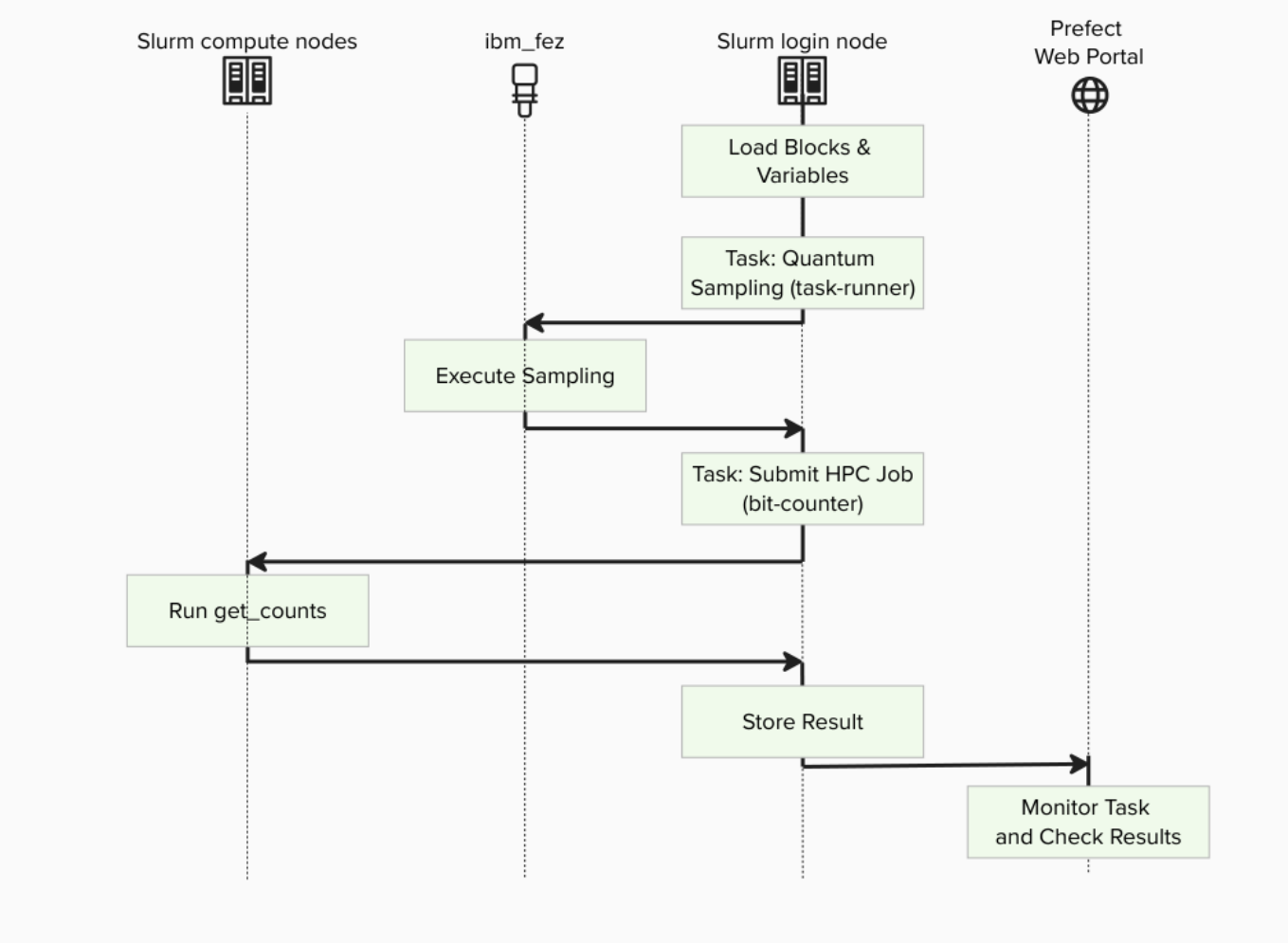
Create Your QCSC Workflow with Prefect

This hands-on tutorial guides you through building a small C++ program on a Slurm cluster and integrating it into a Prefect workflow using a custom `SlurmJobBlock` class. On the Prefect workflow, we also use Prefect Qiskit to show how to write a complete QCSC workflow from scratch.

Our objective is to compute a count dictionary of sampler bitstrings using MPI programming on the QCSC architecture.

