Execution Control Using the Tierkreis Workflow System

Presented by John Children

2025 / 09 / 12

QUANTINUUM

Talk Objectives

- What is Tierkreis?
- What can Tierkreis do for you?
- How do you get started?

QUANTINUUM

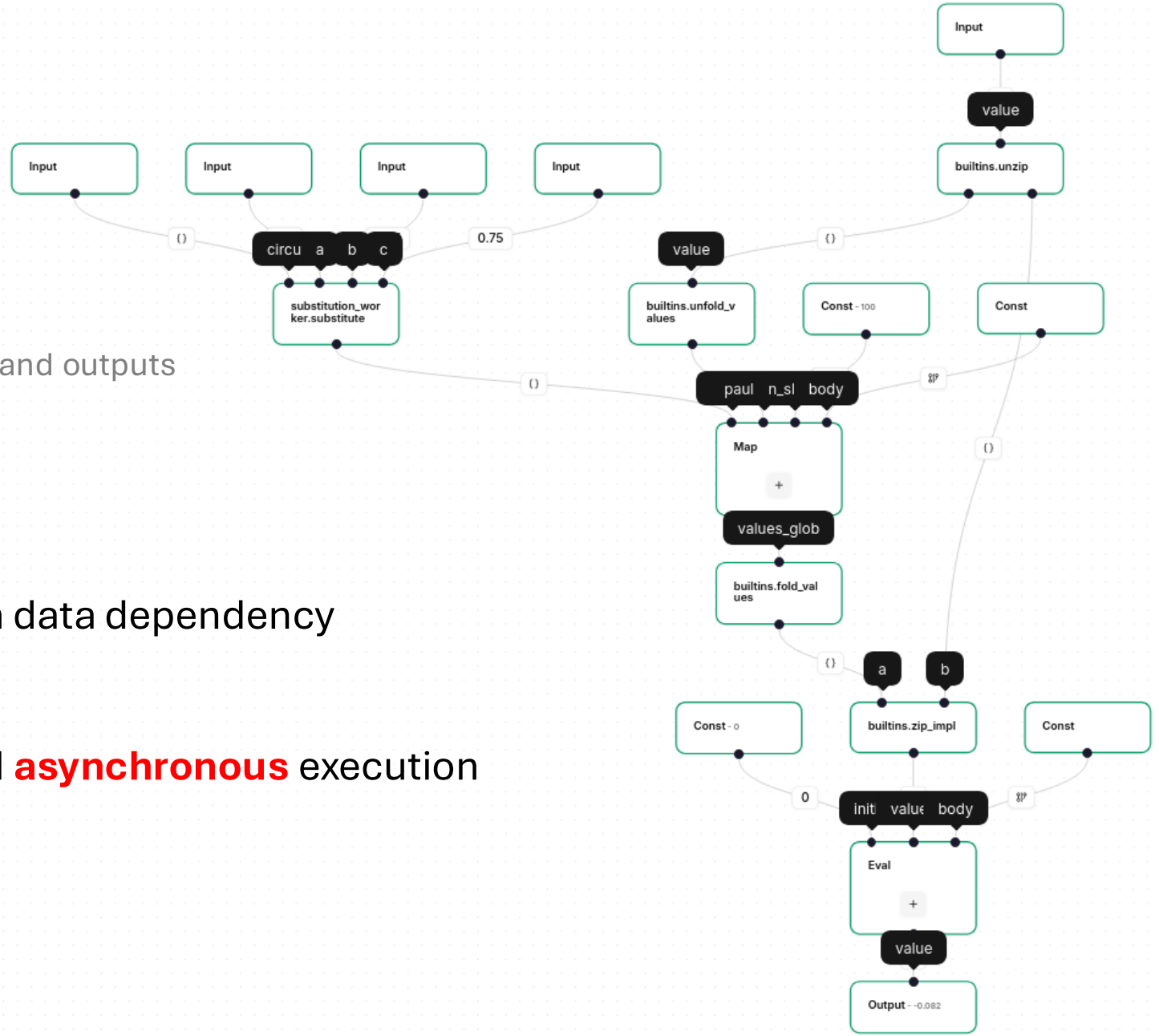
Tierkreis Concepts

Tierkreis Summary

- Workflow orchestration for hybrid workflows
- Asynchronous dataflow model
- Distributed computation
- Concurrent execution
- Adapted for HPC (Fugaku)
  - Focus on minimising HPC idle time

Program model

- Nodes are **computations**
  stateless/pure functions, many inputs and outputs
- Edges carry **data**
  strong static types + data interface
- **Partial order** of execution from data dependency
  strong static types + data interface
- Allows **parallel** scheduling and **asynchronous** execution



Workers

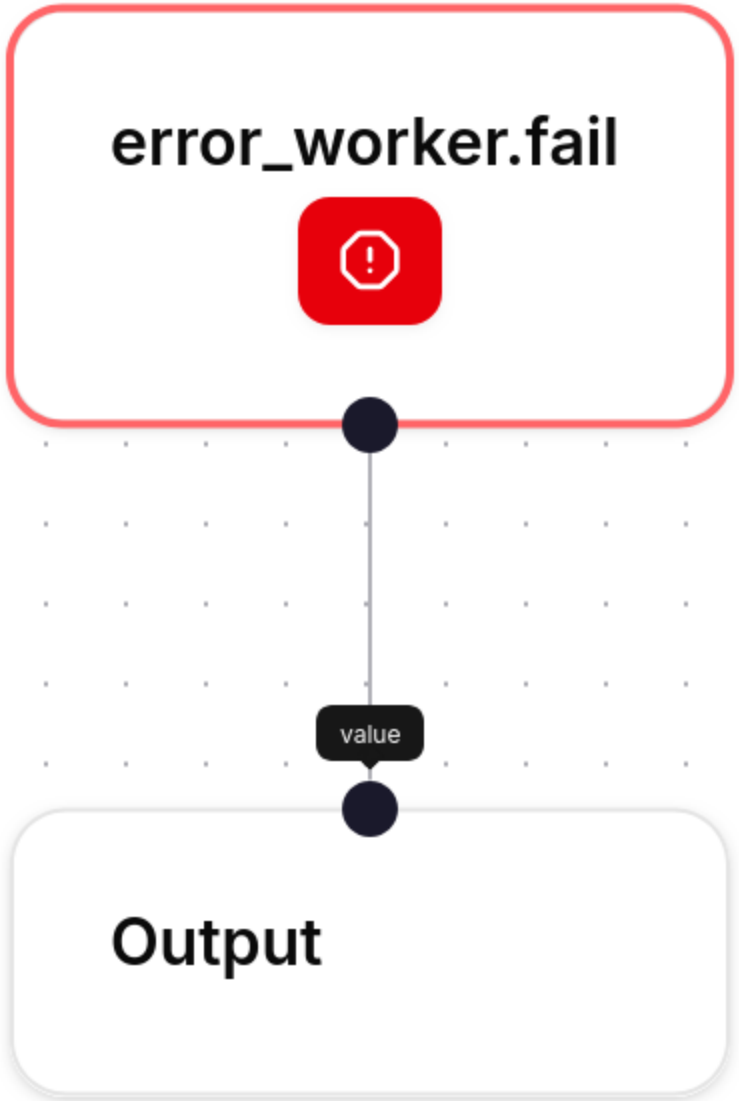
- Implement any desired functionality
  Can be specialised to their host e.g. to exploit GPU
- Can be invoked in a variety of contexts
  Single workers can offer many capabilities
- Can maintain own queue / scheduler
  Or storage, or database, or compiler, or ….
- Easy to wrap existing code as workers
  For python code, this is a one-line change

Checkpointing

- Intermediate values between computations are stored to disk
- Intermediate values can be inspected for debugging
- Workflows can be interrupted and restored from the checkpoints

Visualization

- Built-in web server for visualizing workflows
- Inspect workflow graphs before running
- Debug errors and examine node logs



Configurable Execution Contexts

- Execution can be configured on a per worker basis
- Python workers, shell scripts or HPC jobs can be composed into a single workflow
- Not an exhaustive list, many more things are possible!

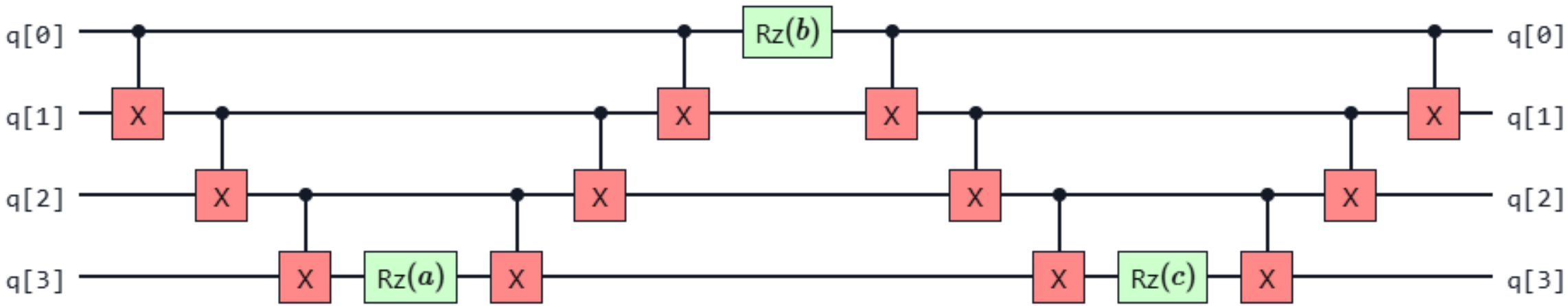
QUANTINUUM

Example Problem

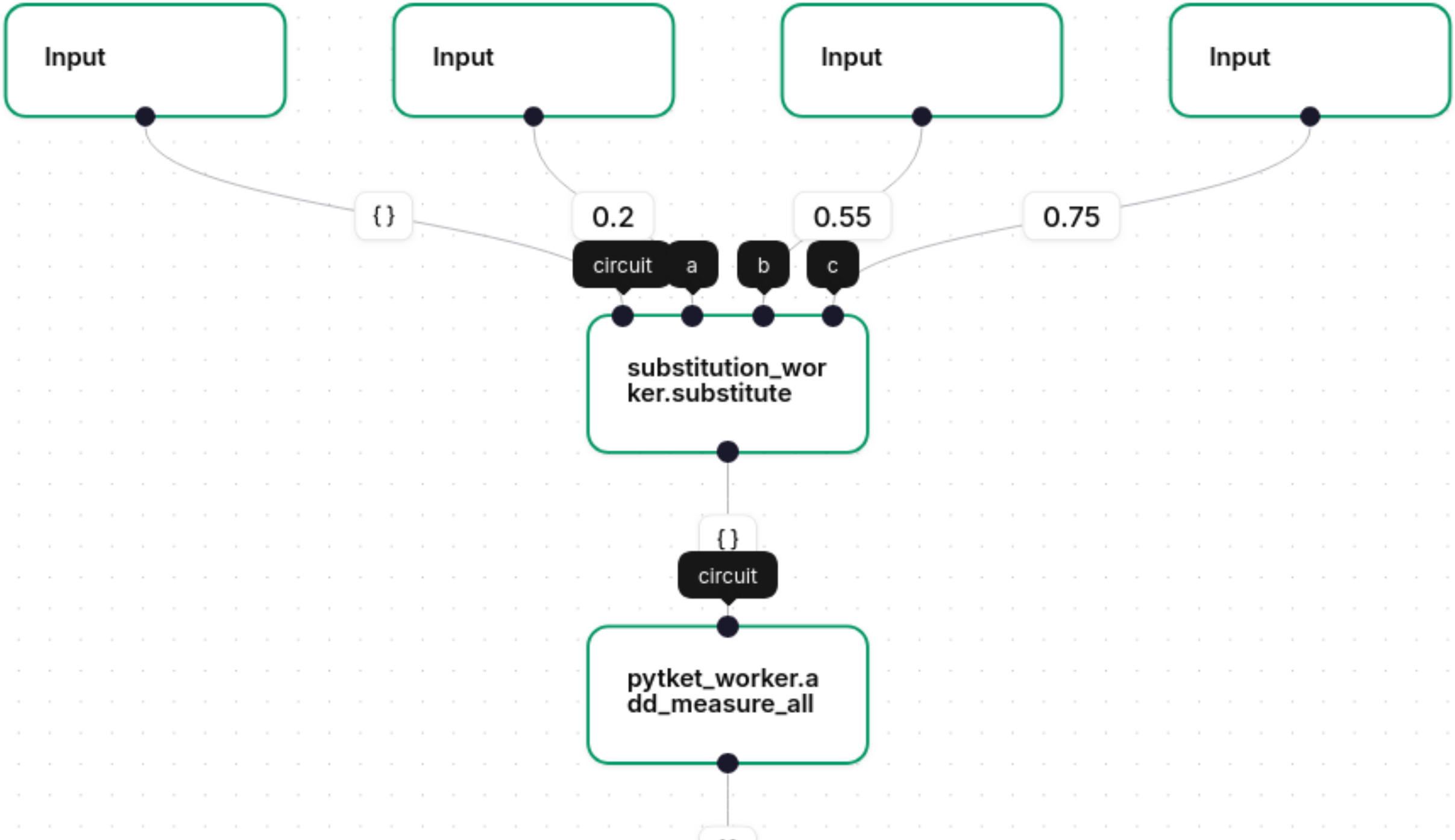
Inputs:
a,b,c -- floating point parameters
ansatz -- parameterised circuit

