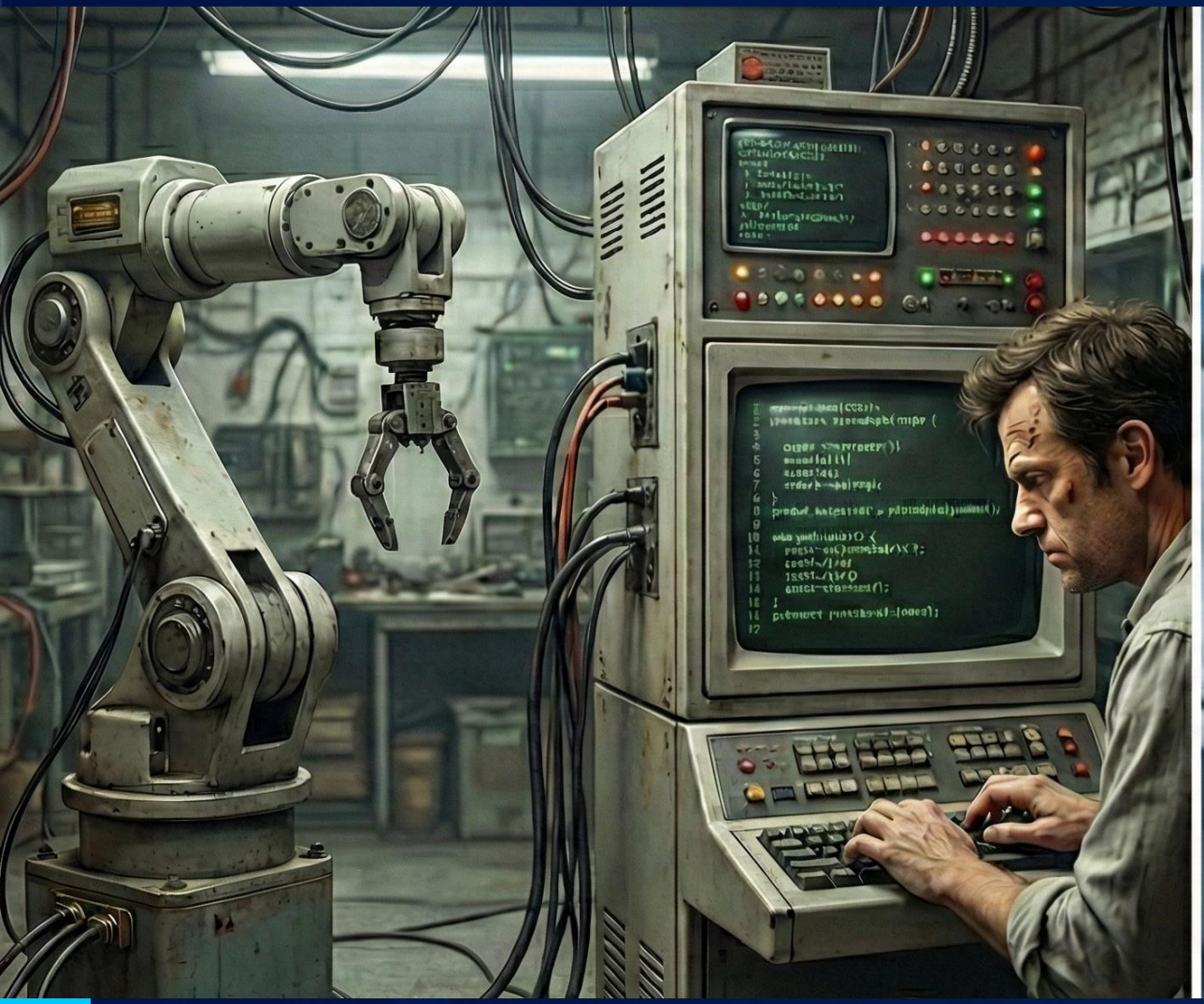
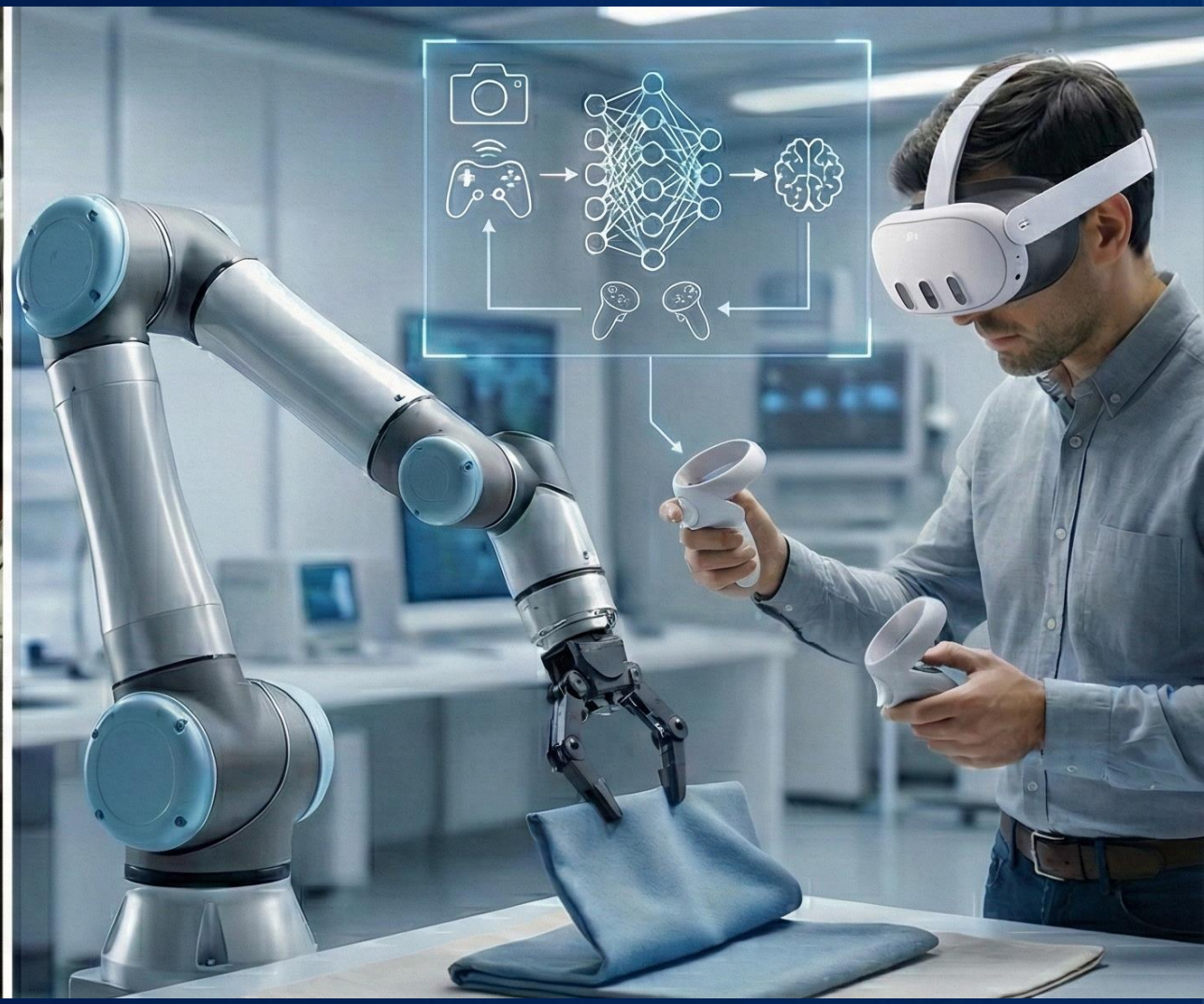
Full-Stack PyTorch Robotics VLA: From Data to Edge Via ExecuTorch/OpenVINO

