QUANTINUUM

Helios and Next-Generation Stack

ORNL Training

January 2026

Zach Massa
Offering Management Lead

Irfan Khan
Sr. Applications Engineer

Image: Visual representation of Helios deployed at a customer site

Agenda

- Introduction to Quantinuum
- Hardware
- Guppy
- Job Submission on Helios
- Next generation emulation with Selene
- Wrap up and Summary

Development Roadmap

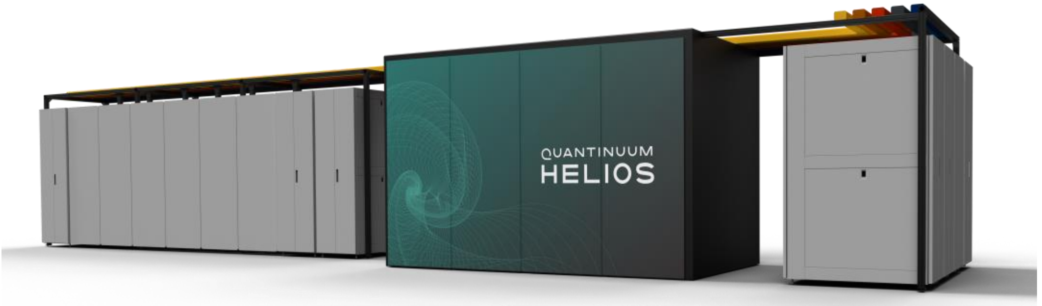
	2020	2023	2025	2027	2029	2030+
SYSTEMS	QUANTINUUM H1	QUANTINUUM H2	QUANTINUUM HELIOS	QUANTINUUM SOL	QUANTINUUM APOLLO	QUANTINUUM LUMOS
PHYSICAL QUBITS	20	56	98	192	1,000s	1,000,000+ QUBITS
PHYSICAL 2-QUBIT GATE ERROR	1×10^{-3}	1×10^{-3}	$< 5 \times 10^{-4}$	$< 2 \times 10^{-4}$	1×10^{-4}	
LOGICAL QUBITS		> 12	50	~ 100	100s	
LOGICAL ERROR RATES		1×10^{-3}	$< 10^{-4}$	$\sim 10^{-5}$	1×10^{-5} to 1×10^{-10}*	
NVIDIA GPU INTEGRATION			QEC Decoding ~1 PFLOP ~100 us latency	QEC Decoding ~10 PFLOP ~10 us latency	Control + QEC Decoding ~100 PFLOP ~0.5 us latency	

*analysis based on recent literature in new, novel error correcting codes predict that error could be as low as 1E-10 in Apollo (ref: arXiv:2403.16054, arXiv:2308.07915).

QUANTINUUM

Quantinuum is pioneering the

next step in quantum computing hardware as a service

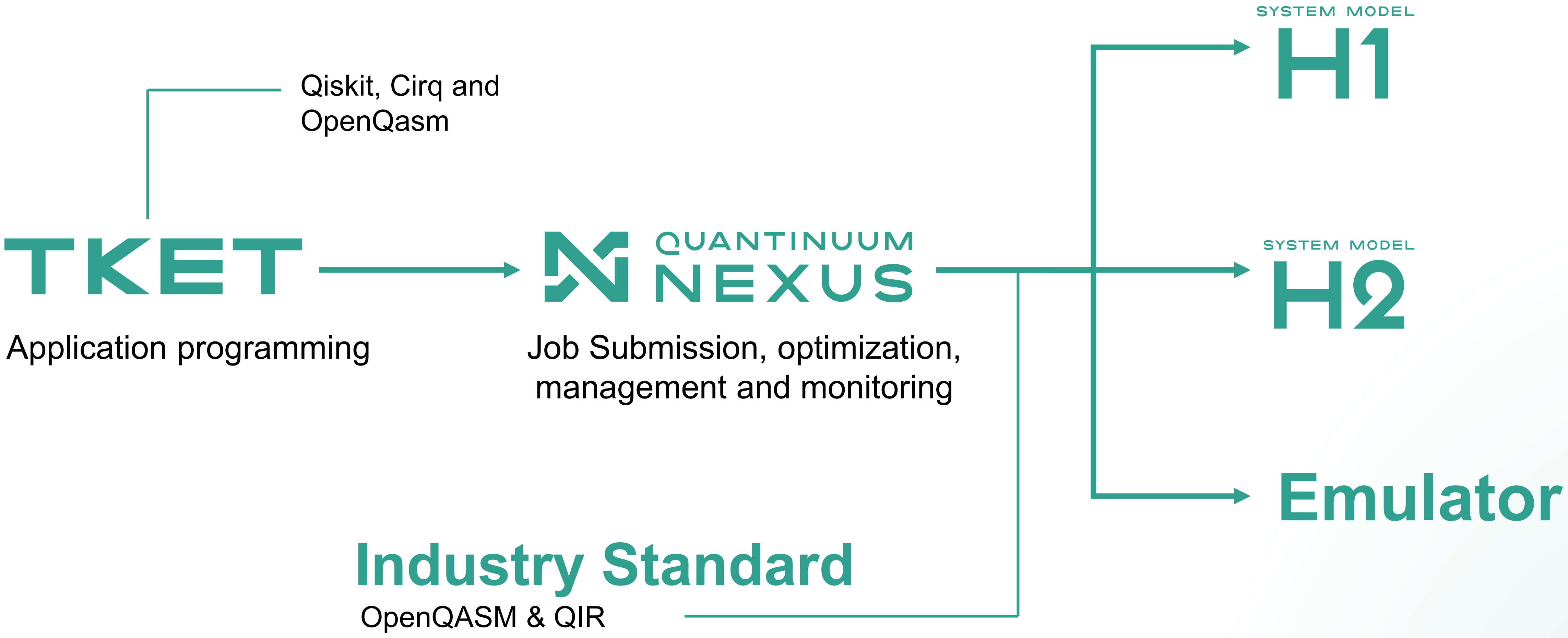


Dynamic Transport to improve time to solution & reduce the strength of memory errors.

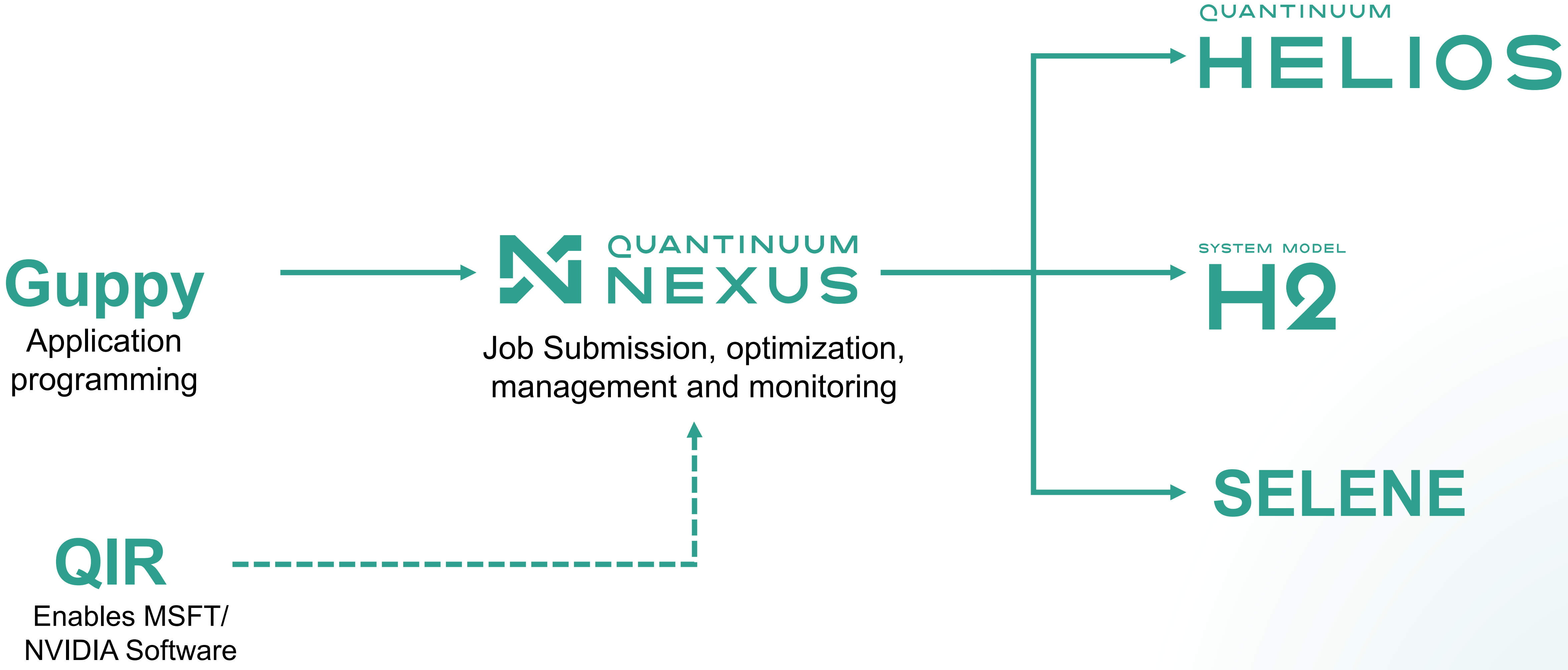
- Heralded Leakage Measurement
- Automated Qubit Resource Management
- All-to-all connectivity
- Mid-circuit measurement & reset
- Qubit Reuse Compiler
- Emulation and debugging (Selene)
- Application-level Leakage Detection
- Early Exit
- Arbitrary control flow
- Real-time Random Number Generator
- Conditional Operations
- Real-time QEC decoding
- Parameterized 2-qubit operations