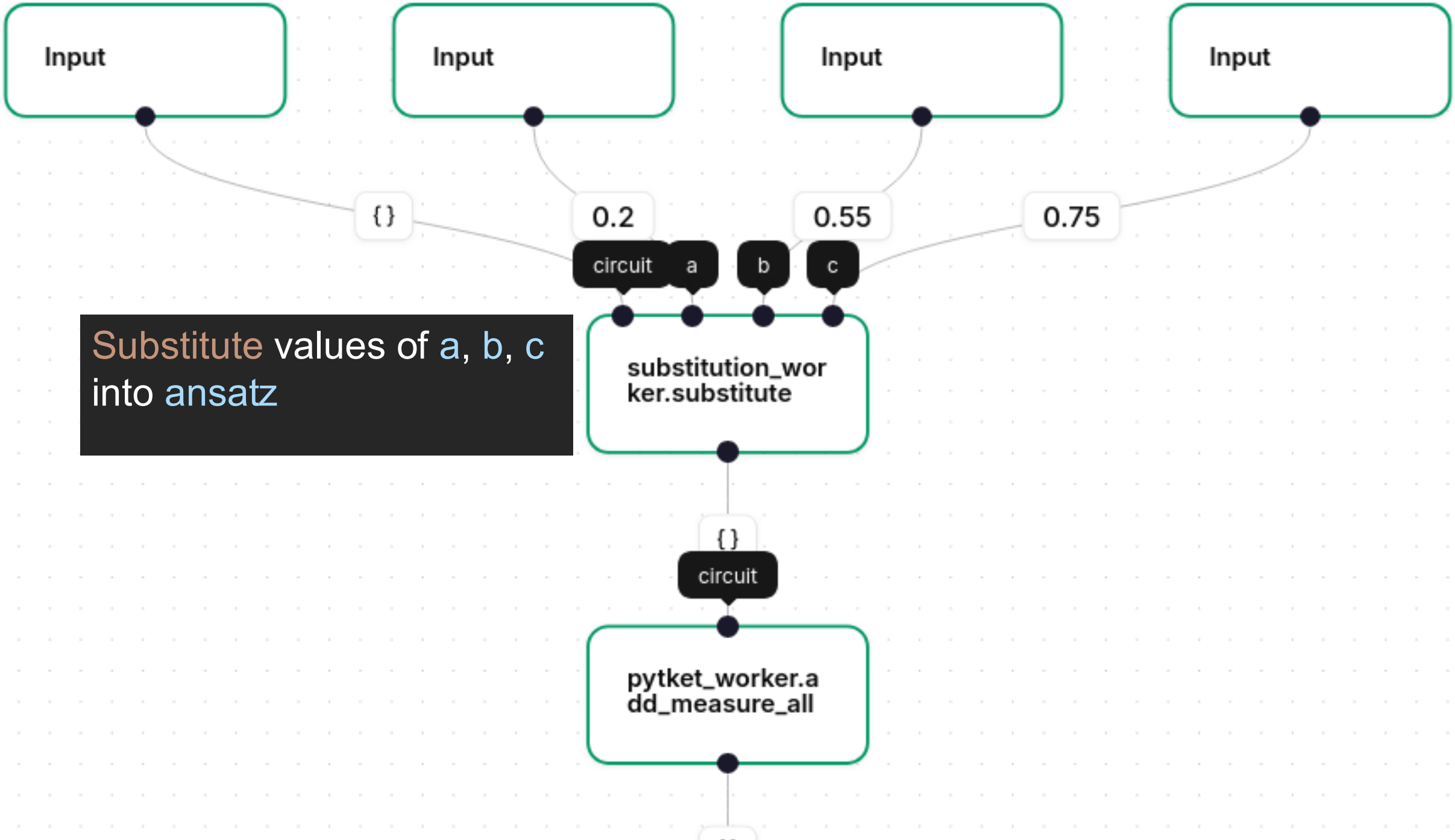
1. Substitute values of a, b, c into ansatz
2. Append measurements to all qubits of ansatz
3. Compile and optimise ansatz circuit
4. Run circuit 100 times
5. Covert counts into expectation

Output:
expectation -- average of ZZZZ observable

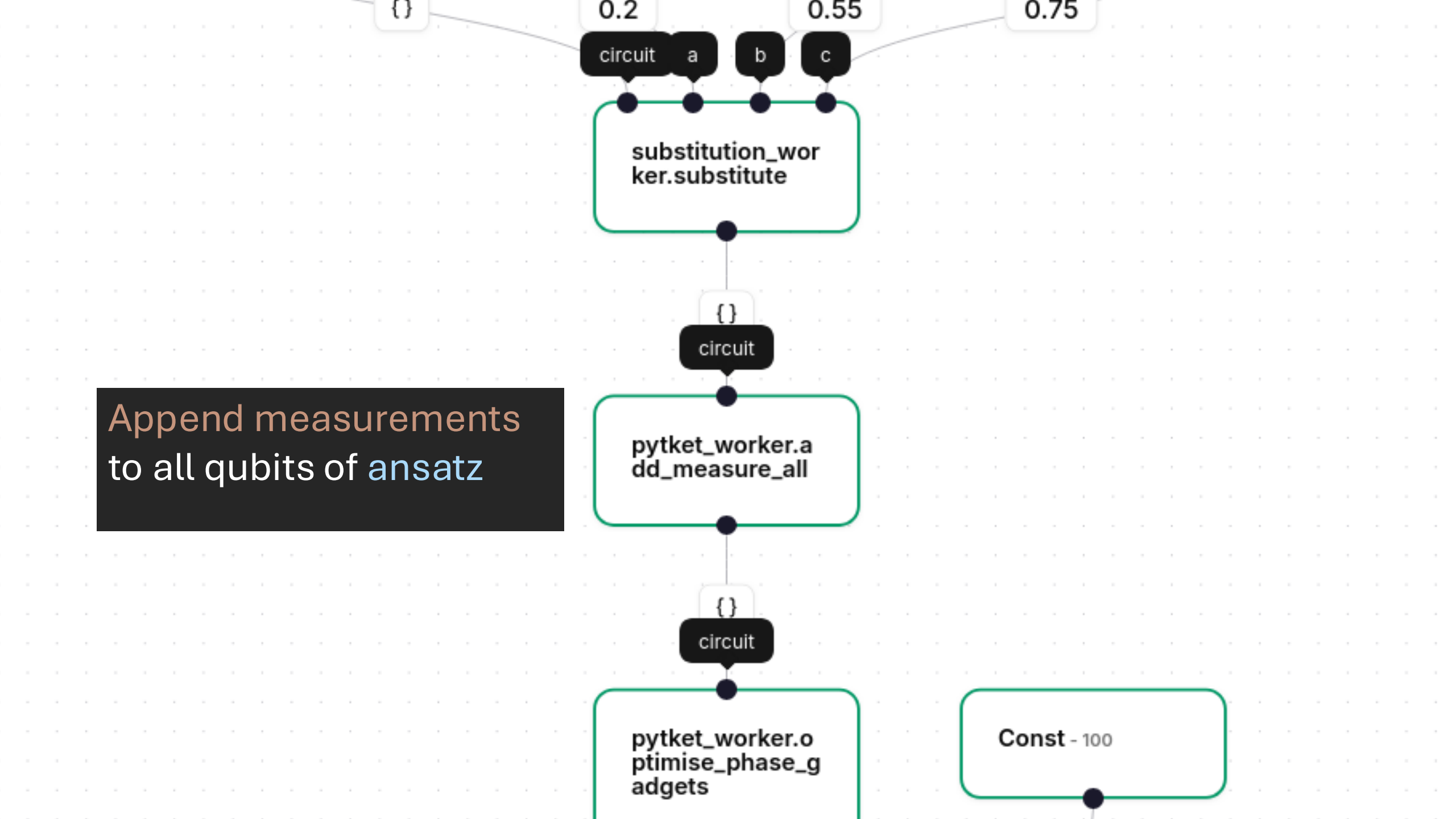


Parametrized Ansatz Circuit


Input
Input
Input
Input
{ }
0.2
0.55
0.75
circuit
a
b
c
substitution_worker.substitute
{ }
circuit
pytket_worker.add_measure_all

