Qorchestration-CERN Codebase Overview

Modular quantum workflow orchestration with Streamlit, Kubernetes and Argo Workflows

Leonardo Lavagna

Journal club
21/05/2026

Presentation goal

Explain the concrete logic of the current `qorchestration-v0.2` codebase: how tasks are defined, submitted, executed, archived, and analyzed for Circuit Cutting and SQD workflows.

Executive summary

- **Purpose:** one user-facing console for launching quantum workflows on a Kubernetes cluster.
- **Two first-class modules:** Circuit Cutting and Sample-based Quantum Diagonalization (SQD).
- **Shared core:** cluster validation, Argo submission, runtime handling, artifact archival, QPU secret setup.
- **Execution model:** Streamlit task builder $\rightarrow$ Argo Workflow $\rightarrow$ CPU/GPU/QPU pods $\rightarrow$ PVC artifacts $\rightarrow$ local archive/analysis.

What the codebase demonstrates

- Modular integration, not a monolithic script.
- Backend-aware workflow specifications.
- Reusable launch/archive/analyze path.
- Early validation tests for structure and task generation.

Executive summary

- **Purpose:** one user-facing console for launching quantum workflows on a Kubernetes cluster.
- **Two first-class modules:** Circuit Cutting and Sample-based Quantum Diagonalization (SQD).
- **Shared core:** cluster validation, Argo submission, runtime handling, artifact archival, QPU secret setup.
- **Execution model:** Streamlit task builder $\rightarrow$ Argo Workflow $\rightarrow$ CPU/GPU/QPU pods $\rightarrow$ PVC artifacts $\rightarrow$ local archive/analysis.

What the codebase demonstrates

- Modular integration, not a monolithic script.
- Backend-aware workflow specifications.
- Reusable launch/archive/analyze path.
- Early validation tests for structure and task generation.

Executive summary

- **Purpose:** one user-facing console for launching quantum workflows on a Kubernetes cluster.
- **Two first-class modules:** Circuit Cutting and Sample-based Quantum Diagonalization (SQD).
- **Shared core:** cluster validation, Argo submission, runtime handling, artifact archival, QPU secret setup.
- **Execution model:** Streamlit task builder → Argo Workflow → CPU/GPU/QPU pods → PVC artifacts → local archive/analysis.

What the codebase demonstrates

- Modular integration, not a monolithic script.
- Backend-aware workflow specifications.
- Reusable launch/archive/analyze path.
- Early validation tests for structure and task generation.

Executive summary

- **Purpose:** one user-facing console for launching quantum workflows on a Kubernetes cluster.
- **Two first-class modules:** Circuit Cutting and Sample-based Quantum Diagonalization (SQD).
- **Shared core:** cluster validation, Argo submission, runtime handling, artifact archival, QPU secret setup.
- **Execution model:** Streamlit task builder $\rightarrow$ Argo Workflow $\rightarrow$ CPU/GPU/QPU pods $\rightarrow$ PVC artifacts $\rightarrow$ local archive/analysis.

What the codebase demonstrates

- Modular integration, not a monolithic script.
- Backend-aware workflow specifications.
- Reusable launch/archive/analyze path.
- Early validation tests for structure and task generation.

Executive summary

- **Purpose:** one user-facing console for launching quantum workflows on a Kubernetes cluster.
- **Two first-class modules:** Circuit Cutting and Sample-based Quantum Diagonalization (SQD).
- **Shared core:** cluster validation, Argo submission, runtime handling, artifact archival, QPU secret setup.
- **Execution model:** Streamlit task builder $\rightarrow$ Argo Workflow $\rightarrow$ CPU/GPU/QPU pods $\rightarrow$ PVC artifacts $\rightarrow$ local archive/analysis.

What the codebase demonstrates

- Modular integration, not a monolithic script.
- Backend-aware workflow specifications.
- Reusable launch/archive/analyze path.
- Early validation tests for structure and task generation.

Executive summary

- **Purpose:** one user-facing console for launching quantum workflows on a Kubernetes cluster.
- **Two first-class modules:** Circuit Cutting and Sample-based Quantum Diagonalization (SQD).
- **Shared core:** cluster validation, Argo submission, runtime handling, artifact archival, QPU secret setup.
- **Execution model:** Streamlit task builder $\rightarrow$ Argo Workflow $\rightarrow$ CPU/GPU/QPU pods $\rightarrow$ PVC artifacts $\rightarrow$ local archive/analysis.

What the codebase demonstrates

- Modular integration, not a monolithic script.
- Backend-aware workflow specifications.
- Reusable launch/archive/analyze path.
- Early validation tests for structure and task generation.

Executive summary

- **Purpose:** one user-facing console for launching quantum workflows on a Kubernetes cluster.
- **Two first-class modules:** Circuit Cutting and Sample-based Quantum Diagonalization (SQD).
- **Shared core:** cluster validation, Argo submission, runtime handling, artifact archival, QPU secret setup.
- **Execution model:** Streamlit task builder $\rightarrow$ Argo Workflow $\rightarrow$ CPU/GPU/QPU pods $\rightarrow$ PVC artifacts $\rightarrow$ local archive/analysis.

What the codebase demonstrates

- Modular integration, not a monolithic script.
- Backend-aware workflow specifications.
- Reusable launch/archive/analyze path.
- Early validation tests for structure and task generation.

Executive summary

- **Purpose:** one user-facing console for launching quantum workflows on a Kubernetes cluster.
- **Two first-class modules:** Circuit Cutting and Sample-based Quantum Diagonalization (SQD).
- **Shared core:** cluster validation, Argo submission, runtime handling, artifact archival, QPU secret setup.
- **Execution model:** Streamlit task builder $\rightarrow$ Argo Workflow $\rightarrow$ CPU/GPU/QPU pods $\rightarrow$ PVC artifacts $\rightarrow$ local archive/analysis.

What the codebase demonstrates

- Modular integration, not a monolithic script.
- Backend-aware workflow specifications.
- Reusable launch/archive/analyze path.
- Early validation tests for structure and task generation.

Why orchestration is needed

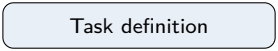
Quantum workflow experiments are not a single Python call. They require:

- parameterized task generation;
- backend-specific execution environments;
- repeatable cluster submission;
- shared storage for intermediate artifacts;
- logs, summaries and experiment provenance;
- a UI that hides operational complexity from the user.

Why orchestration is needed

Quantum workflow experiments are not a single Python call. They require:

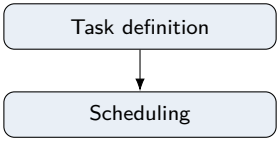
- parameterized task generation;
- backend-specific execution environments;
- repeatable cluster submission;
- shared storage for intermediate artifacts;
- logs, summaries and experiment provenance;
- a UI that hides operational complexity from the user.



Why orchestration is needed

Quantum workflow experiments are not a single Python call. They require:

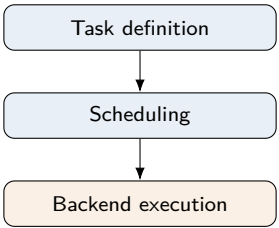
- parameterized task generation;
- backend-specific execution environments;
- repeatable cluster submission;
- shared storage for intermediate artifacts;
- logs, summaries and experiment provenance;
- a UI that hides operational complexity from the user.



Why orchestration is needed

Quantum workflow experiments are not a single Python call. They require:

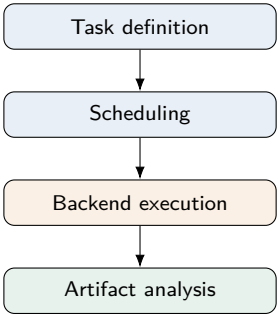
- parameterized task generation;
- backend-specific execution environments;
- repeatable cluster submission;
- shared storage for intermediate artifacts;
- logs, summaries and experiment provenance;
- a UI that hides operational complexity from the user.



Why orchestration is needed

Quantum workflow experiments are not a single Python call. They require:

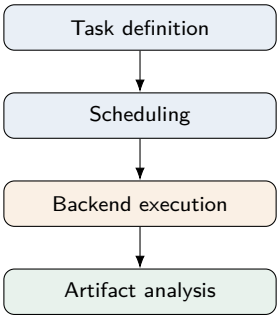
- parameterized task generation;
- backend-specific execution environments;
- repeatable cluster submission;
- shared storage for intermediate artifacts;
- logs, summaries and experiment provenance;
- a UI that hides operational complexity from the user.



