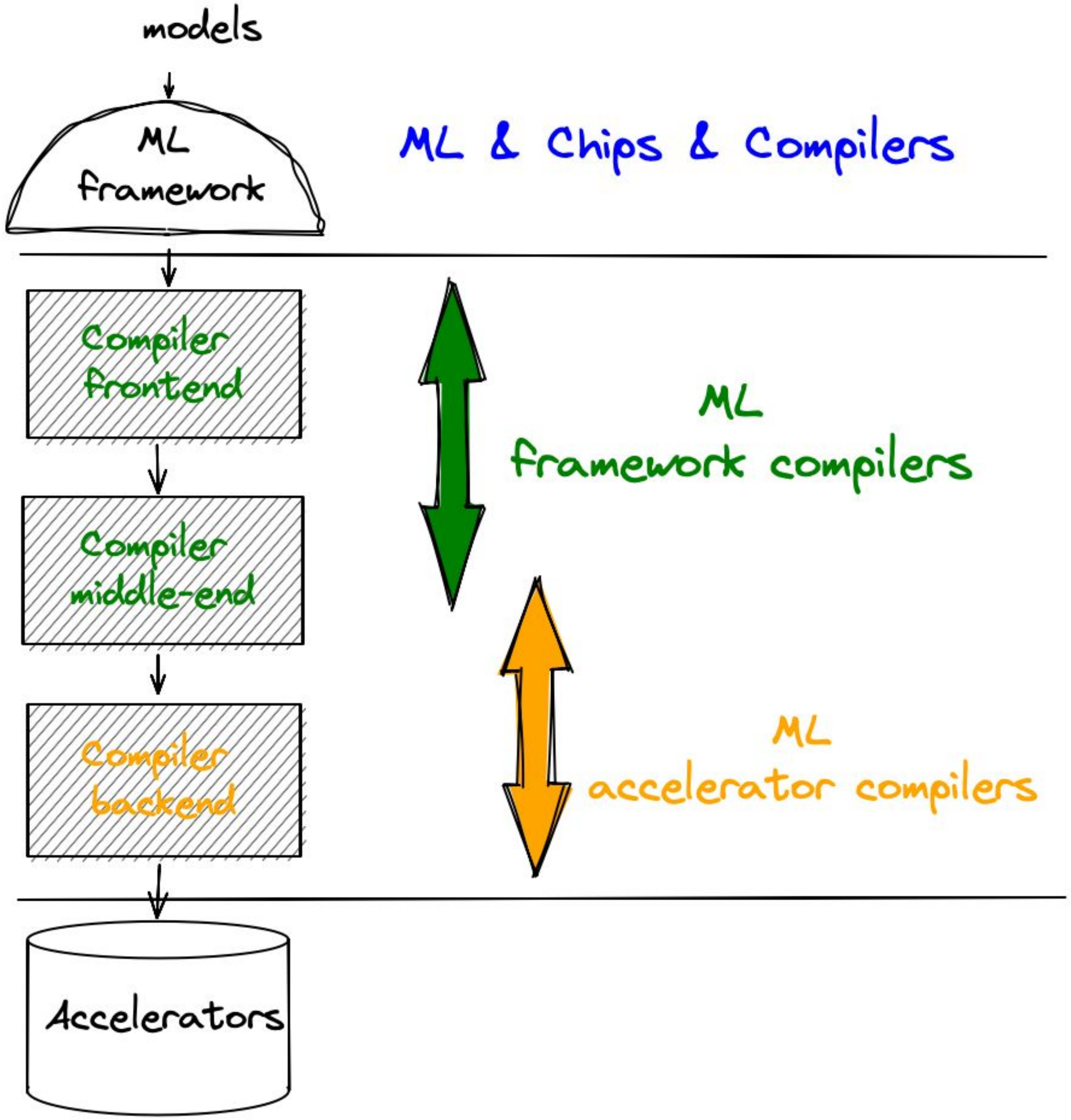
What makes PyTorch beloved makes it hard to compile

– The nuanced story of PyTorch Compiler(s)

[Peng Wu](pengwu@fb.com) (pengwu@fb.com)
Engineering Manager, PyTorch Compilers

Meta AI



A ML framework's **design philosophy** determines **unique challenges** of its framework compiler

- **PyTorch graph capture is non-trivial**
- **Impedance mismatch** between accelerator and PyTorch opsets, e.g.,
  - non functional ops (inplace, view)
  - dynamic shape
  - data-dependent control-flow
- **Large op surface** makes it hard to achieve complete IR coverage

Why should you care about PyTorch graph capture?

Chip designers

Eager-mode execution is considered **prohibitively costly** for accelerators

Production engineers

Most PyTorch **inference deployments** are ***exported* out of Eager** via graph capture

Compiler engineers

No graph no compiler

Why are PyTorch Compilers plural?

- **TorchScript (torch.jit.trace, torch.jit.script)**, Static Runtime, Lite Interpreter
- nnc, nvfuser
- **torch.fx** (incl. fxtrt, fxacc, fxait)
- torch.package, torch.deploy
- torch-mlir, pytorch/xla, **Lazy Tensor Core**
- **TorchDynamo**, TorchInductor
- …

TorchScript

- TS frontend supports **subset of Python w/ user-annotated types**;
- **TS IR** supports **aten ops, control-flow**, **mutation**, complex data types;
- TS middle-end does IR cleansing, property propagation, and optimizations;
- TS is **executed by TS interpreter** – *exported out of Python*

```
class MyCell(torch.nn.Module):
    def __init__(self):
        super(MyCell, self).__init__()
        self.linear = torch.nn.Linear(4, 4)

    def forward(self, x: Tensor, h: Tensor):
        new_h = torch.tanh(self.linear(x) + h)
        return new_h, new_h

my_cell = MyCell()
x, h = torch.rand(3, 4), torch.rand(3, 4)
scripted_cell = torch.jit.script(my_cell)
scripted_cell(x, h)
```

TorchScript – the very 1st PyTorch Compiler

- **Frontend**: **ahead-of-time**, **whole-graph** capture
  - Capture once replay many times
- **Deployment**: **export-path**
  - good for inference-at-scale and edge devices

Limitations

- **UX**: either not out-of-box (e.g., scripting) or unsound (e.g., tracing)
- **Training**: support incomplete

TorchDynamo – the 1st out-of-the-box PyTorch graph capture

Dynamo + a good backend makes *unmodified* PyTorch models faster

TorchDynamo vs TorchScript FE

- Dynamo does not require changing the model (aka out-of-the-box capture)
  - Dynamo captures partial graphs and falls back to eager
  - Dynamo captures graphs with guards and recapture when guards mismatch replay
- Dynamo reliably captures backward graphs (aka training)

An Example

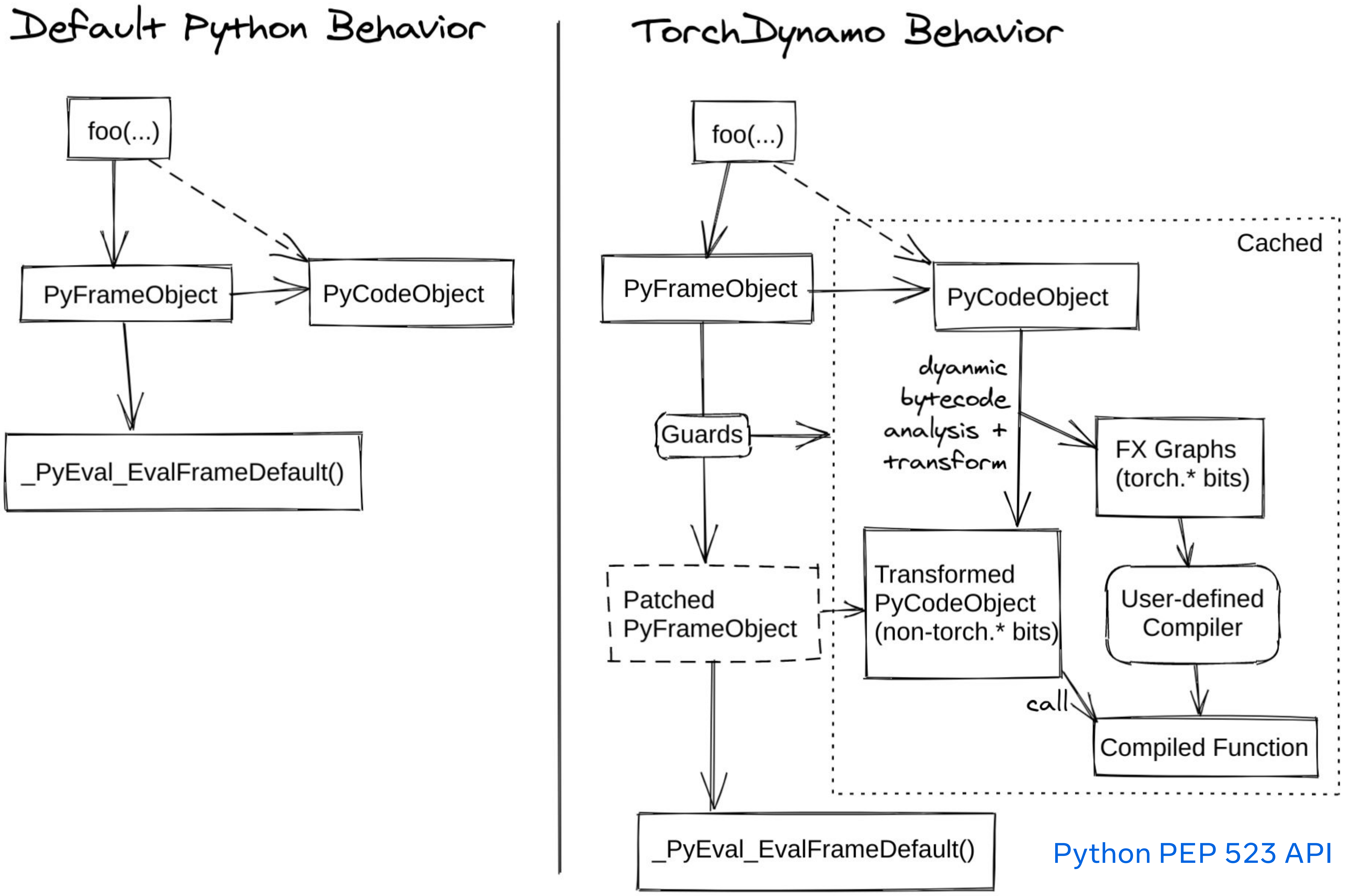
```python
from typing import List
import torch
import torchdynamo

def toy_example(a, b):
    x = a / (torch.abs(a) + 1)
    if b.sum() < 0:
        b = b * -1
    return x * b

def my_compiler(gm: torch.fx.GraphModule,
                example_inputs: List[torch.Tensor]):
    print("my_compiler() called with FX graph:")
    gm.graph.print_tabular()
    return gm.forward  # return a python callable

with torchdynamo.optimize(my_compiler):
    for _ in range(100):
        toy_example(torch.randn(10),
                    torch.randn(10))
```

Example Output

```
my_compiler() called with FX graph:
opcode         name     target                   args
-------------  -------  -----------------------  -----------------
placeholder    a        a                        ()
placeholder    b        b                        ()
call_function  abs_1    torch.abs                (a,)
call_function  add      operator.add             (abs_1, 1)
call_function  truediv  operator.truediv         (a, add)
call_method    sum_1    sum                      (b,)
call_function  lt       operator.lt              (sum_1, 0)
output         output   output                   ((truediv, lt),)
```

```
my_compiler() called with FX graph:
opcode         name    target                   args
-------------  ------  -----------------------  -----------
placeholder    b       b                        ()
placeholder    x       x                        ()
call_function  mul     operator.mul             (b, -1)
call_function  mul_1   operator.mul             (x, mul)
output         output  output                   ((mul_1,),)
```

```
my_compiler() called with FX graph:
opcode         name    target                   args
-------------  ------  -----------------------  ---------
placeholder    b       b                        ()
placeholder    x       x                        ()
call_function  mul     operator.mul             (x, b)
output         output  output                   ((mul,),)
```



Python PEP 523 API

What makes TorchDynamo graph capture sound and out-of-the-box?

	Soundness characteristics
Partial graph capture	Ability to skip unwanted parts of eager
Guarded graphs	Ability to check if captured graph is valid for execution
Just-in-time recapture	recapture a graph if captured graph is invalid for execution

AOT Autograd – Get Backward Graph from Forward

- TorchDynamo captures the forwards
- Backwards in PyTorch is done through dynamic autograd tape
- We need to capture the dynamic autograd behavior at compile time

AOT Autograd

- Traces the behavior of the PyTorch autograd tape
- Works on partial graph fragments

Creating New Backends is Easy

```
def my_compiler(gm: torch.fx.GraphModule,
                example_inputs: List[torch.Tensor]):
    scripted = torch.jit.trace(gm, example_inputs)
    return torch.jit.optimize_for_inference(scripted)
```

- **Dynamo workflow**
  - Capture FX graphs
  - Passe FX graphs to registered compiler hook to compile
  - Executes the Callable objects returned by invoking the compiler hook
- **Custom compiler hooks can be other PyTorch compilers**
  - e.g., Dynamo + torch.jit.trace, Dynamo + TRT, Dynamo + Cudagraph
  - e.g., Dynamo + LTC

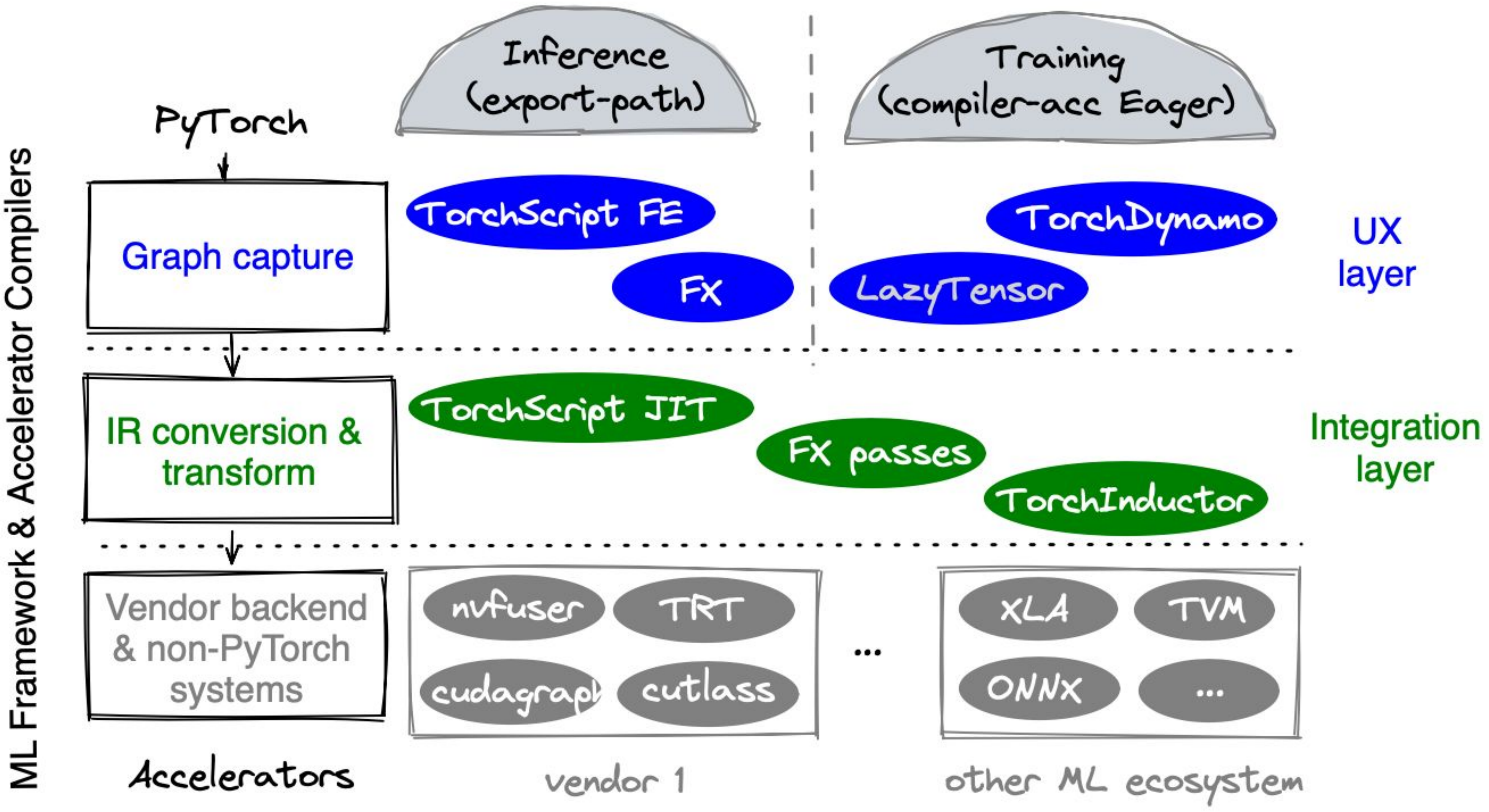
TorchDynamo today

- **OOTB graph capture demonstrated** on
  - 7K+ crawled github models
- **Easy backend integration demonstrated**
  - 20+ inference backends (e.g., TS, TRT, LTC)
  - 2 training backends (nvfuser, TorchInductor)
- **Training speedup demonstrated w/ Just-in-time partial graph capture**
  - 30%+ geomean OOTB speedup over TB, TIMM, HF benchmarks (150+ models, single-node, A100)

Ongoing work

- Hardening & tools
- Dynamic shape
- Distributed