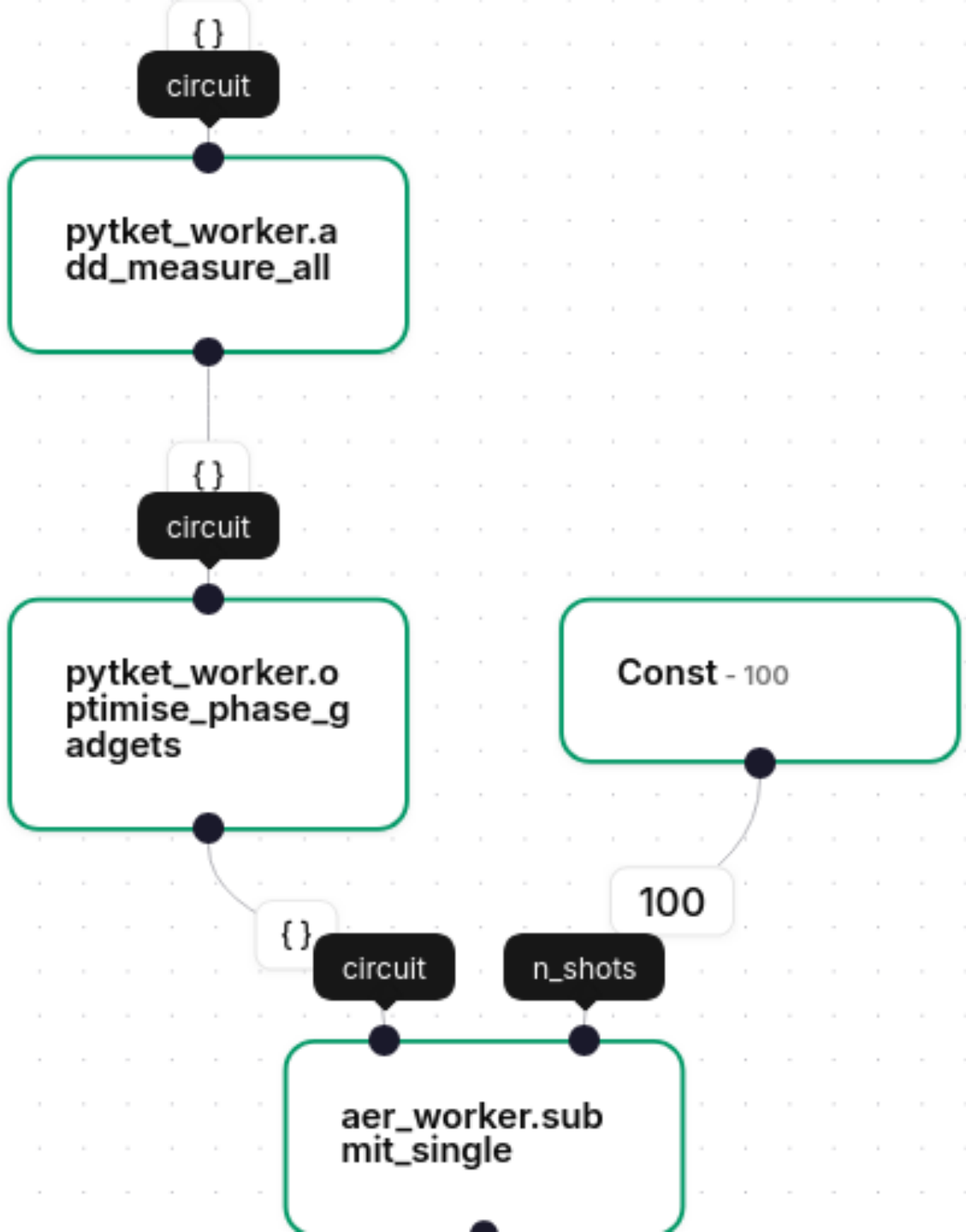


Substitute values of a, b, c into ansatz

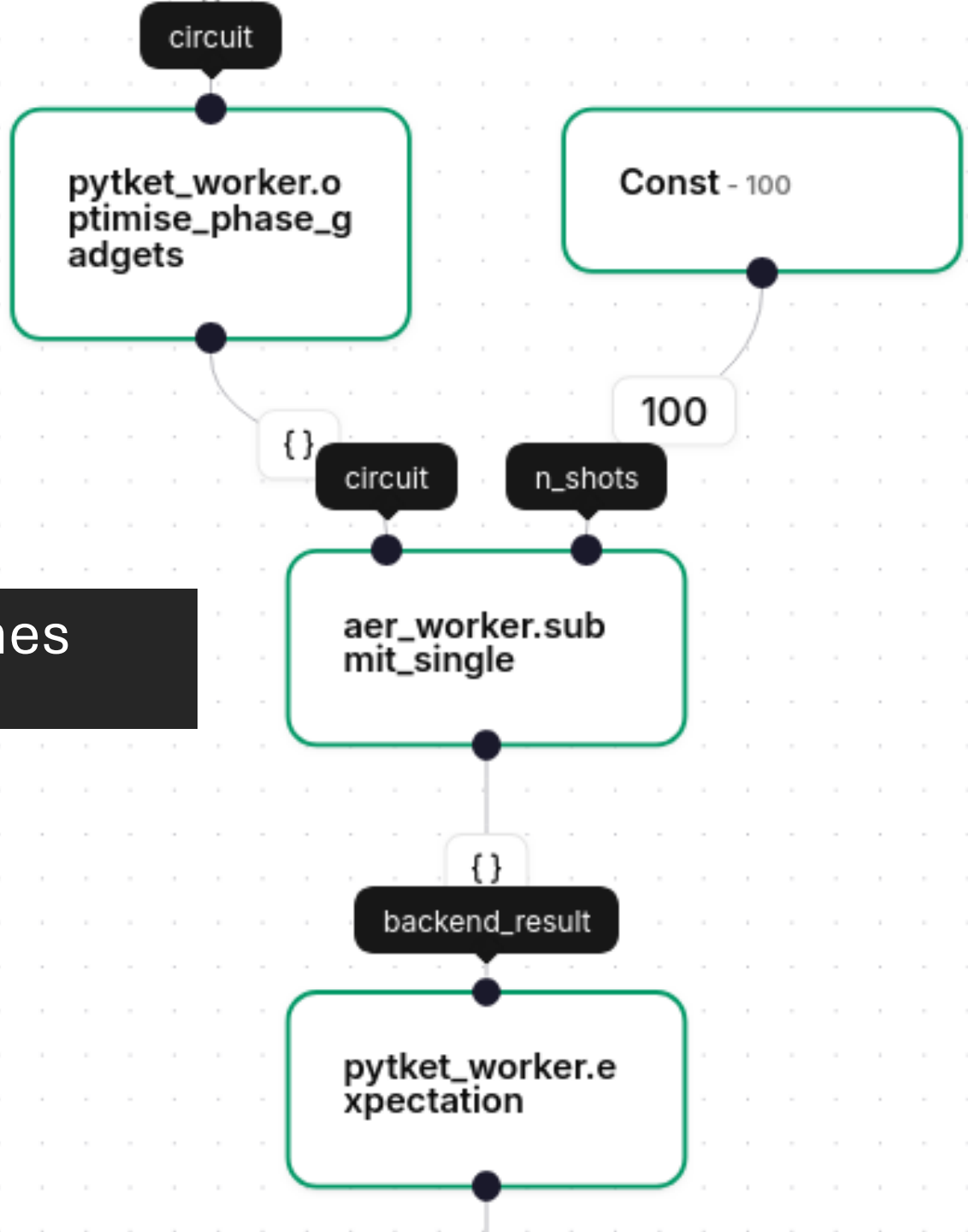


Append measurements
to all qubits of ansatz

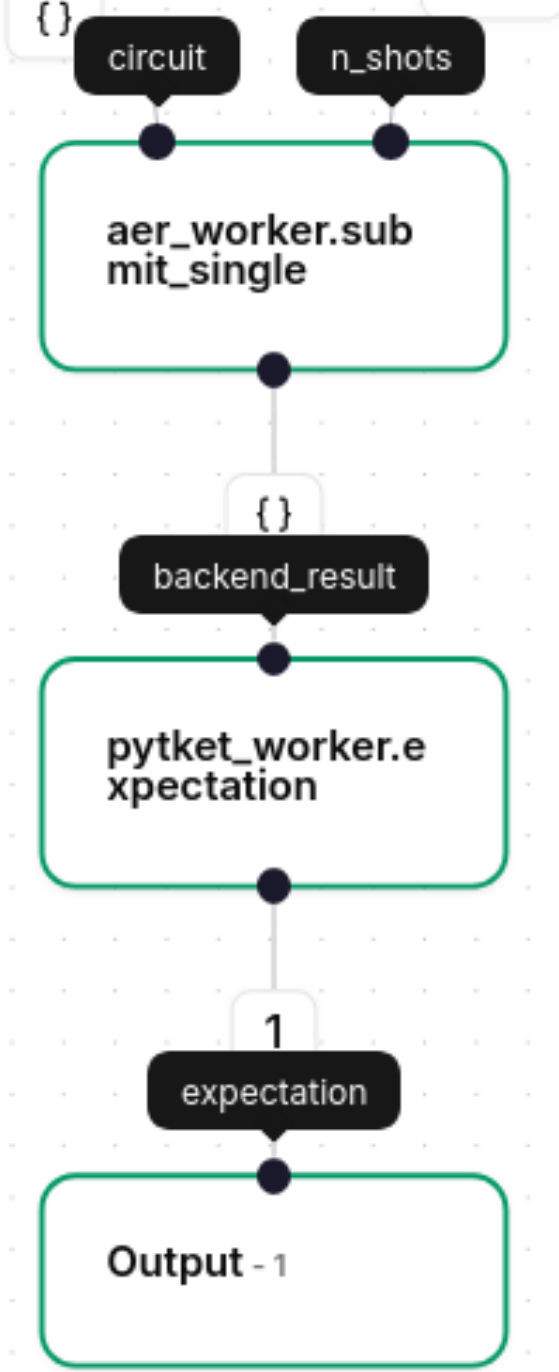
Compile and optimise ansatz circuit

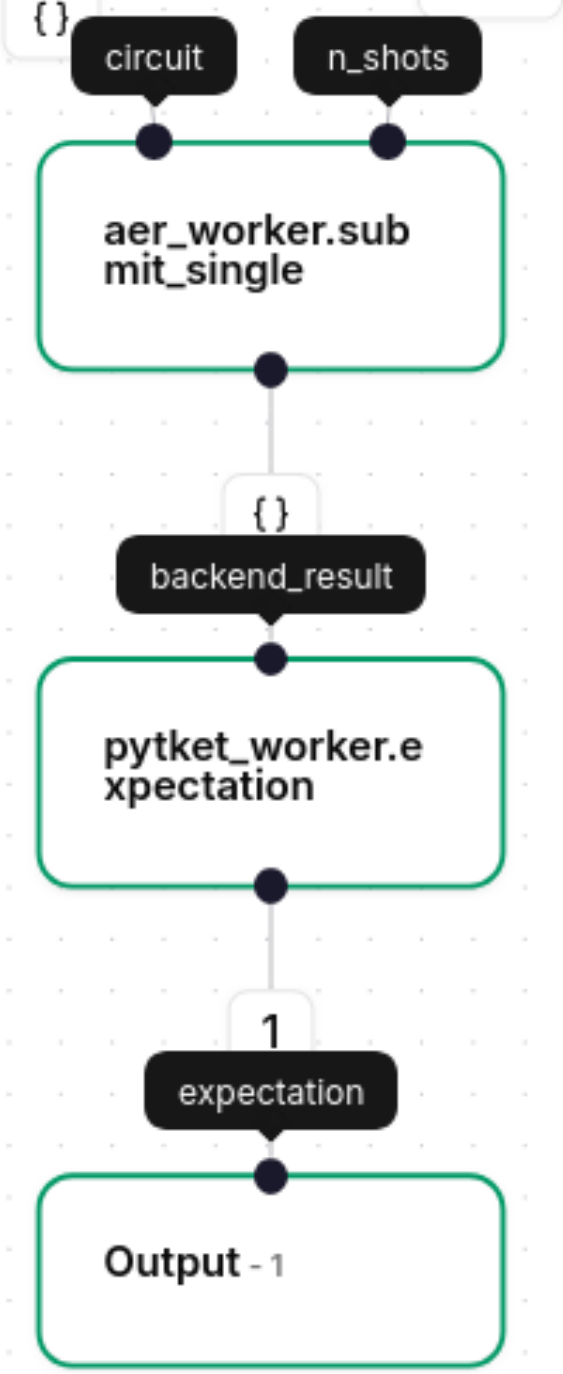


Run circuit 100 times

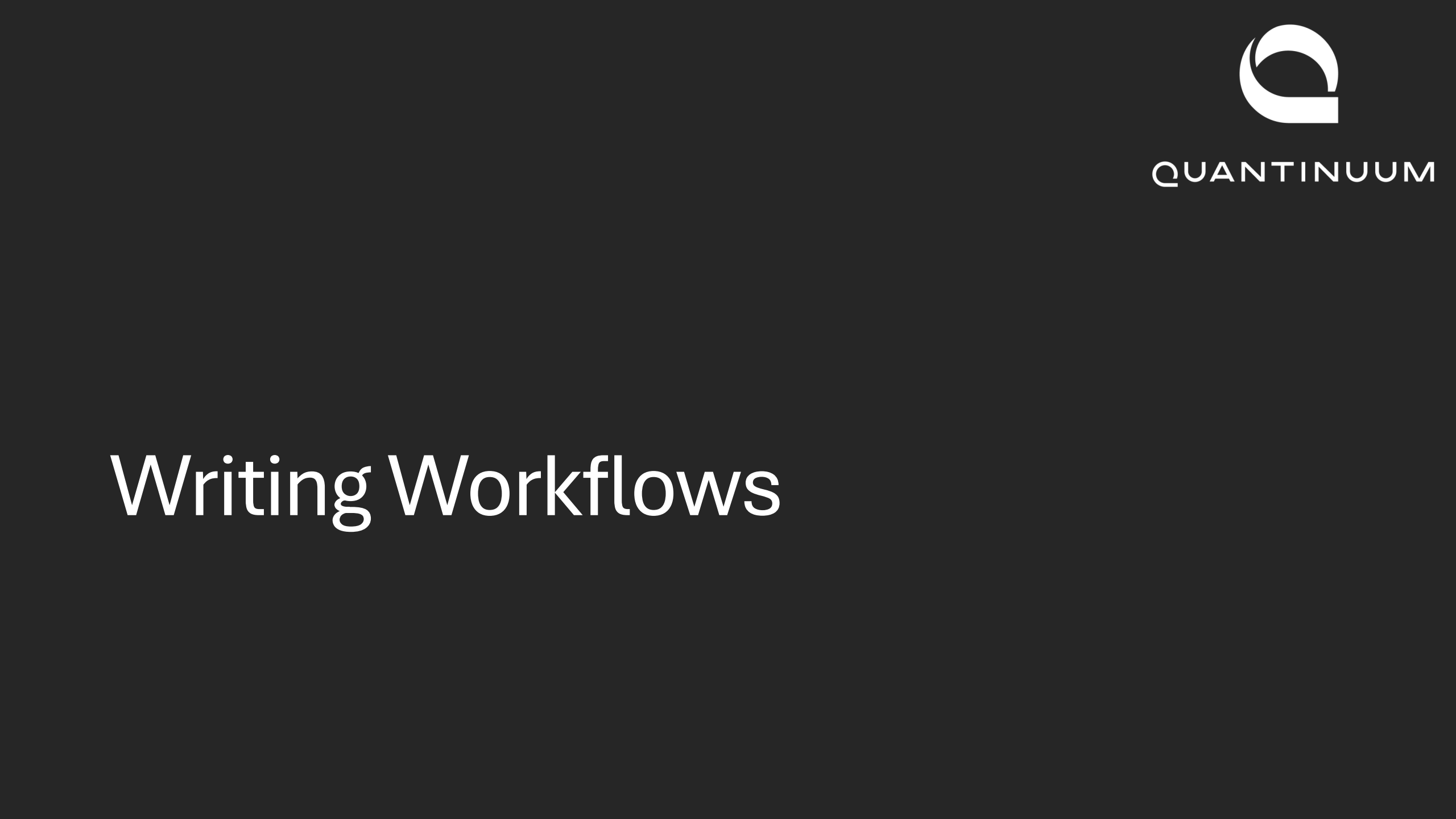


Covert counts into expectation





expectation -- average of ZZZZ observable



Writing Workflows

```python
def main() -> None:
    """Configure our workflow execution and run it to completion."""
    ansatz = build_ansatz()

     … # Some Boilerplate

    run_graph(
        storage,
        multi_executor,
        symbolic_execution(),
        {
            "ansatz": ansatz,
            "a": 0.2,
            "b": 0.55,
            "c": 0.75,
        },
        polling_interval_seconds=0.1,
    )
```