Why orchestration is needed

Quantum workflow experiments are not a single Python call. They require:

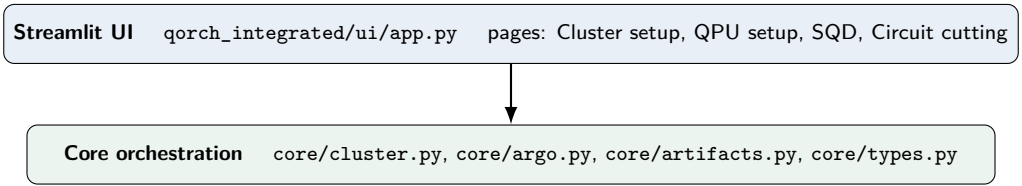
- parameterized task generation;
- backend-specific execution environments;
- repeatable cluster submission;
- shared storage for intermediate artifacts;
- logs, summaries and experiment provenance;
- a UI that hides operational complexity from the user.



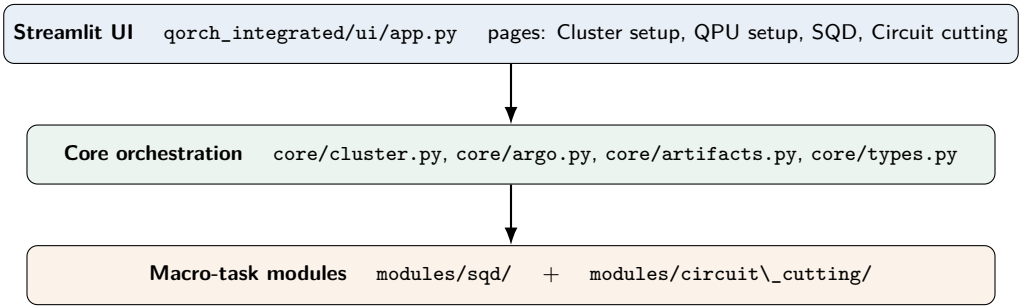
Top-level architecture

Streamlit UI `qorch_integrated/ui/app.py` pages: Cluster setup, QPU setup, SQD, Circuit cutting

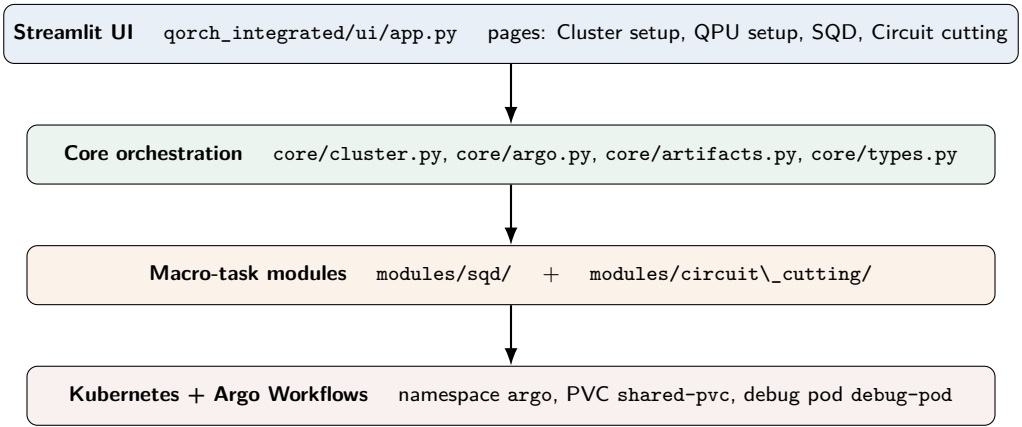
Top-level architecture



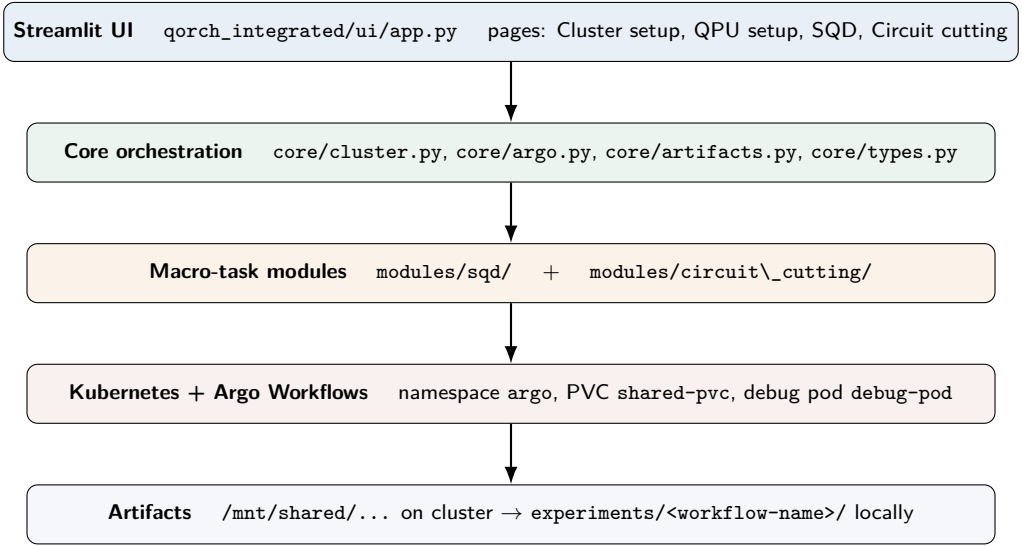
Top-level architecture



Top-level architecture



Top-level architecture



Codebase organization: separation of concerns

Where users enter

`qorch_integrated/ui/app.py`

Streamlit collects configuration and delegates execution to registered modules.

Shared infrastructure

```
core/types.py        # module contract
core/cluster.py      # kubeconfig + checks
core/argo.py         # kubectl wrapper
core/artifacts.py    # archive/log copy
core/qpu.py          # QPU secret setup
```

Scientific macro-task modules

```
modules/registry.py
modules/sqd/
  task_model.py, task_builder.py, ui.py
  k8s/argo-sqd-workflow.yaml
  executors/app/*, analysis.py
modules/circuit_cutting/
  task_model.py, task_builder.py, ui.py
  k8s/argo-workflow.yaml
  executors/app/*, analysis.py
```

Design rule

Modules own task schemas, workflow YAMLs, Docker assets, executors and analysis. The core layer sees only `MacroTaskModule` and `WorkflowSpec`.

Codebase organization: separation of concerns

Where users enter

`qorch_integrated/ui/app.py`

Streamlit collects configuration and delegates execution to registered modules.

Shared infrastructure

```
core/types.py        # module contract
core/cluster.py      # kubeconfig + checks
core/argo.py         # kubectl wrapper
core/artifacts.py    # archive/log copy
core/qpu.py          # QPU secret setup
```

Scientific macro-task modules

```
modules/registry.py
modules/sqd/
  task_model.py, task_builder.py, ui.py
  k8s/argo-sqd-workflow.yaml
  executors/app/*, analysis.py
modules/circuit_cutting/
  task_model.py, task_builder.py, ui.py
  k8s/argo-workflow.yaml
  executors/app/*, analysis.py
```

Design rule

Modules own task schemas, workflow YAMLs, Docker assets, executors and analysis. The core layer sees only `MacroTaskModule` and `WorkflowSpec`.

Codebase organization: separation of concerns

Where users enter

```
qorch_integrated/ui/app.py
```

Streamlit collects configuration and delegates execution to registered modules.

Shared infrastructure

```
core/types.py        # module contract
core/cluster.py      # kubeconfig + checks
core/argo.py         # kubectl wrapper
core/artifacts.py    # archive/log copy
core/qpu.py          # QPU secret setup
```

Scientific macro-task modules

```
modules/registry.py
modules/sqd/
  task_model.py, task_builder.py, ui.py
  k8s/argo-sqd-workflow.yaml
  executors/app/*, analysis.py
modules/circuit_cutting/
  task_model.py, task_builder.py, ui.py
  k8s/argo-workflow.yaml
  executors/app/*, analysis.py
```

Design rule

Modules own task schemas, workflow YAMLs, Docker assets, executors and analysis. The core layer sees only `MacroTaskModule` and `WorkflowSpec`.

Codebase organization: separation of concerns

Where users enter

```
qorch_integrated/ui/app.py
```

Streamlit collects configuration and delegates execution to registered modules.

Shared infrastructure

```
core/types.py        # module contract
core/cluster.py      # kubeconfig + checks
core/argo.py         # kubectl wrapper
core/artifacts.py    # archive/log copy
core/qpu.py          # QPU secret setup
```

Scientific macro-task modules

```
modules/registry.py
modules/sqd/
  task_model.py, task_builder.py, ui.py
  k8s/argo-sqd-workflow.yaml
  executors/app/*, analysis.py
modules/circuit_cutting/
  task_model.py, task_builder.py, ui.py
  k8s/argo-workflow.yaml
  executors/app/*, analysis.py
```

Design rule

Modules own task schemas, workflow YAMLs, Docker assets, executors and analysis. The core layer sees only `MacroTaskModule` and `WorkflowSpec`.

