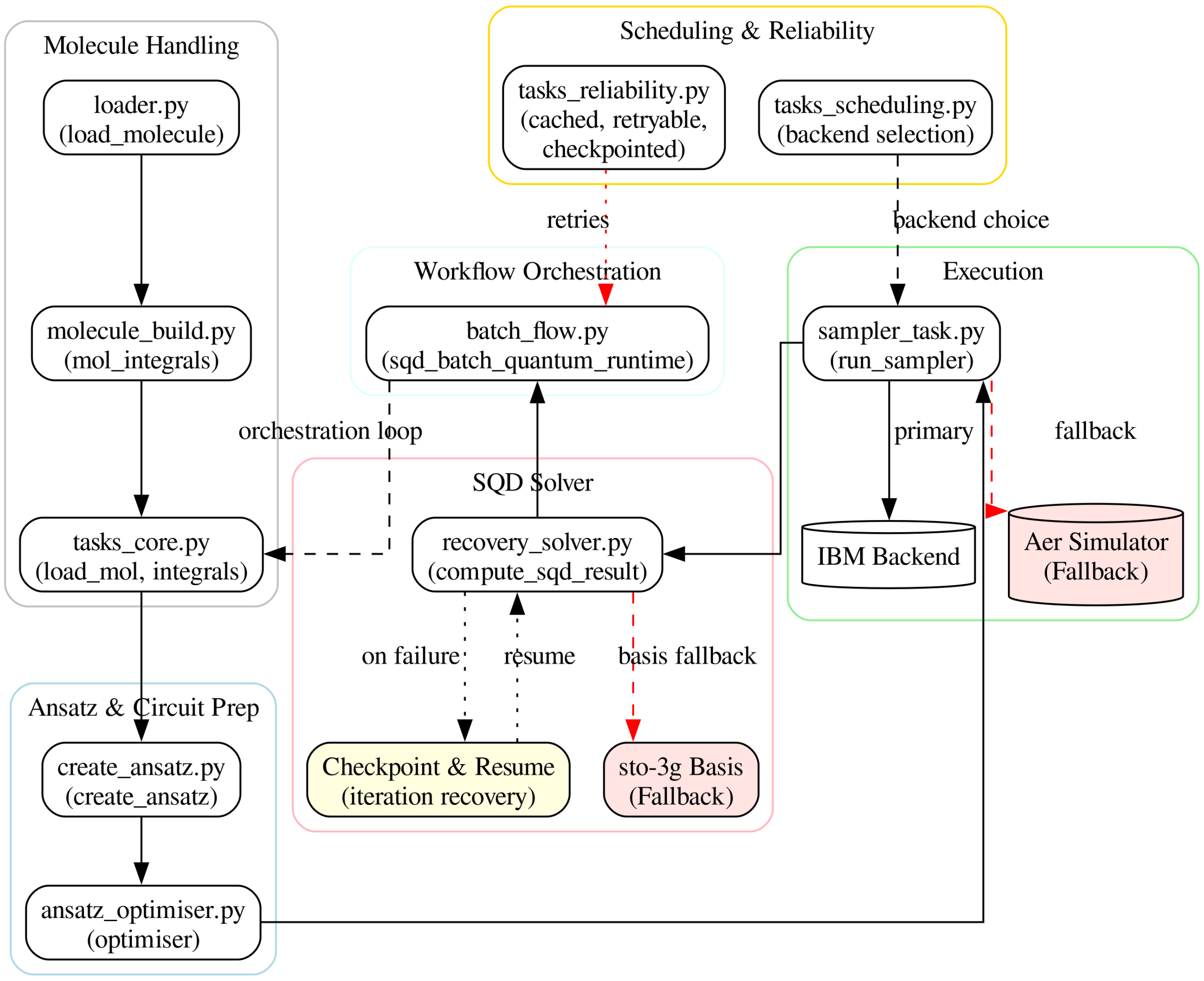
Quantum Chemistry Workflow Documentation

Overview

This project implements a quantum chemistry computational pipeline that leverages IBM Quantum hardware through Qiskit and is orchestrated using **Prefect** for robust, fault-tolerant execution. The system performs **Sampled Quantum Diagonalization (SQD)** calculations on molecular systems with a custom algorithm for backend selection, load balancing between backends, and multi-level error recovery with fallback mechanism. The goal is to provide a **Quantum Centric Supercomputing** orchestrated pipeline.

