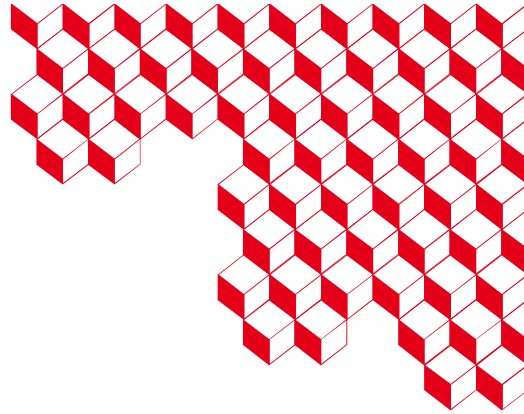
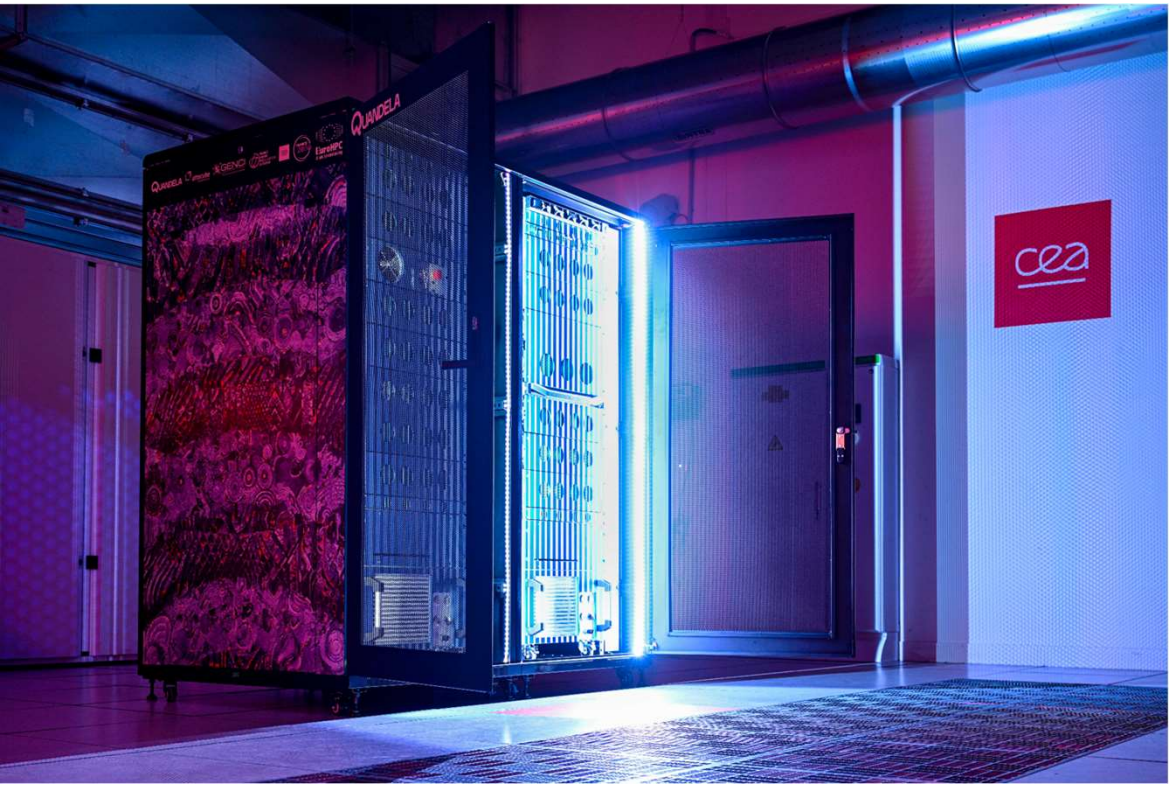


Integrating actual QPU in Computing Center architecture with Qaptiva

Cyril ALLOUCHE (cyril.allouche@eviden.com), Jean SENELLART (jean.senellart@quandela.com), Philippe DENIEL (philippe.deniel@cea.fr)