*New features

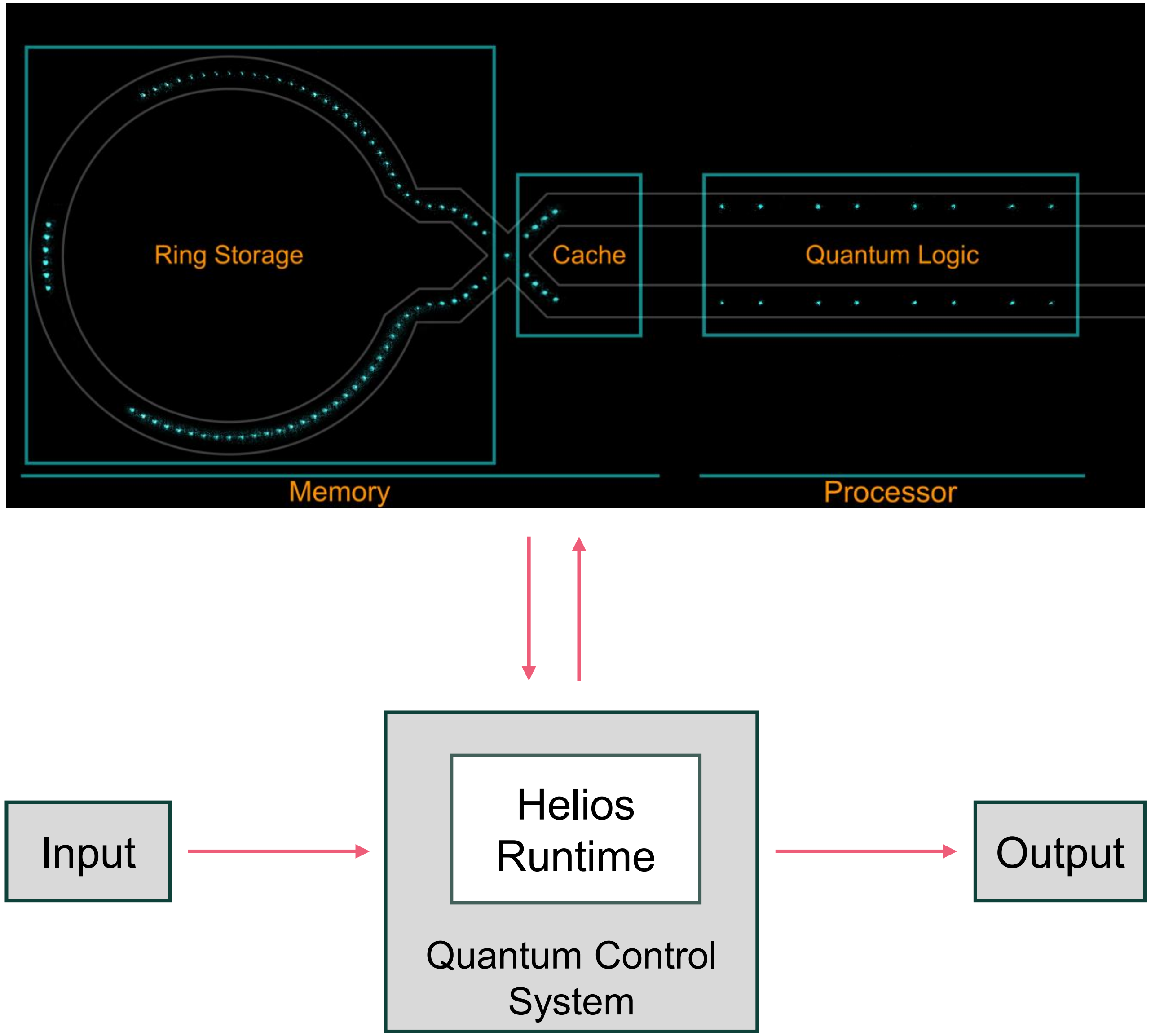
Legacy Technology Stack



Next-Generation Technology Stack



Quantinuum Helios

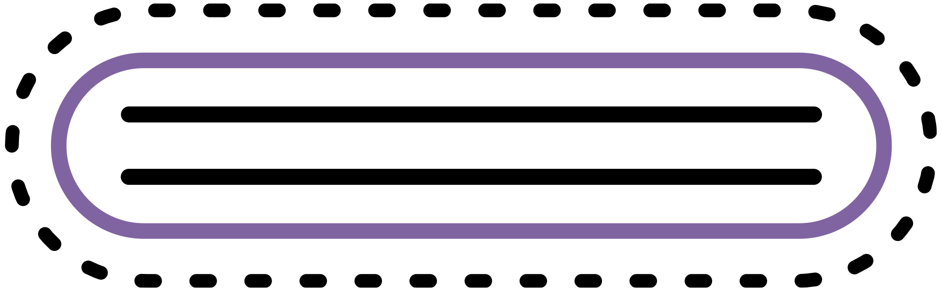


Quantum Charge Coupled Device

- **Qubits** are Barium ions
- **Dedicated zones for** logic/initialization/measure
- **High-fidelity ops** via visible light laser pulses on short ion chains
- **Sympathetic Cooling** via Ytterbium ions
- **All-to-all Connectivity** by physical transport
- **Scalability** enabled by micro-fabricated traps.

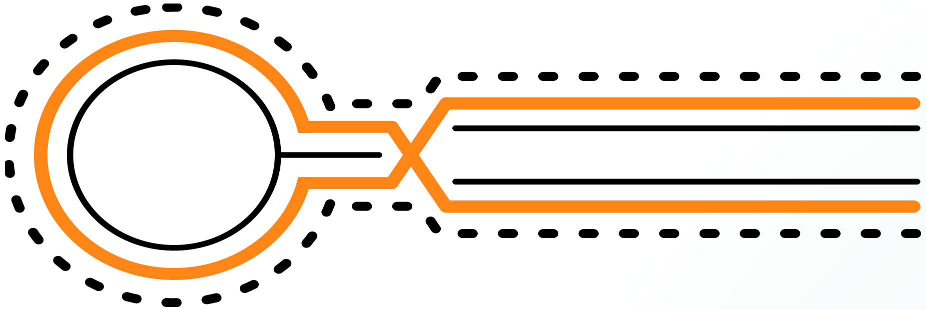
Evolution of Trap Architecture

SYSTEM MODEL
H2



Logic and storage region

QUANTINUUM
HELIOS

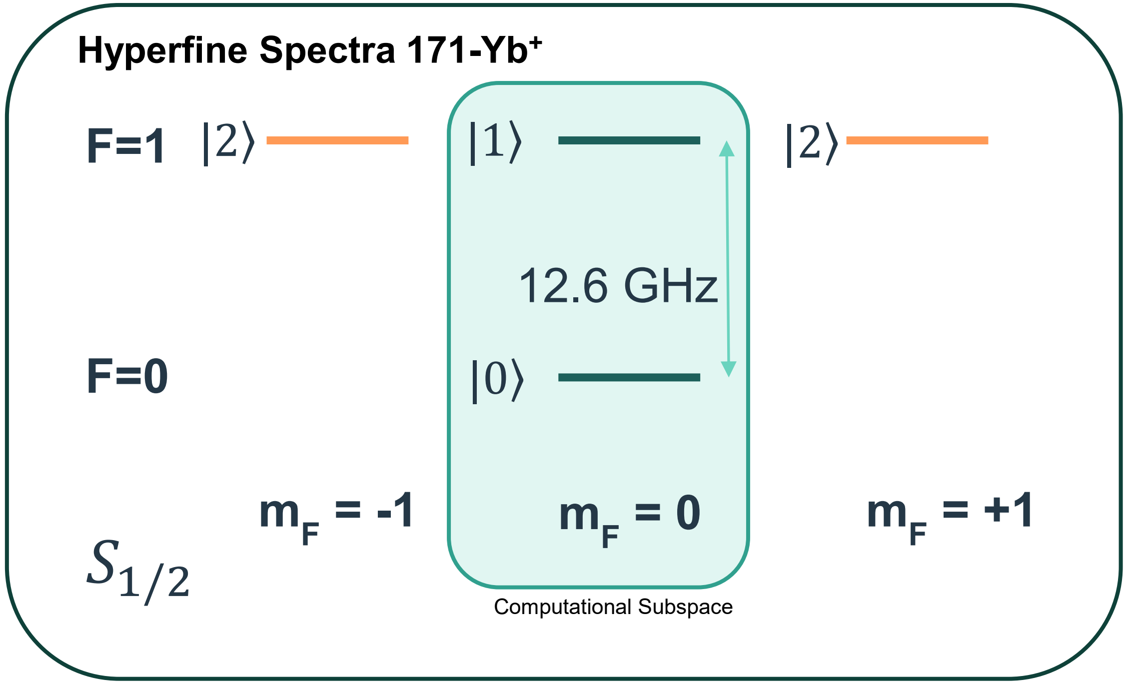


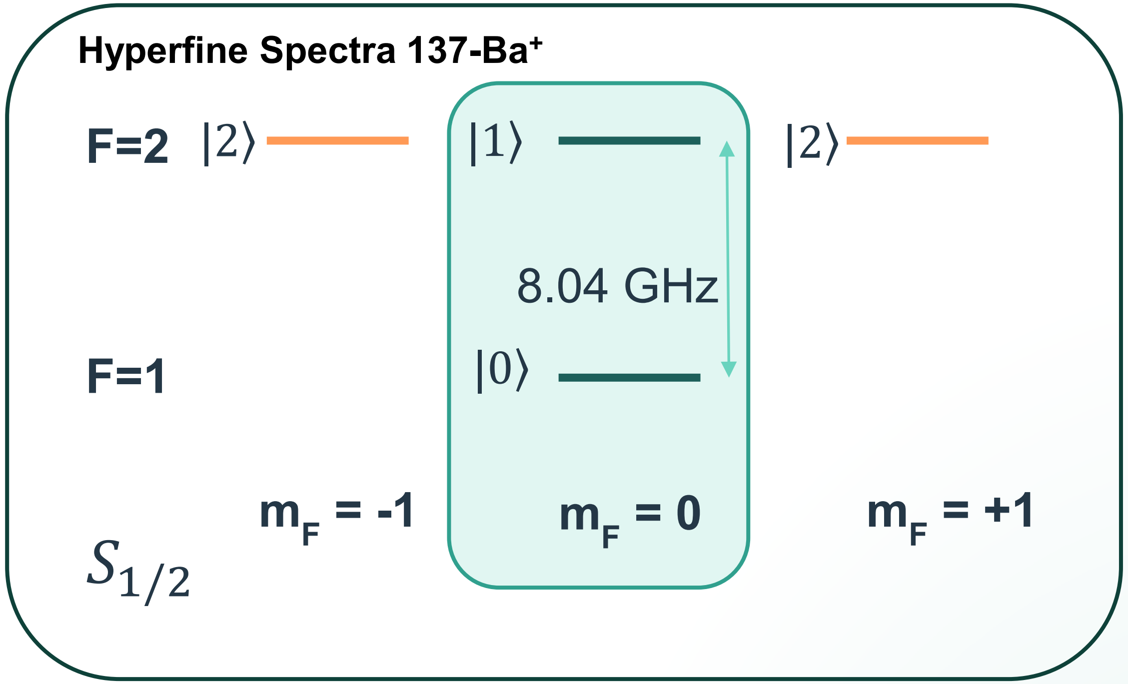
storage region

Logic region

Computational Space

Encoding in S-manifold





KEY

- Non-computational states
- Computational states

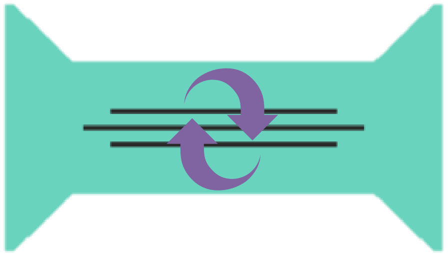
Universal Native Gateset

Gate	Expression	Guppy
Parameterized 1-Qubit Gate	$R_{xy}(\theta,\phi) = e^{\left(-\frac{i\theta}{2}\right)(\cos(\phi)X+\sin(\phi)\hat{Y})}$	`phased_x`
Parameterized 1-Qubit Software Gate	$R_z(\lambda) = e^{-\frac{i\lambda}{2}\hat{Z}}$	`rz`
Fully Entangling 2-Qubit Gate	$ZZ() = e^{-\frac{i\pi}{4}\hat{Z}\otimes\hat{Z}}$	`zz_max`
Parametrized 2-Qubit Gate	$RZZ(\theta) = e^{-\frac{i\theta}{2}\hat{Z}\otimes\hat{Z}}$	`zz_phase`

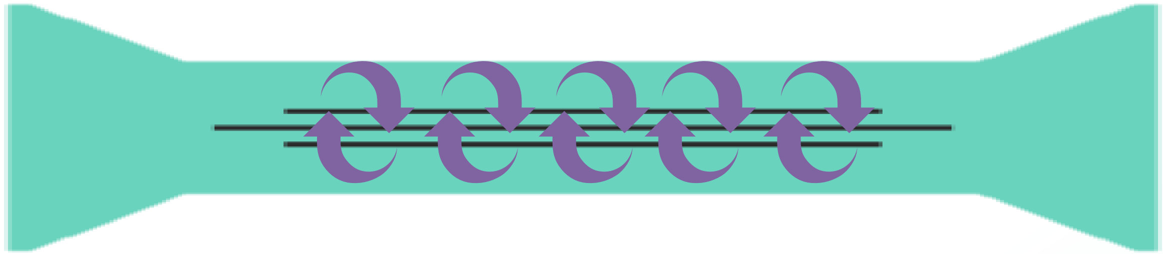
What is a QCCD junction?