---

1 Pipeline



Molecule Handling
loader.py (load_molecule)
molecule_build.py (mol_integrals)
tasks_core.py (load_mol, integrals)
Scheduling & Reliability
tasks_reliability.py (cached, retryable, checkpointed)
tasks_scheduling.py (backend selection)
retries
backend choice
Workflow Orchestration
batch_flow.py (sqd_batch_quantum_runtime)
Execution
sampler_task.py (run_sampler)
orchestration loop
primary
fallback
SQD Solver
recovery_solver.py (compute_sqd_result)
IBM Backend
Aer Simulator (Fallback)
on failure
resume
basis fallback
Ansatz & Circuit Prep
create_ansatz.py (create_ansatz)
ansatz_optimiser.py (optimiser)
Checkpoint & Resume (iteration recovery)
sto-3g Basis (Fallback)

Figure 1: Quantum Chemistry Workflow Pipeline

2 Project Architecture

```
SQD_pipeline/
    | chemistry/                  # Core quantum chemistry logic
        | create_ansatz.py            # Quantum circuit ansatz construction
        | molecule_build.py           # Structure & integral computation
        | ansatz_optimiser.py         # Circuit optimization for backends
        | zigzag_layout.py            # Qubit layout optimization
        | recovery_solver.py          # SQD solver implementation
        | loader.py                   # Molecule file parser
    | flows/                      # Prefect workflow orchestration
        | batch_flow.py               # Main workflow orchestrator
        | sampler_task.py             # Quantum circuit execution engine
        | tasks_scheduling.py         # Backend selection & load balancing
        | tasks_reliability.py        # Custom decorators & utilities
        | tasks_core.py               # Molecules & integrals comp. & data
    | compounds/                  # Input compounds data
    | compounds_fallback/         # Fallback update data(sto-3g)
    | backend_logs/               # IBM Quantum backend logs
    | .prefect_cache/              # Prefect task caches directory
```

---

3 Prefect usage throughout the project

3.1 Main Flow Definition (batch_flow.py)

```python
@flow(log_prints=True, task_runner=ThreadPoolTaskRunner(max_workers=
    MAX_CONCURRENT_PREPARATIONS))
def sqd_batch_quantum_runtime(compounds_folder="compounds/"):
```

- **Purpose**: Defines the main orchestration workflow.
- **Features**:
  - `log_prints=True`: Captures all print statements in Prefect logs.
  - `ThreadPoolTaskRunner`: Enables controlled parallel execution.
  - Orchestrates two-phase execution (preparation → SQD).

3.2 Quantum Circuit Execution (sampler_task.py)

```python
@task(retries=3, retry_delay_seconds=30, cache_policy=None)
def run_sampler(circ, backend_obj, backend_name, options):
```

- **Purpose**: Executes quantum circuits on IBM hardware with AER fallback (AER runs on the simulated real backened with current backend state information).
- **Prefect Features**:
  - **Native retries**: 3 automatic retries with 30-second delays.
  - **No caching**: Fresh execution for each quantum job.
  - **Structured logging**: via `get_run_logger()`.

3.3 Parallel Preparation Tasks (`batch_flow.py`)

```python
@task(cache_policy=NONE, tags=["preparation"])
def prepare_compound_for_sqd_with_load_check(mol_file: str, backend_name: str, reps: int
    = 1, opt: int = 3):
```

- **Purpose**: Prepares molecules for quantum computation in parallel.
- **Prefect Features**:
  - **Task tagging**: `["preparation"]` for organization.
  - **No caching**: Fresh preparation each time.
  - **Async submission**: via `.submit()` for parallel pre-computation before sending on quantum backend.

3.4 Backend Selection (`tasks_scheduling.py`)

```python
@task
def analyze_compounds_and_select_backends(compounds_folder="compounds/", load_factor=10):

@task(cache_policy=None)
def choose_backend_for_molecule(mol_file: str, backend_assignments: dict):
```

- **Purpose**: Custom algorithm based backend selection and load balancing.
- **Prefect Features**:
  - **Task isolation**: Each function runs in a controlled environment.
  - **Structured logging**: Detailed backend analysis logs on prefect dashboard.