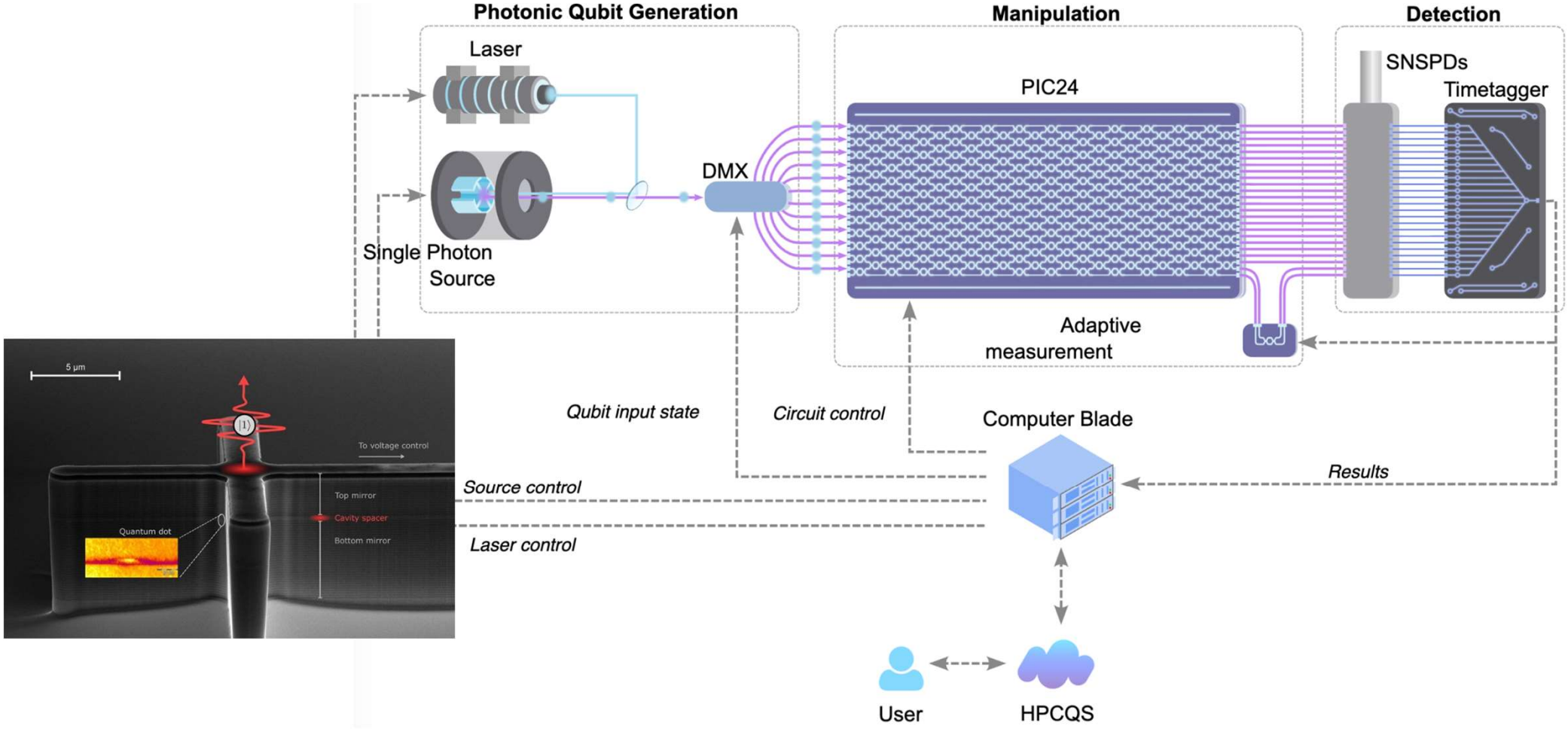
Lucy, a new Photonic QPU in CEA HPC



- Delivered in October 2025 and deployed in 3 days.
- 12-photon universal photonic quantum computer
- Comes with perceval, MerLin and Quandela Quantum Toolbox
- Designed for cloud deployment with integrated scheduler/user management/orchestration