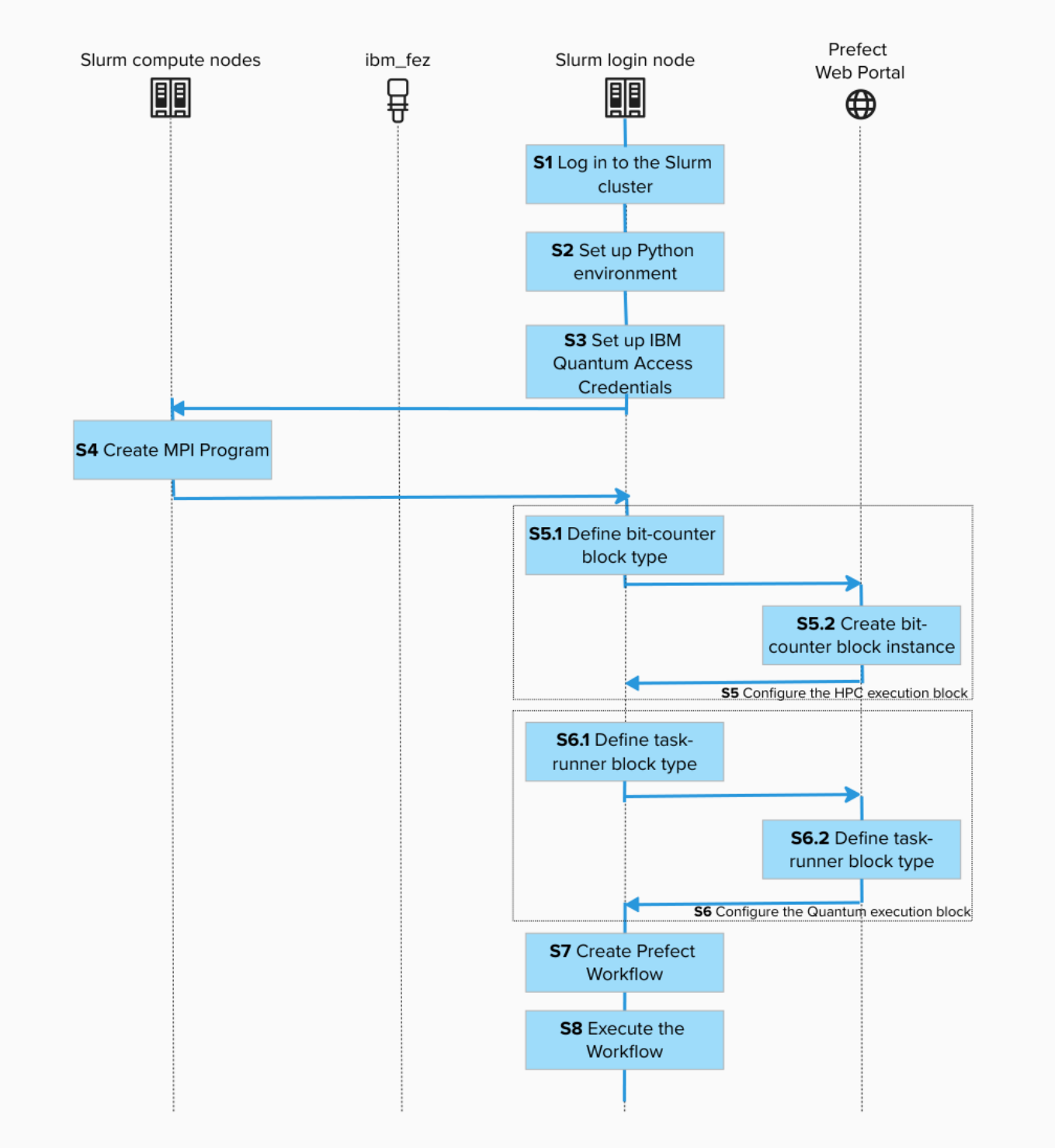


Prefect Core Concepts

We will use these terms:

- **Flow**: the end-to-end workflow defined in sampler_workflow_qrmi.py
- **Task**: individual steps inside the flow (e.g., runtime.sampler(...), counter.get(...))
- **Block**: reusable configuration + credentials stored in Prefect server
  - `ibm-quantum-credentials` : IBM Cloud CRN + API key
  - `quantum-runtime` : runtime settings referencing credentials
  - `bit-counter` : HPC job configuration (queue, nodes, executable path, modules)
  - `task-runner`: Quantum job configuration (executable, output)
- **Variable**: run-time parameter stored server-side (sampler shots etc.)