3.5 Core Molecular Tasks (`tasks_core.py`)

```python
@cached
def load_mol(path):

@cached
@retryable()
def integrals(mi):
```

- **Purpose**: Operations for molecule loading and integral computation.
- **Prefect Features**:
  - **Custom caching**: Custom wrapper function based on prefect's caching (can be customised in task_reliability.py.
  - **Custom retries**: Retry capability if molecule related computation fails.

---

4 Custom Decorators

4.1 `@retryable` Decorator

Location: `tasks_reliability.py`

```python
def manual_retry_decorator(max_tries=3, delay_s=30):
    """Decorator with manual retry capability"""
    def decorator(fn):
        @task(retries=0, cache_policy=None)
        def wrapper(*args, **kwargs):
            for attempt in range(max_tries):
                try:
                    return fn(*args, **kwargs)
                except Exception as e:
                    log = get_run_logger()
                    log.error(f"Attempt {attempt + 1}/{max_tries} failed: {str(e)}")
                    if attempt < max_tries - 1:
                        log.info(f"Press ENTER to retry immediately or wait {delay_s}s...
    ")
                        try:
                            i, _, _ = select.select([sys.stdin], [], [], delay_s)
                            if i:
                                sys.stdin.readline()
                                log.info("Manual retry triggered")
                            else:
                                log.info("Auto-retry after timeout")
                        except:
                            time.sleep(delay_s)
            raise RuntimeError(f"All {max_tries} attempts failed")
        return wrapper
    return decorator

#called as

def retryable(max_tries=3, delay_s=30):
    return manual_retry_decorator(max_tries, delay_s)
```

Key Features:

- **Interactive Control**: Allows manual retry via console input.
- **Prefect Integration**: Uses `@task` internally with `retries=0` to handle retries manually.
- **Development-Friendly**: Press ENTER to retry immediately instead of waiting.

Used In:

- `build_isa()`: Circuit construction and optimization.
- `integrals()`: Molecular integral computation.

4.2 `@cached` Decorator

Location: `tasks_reliability.py`

```python
def cached(fn):
    return task(
        cache_key_fn=lambda *a, **k: f"cache_{fn.__name__}_{hash(str(a) + str(k)) %
    10000}",
        cache_expiration=CACHE_EXP,  # 24 hours
        persist_result=False
    )(fn)
```

Key Features:

- **Prefect Caching**: Uses Prefect's native caching mechanism.

- **Custom Cache Expiration Time**: Cached results can be saved for custom time intervals.
- **Hash-Based Keys**: Cache keys based on function name and parameters.

Used In:

- `load_mol()`: Molecule file parsing .
- `integrals()`: Quantum chemistry integral computation .

4.3 `@checkpointed` Decorator

Location: `tasks_reliability.py`

```python
def checkpointed(fn):
    @task(retries=0, persist_result=False, cache_policy=None)
    def wrapper(*a, ckpt_key: str, **k):
        log = get_run_logger()
        pkl = CACHE_DIR / f"{ckpt_key}.pkl"
        state = None

        if pkl.exists():
            state = pickle.loads(pkl.read_bytes())
            log.info(f"Loaded checkpoint {pkl}")

        result = fn(*a, init_state=state, ckpt_key=ckpt_key, **k)

        pkl.write_bytes(pickle.dumps(result))
        log.info(f"Saved checkpoint {pkl}")
        return result
    return wrapper
```

Key Features:

- **Stateful Recovery**: Saves intermediate computation state to disk.
- **Pickle Persistence**: Uses Python pickle for state serialization.
- **Resume Capability**: Can resume long-running SQD calculations.
- **Prefect Task Integration**: Wraps function as a Prefect task with checkpointing.

Used In:

- `run_sqd()`: Long-running quantum diagonalization computations.