Module contract: the plug-in mechanism

1. Registry
`modules/registry.py`

UI-facing methods

`render_task_form()` `launch()`
`latest()` `status_line()`

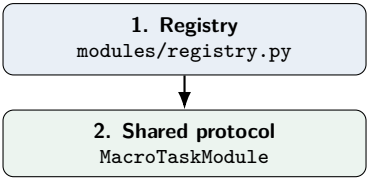
WorkflowSpec declares

Workflow file, launcher script, remote task path, artifact roots and workflow-name prefix.

Reason for the contract

Adding a macro-task should not require rewriting `app.py`; it only needs to respect the interface.

Module contract: the plug-in mechanism



UI-facing methods

`render_task_form()` `launch()`
`latest()` `status_line()`

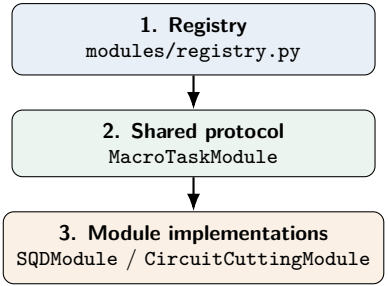
WorkflowSpec declares

Workflow file, launcher script, remote task path, artifact roots and workflow-name prefix.

Reason for the contract

Adding a macro-task should not require rewriting `app.py`; it only needs to respect the interface.

Module contract: the plug-in mechanism



UI-facing methods

`render_task_form()` `launch()`
`latest()` `status_line()`

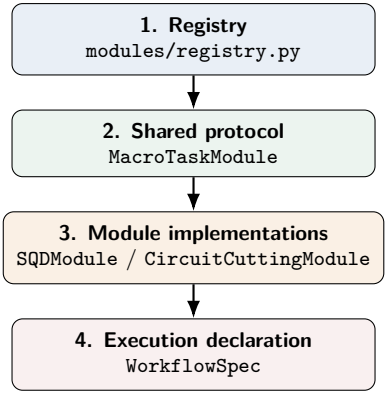
WorkflowSpec declares

Workflow file, launcher script, remote task path, artifact roots and workflow-name prefix.

Reason for the contract

Adding a macro-task should not require rewriting `app.py`; it only needs to respect the interface.

Module contract: the plug-in mechanism



UI-facing methods

`render_task_form()` `launch()`
`latest()` `status_line()`

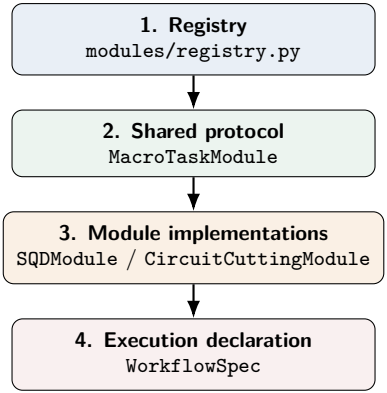
WorkflowSpec declares

Workflow file, launcher script, remote task path, artifact roots and workflow-name prefix.

Reason for the contract

Adding a macro-task should not require rewriting `app.py`; it only needs to respect the interface.

Module contract: the plug-in mechanism



UI-facing methods

`render_task_form()` `launch()`
`latest()` `status_line()`

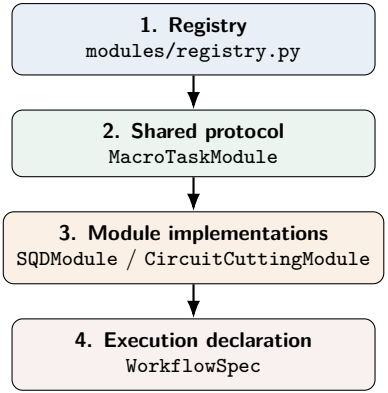
WorkflowSpec declares

Workflow file, launcher script, remote task path, artifact roots and workflow-name prefix.

Reason for the contract

Adding a macro-task should not require rewriting `app.py`; it only needs to respect the interface.

Module contract: the plug-in mechanism



UI-facing methods

`render_task_form()` `launch()`
`latest()` `status_line()`

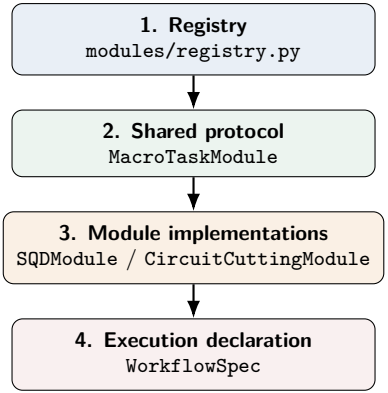
WorkflowSpec declares

Workflow file, launcher script, remote task path, artifact roots and workflow-name prefix.

Reason for the contract

Adding a macro-task should not require rewriting `app.py`; it only needs to respect the interface.

Module contract: the plug-in mechanism



UI-facing methods

`render_task_form()` `launch()`
`latest()` `status_line()`

WorkflowSpec declares

Workflow file, launcher script, remote task path, artifact roots and workflow-name prefix.

Reason for the contract

Adding a macro-task should not require rewriting `app.py`; it only needs to respect the interface.

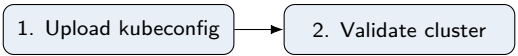
User-facing Streamlit flow

1. Upload kubeconfig

Specific current behavior

`app.py` saves uploaded kubeconfig files under `.runtime/kubeconfigs/`, activates `KUBECONFIG` in-process, validates Argo/PVC/debug-pod access, and provides Back/Next navigation plus a Quit button.

User-facing Streamlit flow



Specific current behavior

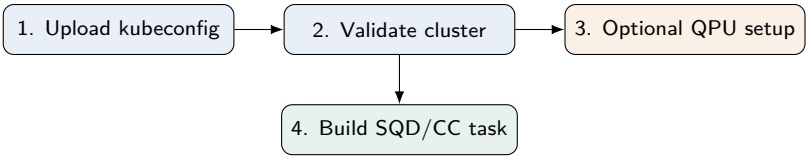
`app.py` saves uploaded kubeconfig files under `.runtime/kubeconfigs/`, activates `KUBECONFIG` in-process, validates Argo/PVC/debug-pod access, and provides Back/Next navigation plus a Quit button.

User-facing Streamlit flow

Specific current behavior

`app.py` saves uploaded kubeconfig files under `.runtime/kubeconfigs/`, activates `KUBECONFIG` in-process, validates Argo/PVC/debug-pod access, and provides Back/Next navigation plus a Quit button.

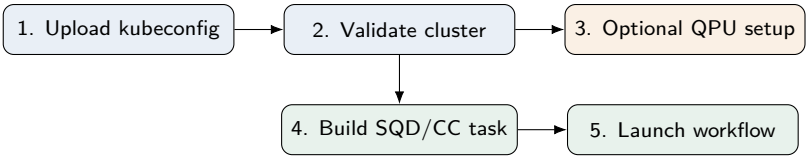
User-facing Streamlit flow



Specific current behavior

`app.py` saves uploaded kubeconfig files under `.runtime/kubeconfigs/`, activates `KUBECONFIG` in-process, validates Argo/PVC/debug-pod access, and provides Back/Next navigation plus a Quit button.

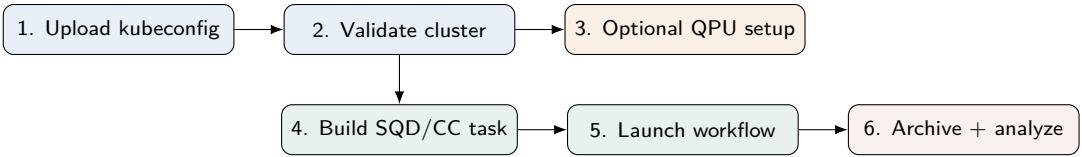
User-facing Streamlit flow



Specific current behavior

`app.py` saves uploaded kubeconfig files under `.runtime/kubeconfigs/`, activates `KUBECONFIG` in-process, validates Argo/PVC/debug-pod access, and provides Back/Next navigation plus a Quit button.

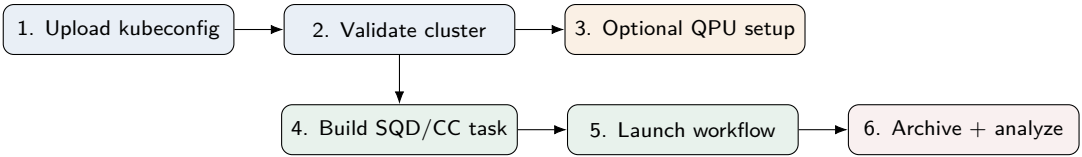
User-facing Streamlit flow



Specific current behavior

`app.py` saves uploaded kubeconfig files under `.runtime/kubeconfigs/`, activates `KUBECONFIG` in-process, validates Argo/PVC/debug-pod access, and provides Back/Next navigation plus a Quit button.

