COE 530
Quantum Computers and Architecture

Lecture 16

Quantum Software Stack

References:

Burgholzer et al. "The Munich Quantum Software Stack: Connecting End Users, Integrating Diverse Quantum Technologies, Accelerating HPC."

Giortamis, et al. "{QOS}: Quantum operating system." In *19th USENIX (OSDI 25)*

Quantum Software Stacks

- Goal:
  - Bridging the gap between quantum algorithms and physical quantum hardware
  - Putting everything we learned into a full software stack including: architecture, compilation, and integration of quantum resources with High-Performance Computing (HPC)
- Definition
  - Critical software infrastructure that translates high-level problem descriptions into low-level, hardware-specific instructions for the quantum computer
- Core Purpose
  - Abstracts diverse quantum hardware back-ends (superconducting circuits, trapped ions) and integrates QPUs into classical HPC infrastructures for hybrid workflows
- Analogy to Classical Accelerators
  - QPUs act as specialized accelerators
  - A QSS requires control interfaces, JIT compilation, and resource scheduling

The need for a QSS

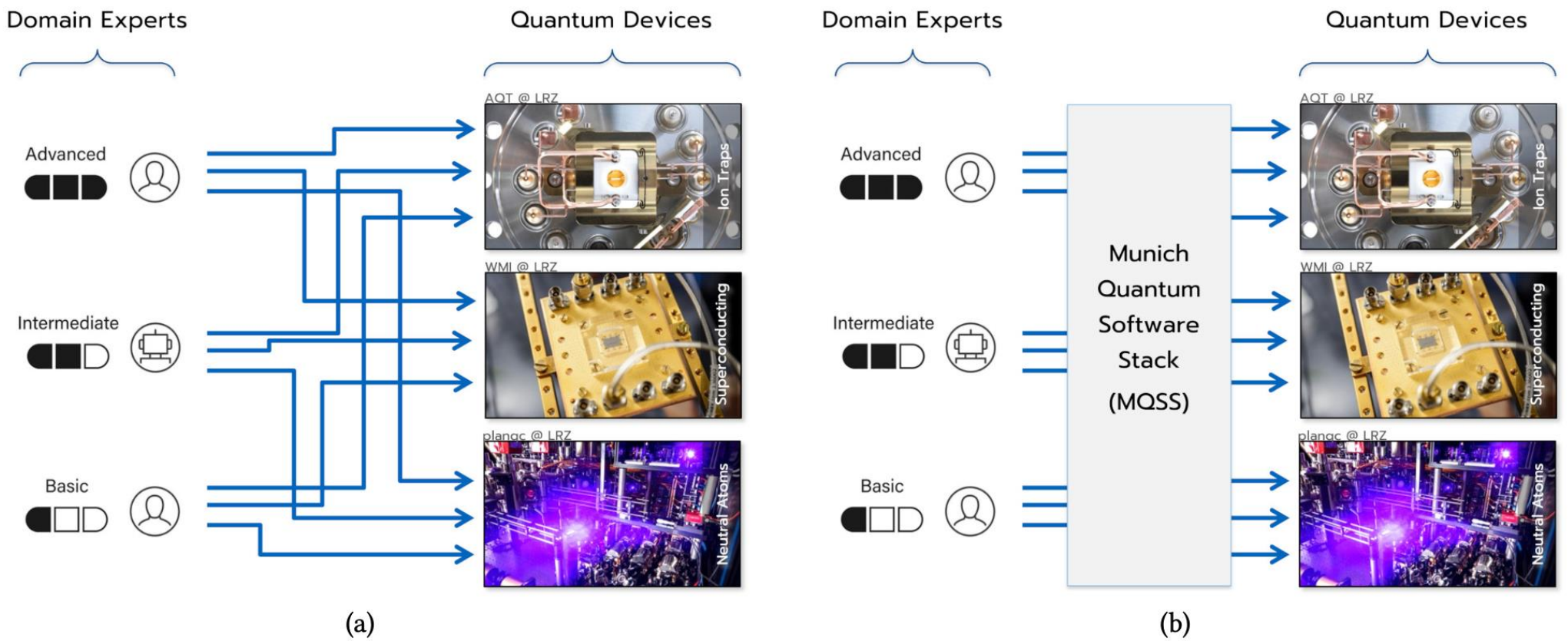


Figure 1: a) Traditional view: many users, many systems, many stacks. b) Vision going forward: decoupling front-end and back-end in a single stack via shared IRs.