---

5 Prefect Workflow Execution Pattern

5.1 Async Task Submission

```python
# Submit tasks for parallel execution
preparation_futures = []
for mol_file in molecule_files:
    future = prepare_compound_for_sqd_with_load_check.submit(mol_file, backend_name)
    preparation_futures.append((mol_file, mol_name, future))

# Collect results as they complete
for mol_file, mol_name, future in preparation_futures:
    compound_data = future.result()  # Blocks until task completes
```

5.2 Structured Logging Throughout

```python
from prefect import get_run_logger
log = get_run_logger()

log.info(f"JOB SUBMITTED: {job_id} on {backend_name}")
log.error(f"Real quantum execution failed: {real_error}")
```

5.3 System Load Integration

```python
def wait_for_system_capacity():
    while check_system_load():
        print("System overloaded, waiting 30 seconds...")
        time.sleep(30)

# Used before task execution
wait_for_system_capacity()  # Respects system resources
```

---

6 Workflow Phases

6.1 Phase 1: Parallel Preparation

```python
# Controlled concurrency with Prefect
@flow(task_runner=ThreadPoolTaskRunner(max_workers=3))
def sqd_batch_quantum_runtime():
    # All molecule preparations run in parallel
    for mol_file in molecule_files:
        future = prepare_compound_for_sqd_with_load_check.submit(...)
```

What Happens:

1. Molecules analyzed for complexity.
2. Backends assigned via load balancing.
3. Quantum circuits built and optimized in parallel.
4. Circuits executed on IBM hardware (with Aer fallback).
5. System load monitored throughout.

6.2 Phase 2: Sequential SQD Computation

```python
# Sequential execution for data dependencies
for compound_data in prepared_compounds:
    result = run_sqd_for_compound(compound_data)  # Uses @checkpointed
    if result is None:
        # Automatic fallback to sto-3g basis
        result = rerun_compound_with_sto3g_fallback(...)
```

What Happens:

1. SQD algorithm applied sequentially to each molecule.
2. Checkpoint recovery for interrupted calculations.
3. Automatic fallback to simpler basis sets on IndexErrors.
4. Results saved with timestamps and metadata.

---

7 Error Recovery Strategy

- **Prefect Native Retries**: Quantum circuit execution: 3 retries, 30s delay. Automatic handling of connection or hardware failures.
- **Custom Interactive Retries**: Circuit optimization and integrals: Manual retry via terminal.
- **Hardware Fallbacks**: IBM Quantum → Aer Simulator (automatic). Real backend → Simulator backend (for AER).
- **Algorithmic Fallbacks**: Complex basis sets → sto-3g basis (for IndexErrors). Maintains scientific validity but trade-offs of accuracy is faced.
- **Checkpointing**: Long-running SQD computations saved to disk. Resume from last checkpoint on restart.

---

8 Key Configuration Parameters

```
# System Resource Management
LOAD_THRESHOLD = 90                    # CPU/memory threshold (%)
MAX_CONCURRENT_PREPARATIONS = 3        # Parallel preparation limit

# Backend Load Balancing
load_factor = 20000                    # Load penalty weight (found 20000 to work on many
    trials)

# Prefect Task Settings
retries=3, retry_delay_seconds=30      # Native Prefect retries
cache_expiration=24h                   # Custom caching duration

# SQD Algorithm (values are taken from the offical SQD documentation)
energy_tol = 1e-3                      # Convergence tolerance
max_iterations = 5                     # Maximum iterations
samples_per_batch = 300                # Quantum samples per batch
```

---

9 Usage Instructions

9.1 Setup and Execution

1. **Install Dependencies**: Be sure to use **virtual environment**

```
pip install prefect qiskit qiskit-ibm-runtime pyscf ffsim
```

2. **Submitting Molecule or Compound Data**:

   - **Data Submission**: Add the molecule or compound to be computed in the following manner in /compounds folder.

```
atom = [
    ["C", (0.000000, 0.000000, 0.000000)],
    ["H", (0.629118, 0.629118, 0.629118)],
    ["H", (-0.629118, -0.629118, 0.629118)],
    ["H", (0.629118, -0.629118, -0.629118)],
    ["H", (-0.629118, 0.629118, -0.629118)]
]
basis = "6-31g"
charge = 0
```

```
spin_sq = 0
symmetry = False
n_frozen = 1
```

3. **Configure Prefect Blocks**
   - **IBM Quantum Credentials**: Store IBM Quantum API key and CRN details.
     - Name: my-ibm-client (for naming consistency with code)
   - **Quantum Runtime**: Configure default quantum backend. Redundant but can be tinkered for complete prefect-qiskit usage but current code uses mainly QISKIT SDK with the block only for credentials.
     - Name: default-runtime (for naming consistency with code)
   - **Local File System**: To store cache using prefect in local.
     - