SCA/HPCAsia 2026 Tutorial

Workload and Workflow Management in Quantum-Classical HPC

Shweta Salaria
IBM Research

Why Scripts Are Not Enough for QCSC Workflows?

Limitations of a typical script-based approach

Execution history is not systematically recorded

It becomes unclear which parameters were used and when jobs were executed.

Recovery from failures is difficult

A failure in the middle often requires restarting the entire workflow.

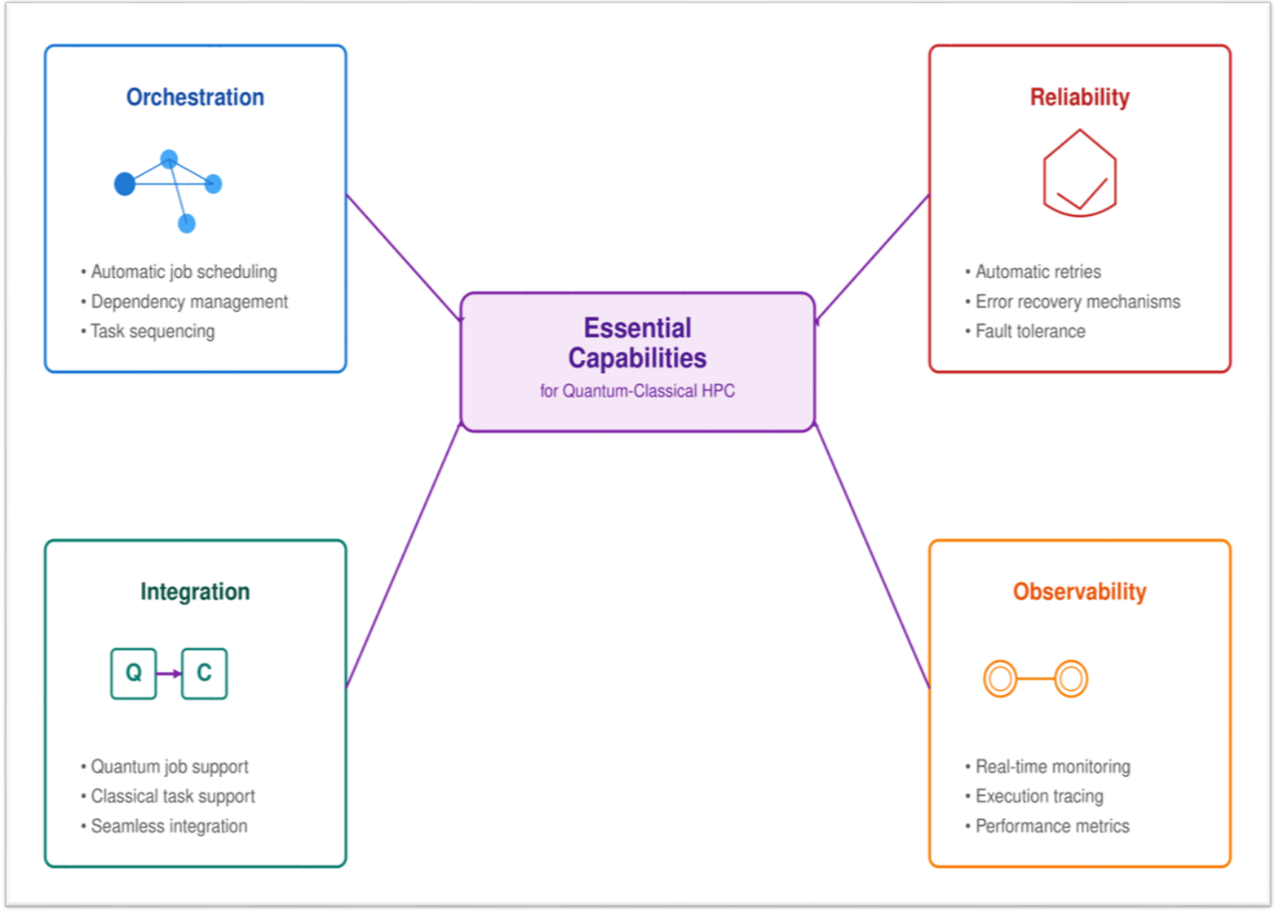
Quantum and classical steps are fragmented

Quantum executions, HPC jobs, and post-processing are managed separately.

Reproducibility and comparison are hard

It is difficult to trace why a particular result was obtained.

What Should a QCSC Orchestrator Deliver?



Explicit execution flow

The execution order and dependencies between steps are clearly defined and managed by the system.

Efficient recovery from failures

Only failed tasks need to be re-executed, avoiding unnecessary re-runs of expensive computations.

Unified quantum–classical workflows

Quantum executions and HPC jobs are handled within a single, coherent workflow

Reproducible experimental runs

Results are automatically linked with parameters and execution context, enabling reliable comparison.

Prefect: Python-based Workflow Orchestration Tool

- **Native Python**
  Write workflows using pure Python code with decorators
- **Dynamic Execution**
  Support for conditional logic, loops and runtime parameters
- **Observability**
  Built-in UI for monitoring, logging and debugging workflows
- **Error Handling**
  Automatic retries, failure notifications and recovery mechanisms
- **Deployment Options**
  Run locally, on-premise or in Prefect Cloud
- **Scheduling**
  Cron-based, interval and event-driven scheduling

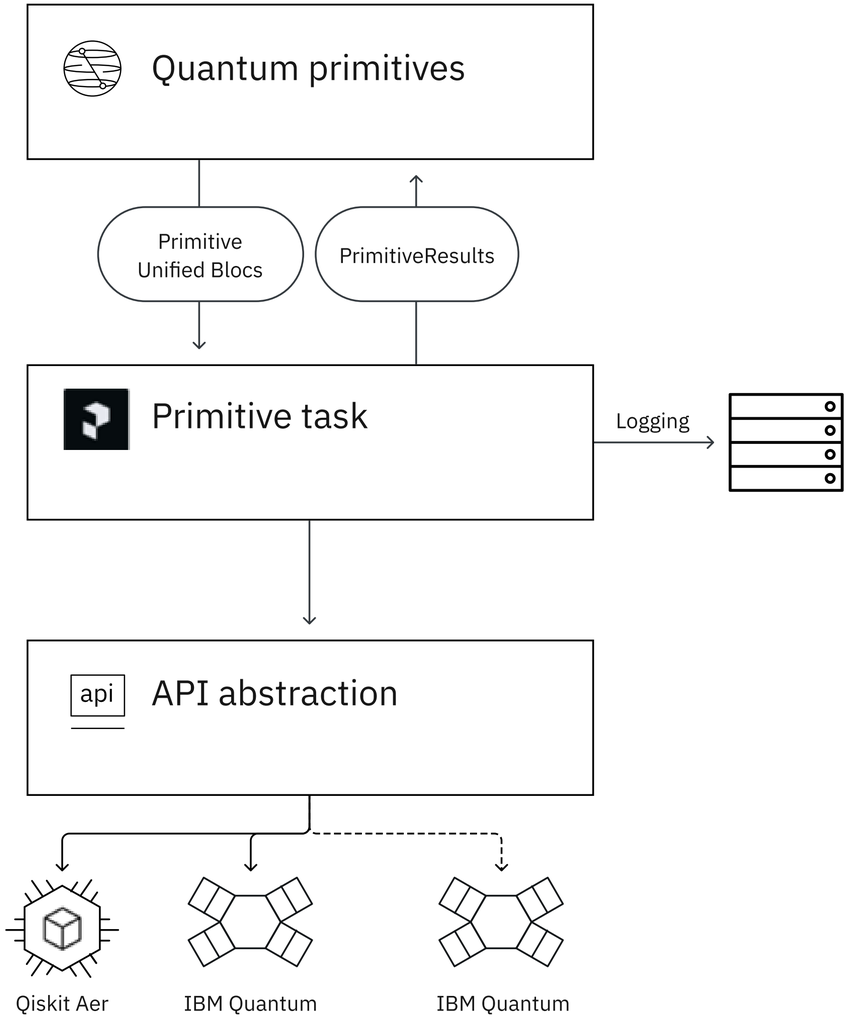
Prefect

```python
import requests
from prefect import flow, task

@task(retries=3, retry_delay_seconds=5)
def fetch_data(url: str):
    return requests.get(url).json()

@flow(name="Data Ingestion Pipeline")
def my_workflow(api_url: str):
    # This looks like standard Python, but it's now orchestrated!
    data = fetch_data(api_url)
    print(f"Processed: {data}")

if __name__ == "__main__":
    my_workflow(api_url="https://api.example.com/data")
```

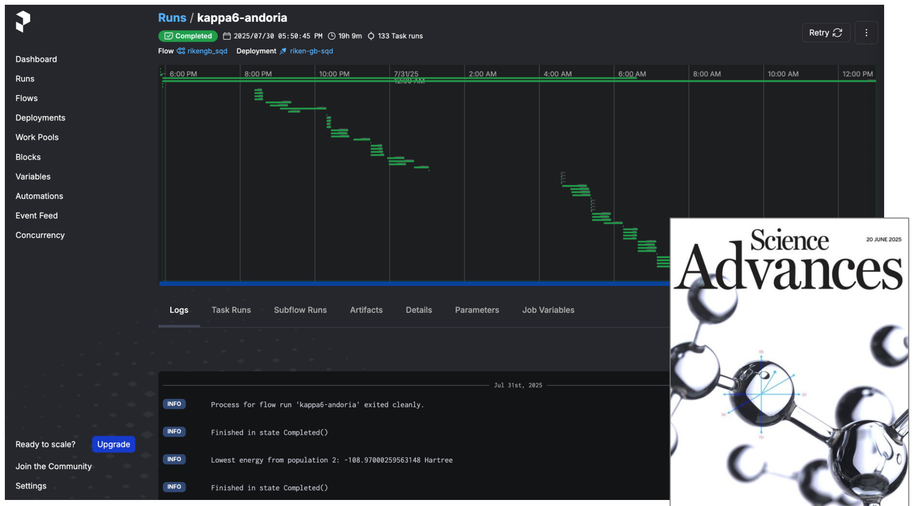
Prefect Supports Qiskit

Features:

- Qiskit primitive programming with workflow syntax
- Multiple vendor support on API abstraction
- Async execution support for parallel primitive execution
- Built-in repeat until success mechanism for backend error handling
- Result cache mechanism for identical inputs
- Artifact generation for job performance metadata




```python
@flow
def sample_bell():
    logger = get_run_logger()

    # 1. Load QuantumRuntime from Prefect Server
    runtime = QuantumRuntime.load("default-runtime")
    options = Variable.get("sampler_options")

    # 2. Create ISA quantum circuit
    bell = QuantumCircuit(2)
    bell.h(0)
    bell.cx(0, 1)
    bell.measure_all()
    isa_circ = transpile_task(
        circuit=bell,
        target=runtime.get_target(),
    )

    # 3. Execute workflow sampler
    result = runtime.sampler([isa_circ], options=options)
    logger.info(result[0].data.meas.get_counts())
```



