ORNL Quantum Software Stack

Amir Shehata, Peter Groszkowski, Thomas Naughton, Murali Gopalakrishnan Meena, Elaine Wong, Daniel Claudino, Rafael Ferreira da Silva, Thomas Beck

shehataa@ornl.gov

https://arxiv.org/abs/2503.01787

Motivation for HPC/QC Integration

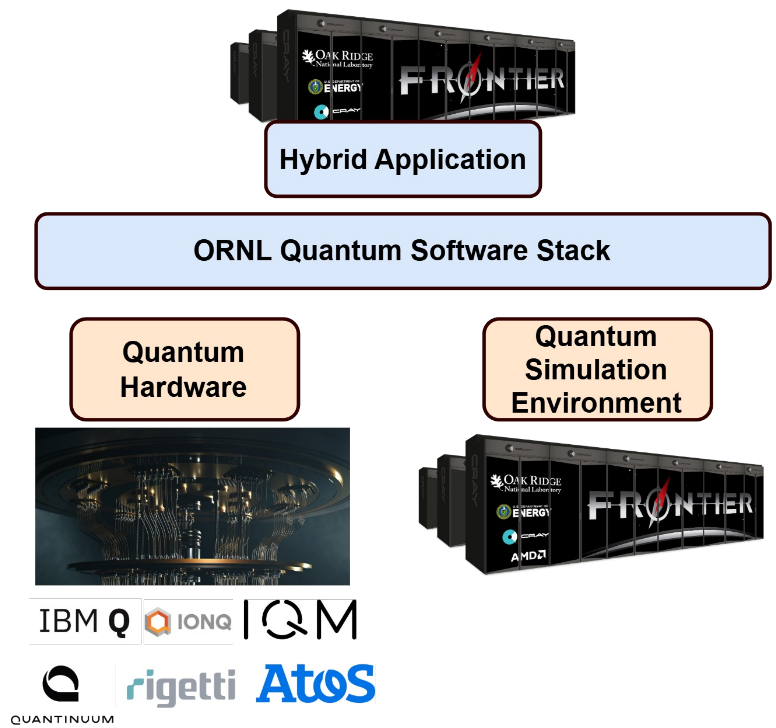
- Quantum Computing (QC) could **solve specific problems** exponentially **faster** than classical computers
  - But the technology is still in its infancy
  - Costly, hardware limitations, environment limitations, errors, etc.

- In the foreseeable future,
  - QC will **coexist** and collaborate with classical High-Performance Computing (**HPC**) environments
  - Use **QPUs** as accelerators like we use GPUs

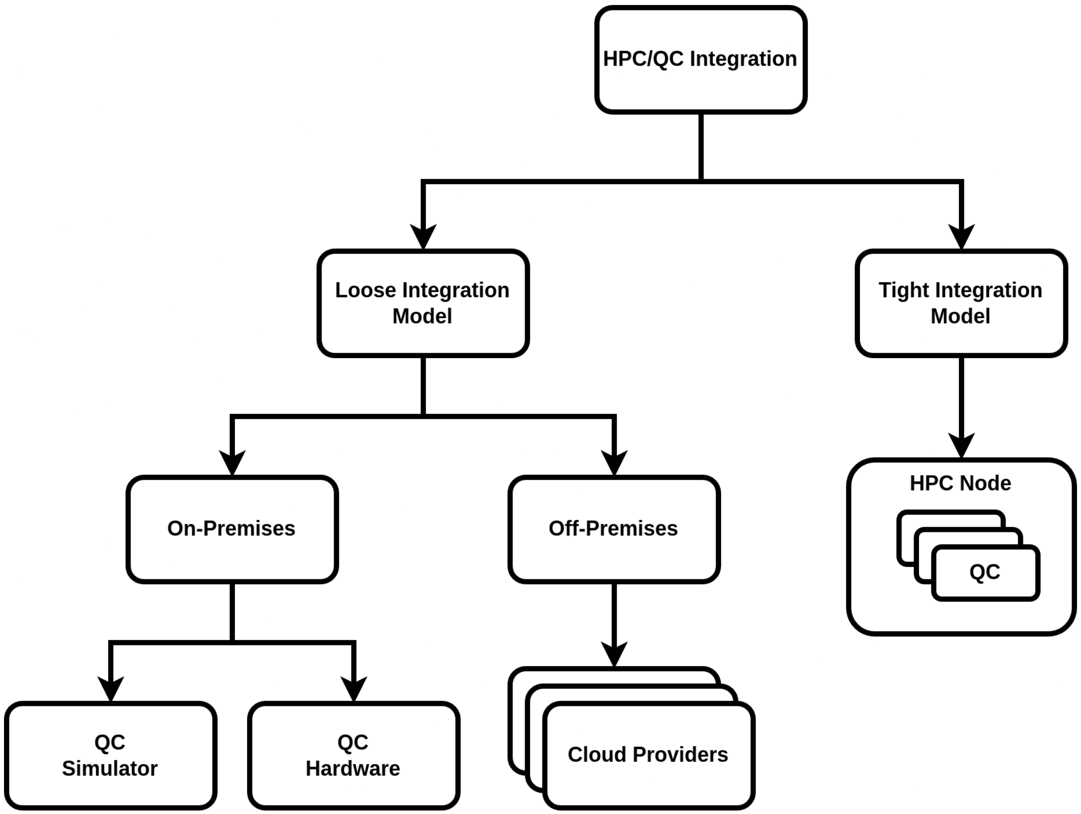
- **Early involvement is** crucial for understanding the challenges of integration

Integration Goals

- Support **Hybrid Application** workflows
  - Integrate with **quantum hardware** and **quantum simulators**
  - **Resource management** integration
- Quantum Software Stack
  - **Programming environment** flexibility
  - **Hardware/simulator** flexibility
  - **Standardize** quantum platform access
  - Enable **tool integration** (e.g. circuit cutting, gate reduction)
  - **Optimize** HPC and QC resource usage