User-facing Streamlit flow



Specific current behavior

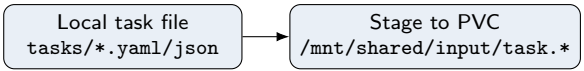
app.py saves uploaded kubeconfig files under `.runtime/kubeconfigs/`, activates `KUBECONFIG` in-process, validates Argo/PVC/debug-pod access, and provides Back/Next navigation plus a Quit button.

Core execution path: control flow and data flow

Local task file
`tasks/*.yaml/json`

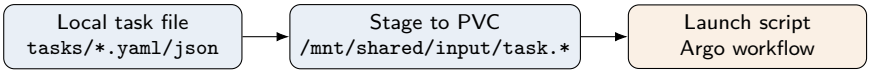
- `ArgoClient` wraps `kubectl`, discovers latest workflow names, optionally waits for terminal phases, and formats status lines.
- `archive_workflow()` stores workflow YAML, pods table/JSON, pod logs, task copy and copied `/mnt/shared` artifacts.

Core execution path: control flow and data flow



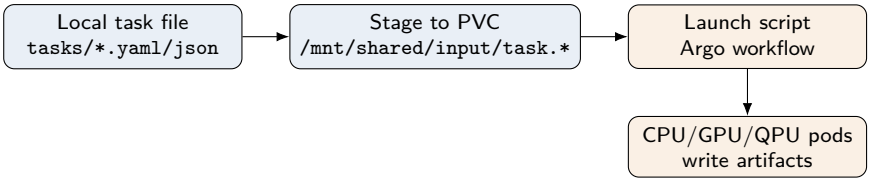
- `ArgoClient` wraps `kubectl`, discovers latest workflow names, optionally waits for terminal phases, and formats status lines.
- `archive_workflow()` stores workflow YAML, pods table/JSON, pod logs, task copy and copied `/mnt/shared` artifacts.

Core execution path: control flow and data flow



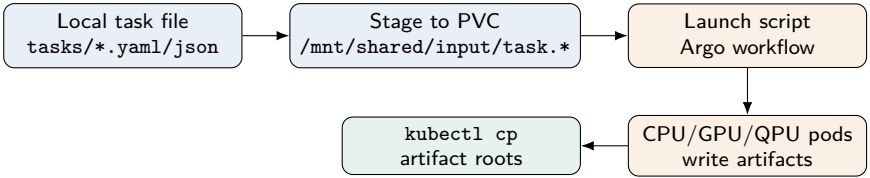
- `ArgoClient` wraps `kubectl`, discovers latest workflow names, optionally waits for terminal phases, and formats status lines.
- `archive_workflow()` stores workflow YAML, pods table/JSON, pod logs, task copy and copied `/mnt/shared` artifacts.

Core execution path: control flow and data flow



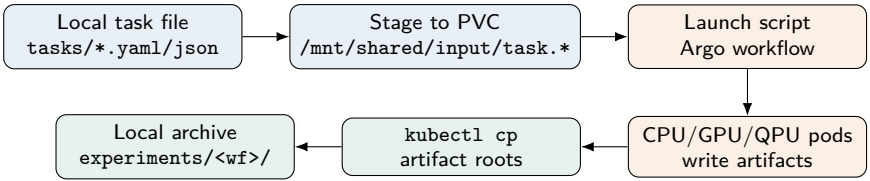
- `ArgoClient` wraps `kubectl`, discovers latest workflow names, optionally waits for terminal phases, and formats status lines.
- `archive_workflow()` stores workflow YAML, pods table/JSON, pod logs, task copy and copied `/mnt/shared` artifacts.

Core execution path: control flow and data flow



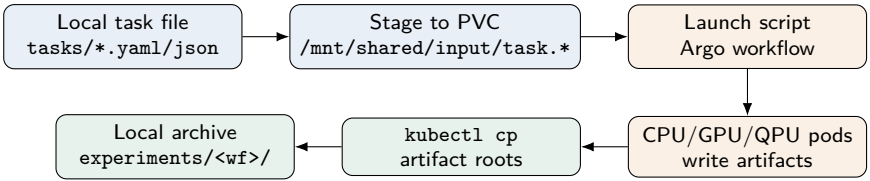
- `ArgoClient` wraps `kubectl`, discovers latest workflow names, optionally waits for terminal phases, and formats status lines.
- `archive_workflow()` stores workflow YAML, pods table/JSON, pod logs, task copy and copied `/mnt/shared` artifacts.

Core execution path: control flow and data flow



- `ArgoClient` wraps `kubectl`, discovers latest workflow names, optionally waits for terminal phases, and formats status lines.
- `archive_workflow()` stores workflow YAML, pods table/JSON, pod logs, task copy and copied `/mnt/shared` artifacts.

Core execution path: control flow and data flow



- `ArgoClient` wraps `kubectl`, discovers latest workflow names, optionally waits for terminal phases, and formats status lines.
- `archive_workflow()` stores workflow YAML, pods table/JSON, pod logs, task copy and copied `/mnt/shared` artifacts.

Circuit Cutting module: from a circuit to reconstructed results

Task JSON
circuit, cuts, routing

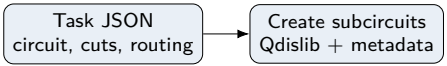
Key idea

Circuit cutting is a **fan-out/fan-in workflow**: one preparation step creates many subcircuit execution units; reconstruction joins the branch results.

Main files to point at

`task_model.py`, `task_builder.py`, `k8s/argo-workflow.yaml`, `create_subcircuits.py`, `execute_subcircuits_*.py`, `reconstruct.py`, `summarize_workflow.py`.

Circuit Cutting module: from a circuit to reconstructed results



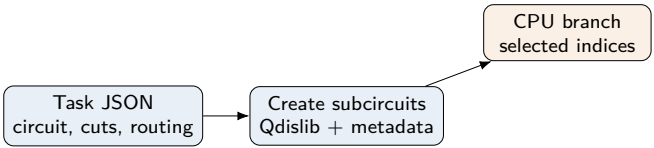
Key idea

Circuit cutting is a **fan-out/fan-in workflow**: one preparation step creates many subcircuit execution units; reconstruction joins the branch results.

Main files to point at

`task_model.py`, `task_builder.py`, `k8s/argo-workflow.yaml`, `create_subcircuits.py`, `execute_subcircuits_*.py`, `reconstruct.py`, `summarize_workflow.py`.

Circuit Cutting module: from a circuit to reconstructed results



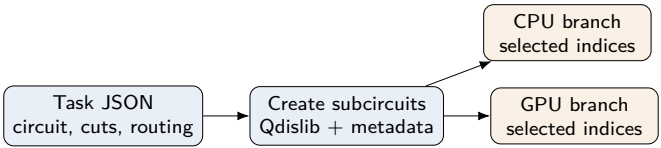
Key idea

Circuit cutting is a **fan-out/fan-in workflow**: one preparation step creates many subcircuit execution units; reconstruction joins the branch results.

Main files to point at

`task_model.py`, `task_builder.py`, `k8s/argo-workflow.yaml`, `create_subcircuits.py`, `execute_subcircuits_*.py`, `reconstruct.py`, `summarize_workflow.py`.

Circuit Cutting module: from a circuit to reconstructed results



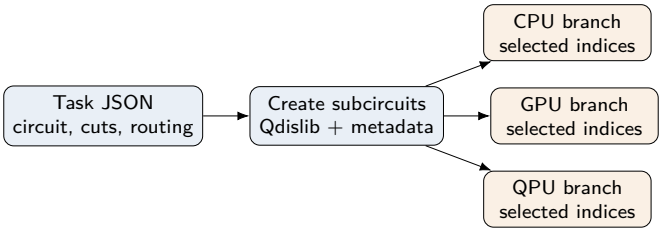
Key idea

Circuit cutting is a **fan-out/fan-in workflow**: one preparation step creates many subcircuit execution units; reconstruction joins the branch results.

Main files to point at

`task_model.py`, `task_builder.py`, `k8s/argo-workflow.yaml`, `create_subcircuits.py`, `execute_subcircuits_*.py`, `reconstruct.py`, `summarize_workflow.py`.