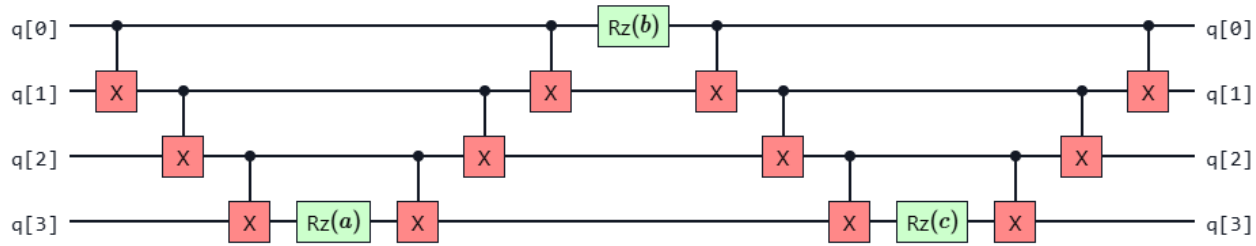
```python
def main() -> None:
    """Configure our workflow execution and run it to completion."""
    ansatz = build_ansatz()

     … # Some Boilerplate

    run_graph(
        storage,
        multi_executor,
        symbolic_execution(),
        {
            "ansatz": ansatz,
            "a": 0.2,
            "b": 0.55,
            "c": 0.75,
        },
        polling_interval_seconds=0.1,
    )
```

```python
def main() -> None:
    """Configure our workflow execution and run it to completion."""
    ansatz = build_ansatz()

     … # Some Boilerplate

    run_graph(
        storage,
        multi_executor,
        symbolic_execution(),
        {
            "ansatz": ansatz,
            "a": 0.2,
            "b": 0.55,
            "c": 0.75,
        },
        polling_interval_seconds=0.1,
    )
```

```python
def main() -> None:
    """Configure our workflow execution and run it to completion."""
    ansatz = build_ansatz()

     … # Some Boilerplate

    run_graph(
        storage,
        multi_executor,
        symbolic_execution(),
        {
            "ansatz": ansatz,
            "a": 0.2,
            "b": 0.55,
            "c": 0.75,
        },
        polling_interval_seconds=0.1,
    )
```

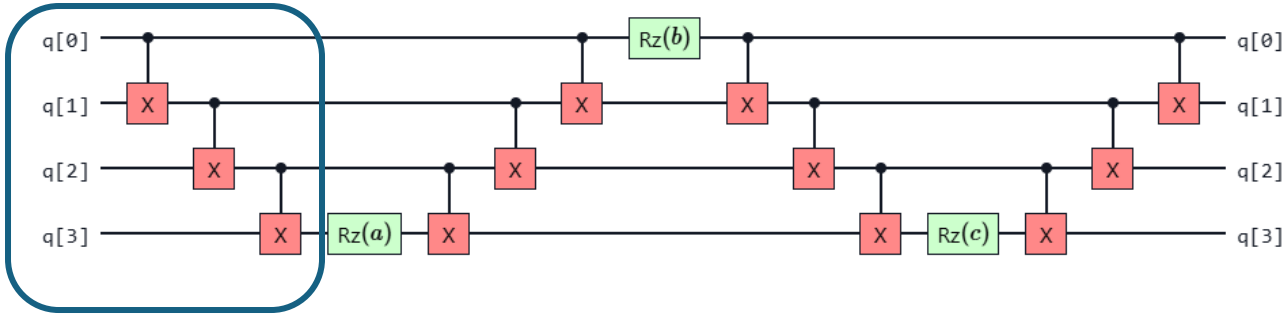
```
a = fresh_symbol("a")
b = fresh_symbol("b")
c = fresh_symbol("c")

circ = Circuit(4)
circ.CX(0, 1).CX(1, 2).CX(2, 3)

circ.Rz(a, 3)
circ.CX(2, 3).CX(1, 2).CX(0, 1)

circ.Rz(b, 0)
circ.CX(0, 1).CX(1, 2).CX(2, 3)

circ.Rz(c, 3)
circ.CX(2, 3).CX(1, 2).CX(0, 1)
return circ
```

Building the Ansatz with pytket



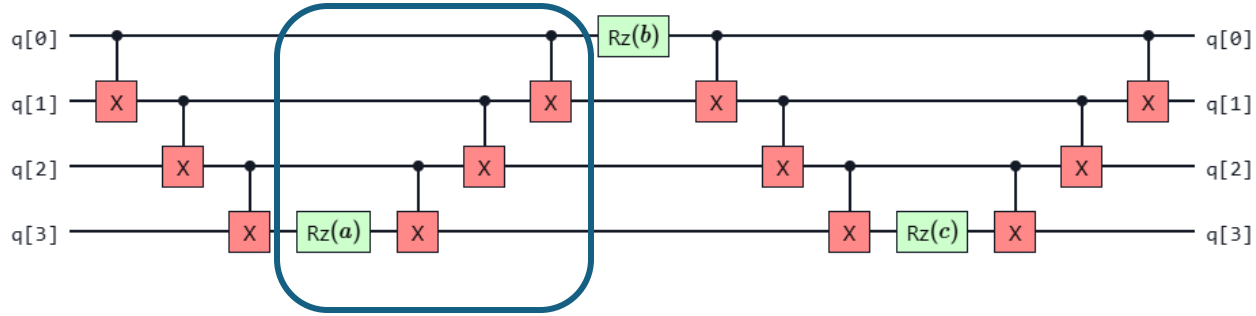
```
a = fresh_symbol("a")
b = fresh_symbol("b")
c = fresh_symbol("c")

circ = Circuit(4)
circ.CX(0, 1).CX(1, 2).CX(2, 3)

circ.Rz(a, 3)
circ.CX(2, 3).CX(1, 2).CX(0, 1)

circ.Rz(b, 0)
circ.CX(0, 1).CX(1, 2).CX(2, 3)

circ.Rz(c, 3)
circ.CX(2, 3).CX(1, 2).CX(0, 1)
return circ
```

Building the Ansatz with pytket



```
a = fresh_symbol("a")
b = fresh_symbol("b")
c = fresh_symbol("c")

circ = Circuit(4)
circ.CX(0, 1).CX(1, 2).CX(2, 3)

circ.Rz(a, 3)
circ.CX(2, 3).CX(1, 2).CX(0, 1)

circ.Rz(b, 0)
circ.CX(0, 1).CX(1, 2).CX(2, 3)

circ.Rz(c, 3)
circ.CX(2, 3).CX(1, 2).CX(0, 1)
return circ
```

Building the Ansatz with pytket



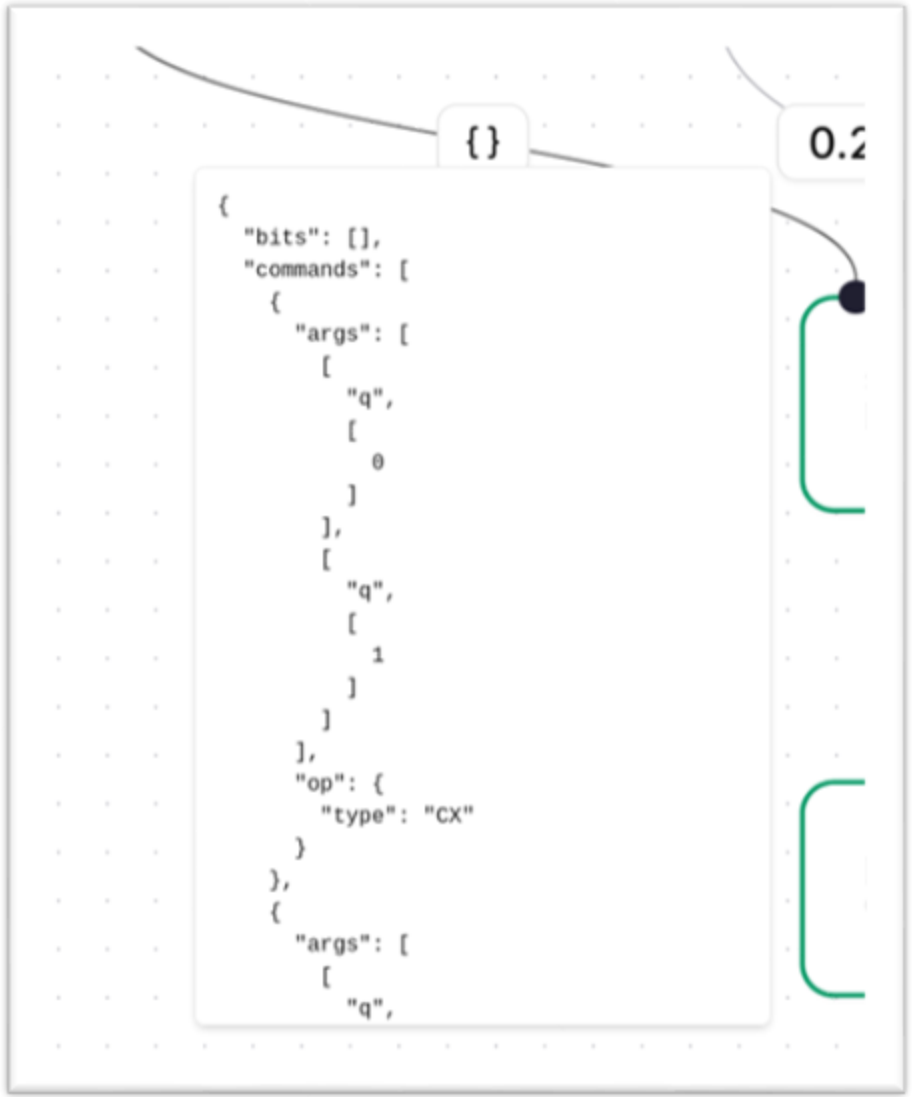
```python
def main() -> None:
    """Configure our workflow execution and run it to completion."""
    ansatz = build_ansatz()

     … # Some Boilerplate

    run_graph(
        storage,
        multi_executor,
        symbolic_execution(),
        {
            "ansatz": ansatz,
            "a": 0.2,
            "b": 0.55,
            "c": 0.75,
        },
        polling_interval_seconds=0.1,
    )
```

```
g = GraphBuilder(
    SymbolicCircuitsInputs,
    SymbolicCircuitsOutputs
)
```