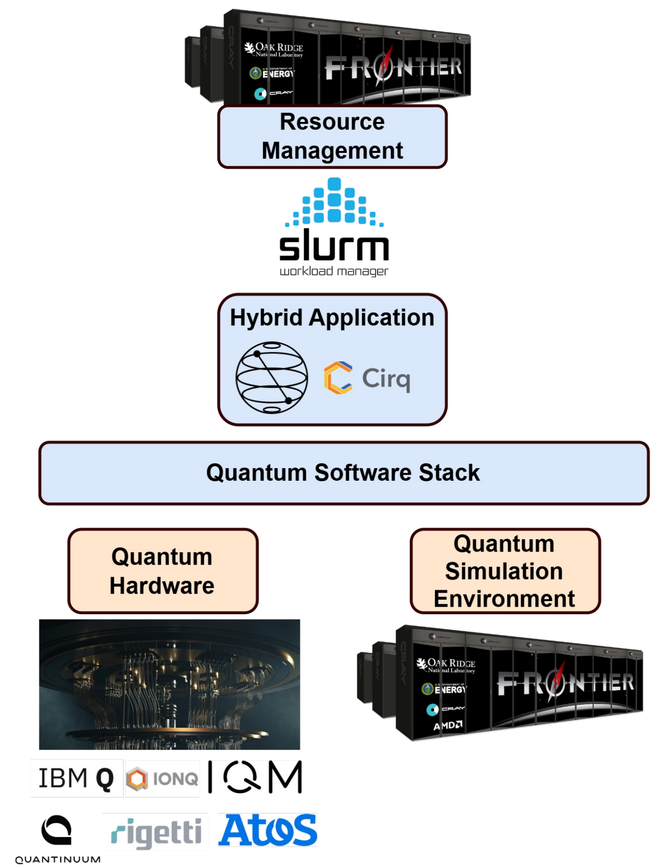


Integration Space



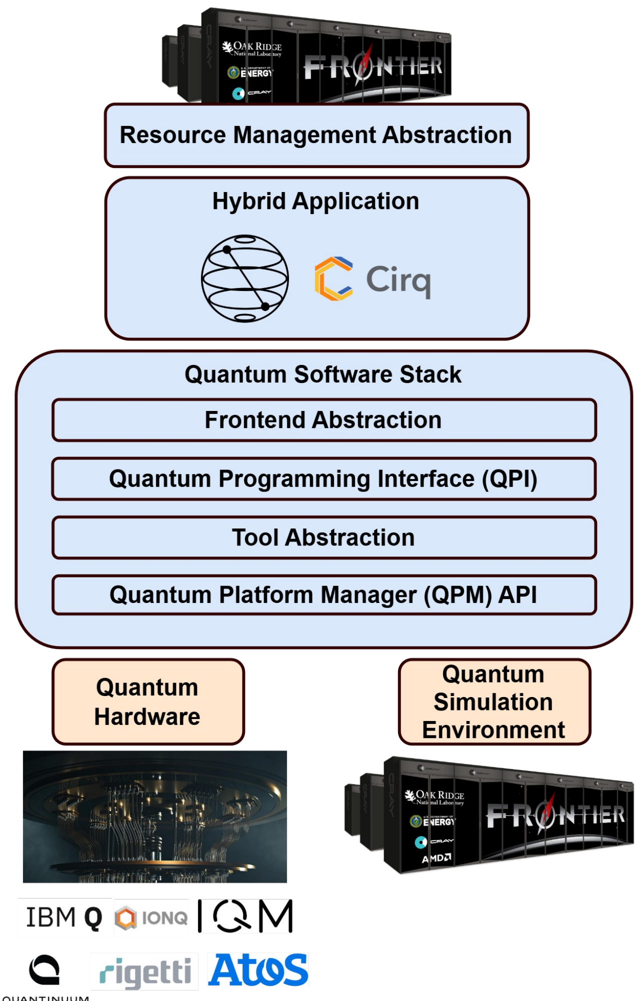
Challenges

- HPC/QC **diverse usage**
- Diverse **front and backend usage**
- Resource management
  - **Allocation** of HPC and QC resources
  - Job **Scheduling**
  - **Placement** of tasks on resources
  - **Coordinated** HPC/QC task scheduling
- **Efficient** use of HPC resources for classical quantum simulators

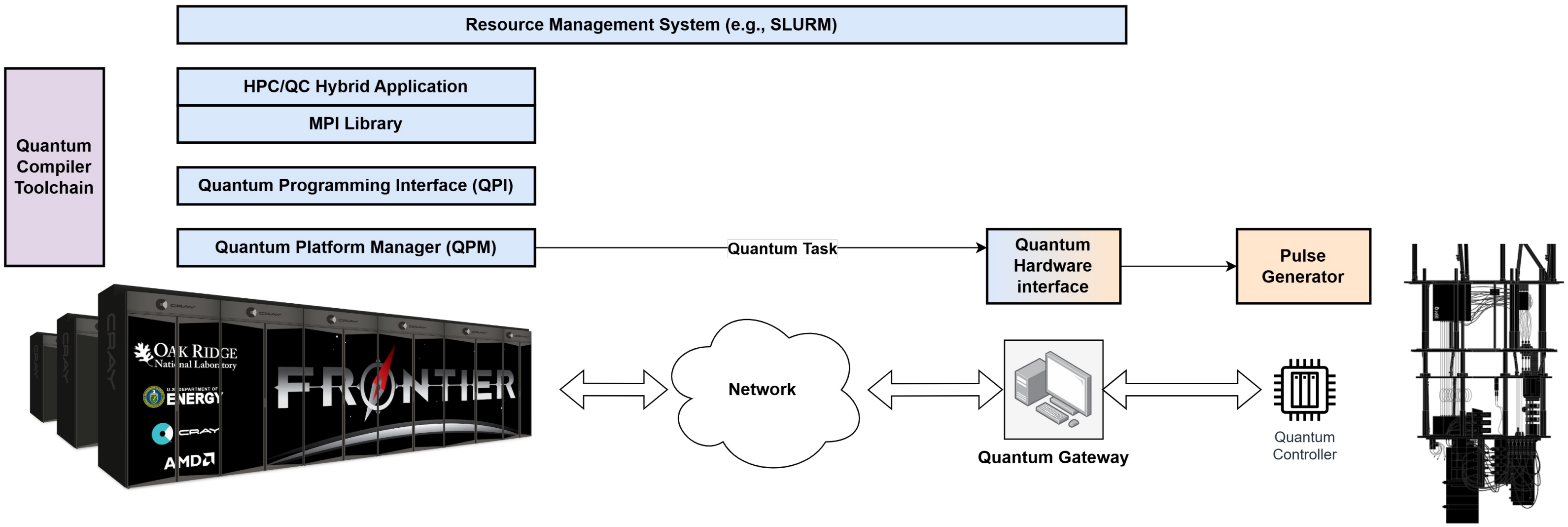


Quantum Software Stack

- To **effectively** address challenges, need to develop the **correct** levels of **software abstractions** in the software stack
- **Levels of abstractions**:
  - Resource management & task placement
  - Frontend
  - Quantum Programming Interface (QPI)
  - Tool Interface
  - Quantum Platform Manager (QPM)



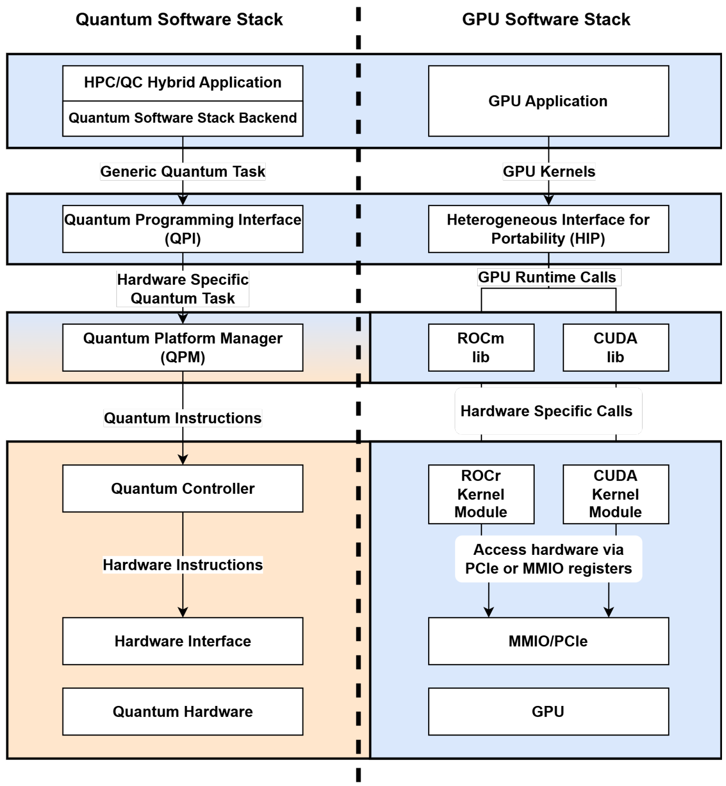
Overview



- From right to left, the quantum machine is connected to a **quantum controller**
  - The quantum controller **generates**, **manipulates**, and **reads** out signals that control the quantum computation.
- A "**quantum gateway**" is directly connected to the quantum controller
  - The quantum gateway runs **resource management** processes and other **low latency software**
- The **HPC system** runs the **hybrid application** and the bulk of the **QC/HPC software stack**

GPU Lessons

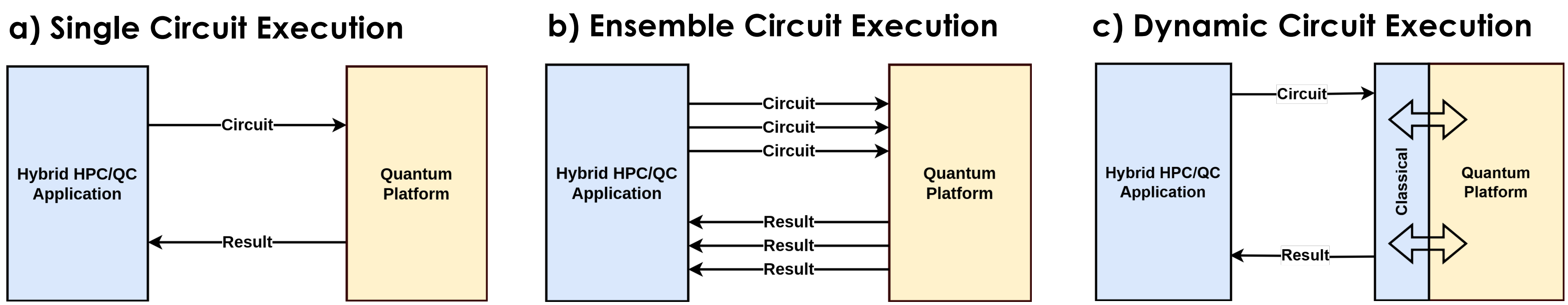
- Like GPUs Quantum machines will act as **accelerators**
- Lessons learned from **GPU/CPU integration**:
  - The QC stack will expose a set of **application facing APIs** like HIP APIs
  - The QC **compilation process** will be similar to the GPU one, including the **Just-In-Time** compilation step from an intermediate representation to the target architecture
  - Quantum circuits will need to be **scheduled and executed** on the target platform raising similar challenges to host/GPU allocation and coordination