QCSC Workflow with Prefect

Program: BitCount (Quantum + MPI)

```python
import asyncio
import json
from prefect import flow
from prefect.artifacts import create_table_artifact
from prefect.variables import Variable
from prefect_qiskit import QuantumRuntime
from qiskit import QuantumCircuit
from qiskit.transpiler import generate_preset_pass_manager
from get_counts_integration import BitCounter
from qiskit.primitives.containers.sampler_pub import SamplerPub
from qiskit import qasm3
from get_task_runner import TaskRunner
from qiskit_ibm_runtime.utils import RuntimeEncoder
from qiskit_ibm_runtime.utils.result_decoder import ResultDecoder

BITLEN = 10

@flow(name="slurm_tutorial")
async def main():
    # Load configurations
    runtime = await QuantumRuntime.load("ibm-runner")
    counter = await BitCounter.load("slurm-tutorial")
    options = await Variable.get("slurm-tutorial")
    taskrunner = await TaskRunner.load("slurm-tutorial")

    # Create a PUB payload
    target = await runtime.get_target()
    qc_ghz = QuantumCircuit(BITLEN)
    qc_ghz.h(0)
    qc_ghz.cx(0, range(1, BITLEN))
    qc_ghz.measure_active()

    pm = generate_preset_pass_manager(
        optimization_level=3,
        target=target,
        seed_transpiler=123,
    )
```

- Sample bitstrings using a quantum computer

```python
isa = pm.run(qc_ghz)
pub_like = (isa,)  # Create a Primitive Unified Bloc

# Extract shots
shots = options.get("shots", 100000) # default to 100000 if not set

dict_pubs = []

# Create input.json for task_runner
coerced_pub = SamplerPub.coerce(pub_like, shots=shots)

# Generate OpenQASM3 string which can be consumed by IBM Quantum APIs
qasm3_str = qasm3.dumps(
        coerced_pub.circuit,
        disable_constants=True,
        allow_aliasing=True,
        experimental=qasm3.ExperimentalFeatures.SWITCH_CASE_V1,
)

if len(coerced_pub.circuit.parameters) == 0:
    if coerced_pub.shots:
        dict_pubs.append((qasm3_str, None, coerced_pub.shots))
    else:
        dict_pubs.append((qasm3_str))
else:
    param_array = coerced_pub.parameter_values.as_array(
    coerced_pub.circuit.parameters
            ).tolist()

    if coerced_pub.shots:
        dict_pubs.append((qasm3_str, param_array, coerced_pub.shots))
    else:
        dict_pubs.append((qasm3_str, param_array))
```

- Count how many times each bitstring appears using MPI

```python
    # Create SamplerV2 input
    input_json = {
            "pubs": dict_pubs,
            "shots": shots,
            "options": options,
            "version": 2,
            "support_qiskit": True,
    }
    print("Here is the input json:", input_json)

    taskrunner_json = {"parameters": input_json, "program_id": "sampler"}
    print("Here is the taskrunner json:", taskrunner_json)

    filename = "/mnt/data/salaria/slurm_tutorial/input.json"
    with open(filename, "w", encoding="utf-8") as primitive_input_file:
        json.dump(taskrunner_json, primitive_input_file, cls=RuntimeEncoder, indent=2)

    # Quantum execution
    result = await taskrunner.run()

    # Read output
    with open('/mnt/data/salaria/slurm_tutorial/output.json', 'r') as f:
        results = ResultDecoder.decode(f.read())

    # MPI execution
    bitstrings = results[0].data.meas.get_bitstrings()
    counts = await counter.get(bitstrings)

    # Save in Prefect artifacts
    await create_table_artifact(
        table=[list(counts.keys()), list(counts.values())],
        key="sampler-count-dict",
     )


if __name__ == "__main__":
    asyncio.run(main())
```

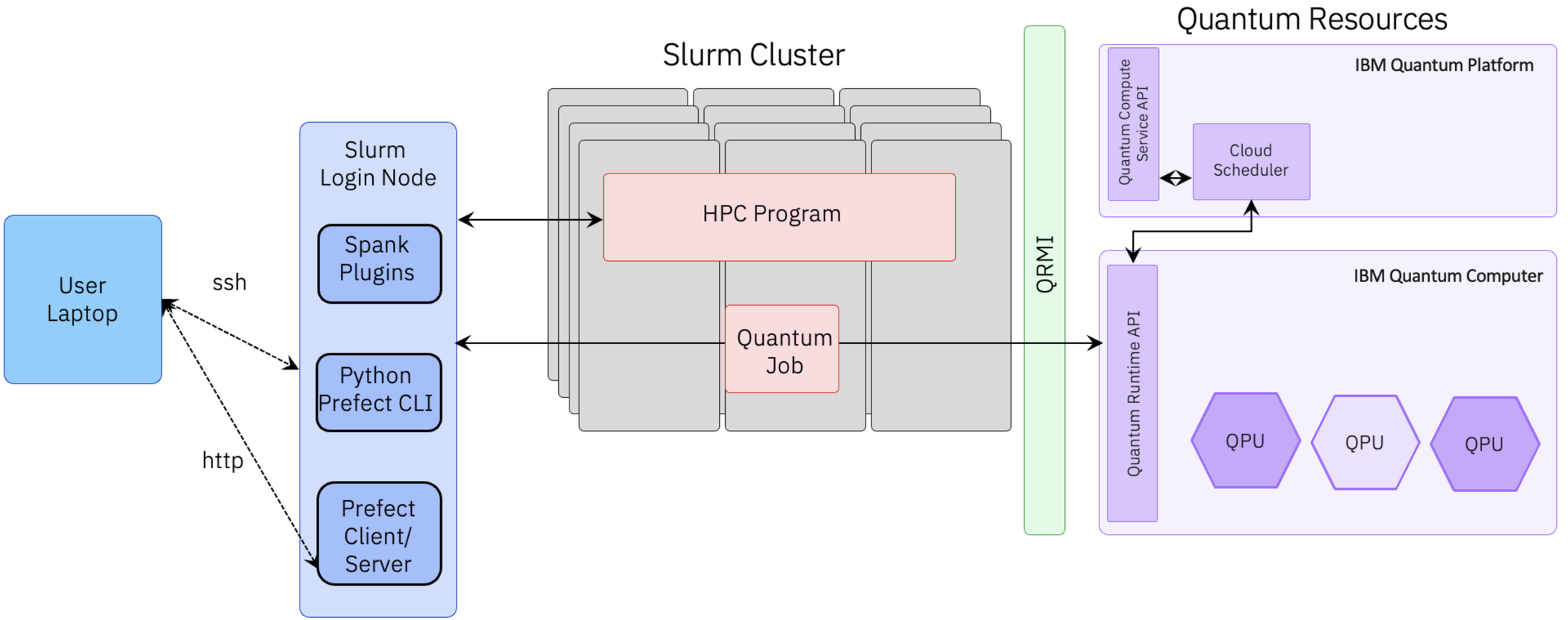
Create QCSC workflow: Quantum + MPI

Program: BitCount (Quantum + MPI)

- Sample bitstrings using a quantum computer
- Count how many times each bitstring appears using MPI

```python
import asyncio
import json
from prefect import flow
from prefect.artifacts import create_table_artifact
from prefect.variables import Variable
from prefect_qiskit import QuantumRuntime
from qiskit import QuantumCircuit
from qiskit.transpiler import generate_preset_pass_manager
from get_counts_integration import BitCounter
from qiskit.primitives.containers.sampler_pub import SamplerPub
from qiskit import qasm3
from get_task_runner import TaskRunner
from qiskit_ibm_runtime.utils import RuntimeEncoder
from qiskit_ibm_runtime.utils.result_decoder import ResultDecoder

BITLEN = 10

@flow(name="slurm_tutorial")
async def main():
    # Load configurations
    runtime = await QuantumRuntime.load("ibm-runner")
    counter = await BitCounter.load("slurm-tutorial")
    options = await Variable.get("slurm-tutorial")
    taskrunner = await TaskRunner.load("slurm-tutorial")

    # Create a PUB payload
    target = await runtime.get_target()
    qc_ghz = QuantumCircuit(BITLEN)
    qc_ghz.h(0)
    qc_ghz.cx(0, range(1, BITLEN))
    qc_ghz.measure_active()

    pm = generate_preset_pass_manager(
        optimization_level=3,
        target=target,
        seed_transpiler=123,
    )
    isa = pm.run(qc_ghz)
    pub_like = (isa,)  # Create a Primitive Unified Bloc

    # Extract shots
    shots = options.get("shots", 100000) # default to 100000 if not set

    dict_pubs = []

    # Create input.json for task_runner
    coerced_pub = SamplerPub.coerce(pub_like, shots=shots)

    # Generate OpenQASM3 string which can be consumed by IBM Quantum APIs
    qasm3_str = qasm3.dumps(
            coerced_pub.circuit,
            disable_constants=True,
            allow_aliasing=True,
            experimental=qasm3.ExperimentalFeatures.SWITCH_CASE_V1,
    )

    if len(coerced_pub.circuit.parameters) == 0:
        if coerced_pub.shots:
            dict_pubs.append((qasm3_str, None, coerced_pub.shots))
        else:
            dict_pubs.append((qasm3_str))
    else:
        param_array = coerced_pub.parameter_values.as_array(
        coerced_pub.circuit.parameters
                ).tolist()

        if coerced_pub.shots:
            dict_pubs.append((qasm3_str, param_array, coerced_pub.shots))
        else:
            dict_pubs.append((qasm3_str, param_array))

    # Create SamplerV2 input
    input_json = {
            "pubs": dict_pubs,
            "shots": shots,
            "options": options,
            "version": 2,
            "support_qiskit": True,
    }
    print("Here is the input json:", input_json)

    taskrunner_json = {"parameters": input_json, "program_id": "sampler"}
    print("Here is the taskrunner json:", taskrunner_json)

    filename = "/mnt/data/salaria/slurm_tutorial/input.json"
    with open(filename, "w", encoding="utf-8") as primitive_input_file:
        json.dump(taskrunner_json, primitive_input_file, cls=RuntimeEncoder, indent=2)

    # Quantum execution
    result = await taskrunner.run()

    # Read output
    with open('/mnt/data/salaria/slurm_tutorial/output.json', 'r') as f:
        results = ResultDecoder.decode(f.read())

    # MPI execution
    bitstrings = results[0].data.meas.get_bitstrings()
    counts = await counter.get(bitstrings)

    # Save in Prefect artifacts
    await create_table_artifact(
        table=[list(counts.keys()), list(counts.values())],
        key="sampler-count-dict",
     )


if __name__ == "__main__":
    asyncio.run(main())
```