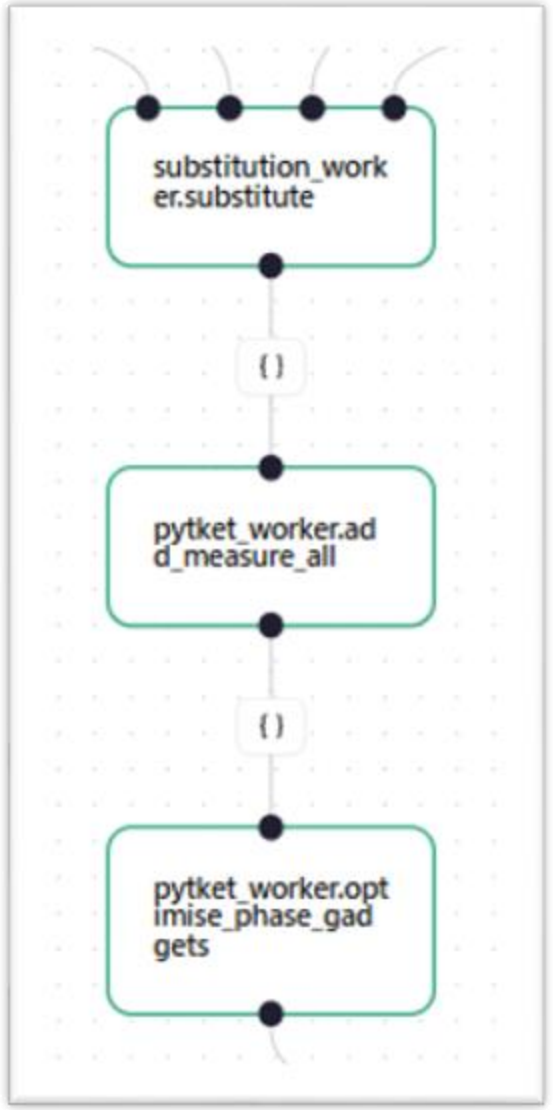
```
class SymbolicCircuitsInputs(NamedTuple):
   a: TKR[float]
   b: TKR[float]
   c: TKR[float]
   ansatz: TKR[OpaqueType["pytket._tket.circuit.Circuit"]]
```

```
class SymbolicCircuitsOutputs(NamedTuple):
        expectation: TKR[float]
```

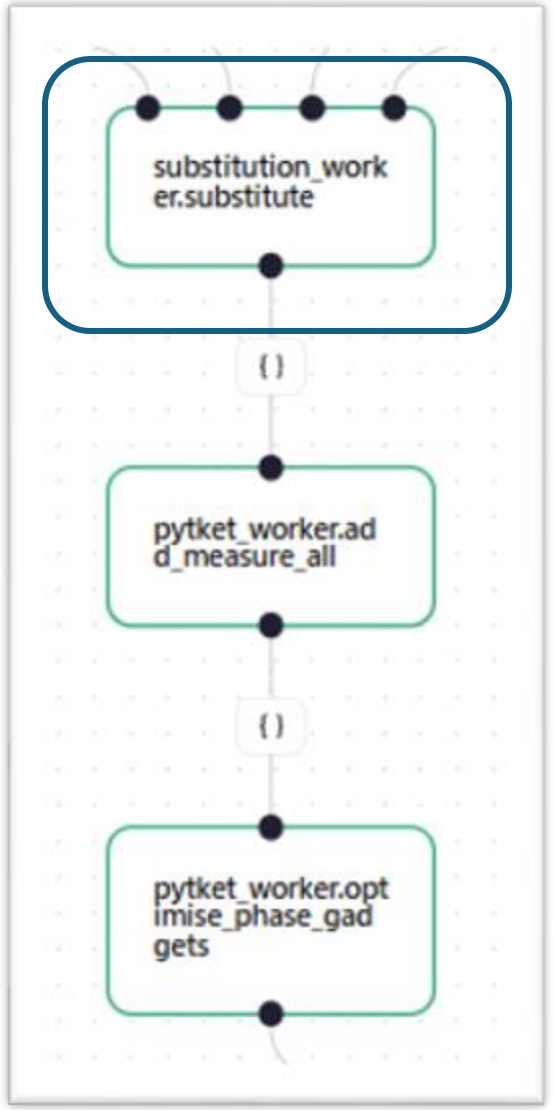
```
a = g.inputs.a

b = g.inputs.b

c = g.inputs.c

ansatz = g.inputs.ansatz

n_shots = g.const(100)
```



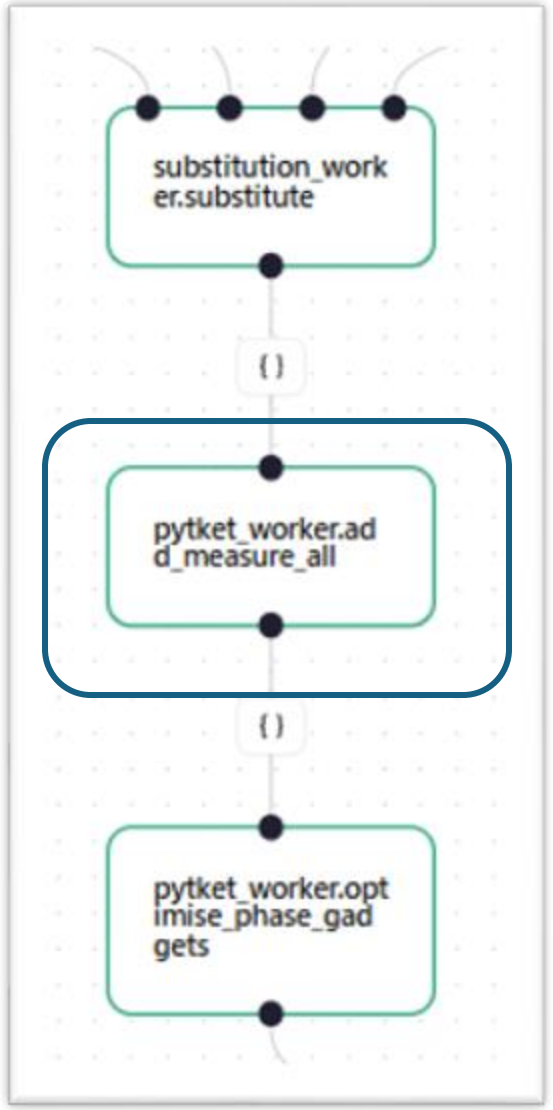
```
substituted_circuit = g.task(
    substitution_worker.substitute(a=a, b=b, c=c, circuit=ansatz)
)


measurement_circuit = g.task(
    pytket_worker.add_measure_all(circuit=substituted_circuit)
)


compiled_circuit = g.task(
    pytket_worker.optimise_phase_gadgets(circuit=measurement_circuit)
)
```



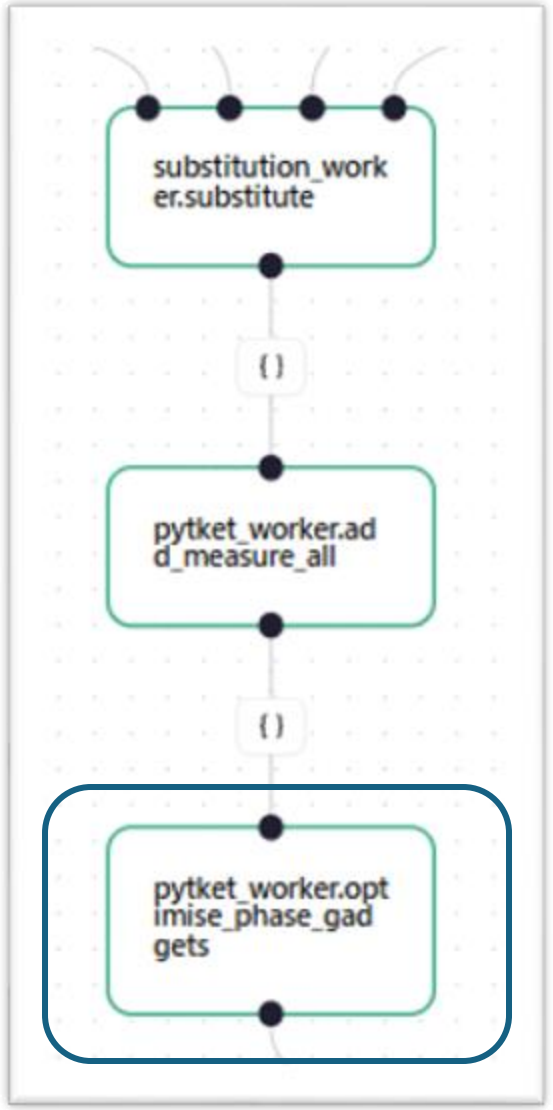
```
substituted_circuit = g.task(
    substitution_worker.substitute(a=a, b=b, c=c, circuit=ansatz)
)

measurement_circuit = g.task(
    pytket_worker.add_measure_all(circuit=substituted_circuit)
)

compiled_circuit = g.task(
    pytket_worker.optimise_phase_gadgets(circuit=measurement_circuit)
)
```



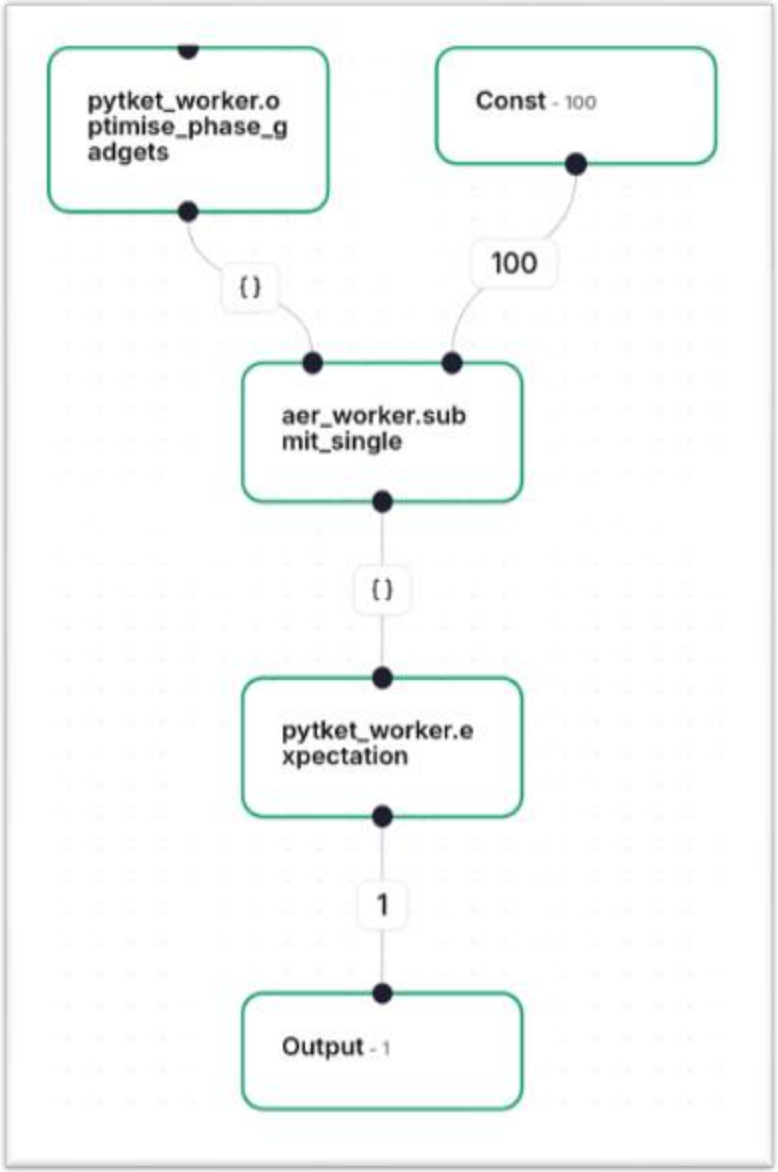
```
substituted_circuit = g.task(
    substitution_worker.substitute(a=a, b=b, c=c, circuit=ansatz)
)

measurement_circuit = g.task(
    pytket_worker.add_measure_all(circuit=substituted_circuit)
)

compiled_circuit = g.task(
    pytket_worker.optimise_phase_gadgets(circuit=measurement_circuit)
)
```