General Architecture of a MQSS

Front-end Layer

- User-facing programming interfaces and high-level abstractions; adapters for Qiskit, PennyLane, and C/C++ HPC interfaces

Middle-end (Toolchain & Orchestration)

- Compiles circuits into intermediate representations (IR); device-agnostic optimizations like gate reduction; device-specific transpilation like qubit routing; dynamic resource scheduling

Back-end Layer

- Unified hardware abstraction (e.g., QDMI, QPM) standardizing job submission, pulse generation, and hardware querying across diverse device types

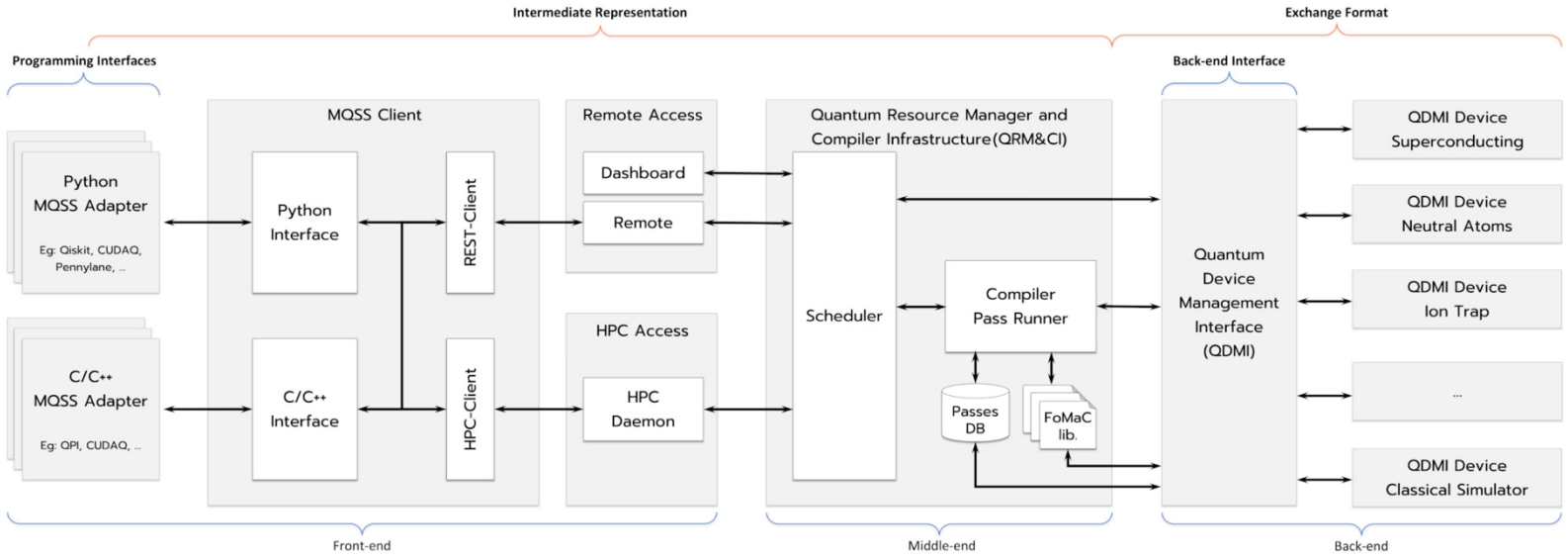


Figure 7: Low-level overview of the compilation and execution pipeline of the MQSS.

