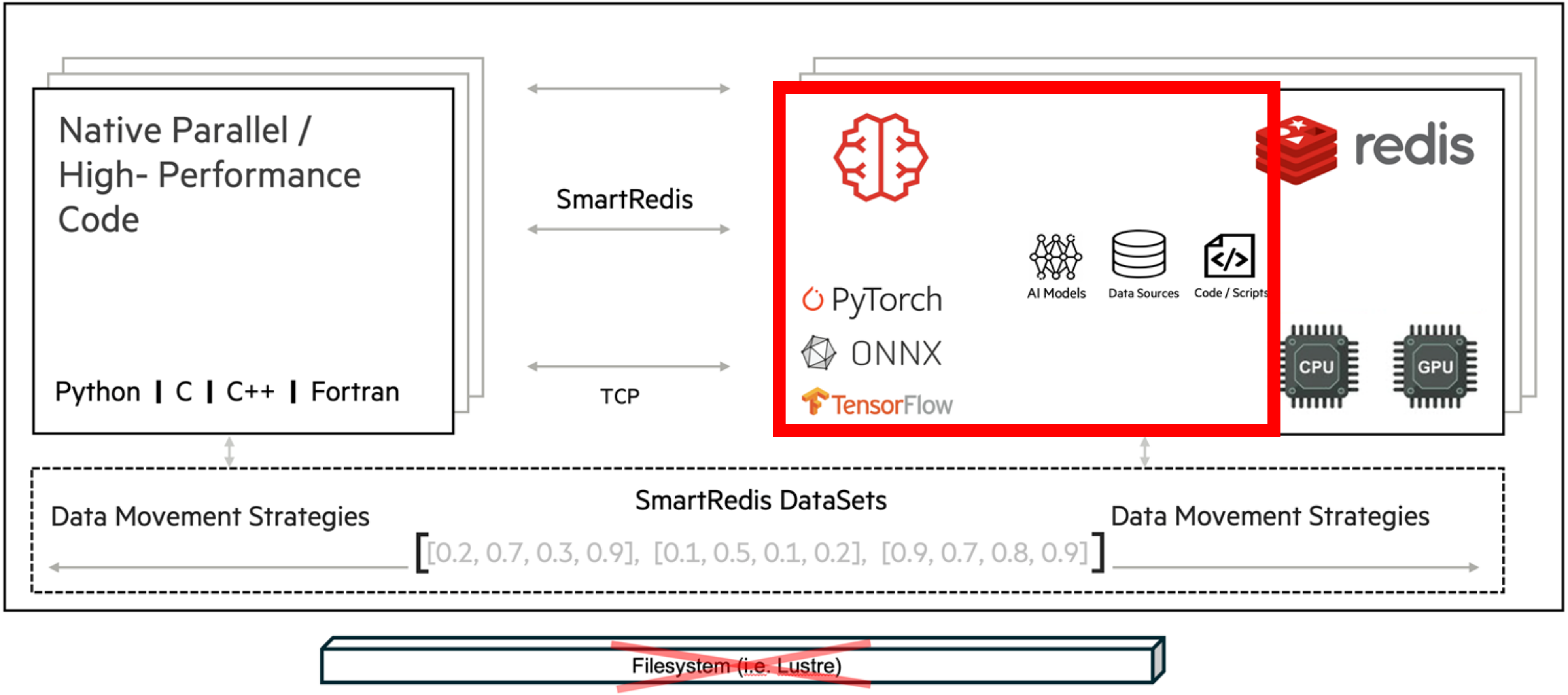
automate process of deploying HPC workloads alongside **in-memory database**

deploy **distributed**, shared-nothing, **in-memory cluster of Redis instances** across compute nodes → **Orchestrator**

Orchestrator + HPC applications
→ connect workloads (e.g. trained ML models) to other applications with SmartRedis clients

- An API to start, monitor, and stop HPC jobs from Python or from a Jupyter notebook.
- Automated deployment of in-memory data staging (Redis) and computational storage (RedisAI).
- Programmatic launches of batch and in-allocation jobs on PBS, Slurm, LSF, and Cobalt systems.
- Creating and configuring ensembles of workloads with isolated communication channels.