First commercial QCCD junction

Linear ion trap geometry



Swap sorting time grows out of control

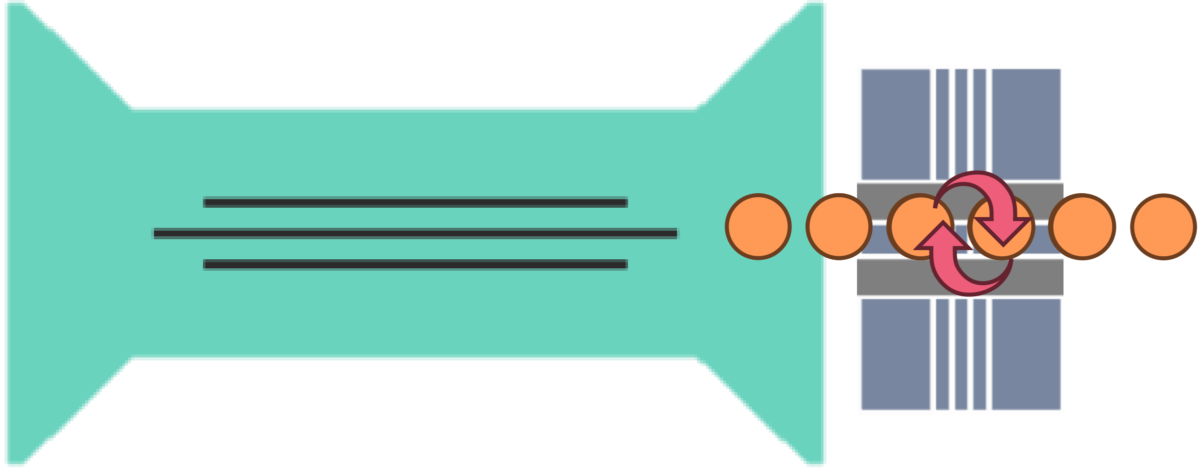


Junctions merge short linear traps

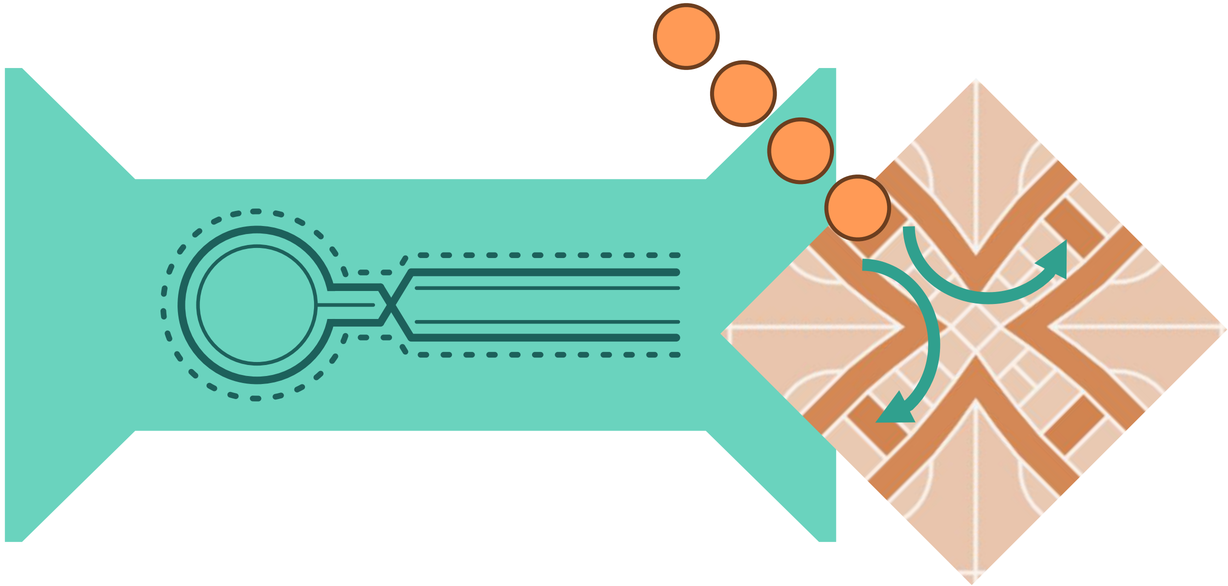
Swap sorting replaced by junction sorting

Junctions speed up qubit routing

First commercial QCCD junction



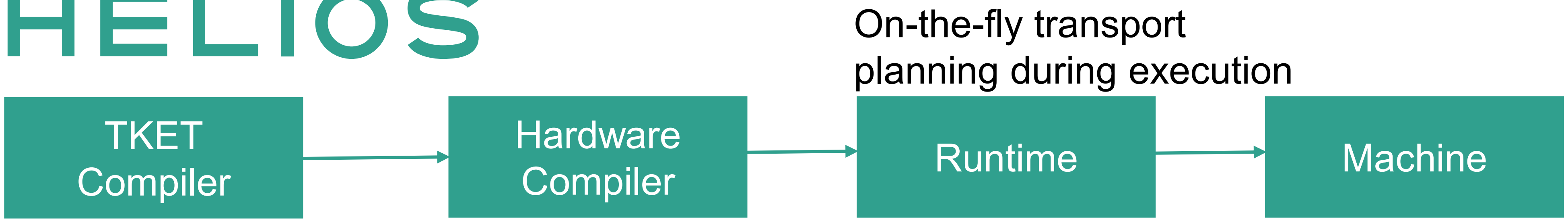
Swap sorting for all N qubits requires N operations per qubits to achieve target configuration



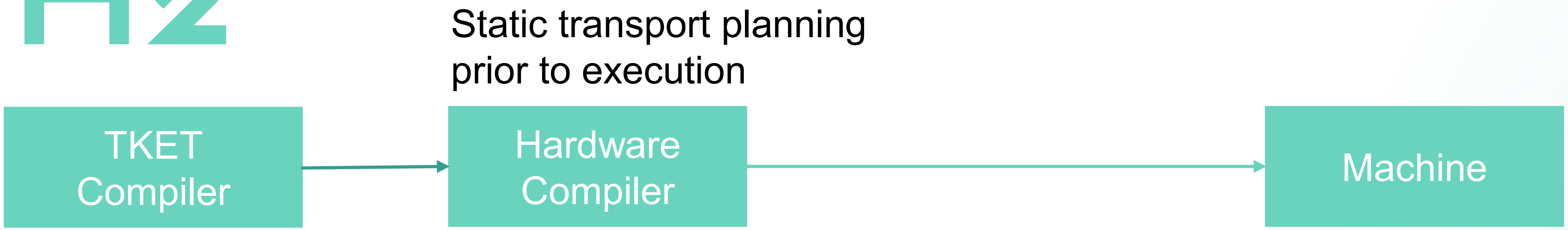
Junction sorting to access N qubits requires 1 operation per N qubits to sort

Dynamic Transport

QUANTINUUM
HELIOS



SYSTEM MODEL
H2



Quantinuum Helios

Key improvements over System Model H1/2

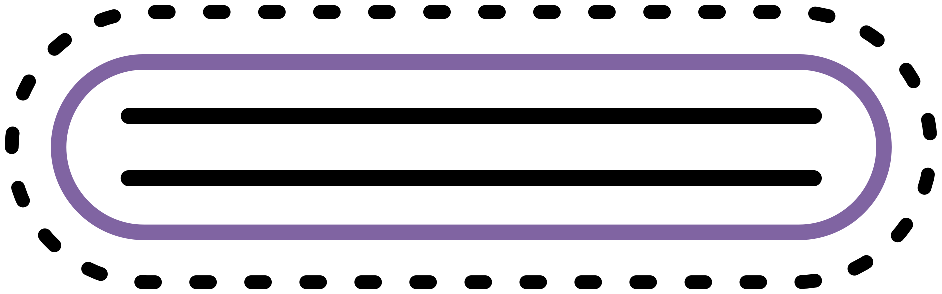
1. On-the-fly native operations (transport planning and gating operations).
2. Different qubit ion (Yb to Ba) to reduce gating errors and introduce heralded leakage measurement.
3. New ion trap with junction transport to deliver “twice-as-good” system.

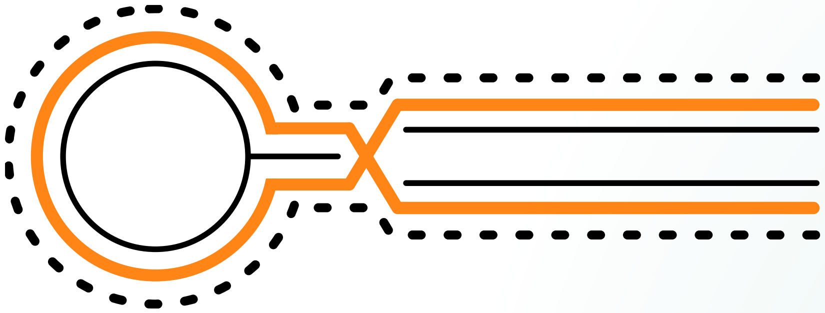
	H2	Helios
# of physical qubits	56	98
Physical 2-qubit gate infidelity	8.3×10^{-4}	7.9×10^{-4}
# of logical qubits	10+	~50
Logical 2-qubit gate infidelity	1×10^{-3}	$< 1 \times 10^{-4}$

*See below for source

SYSTEM MODEL H2

QUANTINUUM HELIOS



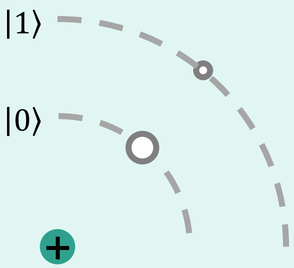


* https://docs.quantinuum.com/systems/user_guide/hardware_user_guide/performance_validation.html

QCCD architecture

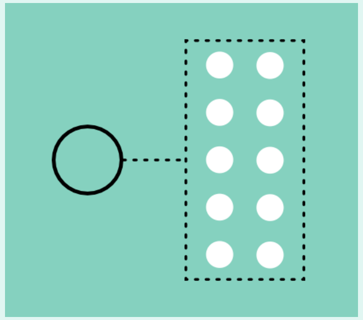
differentiating features

Long Coherence Times



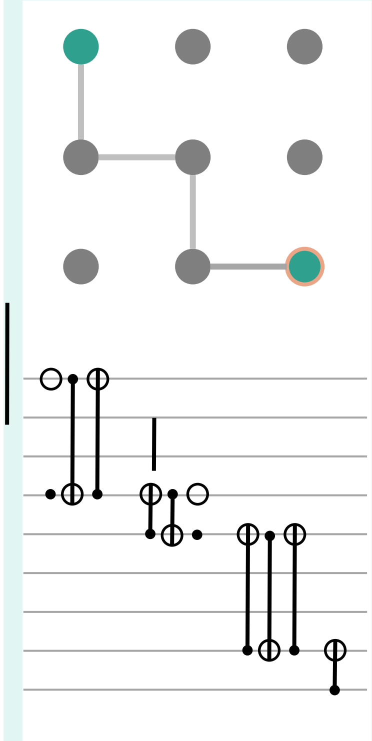
Flexible and reconfigurable

MCMR and Arbitrary Control Flow

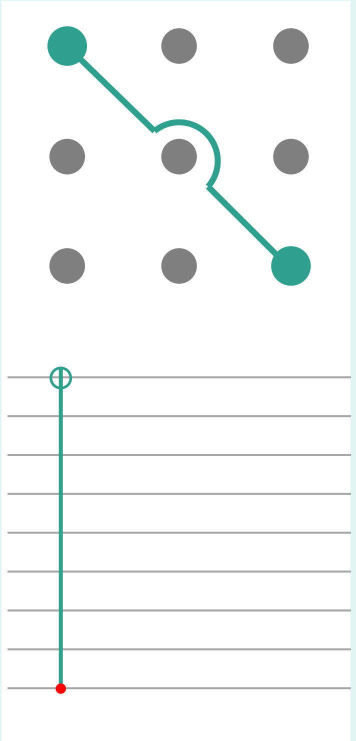


All-to-All Connectivity

Nearest Neighbor



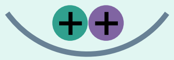
All-to-All



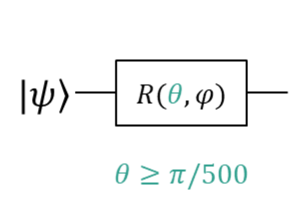
High-Fidelity Ops

Isolated Ops Zones

Sympathetic Cooling



Parameterized Angle
SQ and TQ gates



$$RZZ(\theta) = e^{-\frac{i\theta}{2}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{i\theta} & 0 & 0 \\ 0 & 0 & e^{i\theta} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Current-Generation Stack

Pytket

Nexus

Application Programming	Lowers to IR	Compilation and optimisation	Third-party conversions	Backend Submission API

IR = Intermediate Representation

Next-Generation Stack

Guppy	HUGR	TKET2	pytket	Nexus
Application Programming	Intermediate Representation (IR)	Compilation and optimisation	Third-party conversions to Guppy	Backend Submission API

Why Guppy?