Circuit Cutting module: from a circuit to reconstructed results



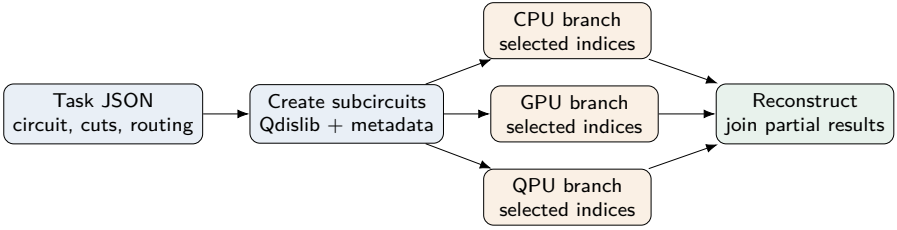
Key idea

Circuit cutting is a **fan-out/fan-in workflow**: one preparation step creates many subcircuit execution units; reconstruction joins the branch results.

Main files to point at

`task_model.py`, `task_builder.py`, `k8s/argo-workflow.yaml`, `create_subcircuits.py`, `execute_subcircuits_*.py`, `reconstruct.py`, `summarize_workflow.py`.

Circuit Cutting module: from a circuit to reconstructed results



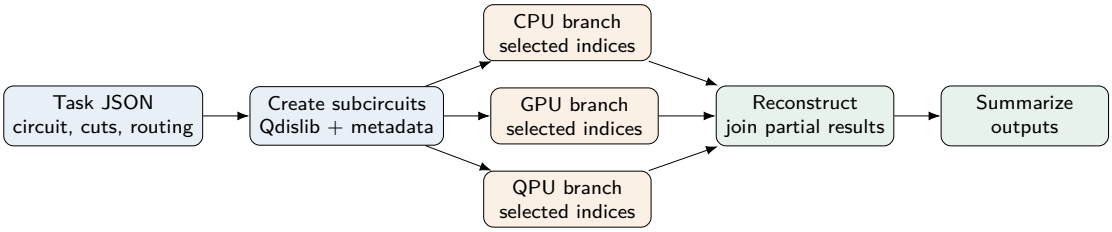
Key idea

Circuit cutting is a **fan-out/fan-in workflow**: one preparation step creates many subcircuit execution units; reconstruction joins the branch results.

Main files to point at

`task_model.py`, `task_builder.py`, `k8s/argo-workflow.yaml`, `create_subcircuits.py`, `execute_subcircuits_*.py`, `reconstruct.py`, `summarize_workflow.py`.

Circuit Cutting module: from a circuit to reconstructed results



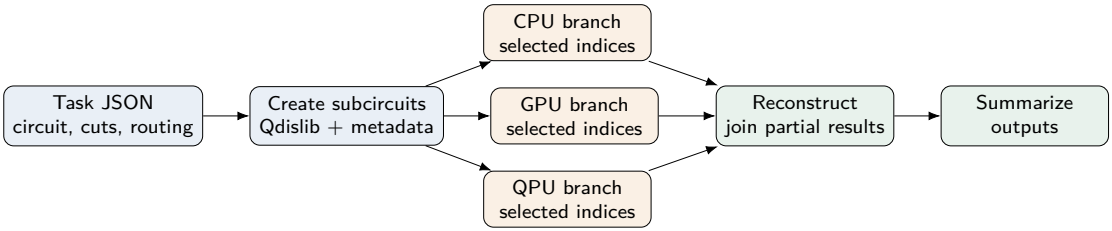
Key idea

Circuit cutting is a **fan-out/fan-in workflow**: one preparation step creates many subcircuit execution units; reconstruction joins the branch results.

Main files to point at

`task_model.py`, `task_builder.py`, `k8s/argo-workflow.yaml`, `create_subcircuits.py`, `execute_subcircuits_*.py`, `reconstruct.py`, `summarize_workflow.py`.

Circuit Cutting module: from a circuit to reconstructed results



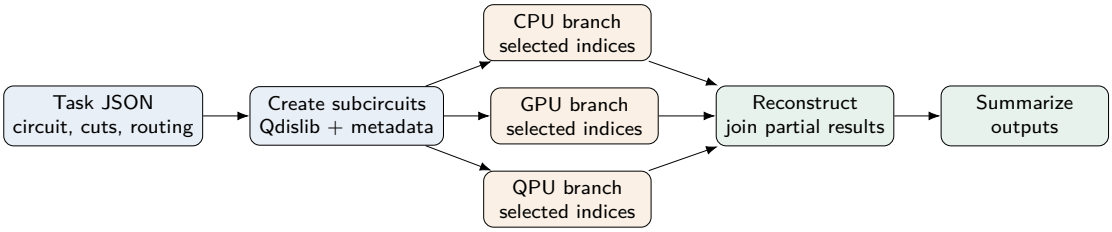
Key idea

Circuit cutting is a **fan-out/fan-in workflow**: one preparation step creates many subcircuit execution units; reconstruction joins the branch results.

Main files to point at

`task_model.py`, `task_builder.py`, `k8s/argo-workflow.yaml`, `create_subcircuits.py`, `execute_subcircuits_*.py`, `reconstruct.py`, `summarize_workflow.py`.

Circuit Cutting module: from a circuit to reconstructed results



Key idea

Circuit cutting is a **fan-out/fan-in workflow**: one preparation step creates many subcircuit execution units; reconstruction joins the branch results.

Main files to point at

`task_model.py`, `task_builder.py`, `k8s/argo-workflow.yaml`, `create_subcircuits.py`, `execute_subcircuits_*.py`, `reconstruct.py`, `summarize_workflow.py`.

Circuit Cutting: task and routing model

- Supported circuit families: `ring`, `chain`, `star`, hardware-efficient variants.
- Gate layers include `ry_rz`, `h_ry_rz`, `rx_ry_rz`; entanglers include `cz` and `cx`.
- Cut strategies: explicit, first/last k, evenly spaced, random.
- Routing modes include CPU-only, GPU-only, hybrid, custom, forced CPU/GPU split and automatic.

Automatic execution logic

`CircuitCuttingTaskConfig.validate()` enforces automatic execution only with automatic routing. Smoke tests check that the generated policy matches downstream execution.

Artifact roots

```
/mnt/shared/metadata
/mnt/shared/summaries
/mnt/shared/results
```

Circuit Cutting: task and routing model

- Supported circuit families: `ring`, `chain`, `star`, hardware-efficient variants.
- Gate layers include `ry_rz`, `h_ry_rz`, `rx_ry_rz`; entanglers include `cz` and `cx`.
- Cut strategies: explicit, first/last k, evenly spaced, random.
- Routing modes include CPU-only, GPU-only, hybrid, custom, forced CPU/GPU split and automatic.

Automatic execution logic

`CircuitCuttingTaskConfig.validate()` enforces automatic execution only with automatic routing. Smoke tests check that the generated policy matches downstream execution.

Artifact roots

```
/mnt/shared/metadata
/mnt/shared/summaries
/mnt/shared/results
```

Circuit Cutting: task and routing model

- Supported circuit families: `ring`, `chain`, `star`, hardware-efficient variants.
- Gate layers include `ry_rz`, `h_ry_rz`, `rx_ry_rz`; entanglers include `cz` and `cx`.
- Cut strategies: explicit, first/last k, evenly spaced, random.
- Routing modes include CPU-only, GPU-only, hybrid, custom, forced CPU/GPU split and automatic.

Automatic execution logic

`CircuitCuttingTaskConfig.validate()` enforces automatic execution only with automatic routing. Smoke tests check that the generated policy matches downstream execution.

Artifact roots

`/mnt/shared/metadata`
`/mnt/shared/summaries`
`/mnt/shared/results`

Circuit Cutting: task and routing model

- Supported circuit families: `ring`, `chain`, `star`, hardware-efficient variants.
- Gate layers include `ry_rz`, `h_ry_rz`, `rx_ry_rz`; entanglers include `cz` and `cx`.
- Cut strategies: explicit, first/last k, evenly spaced, random.
- Routing modes include CPU-only, GPU-only, hybrid, custom, forced CPU/GPU split and automatic.

Automatic execution logic

`CircuitCuttingTaskConfig.validate()` enforces automatic execution only with automatic routing. Smoke tests check that the generated policy matches downstream execution.

Artifact roots

`/mnt/shared/metadata`
`/mnt/shared/summaries`
`/mnt/shared/results`

Circuit Cutting: task and routing model

- Supported circuit families: `ring`, `chain`, `star`, hardware-efficient variants.
- Gate layers include `ry_rz`, `h_ry_rz`, `rx_ry_rz`; entanglers include `cz` and `cx`.
- Cut strategies: explicit, first/last k, evenly spaced, random.
- Routing modes include CPU-only, GPU-only, hybrid, custom, forced CPU/GPU split and automatic.