MQSS Core Desisgn Principles

- **Hardware heterogeneity:** modular architecture designed for extensibility and flexibility, allowing for the seamless integration of new components and support for a growing number of quantum platforms
- **Integration with HPC:** enabling the tightly-coupled hybrid workflows that are essential for quantum acceleration
- **Open interface:** to foster a collaborative ecosystem and overcome the lack of standards using the Quantum Device Management Interface

QDMI Concept

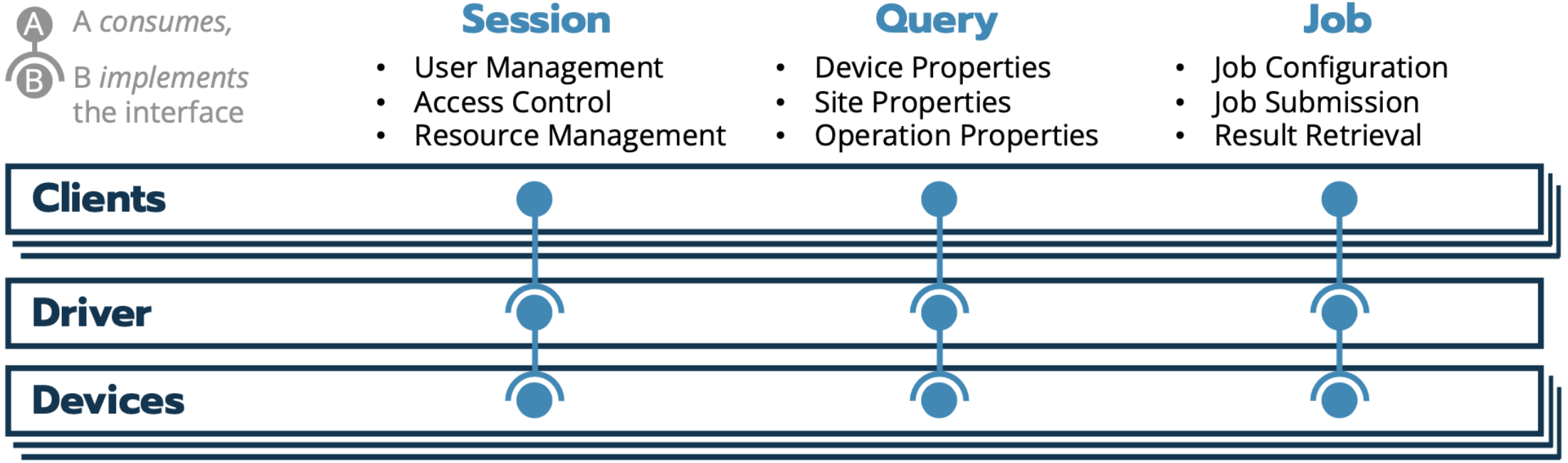


Figure 4: The three main components of the QDMI: Session Management, Job Submission, and Query Interface.

Trageted Groups

- **Domain Experts:**
  - Scientists and researchers from material science, chemistry, operational research, or finance
  - MQSS provides high-level abstractions that allow them to formulate problems without needing to manage the complexity of quantum hardware
- **HPC Experts:**
  - Users who are accustomed to heterogeneous computing environments and wish to use QPU as another accelerator
  - MQSS facilitates this by integrating QPUs as offload devices as loosely-coupled or highly-coupled
- **Quantum Algorithm and Software Developers:**
  - Developers want to design, test, and benchmark new algorithms, compilation strategies, and software tools
- **Hardware Providers and Physicists:**
  - Experts who wdevelop and operate quantum computing hardware
  - MQSS offers (quasi-)standardized interfaces, like QDMI, for integrating new devices into the ecosystem
  - Enables low-level access through the same infrastructure to facilitate low-level experiments and manual access
- **System Administrators and User Managers:**
  - Admin responsible for operating QC or HPCQC systems
  - MQSS offers introspection interfaces to understand system health and status, as well as interfaces to integrate issues, like authentication and quota management, with the matching HPC components.

Comparing Existing Quantum Software Stacks

Vendor-Specific (IBM Qiskit, Rigetti Quil)

- Feature-rich but tightly coupled to one vendor's hardware
- Qiskit's naive lexicographic qubit mapping yields lower success rates than noise-aware approaches

Hardware-Agnostic Research (TriQ)

- Multi-platform toolflow targeting IBM, Rigetti, and UMD trapped-ion systems
- Up to 28x success-rate gain over Qiskit and 2.3x over Quil via noise-aware mapping and communication optimization

HPC-Integrated (Munich QSS / MQSS)

- Open, modular, community-driven
- Uses MLIR for hybrid classical-quantum compilation and QDMI for vendor-agnostic interfaces