Expressive

Write code that reads like the problem you're solving. Guppy's syntax is clear, concise, and flexible enough to capture complex quantum and classical logic without unnecessary boilerplate.

Safe

Guppy's type system catches costly mistakes before they happen — including quantum-specific hazards.

Pythonic

If you can read Python, you can read Guppy. Its design is inspired by Python's readability and simplicity, making it easy to learn. Guppy also lives inside Python, allowing seamless inter-op.

```
@guppy
def repeat_until_success() -> None:

    q = qubit()

    for i in range(5):
        a, b = h(qubit()), h(qubit())

        toffoli(a, b, q)
        s(q)
        toffoli(a, b, q)

        ra = measure(h(a))
        rb = measure(h(b)):

        if not (ra | rb):
            result("rus_attempts", i)

    result("q", measure(q))
```

Guppy Features

Arbitrary Control Flow

Full quantum measurement-dependent control flow: write if-else conditions, for and while loops, all in Python style.

Strongly Typed

The guppylang compiler uses a powerful but unobtrusive type system to provide helpful error messages..

Qubit Safety

Qubits have a linear type, following an intuitive ownership model that prevents no-cloning errors and memory leaks

Metaprogramming

Generate and transform Guppy code at compile time to automate patterns, optimise circuits, and reduce repetition.

Classical Compute

Perform classical calculations and data manipulation alongside quantum operations seamlessly.

Data Structures

Work with arrays, tuples, and user-defined types in both classical and quantum contexts.

First-class Functions

Define functions to write structured quantum software and pass them just like any other value.

Legacy Support

Easily integrate with existing quantum toolchains like pytket — bridging the gap with legacy circuit-building tools.

Guppy Program

```
pip install guppylang
```

- Application programming
  - @guppy decorator required to define functions.
  - All functions annotated with input types and return types.
  - main() is the program entry point.
  - result: tag measurement outcomes to results stream

- Local Compilation to *hugr*

- Execution with *selene-sim*

```python
from guppylang.decorator import guppy
from guppylang.std.quantum import qubit, cx, h

@guppy
def bell() -> tuple[qubit, qubit]:
    q0, q1 = qubit(), qubit()
    h(q0)
    cx(q0, q1)
    return q0, q1
```

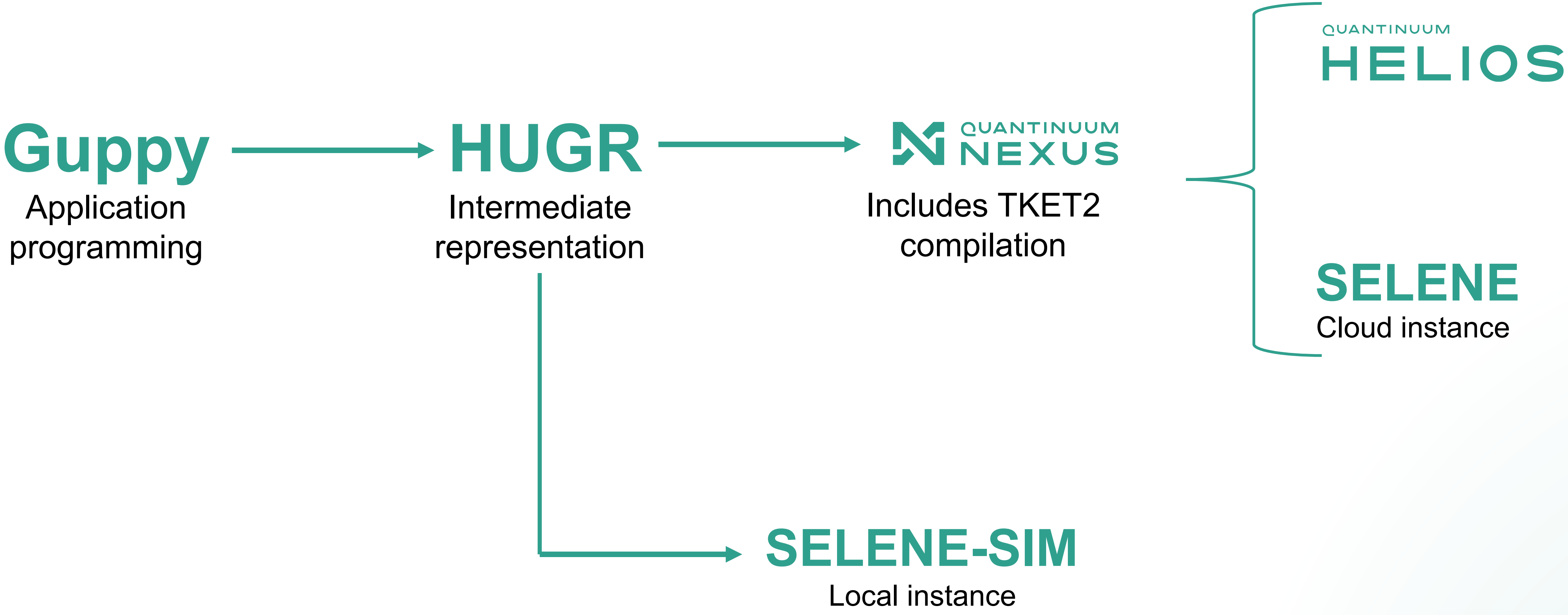
```python
from guppylang.std.quantum import measure
from guppylang.std.builtins import result

@guppy
def main() -> None:
    q0, q1 = bell()
    v0 = measure(q0)
    v1 = measure(q1)
    result("q0", v0)
    result("q1", v1)
```

```python
hugr = main.compile()
```

```python
from selene_sim import Quest

main.emulator(n_qubits=2).with_simulator(Quest()).run()
```

Technology Stack

A new language for application programming



Qubit Safety

Linearly Typed Qubit

No-cloning Theorem

Qubits cannot be copied.

No-deleting Theorem

Qubits cannot be deleted arbitrarily.

Linear Typing

- Qubits can only be used once.
- Qubits cannot be copied
- Qubits cannot be implicitly discarded

Safety & Ownership

Enforcing linearity as a constraint at compile time

1. **Qubits cannot be used after they are deallocated.**
2. A multi-qubit gate cannot use qubits more then once.
3. Qubits cannot be discarded or leaked implicitly.

```python
from guppylang import guppy
from guppylang.std.builtins import owned
from guppylang.std.quantum import qubit, h, cx, measure

@guppy
def owned_qubits(
    alice: qubit @ owned,
    bob: qubit
) -> bool:
    h(alice)
    cx(alice, bob)
    alice_c: bool = measure(alice)
    h(alice) # invalid operation
    return alice_c
```

```
h(alice) # invalid operation
   |        ^^^^^ Variable `alice` with non-copyable type
`qubit` cannot be
   |              borrowed ...
   |
13 |     alice_c: bool = measure(alice)
   |                               ----- since it was already
consumed here
```

Safety & Ownership

Enforcing linearity as a constraint at compile time

1. Qubits cannot be used after they are deallocated.
2. **A multi-qubit gate cannot use qubits more then once.**
3. Qubits cannot be discarded or leaked implicitly.

```python
from guppylang import guppy
from guppylang.std.builtins import owned
from guppylang.std.quantum import qubit, h, cx, measure

@guppy
def borrow_qubits(
    alice: qubit,
    bob: qubit
) -> None:
    x(alice)
    h(alice)
    cx(alice, bob)
    cx(alice, alice) # invalid operation
```

```
25 |     cx(alice, alice) # invalid operation
   |               ^^^^^ Variable `alice` with non-copyable
type `qubit` cannot be
   |                     borrowed ...
   |
25 |     cx(alice, alice) # invalid operation
   |        ----- since it was already borrowed here
```

Safety & Ownership

Enforcing linearity as a constraint at compile time

1. Qubits cannot be used after they are deallocated.
2. A multi-qubit gate cannot use qubits more then once.
3. **Qubits cannot be discarded or leaked implicitly.**

```python
from guppylang.std.quantum import qubit, h

@guppy
def invalid(
    alice: qubit,
    bob: qubit
) -> tuple[qubit, qubit]:
    h(alice)
    cx(alice, bob)
```

```
Error: Drop violation
38 | @guppy
39 | def main() -> None:
40 |     alice = qubit()
   |     ^^^^^ Variable `alice` with non-droppable type `qubit` is leaked

Help: Make sure that `alice` is consumed or returned to avoid the leak
```

Safety & Ownership

Enforcing linearity as a constraint

Ownership required:

- Destructive operations on qubits
- Looping directly on qubits

Owned

```
from guppylang import guppy
from guppylang.std.builtins import owned
from guppylang.std.quantum import qubit, h, cx, measure

@guppy
def owned_qubits(
    alice: qubit @ owned,
    bob: qubit
) -> bool:
    h(alice)
    cx(alice, bob)
    alice_c: bool = measure(alice)
    return alice_c
```

Borrowed

```
from guppylang import guppy
from guppylang.std.quantum import qubit, h, cx

@guppy
def borrow_qubits(
    alice: qubit,
    bob: qubit
) -> None:
    x(alice)
    h(alice)
    cx(alice, bob)
```

Qubits & Collections of Qubits

Guppy

```python
from guppylang.std.quantum import qubit, h, cx, measure
from guppylang.std.builtins import array

@guppy
def get_qubits() -> tuple[qubit, qubit]:
    q0, q1 = qubit(), qubit()
    h(q0)
    cx(q0, q1)
    return (q0, q1)

@guppy
def get_qubit_array() -> array[qubit, 6]:
    qubit_array = array(qubit() for _ in range(6))
    return qubit_array

@guppy.struct
class qubit_struct:
    data_qubits: array[qubit, 16]
    ancilla: array[qubit, 4]
```

TKET

```python
from pytket.circuit import Circuit

def get_teleportation_circ() -> Circuit:
    # Set up quantum and classical registers

    circ = Circuit()
    src = circ.add_q_register("src", 1)
    alice = circ.add_q_register("Alice", 1)

    src_c = circ.add_c_register("src_c", 1)
```