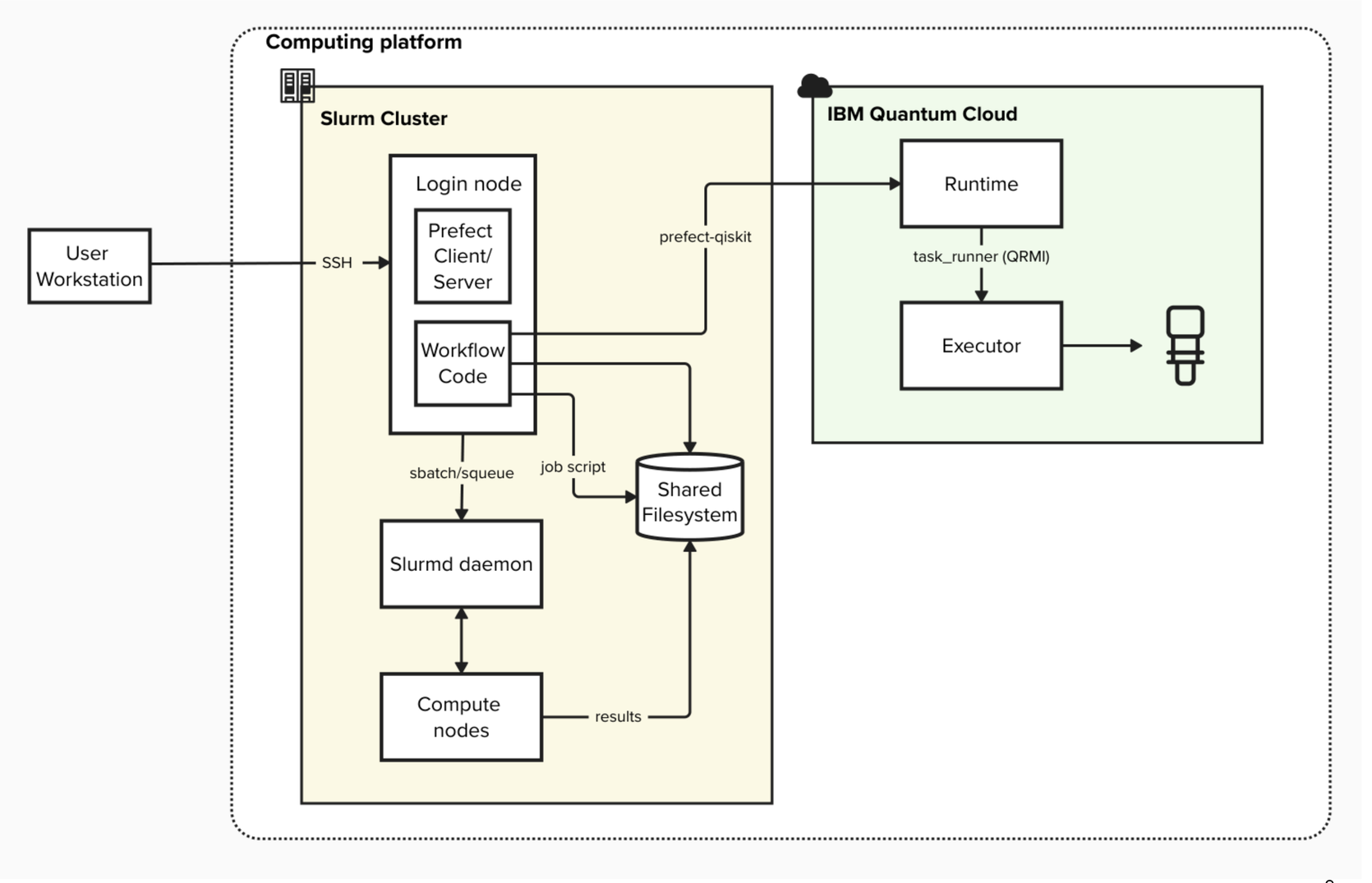
System Architecture



Hybrid Application Development with Prefect Qiskit

- Run a workflow server in login node
  - Submit Quantum jobs
  - Submit HPC jobs
  - Wait for results of jobs independently

- Use Python to write workflow
  - Prefect
  - Prefect Qiskit block to submit quantum jobs
  - Prefect MPI block to submit MPI jobs
    - Predefined jobs: BitCount



Prefect Core Concepts

Flow (end-to-end workflow)

- The Flow represents the full QCSC iteration

Tasks (individual execution steps)

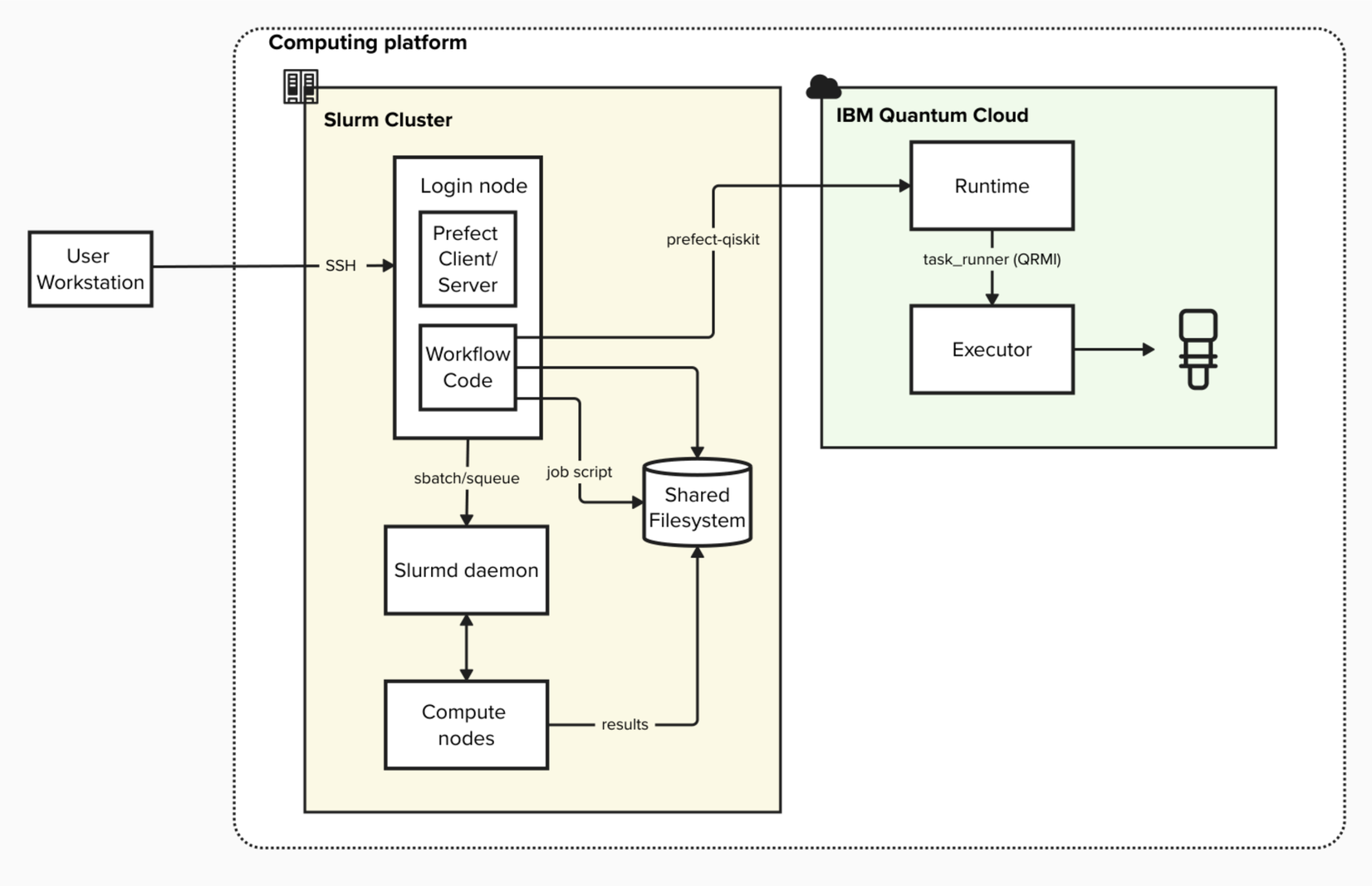
- Sampling Task: submit a slurm job to run Quantum sampling job.
- HPC Task: submit a slurm job to run MPI programs.

Blocks (reusable configuration + credentials)

- Quantum Blocks: IBM Quantum credentials (encrypted), runtime configuration, Slurm job configuration.
- HPC Blocks: Slurm job configuration.

Variables (run-time parameters)

- Quantum parameters: shots, circuit or sampler options.
- Algorithm parameters: iteration used by the workflow logic.



Prefect Core Concepts: Flow

Flow (end-to-end workflow)

- The Flow represents the full QCSC iteration

Tasks (individual execution steps)

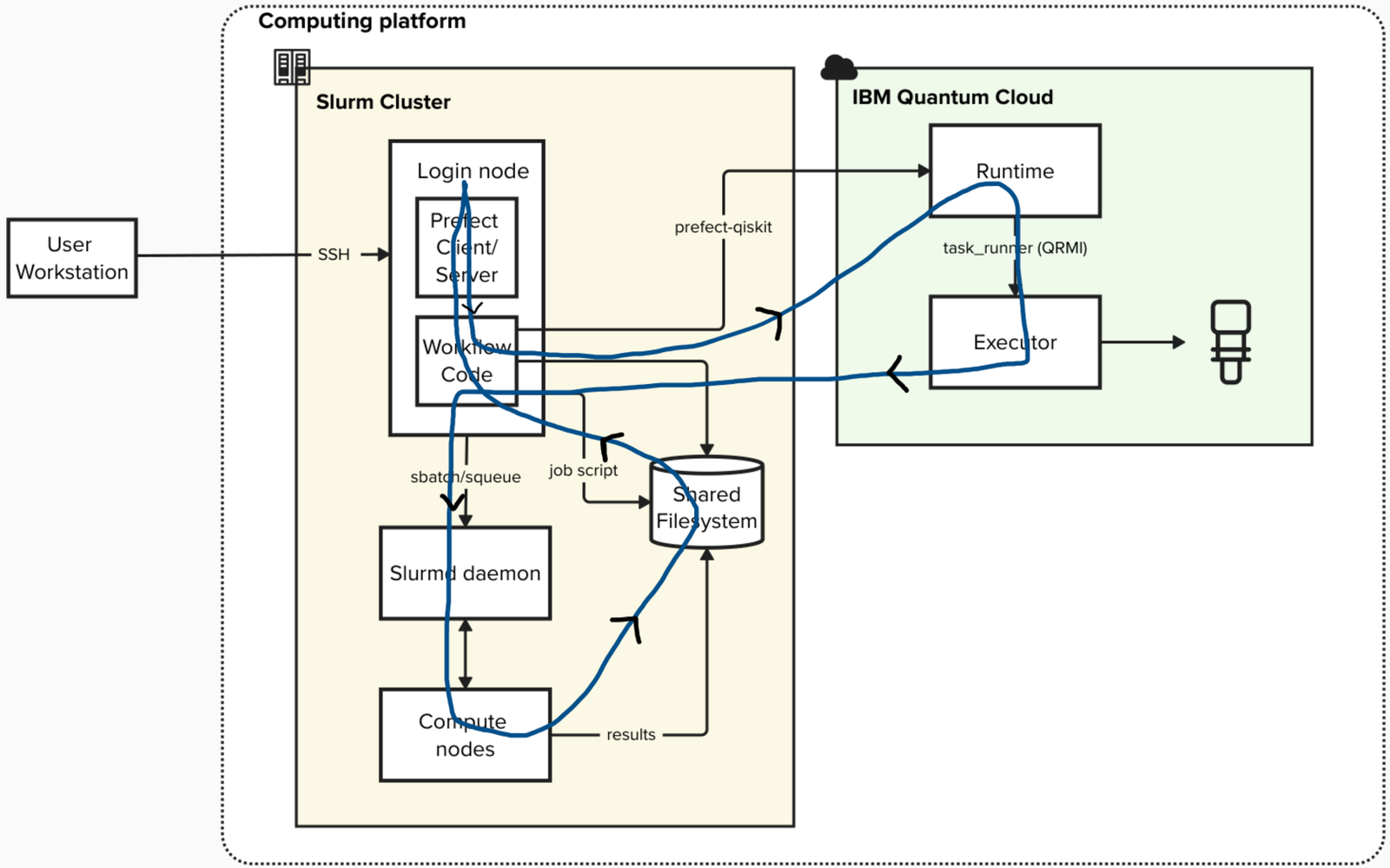
- Sampling Task: submit a slurm job to run Quantum sampling job.
- HPC Task: submit a slurm job to run MPI programs.