OAK RIDGE National Laboratory

Quantum Circuit Execution Patterns

- Hybrid applications are **evolving**
- Focus on general **circuit execution patterns**



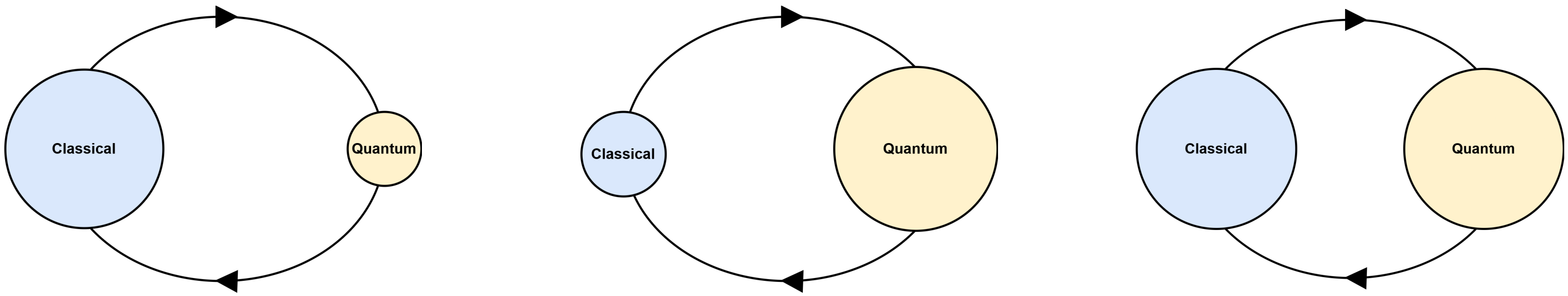
- In pattern (c) classical logic error corrects circuit results in **realtime** via **mid circuit measurements**

OAK RIDGE National Laboratory

Hybrid Applications Usage Patterns

- **Classify** application patterns based on **computational demands**
- Quantum computing deployment will be **limited**, making classification **essential** for **optimal resource management**.

a) High classical/Low Quantum **b) Low classical/High quantum** **c) Roughly equal**



- A **scheduler** which **understands these patterns** helps **minimize** resource **idle time**

Quantum Time Scales

- The **Length of time a quantum** program **(circuit) executes** for
  - Can a classical program wait for quantum program completions?
- Possible scenarios
  - **Low latency Dependency**: Error correction
  - **Manageable latency Dependency**: Classical can wait
  - **No Dependency**: Pre-processing/Post-processing

Resource Management

- Resource management strategy should **balance** two views
  - **Platform View**
    - Maximize the **utilization** of the resource
  - **Application View**
    - Minimize **time to solution**

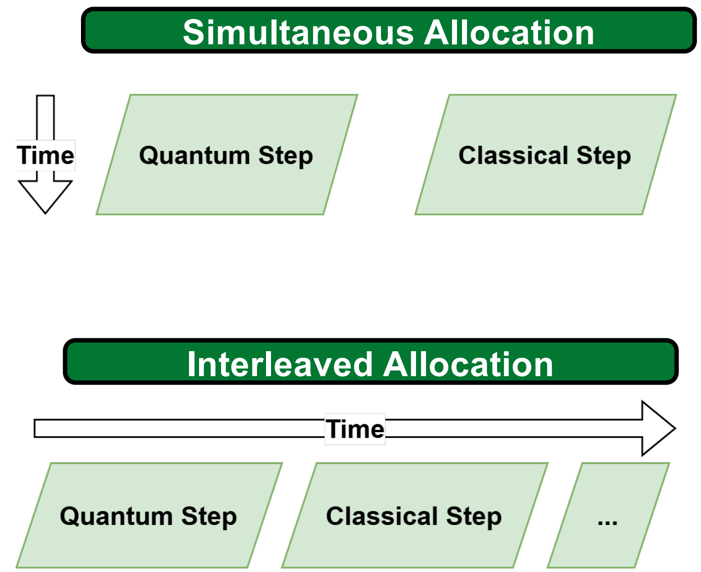
- **HPC strategy** prefers **application view**, where compute resources are dedicated for the **run-time** of the hybrid job
  - If QC requirements are **low**, QC remains **idle** and the same for HPC

- Need to consider **two allocation strategies** to examine the problem; **simultaneous** and **interleaved** allocation

Resource Allocation

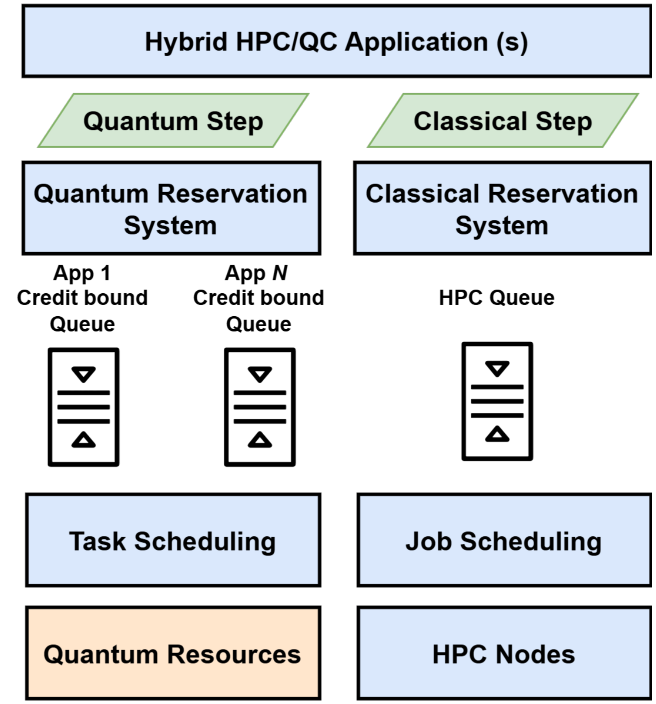
- Allocation of computational assets is driven by application usage patterns
  - **Simultaneous Allocation**
    - QC and HPC resources are allocated concurrently
  - **Interleaved Allocation**
    - QC and HPC resources can be allocated independently and in an interleaved manner

- Proposed software stack needs to work with both allocation strategies



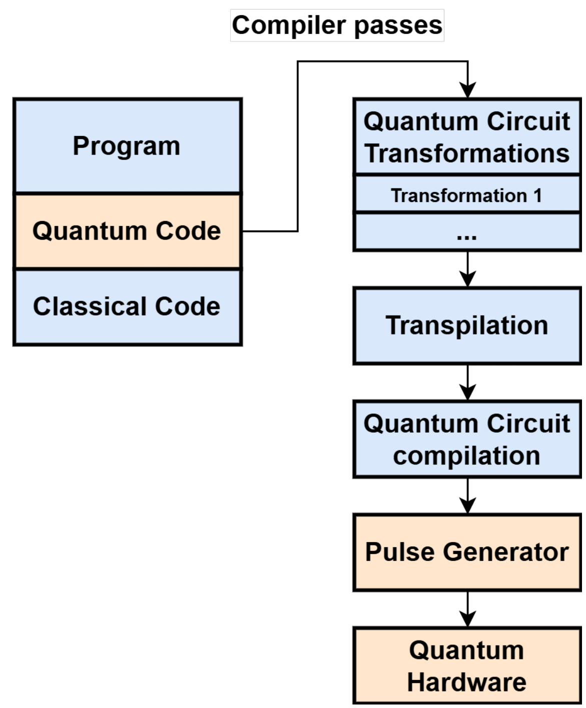
Two-Level Scheduling

- **Goal**: Allow **multiple jobs** to use a **single QC** resource at the same time
- **First-level Scheduling**
  - SLURM or similar can be used to get an initial allocation
    - Allocation granted only if QC resource can handle **job load**
  - SLURM's **Heterogeneous Job** feature can be used to specify **both HPC and QC** resources
    - Supports simultaneous allocation
- **Second-level Scheduling**
  - Circuits from **different jobs** are scheduled to ensure **timely execution**
  - Research **task scheduling** strategies such as a **credit-based** system to **manage** circuits from **multiple jobs**