# RedisAI (Client Library)



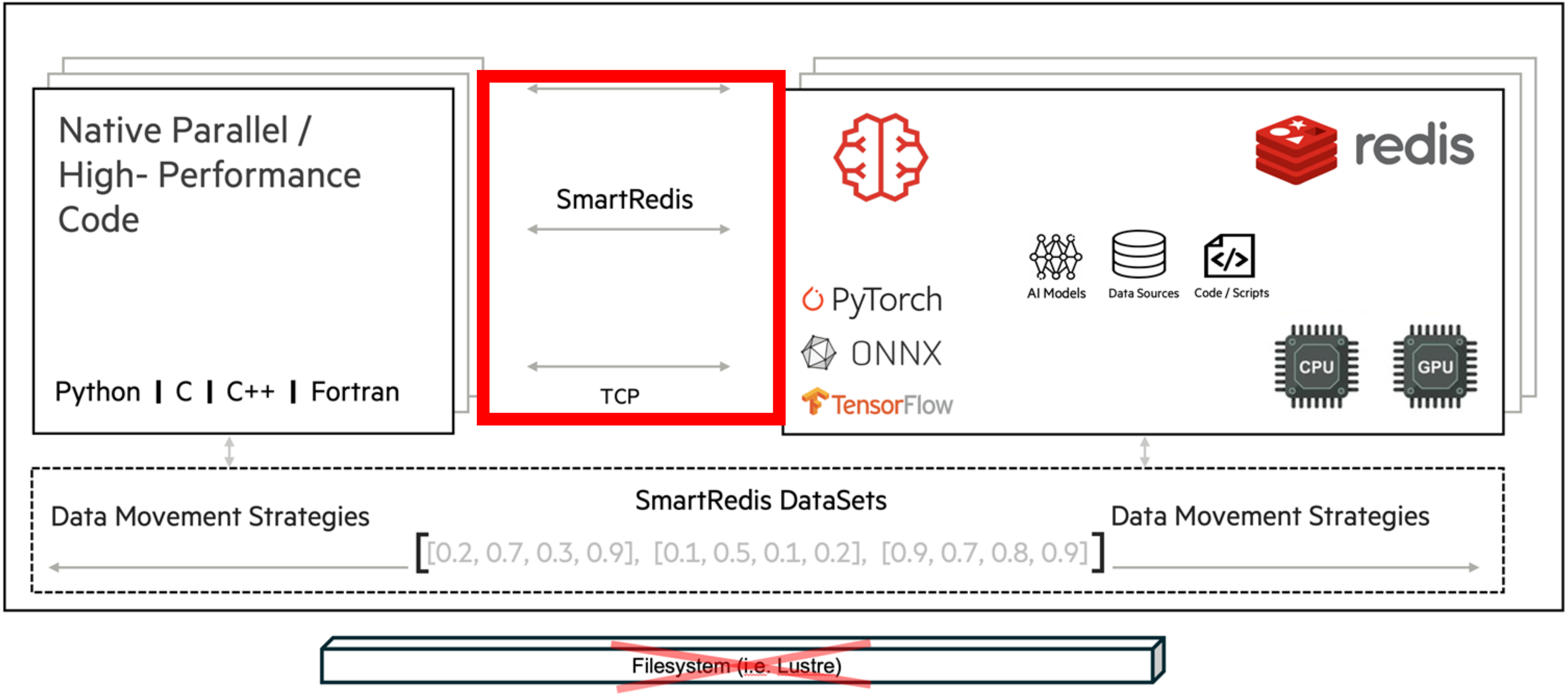
# RedisAI (Client Library)

Redis module for executing Deep Learning/Machine Learning models and managing their data.

**Key features** of **RedisAI** supported by SmartRedis

- A tensor data type in Redis
- TensorFlow, TensorFlow Lite, Torch, and ONNXRuntime backends for model evaluations
- TorchScript storage and evaluation
- Data locality