Gate Operations

Guppy

```python
from guppylang.std.quantum import h, cx, measure, x, qubit

@guppy
def q_ops0() -> None:
    alice, bob = qubit(), qubit()
    h(alice)
    cx(alice, bob)
    alice_c: bool = measure(alice)
    if alice_c:
        x(bob)
```

```python
from guppylang.std.quantum_functional import h, cx

@guppy
def q_ops0() -> None:
    alice, bob = qubit(), qubit()
    alice, bob = cx(h(alice), bob)
```

TKET

```python
from pytket.circuit import Circuit

def get_teleportation_circ() -> Circuit:
    # Set up quantum and classical registers
    circ = Circuit()
    bob = circ.add_q_register("bob", 1)
    alice = circ.add_q_register("Alice", 1)
    alice_c = circ.add_c_register("Alice_c", 1)
    # Alice and Bob share a Bell state
    circ.H(alice[0])
    circ.CX(alice[0], bob[0])
    circ.Measure(alice[0], alice_c[0])

    circ.X(
        bob[0],
        condition_bits=[alice_c[0]],
        condition_value=1
    )
    return circ
```

Conditional Branching

Guppy

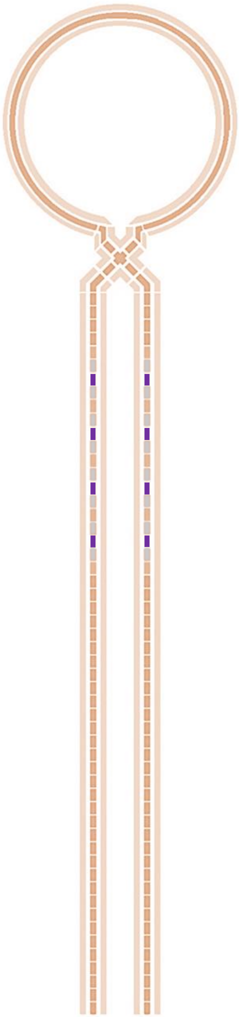
```python
from guppylang.std.quantum import h, cx, measure, x, qubit

@guppy
def q_ops0() -> None:
   alice, bob = qubit(), qubit()
   h(alice)
   cx(alice, bob)
   alice_c: bool = measure(alice)
   if alice_c:
       x(bob)
```

TKET

```python
from pytket.circuit import Circuit

def get_teleportation_circ() -> Circuit:
    # Set up quantum and classical registers
    circ = Circuit()
    bob = circ.add_q_register("bob", 1)
    alice = circ.add_q_register("Alice", 1)
    alice_c = circ.add_c_register("Alice_c", 1)
    # Alice and Bob share a Bell state
    circ.H(alice[0])
    circ.CX(alice[0], bob[0])
    circ.Measure(alice[0], alice_c[0])

    circ.X(
        bob[0],
        condition_bits=[alice_c[0]],
        condition_value=1
    )
    return circ
```

Dynamic Qubit Allocation

Automatic management of hardware qubit resources



Helios Runtime **dynamically** determines if new qubits are *initialized*, or existing qubits are ***reused*** upon program request for qubits.

```
from guppylang import guppy
from guppylang.std.quantum import h, cx, measure, qubit, x, reset, discard

@guppy
def q_ops0() -> None:
    alice, bob = qubit(), qubit()
    h(alice)
    cx(alice, bob)
    alice_c: bool = measure(alice)
    if alice_c:
        x(bob)
    alice1= qubit() # can reuse the same ion
```

Automated Qubit Management

Logical Qubit

A “protected” qubit built from many “virtual qubits”, or logical qubits.

Users specify and control how logical qubits equate to virtual qubits at the library level.

Virtual Qubit

Guppy qubit that can be programmed by the user.

Users can request qubits on-the-fly with Guppy

Physical Qubit

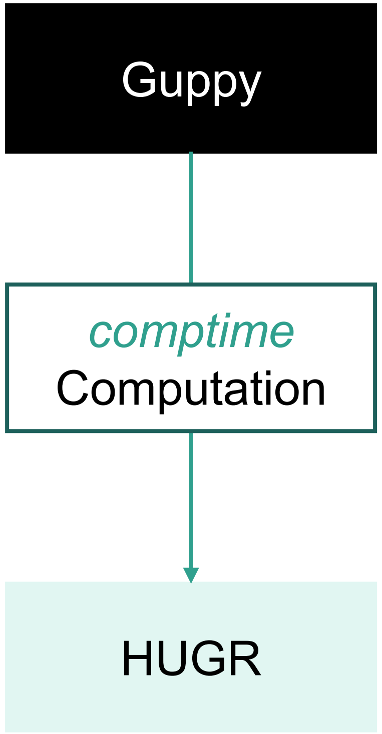
Ion transport and operations

System is responsible for ion allocation.

Classical Computation

Compile Time

Local HUGR Compilation



Compile time computation

```
@guppy
def compute_arctan() -> float:
    return comptime(
        math.atan(2.0)/math.pi
    )
```

Why?

Local classical computation during HUGR compilation

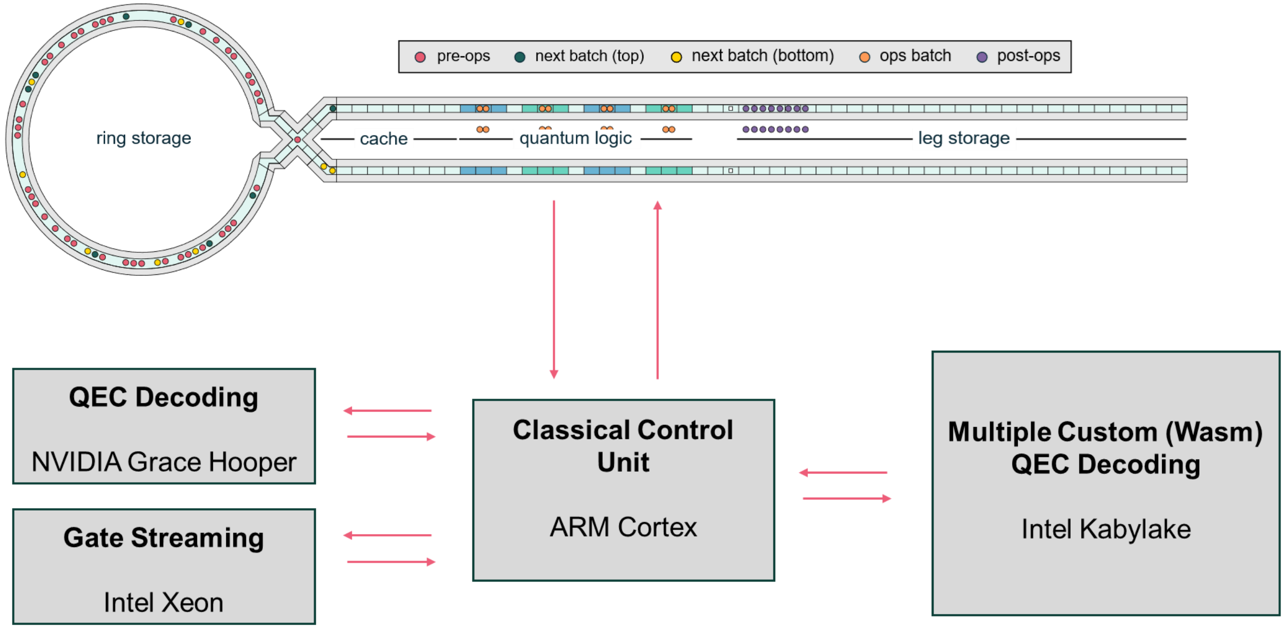
Third-party library usage

Inject Python values

Classical Computation

Runtime Computation

- Arbitrary control flow
- Arbitrary classical Compute



Runtime Computation

```
@guppy
def decode_result(
    syndromes: array[bool, 2] @ owned
) -> Int:
    result = 0
    for s in syndromes:
        result += 2**int(s)

    if result == 1:
        return 0
    if result == 2:
        return 2
    if result == 3:
        return 1
```

qsystem submodule

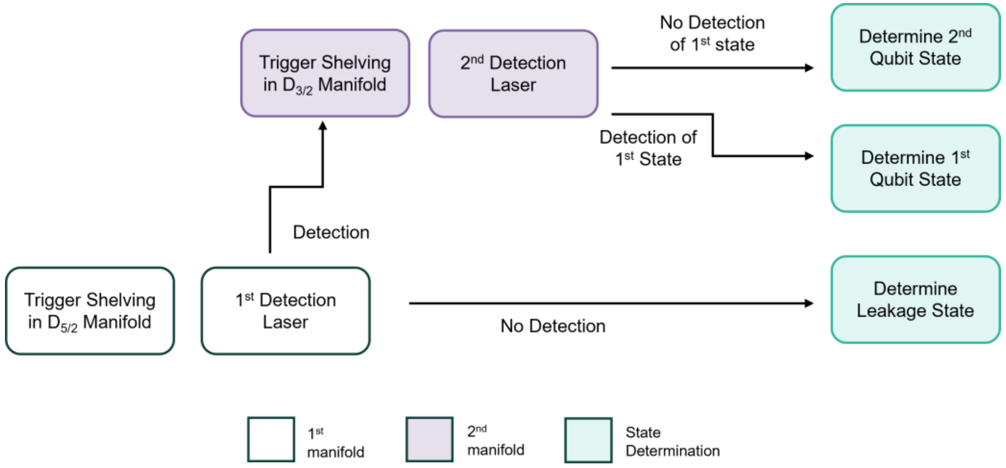
Native Quantinuum operations

Current

- Native Quantum Operations
  - zzphase
  - zzmax
  - phasedx
  - rz
  - measure
  - reset
- RNG: on-chip classical pseudo-random number generator
- get_shot_number: request index of current shot
- measure_and_reset: measure and reset the same ion

```
from guppylang import guppy
from guppylang.std.qsystem import phased_x, zz_phase,
measure
from guppylang.std.qsystem.random import RNG
from guppylang.std.builtins import array
from guppylang.std.quantum import qubit, angle

@guppy
def brickwork(
    qubits: array[qubit, 8],
    rng: RNG
) -> None:
    for i in range(len(qubits)):
        arg0 = rng.random_float()
        arg1 = rng.random_float()
        phased_x(qubits[i], angle(arg0), angle(arg1))
    for i in range(len(qubits) - 1):
        arg2 = rng.random_float()
        zz_phase(qubits[i], qubits[i + 1], angle(arg2))

@guppy
def main() -> None:
    qubits = array(qubit() for _ in range(8))
    rng = RNG(0)
    brickwork(qubits, rng)
    for q in qubits:
        measure(q)
    rng.discard()

guppy.compile_module()
```