Blocks (reusable configuration + credentials)

- Quantum Blocks: IBM Quantum credentials, runtime configuration, Slurm job configuration.
- HPC Blocks: Slurm job configuration.

Variables (run-time parameters)

- Quantum parameters: shots, circuit or sampler options.
- Algorithm parameters: iteration used by the workflow logic.



Prefect Core Concepts: Tasks

Flow (end-to-end workflow)

- The Flow represents the full QCSC iteration

Tasks (individual execution steps)

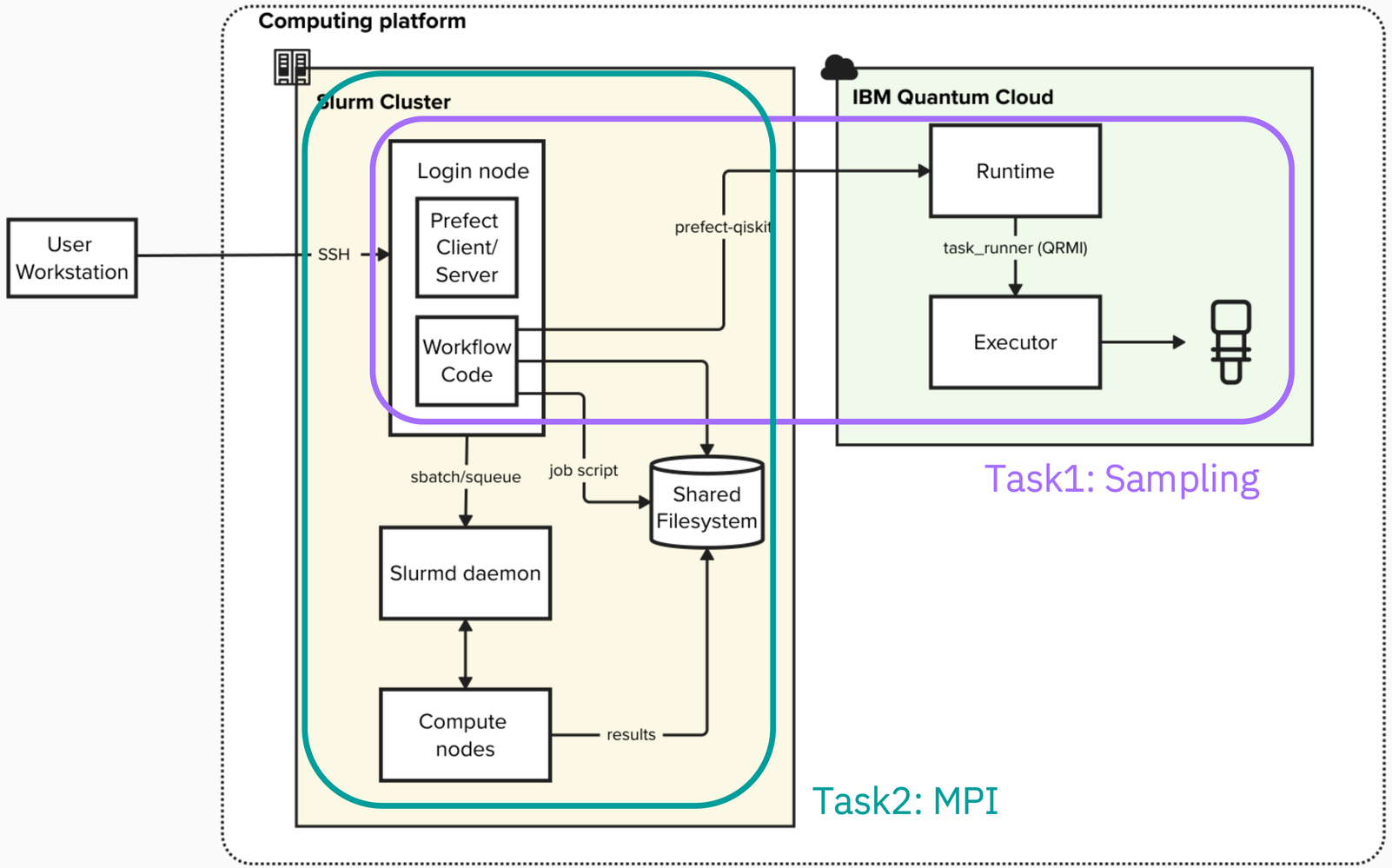
- Sampling Task: submit a slurm job to run Quantum sampling job.
- HPC Task: submit a slurm job to run MPI programs.

Blocks (reusable configuration + credentials)

- Quantum Blocks: IBM Quantum credentials, runtime configuration, Slurm job configuration.
- HPC Blocks: Slurm job configuration.

Variables (run-time parameters)

- Quantum parameters: shots, circuit or sampler options.
- Algorithm parameters: iteration used by the workflow logic.



Prefect Core Concepts: Blocks

Flow (end-to-end workflow)

- The Flow represents the full QCSC iteration

Tasks (individual execution steps)

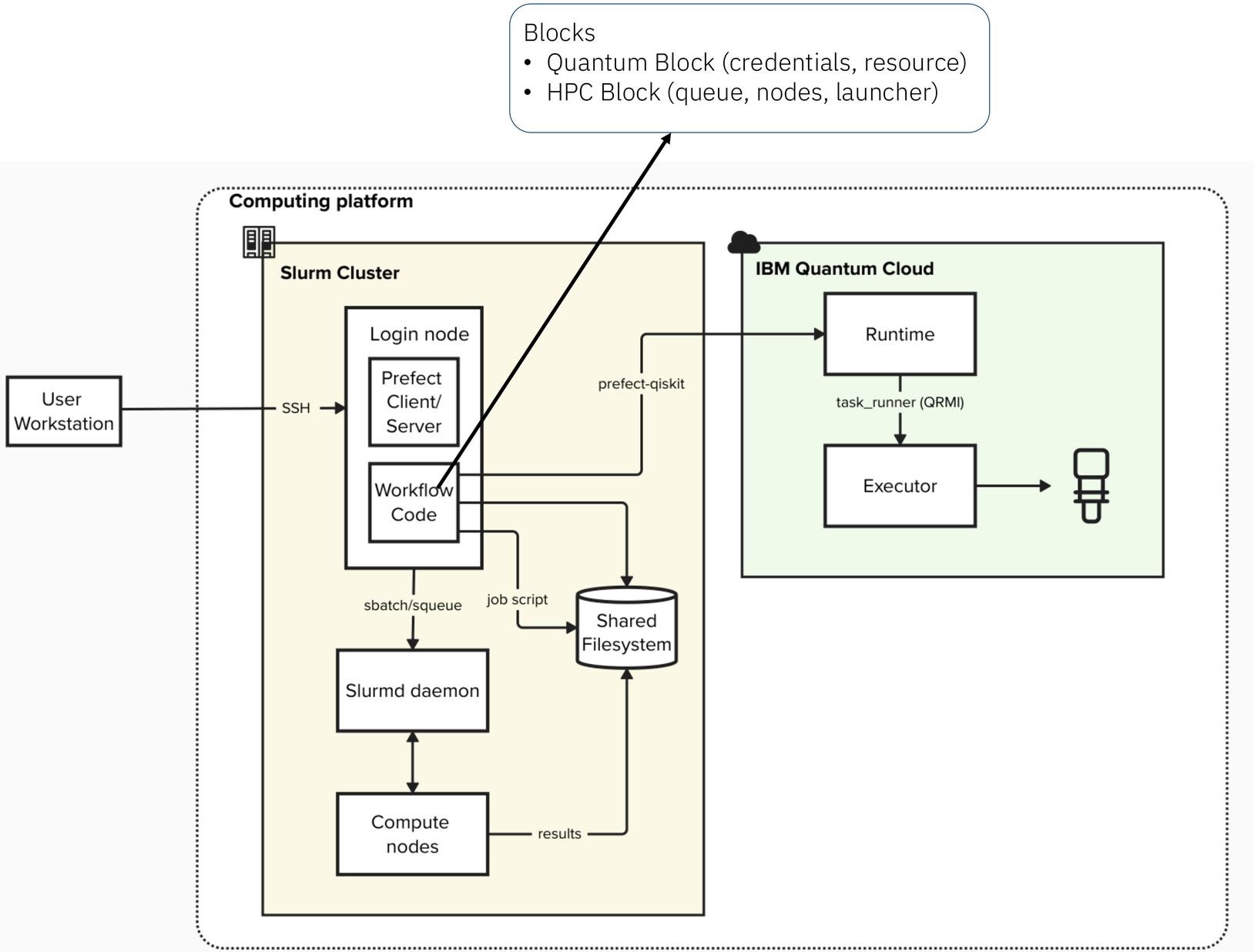
- Sampling Task: submit a slurm job to run Quantum sampling job.
- HPC Task: submit a slurm job to run MPI programs.

Blocks (reusable configuration + credentials)

- Quantum Blocks: IBM Quantum credentials, runtime configuration, Slurm job configuration.
- HPC Blocks: Slurm job configuration.

Variables (run-time parameters)

- Quantum parameters: shots, circuit or sampler options.
- Algorithm parameters: iteration used by the workflow logic.