# SmartRedis (Client Library)



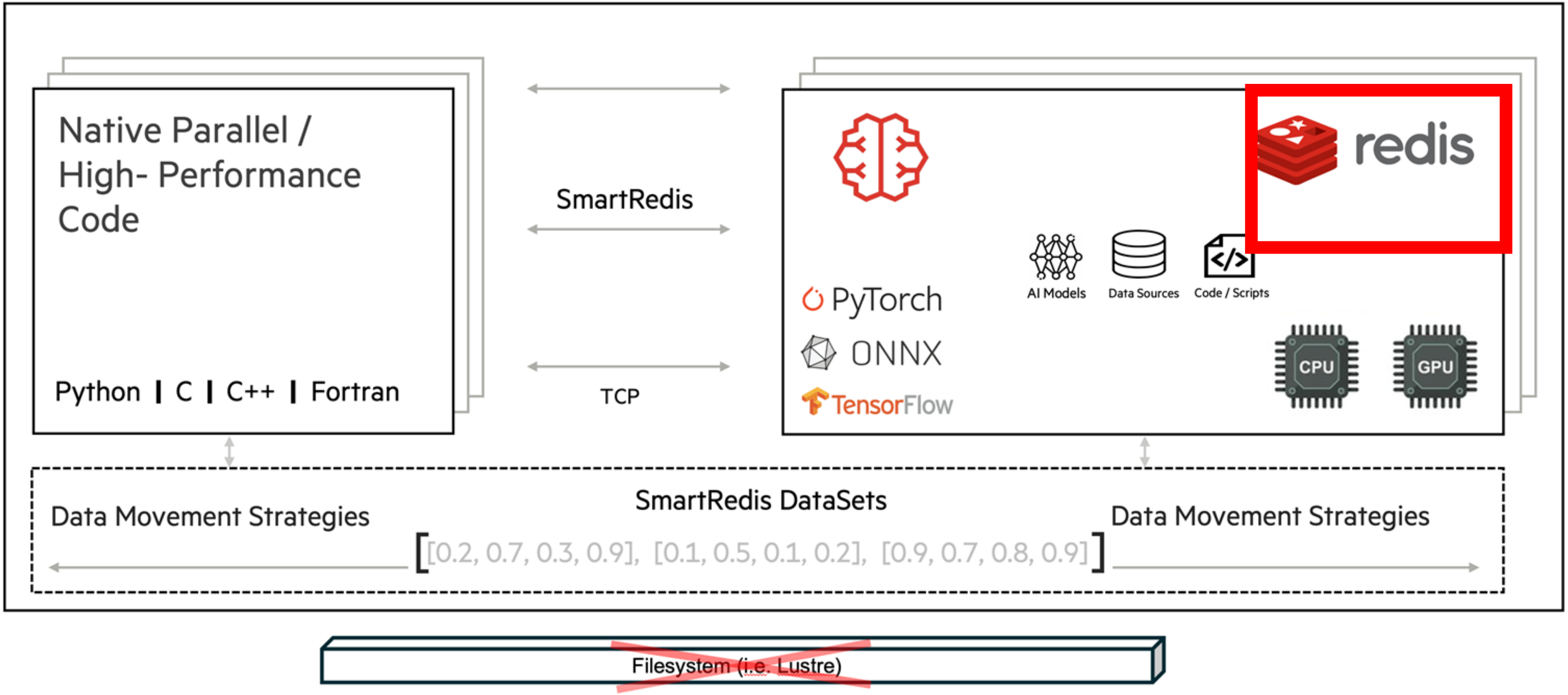
# SmartRedis (Client Library)

collection of Redis clients supporting RedisAI capabilities and include additional features for HPC applications

**Key features** of **SmartRedis** developed for large, distributed HPC architectures

- Redis cluster support for RedisAI data types (tensors, models, and scripts).
- Distributed model and script placement for parallel evaluation that **maximizes hardware utilisation** and throughput
- A **DataSet storage format** to **aggregate multiple tensors** and metadata into a **single Redis cluster** hash slot to **prevent data scatter** on Redis clusters and maintain contextual relationships between tensors. Useful when clients produce tensors and metadata that are referenced or utilised together.
- An **API for efficiently aggregating DataSet objects** distributed on one or more database nodes.
- **Compatibility with SmartSim ensemble capabilities** to prevent key collisions with tensors, DataSet, models, and scripts when clients are part of an ensemble of applications.

# Redis



Native Parallel / High- Performance Code
Python | C | C++ | Fortran
SmartRedis
TCP
redis
AI Models
Data Sources
Code / Scripts
PyTorch
ONNX
TensorFlow
CPU
GPU
Data Movement Strategies
SmartRedis DataSets
Data Movement Strategies
[[0.2, 0.7, 0.3, 0.9], [0.1, 0.5, 0.1, 0.2], [0.9, 0.7, 0.8, 0.9]]
Filesystem (i.e. Lustre)

# Redis

open source, **in-memory** data store.

Can be used as:

- Database
- Cache
- Message broker

Provides data structures (like strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes, and streams).

Built-in replication, Lua scripting, LRU eviction, transactions, and different levels of on-disk persistence.

Atomic operations

**Asynchronous replication**, with fast non-blocking synchronization and auto-reconnection with partial resynchronization on net split.

# SmartSim Example

Install SmartSim:

- https://www.craylabs.org/docs/installation_instructions/basic.html

Or Use our installed version:

- Login to Cirrus
- Activate the Miniconda environment and load the modules
- Bout++ and SmartSim are installed in:
  - /work/tc045/tc045/shared/bpp_5_0_0_ss_0_4_2/BOUT-dev
  - /work/tc045/tc045/shared/bpp_5_0_0_ss_0_4_2/SmartSim

# SmartSim Example

```
from smartsim import Experiment
```

# SmartSim Example

from smartsim import Experiment

Import **Experiment** from smartsim

# SmartSim Example

```
from smartsim import Experiment

# Init Experiment and specify to launch locally
exp = Experiment(name="getting-started", launcher="local")
```

# SmartSim Example

```
from smartsim import Experiment

# Init Experiment and specify to launch locally
exp = Experiment(name="getting-started", launcher="local")
```

Initialise the Experiment; called **exp**.
Provide a name.
For simplicity, we will start on a single host and only launch single-host jobs.