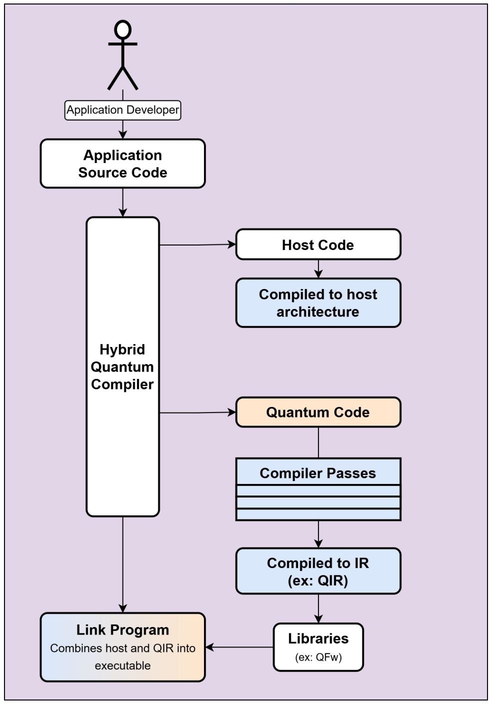
Hybrid HPC/QC Application Preparation

- Hybrid Applications consist of classical and quantum code
  - **Classical code** follows **standard handling** procedures
  - **Quantum code** undergoes several **compiler passes** before reaching the hardware
- Interpreted applications
  - All the QC **compiler passes** and transpilation occurs at **run-time**
- Compiled application
  - Some QC **compiler passes** can be done at **compile time** while others at **run-time**
- Highlights the need for a **unified interface** to the QC compilation passes



Hybrid HPC/QC Application Compilation Process

- Both compiled and interpreted applications follow the same logical steps
  - **Separation** of **host** and **quantum** code
    - Host can be CPU and GPU
    - These are handled in the traditional manner
  - Quantum code passes through a set of **tools/compiler passes**
  - Quantum code is **lowered** to an **intermediate representation (IR)**
  - Host and quantum **code** is **linked** against required libraries and **packaged** in the same **binary**
  - When binary is executed the quantum code can be **JIT compiled** down to hardware format

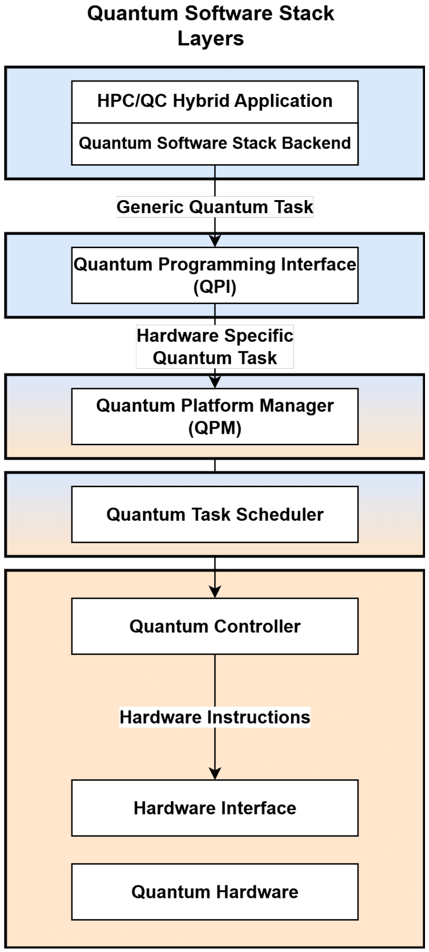


So far …

- **Identified** integration **goals** and **challenges**
- High-level **overview** of the proposed **software stack**
- Overview of **circuit execution** and **hybrid application patterns**
- The need for **efficient** HPC/QC **resource management** and possible strategies
- Hybrid HPC/QC application **preparation** and **compilation** passes
- Questions?

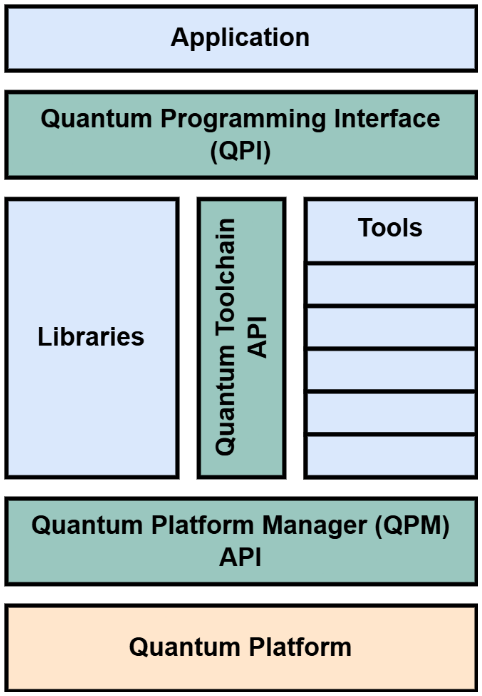
Architecture: Software Layers

- Breakdown software stack into layers
  - **Hybrid HPC/QC Application Layer**
    - Applications use existing circuit building frameworks such as Qiskit to formulate their quantum tasks
  - **Quantum Software Stack Backend**
    - Circuit building framework backend which interfaces with the software stack
  - **Quantum Programming Interface**
    - Provides APIs for version and execution control, error handling, etc
  - **Quantum Platform Manager API**
    - Hardware agnostic API implemented by the hardware provider
  - **Quantum Task Scheduler**
    - Responsible for efficient use of the QC resource
  - **Quantum Controller**
    - Generates pulses towards the quantum hardware



Architecture: Normalization

- **Standardized** interfaces in software system architecture is **important** to ensure, among other benefits, **interoperability**, **scalability** and **reduced vendor lock-in**
- Three interfaces in need of normalization:
  - Quantum Programming Interface (QPI)
    - **Application facing** interface
    - Congruent to HIP
    - Provides **APIs** such as **version checks**, **execution control**, **error** and **event handling** and hardware queries
  - Quantum Platform Manager (QPM)
    - **Platform facing** interface
    - **Abstracts hardware features** for seamless, platform-independent access.
  - Quantum Toolchain API
    - Enables new circuit transformation tool integration into the software stack



Architecture: Quantum Platform Manager API

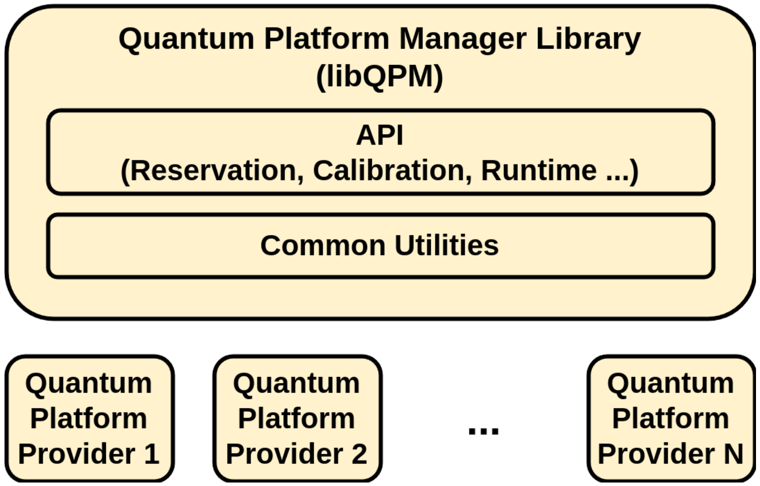
A. Handle **all aspects** of the quantum platform