Samet Akcay & Dmitriy Pastushenkov

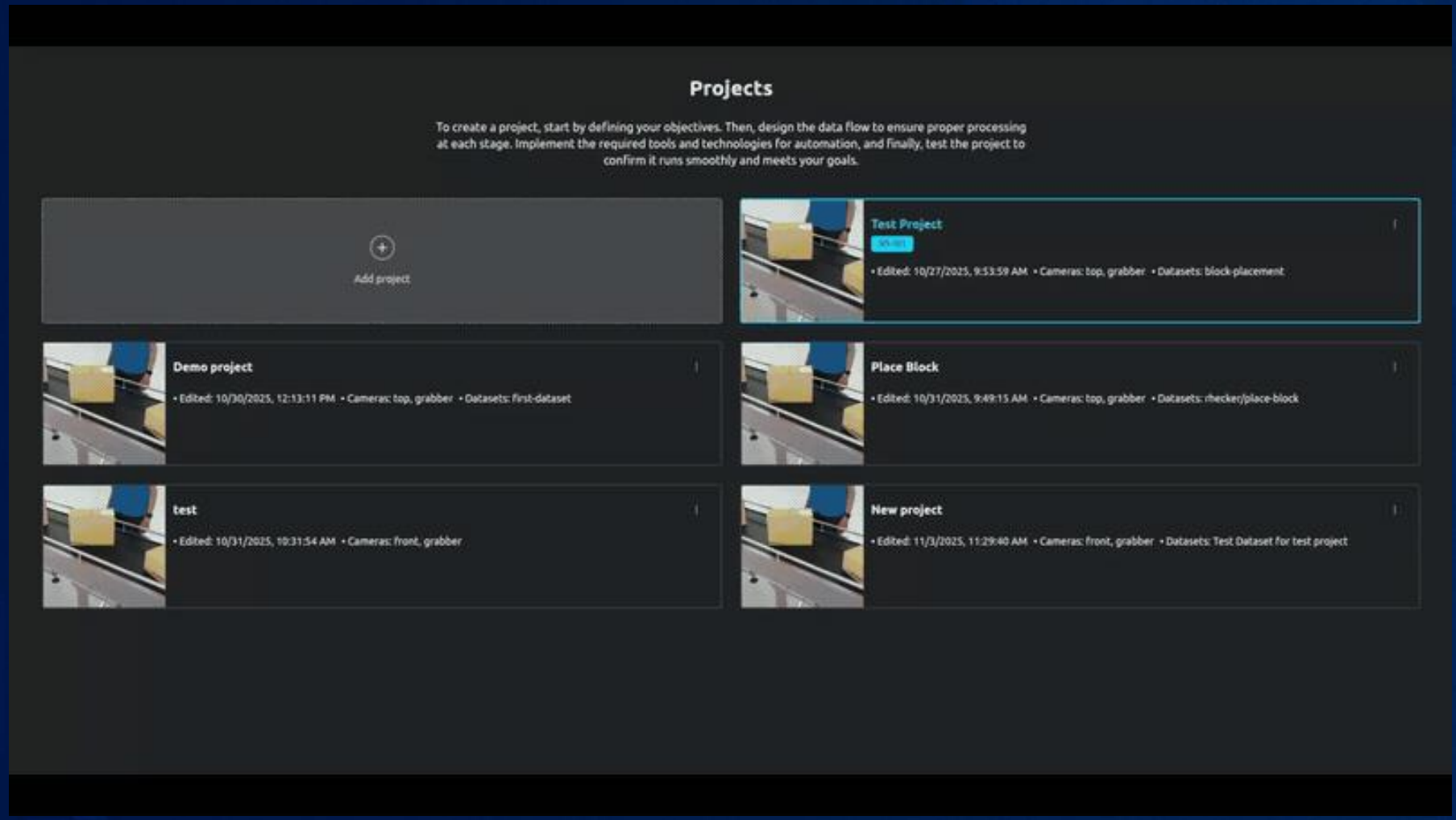
Robotics Then/Now





End-to-end Robot Learning via GUI, CLI or API

GUI Experience



CLI & API Experience

CLI

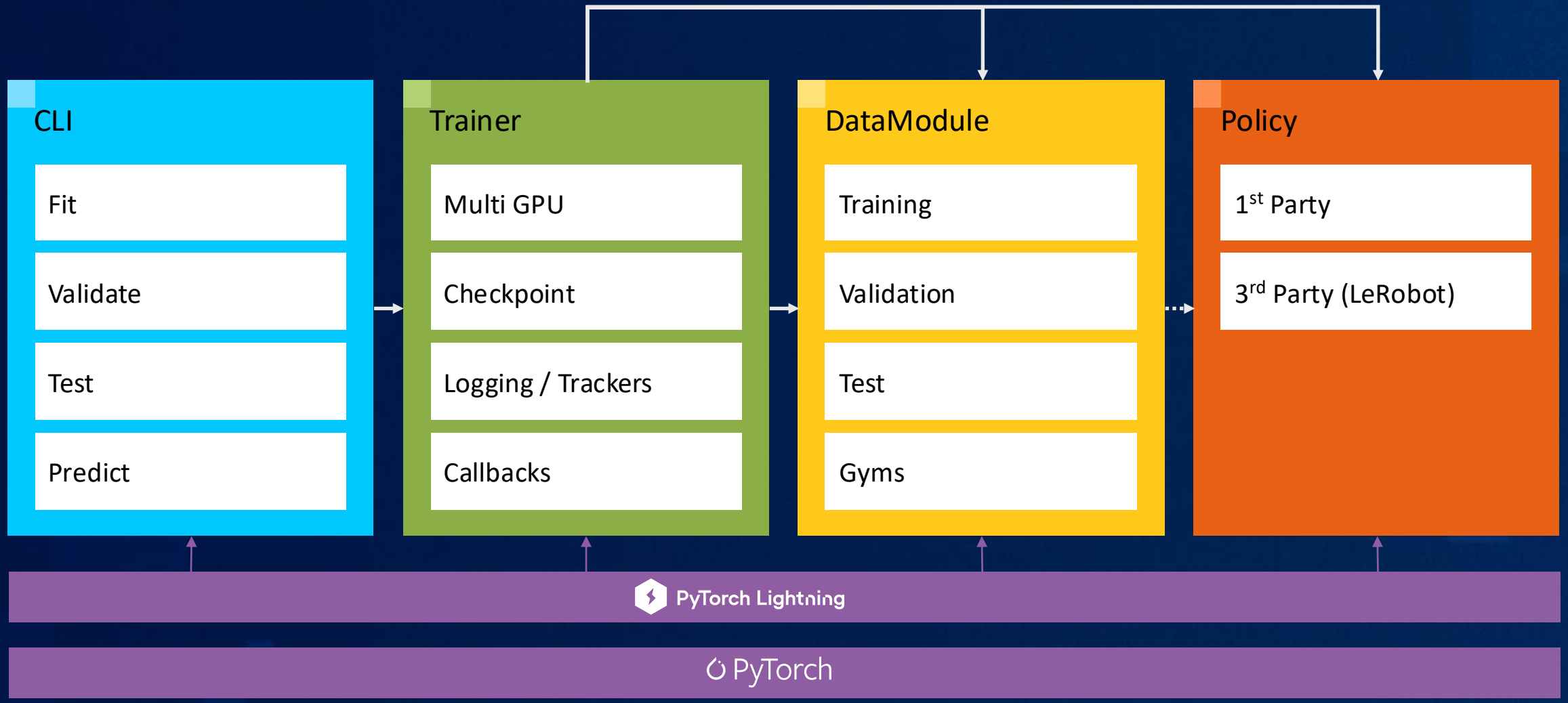
```yaml
data:
    class_path: physicalai.data.LeRobotDataModule
    init_args:
        repo_id: lerobot/aloha_sim_transfer_cube_human
        batch_size: 8

model:
    class_path: physicalai.policies.Pi05
    init_args:
        paligemma_variant: lerobot/pusht
        dtype: bfloat16
        chunk_size: 50
        learning_rate: 2.5e-5
```

```
❯ physicalai fit --config config.yaml
```