Automatic execution logic

`CircuitCuttingTaskConfig.validate()` enforces automatic execution only with automatic routing. Smoke tests check that the generated policy matches downstream execution.

Artifact roots

`/mnt/shared/metadata`
`/mnt/shared/summaries`
`/mnt/shared/results`

Circuit Cutting: task and routing model

- Supported circuit families: `ring`, `chain`, `star`, hardware-efficient variants.
- Gate layers include `ry_rz`, `h_ry_rz`, `rx_ry_rz`; entanglers include `cz` and `cx`.
- Cut strategies: explicit, first/last k, evenly spaced, random.
- Routing modes include CPU-only, GPU-only, hybrid, custom, forced CPU/GPU split and automatic.

Automatic execution logic

`CircuitCuttingTaskConfig.validate()` enforces automatic execution only with automatic routing. Smoke tests check that the generated policy matches downstream execution.

Artifact roots

```
/mnt/shared/metadata
/mnt/shared/summaries
/mnt/shared/results
```

SQD module: workflow logic

Hamiltonian source
manual / IBM N2 / catalog

Specific files

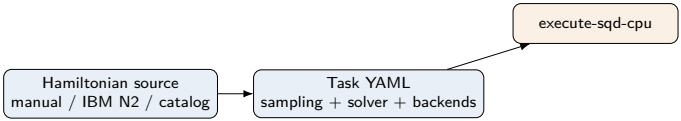
`modules/sqd/task_model.py`, `task_builder.py`
`k8s/argo-sqd-workflow.yaml`
`executors/app/execute_sqd_*.py`
`summarize_sqd.py`, `benchmarks/catalog.yaml`.

SQD module: workflow logic

Specific files

`modules/sqd/task_model.py`, `task_builder.py`
`k8s/argo-sqd-workflow.yaml`
`executors/app/execute_sqd_*.py`
`summarize_sqd.py`, `benchmarks/catalog.yaml`.

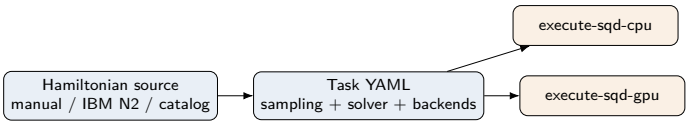
SQD module: workflow logic



Specific files

`modules/sqd/task_model.py`, `task_builder.py`
`k8s/argo-sqd-workflow.yaml`
`executors/app/execute_sqd_*.py`
`summarize_sqd.py`, `benchmarks/catalog.yaml`.

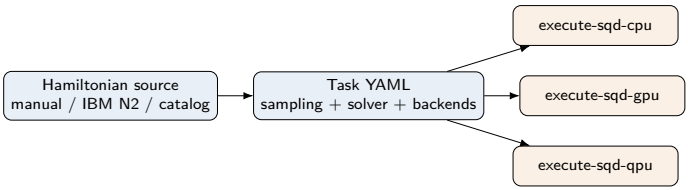
SQD module: workflow logic



Specific files

`modules/sqd/task_model.py`, `task_builder.py`
`k8s/argo-sqd-workflow.yaml`
`executors/app/execute_sqd_*.py`
`summarize_sqd.py`, `benchmarks/catalog.yaml`.

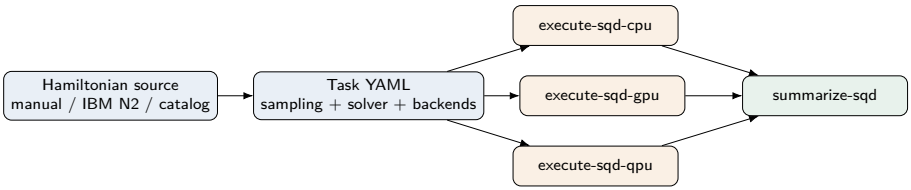
SQD module: workflow logic



Specific files

`modules/sqd/task_model.py`, `task_builder.py`
`k8s/argo-sqd-workflow.yaml`
`executors/app/execute_sqd_*.py`
`summarize_sqd.py`, `benchmarks/catalog.yaml`.

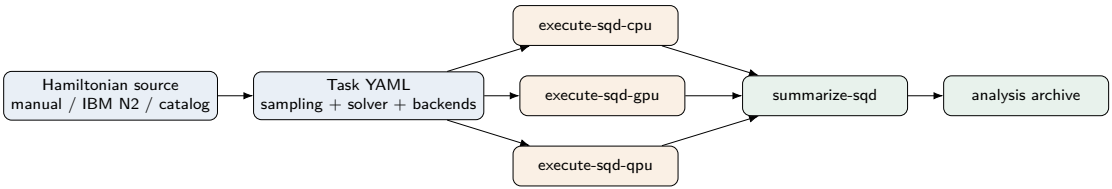
SQD module: workflow logic



Specific files

`modules/sqd/task_model.py`, `task_builder.py`
`k8s/argo-sqd-workflow.yaml`
`executors/app/execute_sqd_*.py`
`summarize_sqd.py`, `benchmarks/catalog.yaml`.

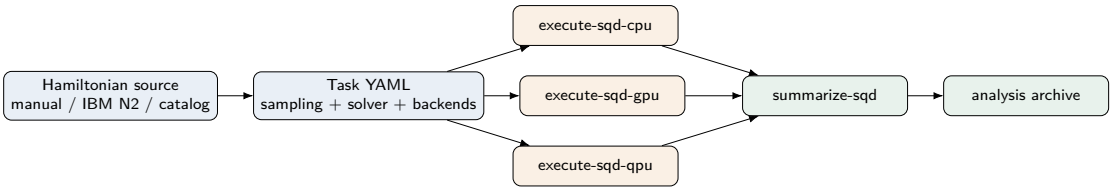
SQD module: workflow logic



Specific files

`modules/sqd/task_model.py`, `task_builder.py`
`k8s/argo-sqd-workflow.yaml`
`executors/app/execute_sqd_*.py`
`summarize_sqd.py`, `benchmarks/catalog.yaml`.

SQD module: workflow logic



Specific files

`modules/sqd/task_model.py`, `task_builder.py`
`k8s/argo-sqd-workflow.yaml`
`executors/app/execute_sqd_*.py`
`summarize_sqd.py`, `benchmarks/catalog.yaml`.

SQD: Hamiltonian and benchmark support

- **Manual Pauli source:** user writes Pauli terms directly.
- **IBM N2 reference:** built-in reference task support through `build_ibm_n2_task()`.
- **Molecule catalog:** catalog-driven task generation through molecule fixture IDs.
- **Execution settings:** backends, images, number of samples, random seed, eigensolver options `k` and `which`, exact diagonalization flag.

Current benchmark assets

- `benchmarks/catalog.yaml`
- `benchmarks/ibm_sqd/n2_equilibrium.yaml`
- examples: `h2_sto3g.yaml`, `h2o_small_active_space.yaml`, `ibm_n2_reference.yaml`

Artifact root

`/mnt/shared/sqd`

SQD: Hamiltonian and benchmark support

- **Manual Pauli source:** user writes Pauli terms directly.
- **IBM N2 reference:** built-in reference task support through `build_ibm_n2_task()`.
- **Molecule catalog:** catalog-driven task generation through molecule fixture IDs.
- **Execution settings:** backends, images, number of samples, random seed, eigensolver options `k` and `which`, exact diagonalization flag.

Current benchmark assets

- `benchmarks/catalog.yaml`
- `benchmarks/ibm_sqd/n2_equilibrium.yaml`
- examples: `h2_sto3g.yaml`, `h2o_small_active_space.yaml`, `ibm_n2_reference.yaml`

Artifact root

`/mnt/shared/sqd`

SQD: Hamiltonian and benchmark support

- **Manual Pauli source:** user writes Pauli terms directly.
- **IBM N2 reference:** built-in reference task support through `build_ibm_n2_task()`.
- **Molecule catalog:** catalog-driven task generation through molecule fixture IDs.
- **Execution settings:** backends, images, number of samples, random seed, eigensolver options `k` and `which`, exact diagonalization flag.

Current benchmark assets

- `benchmarks/catalog.yaml`
- `benchmarks/ibm_sqd/n2_equilibrium.yaml`
- examples: `h2_sto3g.yaml`, `h2o_small_active_space.yaml`, `ibm_n2_reference.yaml`

Artifact root

`/mnt/shared/sqd`

SQD: Hamiltonian and benchmark support

- **Manual Pauli source:** user writes Pauli terms directly.
- **IBM N2 reference:** built-in reference task support through `build_ibm_n2_task()`.
- **Molecule catalog:** catalog-driven task generation through molecule fixture IDs.
- **Execution settings:** backends, images, number of samples, random seed, eigensolver options `k` and `which`, exact diagonalization flag.