B. Provides **soft standardization** to access quantum platform

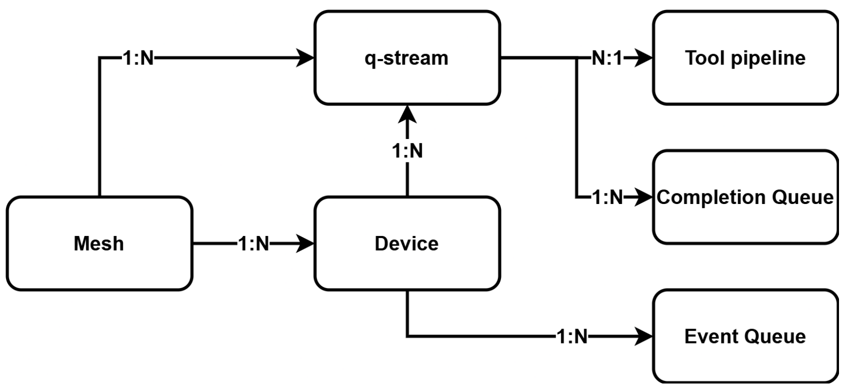
C. Packaged as a library which provides a set of common features

D. Hardware providers are **responsible** for implementing their **specific plugins**

E. Provides a **hardware-friendly** API



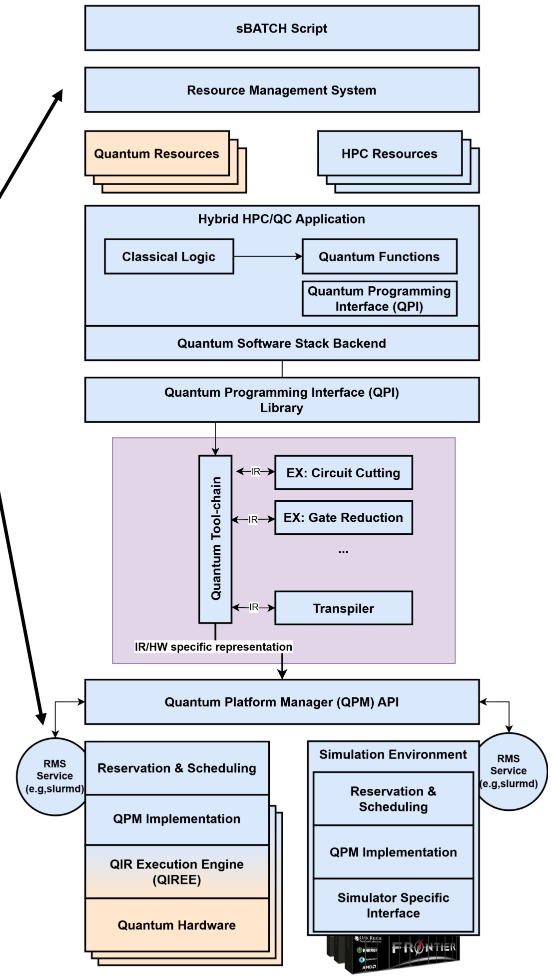
Architecture: Quantum Programming Interface (QPI)



Mesh: Container of devices
Device: Represents a quantum resource
q-stream: A set of operations to perform
Tool pipeline: A set of tools to apply on quantum tasks
Completion Queue: Receives completion notifications
Event Queue: Receives event notifications

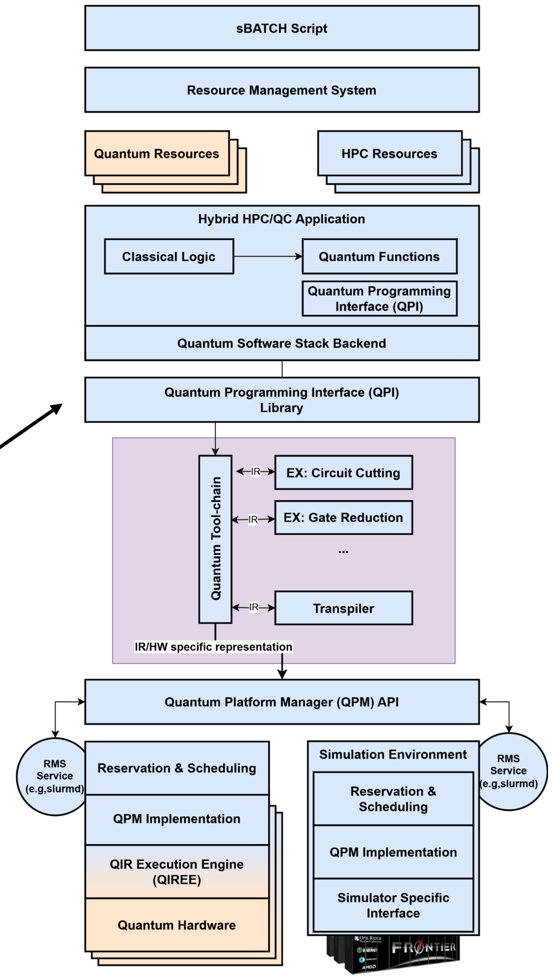
Architecture: Detailed

- User writes an **sbatch** script **outlining** the **resources** needed for their hybrid application
- The resource manager **reserves** both **Quantum** and Classical **HPC** resources
- The SLURMd (or similar) runs on the **quantum gateway** and manages hardware specific **reservation policies**



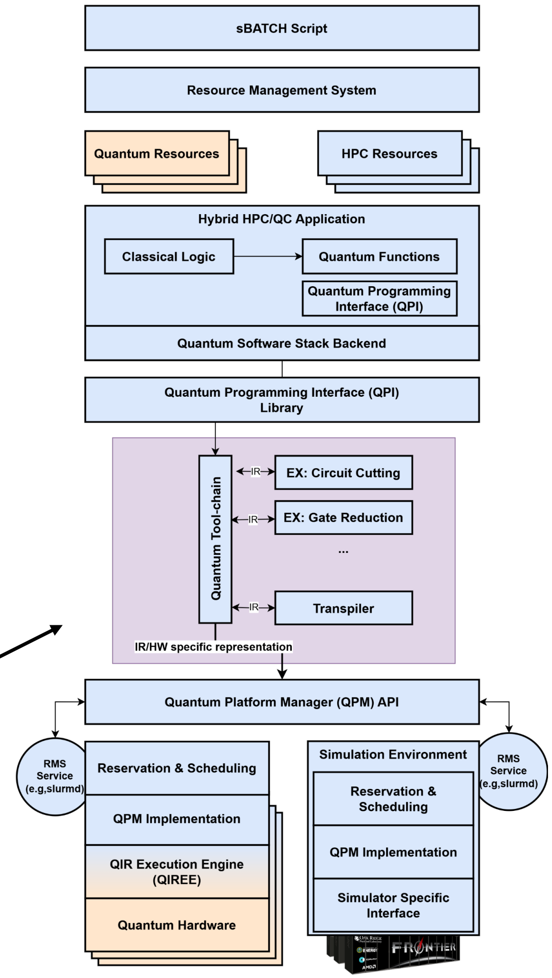
Architecture: Detailed

- The hybrid **application** starts **running** on the **HPC** allocation
- It uses the **Quantum Programming Interface (QPI)** to initiate operations which require quantum resources



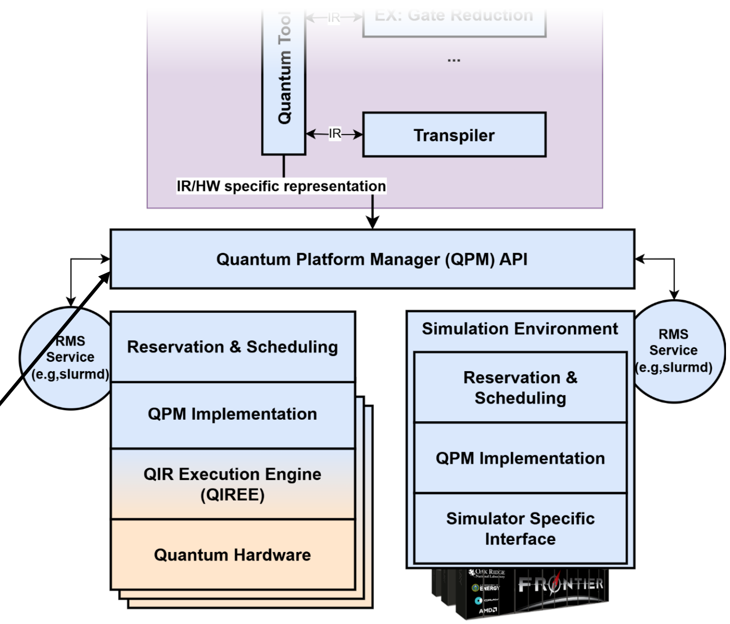
Architecture: Detailed

- Quantum circuits are fed through a tool pipeline which performs user specified operations on the circuit
- The last step is to transpile the circuit to a hardware specific format which is then passed to the HW for execution via the Quantum Platform Manager (QPM)
- The QPM is used to abstract the hardware platform



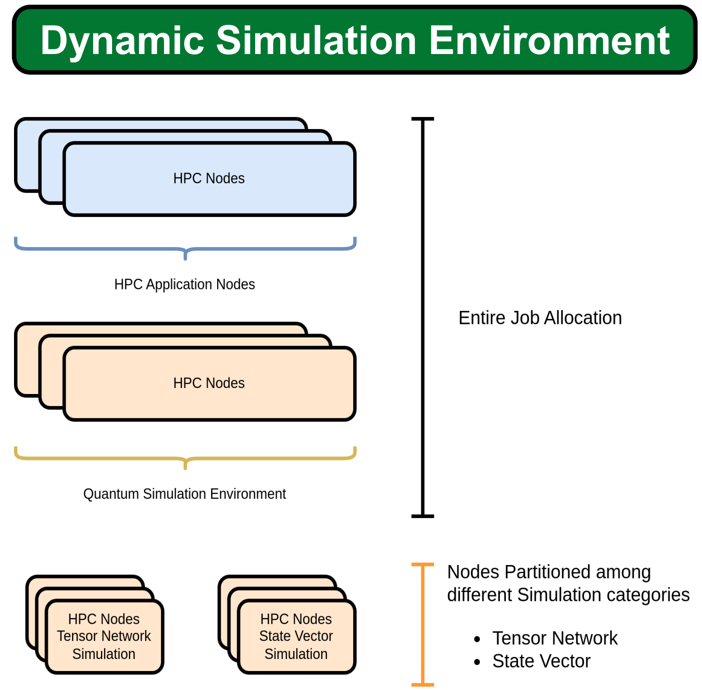
Architecture: Detailed

- Hardware specific **quantum circuits** are **queued** on the **quantum gateway**
- Quantum gateway has the **hardware specific** QPM API **implementation**
- QPM provides a **scheduling library** which can be used by the QPM to **schedule tasks** from different jobs
- **QIR Execution Engine** can be an option for driving the **quantum controller**



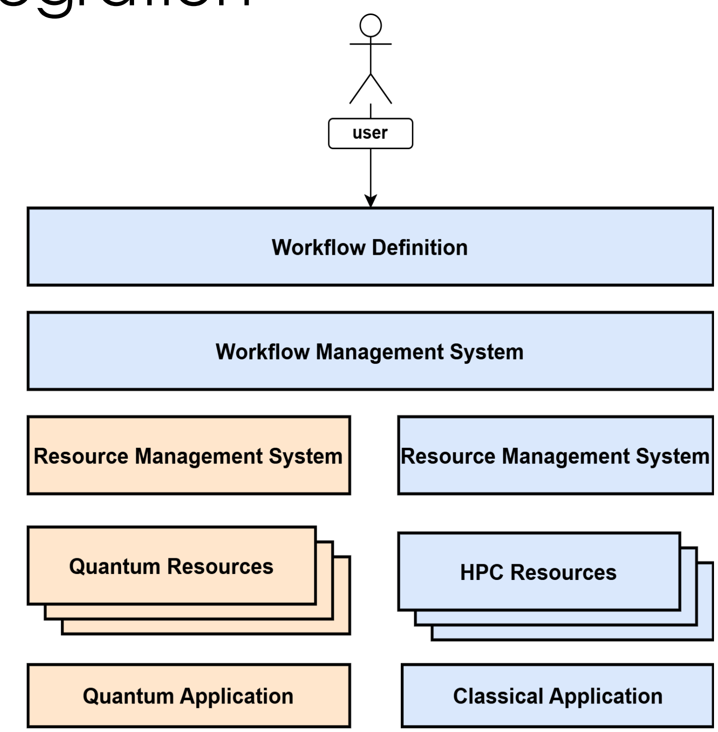
Simulation Environment

- Leverage HPC compute resources to power the Quantum Simulation
  - Allocated distinct set of resources
- Different types of simulators
  - Trade-offs in representing quantum states on classical resources, e.g.,
    - State Vector
    - Tensor Network
- Quantum Platform Manager (QPM)
  - Can partition QC resources for usage scenarios & types of simulators



Workflow Management System Integration

- The proposed software stack handles both interleaved and simultaneous allocation modes
- WMS can leverage this capability to schedule classical jobs, quantum jobs or hybrid HPC/QC Jobs
- This design also allows for the eventual integration with OLCF's Secure Scientific Mesh (S3M)
  - S3M will enable controlled access to QC/HPC resources through policy-driven interfaces



The QFw Deep Dive - SLURM Batch Script

```
#!/bin/bash

# job component 1
#SBATCH -A stf008
#SBATCH -N 1
#SBATCH --partition=compute
#SBATCH --ntasks 1
#SBATCH --ntasks-per-node=1
#SBATCH --threads-per-core 1
#SBATCH -t 1:00:00

#SBATCH hetjob

# Heterogeneous job definition for the QC node
#SBATCH --partition=quantum          # Partition for QC resources
#SBATCH --nodes=1                    # Request 1 QC node
#SBATCH --ntasks=1                   # Typically, a QC node would handle one task at a time
#SBATCH --gres=qc:superconducting:1   # Request 1 superconducting QC node
#SBATCH --time=01:00:00              # Job time limit for QC tasks (1 hour)
```

Heterogeneous Feature

```
#!/bin/bash

# job component 1
#SBATCH -A stf008
#SBATCH -N 1
#SBATCH --partition=compute
#SBATCH --ntasks 1
#SBATCH --ntasks-per-node=1
#SBATCH --threads-per-core 1
#SBATCH -t 1:00:00

#SBATCH hetjob

# Heterogeneous job definition for the QC node
#SBATCH --partition=quantum          # Partition for QC resources
#SBATCH --nodes=1                    # Request 1 QC node
#SBATCH --ntasks=1                   # Typically, a QC node would handle one task at a time
#SBATCH --gres=qc:superconducting:1   # Request 1 superconducting QC node
#SBATCH --time=01:00:00              # Job time limit for QC tasks (1 hour)
```

General Resource (GRES) Feature

```
#!/bin/bash

# job component 1
#SBATCH -A stf008
#SBATCH -N 1
#SBATCH --partition=compute
#SBATCH --ntasks 1
#SBATCH --ntasks-per-node=1
#SBATCH --threads-per-core 1
#SBATCH -t 1:00:00

#SBATCH hetjob

# Heterogeneous job definition for the QC node
#SBATCH --partition=quantum          # Partition for QC resources
#SBATCH --nodes=1                    # Request 1 QC node
#SBATCH --ntasks=1                   # Typically, a QC node would handle one task at a time
#SBATCH --gres=qc:superconducting:1   # Request 1 superconducting QC node
#SBATCH --time=01:00:00              # Job time limit for QC tasks (1 hour)
```

SLURM Job Header

```
uname -a
echo "# START-TIME: $(date)"
echo "#            SLURM_NNODES: $SLURM_NNODES"
echo "#            SLURM_NPROCS: $SLURM_NPROCS"
echo "#             SLURM_JOBID: $SLURM_JOBID"
echo "# SLURM_JOB_CPUS_PER_NODE: $SLURM_JOB_CPUS_PER_NODE"
echo "#  SLURM_THREADS_PER_CORE: $SLURM_THREADS_PER_CORE"
echo "#----"

module list

echo "##################################"

./qfw_supermarq.sh asyn 4 20 0

echo "# RC=$?"
echo "#########"

echo "# END-TIME: $(date)"
```

Run the Application

```
uname -a
echo "# START-TIME: $(date)"
echo "#            SLURM_NNODES: $SLURM_NNODES"
echo "#            SLURM_NPROCS: $SLURM_NPROCS"
echo "#             SLURM_JOBID: $SLURM_JOBID"
echo "# SLURM_JOB_CPUS_PER_NODE: $SLURM_JOB_CPUS_PER_NODE"
echo "#  SLURM_THREADS_PER_CORE: $SLURM_THREADS_PER_CORE"
echo "#----"

module list

echo "##################################"

./qfw_supermarq.sh asyn 4 20 0

echo "# RC=$?"
echo "#########"

echo "# END-TIME: $(date)"
```

Simulation case

```
#!/bin/bash

#SBATCH --output=/ccs/home/shehataa/batch/hetero_comp01.out

# job component 1
#SBATCH -A stf008
#SBATCH -N 1
#SBATCH --ntasks 1
#SBATCH --ntasks-per-node=1
#SBATCH --threads-per-core 1
#SBATCH -t 1:00:00

#SBATCH hetjob

# job component 2
#SBATCH -A stf008
#SBATCH -N 2
#SBATCH --ntasks 1
#SBATCH --ntasks-per-node=1
#SBATCH --threads-per-core 1
#SBATCH -t 1:00:00
```

Run the Application in the Simulation Environment

```
uname -a
echo "# START-TIME: $(date)"
echo "#             SLURM_NNODES: $SLURM_NNODES"
echo "#             SLURM_NPROCS: $SLURM_NPROCS"
echo "#              SLURM_JOBID: $SLURM_JOBID"
echo "# SLURM_JOB_CPUS_PER_NODE: $SLURM_JOB_CPUS_PER_NODE"
echo "#  SLURM_THREADS_PER_CORE: $SLURM_THREADS_PER_CORE"
echo "#----"

module list

echo "##################################"

./qfw_supermarq.sh asyn 4 20 0

echo "# RC=$?"
echo "#########"

echo "# END-TIME: $(date)"
```

Setting up the QFw

```bash
#!/bin/bash

module use /sw/frontier/qhpc/modules/
module load quantum/qsim

module list

set -xe

qfw_setup.sh

run_application.sh "$QFW_PATH/../applications/test_supermarq.py" --run $1 \
                        --iterations $2 --startqbit $3 --increase $4

qfw_teardown.sh

```

NOTE: Script run in allocation

Running the Application

```
#!/bin/bash

module use /sw/frontier/qhpc/modules/
module load quantum/qsim

module list

set -xe

qfw_setup.sh

run_application.sh "$QFW_PATH/../applications/test_supermarq.py" --run $1 \
                        --iterations $2 --startqbit $3 --increase $4

qfw_teardown.sh

```

Tearing down the QFw

```
#!/bin/bash

module use /sw/frontier/qhpc/modules/
module load quantum/qsim

module list

set -xe

qfw_setup.sh

run_application.sh "$QFW_PATH/../applications/test_supermarq.py" --run $1 \
                        --iterations $2 --startqbit $3 --increase $4

qfw_teardown.sh

```

Manual Resource Allocation for Simulation

```
shehataa@login1.borg:~$ salloc -N 1 -t 1:0:00 -A stf008 : -N 2 -t 1:0:00 -A stf008
salloc: Pending job allocation 208012
salloc: job 208012 queued and waiting for resources
salloc: job 208012 has been allocated resources
salloc: Granted job allocation 208012
salloc: Waiting for resource configuration
salloc: Nodes borg005 are ready for job
shehataa@borg005.borg:~$
shehataa@borg005.borg:~$
shehataa@borg005.borg:~$ squeue -u shehataa
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
          208012+0     batch interact shehataa  R       0:07      1 borg005
          208012+1     batch interact shehataa  R       0:07      2 borg[010-011]
shehataa@borg005.borg:~$
```

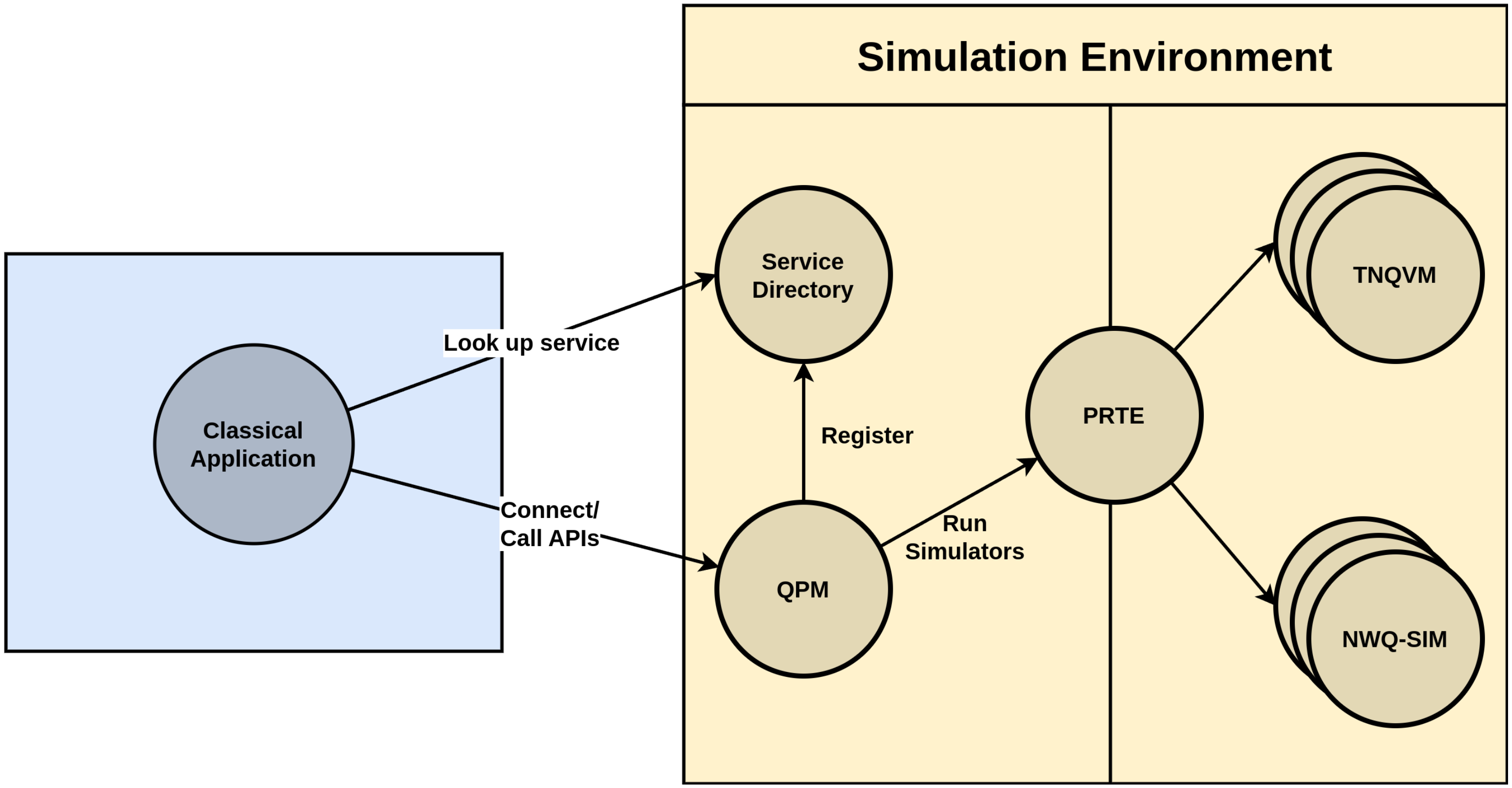
Heterogeneous Allocation

```
shehataa@login1.borg:~$ salloc -N 1 -t 1:0:00 -A stf008 : -N 2 -t 1:0:00 -A stf008
salloc: Pending job allocation 208012
salloc: job 208012 queued and waiting for resources
salloc: job 208012 has been allocated resources
salloc: Granted job allocation 208012
salloc: Waiting for resource configuration
salloc: Nodes borg005 are ready for job
shehataa@borg005.borg:~$
shehataa@borg005.borg:~$
shehataa@borg005.borg:~$ squeue -u shehataa
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
          208012+0     batch interact shehataa  R       0:07      1 borg005
          208012+1     batch interact shehataa  R       0:07      2 borg[010-011]
shehataa@borg005.borg:~$
```

Run 1 Iterations of a 20 Qubit Circuit

```
shehataa@borg005.borg:examples$ ./qfw_supermarq.sh async 1 20 0
```