API

```python
from physicalai.data import LeRobotDataModule
from physicalai.policies import Pi05
from physicalai.train import Trainer

datamodule = LeRobotDataModule(
    repo_id="lerobot/aloha_sim_transfer_cube_human",
    batch_size=8,
)

policy = Pi05(
    paligemma_variant="gemma_2b",
    dtype="bfloat16",
    chunk_size=50,
    learning_rate=2.5e-5,
)

trainer = Trainer(max_steps=10000, accelerator="xpu")
trainer.fit(policy, datamodule=datamodule)
```

Library Architecture: High-Level



Built-in Best Practices

Hardware Agnostic

CPU/GPU/TPU, `accelerator="xpu"`

Multi GPU

One parameter: `devices=4`

Mixed Precision

One parameter, `precision="16-mixed"`

Checkpointing

Automatic best model saving

Logging

TensorBoard, WandB, CSV, TrackIO

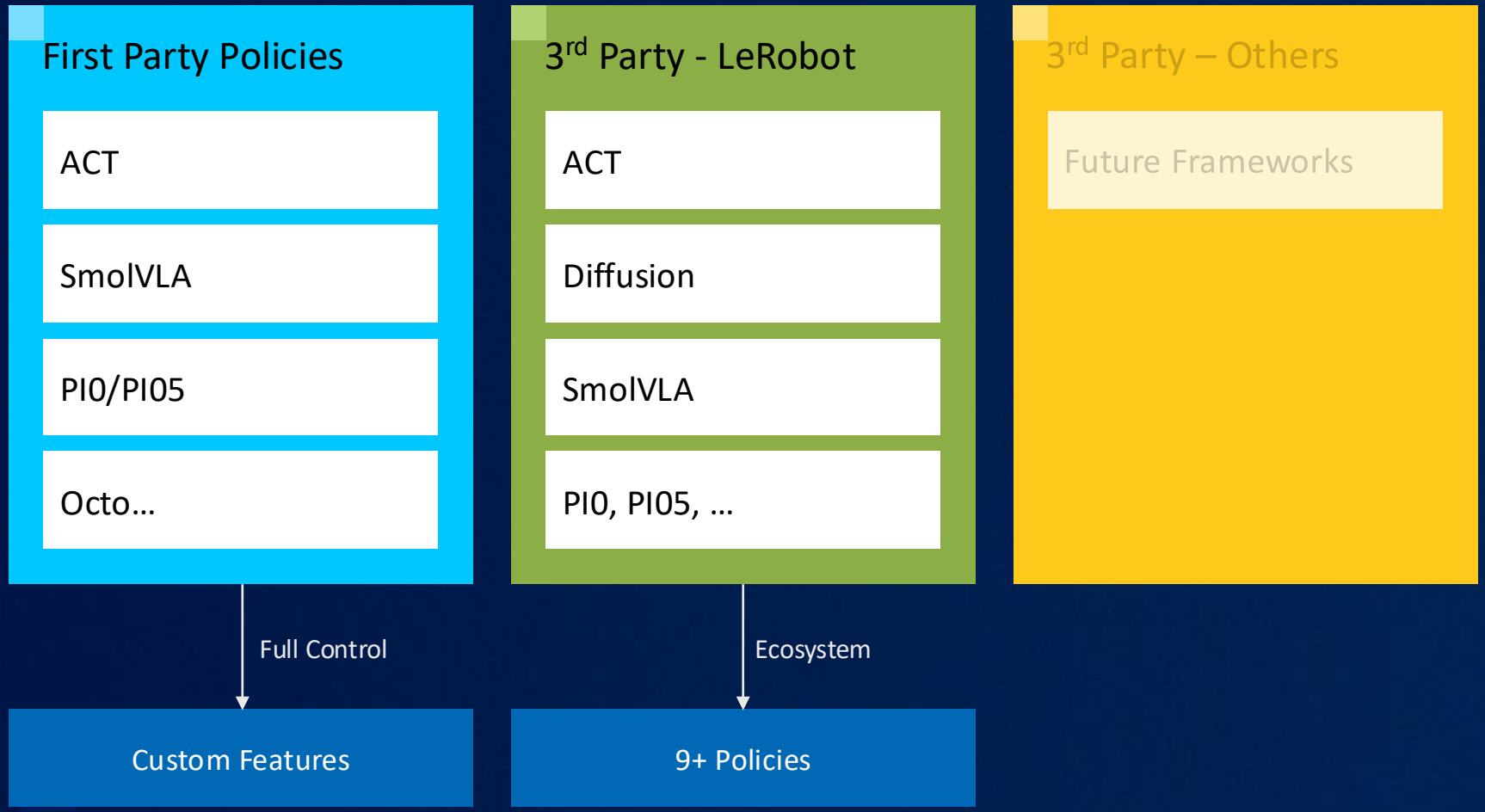
CLI

YAML configs with type validation

Result: Focus on research and product, not boilerplate inrastructure

Policy Architecture

1st and 3rd Party Policy Integration



First Party Policy Integration

```python
class ACT(Policy):
    """Our ACT implementation with custom features."""
    def __init__(self, chunk_size: int = 100, ...):
        super().__init__()
        self.encoder = TransformerEncoder(...)
        self.decoder = TransformerDecoder(...)
    def forward(self, batch: Observation) -> Tensor:
        encoded = self.encoder(batch.images["top"])
        decoded = self.decoder(encoded, batch.states)
        return self.action_head(decoded)
    def training_step(self, batch, idx):
        actions = self.forward(batch)
        loss = F.mse_loss(actions, batch.actions)
        self.log("train_loss", loss)
        return loss
```