Current benchmark assets

- `benchmarks/catalog.yaml`
- `benchmarks/ibm_sqd/n2_equilibrium.yaml`
- examples: `h2_sto3g.yaml`, `h2o_small_active_space.yaml`, `ibm_n2_reference.yaml`

Artifact root

`/mnt/shared/sqd`

SQD: Hamiltonian and benchmark support

- **Manual Pauli source:** user writes Pauli terms directly.
- **IBM N2 reference:** built-in reference task support through `build_ibm_n2_task()`.
- **Molecule catalog:** catalog-driven task generation through molecule fixture IDs.
- **Execution settings:** backends, images, number of samples, random seed, eigensolver options `k` and `which`, exact diagonalization flag.

Current benchmark assets

- `benchmarks/catalog.yaml`
- `benchmarks/ibm_sqd/n2_equilibrium.yaml`
- examples: `h2_sto3g.yaml`, `h2o_small_active_space.yaml`, `ibm_n2_reference.yaml`

Artifact root

`/mnt/shared/sqd`

SQD: Hamiltonian and benchmark support

- **Manual Pauli source:** user writes Pauli terms directly.
- **IBM N2 reference:** built-in reference task support through `build_ibm_n2_task()`.
- **Molecule catalog:** catalog-driven task generation through molecule fixture IDs.
- **Execution settings:** backends, images, number of samples, random seed, eigensolver options `k` and `which`, exact diagonalization flag.

Current benchmark assets

- `benchmarks/catalog.yaml`
- `benchmarks/ibm_sqd/n2_equilibrium.yaml`
- examples: `h2_sto3g.yaml`, `h2o_small_active_space.yaml`, `ibm_n2_reference.yaml`

Artifact root

`/mnt/shared/sqd`

SQD: Hamiltonian and benchmark support

- **Manual Pauli source:** user writes Pauli terms directly.
- **IBM N2 reference:** built-in reference task support through `build_ibm_n2_task()`.
- **Molecule catalog:** catalog-driven task generation through molecule fixture IDs.
- **Execution settings:** backends, images, number of samples, random seed, eigensolver options `k` and `which`, exact diagonalization flag.

Current benchmark assets

- `benchmarks/catalog.yaml`
- `benchmarks/ibm_sqd/n2_equilibrium.yaml`
- examples: `h2_sto3g.yaml`, `h2o_small_active_space.yaml`, `ibm_n2_reference.yaml`

Artifact root

`/mnt/shared/sqd`

SQD: Hamiltonian and benchmark support

- **Manual Pauli source:** user writes Pauli terms directly.
- **IBM N2 reference:** built-in reference task support through `build_ibm_n2_task()`.
- **Molecule catalog:** catalog-driven task generation through molecule fixture IDs.
- **Execution settings:** backends, images, number of samples, random seed, eigensolver options `k` and `which`, exact diagonalization flag.

Current benchmark assets

- `benchmarks/catalog.yaml`
- `benchmarks/ibm_sqd/n2_equilibrium.yaml`
- examples: `h2_sto3g.yaml`, `h2o_small_active_space.yaml`, `ibm_n2_reference.yaml`

Artifact root

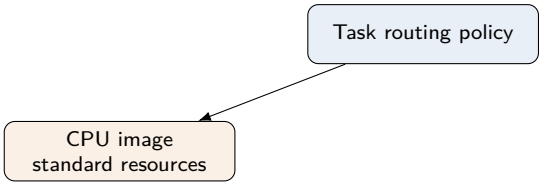
`/mnt/shared/sqd`

Backend execution model

Task routing policy

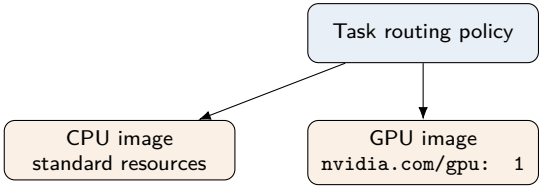
- Circuit Cutting currently hard-codes CC CPU/GPU images in `argo-workflow.yaml`; SQD exposes workflow parameters for CPU/GPU/QPU images.
- QPU credentials are isolated through Kubernetes secrets; the UI supports IBM, IQM or an existing secret.

Backend execution model



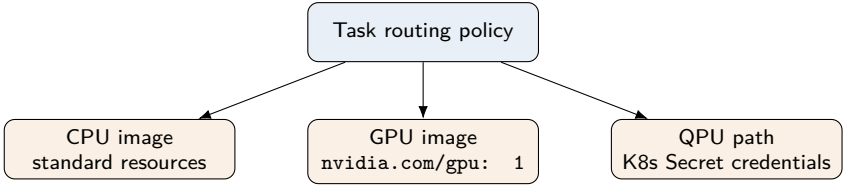
- Circuit Cutting currently hard-codes CC CPU/GPU images in `argo-workflow.yaml`; SQD exposes workflow parameters for CPU/GPU/QPU images.
- QPU credentials are isolated through Kubernetes secrets; the UI supports IBM, IQM or an existing secret.

Backend execution model



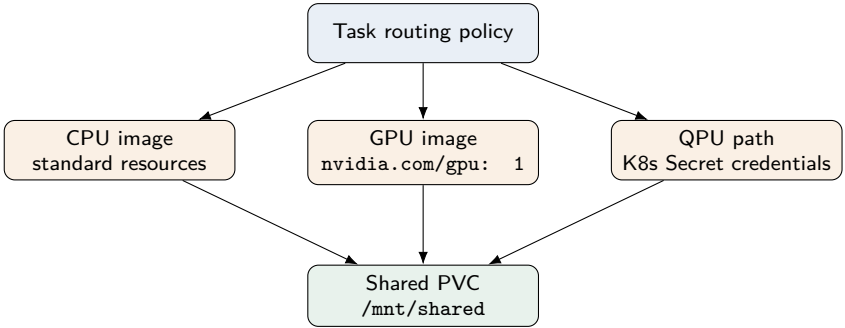
- Circuit Cutting currently hard-codes CC CPU/GPU images in `argo-workflow.yaml`; SQD exposes workflow parameters for CPU/GPU/QPU images.
- QPU credentials are isolated through Kubernetes secrets; the UI supports IBM, IQM or an existing secret.

Backend execution model



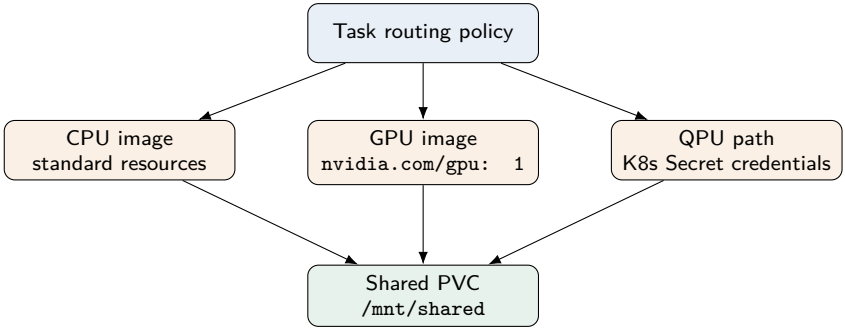
- Circuit Cutting currently hard-codes CC CPU/GPU images in `argo-workflow.yaml`; SQD exposes workflow parameters for CPU/GPU/QPU images.
- QPU credentials are isolated through Kubernetes secrets; the UI supports IBM, IQM or an existing secret.

Backend execution model



- Circuit Cutting currently hard-codes CC CPU/GPU images in `argo-workflow.yaml`; SQD exposes workflow parameters for CPU/GPU/QPU images.
- QPU credentials are isolated through Kubernetes secrets; the UI supports IBM, IQM or an existing secret.

Backend execution model



- Circuit Cutting currently hard-codes CC CPU/GPU images in `argo-workflow.yaml`; SQD exposes workflow parameters for CPU/GPU/QPU images.
- QPU credentials are isolated through Kubernetes secrets; the UI supports IBM, IQM or an existing secret.

Artifact archival and analysis pipeline

Archived by the shared core

- original task file;
- workflow YAML;
- pods table and pods JSON;
- per-pod logs from the `main` container;
- copied shared-storage artifact roots;
- optional live status log.

Module-specific analysis

- `sqd/analysis.py`: parses SQD results/summaries.
- `circuit_cutting/analysis.py`: handles integrated archive layout and creates tables such as backend usage, duration summaries and pod summaries.

Local convention

`modules/<module>/experiments/<workflow-name>/analysis/`

Artifact archival and analysis pipeline

Archived by the shared core

- original task file;
- workflow YAML;
- pods table and pods JSON;
- per-pod logs from the `main` container;
- copied shared-storage artifact roots;
- optional live status log.