# SmartSim Example

```
from smartsim import Experiment

# Init Experiment and specify to launch locally
exp = Experiment(name="getting-started", launcher="local")

# settings to execute the command "echo hello!"
settings = exp.create_run_settings(exe="echo", exe_args="hello!", run_command=None)

# create the simple model instance so we can run it.
M1 = exp.create_model(name="tutorial-model", run_settings=settings)
```

# SmartSim Example

```
from smartsim import Experiment

# Init Experiment and specify to launch locally
exp = Experiment(name="getting-started", launcher="local")

# settings to execute the command "echo hello!"
settings = exp.create_run_settings(exe="echo", exe_args="hello!", run_command=None)

# create the simple model instance so we can run it.
M1 = exp.create_model(name="tutorial-model", run_settings=settings)
```

Experiment.create_run_settings is used to create a **RunSettings** instance for our **Model**. **RunSettings** describe how a **Model** should be executed provided the system and available computational resources.

Our first Model will simply print **hello** using the shell command **echo**.

# SmartSim Example

```
from smartsim import Experiment

# Init Experiment and specify to launch locally
exp = Experiment(name="getting-started", launcher="local")

# settings to execute the command "echo hello!"
settings = exp.create_run_settings(exe="echo", exe_args="hello!", run_command=None)

# create the simple model instance so we can run it.
M1 = exp.create_model(name="tutorial-model", run_settings=settings)

exp.start(M1, block=True, summary=True)
```

# SmartSim Example

```
from smartsim import Experiment

# Init Experiment and specify to launch locally
exp = Experiment(name="getting-started", launcher="local")

# settings to execute the command "echo hello!"
settings = exp.create_run_settings(exe="echo", exe_args="hello!", run_command=None)

# create the simple model instance so we can run it.
M1 = exp.create_model(name="tutorial-model", run_settings=settings)

exp.start(M1, block=True, summary=True)
```

Once the **Model** has been created by the **Experiment**, it can be **started**.
By setting **summary=True**, we can see a summary of the experiment printed before it is launched.
We also explicitly set **block=True** (even though it is the default), so that Experiment.start waits until the last **Model** has finished before returning.

# SmartSim Example

```
from smartsim import Experiment

# Init Experiment and specify to launch locally
exp = Experiment(name="getting-started", launcher="local")

# settings to execute the command "echo hello!"
settings = exp.create_run_settings(exe="echo", exe_args="hello!", run_command=None)

# create the simple model instance so we can run it.
M1 = exp.create_model(name="tutorial-model", run_settings=settings)

exp.start(M1, block=True, summary=True)

outputfile = './tutorial-model.out'
errorfile = './tutorial-model.err'

print("Content of tutorial-model.out:")
with open(outputfile, 'r') as fin:
    print(fin.read())
print("Content of tutorial-model.err:")
with open(errorfile, 'r') as fin:
    print(fin.read())
```

The model has completed and two output files have been created.
Normal output in **tutorial-model.out**
Error output in **tutorial-model.err**

# SmartSim Example

```
from smartsim import Experiment

# Init Experiment and specify to launch locally
exp = Experiment(name="getting-started", launcher="local")

# settings to execute the command "mpirun -np 2 echo hello world!"
openmpi_settings = exp.create_run_settings(exe="echo",
                                           exe_args="hello world!",
                                           run_command="mpirun")
openmpi_settings.set_tasks(2)

# create and start the MPI model
ompi_model = exp.create_model("tutorial-model-mpirun", openmpi_settings)
exp.start(ompi_model, summary=True)

exp.start(ompi_model, block=True, summary=True)

outputfile = './ tutorial-model-mpirun.out'

print("Content of tutorial-model-mpirun.out:")
with open(outputfile, 'r') as fin:
    print(fin.read())
```

# SmartSim Example

```
from smartsim import Experiment

# Init Experiment and specify to launch locally
exp = Experiment(name="getting-started", launcher="local")

# settings to execute the command "mpirun -np 2 echo hello world!"
openmpi_settings = exp.create_run_settings(exe="echo",
                                           exe_args="hello world!",
                                           run_command="mpirun")
openmpi_settings.set_tasks(2)

# create and start the MPI model
ompi_model = exp.create_model("tutorial-model-mpirun", openmpi_settings)
exp.start(ompi_model, summary=True)

exp.start(ompi_model, block=True, summary=True)

outputfile = './ tutorial-model-mpirun.out'

print("Content of tutorial-model-mpirun.out:")
with open(outputfile, 'r') as fin:
    print(fin.read())
```

Use **mpirun**