Benefits: Full control, Custom Features, Optimised

Third Party Policy Integration

LeRobot Universal Wrapper: One for All

```
class LeRobotPolicy(Policy):
    """Universal wrapper for ANY LeRobot policy."""
    def __init__(self, policy_name: str, config_kwargs: dict = None):
        self.policy_name = policy_name # "act", "diffusion", etc.

def setup(self, stage: str):
        # Called by Lightning before training
        # 1. Get policy class dynamically
        policy_cls = get_policy_class(self.policy_name)
        # 2. Extract features from dataset
        dataset = self.trainer.datamodule.train_dataset
        input_features = extract_features(dataset, "input")
        output_features = extract_features(dataset, "output")
        # 3. Build config and instantiate
        config = make_policy_config(
            self.policy_name,
            input_features,
            output_features,
            **self._config_kwargs
        )
        self._policy = policy_cls(config, dataset.stats)
```

Third Party Policy Integration

LeRobot Universal Wrapper: One for All

CLI

```yaml
data:
    class_path: physicalai.data.LeRobotDataModule
    init_args:
        repo_id: lerobot/pusht
        data_format: lerobot


model:
    class_path: physicalai.policies.LeRobotPolicy
    init_args:
        policy_name: diffusion # ← Just change this!
        config_kwargs:
            num_steps: 100
            noise_scheduler: ddpm
```

```
❯ physicalai fit --config config.yaml
```

API

```python
from physicalai.data import LeRobotDataModule
from physicalai.policies import LeRobotPolicy
from physicalai.train import Trainer

datamodule = LeRobotDataModule(
    repo_id="lerobot/ pusht, data_format="lerobot"
)

policy = LeRobotPolicy(
    policy_name="diffusion",
    config_kwargs={
        "num_steps": 100,
        "noise_scheduler": "ddpm",
    },
)

trainer = Trainer(max_epochs=10, accelerator="xpu")
trainer.fit(policy, datamodule=datamodule)
```

Try different LeRobot policies without code changes!

Third Party Policy Integration

LeRobot Explicit Wrapper

CLI

```yaml
data:
    class_path: physicalai.data.LeRobotDataModule
    init_args:
        repo_id: lerobot/pusht
        batch_size: 64

model:
    class_path: physicalai.policies.lerobot.Diffusion
    init_args:
        n_obs_steps: 2
        horizon: 16
        n_action_steps: 8
        down_dims: [512, 1024, 2048]
```

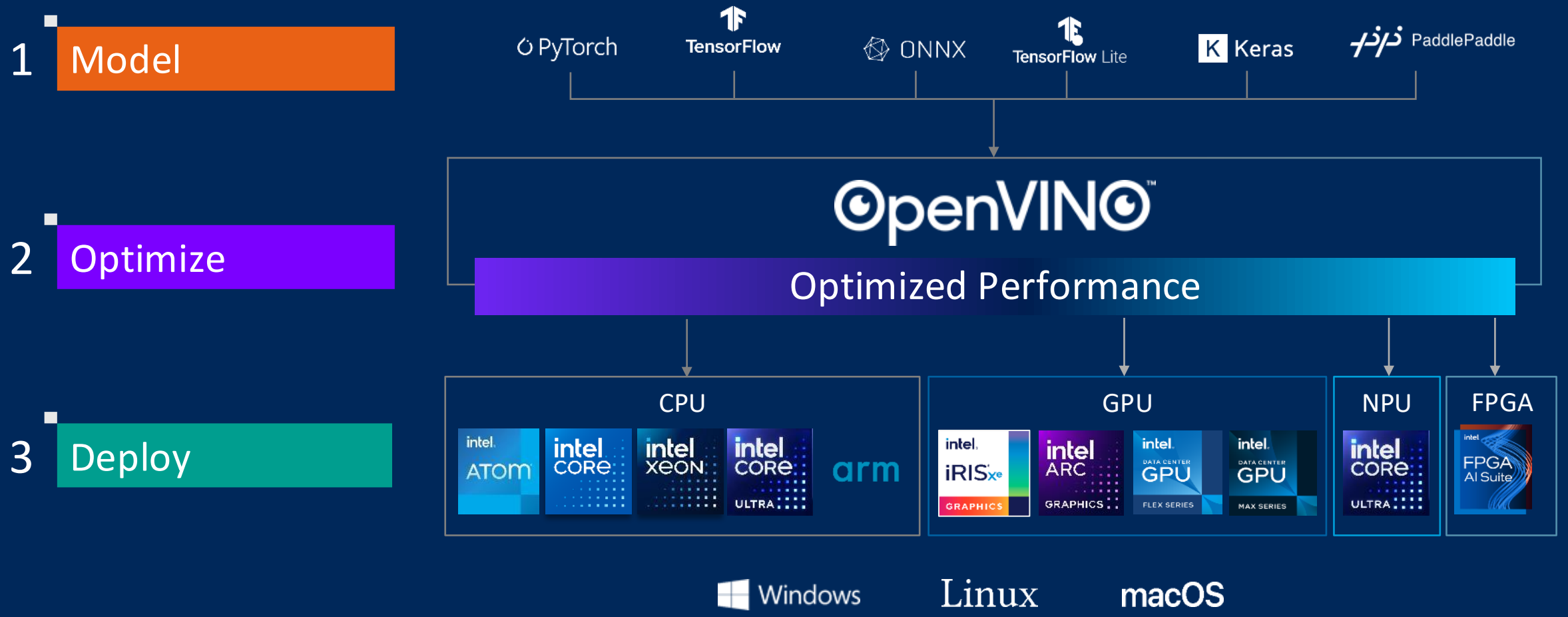
```
❯ physicalai fit --config config.yaml
```