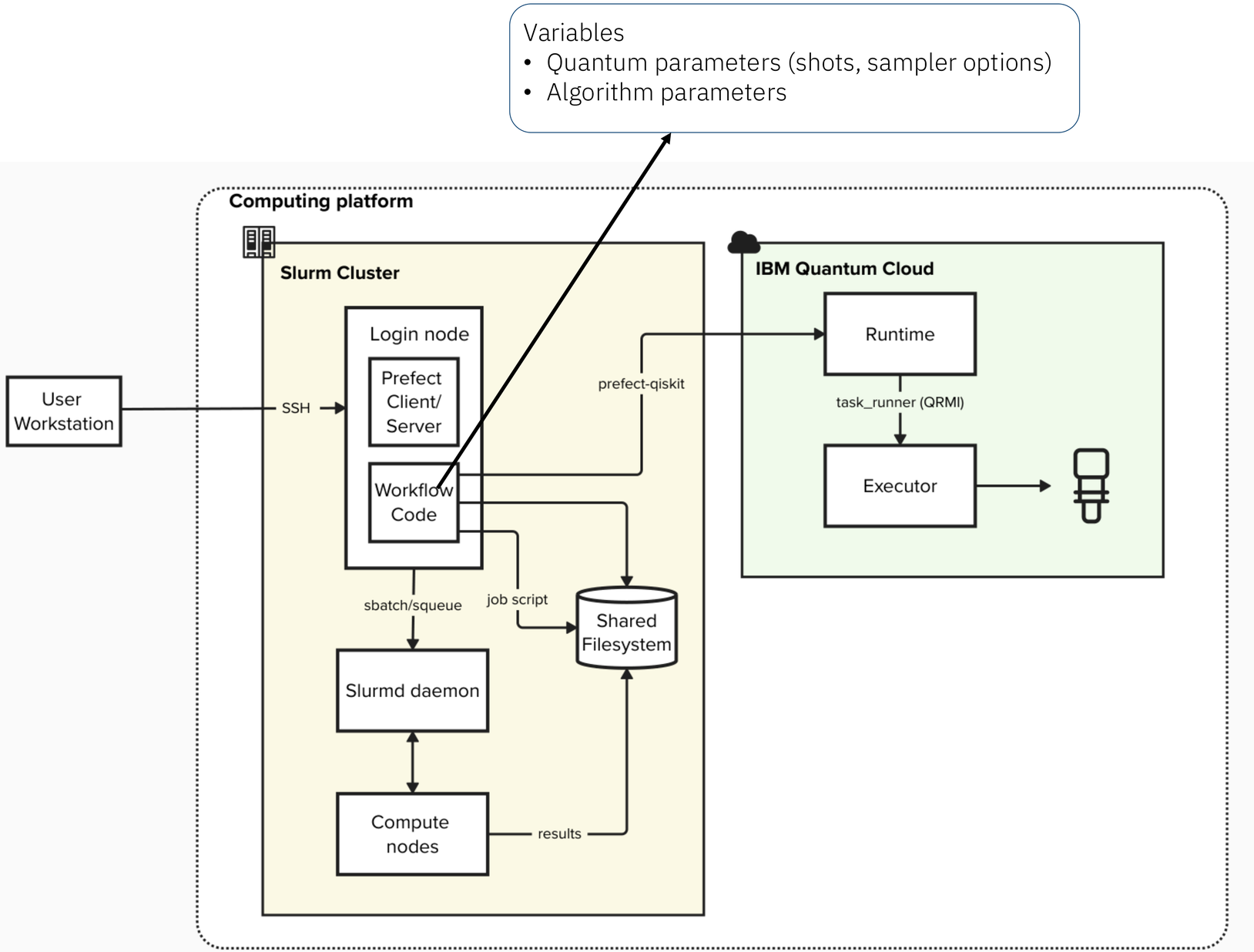
Prefect Core Concepts: Variables

Flow (end-to-end workflow)
- The Flow represents the full QCSC iteration

Tasks (individual execution steps)
- Sampling Task: submit a slurm job to run Quantum sampling job.
- HPC Task: submit a slurm job to run MPI programs.

Blocks (reusable configuration + credentials)
- Quantum Blocks: IBM Quantum credentials, runtime configuration, Slurm job configuration.
- HPC Blocks: Slurm job configuration.

Variables (run-time parameters)
- Quantum parameters: shots, circuit or sampler options.
- Algorithm parameters: iteration used by the workflow logic.



Prefect: All Together

Flow (end-to-end workflow)

- The Flow represents the full QCSC iteration

Tasks (individual execution steps)

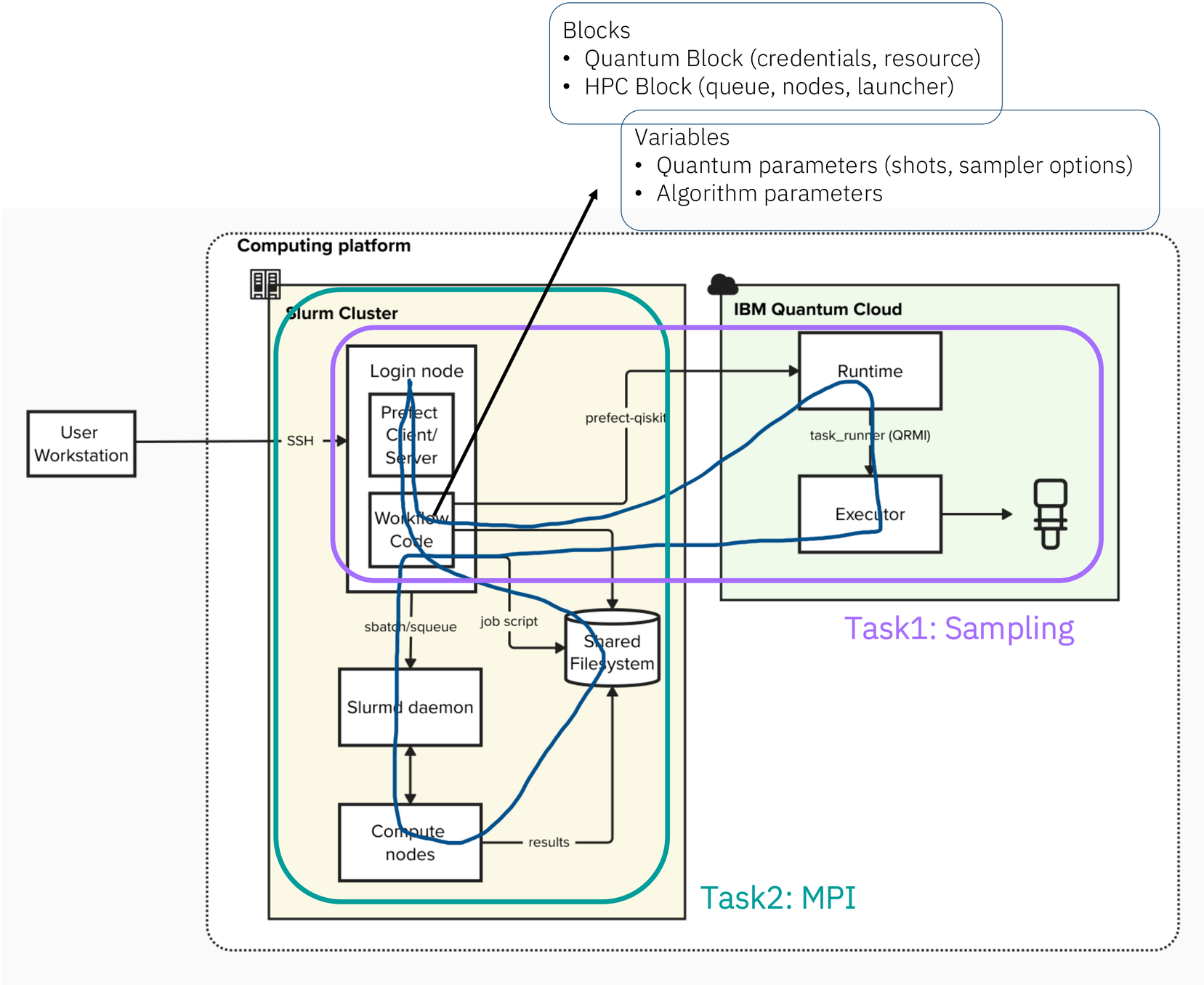
- Sampling Task: submit a slurm job to run Quantum sampling job.
- HPC Task: submit a slurm job to run MPI programs.

Blocks (reusable configuration + credentials)

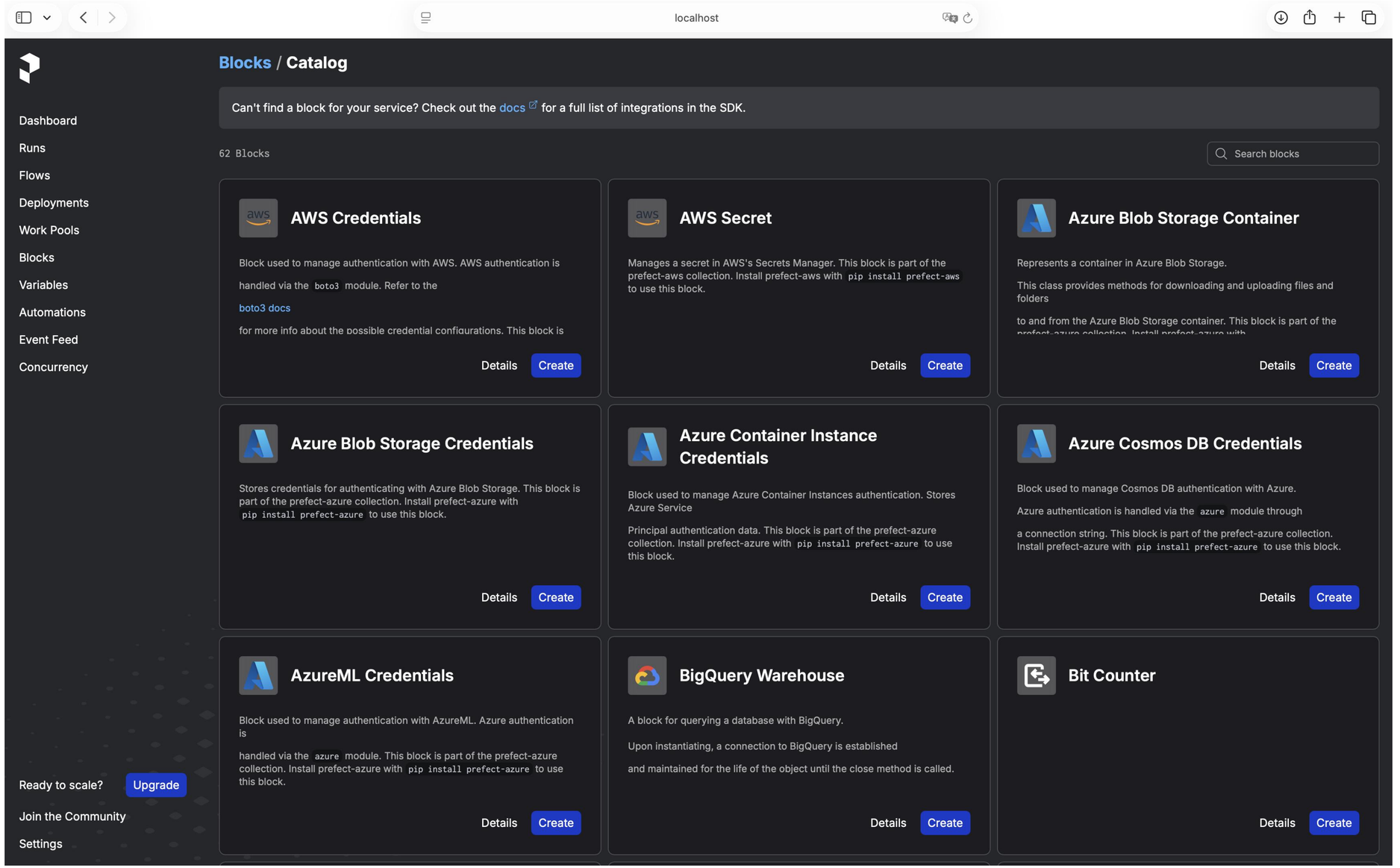
- Quantum Blocks: IBM Quantum credentials, runtime configuration, Slurm job configuration.
- HPC Blocks: Slurm job configuration.

Variables (run-time parameters)

- Quantum parameters: shots, circuit or sampler options.
- Algorithm parameters: iteration used by the workflow logic.