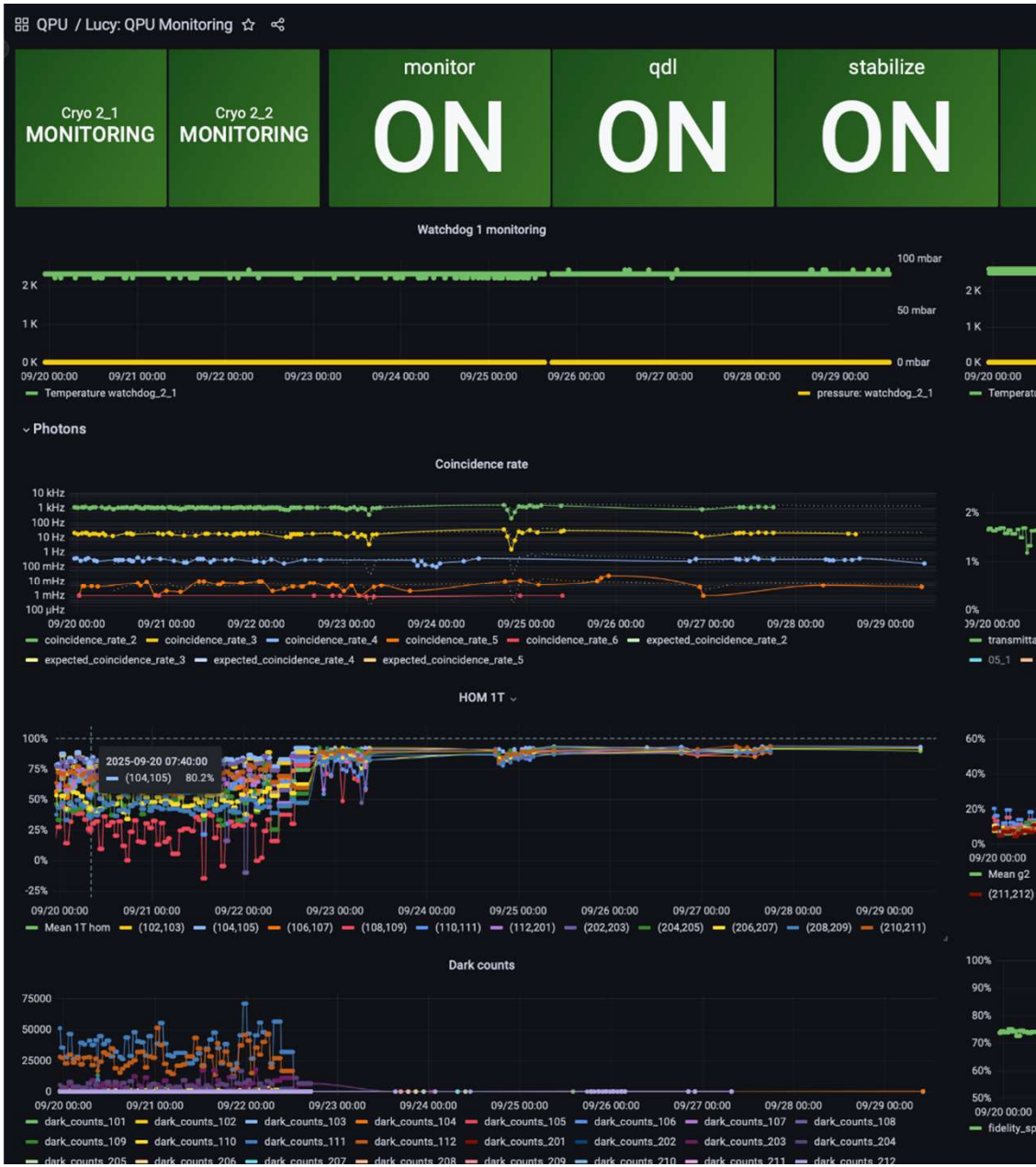
cea EVIDEN QUANDELA

Lucy, under the hood



Lucy, designed for HPC

- Fully integrated to fit in standard 19” racks
- Monitor almost 100 performance indicators to check for anomaly
- Automated calibration running on an hourly basis to adapt to changing environment
- Internal redundancy: double cryogenic system for failover of single photon sources