API

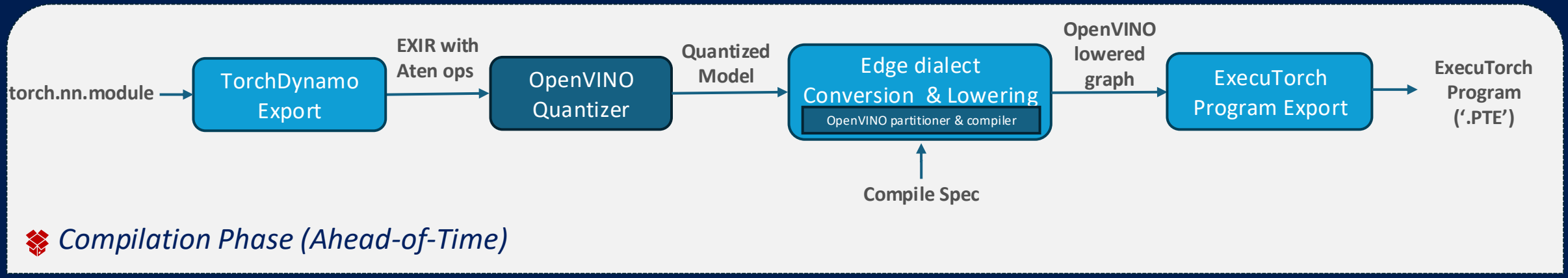
```python
from physicalai.data import LeRobotDataModule
from physicalai.policies.lerobot import Diffusion
from physicalai.train import Trainer

datamodule = LeRobotDataModule(
    repo_id="lerobot/pusht ",
    batch_size=64,
)

policy = Diffusion(
    n_obs_steps=2,
    horizon=16,
    n_action_steps=8,
    down_dims=(256, 512, 1024),
)

trainer = Trainer(max_epochs=10, accelerator="xpu")
trainer.fit(policy, datamodule=datamodule)
```

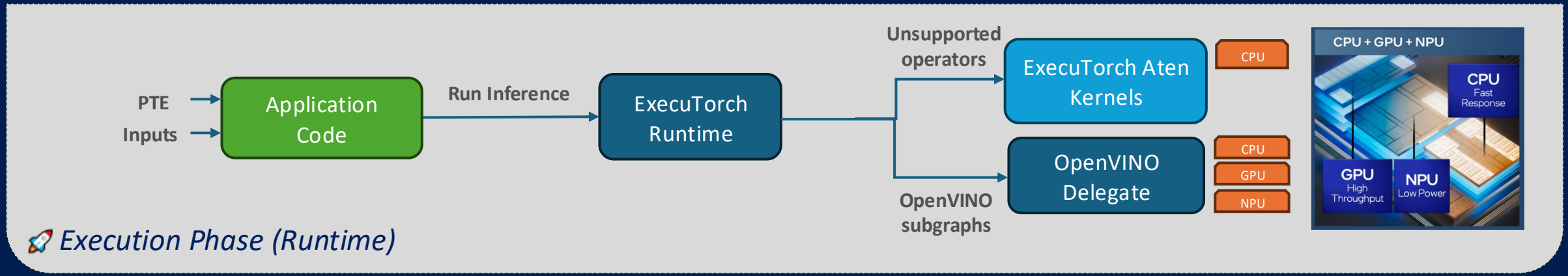
Inference Workflow

Optimized AI Performance on Intel Hardware



Optimized AI Performance on Intel Hardware





Key Features

- Optimized for the Client
- Maximum Performance
- Deploy with Confidence
- Seamless PyTorch Workflow

Key Benefits

- Platform Support
- Quantization
- Model Support
- Configurability

Model Export

```
trainer.fit(policy, datamodule)
# Training done!

# Load best checkpoint
best = trainer.checkpoint_callback.best_model_path
policy = Policy.load_from_checkpoint(best)
policy.eval()

# Export options
policy.export("./model", backend="openvino")
policy.to_openvino("model.bin", example_input)

policy.export("./model", backend="executorch")
policy.to_executorch("model.pte", example_input)
```