Using Prefect Blocks

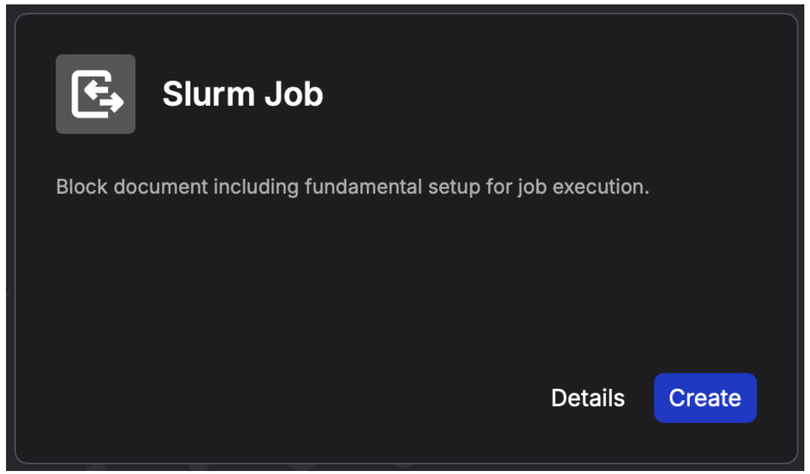


Prefect provides some pre-configured configurations

Creating custom Prefect block

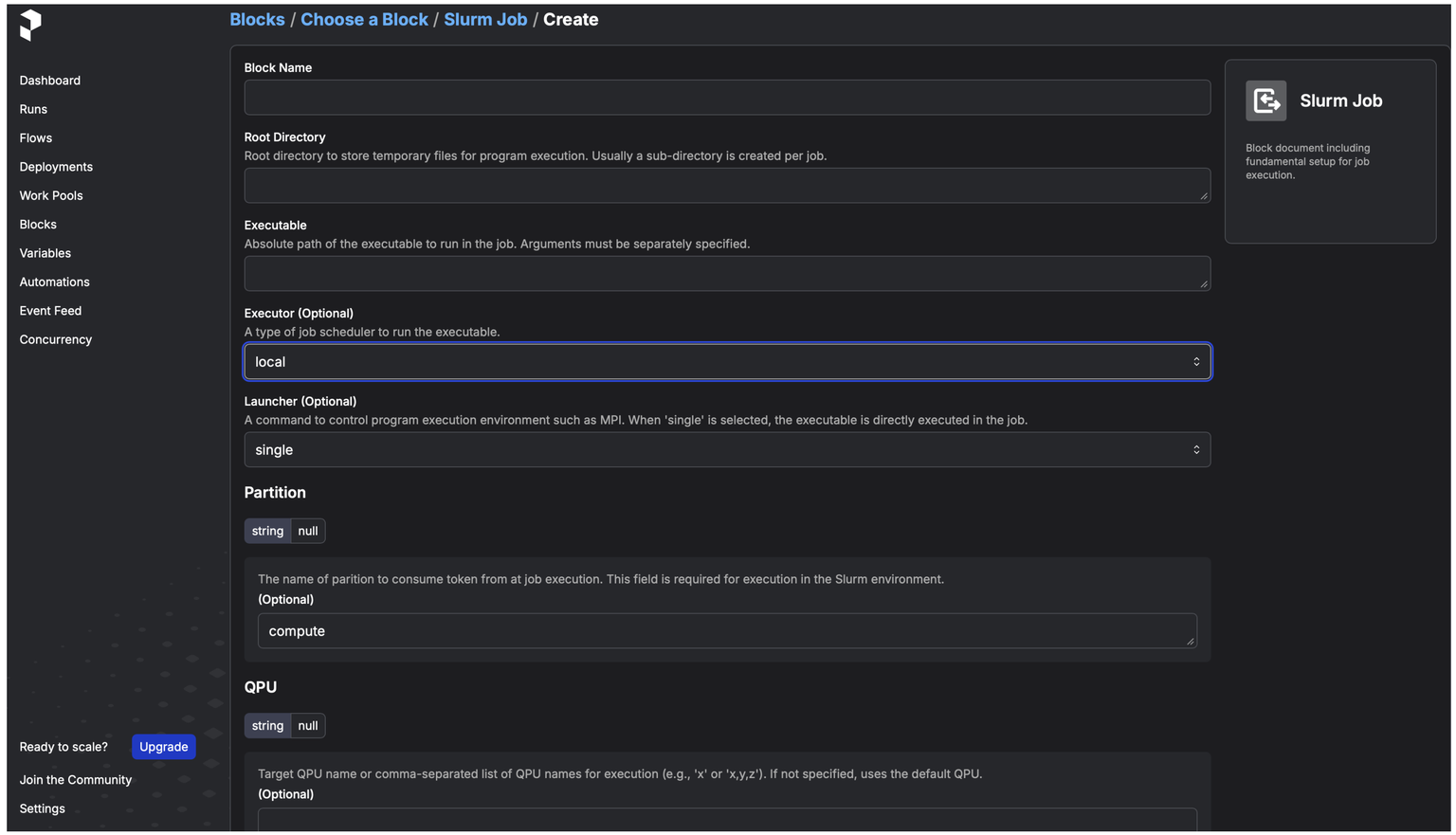
- class SlurmJobBlock:
  encapsulates slurm job configuration and execution logic

- Provides reusable interface to submit jobs to the slurm cluster




```python
"""Base block document for HPC job configuration."""

from pathlib import Path
from typing import Literal

from prefect.blocks.core import Block
from pydantic import Field

from .executor import BatchExecutor, JobExecutorBase, LocalExecutor


class SlurmJobBlock(Block):
    """Block document including fundamental setup for job execution."""

    _block_type_name = "Slurm Job"
    _block_type_slug = "slurm_job"

    work_root: str = Field(
        description=(
            "Root directory to store temporary files for program execution. Usually a sub-directory is created per job."
        ),
        title="Root Directory",
    )

    executable: str = Field(
        description="Absolute path of the executable to run in the job. Arguments must be separately specified.",
        title="Executable",
    )

    executor: Literal[
        "sbatch",
        "local",
    ] = Field(
        default="local",
        description="A type of job scheduler to run the executable.",
        title="Executor",
    )

    launcher: Literal[
        "single",
        "srun",
        "mpirun",
    ] = Field(
        default="single",
        description=(
            "A command to control program execution environment such as MPI. "
            "When 'single' is selected, the executable is directly executed in the job."
        ),
        title="Launcher",
    )
```

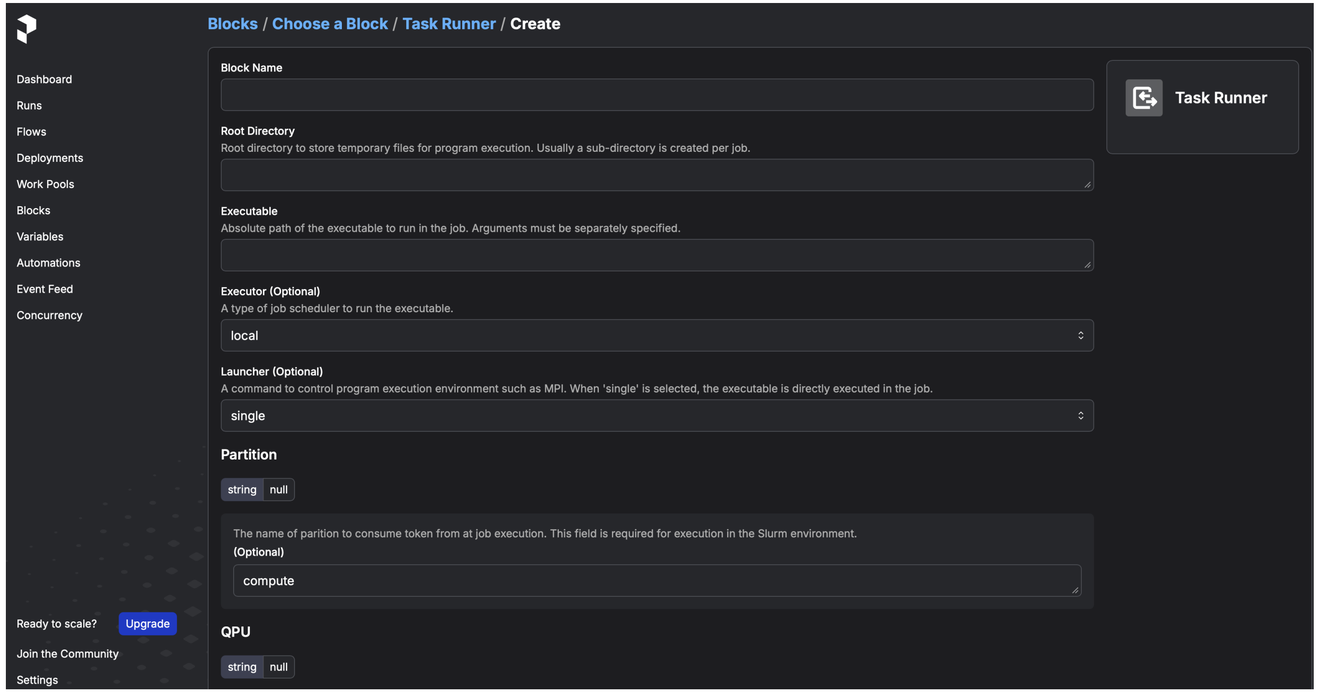
Custom Prefect block: Slurm Job




```python
"""Base block document for HPC job configuration."""

from pathlib import Path
from typing import Literal

from prefect.blocks.core import Block
from pydantic import Field

from .executor import BatchExecutor, JobExecutorBase, LocalExecutor


class SlurmJobBlock(Block):
    """Block document including fundamental setup for job execution."""

    _block_type_name = "Slurm Job"
    _block_type_slug = "slurm_job"

    work_root: str = Field(
        description=(
            "Root directory to store temporary files for program execution. Usually a sub-directory is created per job."
        ),
        title="Root Directory",
    )

    executable: str = Field(
        description="Absolute path of the executable to run in the job. Arguments must be separately specified.",
        title="Executable",
    )

    executor: Literal[
        "sbatch",
        "local",
    ] = Field(
        default="local",
        description="A type of job scheduler to run the executable.",
        title="Executor",
    )

    launcher: Literal[
        "single",
        "srun",
        "mpirun",
    ] = Field(
        default="single",
        description=(
            "A command to control program execution environment such as MPI. "
            "When 'single' is selected, the executable is directly executed in the job."
        ),
        title="Launcher",
    )
```

Custom Prefect block: Quantum Job

- class TaskRunner:
  encapsulates the execution of quantum circuits as jobs managed by task_runner (QRMI)

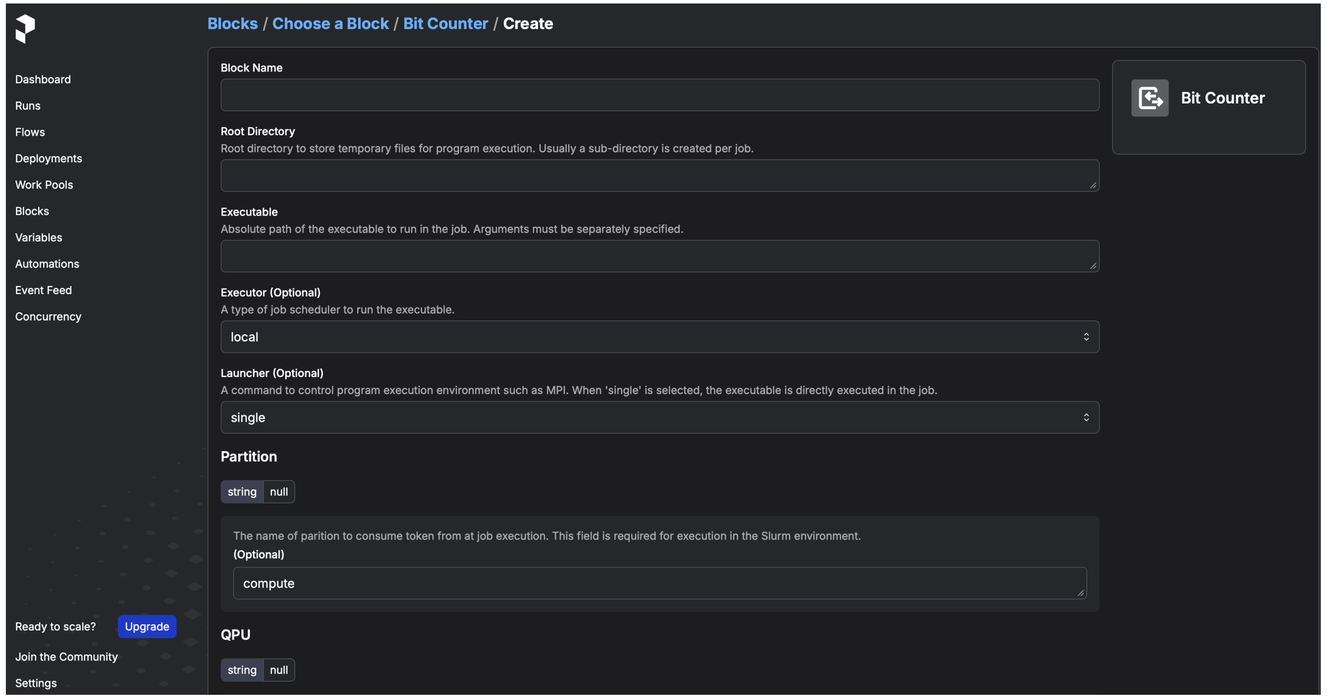



```python
import json
import numpy as np
from prefect import task
from prefect_slurm import SlurmJobBlock
from pydantic import Field

class TaskRunner(SlurmJobBlock):

    _block_type_name = "Task Runner"
    _block_type_slug = "task-runner"

    backend_name: str = Field(
        description="Backend name passed to task_runner as the first
argument.",
        title="Backend Name",
    )

    input_file: str = Field(
        default="input.json",
        description="Input JSON file name.",
        title="Input JSON File",
    )

    output_file: str = Field(
        default="output.json",
        description="Output JSON file name.",
        title="Output JSON File",
    )

    async def run(self):
        return await run_inner(self)
```

```python
@task(name="run_task_runner")
async def run_inner(job: TaskRunner):
    with job.get_executor() as executor:
        input_path = executor.work_dir / job.input_file
        output_path = executor.work_dir / job.output_file

        args: List[str] = [
            job.backend_name,
            str(input_path),
            str(output_path),
        ]
        exit_code = await executor.execute_job(
            arguments=args,
            **job.get_job_variables(),
        )
```