Create BitCounts Workflow



Step 1. Log in to the Slurm Cluster

Connect to the Slurm cluster login node using SSH. This is where we will develop the workflow.

PC

```
ssh -J salaria@150.240.167.86 salaria@10.240.0.37
```

Step 2. Set up Python environment

Create a project directory:

Slurm

```
mkdir /mnt/data/salaria/slurm_tutorial && cd
/mnt/data/salaria/slurm_tutorial
```

Create a virtual environment and activate:

Slurm

```
uv venv -p 3.12 && source .venv/bin/activate
```

Install necessary packages:

Slurm

```
uv pip install prefect-qiskit
uv pip install git+ssh://git@github.com:shwetasalaria/qii-miyabi-
kawasaki.git@main#subdirectory=framework/prefect-slurm
```

Check installations:

Slurm

```
uv pip list | grep prefect
```

You should see output like:

```
prefect                          3.6.9
prefect-qiskit                   0.2.0
prefect-slurm                    0.1.0
```

Step 3. Set up IBM Quantum Access Credentials

Make sure the virtual environment is activated

```
source .venv/bin/activate
```

Create and switch to a new Prefect profile:

Slurm

```
prefect profile create sca26 && prefect profile use sca26
```

Prefect server is configured on the login node at port 4200. Set the config as:

Slurm

```
prefect config set PREFECT_API_URL=http://127.0.0.1:4200/api
```

Register the block schemas for Qiskit integration:

Slurm

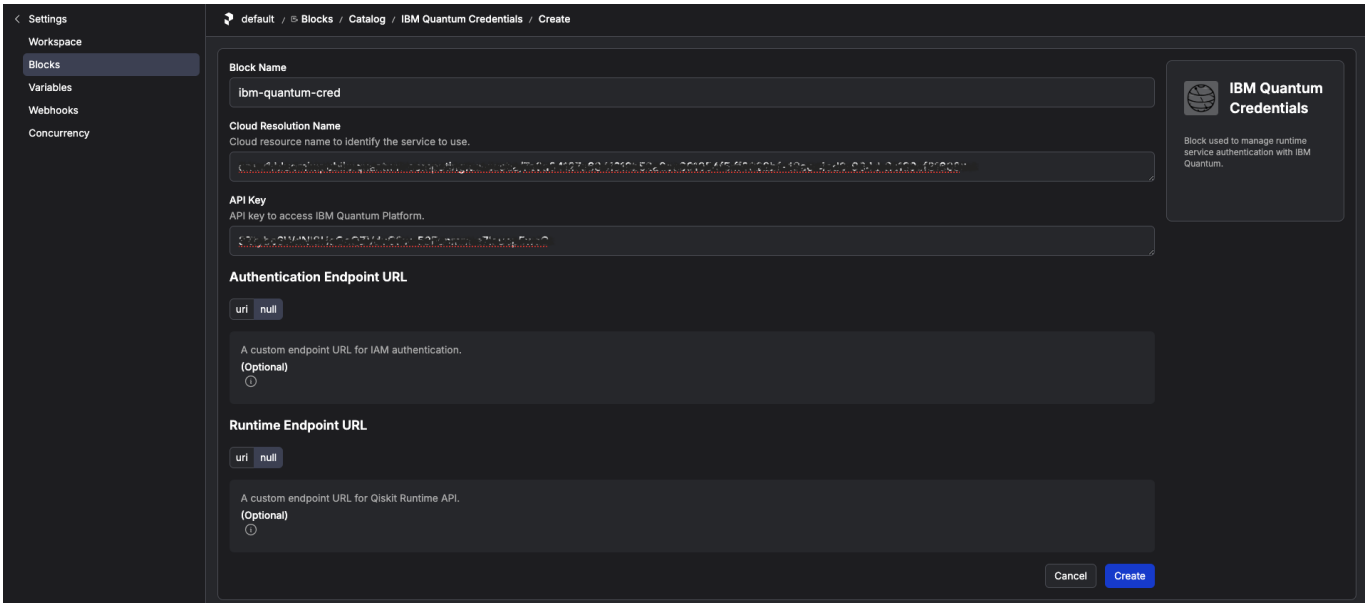
```
prefect block register -m prefect_qiskit
prefect block register -m prefect_qiskit.vendors
```

Create the IBM Quantum Credentials block:

Slurm

```
prefect block create ibm-quantum-credentials
```

This command will display a URL to the Prefect console. Open it in your browser and enter your IBM Quantum instance's CRN and API Key to create the block.