Physical AI Framework: High-Level Overview

Unified Inference: **A lightweight package with CLI and API parity**

Installation

```
❯ pip install physicalai
```

Inference via CLI

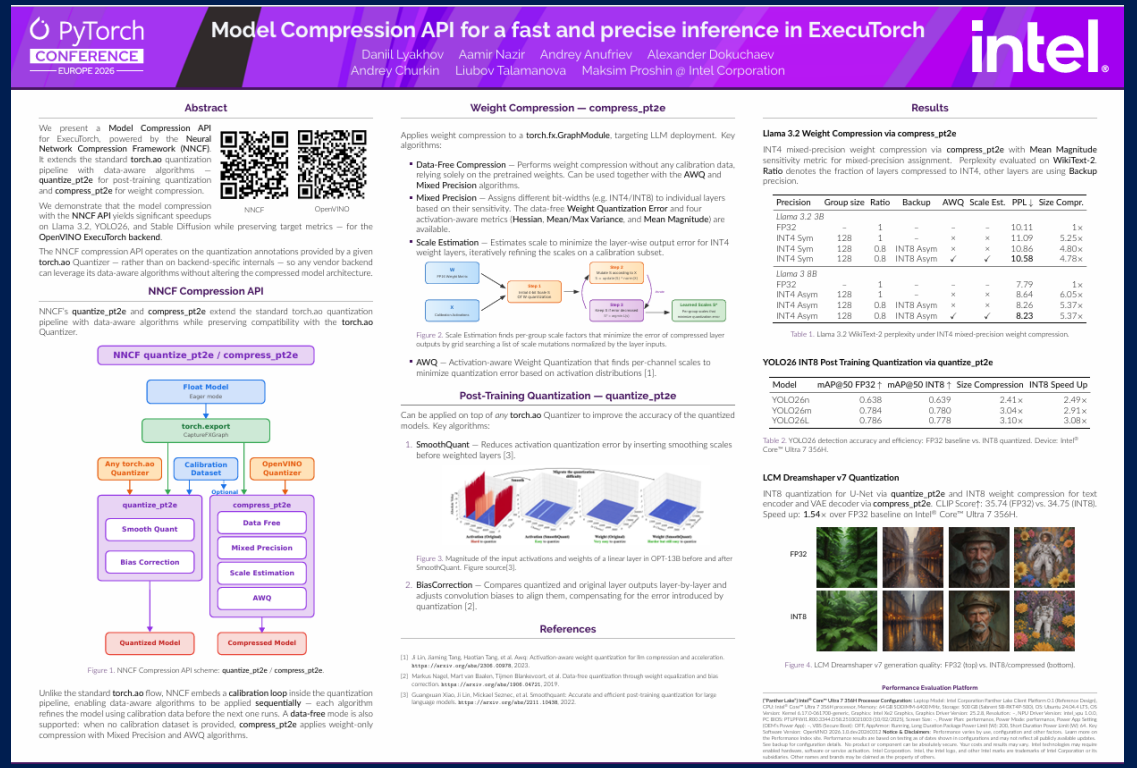
```
❯ physicalai run \     # or phyai
    --model hf://getiaction/act_policy \
    --robot robot.yaml
```

Inference via API

```
from physicalai.inference import InferenceModel
from physicalai.robots import SO101

policy = InferenceModel("./exports/act_policy")
robot = SO101.from_config("robot.yaml")

with robot:
    obs = robot.get_observation()
    action = policy.select_action(obs)
    robot.send_action(action)
```

Read the blog

Visit the poster



PyTorch CONFERENCE EUROPE 2026

Model Compression API for a fast and precise inference in ExecuTorch

Daniil Lyakhov Aamir Nazir Andrey Anufriev Alexander Dokuchaev
Andrey Churkin Liubov Talamanova Maksim Proshin @ Intel Corporation

intel

Abstract

We present a **Model Compression API** for ExecuTorch, powered by the **Neural Network Compression Framework (NNCF)**. It extends the standard **torch.ao** quantization pipeline with data-aware algorithms — **quantize_pt2e** for post-training quantization and **compress_pt2e** for weight compression.

We demonstrate that the model compression with the **NNCF API** yields significant speedups on Llama 3.2, YOLO26, and Stable Diffusion while preserving target metrics — for the **OpenVINO ExecuTorch backend**.

The NNCF compression API operates on the quantization annotations provided by a given **torch.ao** Quantizer — rather than on backend-specific internals — so any vendor backend can leverage its data-aware algorithms without altering the compressed model architecture.

NNCF OpenVINO

NNCF Compression API

NNCF's **quantize_pt2e** and **compress_pt2e** extend the standard torch.ao quantization pipeline with data-aware algorithms while preserving compatibility with the **torch.ao** Quantizer.

Figure 1. NNCF Compression API scheme: **quantize_pt2e** / **compress_pt2e**.

Unlike the standard **torch.ao** flow, NNCF embeds a **calibration loop** inside the quantization pipeline, enabling data-aware algorithms to be applied **sequentially** — each algorithm refines the model using calibration data before the next one runs. A **data-free** mode is also supported: when no calibration dataset is provided, **compress_pt2e** applies weight-only compression with Mixed Precision and AWQ algorithms.

Weight Compression — compress_pt2e

Applies weight compression to a **torch.fx.GraphModule**, targeting LLM deployment. Key algorithms:

- **Data-Free Compression** — Performs weight compression without any calibration data, relying solely on the pretrained weights. Can be used together with the **AWQ** and **Mixed Precision** algorithms.
- **Mixed Precision** — Assigns different bit-widths (e.g. INT4/INT8) to individual layers based on their sensitivity. The data-free **Weight Quantization Error** and four activation-aware metrics (**Hessian**, **Mean/Max Variance**, and **Mean Magnitude**) are available.
- **Scale Estimation** — Estimates scale to minimize the layer-wise output error for INT4 weight layers, iteratively refining the scales on a calibration subset.