Custom Prefect block: MPI Job

- class BitCounter:
  encapsulates the logic for counting bitstrings using MPI

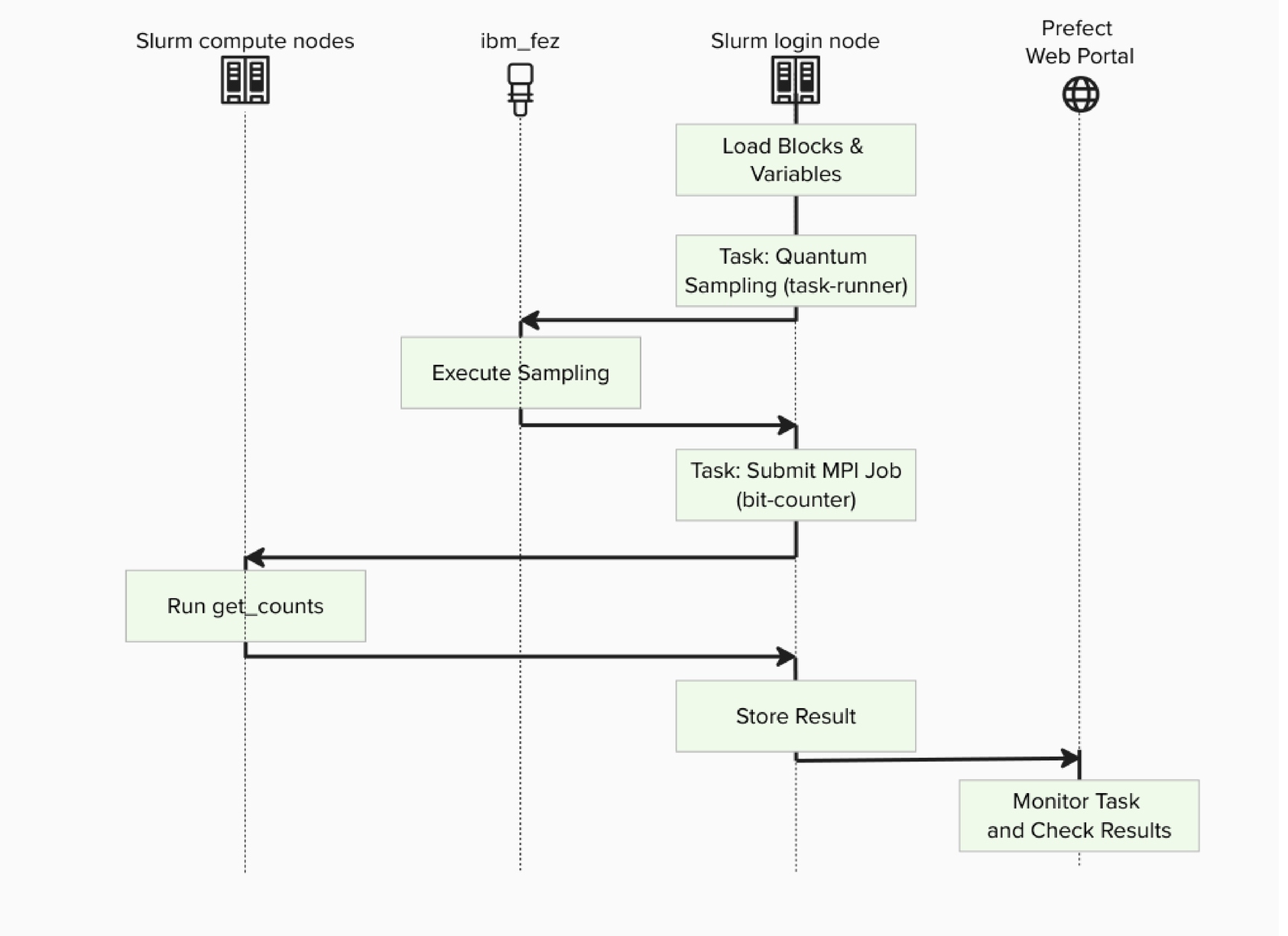



```python
import json
import numpy as np
from prefect import task
from prefect_slurm import SlurmJobBlock

BITLEN = 10

class BitCounter(SlurmJobBlock):

    _block_type_name = "Bit Counter"
    _block_type_slug = "bit-counter"

    async def get(
        self,
        bitstrings: list[str],
    ) -> dict[str, int]:
        return await get_inner(self, bitstrings)

@task(name="get_counts_mpi")
async def get_inner(
    job: BitCounter,
    bitstrings: list[str],
) -> dict[str, int]:
    with job.get_executor() as executor:
        # Write file
        u32int_array = np.array(
            [int(b, base=2) for b in bitstrings],
            dtype=np.uint32,
        )
        u32int_array.tofile(executor.work_dir / "input.bin")

        # Run MPI program
        exit_code = await executor.execute_job(**job.get_job_variables())
        assert exit_code == 0
```

QCSC Workflow: BitCount

```python
import asyncio
import json
from prefect import flow
from prefect.artifacts import create_table_artifact
from prefect.variables import Variable
from prefect_qiskit import QuantumRuntime
from qiskit import QuantumCircuit
from qiskit.transpiler import generate_preset_pass_manager
from get_counts_integration import BitCounter
from qiskit.primitives.containers.sampler_pub import SamplerPub
from qiskit import qasm3
from get_task_runner import TaskRunner
from qiskit_ibm_runtime.utils import RuntimeEncoder
from qiskit_ibm_runtime.utils.result_decoder import ResultDecoder

BITLEN = 10

@flow(name="slurm_tutorial")
async def main():
    # Load configurations
    runtime = await QuantumRuntime.load("ibm-runner")
    counter = await BitCounter.load("slurm-tutorial")
    options = await Variable.get("slurm-tutorial")
    taskrunner = await TaskRunner.load("slurm-tutorial")

    # Create a PUB payload
    target = await runtime.get_target()
    qc_ghz = QuantumCircuit(BITLEN)
    qc_ghz.h(0)
    qc_ghz.cx(0, range(1, BITLEN))
    qc_ghz.measure_active()

    pm = generate_preset_pass_manager(
        optimization_level=3,
        target=target,
        seed_transpiler=123,
    )
    isa = pm.run(qc_ghz)
    pub_like = (isa,)  # Create a Primitive Unified Bloc

    # Extract shots
    shots = options.get("shots", 100000) # default to 100000 if not set

    dict_pubs = []

    # Create input.json for task_runner
    coerced_pub = SamplerPub.coerce(pub_like, shots=shots)

    # Generate OpenQASM3 string which can be consumed by IBM Quantum APIs
    qasm3_str = qasm3.dumps(
            coerced_pub.circuit,
            disable_constants=True,
            allow_aliasing=True,
            experimental=qasm3.ExperimentalFeatures.SWITCH_CASE_V1,
    )

    if len(coerced_pub.circuit.parameters) == 0:
        if coerced_pub.shots:
            dict_pubs.append((qasm3_str, None, coerced_pub.shots))
        else:
            dict_pubs.append((qasm3_str))
    else:
        param_array = coerced_pub.parameter_values.as_array(
        coerced_pub.circuit.parameters
                ).tolist()

        if coerced_pub.shots:
            dict_pubs.append((qasm3_str, param_array, coerced_pub.shots))
        else:
            dict_pubs.append((qasm3_str, param_array))

    # Create SamplerV2 input
    input_json = {
            "pubs": dict_pubs,
            "shots": shots,
            "options": options,
            "version": 2,
            "support_qiskit": True,
    }
    print("Here is the input json:", input_json)

    taskrunner_json = {"parameters": input_json, "program_id": "sampler"}
    print("Here is the taskrunner json:", taskrunner_json)

    filename = "/mnt/data/salaria/slurm_tutorial/input.json"
    with open(filename, "w", encoding="utf-8") as primitive_input_file:
        json.dump(taskrunner_json, primitive_input_file, cls=RuntimeEncoder, indent=2)

    # Quantum execution
    result = await taskrunner.run()

    # Read output
    with open('/mnt/data/salaria/slurm_tutorial/output.json', 'r') as f:
        results = ResultDecoder.decode(f.read())

    # MPI execution
    bitstrings = results[0].data.meas.get_bitstrings()
    counts = await counter.get(bitstrings)

    # Save in Prefect artifacts
    await create_table_artifact(
        table=[list(counts.keys()), list(counts.values())],
        key="sampler-count-dict",
     )


if __name__ == "__main__":
    asyncio.run(main())
```

Prefect Workflow: BitCount



BitCount – Load Configurations

- Line 21: Quantum runtime settings referencing credentials
- Line 22: MPI block
- Line 23: Variables
- Line 24: Quantum block

```
1  import asyncio
2  import json
3  from prefect import flow
4  from prefect.artifacts import create_table_artifact
5  from prefect.variables import Variable
6  from prefect_qiskit import QuantumRuntime
7  from qiskit import QuantumCircuit
8  from qiskit.transpiler import generate_preset_pass_manager
9  from get_counts_integration import BitCounter
10 from qiskit.primitives.containers.sampler_pub import SamplerPub
11 from qiskit import qasm3
12 from get_task_runner import TaskRunner
13 from qiskit_ibm_runtime.utils import RuntimeEncoder
14 from qiskit_ibm_runtime.utils.result_decoder import ResultDecoder
15
16 BITLEN = 10
17
18 @flow(name="slurm_tutorial")
19 async def main():
20     # Load configurations
21     runtime = await QuantumRuntime.load("ibm-runner")
22     counter = await BitCounter.load("slurm-tutorial")
23     options = await Variable.get("slurm-tutorial")
24     taskrunner = await TaskRunner.load("slurm-tutorial")
25
26     # Create a PUB payload
27     target = await runtime.get_target()
28     qc_ghz = QuantumCircuit(BITLEN)
29     qc_ghz.h(0)
30     qc_ghz.cx(0, range(1, BITLEN))
31     qc_ghz.measure_active()
```