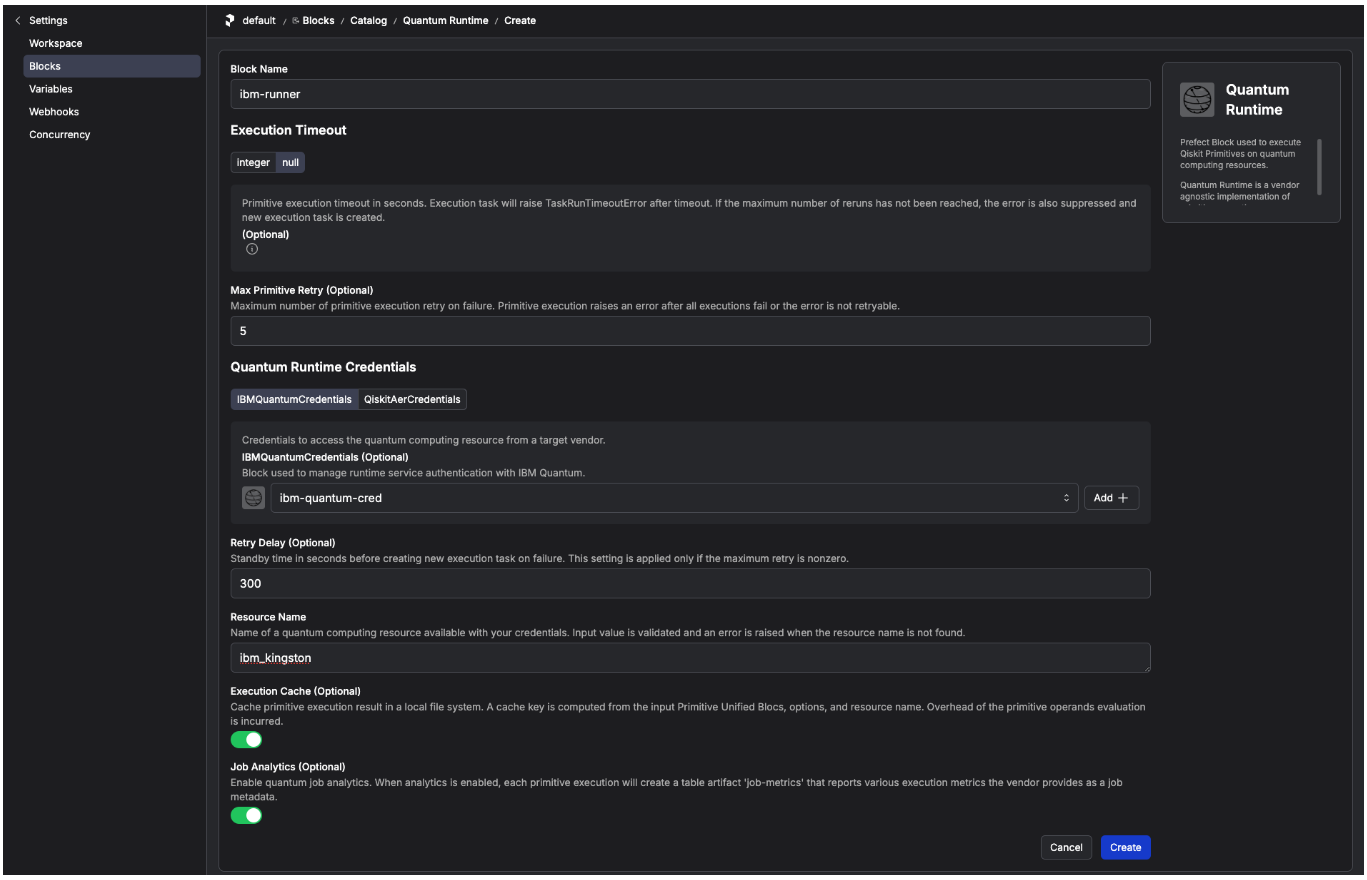


Then, enter the following:

Slurm

```
prefect block create quantum-runtime
```

Follow the URL shown to configure the runtime block. Specify the IBM Quantum backend name and link the credentials block you created above. You can also configure preferences for Qiskit primitive execution.