The QFw Startup: Process Diagram



Directly interfacing with the QFw

- Get the instance of the Resource manager
- Get a reference to the QPM API

```
# Grab a qpm if one exists
rmgr = defw_get_resource_mgr()
qpm = defw_reserve_service_by_name(rmgr, 'QPM')[0]
```

Directly interfacing with the QFw

- Get the instance of the Resource manager
- Get a reference to the QPM API

```
# Grab a qpm if one exists
rmgr = defw_get_resource_mgr()
qpm = defw_reserve_service_by_name(rmgr, 'QPM')[0]
```

- Run the circuit

```
if runtype == "sync":
    run_circuit(qpm, startqbit, startqbit+iterations)
elif runtype == "async":
    async_run_circuit(qpm, start_qubits=startqbit,
                      itr=iterations, increase=increase)
```

Generating the Circuit

- Generate circuit
- Convert to QASM 2.0 and create info structure

```
def run_circuit(api, start, end):
    for x in range(start, end):
        ghz = supermarq.benchmarks.ghz.GHZ(num_qubits=x)
        cir = ghz.circuit()
        qasm = cir.to_qasm()

        info = {}
        info['qasm'] = qasm
        info['num_qubits'] = x
        info['num_shots'] = 1
        info['compiler'] = 'staq'
```

Create the Circuit with the QFw

- Circuit ID (cid) is returned by the QFw

```
cid = qpm.create_circuit(info)
prformat(fg.orange+fg.bold, f"running {cid}:\n{qasm}")
qpm.async_run(cid)
```

Run the Circuit with the QFw

- Synchronous run returns the circuit result
- Asynchronous run returns immediately

```
cid = qpm.create_circuit(info)
prformat(fg.orange+fg.bold, f"running {cid}:\n{qasm}")
qpm.async_run(cid)
```

Result output

```
finished 87c9cd81-29a9-4331-9e27-a2c4802a1c01:

AcceleratorBuffer:
 Information: {}
 Measurements:
   '11111111111111111111': 1
 name: qreg_0x767470
 size: 20


****1 20 qubit circuits completed in 65.61467981338501
```

Run 4 Iterations of a 20 Qubit Circuit

```
shehataa@borg005.borg:examples$ ./qfw_supermarq.sh async 4 20 0
```

Result Obtained in about the Same Time

```
finished a01dc596-8048-4729-8c8d-e2f66ffff4f8:

AcceleratorBuffer:
 Information: {}
 Measurements:
   '00000000000000000000': 1
 name: qreg_0x767470
 size: 20

****4 20 qubit circuits completed in 69.38672184944153
```

Using the QFw Backend with a QAOA

- Import the QFw Simulator backend

```
from qiskit_aer import AerSimulator
# ------------------ QFW simulator ------------------- #
from qfw_qiskit import QFWSimulator
# ---------------------------------------------------- #
```

Importing and Instantiating the QFw Backend

- Import the QFw simulator backend

```
from qiskit_aer import AerSimulator
# ------------------- QFW simulator ------------------- #
from qfw_qiskit import QFWSimulator
# ----------------------------------------------------- #
```

- Create a QFw simulator backend instance

```
# ------------------- AER simulator ------------------- #
# simulator_obj = AerSimulator(method="statevector")
# ----------------------------------------------------- #
# ------------------- QFW simulator ------------------- #
simulator_obj = QFWSimulator(simulator=sim_type)
# ----------------------------------------------------- #
```

Use the QFw backend Instance

- Create a backend sampler to be used with QAOA

```
# ------------------- AER simulator ------------------- #
# simulator_obj = AerSimulator(method="statevector")
# ---------------------------------------------------- #
# ------------------- QFW simulator ------------------- #
simulator_obj = QFWSimulator(simulator=sim_type)
# ---------------------------------------------------- #

backend_sampler = BackendSampler(
    backend = simulator_obj,
    skip_transpilation = False,
    options = {"shots": 1024}
)
```

Define the QAOA

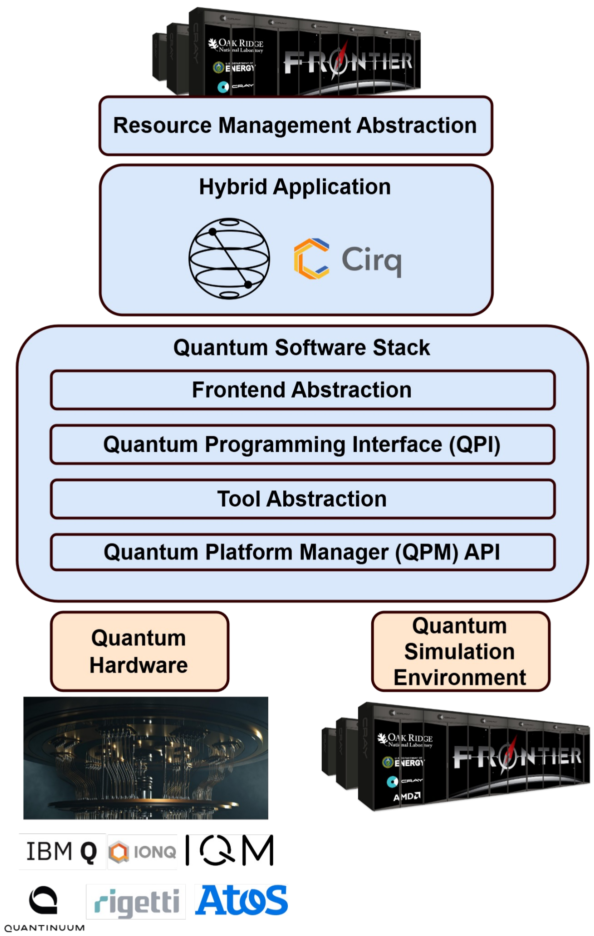
```
# Define QAOA
qaoa_mes = QAOA(
        sampler = backend_sampler,
        optimizer = COBYLA(),
        initial_point = [0.0, 1.0]
)

```

Solve the QAOA using the backend

```

# Solve the problem using QAOA
qaoa_optimizer = MinimumEigenOptimizer(qaoa_mes)
qp = maxcut.to_quadratic_program()

# Simulate on the backend
qaoa_result = qaoa_optimizer.solve(qp)

```

Takeaways

- **Versatile Stack** – Supports NISQ and future quantum systems, integrated with HPC.
- **Formalized Interfaces** – Enables integration of diverse implementations of software layers and tools
- **Optimized Architecture** – Manages scheduling, jobs, and data movement.
- **Seamless Integration** – Works with scientific apps and workflows.
- **Flexible Deployment** – Supports on-prem and potentially cloud quantum hardware.



Future Plans

- Work with the **community** to detail the Quantum Programming Interface (**QPI**)
  - Example: What API categories make sense for quantum applications.
- **Explore** quantum h**ardware** features
  - Engage with vendors: IBM-Q, IonQ, IQM, Quantinuum, etc.
  - Identify the QPM APIs
- Work with the **community** to detail the **tool chain interface**
  - Identify the optimal interface to allow the integration of new circuit transformation tools
- Achieve **efficient** quantum-classical **resource utilization**
  - Research strategies for two-level scheduling
  - Research strategies for virtualizing a Quantum Computer
  - Heuristics for circuit/resource mappings to improve utilization

ORNL Quantum Systems & Software Workshop (OQSSw)

- QCUF: July 21-24, 2025
- OQSSw: July 25, 2025
  - Registration link: https://www.olcf.ornl.gov/calendar/oqssw/

Questions?

- Supported by
  - This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Questions for IQM

- **Error Correction Requirements** – Will quantum error correction require high-bandwidth classical compute resources like GPUs and CPUs, or would a low-power processor (e.g., FPGA) suffice?
- **HPC Simulators** – Do you anticipate a continued need for quantum simulators running on HPC, or will their utility diminish due to increasing memory demands as qubit counts grow?
- **Hardware Access APIs** – Do you provide an API for direct hardware access? We could leverage this to define the Quantum Processing Management (QPM) APIs.
- **Circuit Runtime Estimation** – Have you considered the best approach for calculating how long a circuit will take to execute on your system?
- **Circuit Size Variability** – Based on your experience, do most applications generate circuits of consistent size, or are there cases where circuit size varies significantly?
- **Performance Benchmarking** – What metrics are most relevant for evaluating hybrid HPC-QC performance?