BitCount – Prepare Quantum Circuit

- Line 26-39: Define quantum circuit and transpile

```
26      # Create a PUB payload
27      target = await runtime.get_target()
28      qc_ghz = QuantumCircuit(BITLEN)
29      qc_ghz.h(0)
30      qc_ghz.cx(0, range(1, BITLEN))
31      qc_ghz.measure_active()
32
33      pm = generate_preset_pass_manager(
34          optimization_level=3,
35          target=target,
36          seed_transpiler=123,
37      )
38      isa = pm.run(qc_ghz)
39      pub_like = (isa,)  # Create a Primitive Unified Bloc
```

BitCount – Prepare Quantum Environment

- Line 46-87: Create input file for task_runner (QRMI)

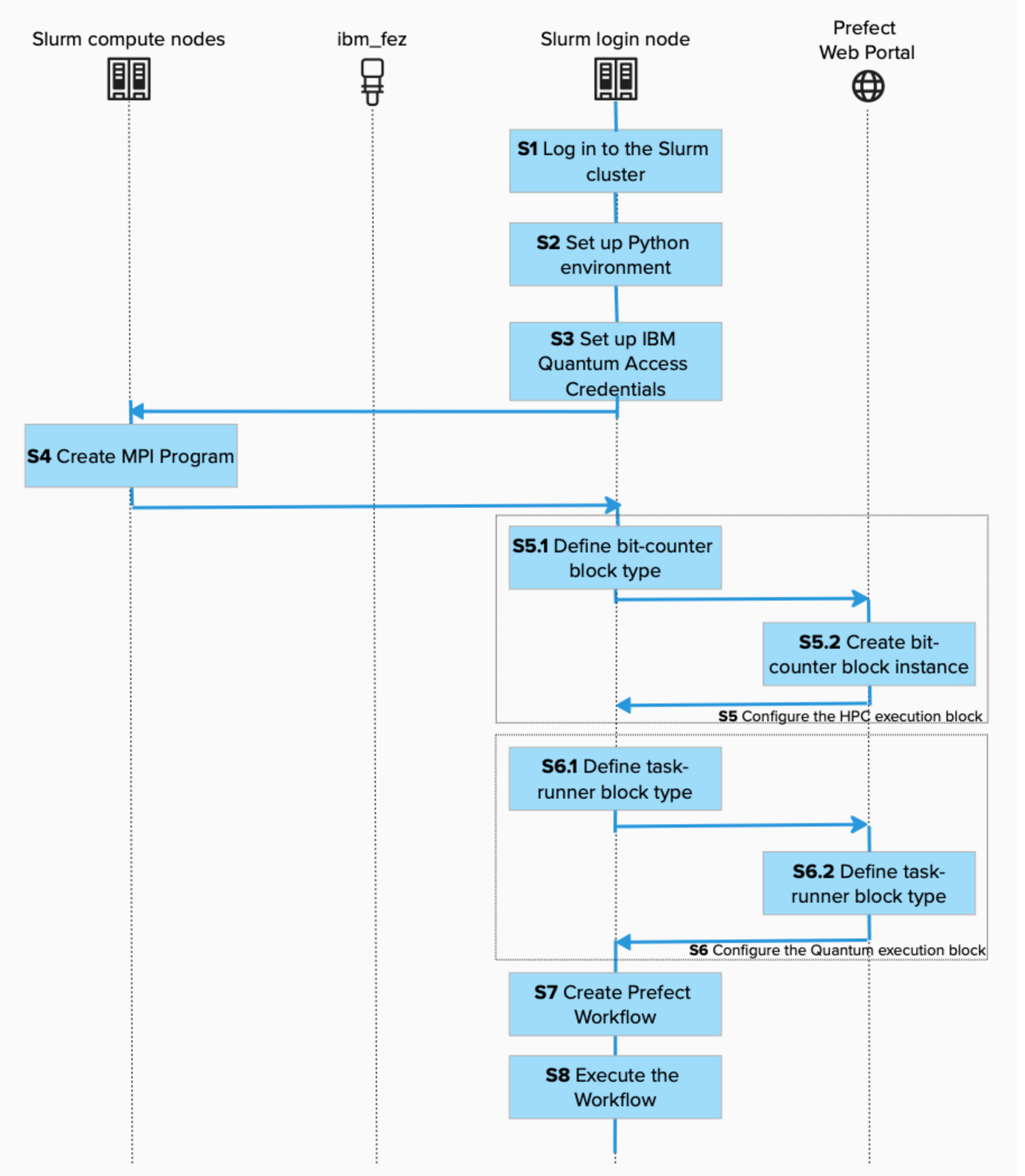
```
46      # Create input.json for task_runner
47      coerced_pub = SamplerPub.coerce(pub_like, shots=shots)
48
49      # Generate OpenQASM3 string which can be consumed by IBM Quantum APIs
50      qasm3_str = qasm3.dumps(
51              coerced_pub.circuit,
52              disable_constants=True,
53              allow_aliasing=True,
54              experimental=qasm3.ExperimentalFeatures.SWITCH_CASE_V1,
55      )
56
57      if len(coerced_pub.circuit.parameters) == 0:
58          if coerced_pub.shots:
59              dict_pubs.append((qasm3_str, None, coerced_pub.shots))
60          else:
61              dict_pubs.append((qasm3_str))
62      else:
63          param_array = coerced_pub.parameter_values.as_array(
64          coerced_pub.circuit.parameters
65                  ).tolist()
66
67          if coerced_pub.shots:
68              dict_pubs.append((qasm3_str, param_array, coerced_pub.shots))
69          else:
70              dict_pubs.append((qasm3_str, param_array))
71
72      # Create SamplerV2 input
73      input_json = {
74              "pubs": dict_pubs,
75              "shots": shots,
76              "options": options,
77              "version": 2,
78              "support_qiskit": True,
79      }
80      print("Here is the input json:", input_json)
81
82      taskrunner_json = {"parameters": input_json, "program_id": "sampler"}
83      print("Here is the taskrunner json:", taskrunner_json)
84
85      filename = "/mnt/data/salaria/slurm_tutorial/input.json"
86      with open(filename, "w", encoding="utf-8") as primitive_input_file:
87          json.dump(taskrunner_json, primitive_input_file, cls=RuntimeEncoder, indent=2)
```

BitCount – Execute blocks and save results

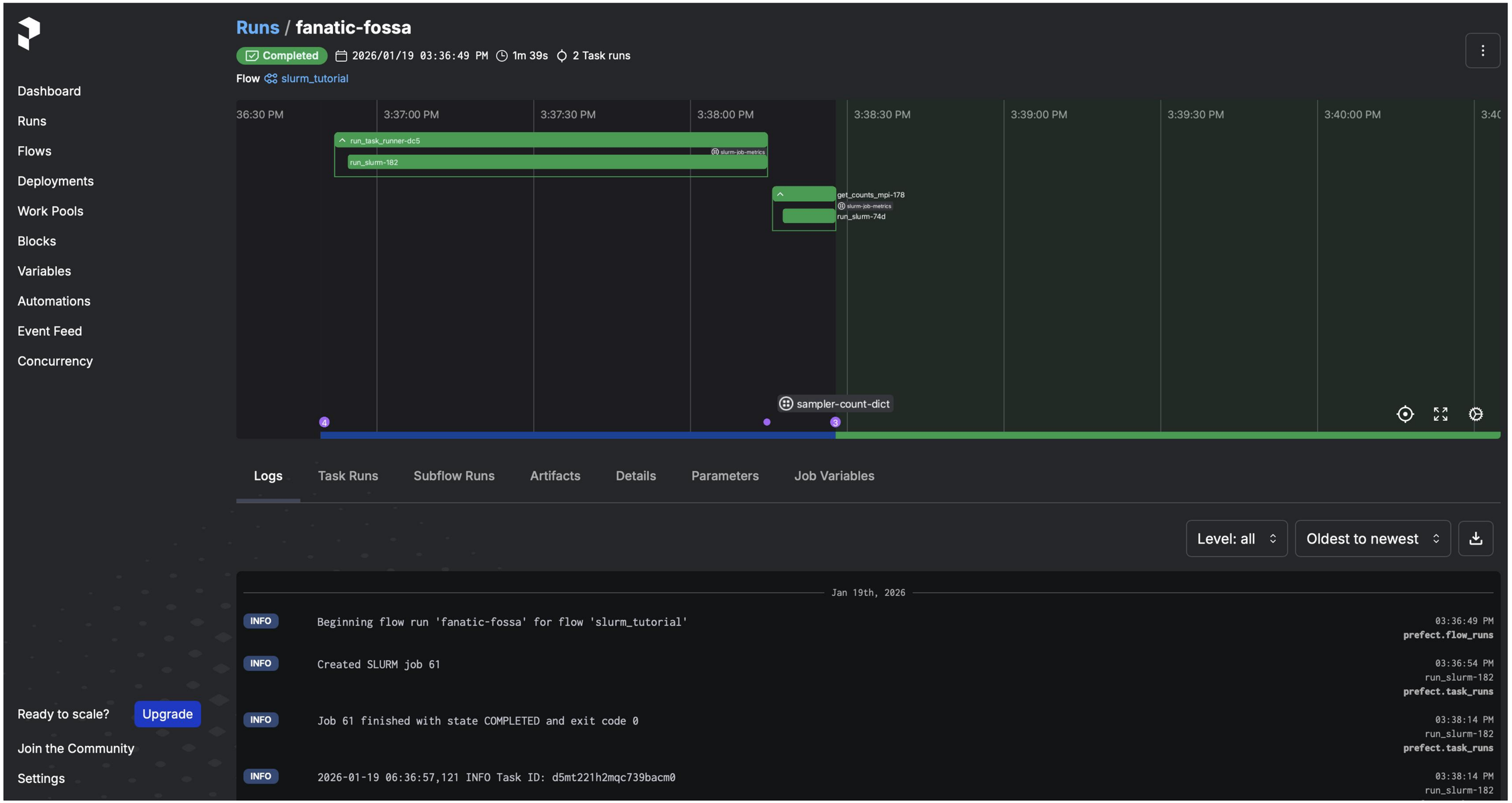
- Line 90: Quantum job execution
- Line 93-94: Get quantum result and decode
- Line 97-98: MPI job execution
- Line 101-104: save result as sampler-count-dict

```
 88
 89      # Quantum execution
 90      result = await taskrunner.run()
 91
 92      # Read output
 93      with open('/mnt/data/salaria/slurm_tutorial/output.json', 'r') as f:
 94          results = ResultDecoder.decode(f.read())
 95
 96      # MPI execution
 97      bitstrings = results[0].data.meas.get_bitstrings()
 98      counts = await counter.get(bitstrings)
 99
100      # Save in Prefect artifacts
101      await create_table_artifact(
102          table=[list(counts.keys()), list(counts.values())],
103          key="sampler-count-dict",
104       )
105
106
107  if __name__ == "__main__":
108      asyncio.run(main())
109
```

BitCount Demo



BitCount Run



Tutorial Files

https://github.com/ohtanim/SCA-HPCAsia-2026