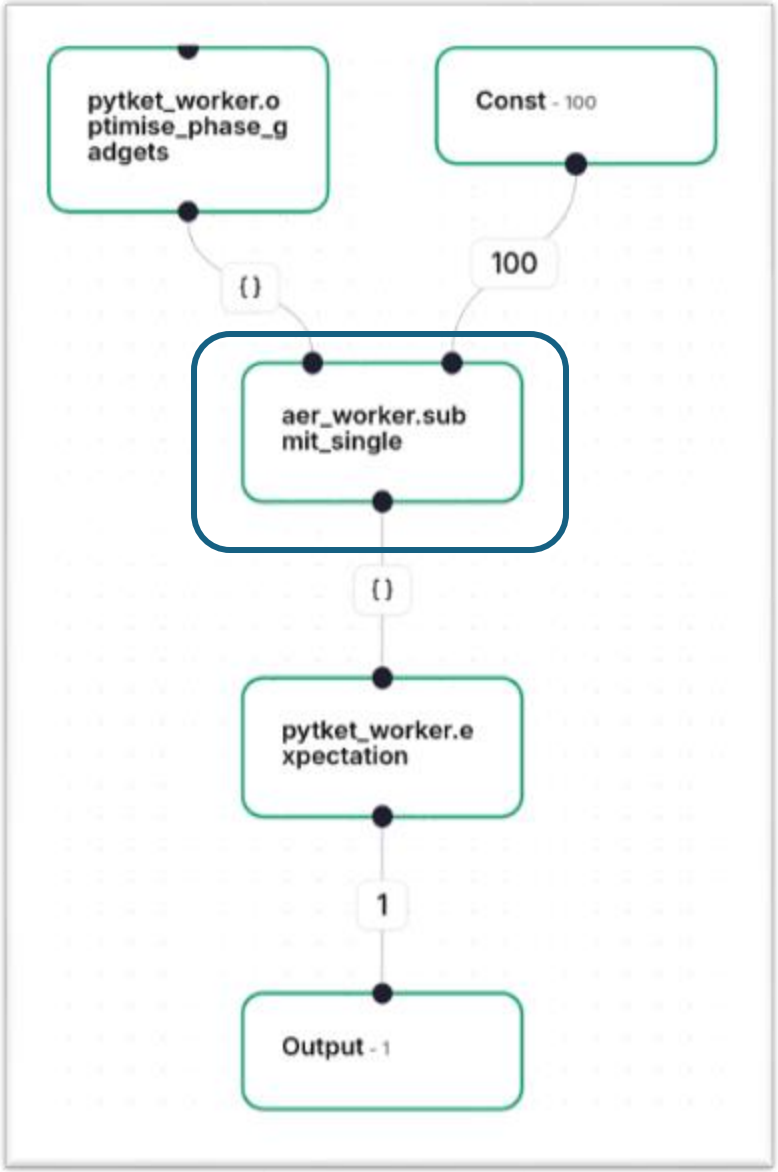
```
substituted_circuit = g.task(
    substitution_worker.substitute(a=a, b=b, c=c, circuit=ansatz)
)


measurement_circuit = g.task(
    pytket_worker.add_measure_all(circuit=substituted_circuit)
)


compiled_circuit = g.task(
    pytket_worker.optimise_phase_gadgets(circuit=measurement_circuit)
)
```



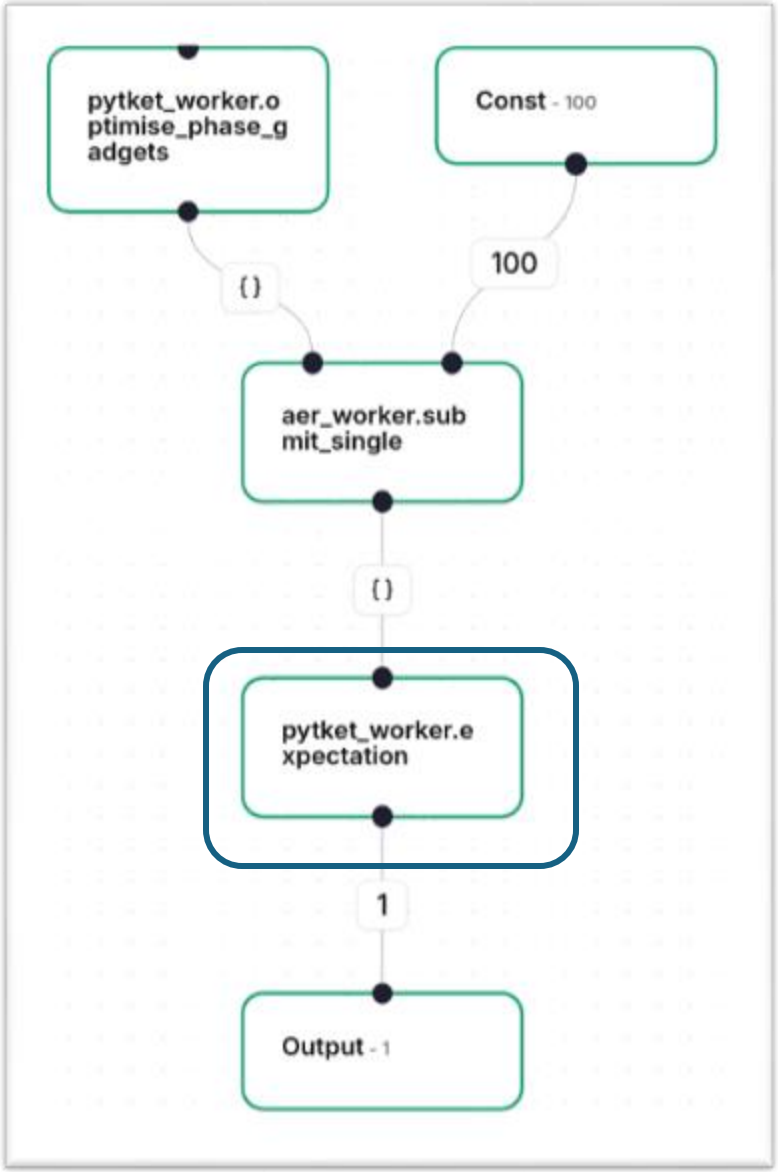
```
backend_result = g.task(
    aer_worker.submit_single(
        circuit=compiled_circuit,
        n_shots=n_shots,
    )
)

av = g.task(
    pytket_worker.expectation(
        backend_result=backend_result,
    )
)


g.outputs(SymbolicCircuitsOutputs(expectation=av))

return g
```



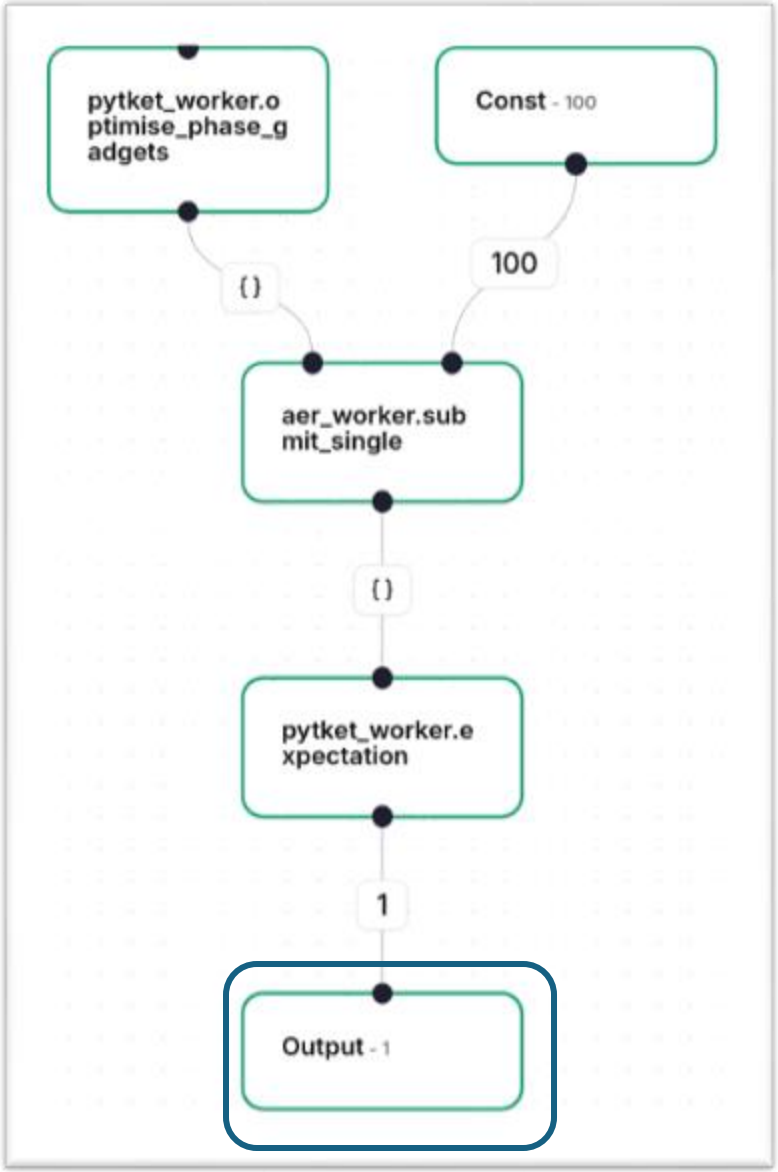
```
backend_result = g.task(
    aer_worker.submit_single(
        circuit=compiled_circuit,
        n_shots=n_shots,
    )
)

av = g.task(
    pytket_worker.expectation(
        backend_result=backend_result,
    )
)


g.outputs(SymbolicCircuitsOutputs(expectation=av))

return g
```



```
backend_result = g.task(
    aer_worker.submit_single(
        circuit=compiled_circuit,
        n_shots=n_shots,
    )
)

av = g.task(
    pytket_worker.expectation(
        backend_result=backend_result,
    )
)

g.outputs(SymbolicCircuitsOutputs(expectation=av))

return g
```

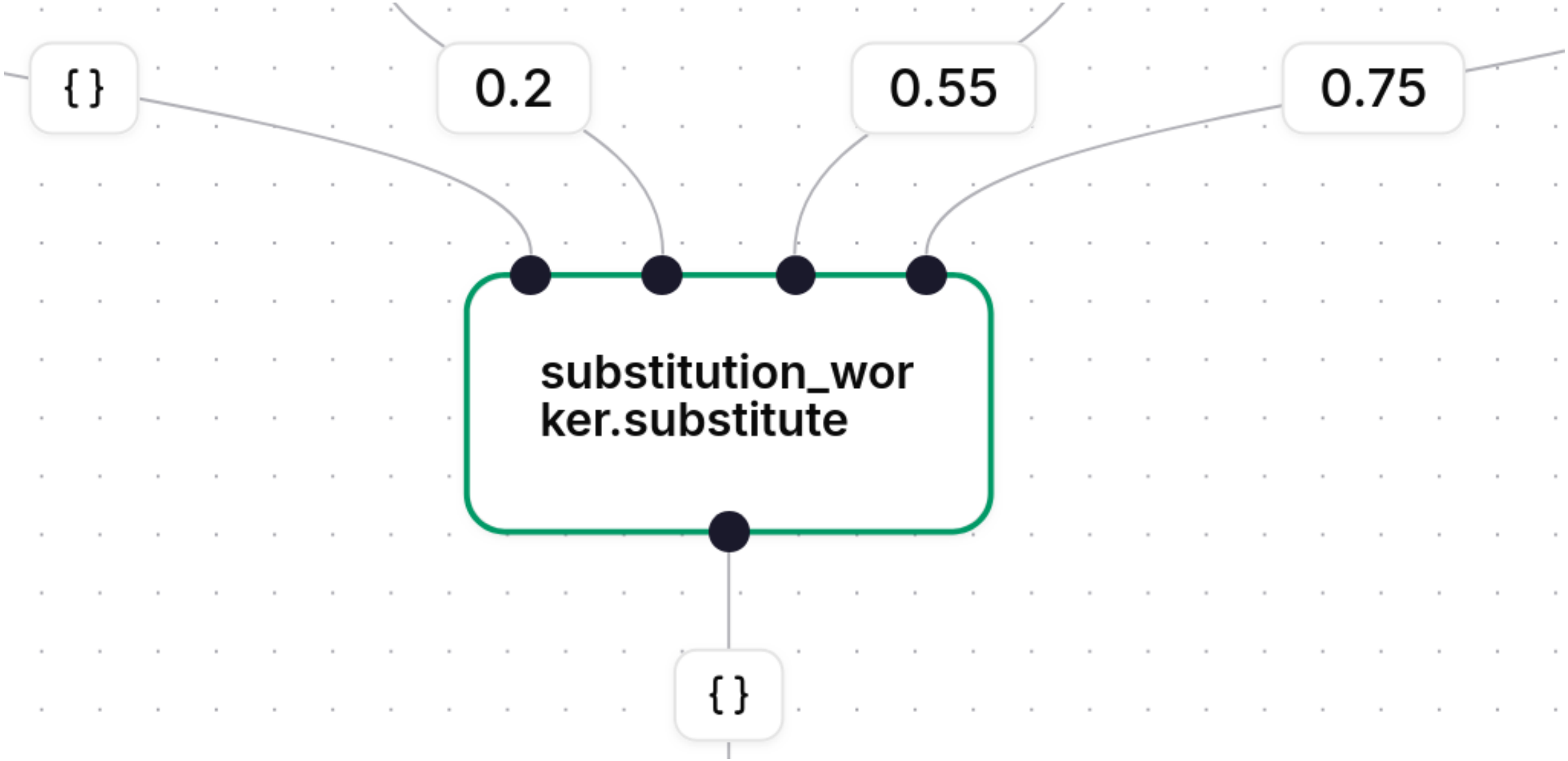


```
backend_result = g.task(
    aer_worker.submit_single(
        circuit=compiled_circuit,
        n_shots=n_shots,
    )
)

av = g.task(
    pytket_worker.expectation(
        backend_result=backend_result,
    )
)

g.outputs(SymbolicCircuitsOutputs(expectation=av))

return g
```