Confirm you have access to the blocks you created:

Slurm

```
prefect block ls
```

Example output:

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━┳━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┓
┃ ID                                   ┃ Type        ┃ Name       ┃ Slug                   ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━┩
```

```
| 9d87e2a8-e7b8-4e3b-98… | IBM Quan… | ibm-quantu… | ibm-quantum-
credentials/ibm-qua… |
| 8c9e4ff7-b09a-4f11-bc… | Quantum … | ibm-runner  | quantum-runtime/ibm-
runner       |
└──────────────────────────┴────────────┴──────────────┴──────────────────────────
──────────────┘
```

Step 4. Create MPI Program

Maske sure you are in your your work directory:

Slurm

```
cd /mnt/data/salaria/slurm_tutorial
```

Open a C++ source code file:

Slurm

```
vi get_counts.cpp
```

Add following lines to the file:

```
#include <mpi.h>
#include <fstream>
#include <vector>
#include <iostream>

const uint32_t BITLEN = 10;
const uint32_t MAXVAL = 1 << BITLEN;

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    uint32_t total_count = 0;
    std::vector<uint32_t> data;

    if (rank == 0) {
        std::ifstream fin("input.bin", std::ios::binary | std::ios::ate);
        total_count = fin.tellg() / sizeof(uint32_t);
        fin.seekg(0, std::ios::beg);
```

```
        data.resize(total_count);
        fin.read(reinterpret_cast<char*>(data.data()), total_count *
sizeof(uint32_t));
        fin.close();
    }

    MPI_Bcast(&total_count, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);

    uint32_t local_n = total_count / size;
    std::vector<uint32_t> local_data(local_n);

    MPI_Scatter(rank == 0 ? data.data() : nullptr, local_n, MPI_UNSIGNED,
                local_data.data(), local_n, MPI_UNSIGNED,
                0, MPI_COMM_WORLD);

    std::vector<int> local_hist(MAXVAL, 0);
    for (auto v : local_data) {
        if (v < MAXVAL) local_hist[v]++;
    }

    std::vector<int> global_hist;
    if (rank == 0) global_hist.resize(MAXVAL, 0);

    MPI_Reduce(local_hist.data(),
               rank == 0 ? global_hist.data() : nullptr,
               MAXVAL, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        std::ofstream fout("output.json");
        fout << "{";
        bool first = true;
        for (uint32_t i = 0; i < MAXVAL; ++i) {
            if (global_hist[i] > 0) {
                if (!first) fout << ",";
                fout << "\"" << i << "\":" << global_hist[i];
                first = false;
            }
        }
        fout << "}\n";
        fout.close();
    }

    MPI_Finalize();
    return 0;
}
```

> [!NOTE] This program reads the `input.bin` file including a 32-bit integer vector of bitstrings, splits the data across MPI processes, and counts how often each value appears. Each process builds a local histgram from its share of the data. MPI then combines all local results into a single global histogram, which rank 0 writes out as `output.json`.

Check the Open MPI library is loaded in your shell:

Slurm

```
mpirun --version
```

By default, Open MPI 4.1.1 is available on all nodes in the cluster.

Since this program is lightweight, it's fine compiling on the login node:

Slurm

```
mpiicxx -o get_counts get_counts.cpp
```

Check the output file:

Slurm

```
ls -l
```

Example output:

```
-rwx------. 1 salaria salaria 130552 Jan  8 12:42 get_counts
-rw-------. 1 salaria salaria   1893 Jan  8 12:42 get_counts.cpp
```

Get the absolute path to the `get_counts` executable:

Slurm

```
realpath ./get_counts
```

Example output:

```
/mnt/data/salaria/slurm-tutorial/get_counts
```

We will need this path in the later step.

Step 5. Configure Prefect block for MPI task

Step 5.1 Prefect Block for MPI task

In this step, we define a Block Type (“bit-counter”) in Python. This is the template/schema that tells Prefect what fields the block has and how it runs `get_counts` on the cluster.

In the project directory (`/mnt/data/salaria/slurm_tutorial`), open a new Python file:

Slurm