Heralded Leakage Measurement



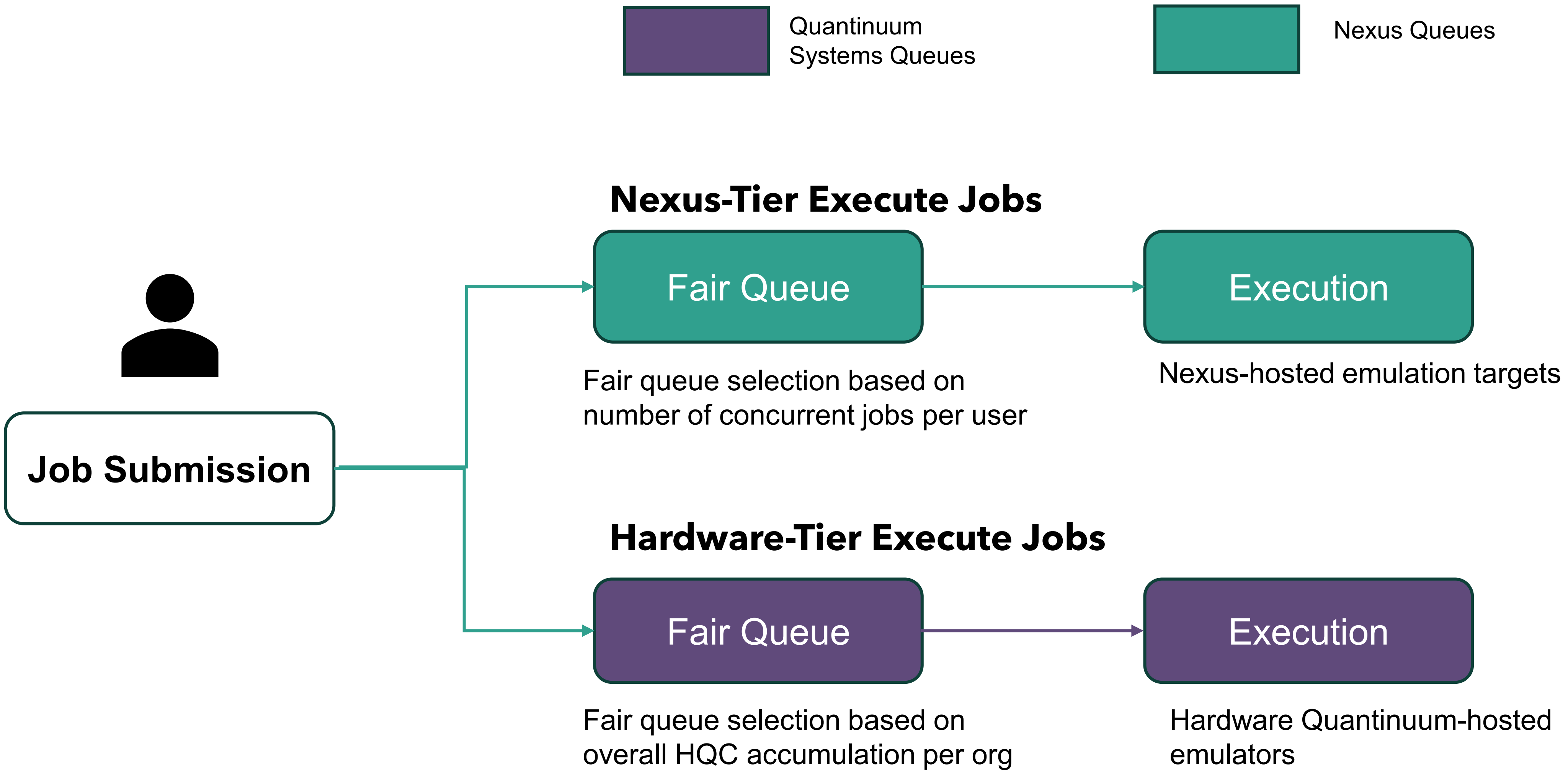
1st Detection Laser	2nd Detection Laser	State Determination
fluorescence	fluorescence	L
fluorescence	no fluorescence	L
no fluorescence	fluorescence	1
no fluorescence	no fluorescence	0

```
from guppylang import guppy
from guppylang.std.builtins import result
from guppylang.std.angles import angle
from guppylang.std.quantum import qubit
from guppylang.std.qsystem import measure_leaked, zz_phase, measure
```

```
@guppy
def main() -> None:
    q0 = qubit()
    q1 = qubit()
    zz_phase(q0, q1, angle(-0.125))
    maybe_leaked = measure_leaked(q1)

    if maybe_leaked.is_leaked():
        q2 = qubit()
        zz_phase(q0, q2, angle(-0.125))
        b1 = measure(q2)
        maybe_leaked.discard()
    else:
        b1 = maybe_leaked.to_result().unwrap()

    result("q0", measure(q0))
    result("q1", b1)
```

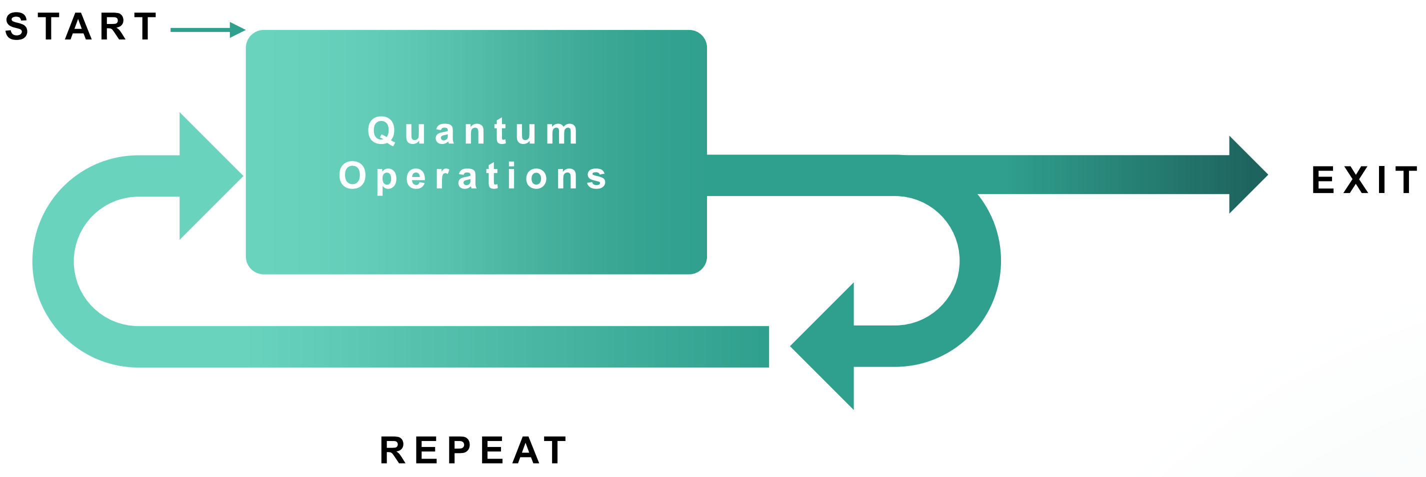
QUANTINUUM

Quantinuum Queues for Execute Job



“Dynamic” Programs

```
@guppy
def repeat_until_success() -> None:
    """
    Repeat-until-success circuit for
Rz(acos(3/5))
    from Nielsen and Chuang, Fig. 4.17.
    """
    q = qubit()

    for i in range(5):

        a, b = h(qubit()), h(qubit())

        toffoli(a, b, q)
        s(q)
        toffoli(a, b, q)

        if not (measure(h(a)) | measure(h(b))):

            result("rus_attempts", i)

    result("q", measure(q))
```



Scenario 1:
- All loops complete successfully
- Condition is never satisfied
- Cost for first shot

Scenario 2:
- loops breaks after 2 iterations
- Condition is satisfied
- Cost for second shot

Max Cost Estimate via Hardware Quantum Credits

HQC = Hardware Quantum Credit

$$HQC = 5 + \frac{N_{1q} + 10N_{2q} + 5N_m}{5000} C$$

- N_{1q}: number of single-qubit gates
- N_{2q}: number of two-qubit gates
- N_m: number of SPAM operations
- C: number of shots

- Required specification before job submission
- Enables management of organization's access
- All operations are costed on an inputted circuits including conditional blocks

Max Cost Estimation

Compile

Upload

Estimate Cost

- Program upload
  - Input: Program
  - Input: User Friendly Name
  - Returns: Program Reference
- Max Cost Estimation
  - Input: Program Reference
  - Input: Number of shots
  - Returns: Max Cost Estimate

```
@guppy
def repeat_until_success() -> None:
    q = qubit()
    for i in range(5):
        a, b = h(qubit()), h(qubit())
        toffoli(a, b, q)
        s(q)
        toffoli(a, b, q)
        if not (measure(h(a)) | measure(h(b))):
            result("rus_attempts", i)
    result("q", measure(q))
```

```
hugr_binary = repeat_until_success.compile()
```

```
ref_hugr = qnx.hugr.upload(
    hugr_binary,
    name=f"repeat-until-success-{unique_suffix}"
)
```

```
prediction = qnx.hugr.cost(
    programs=[ref_hugr],
    n_shots=[10]
)
```

Job Submission

- Device Specification
  - Machine Name
  - **Max Cost for program**
  - **Max Number of Qubits**
- Argument
  - User Friendly Name
  - Backend Configuration
  - Number of shots
  - User Program
- Returns
  - Job Result

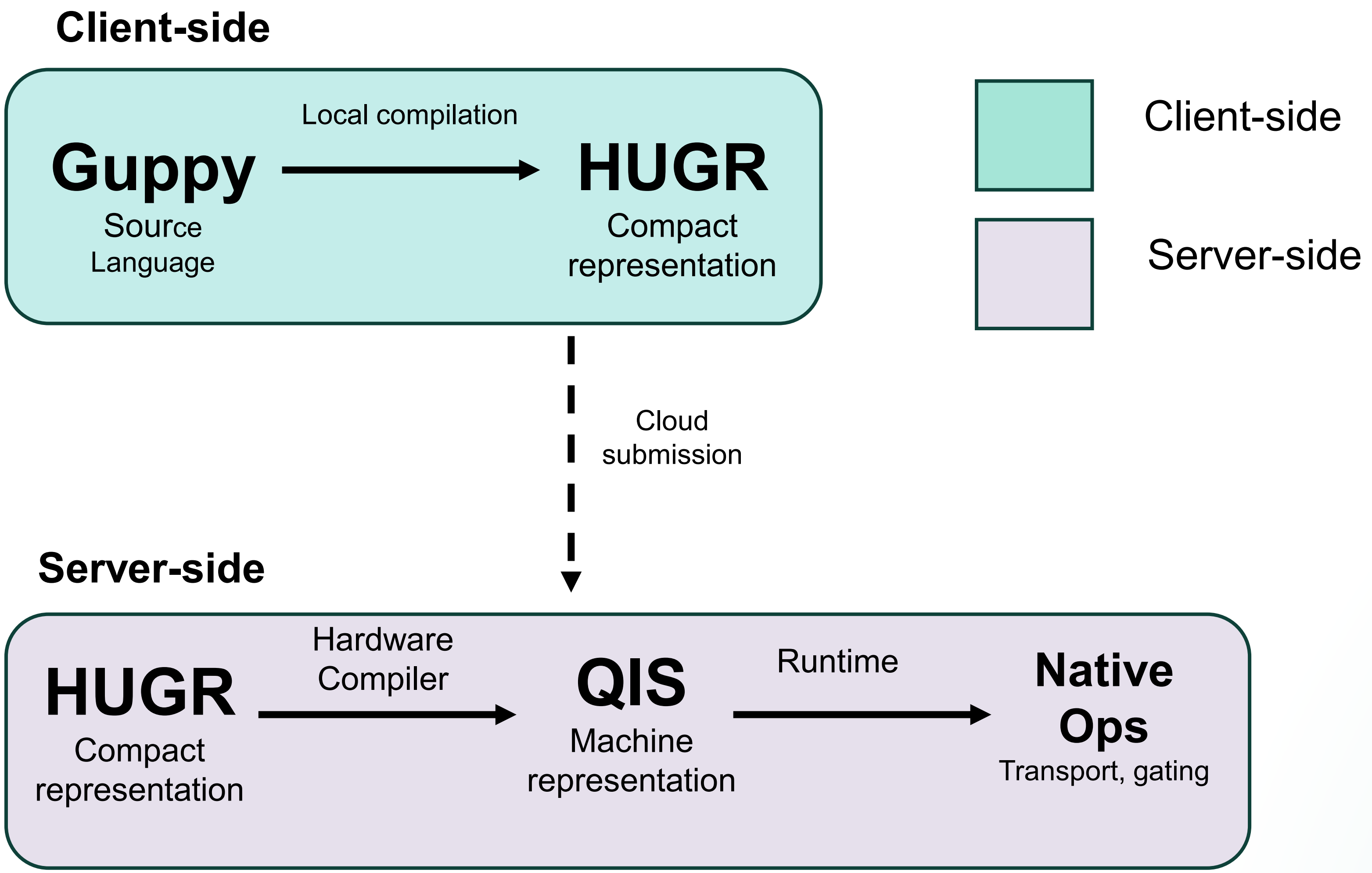
```
config = qnx.models.HeliosConfig(
    system_name="Helios-1E",
    max_cost=prediction,
    emulator_config=qnx.models.HeliosEmulatorConfig(
        n_qubits=3,
        simulator=qnx.models.StatevectorSimulator()
    )
)
```

```
n_shots = 100
ref_execute_job = qnx.start_execute_job(
    programs=[ref_hugr],
    backend_config=config,
    n_shots=[10],
    name=f"execute-job-{name_suffix}"
)
```

