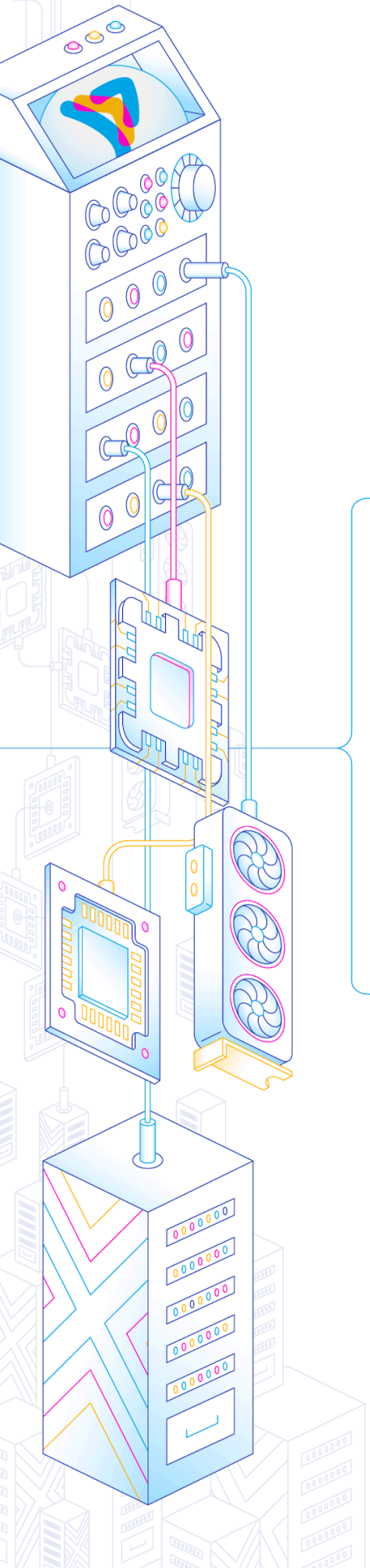


XANADU × AMD

Unlocking real-time speed for the entire quantum ecosystem with PennyLane and Backline

UNLOCKING REAL-TIME SPEED FOR THE ENTIRE QUANTUM ECOSYSTEM WITH PENNYLANE AND BACKLINE

AUTHORS

Josh Izaac, Ben Lau, Lee O'Riordan, Joseph Lee, Mehrdad Malek, Hongsheng Zheng, Cheick Doumbia, Leonhard Neuhaus, Thomas Germain, Thomas Ainsworth, Mehran Zamani, Kalman Szenes, Shuli Shu, Jose M Monsalve Diaz, Muhammad Osama, Yasuko Eckert, Paul Hartke

GRAPHICS

Tarik El-Khateeb

To learn more, visit

Xanadu.ai

AMD.com

© 2026 Xanadu Quantum Technologies Inc. and Advanced Micro Devices, Inc.

This work is licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0).

Instinct, EPYC, ERNIC, Ryzen, Versal, and Threadripper are trademarks of Advanced Micro Devices, Inc.

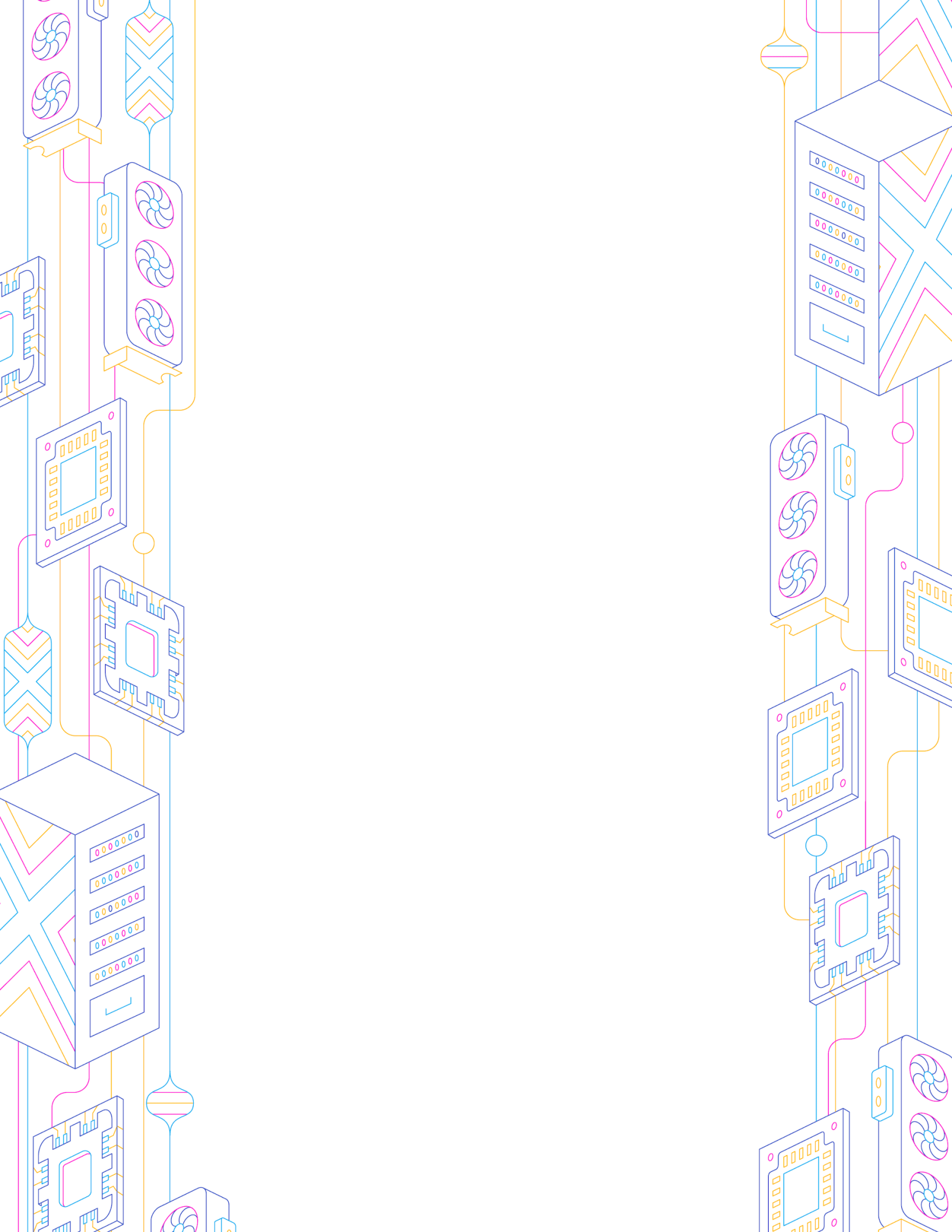
Today, quantum is never just quantum. Moving from quantum R&D to production execution is one of the hardest challenges in the quantum industry at-large — across **quantum computing**, **networking** and **sensing**. Real-time quantum error correction (QEC) demands strict microsecond latency that traditional Python environments simply can't hit.

Today, we are changing that. Meet **Backline**.

Co-designed by Xanadu and AMD, Backline is an open platform for compilation and low-latency execution that dynamically connects quantum workloads to the right classical engine. Further, it is available today in PennyLane, bundling the underlying compiler components so that anyone can build and deploy to quantum and classical hardware. Whether you are a quantum researcher, systems engineer, or hardware vendor, you can use Backline to power your stack.

Design and deploy with Backline — from meaningful quantum applications to quantum error correction.

- **Ultra-low latency**: Supports end-to-end latency looping in under 3-microseconds.
- **All accelerators**: Connects AMD adaptive SOCs and FPGAs, RDMA NICs, AMD Instinct™ GPUs, and AMD EPYC™ and Ryzen™ Threadripper™ CPUs with quantum hardware.
- **Python-first**: Use Python to build quantum applications for production-grade execution across quantum and classical hardware.
- **Infrastructure agnostic and open source**: An open platform for the entire quantum ecosystem.
- **Scale from R&D to production:** Begin prototyping immediately with CPUs. Progressively include consumer-grade and enterprise GPUs, FPGAs, and ASICs — all from the same software environment.



Contents

The quantum execution bottleneck 1
Photonic quantum computing 2
The heterogeneous ecosystem 3
The Python paradox 5
The myth of universal low-latency 6
Backline: an open platform for the quantum community 8
Who is Backline for? Accelerating innovation across disciplines 9
The myth of GPUs everywhere: lowering the barrier to entry 10
Inside Backline 12
Applications and performance 13
Local CPU-to-CPU interaction with a pre-compiled kernel 13
Remote CPU-to-GPU interaction with a Python defined kernel 14
Remote GPU-to-FPGA interaction with a pre-compiled kernel 20
Bring your own Triton decoder 22
Pushing the Limits 23
Expanding the platform 26
Conclusion 27
Getting started 27

The quantum execution bottleneck

Quantum computing promises to solve problems that would take today's best supercomputers thousands of years to unravel. But behind the headline-grabbing promises lies a quiet reality today: **a quantum computer is never purely quantum**. Instead, we rely on a vast array of classical computation and infrastructure, from processing of applications to low-level correction of quantum errors.

To make the move from research and development to production-grade workload execution in quantum, we need to rely on immense, high-speed classical computing infrastructure running alongside it. The classical side must do two things at once: shuttle massive streams of data (**high throughput**), and react and process quantum measurements at extremely fast time scales (**ultra-low latency**).

This challenge is not isolated to quantum computers — it is a universal barrier across **quantum computing**, **quantum sensing**, and **quantum networking**. Without a way to leverage powerful classical compute in real-time with delicate quantum hardware, no quantum technology can scale.

	Core Mechanism	Key Applications
Quantum Computing	Encodes information into quantum states, exploiting entanglement and superposition to perform complex calculations.	Designed to solve specific, intractable problems that classical supercomputers can't tackle in a timely manner. These problems are typically prevalent in industries such as: automotive, energy, pharmaceutical, semiconductor, AI, and finance.
Quantum Networking	Transmits quantum information (typically via entangled photons) within a long-range network. Leverages the "no-cloning theorem" to ensure data security.	Secure communications protected against hacking by quantum mechanics.
Quantum Sensing	Exploits the extreme fragility of quantum states, using them as highly sensitive probes to measure physical quantities.	Applications include enabling GPS-free navigation and timing systems for defense and aerospace, performing high-resolution medical imaging, and mapping underground geological structures for resource extraction.

Comparison of quantum computing, sensing, and networking. All three require ultra-low latency execution across quantum systems and classical hardware (such as FPGAs, GPUs, and CPUs) to drive real-time processing.

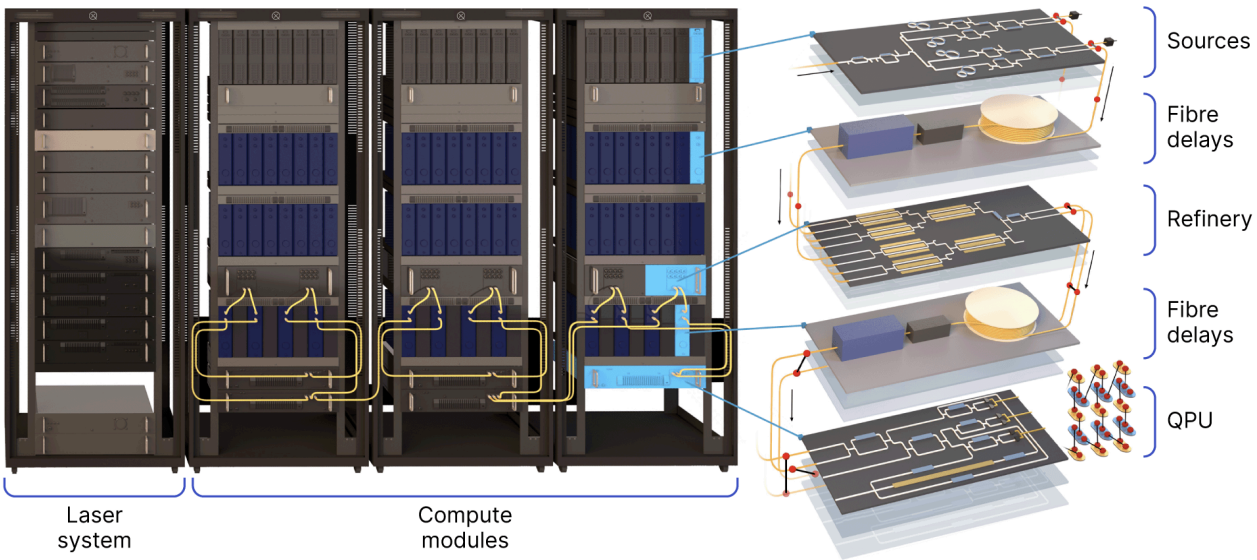
Photonic quantum computing

Different quantum hardware platforms operate on drastically different clock-rates. While superconducting systems operate on the order of microseconds and trapped ions in milliseconds, photonic quantum computing represents the most demanding case of all. Because light moves at light speed, photonic quantum devices run at extremely high clock rates, leaving virtually no margin for computational delays.

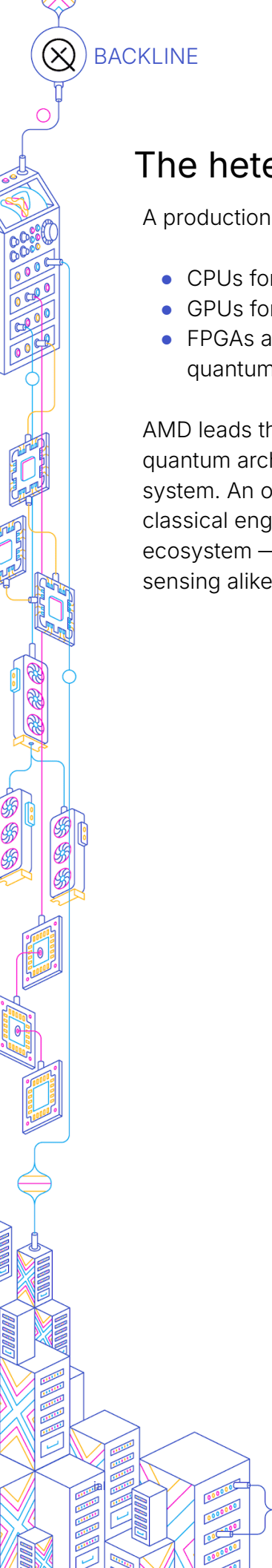
For Xanadu's Aurora photonic quantum system, real-time quantum error correction (QEC) demands an end-to-end feedforward loop — from measuring a physical qubit to adjusting the next operation — that has execution budgets on the order of microseconds.

Because Xanadu's own photonic hardware roadmap represents one of the industry's most extreme timing constraints, we require a software platform capable of orchestrating QEC on ultra-fast FPGAs while simultaneously routing complex, tail-end errors to synchronous co-processors to minimize system stalls.

By solving these intense bottlenecks for quantum architectures with AMD, the broader quantum computing, networking, and sensing ecosystem can leverage this same open platform to overcome their own execution barriers, regardless of the underlying qubit modality.



Overview of Xanadu's quantum photonic hardware.

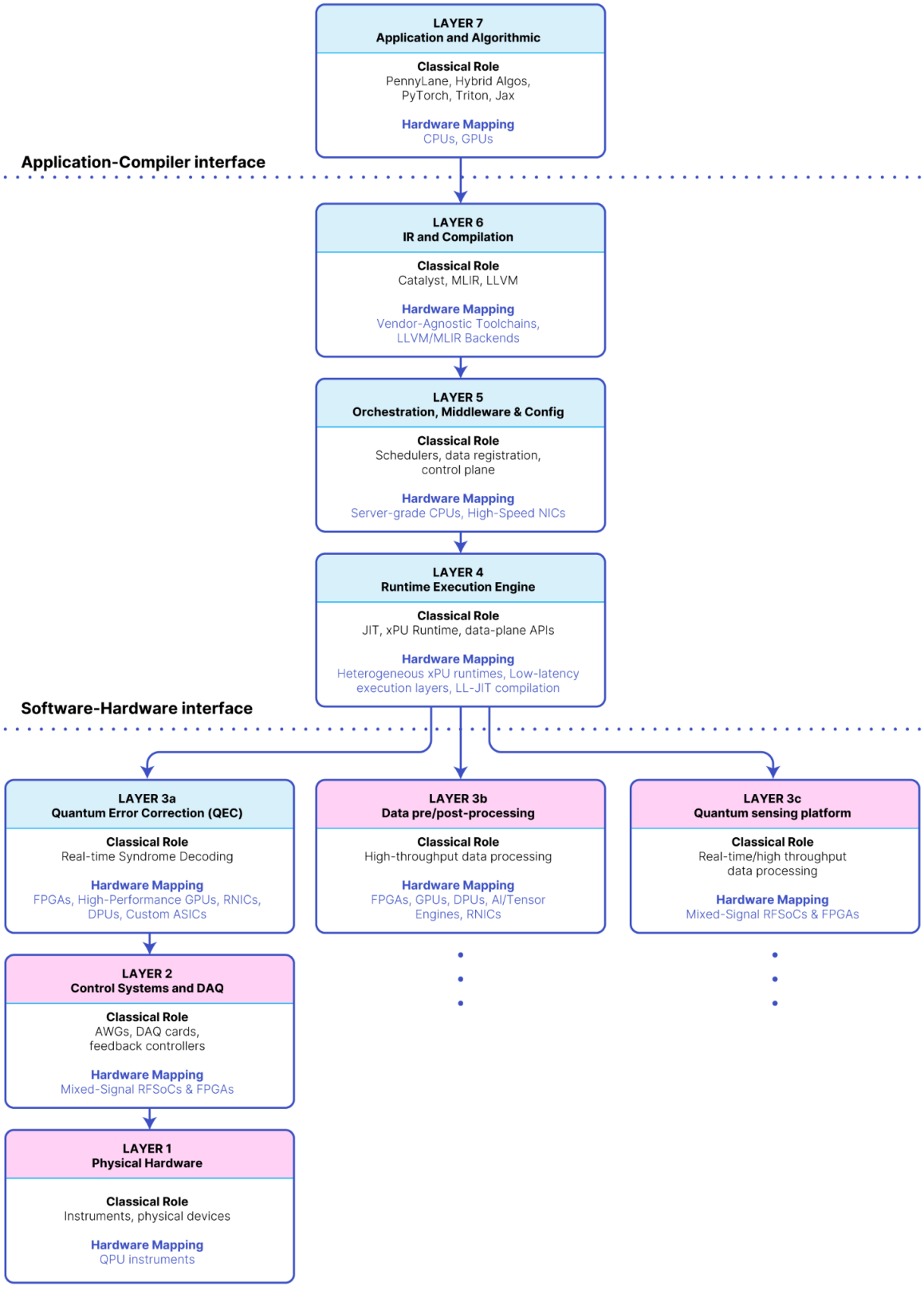


The heterogeneous ecosystem

A production-grade quantum system requires an array of different classical components:

- CPUs for orchestration and high-level control logic.
- GPUs for parallel data crunching and complex decoding.
- FPGAs and ASICs for fixed, ultra-low latency hardware control directly next to the quantum chip.

AMD leads the industry across all three component types, while Xanadu brings the quantum architecture, software, and system-level requirements that define the quantum system. An open platform that dynamically connects quantum workloads to the right classical engine will serve as a comprehensive development kit for the entire quantum ecosystem — empowering innovators across computing, secure networking, and precision sensing alike.



An overview of the different layers in the quantum computing stack, and the various classical hardware and software components that underpin it.

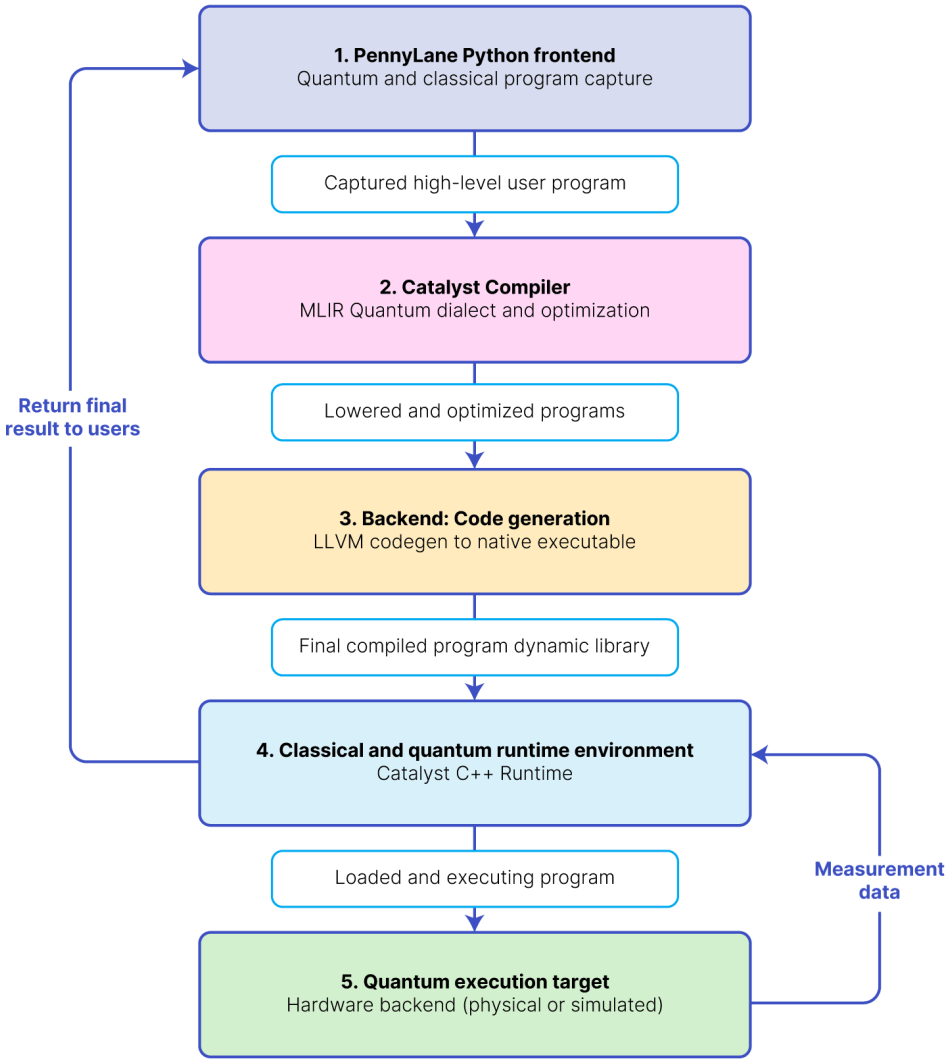
The Python paradox

Python is the universal language of quantum research. Friendly, flexible, and backed by a vast scientific ecosystem, it empowers scientists to seamlessly move from inspiration to fault-tolerant quantum algorithms. The PennyLane software platform is evidence of this, with the world's largest library of research demos and state-of-the-art components to build algorithms in quantum chemistry, quantum information, optimization, and quantum machine learning.

However, Python is an interpreted language and, as a result, is far too slow to run real-time quantum hardware in sub-microsecond windows. To achieve sub-microsecond speed, developers historically had to rewrite everything in low-level code for specialized hardware chips, destroying developer agility, losing context of the broader application, and creating rigid, vendor-locked systems.

Catalyst, PennyLane's quantum compiler, was the first step toward providing the software infrastructure to tackle this challenge and bridge the gap between quantum applications written in Python and the needs of quantum hardware. It utilizes LLVM — a mature classical compilation toolchain — to capture and compile full quantum-classical workflows directly from Python, executing them close to bare metal hardware. However, the programmability of some classical accelerators remains notoriously painful; for example, programming FPGAs can be incredibly time-consuming for large workloads.

We need an environment that maps these complex control systems and hardware components back to a familiar Python interface where high-level quantum and classical code is already defined, while supporting compilation and deployment to tightly coupled accelerators with low-latency requirements.



How quantum and classical workflows flow from the PennyLane Python frontend, through the PennyLane Catalyst compiler, and down to execution — for both classical (4) and quantum code (5).

The myth of universal low-latency: different latencies for different tasks

While instantaneous communication between quantum and classical components is paramount for certain operations — such as quantum error correction (QEC), which must correct physical qubits in near real-time before physical information is lost — a common misconception is that this extreme speed is universally required.

In reality, modern quantum workloads operate across a diverse spectrum of latency regimes and compute requirements. Not every algorithm or processing step maps well to a single, specific device, nor requires the same strict latency budget for communication.

While hardcoding all logic onto tightly-coupled GPUs, FPGAs, or ASICs would effectively mitigate communication delays, it severely limits programmability, sacrifices the heavy processing power of distributed CPUs and GPUs, and is rarely economically viable.

Therefore, a heterogeneous classical ecosystem is essential to provide flexible, programmable computation. We need a heterogeneous ecosystem and matching compilation framework capable of targeting these distinct classical devices, and routing workloads to the correct latency tier, seamlessly.

Latency Tier	Description	Target Hardware
Real-time control (1 ns-1 µs)	Hard real-time tasks with deterministic budgets (e.g., QEC decoders and measurement feedforward)	ASICs, FPGAs.
Synchronous co-processing (1 µs-100 µs)	Soft real-time regime for supportive tasks, like backup QEC decoding.	FPGAs, CPUs, GPUs.
Distributed or co-located runtimes (100 µs+)	Algorithmic compilation and standard data-processing workflows like pre- and post-processing.	General compute resources, distributed remote and local systems, supercomputers.

Quantum error correction (QEC) provides a clear example of a classical workload distributed across multiple latency tiers. The vast majority of standard errors are handled by a primary decoder executing within the real-time control tier. Operating on tightly coupled ASICs or FPGAs, this decoder resolves errors within a strict 1 ns to 1 µs budget to keep pace with the quantum processor's clock rate.

However, when the primary decoder encounters a highly complex or deeply correlated ‘tail-end’ error that it cannot rapidly resolve, it offloads the task to a ‘backup decoder’ in the synchronous co-processing tier. By routing these computationally demanding exceptions to a flexible CPU or GPU, the system ensures accurate decoding while minimizing execution stalls.

Backline: an open platform for the quantum community

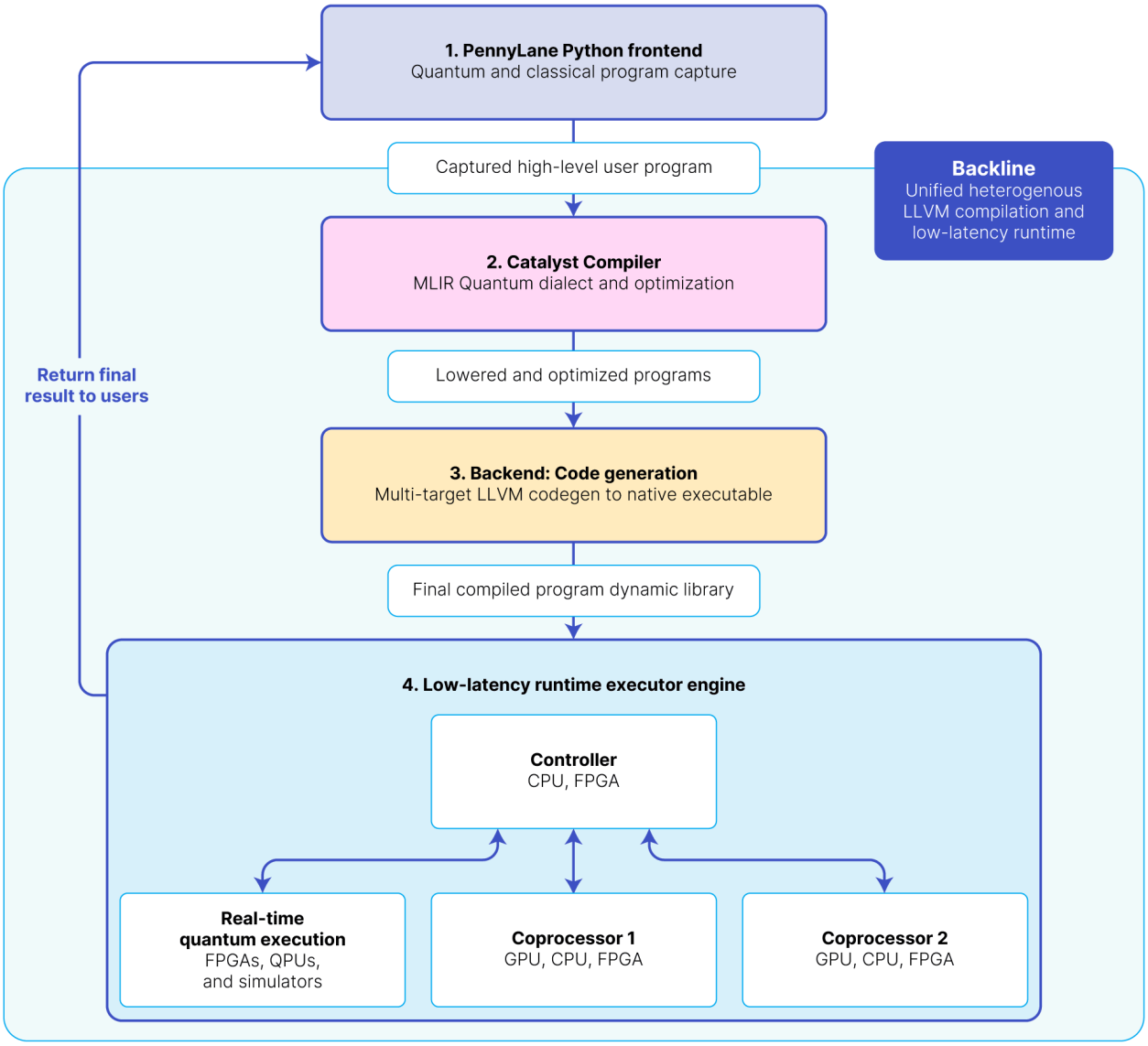
Backline is an open heterogeneous compiler and runtime layer built within the open source PennyLane platform and Catalyst compiler. Designed to directly solve critical bottlenecks for Xanadu's own photonic hardware (characterized by high clock rates and uncompromising real-time requirements), Backline operates as a "one-stop shop" for co-design across applications, QEC, and hardware.

For Xanadu, this means developing an open platform solution that unlocks a step in the development of photonic fault-tolerant computation; for others — this means an openly available platform that is accessible and easily translated for cross-modality use.

With Backline, anyone can write a QEC encoder or decoder from Python, test it with meaningful quantum algorithms, and immediately deploy it for near-real-time prototyping on actual classical and quantum hardware — while supporting the need to drop through abstractions and write increasingly optimized and low-level code.

As the scale and complexity of quantum workloads expand, the need for managed device interactions and distributed environments is becoming central to ecosystem design. Backline is designed to navigate this complexity with several core advantages:

- **Single-digit microsecond latency:** With an under 3-microsecond end-to-end latency looping, Backline can effectively support the tight co-processing required for advanced QEC backup decoding by treating CPUs and GPUs as highly responsive, low-latency execution endpoints.
- **Open source:** We believe in moving fast to unblock ecosystem research. Backline is provided within an open source ecosystem and supported by a rich suite of tutorials and live demos.
- **Python-native:** With Backline, you can write everything from Python. For example, use Python-based libraries like Triton and Gluon to write highly optimized GPU kernels. You can also choose to connect pre-compiled libraries and functions for use with Backline — easily jumping through abstraction layers without switching frameworks.
- **Infrastructure agnostic by design:** Different hardware platforms have drastically different requirements for error correction, utilizing classical accelerators from across the ecosystem. Backline leverages the expansive compilation tools of the LLVM ecosystem, and supports CPUs, GPUs, FPGAs, and custom devices from any hardware vendor.



How quantum and classical workflows flow from the PennyLane Python frontend, through the PennyLane Catalyst compiler, and down to low-latency quantum (and classical) execution with Backline.

Who is Backline for? Accelerating innovation across disciplines

Moving from R&D to production usually means rewriting code from scratch. Backline brings laboratory research and production engineering into the exact same environment. By relying on a shared, Python-native workflow, teams can move faster — ensuring that a theoretical breakthrough in algorithms and architecture translates immediately to scalable hardware execution.

Quantum architecture and algorithm researchers

Skip the framework hopping and rewrites. Write intertwined classical and quantum code entirely in Python, and compile it for low-latency execution across CPUs, GPUs, FPGAs, and QPUs with built-in quantum error correction. Explore how your algorithm breakthroughs — whether applications or error correction — fit with the current requirements and constraints of quantum systems.

Quantum hardware developers

Supercharge your R&D cycles. Backline gives you a single, cohesive environment to prototype control systems, physical quantum hardware, and QEC infrastructure across multiple layers of abstractions. Because it taps directly into the broader PennyLane ecosystem, you can seamlessly connect your hardware to existing state-of-the-art quantum applications and broader workflows — all while hitting the fixed low-latencies your systems demand.

Systems engineers and classical compute vendors

This is the backbone (or some might say, backline) of the quantum revolution. Backline abstracts away the complexities of quantum mechanics, treating quantum processors as another node in a high-performance distributed network. This empowers you to do what you do best: design and orchestrate powerful heterogeneous computing environments. By leveraging the full spectrum of classical hardware — from AMD EPYC™ CPUs and Instinct™ GPUs to ultra-fast AMD adaptive SOCs and FPGAs — you are building the high-throughput, low-latency classical infrastructure that makes scaling quantum computing, sensing, and networking a reality.

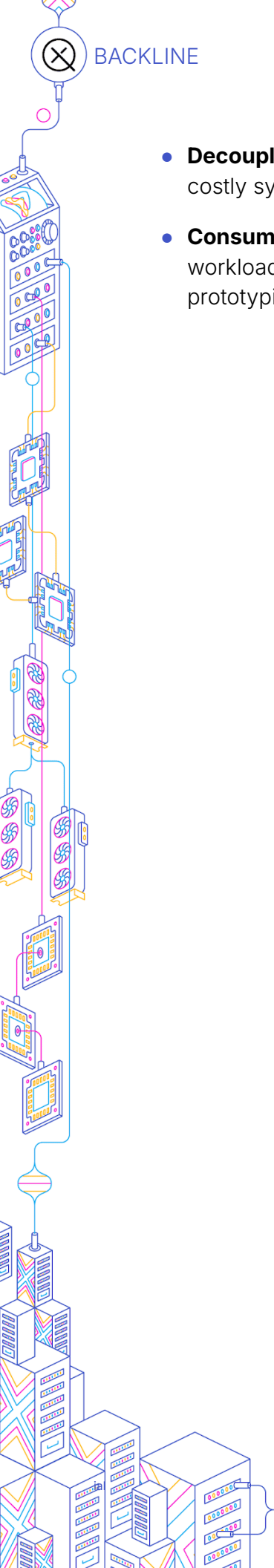
The myth of GPUs everywhere: lowering the barrier to entry

Historically, the quantum industry has operated under the assumption that hyper-expensive, enterprise-grade GPUs are an absolute necessity for QEC. This perception implies that only massive GPU clusters can handle the extreme compute and latency requirements of quantum execution.

While GPUs have an important role to play for certain quantum hardware modalities, this does not have to be a strict requirement. Backline lowers these costly hardware barriers:

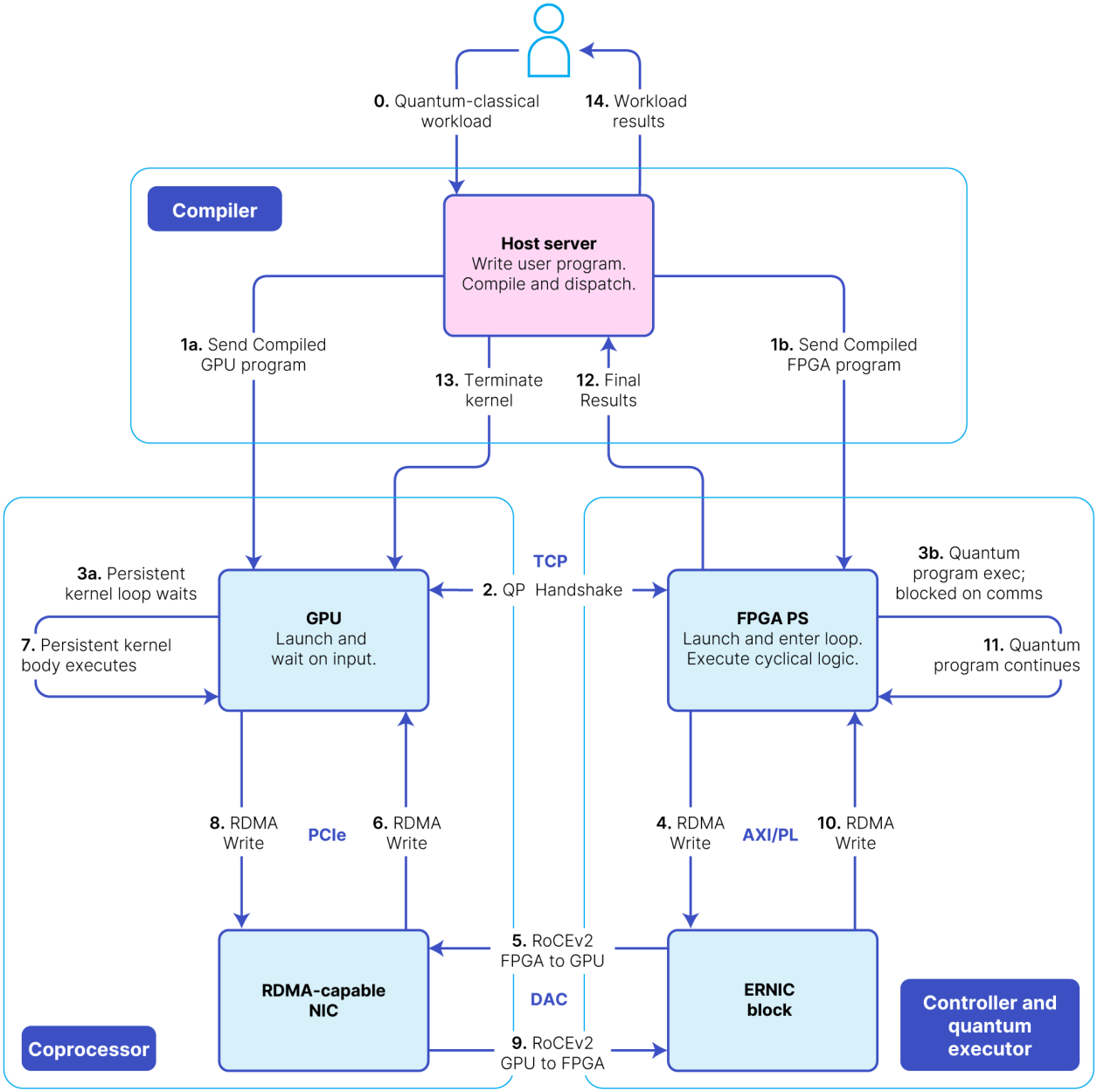
- **CPU-First Efficiency:** Backline leverages standard CPUs to deliver low-latency compute for workloads that do not strictly require heavy parallel GPU processing, with sub-3 microsecond latency. This approach directly bypasses global GPU supply constraints and allows you to start prototyping immediately.

- **Decoupled memory architectures:** The platform does not require specialized, costly systems that mandate shared CPU/GPU memory.
- **Consumer GPU pathway:** Backline opens a viable route to deploy complex workloads directly onto consumer-grade GPUs, freeing researchers from prototyping on massive enterprise servers.



Inside Backline

Backline operates as a dynamic heterogeneous compiler and runtime environment explicitly engineered to sit natively within PennyLane and its Catalyst compiler. At its core, the architecture orchestrates complex quantum-classical workflows through a unified pipeline.



Overview of how quantum and classical workflows flow through the Catalyst compiler (corresponding to boxes (2) and (3) in the previous image) and the low-latency runtime (box (4)) of Backline, which is handling all heterogenous compilation and orchestration.

Applications and performance

As Backline lives inside **PennyLane** and **Catalyst**, it provides a user-friendly Python frontend backed by a C++ codebase and LLVM compilation infrastructure. Developers can write high-level algorithmic logic in Python, but when ultra-fast performance is required, Backline lets developers jump through abstraction layers to write custom, highly optimized low-level hardware kernels — all without switching programming environments.

The following examples show how you can take an application and push it through increasing levels of complexity and optimization, entirely from Python — scaling from local prototyping to low-latency remote hardware execution.

Local CPU-to-CPU interaction with a pre-compiled kernel

First, we'll start with a local CPU-CPU interaction — a great place to begin prototyping, as it comes with significantly fewer specialized hardware barriers. The best thing is, this workflow easily extends to remote workflows on specialized hardware (such as GPUs and FPGAs) with minimal changes.

Firstly, we will register our pre-compiled Steane code QEC decoder via `CoprocessorFunction`, which prepares a function for execution on a Backline coprocessor device. For convenience, we will point to a pre-compiled library that comes with Catalyst, but you can point this to any pre-compiled function.

Python

```python
import os
from pathlib import Path

import pennylane as qp

STEANE_LIB_CPU = str(
    Path(os.environ.get("CATALYST_ROOT", "~/catalyst")).expanduser()
        / "runtime/build/lib"
        / "libsteane_coprocessor_cpu.so"
)

steane_decode = qp.CoprocessorFunction("steane_coprocessor", STEANE_LIB_CPU)
```

Next, we can create our two CPUs: the controller, and the coprocessor. The controller will run quantum instructions on the PennyLane Lightning simulator, while the coprocessor will be performing the decoding.

```
Python
CPU1 = qp.Controller(device=qp.device("lightning.qubit", wires=3))
CPU2 = qp.Coprocessor(coprocessor_fn=steane_decode)
```

And that's it — we can create our backline (using `memcpy` as our transport mechanism[1]), register it to our QNode, and run our workflow. Note that, as we define `qec_code="steane"`, QEC encoding and decoding will be automatically applied; the former during MLIR optimization, and the latter on our coprocessor.

```
Python
dev = qp.Backline(CPU1, [CPU2], transport="memcpy", qec_code="steane")

@qp.qjit(capture=True)
@qp.set_shots(10)
@qp.qnode(dev, mcm_method="one-shot")
def ghz():
    qp.Hadamard(0)
    qp.CNOT([0, 1])
    qp.CNOT([1, 2])
    return qp.sample([qp.measure(0), qp.measure(1), qp.measure(2)])

print("samples:", ghz())
samples: [[0 0 0]
 [1 1 1]
 [1 1 1]
 [0 0 0]
 [1 1 1]
 [1 1 1]
 [1 1 1]
 [0 0 0]
 [0 0 0]
 [0 0 0]]
```

Remote CPU-to-GPU interaction with a Python defined kernel

Next, we'll move to a remote CPU-to-GPU interaction using a Python-defined QEC decoding kernel, written using Triton. This features a native Python QEC encoding of quantum low-density parity codes (qLDPC) via transversal Clifford gadgets, with explicit QEC encoding and decoding handled directly in Python.

[1] You can also configure local CPU-CPU interactions using RDMA.

To begin with, we'll create our server configuration, representing our remote server carrying an AMD GPU and an RDMA capable NIC[2]:

```python
SERVER = {
    'host':"192.168.3.14",
    'user': 'username',  # The SSH account of the user on the remote machine
    'triple': "x86_64-unknown-linux-gnu",
    'sudo': True,
    'deploy': [Path(os.environ["BACKLINE_BUNDLES"]) / "threadripper-bundle"],
    'executor_bin': "numactl -N 0 -m 0 ./catalyst-executor",
    'env': {"LD_LIBRARY_PATH": "."}
}
```

Now, we can define our belief propagation decoder. We do so using two parity-check matrices:

```python
import jax
import numpy as np

N = 13  # data wires
AUX = N  # single reused auxiliary wire

Hx = np.array([[1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0],
               [0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0],
               [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
               [0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0],
               [0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1],
               [0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1]])

Hz = np.array([[1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
               [0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
               [0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0],
               [0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1],
               [0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0],
               [0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1]])

bp_decoder = qp.backline.css_bp_decoder(Hx, Hz, postprocess="osd", num_iters=10,
platform="hip:gfx90a:64")
```

[2] The server configuration will need to be updated as per your server details.

Next, we define a *remote* controller on the server. As in the previous example, the controller is in charge of executing the quantum instructions, the only difference is that now it is on a remote server rather than on the same local machine where the quantum-classical workflow is defined.

Python

```
CPU = qp.Controller(
    name="cpu-controller",
    device=qp.device("lightning.qubit", wires=N + 1),
    remote=True,
    executor_options={**SERVER, "port": 8810},
    init_args={"config": "dev=mlx5_1;gid=3;cpu_pin=4;rt=1"}
)
```

We can now define a remote GPU coprocessor on the server, and specify the Python-defined belief propagation decoding function it will be executing:

Python

```
GPU = qp.Coprocessor(
    name="gpu-coproc",
    coprocessor_fn=bd_decoder,  # our Python-defined BP decoder
    remote=True,
    endpoint=qp.Endpoint("192.168.1.2", 7760),
    executor_options={**SERVER, "port": 8813},
    hardware="gpu",
    init_args={"config": "dev=mlx5_1;gid=3;cpu_pin=4;rt=1;gpu=0"}
)
```

With our controller and coprocessor now defined, we can create our backline object:

Python

```
dev = qp.Backline(controller=CPU, coprocessors=[GPU], transport="rdma")
```

Since we did not specify any automatic quantum error correction, we need to explicitly perform encoding and decoding. Here, we will encode our logical circuit using a qLDPC code:

Python

```
encoder = {
     0: [6, 9, 11],
     1: [7, 9, 10, 11, 12],
     2: [8, 10, 12],
     3: [6, 11],
     4: [7, 11, 12],
     5: [8, 12],
 }

 @qp.qjit(capture=True)
 @qp.set_shots(1)
 @qp.qnode(dev, mcm_method="one-shot")
 def encoded_decoded_circuit(error_kind):
     # ========= Logical circuit =========
     # encode a logical 0 state
     for pivot, targets in encoder.items():
         qp.Hadamard(wires=pivot)

         for t in targets:
             qp.CNOT(wires=[pivot, t])

     # encode a logical X gate
     for w in [6, 7, 8]:
         qp.X(wires=w)

     # encode a logical H gate
     for w in range(N):
         qp.Hadamard(wires=w)
     for a, b in [(1, 3), (2, 6), (5, 7), (10, 11)]:
         qp.SWAP(wires=[a, b])

     # encode a logical Z gate
     for w in [2, 5, 8]:
         qp.Z(wires=w)

     # encode a logical H gate
     for w in range(N):
         qp.Hadamard(wires=w)
     for a, b in [(1, 3), (2, 6), (5, 7), (10, 11)]:
         qp.SWAP(wires=[a, b])

     @qp.for_loop(0, N, 1)
     def correction_rounds(error_qubit):
         # ========= Inject errors =========
         # Injected at compile time
```

```
        add_error(error_qubit, prng_key)

        # ========= Decoding =========
        z_syndrome, x_syndrome = extract_syndromes()

        correction_z = qp.backline.decode(x_syndrome, decoder_id=0)
        correction_x = qp.backline.decode(z_syndrome, decoder_id=1)

        # Apply X to every data qubit whose correction bit is set
        for q in range(N):
            qp.cond(correction_x[q], qp.X)(wires=q)

        # Apply Z to every data qubit whose correction bit is set
        for q in range(N):
            qp.cond(correction_z[q], qp.Z)(wires=q)

    correction_rounds()

    return (qp.expval(mean_stabilizer(Hz, qp.Z)), qp.expval(mean_stabilizer(Hx,
qp.X)))
```

Note the use of `qp.backline.decode` — this special function allows us to explicitly call our coprocessor function during execution[3].

Finally, we define the functions that inserts random depolarizing error on the physical qubits, as well as a function for extracting the QEC syndromes:

Python

```
def add_error(error_qubit, error_kind):
    """Apply I/X/Y/Z to one chosen data qubit."""
    qp.cond(error_kind == 1, qp.X)(wires=error_qubit)
    qp.cond(error_kind == 2, qp.Y)(wires=error_qubit)
    qp.cond(error_kind == 3, qp.Z)(wires=error_qubit)

def extract_syndromes():
    """Measure stabilizers and return syndrome bit lists."""
    z_syndrome = []
    for row in Hz:
        for q in np.flatnonzero(row):
```

[3] `qp.backline.decode` makes a series of runtime calls directly on the coprocessor. The ability to manually perform runtime calls is also available via `qp.runtime_call`.

```
            qp.CNOT(wires=[int(q), AUX])
        z_syndrome.append(qp.measure(AUX, reset=True))

    x_syndrome = []
    for row in Hx:
        qp.Hadamard(wires=AUX)
        for q in np.flatnonzero(row):
            qp.CNOT(wires=[AUX, int(q)])
        qp.Hadamard(wires=AUX)
        x_syndrome.append(qp.measure(AUX, reset=True))

    return z_syndrome, x_syndrome

def mean_stabilizer(checks, pauli):
    return qp.dot(
        [1 / len(checks)] * len(checks),
        [qp.prod(*(pauli(wires=int(q)) for q in np.flatnonzero(row))) for row in
checks],
    )
```

With all the pieces in place, we can run our explicit QEC encoding and decoding example on the backline:

Python

```
>>> for error_kind, error_name in enumerate(["I", "X", "Y", "Z"]):
...     print(error_name, encoded_decoded_circuit(error_kind))
# scp: Connection closed
# [scp] the SFTP backend is declining. Retrying it with the legacy protocol (scp
-O), which hosts without an SFTP subsystem need
# [cpu-controller] catalyst-executor: loaded
/home/catalyst-exec/librt_transport.so
# [cpu-controller] catalyst-executor: loaded /home/catalyst-exec/librt_capi.so
# [cpu-controller] catalyst-executor: loaded
/home/catalyst-exec/liblightning_qubit_catalyst.so
# [cpu-controller] Listening on 127.0.0.1:8810
# [cpu-controller] [127.0.0.1:8810] executor ready, waiting for connections
# scp: Connection closed
# [scp] the SFTP backend is declining. Retrying it with the legacy protocol (scp
-O), which hosts without an SFTP subsystem need
# [gpu-coproc] catalyst-executor: loaded /home/catalyst-exec/librt_transport.so
# [gpu-coproc] catalyst-executor: loaded /home/catalyst-exec/librt_capi.so
```

```
# [gpu-coproc] catalyst-executor: loaded
/home/catalyst-exec/librdma_triton_decoder.so
# [gpu-coproc] Listening on 127.0.0.1:8813
# [gpu-coproc] [127.0.0.1:8813] executor ready, waiting for connections
# [gpu-coproc] [127.0.0.1:8813] accepted connection from 127.0.0.1:56120 on pid
1546394, waiting for next connection
# [cpu-controller] [127.0.0.1:8810] accepted connection from 127.0.0.1:56154 on
pid 1546399, waiting for next connection
# I (Array(1., dtype=float64), Array(1., dtype=float64))
# X (Array(1., dtype=float64), Array(1., dtype=float64))
# Y (Array(1., dtype=float64), Array(1., dtype=float64))
# Z (Array(1., dtype=float64), Array(1., dtype=float64))
# JIT session error: FD-transport disconnected
# JIT session error: disconnecting
# JIT session error: FD-transport disconnected
# JIT session error: disconnecting
```

Remote GPU-to-FPGA interaction with a pre-compiled kernel

We scale the exact same logic to a remote FPGA-to-GPU environment with a pre-compiled kernel, and this time using implicit quantum error correction. Rather than a physical circuit (with the error encoding and decoding being defined in Python as part of the workflow), we instead pass a logical circuit, and utilize Catalyst MLIR optimization passes for QEC encoding using the Steane code. The decoder will then be applied automatically before returning the results.

First, we create the FPGA controller. Here, it is a AMD Versal™ Premium Series VPK120 board:

Python

```python
FPGA_SERVER = {
    'host': "192.168.3.15",
    'user': "username",  # user on the FPGA server
    'triple': "aarch64-unknown-linux-gnu",
    'sudo': True,
    'deploy': [Path(os.environ["BACKLINE_BUNDLES"]) / "vpk-bundle"],
    'env': {
        "XMM_SQ_TYPE": "PL", "XMM_RQ_TYPE": "PL", "XMM_CQ_TYPE": "PL",
        "XMM_APP_MAX_QP": "4", "XMM_APP_RQ_SGE": "1", "XMM_SQ_DEPTH": "64",
        "XMM_RQ_DEPTH": "128", "HWHS_RTT_WARMUP": "0"
```

```
    }
}

FPGA = qp.Controller(
    name="fpga-controller",
    remote=True,
    hardware="fpga",
    executor_options={**FPGA_SERVER, "port": 8811},
    init_args={"config":
"dev=xib_0;gid=1;sq_mem=bram;data_mem=bram;reply_mem=bram;reply_poll=hw;stride_log2
=6"}
)
```

We can use the same remote GPU coprocessor we set up earlier, but replace `coprocessor_fn` with `'catalyst_gpu_steane_launcher'` (an alias for Catalyst's precompiled Steane decoder). Defining the backline, we provide our FPGA as the controller:

Python

```
dev = qp.Backline(controller=FPGA, coprocessors=[GPU], transport="rdma",
                  qec_code="steane")

@qp.qjit(capture=True)
@qp.set_shots(1000)
@qp.qnode(dev, mcm_method="one-shot")
def ghz():
    qp.Hadamard(0)
    qp.CNOT([0, 1])
    qp.CNOT([1, 2])
    return qp.sample([qp.measure(0), qp.measure(1), qp.measure(2)])
```

We can now execute the function to see the results:

Python

```
>>> sample(ghz())
a-controller] catalyst-executor: loaded
/home/petalinux/catalyst-exec/librt_transport.so
[fpga-controller] catalyst-executor: loaded
/home/petalinux/catalyst-exec/librt_capi.so
```

```
[fpga-controller] catalyst-executor: loaded
/home/petalinux/catalyst-exec/librtd_null_qubit.so
[fpga-controller] Listening on 127.0.0.1:7811
[fpga-controller] [127.0.0.1:7811] executor ready, waiting for connections
scp: Connection closed
[gpu-coproc] catalyst-executor: loaded /home/catalyst-exec/librt_transport.so
[gpu-coproc] catalyst-executor: loaded /home/catalyst-exec/librt_capi.so
[gpu-coproc] Listening on 127.0.0.1:7813
[gpu-coproc] [127.0.0.1:7813#3133915] Accepted connection
[fpga-controller] [127.0.0.1:7811] accepted connection from 127.0.0.1:39924 on pid
1209, waiting for next connection
[fpga-controller]
[fpga-controller]=== engine RTT (n=16000, 0 warmup dropped, hardware handshake) ===
[fpga-controller]   min          4600 ns
[fpga-controller]   p50          4855 ns
[fpga-controller]   p95          5095 ns
[fpga-controller]   p99          5570 ns
[fpga-controller]   p99.9        5815 ns
[fpga-controller]   max          9660 ns
[fpga-controller]   mean         4931 ns
samples: [[0 0 0]
 [0 0 0]
 [0 0 0]
 ...
 [0 0 0]
 [0 0 0]
 [0 0 0]]
JIT session error: FD-transport disconnected
JIT session error: disconnecting
JIT session error: FD-transport disconnected
JIT session error: disconnecting
```

Bring your own Triton decoder

With a simple modification to the above remote GPU-FPGA example, we can replace the pre-compiled Steane decoder with a custom Triton decoder — all written from Python. Here, we use Triton to create a decoder for the Steane code. The Steane code uses just seven qubits to encode a logical (error-corrected) qubit — because this number is so low, we can pre-calculate every solution and store them in a lookup table:

Python

```
# The Steane decoding table: the qubit to flip for each three-bit
```

```
# syndrome, with -1 meaning the
# syndrome was zero and nothing needs correcting.
STEANE_QUBIT_BY_SYNDROME = (-1, 0, 4, 1, 6, 3, 5, 2)

def pack_lookup_table(values, no_error=0xF):
    """Pack the lookup table into one integer, four bits per entry."""
    word = 0
    for i, value in enumerate(values):
        word |= (no_error if value < 0 else value) << (4 * i)
    return word

# A compile-time constant, so the lookup below becomes a
# shift and a mask with no memory access.
STEANE_LUT = tl.constexpr(_pack_lookup_table(STEANE_QUBIT_BY_SYNDROME))

def steane_lookup(syndrome):
    """Return the qubit to correct for one syndrome,
    or -1 if there is nothing to do."""
    idx = tl.cast(0, tl.uint32)
    for i in tl.static_range(3):
        idx |= tl.cast((syndrome >> (8 * i)) & 1, tl.uint32) << i
    qubit = (tl.cast(STEANE_LUT, tl.uint32) >> (idx * 4)) & 0xF

    # All ones is -1 read as unsigned, which is how the
    # controller recognises "no correction".
    no_error = tl.cast(0xFFFFFFFFFFFFFFFF, tl.uint64)
    return tl.where(qubit == 0xF, no_error, tl.cast(qubit, tl.uint64))
```

We can use the provided `qp.backline.triton_decoder` to compile this for our target system, and then it is simply a matter of providing `steane_triton_decoder` as our coprocessing function when defining the GPU coprocessor.

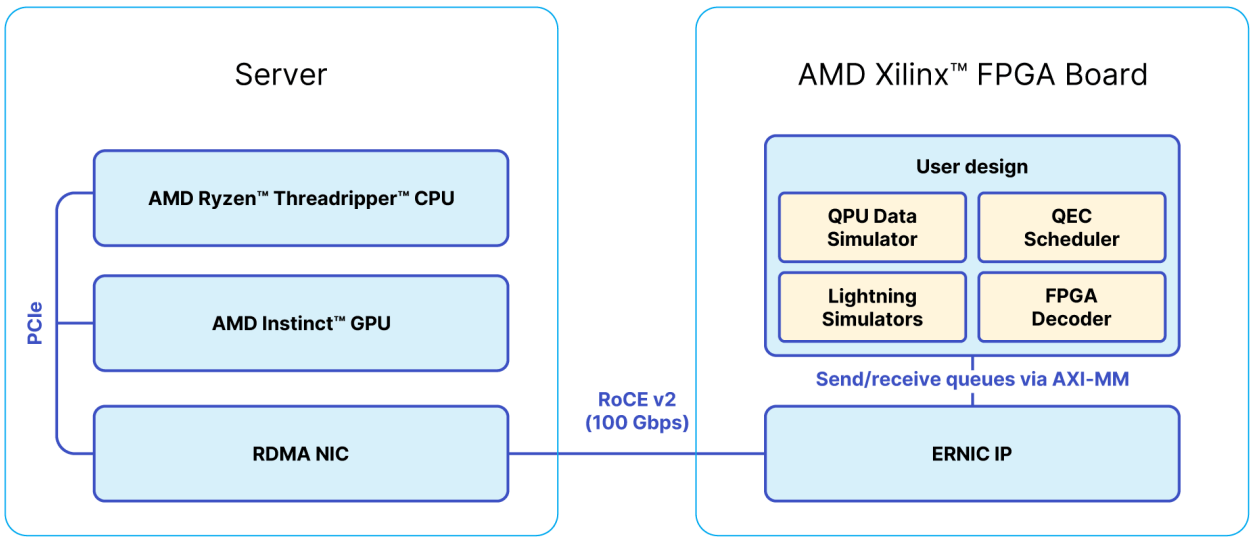
Python

```
steane_triton_decoder = qp.backline.triton_decoder(
    (steane_lookup, steane_lookup),
    platform="hip:gfx90a:64",
)
```