```
vi get_counts_integration.py
```

Add following lines to the file:

```
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

        # Read file
        with open(executor.work_dir / "output.json", "r") as f:
            int_counts = json.load(f)
```

```
        return {
            format(int(k), f"0{BITLEN}b"): v
            for k, v in int_counts.items()
        }
```

> [!NOTE] The `SlurmJobBlock` baseclass implements the mechanism to interact with the Slurm job scheduler. A subclass must implement the data input and output. The `get_inner` function is a trick to turn HPC job executions into Prefect Tasks.

Register the block schema from a file:

Slurm

```
prefect block register -f get_counts_integration.py
```

Step 5.2 Create `bit-counter` block instance

In this step, we create a Block Instance (e.g., “slurm-tutorial”). This is your environment-specific configuration, such as queue name, node count, and the executable path. Create a new configuration for the Bit Counter block:

Slurm

```
prefect block create bit-counter
```

This command will display a URL to the Prefect console. Open it in your browser and fill in the following fields:

Field	Value / Example
Block Name	`slurm-tutorial`
Root Directory	`/mnt/data/salaria/slurm_tutorial` (The absolute path to the `slurm_tutorial` directory)
Executable	`/mnt/data/salaria/slurm_tutorial/get_counts` (The absolute path to the `get_counts` executable)
Executor	`sbatch`
Launcher	`srun`
Num Nodes	`2`
Num MPI Processes	`5`

The other fields can be left blank.

Now this configuration is stored in the Prefect server and it can be used by many Prefect workflows.

Step 6 Configure Prefect block for Quantum task

Now we write a Prefect block to run the quantum sampling task:

Step 6.1 Define `task-runner` block type

In this step, we define a Block Type (“task-counter”) in Python. This is the template/schema that tells Prefect what fields the block has and how it runs quantum job on quantum computer.

In the project directory (`~/slurm_tutorial`), open a new Python file:

Slurm

```
vi get_task_runner.py
```

Add following lines to the file:

```
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)
```

```
@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(),
        )
```

> [!NOTE] The `SlurmJobBlock` baseclass implements the mechanism to interact with the Slurm job scheduler. The `TaskRunnerJobBlock` subclass defines backend, input and output.

Register the block schema from a file:

Slurm

```
prefect block register -f get_task_runner.py
```

Step 6.2 Create `task-runner` block instance

Create a new configuration for the Task Runner block:

Slurm

```
prefect block create task-runner
```

Fill the fields (input, output, backend, executable) in the browser.

This configuration is stored in the Prefect server and it can be used by many Prefect workflows.

Step 7. Create Prefect Workflow

Next, in the same directory, create a separate Python file to define the Prefect workflow:

Slurm

```
vi sampler_workflow_qrmi.py
```

Add following lines to the file:

```
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
```

```
    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)
    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()
```

```
    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())

```

Step 8. Execute the workflow

Make sure the Quantum Runtime block exist:

Slurm

```
prefect block inspect quantum-runtime/ibm-runner
```

The output may look like:

```
┌──────────────────────┬──────────────────────────────────────────────┐
│ Block Type           │ Quantum Runtime                              │
│ Block id             │ b2047dc9-e90e-4930-be6c-269478a4d6b4         │
├──────────────────────┼──────────────────────────────────────────────┤
│ resource_name        │ ibm_kawasaki                                 │
│ execution_cache      │ True                                         │
│ enable_job_analytics │ True                                         │
│ credentials          │ {'crn':                                      │
│                      │ 'crn:v1:bluemix:public:quantum-computing:u… │
│                      │ 'api_key': '********'}                       │
└──────────────────────┴──────────────────────────────────────────────┘
```

Set the sampler options for the IBM Qiskit Runtime API:

Slurm

```
prefect variable set miyabi-tutorial '{"options": {"shots": 100000}}' --
overwrite
```

Verify the variable:

mdx

```
prefect variable inspect slurm-tutorial
```

Example output:

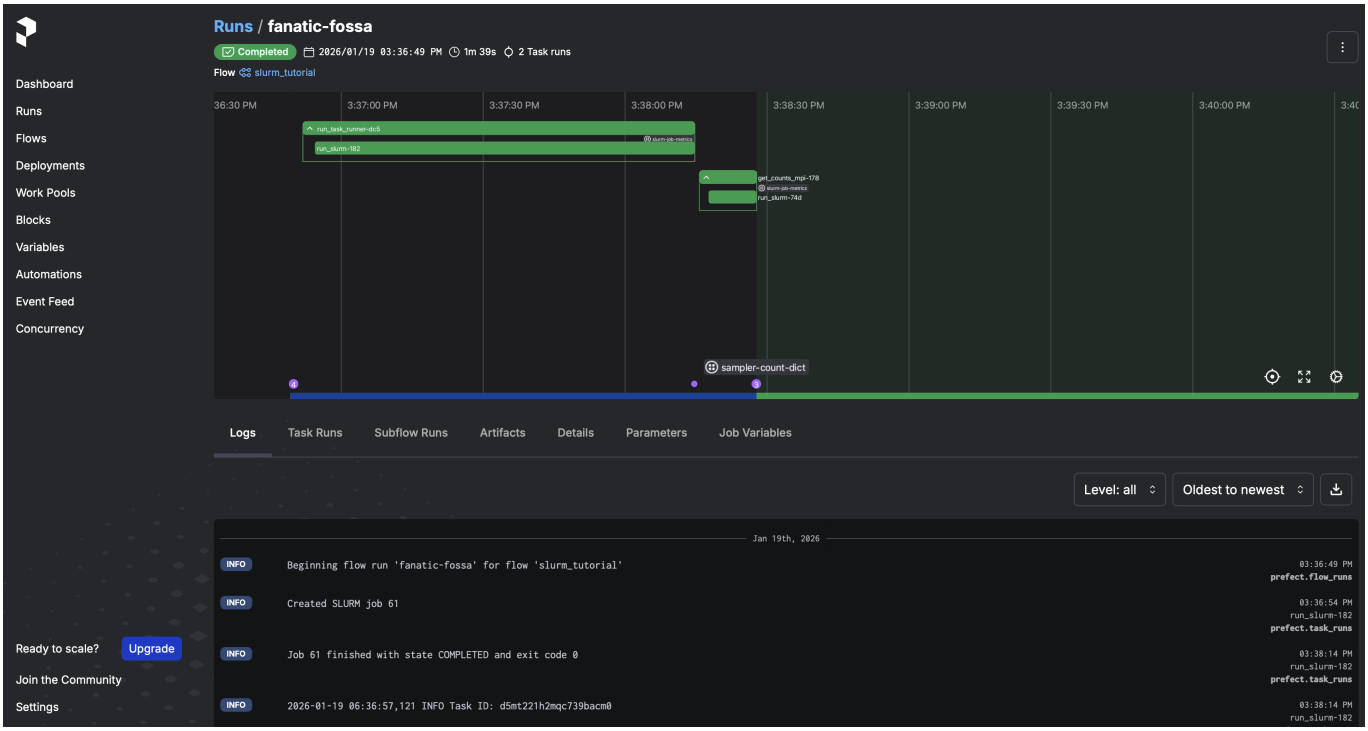
```
Variable(
    id='a6c87c02-681e-4106-8266-a6857028eac8',
    created=DateTime(2025, 9, 30, 6, 24, 49, 19172,
tzinfo=Timezone('UTC')),
    updated=DateTime(2025, 9, 30, 6, 24, 49, 19194,
tzinfo=Timezone('UTC')),
    name='slurm-tutorial',
    value={'options': {'shots': 100000}},
    tags=[]
)
```

To execute the workflow, run the following Python script:

mdx

```
python sampler_workflow_qrmi.py
```

We can also monitor the progress on the Prefect console:



Upon successful completion of the workflow, Prefect will generate the following artifacts:

- `sampler-count-dict`: Count dictionary computed by our MPI program.
- `job-metrics`: Performance metrics of IBM primitive execution.

- `slurm-job-metrics`: Performance metrics of Slurm job execution.

See the official Artifacts guide about Prefect artifacts. Example data is available in here.

The metrics artifacts are automatically generated by Prefect integration libraries. This information might be useful to optimize computing resources.

> [!NOTE] Note that this example does not significantly benefit from MPI parallel execution, as data input and output on the rank 0 process is the dominant performance bottleneck. This example is chosen to demonstrate how MPI programs look like.

END OF TUTORIAL