QUANTINUUM

Writing Workers


{ }
0.2
0.55
0.75
substitution_worker.substitute
{ }

```python
from tierkreis import Worker

worker = Worker("substitution_worker")

@worker.task()
def substitute(circuit: Circuit, a: float, b: float, c: float) -> Circuit:
    circuit.symbol_substitution(
        {Symbol("a"): a, Symbol("b"): b, Symbol("c"): c}
    )
    return circuit

if __name__ == "__main__":
    worker.app(argv)
```

```python
from tierkreis import Worker

worker = Worker("substitution_worker")

@worker.task()
def substitute(circuit: Circuit, a: float, b: float, c: float) -> Circuit:
    circuit.symbol_substitution(
        {Symbol("a"): a, Symbol("b"): b, Symbol("c"): c}
    )
    return circuit

if __name__ == "__main__":
    worker.app(argv)
```

```python
from tierkreis import Worker

worker = Worker("substitution_worker")

@worker.task()
def substitute(circuit: Circuit, a: float, b: float, c: float) -> Circuit:
    circuit.symbol_substitution(
        {Symbol("a"): a, Symbol("b"): b, Symbol("c"): c}
    )
    return circuit

if __name__ == "__main__":
    worker.app(argv)
```

```python
from tierkreis import Worker

worker = Worker("substitution_worker")

@worker.task()
def substitute(circuit: Circuit, a: float, b: float, c: float) -> Circuit:
    circuit.symbol_substitution(
        {Symbol("a"): a, Symbol("b"): b, Symbol("c"): c}
    )
    return circuit

if __name__ == "__main__":
    worker.app(argv)
```

```
python main.py --stubs-path ./stubs.py
```

```
class substitute(NamedTuple):
    circuit: TKR[OpaqueType["pytket._tket.circuit.Circuit"]] # noqa: F821 # fmt: skip
    a: TKR[float] # noqa: F821 # fmt: skip
    b: TKR[float] # noqa: F821 # fmt: skip
    c: TKR[float] # noqa: F821 # fmt: skip

    @staticmethod
    def out() -> type[TKR[OpaqueType["pytket._tket.circuit.Circuit"]]]: # noqa: F821 # fmt: skip
        return TKR[OpaqueType["pytket._tket.circuit.Circuit"]] # noqa: F821 # fmt: skip

    @property
    def namespace(self) -> str:
        return "substitution_worker"
```

QUANTINUUM

Running the Workflow

```
ansatz = build_ansatz()

# Assign a fixed uuid for our workflow.
workflow_id = UUID(int=101)
storage = ControllerFileStorage(
   workflow_id, name="symbolic_circuits", do_cleanup=True
)

# A little more boilerplate

print("Starting workflow at location:", storage.logs_path)
```

```
ansatz = build_ansatz()

# Assign a fixed uuid for our workflow.
workflow_id = UUID(int=101)
storage = ControllerFileStorage(
    workflow_id, name="symbolic_circuits", do_cleanup=True
)

# A little more boilerplate

print("Starting workflow at location:", storage.logs_path)
```

```python
# Look for workers in the `example_workers` directory.
registry_path = Path(__file__).parent / "example_workers"
custom_executor = UvExecutor(
    registry_path=registry_path, logs_path=storage.logs_path
)

common_registry_path = Path(__file__).parent.parent / "tierkreis_workers"
common_executor = UvExecutor(
    registry_path=common_registry_path,   logs_path=storage.logs_path
)

multi_executor = MultipleExecutor(
    common_executor,
    executors={"custom": custom_executor},
    assignments={"substitution_worker": "custom"},
)
```

```python
# Look for workers in the `example_workers` directory.
registry_path = Path(__file__).parent / "example_workers"
custom_executor = UvExecutor(
    registry_path=registry_path, logs_path=storage.logs_path
)
common_registry_path = Path(__file__).parent.parent / "tierkreis_workers"
common_executor = UvExecutor(
    registry_path=common_registry_path,   logs_path=storage.logs_path
)

multi_executor = MultipleExecutor(
    common_executor,
    executors={"custom": custom_executor},
    assignments={"substitution_worker": "custom"},
)
```

```
# Look for workers in the `example_workers` directory.
registry_path = Path(__file__).parent / "example_workers"
custom_executor = UvExecutor(
    registry_path=registry_path, logs_path=storage.logs_path
)

common_registry_path = Path(__file__).parent.parent / "tierkreis_workers"
common_executor = UvExecutor(
    registry_path=common_registry_path,   logs_path=storage.logs_path
)

multi_executor = MultipleExecutor(
    common_executor,
    executors={"custom": custom_executor},
    assignments={"substitution_worker": "custom"},
)
```

```python
# Look for workers in the `example_workers` directory.
registry_path = Path(__file__).parent / "example_workers"
custom_executor = UvExecutor(
    registry_path=registry_path, logs_path=storage.logs_path
)

common_registry_path = Path(__file__).parent.parent / "tierkreis_workers"
common_executor = UvExecutor(
    registry_path=common_registry_path,   logs_path=storage.logs_path
)

multi_executor = MultipleExecutor(
    common_executor,
    executors={"custom": custom_executor},
    assignments={"substitution_worker": "custom"},
)
```

```python
def main() -> None:
    """Configure our workflow execution and run it to completion."""
     … # Some Boilerplate

    run_graph(
        storage,
        multi_executor,
        symbolic_execution(),
        {
            "ansatz": ansatz,
            "a": 0.2,
            "b": 0.55,
            "c": 0.75,
        },
        polling_interval_seconds=0.1,
    )
```

```
python workflow.py
```

QUANTINUUM

Adapting Executables

```
#!/usr/bin/env bash
openssl genrsa --aes128 --passout stdin 4096
```

```
shell = ShellExecutor(registry_path, storage.workflow_dir)
```

```
private_key = g.task(script("genrsa", passphrase))
```

QUANTINUUM

Untyped Workers

```
key_pair = g.data.func( # escape hatch into untyped builder
    "openssl_worker.genrsa",
    {"passphrase": passphrase.value_ref(), "numbits": g.const(4096).value_ref()},
)
private_key: TKR[bytes] = TKR(*key_pair("private_key")) # unsafe cast
public_key: TKR[bytes] = TKR(*key_pair("public_key")) # unsafe cast
```