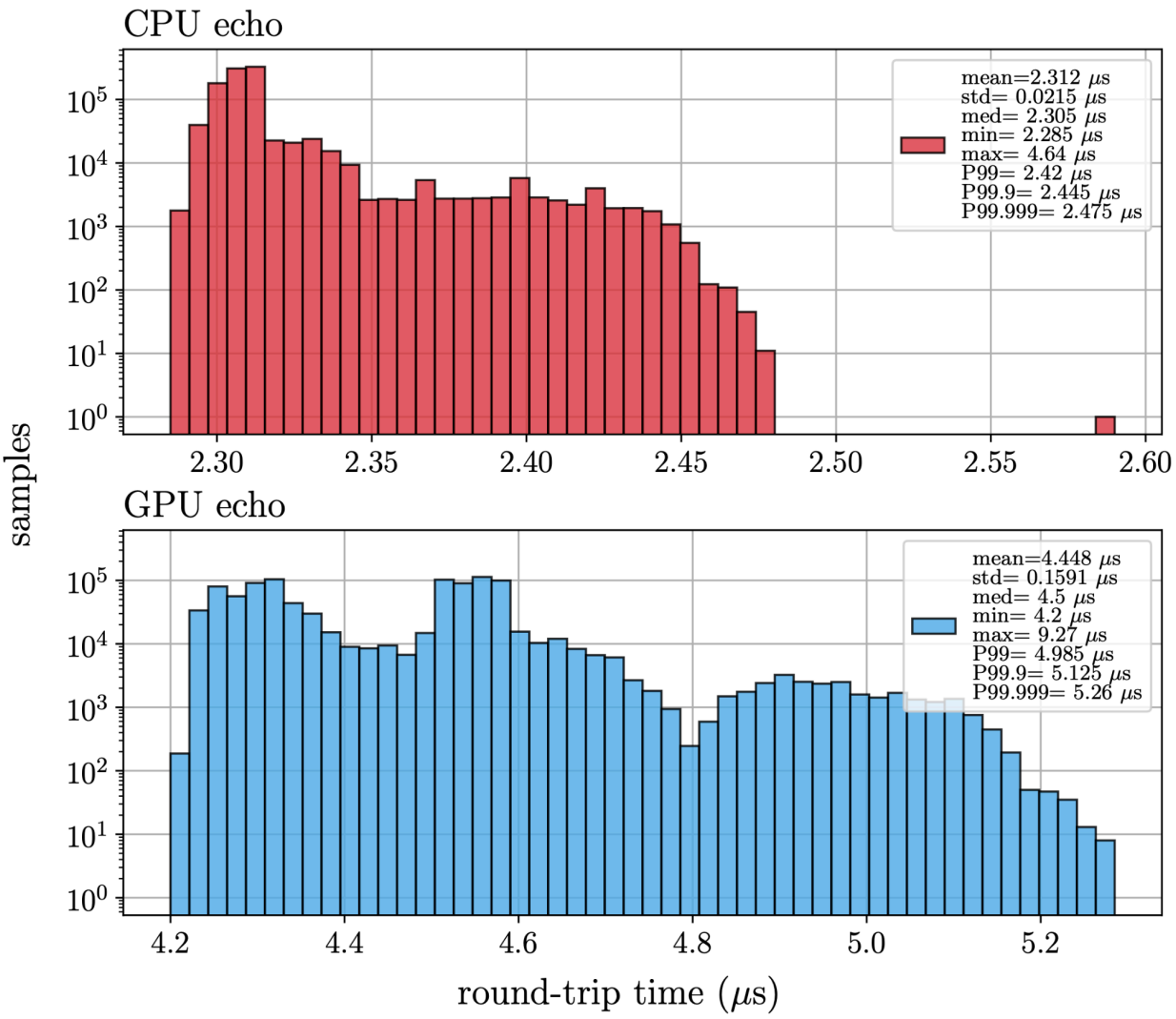
Pushing the Limits

In addition to the application demonstrations above, we also perform benchmarking to demonstrate the performance of Backline.

An FPGA orchestrated workload using ERNIC and an RDMA NIC was set up, submitting a 128-bit payload to a target coprocessor and back. This experiment was repeated for both a GPU (AMD Instinct™ M210) and CPU (AMD Ryzen™ Threadripper™ Processor) coprocessor, demonstrating a mean value of 2.3 µs end-to-end latency for the CPU path, and 4.4 µs end-to-end latency for the GPU path — well within the latency requirements for quantum systems.



A schematic of the benchmarking experimental setup.



Performance benchmarking results from an FPGA ERNIC to workstation and back over ≈ 10^6 samples. The FPGA uses the hardware handshake to lower the latency floor in the receiving end. Warm-up data is executed for the first round to ensure the hardware is stable before gathering data.

Expanding the platform

As the quantum ecosystem transitions from isolated noisy intermediate-scale quantum (NISQ) devices to networked, fault-tolerant systems, classical infrastructure must scale. Our immediate roadmap for Backline focuses on supporting Xanadu's photonic hardware, and expanding the open platform's reach across the industry:

- **Powering photonic milestones**: Backline remains the critical software engine driving our photonic quantum architecture. By orchestrating real-time feedback loops at photonic clock rates, it provides the classical infrastructure needed to reach our fault-tolerant scaling milestones.
- **Progressive programming abstractions**: To maximize usability, introduce a NumPy-like API for co-processors. Developers can start with high-level prototyping and progressively drop down to highly optimized, bare-metal code all within a single environment.
- **Explicit memory management**: For advanced workloads needing fine-grained control, we will explore explicit memory placement capabilities. Powered by AMD's soon-to-be open-source `space-rt` runtime (a spatial runtime for modern accelerators), this feature allows users to manually map data and tasks to specific accelerators natively in Python.
- **Broader device agnosticism**: Continuing to build out support for custom devices and emerging classical accelerators from any classical or quantum hardware vendor, fully leveraging the expansive LLVM ecosystem. In addition, adding support for FPGA coprocessors, and compiling QEC decoders written in Python for FPGAs.
- **Enhanced QEC prototyping**: Providing quantum hardware developers with even more robust tools to prototype QEC infrastructure and control mechanisms natively within Python, drastically reducing R&D turnaround times.
- **Community Integration**: Expanding our suite of live demos and tutorials to accelerate ecosystem-wide algorithms and error correction research and research.

Conclusion

No quantum technology exists in isolation today. Whether the goal is engineering high-performance quantum supercomputers, developing ultra-precise diagnostic sensors, or establishing unhackable secure networks, a singular reality remains: a quantum computer is never purely quantum. Success across the entire ecosystem depends on high-speed classical infrastructure operating in lockstep behind the scenes.

Backline serves as the critical connective tissue for this landscape. By bridging the accessibility of Python with the performance of heterogeneous classical hardware, Backline eliminates the execution bottlenecks that have historically constrained quantum development.

Integrated within the open-source PennyLane platform, and supporting a broad swath of AMD hardware, Backline provides a universal platform for the three core pillars of the quantum industry:

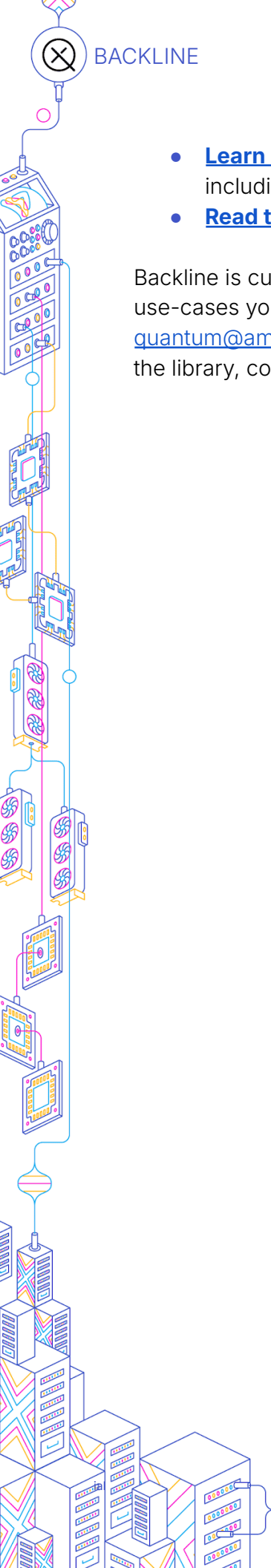
- **Quantum Computing**: Enables hardware teams to prototype real-time QEC systems, providing fragile processors with the ultra-low latency feedback required for complex algorithms without the friction of low-level manual coding.
- **Quantum Networking**: Facilitates control systems that react at nanosecond scales to optical signals, ensuring the reliable routing of entangled information across distributed quantum nodes.
- **Quantum Sensing**: Leverages high-throughput classical compute to instantaneously convert delicate physical measurements into actionable data for geological, medical, and defense applications.

By transforming complex hardware orchestration into a streamlined, Python-first workflow, it ensures that every innovator — from research labs to enterprise engineering teams — can design, test, and deploy production-grade quantum technologies.

Getting started

Backline is available today to help you build and deploy to both quantum and classical hardware.

- **Documentation:** Install Backline, and get started with the `qp.Backline` functionality in PennyLane today.
- **Tutorials:** Explore our tutorials demonstrating single-digit microsecond latency execution across heterogeneous hardware.
- **AMD Developer Cloud:** Start projects with PennyLane on AMD Instinct™ GPUs



- **Learn more in our technical paper:** Delve into the technical details behind Backline, including the experimental setups and benchmarks discussed in this blog post.
- **Read the AMD blog post:** Learn more about Backline on the AMD Technical Blog.

Backline is currently under heavy development — if you have suggestions on the API or use-cases you'd like covered, please open a GitHub issue, or reach out to quantum@amd.com and backline@xanadu.ai. We'd love to hear about how you're using the library, collaborate on development, or integrate additional devices and frontends.