QOS: A Quantum Operating System — What and Why

- What is QOS?
  - A cloud operating system for managing quantum resources in a quantum-as-a-service setting
  - Exposes a hardware-agnostic API for transparent quantum job execution
- The Problem
  - QPUs are noisy, small, and heterogeneous; managing them through rudimentary cloud interfaces creates fundamental tensions between fidelity, utilization, waiting time, and load balance
- Why We Need It
  - Demand for QPUs vastly exceeds supply, yet existing tools optimize one metric at a time
  - QOS systematically balances these tradeoffs across many users and devices

Technical Foundations

- Qubits and Quantum Circuits
  - Qubits exploit superposition and entanglement; programs are quantum circuits of gates applied across qubits, then measured to produce probabilistic bitstrings over many shots
- NISQ Devices
  - Today's QPUs are Noisy Intermediate-Scale Quantum: small qubit counts, high gate error rates, and limited coherence times — error correction is not yet feasible
- Fidelity as the Quality Metric
  - Fidelity measures how close the QPU's output distribution is to the ideal one; it drops sharply with circuit size (98.9% reduction from 4 to 24 qubits in the paper's measurements)
- QAOA as a Driving Workload
  - Variational algorithms like the Quantum Approximate Optimization Algorithm (QAOA) for problems such as max-cut are the practical NISQ workload that motivates QOS
- Quantum-as-a-Service
  - Cloud providers (IBM, AWS, Azure, Google) expose QPUs as remote services, creating an OS-style resource management problem across many users and heterogeneous devices

Key Ideas: Heterogeneity, Utilization, Load Balancing

- A. Heterogeneity
  - QPUs differ across vendors, models, and even calibrations of the same device — the paper measures a 38% fidelity gap between best and worst IBM QPU on a 12-qubit GHZ circuit, plus large day-to-day temporal variance
- → QOS Response
  - An analytical Estimator predicts per-circuit fidelity on every available QPU without expensive simulation, letting the scheduler pick the device most likely to give a good answer right now
- B. Utilization
  - A single quantum job typically uses only a fraction of a QPU's qubits, leaving the rest idle; naive scheduling wastes scarce hardware
- → QOS Response
  - The Multi-programmer spatially multiplexes compatible jobs onto the same QPU, quantifying compatibility so co-location raises utilization without large fidelity loss
- C. Load Balancing
  - Heavily loaded QPUs accumulate long queues while others sit idle — and naive load balancing pushes jobs to lower-fidelity devices
- → QOS Response
  - A fidelity-aware multi-objective Scheduler trades a small (1–3%) fidelity hit for up to 5× shorter waits and balanced QPU load

QOS Architecture: Components

- QOS API
  - Hardware-agnostic interface -> transforms user circuits into Qernels an hides QPU-specific complexity
  - run(circ, cnfgs), results(jID), status(jID), backends(), backend_props(bID);
- Qernel Abstraction
  - Common execution unit shared across all QOS components
  - Static properties (qubit count, depth, gate types, Supermarq features)
  - Dynamic state (status, fidelity estimate, results)
- Error Mitigator
  - Pre- and post-processing pipeline that composes complementary error-mitigation techniques
- Estimator
  - Predicts fidelity of a Qernel on each candidate QPU using lightweight analytical models
- Multi-programmer
  - Bundles compatible Qernels for spatial co-execution on the same QPU
  - Fffective utilization vs. fidelity loss to pack jobs without disrupting each other
- Scheduler
  - Fidelity-aware, multi-objective scheduler over heterogeneous QPUs
  - Trades minimal fidelity for large reductions in waiting time and balanced load across devices