QUANTINUUM

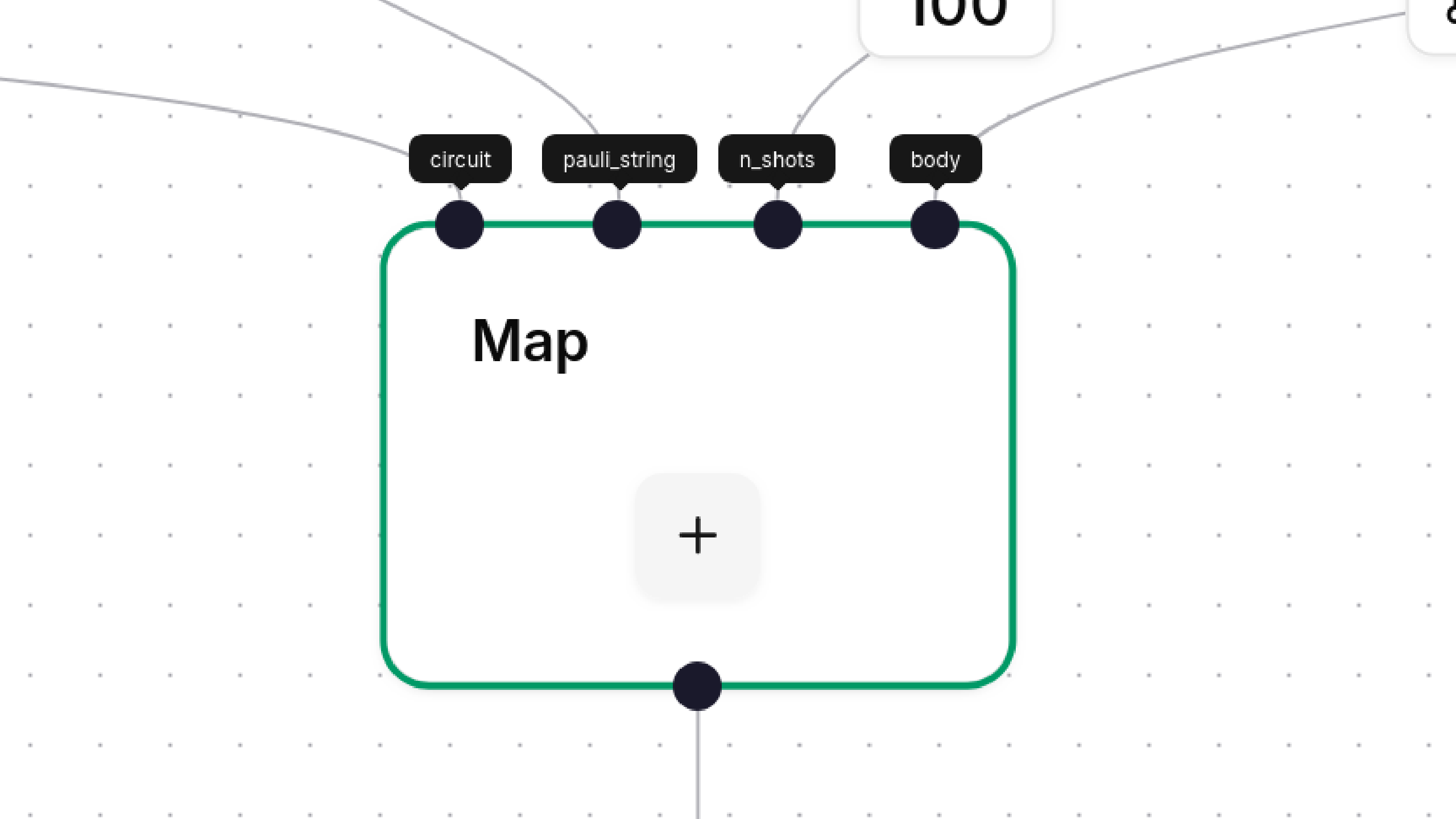
Running on HPC

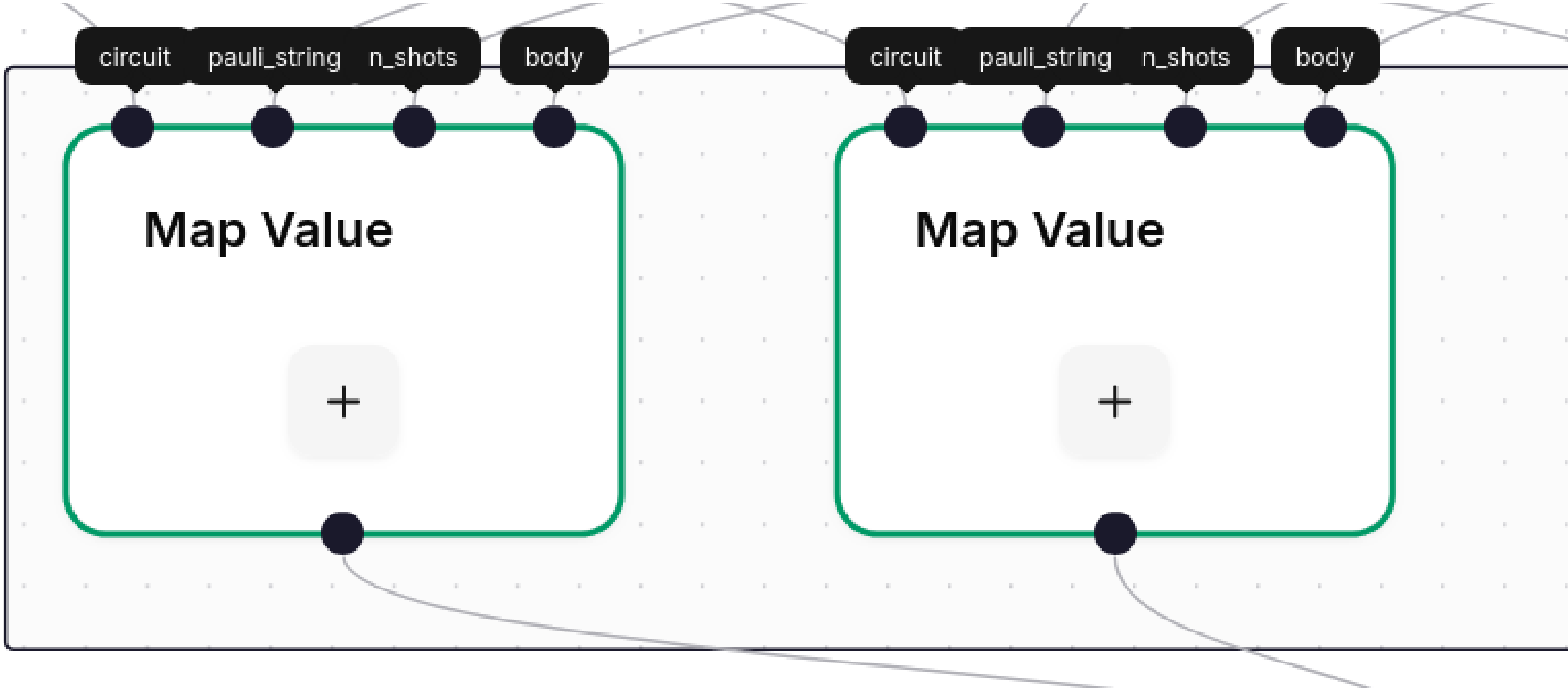
```
spec = JobSpec(
    job_name="tkr_symbolic_circuits",
    account=group_name,
    command="env UV_PROJECT_ENVIRONMENT=compute_venv uv run main.py",
    resource=ResourceSpec(nodes=1, memory_gb=None, gpus_per_node=None),
    walltime="00:15:00",
    output_path=Path(logs_path),
    error_path=Path(logs_path),
    include_no_check_directory_flag=True,
)
custom_executor =
    PJSUBExecutor(spec=spec, registry_path=registry_path, logs_path=logs_path)
```

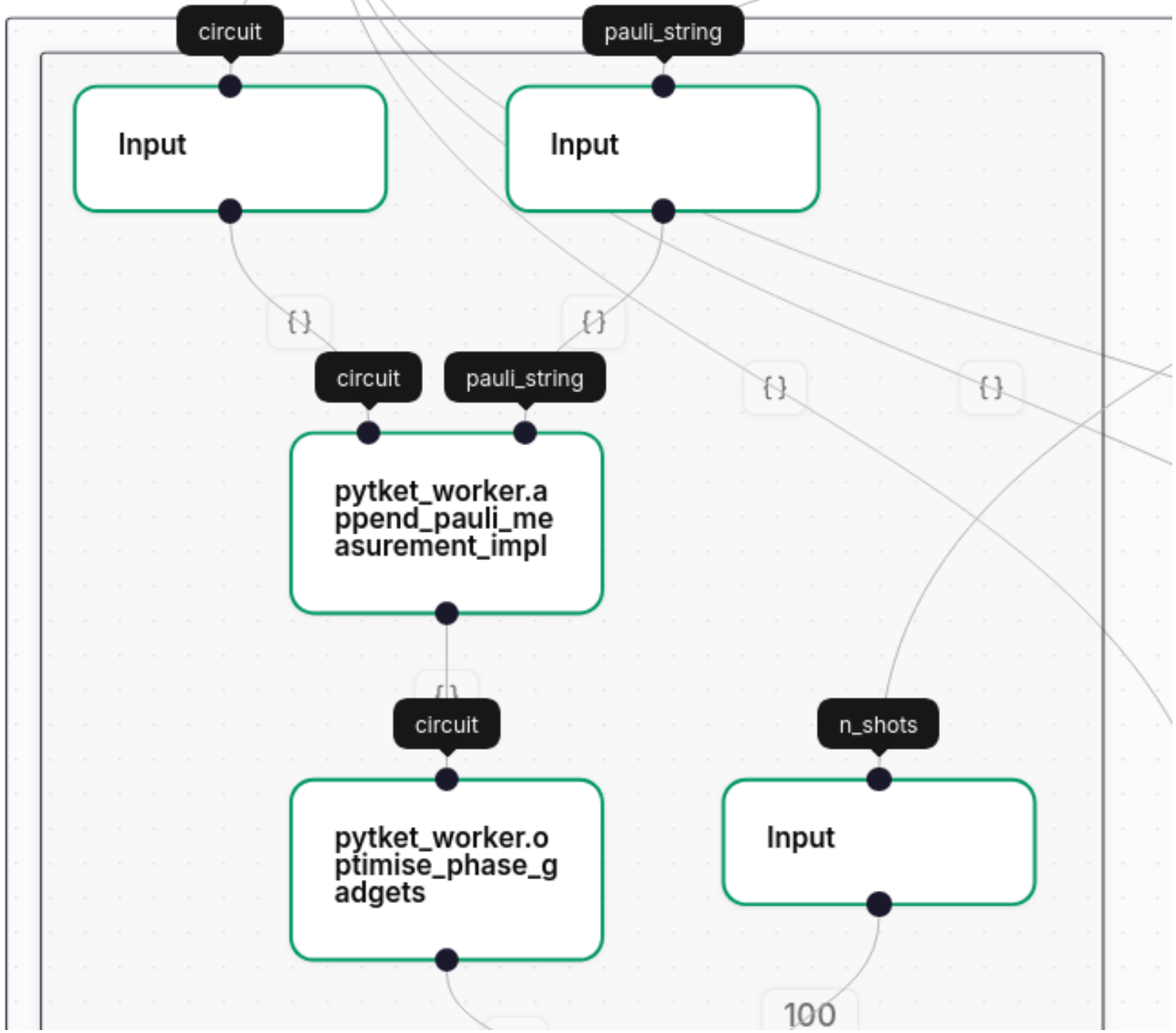
QUANTINUUM

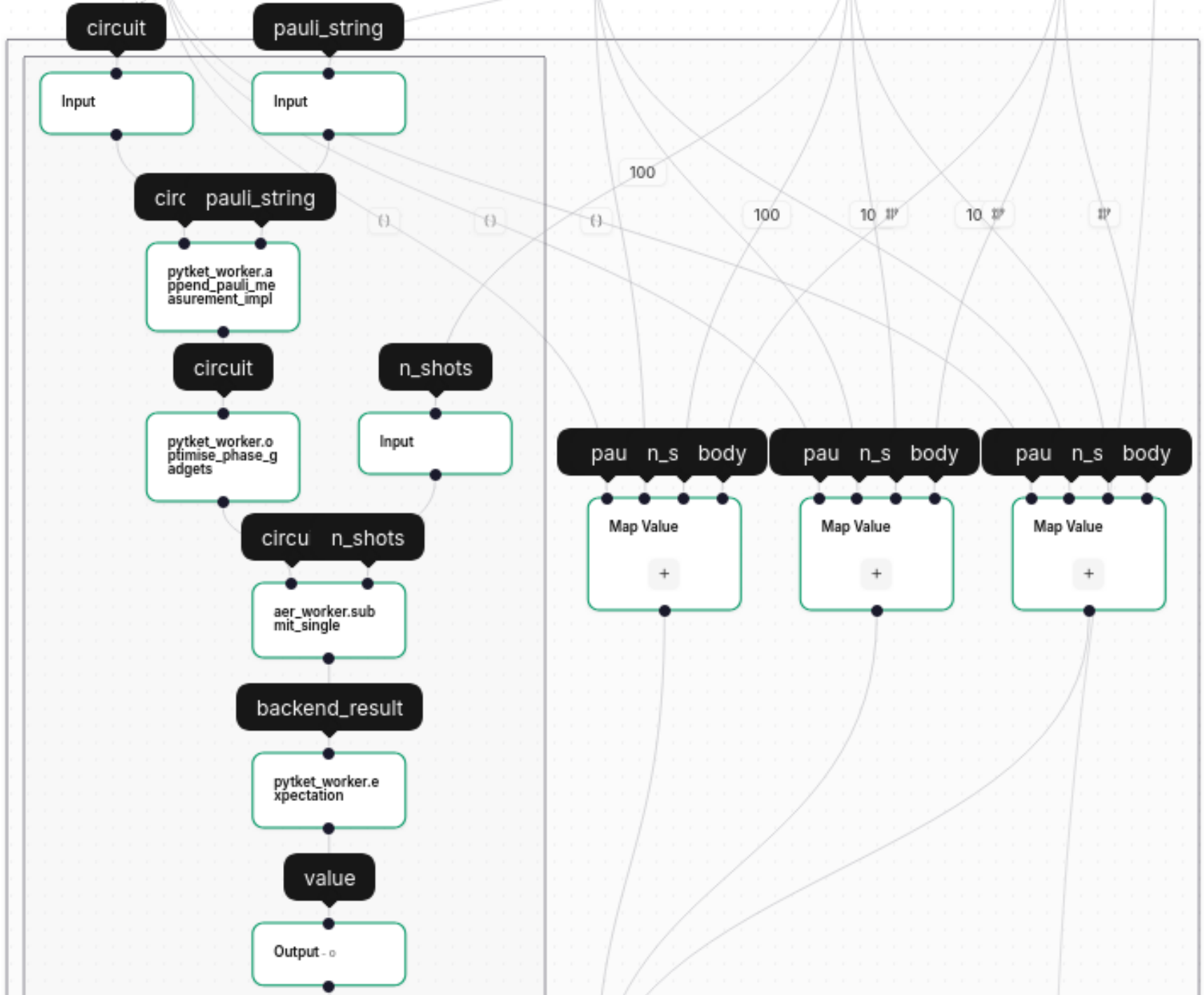
Parallelism

```
aes = g.map(
    lambda x: SubgraphInputs(substituted_circuit, x, g.const(100)),
    pauli_strings_list,
)
m = g.map(_subgraph(), aes)
```


100
circuit
pauli_string
n_shots
body
Map


circuit
pauli_string
n_shots
body
Map Value
+
circuit
pauli_string
n_shots
body
Map Value
+


circuit
pauli_string
Input
Input
{}
{}
circuit
pauli_string
{}
{}
pytket_worker.a
ppend_pauli_me
asurement_impl
circuit
n_shots
pytket_worker.o
ptimise_phase_g
adgets
Input
100


circuit
pauli_string
Input
Input
100
circ pauli_string
100
10
10
pytket_worker.append_pauli_measurement_impl
circuit
n_shots
pytket_worker.optimise_phase_gadgets
Input
pau n_s body
pau n_s body
pau n_s body
Map Value
Map Value
Map Value
circu n_shots
aer_worker.submit_single
backend_result
pytket_worker.expectation
value
Output

QUANTINUUM

Getting Started

Getting Started

- Documentation https://cqcl.github.io/tierkreis/
- From pypi: pip install tierkreis
- Repository: https://github.com/CQCL/tierkreis