cea EVIDEN QUANDELA

Why Qaptiva helps integrating QPUs in compute center

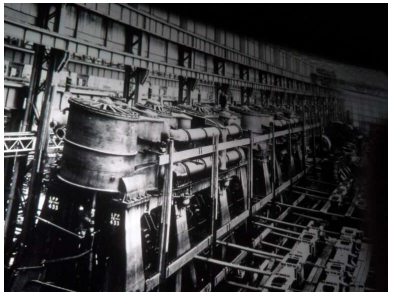
- Running jobs on a QPU means
  - scheduling the users jobs wisely on an exclusive and unique compute resource
  - knowing precisely the compute time consumption on the QPU, user by user (accounting)
  - identify / authenticate the users whose time consumption is to be accounted (authentication)
- For each QPU, the authentication/accounting/scheduling “tripod” is required
  - every vendor has to reinvent the wheel, without knowing what the other vendors do
- Qaptiva offers this tripod, plus the qlm/interop feature what turns it to a proxy
  - users submit jobs to Qaptiva
  - Qaptiva does all the “tripod black magic” and submit the job to the QPU in a standard way
  - all QPU have the same kind of interface and similar API
  - as similar APIs are used, it becomes possible for compute code to use several QPUs
- Qaptiva can act as an emulated, using the exact same API
  - Switching from real QPU / emulated QPU cost very small changes in the code

What is needed

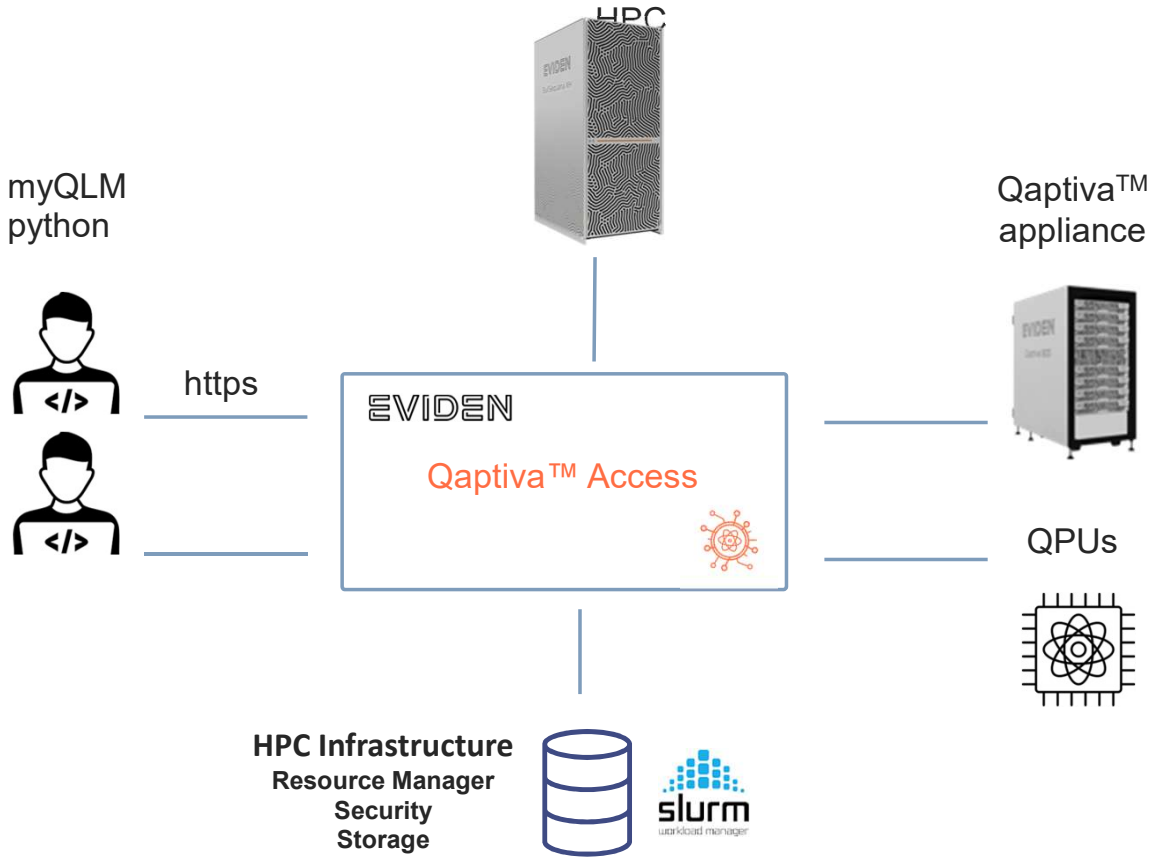
- In order to use Qaptiva as a proxy
  - a QPU Handler instance as to be developed
  - this library will bridge the QPU and Qaptiva
  - The actual QPU has to be defined as a “Remote QPU” in Qaptiva
- Perceval now has a perceval/interop library
  - wraps the Perceval code to Qaptiva compute jobs
  - Those jobs can be either emulated on Qaptiva or pushed to the actual QPU

Monitoring : machine rooms are no safe environment

- As seen this morning, integrating a QPU into a compute center machine room is challenging
- QPU are complex engines and most of their failures are uncommon to the classical HPC systems
- This makes the supervision and monitoring of QPU critical
  - to a supervision/control room
  - the QPU should be able to advertise its current state to the end user
  - Qaptiva makes this possible via the qlm/interop feature*
  - As the QPU is unique and is an exclusive system, users will use this status to manage their computation smarting, eventually pausing/freezing it if the QPU is not available.