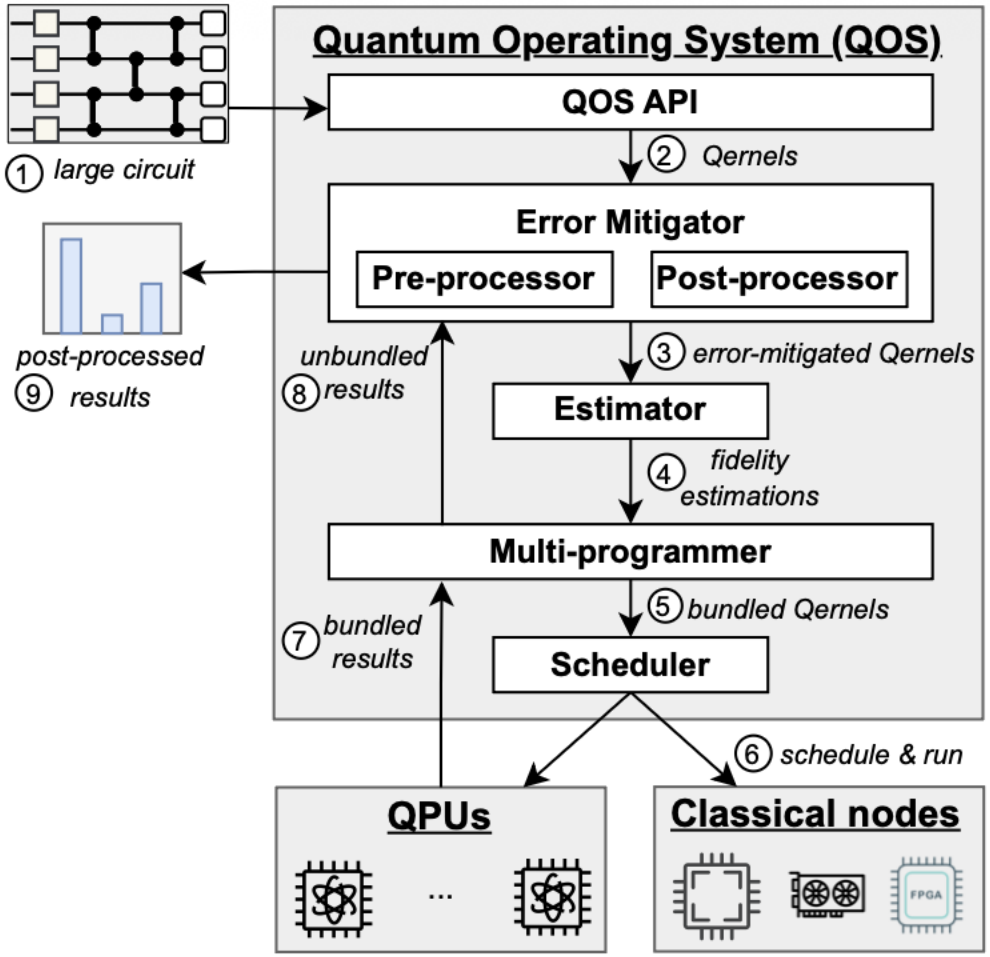


Figure 5: QOS overview (§ 4): *QOS consists of four main components: the error mitigator, estimator, multi-programmer, and scheduler.*

The call for an in-house QSS

- Do we need one? Or should we adopt an existing vendor or open-source stack?
- What features should be there?
- Who should maintain it?

Discussion: Should We Build Our Own In-House QSS?

- Strategic Control
  - If we depend on Qiskit, Quil, or another vendor stack, what happens to our research roadmap when their priorities — or pricing — diverge from ours? Do we control how our hybrid HPC-QC workflows evolve?
- Hardware Optimality
  - TriQ showed up to 28× fidelity gains over vendor compilers by exploiting hardware details. What success rates are we leaving on the table by accepting one-size-fits-all vendor compilation for our specific QPU(s)?
- HPC Integration
  - Our local supercomputer scheduler, storage, and security model are unique. Can a vendor stack ever integrate as deeply with SLURM, our filesystems, and our identity stack as a stack we build ourselves — like ORNL's QFw or LRZ's MQSS?
- Vendor Lock-in vs. Heterogeneity
  - If we host (or plan to host) multiple QPU types — superconducting, trapped ion, neutral atom — will a vendor-specific stack ever serve all of them well, or do we need a QDMI-style abstraction we own?
- Talent and IP
  - Building a local QSS develops in-house compiler, systems, and quantum expertise. What is that capability worth over a 5–10 year horizon, compared to perpetual reliance on external roadmaps?
- Cost of Ownership
  - What is the realistic build, maintain, and staffing cost of a local stack — and how does it compare against vendor licensing, lost research velocity, and the opportunity cost of suboptimal compilation?
- Community vs. Custom
  - Could we contribute to and extend an open community stack (e.g., MQSS, QFw) instead of building from scratch — gaining most of the control with a fraction of the engineering cost?