Figure 2. Scale Estimation finds per-group scale factors that minimize the error of compressed layer outputs by grid searching a list of scale mutations normalized by the layer inputs.

- **AWQ** — Activation-aware Weight Quantization that finds per-channel scales to minimize quantization error based on activation distributions [1].

Post-Training Quantization — quantize_pt2e

Can be applied on top of any **torch.ao** Quantizer to improve the accuracy of the quantized models. Key algorithms:

1. **SmoothQuant** — Reduces activation quantization error by inserting smoothing scales before weighted layers [3].

Figure 3. Magnitude of the input activations and weights of a linear layer in OPT-13B before and after SmoothQuant. Figure source[3].

2. **BiasCorrection** — Compares quantized and original layer outputs layer-by-layer and adjusts convolution biases to align them, compensating for the error introduced by quantization [2].

References

[1] J Lin, Jiaming Tang, Haotian Tang, et al. Awq: Activation-aware weight quantization for llm compression and acceleration. https://arxiv.org/abs/2306.00978, 2023

[2] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, et al. Data-free quantization through weight equalization and bias correction. https://arxiv.org/abs/1906.04721, 2019.

[3] Guangxuan Xiao, Ji Lin, Mickael Seznec, et al. SmoothQuant: Accurate and efficient post-training quantization for large language models. https://arxiv.org/abs/2211.10438, 2022.

Results

Llama 3.2 Weight Compression via compress_pt2e

INT4 mixed-precision weight compression via **compress_pt2e** with **Mean Magnitude** sensitivity metric for mixed-precision assignment. Perplexity evaluated on **WikiText-2**. **Ratio** denotes the fraction of layers compressed to INT4, other layers are using **Backup** precision.

Precision	Group size	Ratio	Backup	AWQ	Scale Est.	PPL ↓	Size Compr.
Llama 3.2 3B							
FP32	–	1	–	–	–	10.11	1×
INT4 Sym	128	1	–	×	×	11.09	5.25×
INT4 Sym	128	0.8	INT8 Asym	×	×	10.86	4.80×
INT4 Sym	128	0.8	INT8 Asym	✓	✓	**10.58**	4.78×
Llama 3 8B							
FP32	–	1	–	–	–	7.79	1×
INT4 Asym	128	1	–	×	×	8.64	6.05×
INT4 Asym	128	0.8	INT8 Asym	×	×	8.26	5.37×
INT4 Asym	128	0.8	INT8 Asym	✓	✓	**8.23**	5.37×

Table 1. Llama 3.2 WikiText-2 perplexity under INT4 mixed-precision weight compression.

YOLO26 INT8 Post Training Quantization via quantize_pt2e

Model	mAP@50 FP32 ↑	mAP@50 INT8 ↑	Size Compression	INT8 Speed Up
YOLO26n	0.638	0.639	2.41×	2.49×
YOLO26m	0.784	0.780	3.04×	2.91×
YOLO26L	0.786	0.778	3.10×	3.08×

Table 2. YOLO26 detection accuracy and efficiency: FP32 baseline vs. INT8 quantized. Device: Intel® Core™ Ultra 7 356H.

LCM Dreamshaper v7 Quantization

INT8 quantization for U-Net via **quantize_pt2e** and INT8 weight compression for text encoder and VAE decoder via **compress_pt2e**. CLIP Score↑: 35.74 (FP32) vs. 34.75 (INT8). Speed up: **1.54**× over FP32 baseline on Intel® Core™ Ultra 7 356H.

Figure 4. LCM Dreamshaper v7 generation quality: FP32 (top) vs. INT8/compressed (bottom).

Performance Evaluation Platform

("Panther Lake") Intel® Core™ Ultra 7 356H Processor Configuration: Laptop Model: Intel Corporation Panther Lake Client Platform 0.1 (Reference Design), CPU: Intel® Core™ Ultra 7 356H (Performance-Memory: 64 GiB 9200 MHz 6400 MHz, Storage: 930 GB (Sabrent SB-RKT4P-1001), OS: Ubuntu 24.04.4 LTS, OS Version: Kernel 6.17.0-061700-generic, Graphics: Intel Xe2 Graphics, Graphics Driver Version: 25.2.0, Resolution: -, NPU Driver Version: Intel, uay, 1.0.0, PC BIOS: PTLPFWI1.R00.3344.D38.2510021003 (10/02/2025), Screen Size: -, Power Plan: performance, Power Mode: performance, Power App: Setting (OEM's Power App): -, VBS (Secure Boot): OFF, Appliance: Running, Long Duration Package Power Limit (W): 300, Short Duration Power Limit (W): 64, Key Software Version: OpenVINO 2026.1.0-dev20260217, **Notice & Disclaimers**: Performance varies by use, configuration and other factors. Learn more on the Performance Index site. Performance results are based on testing as of dates shown in configurations and may not reflect all publicly-available updates. See backup for configuration details. No product or component can be absolutely secure. Your costs and results may vary. Intel technologies may require enabled hardware, software or service activation. Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

OpenVINO™

intel ai


intel