Module-specific analysis

- `sqd/analysis.py`: parses SQD results/summaries.
- `circuit_cutting/analysis.py`: handles integrated archive layout and creates tables such as backend usage, duration summaries and pod summaries.

Local convention

`modules/<module>/experiments/<workflow-name>/analysis/`

Artifact archival and analysis pipeline

Archived by the shared core

- original task file;
- workflow YAML;
- pods table and pods JSON;
- per-pod logs from the `main` container;
- copied shared-storage artifact roots;
- optional live status log.

Module-specific analysis

- `sqd/analysis.py`: parses SQD results/summaries.
- `circuit_cutting/analysis.py`: handles integrated archive layout and creates tables such as backend usage, duration summaries and pod summaries.

Local convention

`modules/<module>/experiments/<workflow-name>/analysis/`

Artifact archival and analysis pipeline

Archived by the shared core

- original task file;
- workflow YAML;
- pods table and pods JSON;
- per-pod logs from the `main` container;
- copied shared-storage artifact roots;
- optional live status log.

Module-specific analysis

- `sqd/analysis.py`: parses SQD results/summaries.
- `circuit_cutting/analysis.py`: handles integrated archive layout and creates tables such as backend usage, duration summaries and pod summaries.

Local convention

`modules/<module>/experiments/<workflow-name>/analysis/`

Artifact archival and analysis pipeline

Archived by the shared core

- original task file;
- workflow YAML;
- pods table and pods JSON;
- per-pod logs from the `main` container;
- copied shared-storage artifact roots;
- optional live status log.

Module-specific analysis

- `sqd/analysis.py`: parses SQD results/summaries.
- `circuit_cutting/analysis.py`: handles integrated archive layout and creates tables such as backend usage, duration summaries and pod summaries.

Local convention

`modules/<module>/experiments/<workflow-name>/analysis/`

Artifact archival and analysis pipeline

Archived by the shared core

- original task file;
- workflow YAML;
- pods table and pods JSON;
- per-pod logs from the `main` container;
- copied shared-storage artifact roots;
- optional live status log.

Module-specific analysis

- `sqd/analysis.py`: parses SQD results/summaries.
- `circuit_cutting/analysis.py`: handles integrated archive layout and creates tables such as backend usage, duration summaries and pod summaries.

Local convention

`modules/<module>/experiments/<workflow-name>/analysis/`

Artifact archival and analysis pipeline

Archived by the shared core

- original task file;
- workflow YAML;
- pods table and pods JSON;
- per-pod logs from the `main` container;
- copied shared-storage artifact roots;
- optional live status log.

Module-specific analysis

- `sqd/analysis.py`: parses SQD results/summaries.
- `circuit_cutting/analysis.py`: handles integrated archive layout and creates tables such as backend usage, duration summaries and pod summaries.

Local convention

`modules/<module>/experiments/<workflow-name>/analysis/`

Artifact archival and analysis pipeline

Archived by the shared core

- original task file;
- workflow YAML;
- pods table and pods JSON;
- per-pod logs from the `main` container;
- copied shared-storage artifact roots;
- optional live status log.

Module-specific analysis

- `sqd/analysis.py`: parses SQD results/summaries.
- `circuit_cutting/analysis.py`: handles integrated archive layout and creates tables such as backend usage, duration summaries and pod summaries.

Local convention

`modules/<module>/experiments/<workflow-name>/analysis/`

Artifact archival and analysis pipeline

Archived by the shared core

- original task file;
- workflow YAML;
- pods table and pods JSON;
- per-pod logs from the `main` container;
- copied shared-storage artifact roots;
- optional live status log.

Module-specific analysis

- `sqd/analysis.py`: parses SQD results/summaries.
- `circuit_cutting/analysis.py`: handles integrated archive layout and creates tables such as backend usage, duration summaries and pod summaries.

Local convention

`modules/<module>/experiments/<workflow-name>/analysis/`

Artifact archival and analysis pipeline

Archived by the shared core

- original task file;
- workflow YAML;
- pods table and pods JSON;
- per-pod logs from the `main` container;
- copied shared-storage artifact roots;
- optional live status log.

Module-specific analysis

- `sqd/analysis.py`: parses SQD results/summaries.
- `circuit_cutting/analysis.py`: handles integrated archive layout and creates tables such as backend usage, duration summaries and pod summaries.

Local convention

```
modules/<module>/experiments/
  <workflow-name>/analysis/
```

Technical demo checklist

Local checks

- compile the package and run smoke tests;
- parse all YAML/JSON task examples;
- generate one SQD task and one Circuit Cutting task;
- verify that generated routing policies match UI choices.

Cluster checks

- upload kubeconfig from the UI;
- validate Argo, debug pod and shared PVC;
- launch a small workflow with wait enabled;
- archive artifacts and inspect analysis tables/logs.

Technical demo checklist

Local checks

- compile the package and run smoke tests;
- parse all YAML/JSON task examples;
- generate one SQD task and one Circuit Cutting task;
- verify that generated routing policies match UI choices.

Cluster checks

- upload kubeconfig from the UI;
- validate Argo, debug pod and shared PVC;
- launch a small workflow with wait enabled;
- archive artifacts and inspect analysis tables/logs.

Technical demo checklist

Local checks

- compile the package and run smoke tests;
- parse all YAML/JSON task examples;
- generate one SQD task and one Circuit Cutting task;
- verify that generated routing policies match UI choices.

Cluster checks

- upload kubeconfig from the UI;
- validate Argo, debug pod and shared PVC;
- launch a small workflow with wait enabled;
- archive artifacts and inspect analysis tables/logs.

Technical demo checklist

Local checks

- compile the package and run smoke tests;
- parse all YAML/JSON task examples;
- generate one SQD task and one Circuit Cutting task;
- verify that generated routing policies match UI choices.

Cluster checks

- upload kubeconfig from the UI;
- validate Argo, debug pod and shared PVC;
- launch a small workflow with wait enabled;
- archive artifacts and inspect analysis tables/logs.

Technical demo checklist

Local checks

- compile the package and run smoke tests;
- parse all YAML/JSON task examples;
- generate one SQD task and one Circuit Cutting task;
- verify that generated routing policies match UI choices.

Cluster checks

- upload kubeconfig from the UI;
- validate Argo, debug pod and shared PVC;
- launch a small workflow with wait enabled;
- archive artifacts and inspect analysis tables/logs.

Technical demo checklist

Local checks

- compile the package and run smoke tests;
- parse all YAML/JSON task examples;
- generate one SQD task and one Circuit Cutting task;
- verify that generated routing policies match UI choices.

Cluster checks

- upload kubeconfig from the UI;
- validate Argo, debug pod and shared PVC;
- launch a small workflow with wait enabled;
- archive artifacts and inspect analysis tables/logs.

Technical demo checklist

Local checks

- compile the package and run smoke tests;
- parse all YAML/JSON task examples;
- generate one SQD task and one Circuit Cutting task;
- verify that generated routing policies match UI choices.

Cluster checks

- upload kubeconfig from the UI;
- validate Argo, debug pod and shared PVC;
- launch a small workflow with wait enabled;
- archive artifacts and inspect analysis tables/logs.

Technical demo checklist

Local checks

- compile the package and run smoke tests;
- parse all YAML/JSON task examples;
- generate one SQD task and one Circuit Cutting task;
- verify that generated routing policies match UI choices.

Cluster checks

- upload kubeconfig from the UI;
- validate Argo, debug pod and shared PVC;
- launch a small workflow with wait enabled;
- archive artifacts and inspect analysis tables/logs.

Take-home message

Main message

The current codebase is not just a collection of quantum scripts: it is a modular orchestration skeleton that turns quantum experiment definitions into reproducible, backend-aware, Kubernetes-executed workflows with a path toward archival, analysis and UI-based operation.

Take-home message

Main message

The current codebase is not just a collection of quantum scripts: it is a modular orchestration skeleton that turns quantum experiment definitions into reproducible, backend-aware, Kubernetes-executed workflows with a path toward archival, analysis and UI-based operation.

Core
reusable infrastructure

Take-home message

Main message

The current codebase is not just a collection of quantum scripts: it is a modular orchestration skeleton that turns quantum experiment definitions into reproducible, backend-aware, Kubernetes-executed workflows with a path toward archival, analysis and UI-based operation.

Core
reusable infrastructure

SQD
Hamiltonian workflows

Take-home message

Main message

The current codebase is not just a collection of quantum scripts: it is a modular orchestration skeleton that turns quantum experiment definitions into reproducible, backend-aware, Kubernetes-executed workflows with a path toward archival, analysis and UI-based operation.

Core
reusable infrastructure

SQD
Hamiltonian workflows

Circuit Cutting
subcircuit workflows