QaptivaTM Access



cea

EVIDEN QUANDELA

Qaptiva / Lucy at TGCC

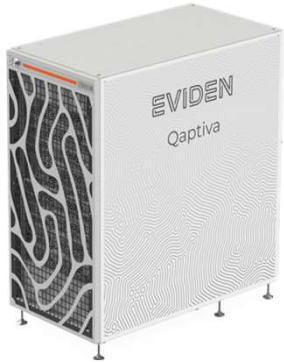
Large quantum simulations
Can be run on cluster

myQLM library deployed on compute nodes

requests for quantum resources go thru Qaptiva Access

Qaptiva Access

Qaptiva appliance handles fat node simulations, compilations, etc..



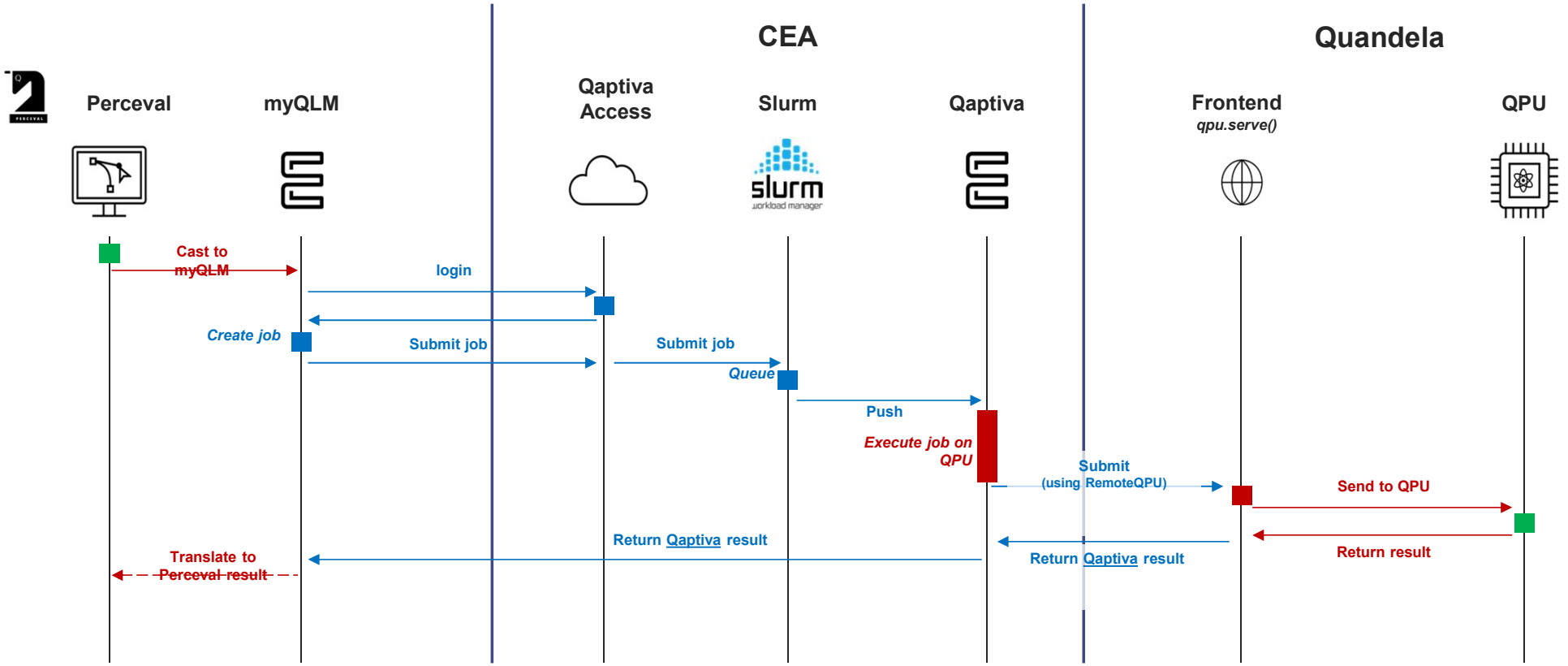
Lucy QPU receives quantum instructions from Access, with respect with Slurm scheduling