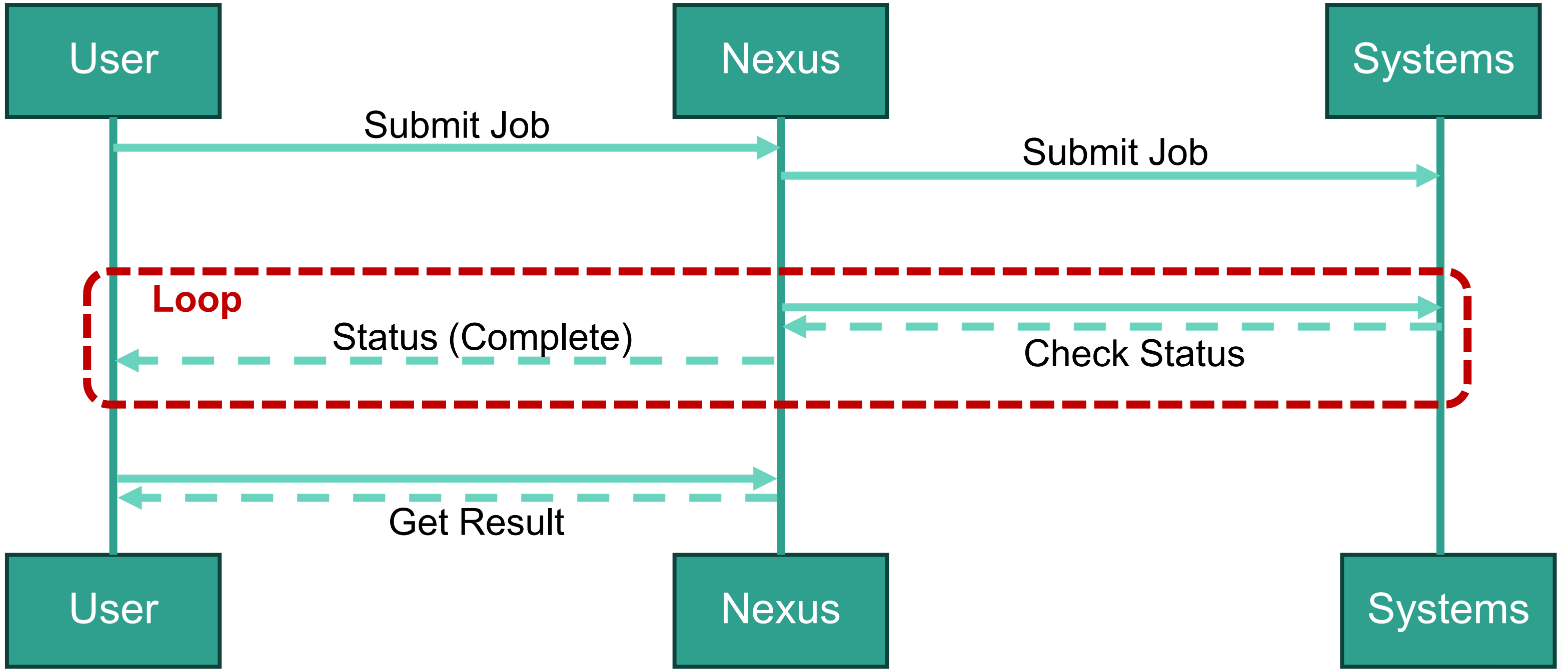
```
qnx.jobs.wait_for(ref_execute_job)
ref_result = qnx.jobs.results(ref_execute_job)[0]
```

```
result = ref_result.download_result()
```

Compilation Lifecycle



Job Lifecycle



Selene

Emulation framework

Machine Emulation with customized noise model and advanced hardware features.

Resource Estimation to estimate program metrics across all conditional branches.

Program Debugging to test successful traversal of conditional branches.

Access

- Availability:
  - Client-side package *selene_sim (current)*
  - *Nexus-hosted Selene instance (upcoming)*

```
pip install selene-sim
```

- Available plugins to optionally specify and consume alternate features.

- Users can develop custom plugins upon extensibility API release.

Selene

Digital twin for Helios

Includes emulation, resource estimation and program debugging capabilities.

Simulation

- Statevector
- Stabilizer
- Matrix Product States
- Coinflip
- Classical Replay
- Quantum Replay

Runtime

- Helios Runtime
- Simple Runtime

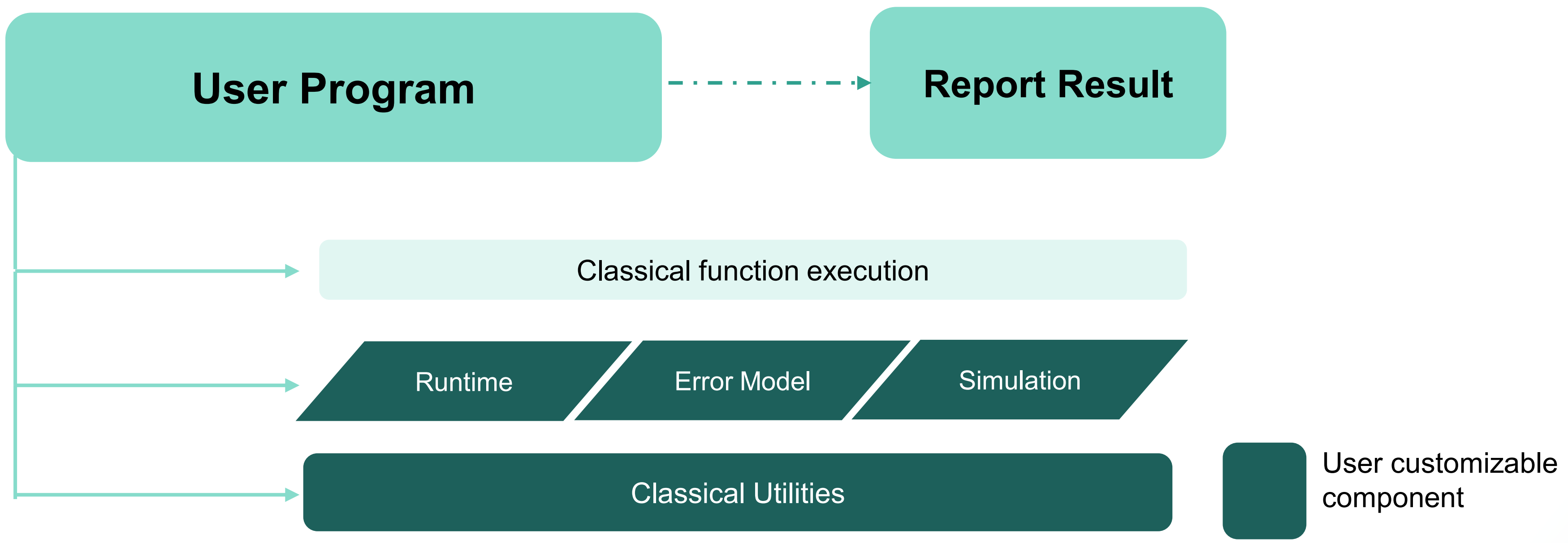
Error Model

- No error model
- Simple Error Model
- Machine-specific Error Model

Nexus-only

Upcoming

Selene Operation



- **Runtime:** Route quantum (and transport) operations from program to simulator.
- **Error Model:** modify routed quantum operations to introduce noise mechanisms.
- **Simulation:** Different modalities available as user-specifiable plugins.

Emulation

Statevector emulation

```python
from selene_sim import build
from selene_sim import Quest
from hugr.qsystem.result import QsysResult


simulator = Quest()

runner = build(hugr)

qsys_result = QsysResult(runner.run_shots(
    simulator,
    n_qubits=4,
))

qsys_result.collated_counts()
```

- QsysResult(results=[QsysShot(entries=[('result', 1), ('result', 1), ('result', 1), ('result', 1)]), QsysShot(entries=[('result', 0), ('result', 0), ('result', 0), ('result', 0)]), QsysShot(entries=[('result', 1), ('result', 1), ('result', 1), ('result', 1)]), …])

```
Counter({(('result', '1111'),): 54, (('result', '0000'),): 46})
```

Emulation

Error Model Specification

```python
from selene_sim import DepolarizingErrorModel

p_1q = 1e-4
p_2q = 2e-3
p_meas = 1e-6
p_init = 1e-6

error_model = DepolarizingErrorModel(
    p_1q=p_1q,
    p_2q=p_2q,
    p_meas=p_meas,
    p_init=p_init
)

runner = build(hugr)
shots = list(runner.run_shots(
    simulator,
    n_qubits=4,
    error_model=error_model
))
```

- Depolarazing error model allows the following values to be assigned:
  - `p_1q`
  - `p_2q`
  - `p_meas`
  - `p_init`

- Added to *`SeleneInstance.run_shots`* using **`error_model`** kwarg.

Debugging

Coinflip

```
from selene_sim import build, CoinFlip

bias: float = 0.0

simulator = CoinFlip(bias=bias)

runner = build(hugr)

shots = list(runner.run_shots(
    simulator,
    n_qubits=4,
))
```

- All measurements return pseudo-random Booleans
- No quantum operations
- Coinflip bias can be defined to replicate "syntax checker" results.
  - bias=0 returns a False (0) for all measurements

Debugging

Classical Replay

```
@guppy
def main() -> None:
    q0: qubit = qubit()
    h(q0)
    c0 = measure(q0)
    result("c0", c0)
    if c0:
        q1: qubit = qubit()
        h(q1)
        c1 = measure(q1)
        result("c1", c1)
```

```
from selene_sim import ClassicalReplay

measurements = [
    [False],
    [True, False]
]

simulator = ClassicalReplay(measurements=measurements)
```

- Classical replay allows branches in users program to be traversed
- Does not perform quantum operations
- Requires input measurements.