- Recompilation UX improv.
- Whole-graph mode

Mental models of PyTorch Compilers



When to use which graph capture?

Current recommendations

- For training
  - **All (but XLA or TPU) ⇒ Dynamo**
  - Export to XLA or TPU ⇒ Lazy Tensor
- For inference
  - **Embedded ⇒ TS**
  - Non-embedded ⇒ TS or FX
- For human-in-the-loop tools ⇒ **FX**

Vision for the future?

- Consolidate graph capture across eager and export-path for a smooth UX

Take-aways

- PyTorch's Eager-first design makes **graph capture a unique challenge**
  - **TorchDynamo** – 3rd-gen PT compiler FE but **1st out-of-the-box** one
- With more models *effortlessly* funnelled into graph mode, **the era of compiler-accelerated PyTorch is coming**
  - training -> dynamic shape -> prim -> export-path -> distributed
- **Mindshifts** for ML chip and compiler designers
  - from whole graphs to **partial graphs**
  - from export-path deployment to **eager or eager-export-hybrid deployment**

For more information

- Repo – https://github.com/pytorch/torchdynamo
- PyTorch Dev Discussion – compiler category

Meta AI

Mental models of PyTorch Compilers

Is it a **frontend** (graph capture), a **middle-end** (graph compiler), a **backend**, or a **runtime**?

1. On frontend
   a. Is it for **export-path deployment** (e.g., inference) or **eager deployment** (e.g., training)?
   b. Does it require **(ahead-of-time) whole-graph** or **(just-in-time) partial-graph** capture?
2. On middle-end
   a. Is it a **torch-native** graph optimizer or just a bridge to another IR?
3. On runtime
   a. Does it implement **torch-native opsets or not**?
   b. Is it driven by **Eager**?