slurm
workload manager

Qaptiva delegates scheduling of quantum resources to Slurm

cea EVIDEN QUANDELA

QLM/interop in actions



Example workflow

```python
from qat.qlmaas import QLMaaSConnection
from perceval import Experiment, FockState, Circuit, Matrix, pdisplay
from perceval_interop import MyQLMHelper


command = "sample_count"
platform_qaptiva = "qat.qpus:LucyQPU"

# Create an access to the QPU
qaptiva_connection = QLMaaSConnection()
qpu_access = qaptiva_connection.get_qpu(platform_qaptiva)()

# Retrieve QPU specifications
specs = MyQLMHelper.retrieve_specs(qpu_access.get_specs())
pdisplay(specs['specific_circuit'])  # Renders the chip architecture
assert command in specs['available_commands']

# Create your Perceval experiment
experiment = Experiment(8)
experiment.set_circuit(Circuit(Matrix.random_unitary(8)))
experiment.with_input(FockState([1, 0, 1, 0, 0, 0, 0, 0]))
exp.min_detected_photons_filter(2)

# Create your job, run it and retrieve results
job = MyQLMHelper.make_job(command, exp, max_shots=10_000_000)
results = qpu_access.submit_job(job)
quandela_results_dict = MyQLMHelper.retrieve_results(results)
pdisplay(quandela_results_dict['results'])
```

qlm ← perceval_interop → perceval

connect to Lucy platform

retrieve dynamically Lucy latest specifications

define quantum circuit to run

submit the task through **myqlm** and retrieve results in **perceval**

cea EVIDEN QUANDELA

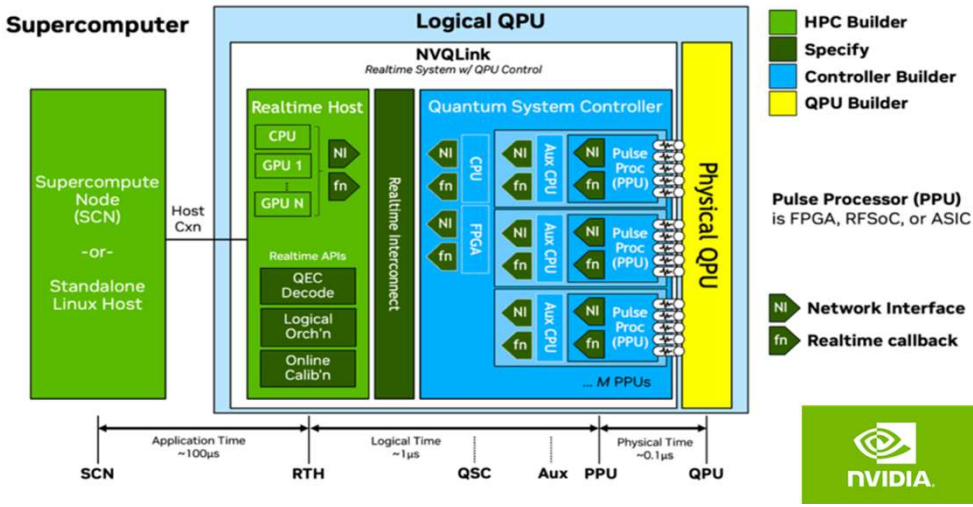
Toward a tigher integration of HPCQCAI – why and how?

why

- In NISQ regime, specific interest in hybrid applications with fast iterations between classical and quantum nodes
- Specifically, very high latencies between QPUs and GPUs can compromise practical applications of hybrid QML algorithms. Real time QEC and decoding are also challenges to achieve useful quantum computing applications.
- The larger the system, the more “classical processing” will be needed to calibrate system, process results, mitigate errors, etc… requiring larger classical resources

how

- NVQLink initiative to provide the low-latency, high throughput integration of quantum hardware and AI supercomputing needed to scale quantum computers.



cea

EVIDEN

QUANDELA

NVIDIA