Debugging

Classical Replay and Circuit Extraction

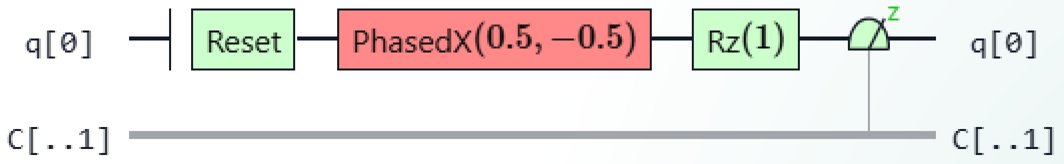
```python
from hugr.qsystem.result import QsysResult
from selene_sim import build
from selene_sim.event_hooks import CircuitExtractor

event_hook = CircuitExtractor()

runner = build(hugr)
shots = QsysResult(runner.run_shots(
    simulator,
    n_qubits=2,
    n_shots=2,
    event_hook=event_hook
))
```

```python
print(shots)
```

```python
event_hook.shots[0].get_user_circuit()
```

- Classical replay allows branches in users program to be traversed.
- Event hooks used to extract circuits.
- Circuit returned for first block prior to conditional branch

```
QsysResult(results=[
        QsysShot(entries=[('c0', 0)])
        QsysShot(entries=[('c0', 1), ('c1', 0)])
])
```

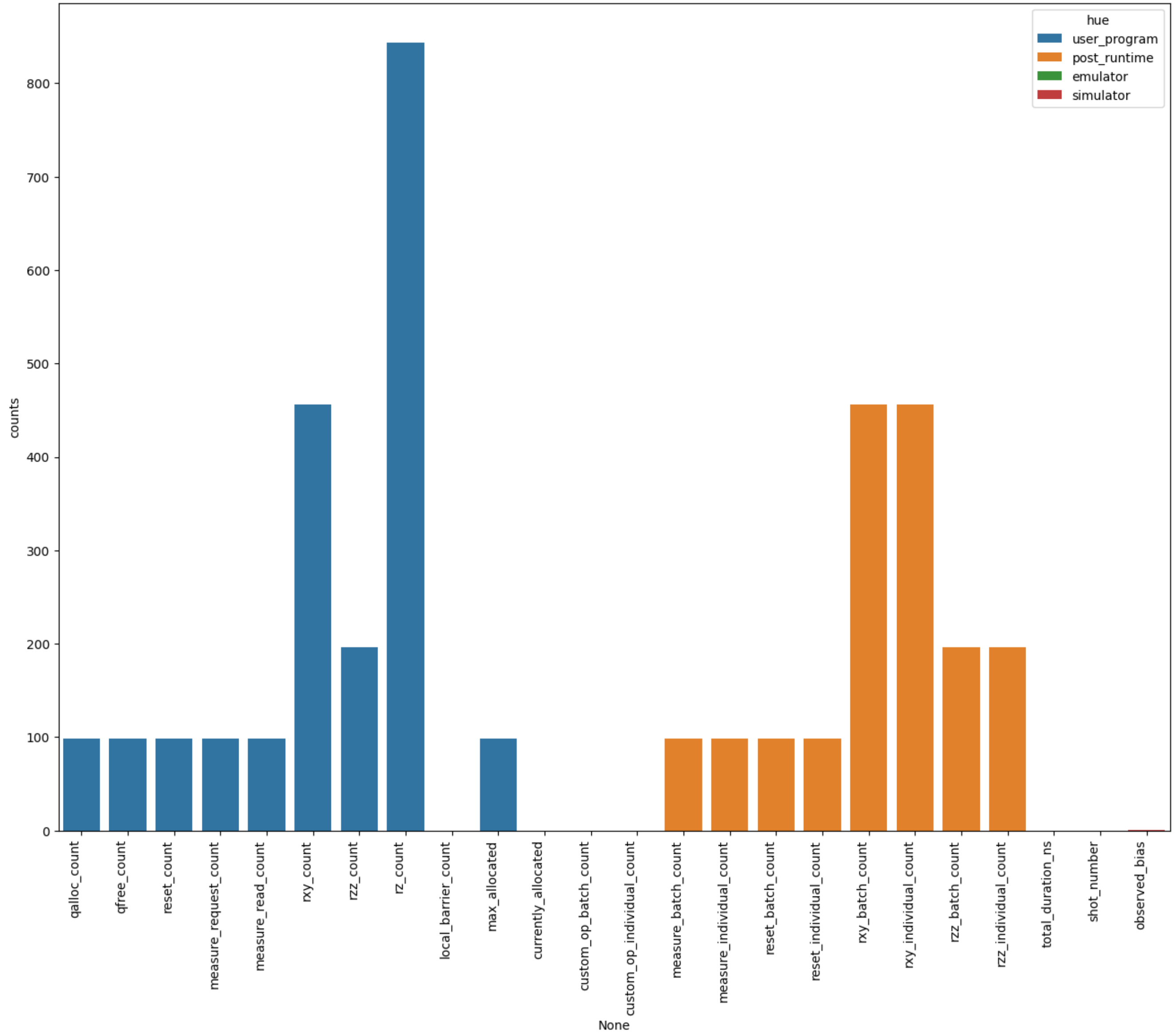


Resource Estimation

Reporting Metrics

```python
from selene_sim.event_hooks import (
    CircuitExtractor,
    MultiEventHook,
    MetricStore
)

event_hook = MultiEventHook(
    event_hooks=[
        CircuitExtractor(),
        MetricStore()
])
```

```python
shots = QsysResult(runner.run_shots(
    simulator,
    n_qubits=2,
    n_shots=2,
    event_hook=event_hook,
))
print(event_hook.event_hooks[1].shots)
```



Summary

- Introducing
  - Guppy, a python-like quantum-classical programming language.
  - Selene, a digital little sister for Helios
  - Helios, a new QCCD ion trap machine with real time capabilities, including dynamic transport.
  - Nexus support for Guppy and HUGR
  - Availability of cloud Selene instances

- Legacy Tools
  - Pytket will be supported for a time
  - Pytket-quantinuum will provide local compilation capability, but submission API is deprecated

Opensource Repositories

github.com/CQCL/guppylang

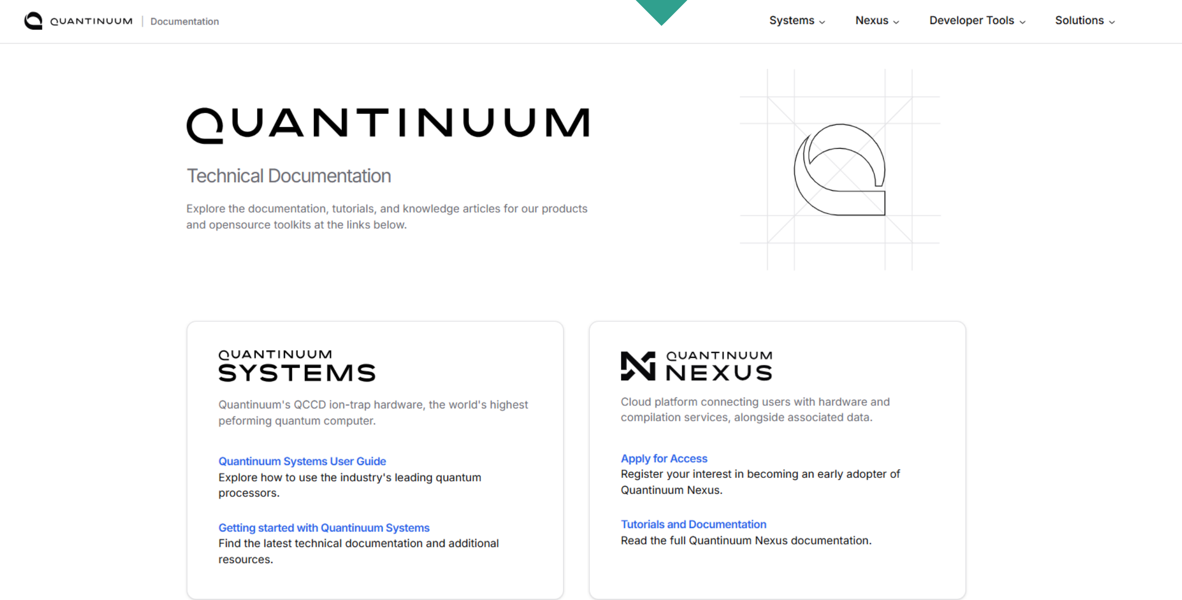
github.com/CQCL/selene

QUANTINUUM

Resources

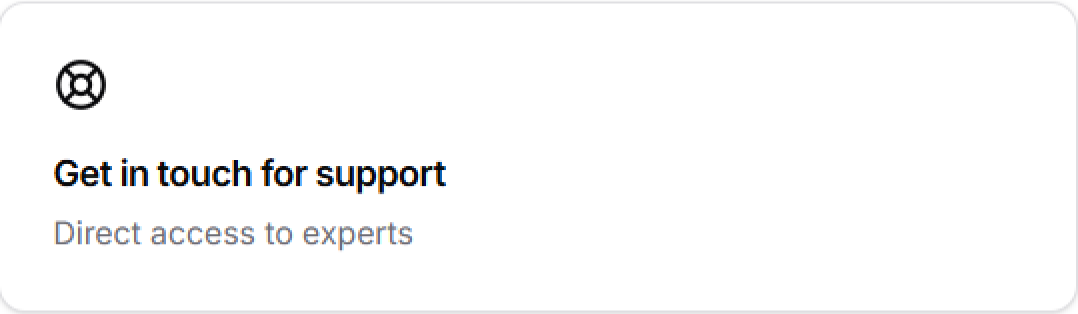
Documentation

docs.quantinuum.com

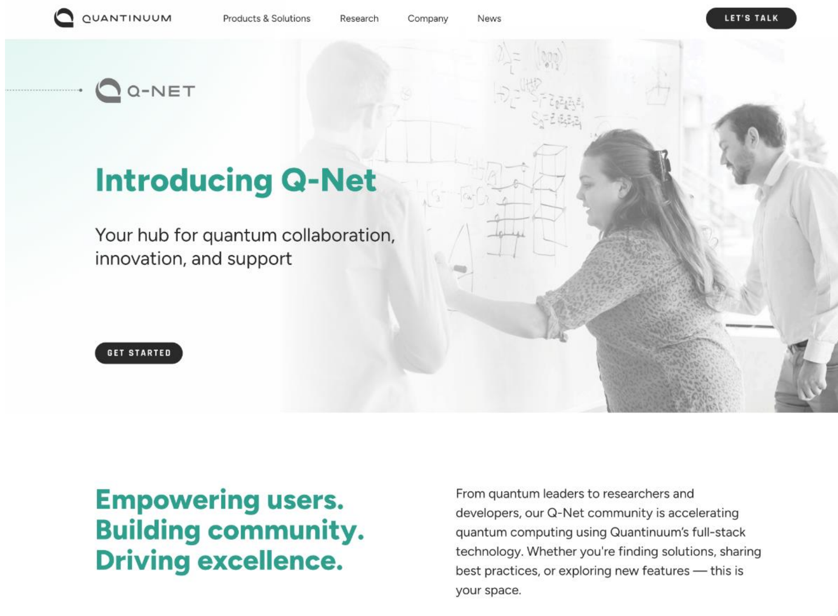


Support

QCsupport@quantinuum.com



Q-NET



- **What is it?** Your hub for quantum collaboration, innovation, and support.
- **What is the goal of Q-Net?**
  - **Enhance Quantinuum user satisfaction** through peer support and shared knowledge.
  - **Drive adoption and retention** by showcasing results and use cases.
  - **Gather actionable feedback** to inform product development.
  - **Build brand loyalty** by recognizing and empowering users.
  - **Create a thriving ecosystem** of advocates, contributors, and learners.
- **Who can join?** New users, power users, community leaders, partners, developers.
- **How can people join?** Visit interest page.
  - Receive the latest news, important announcements, info about the annual meeting, and upcoming webinars.

Join us!

QUANTINUUM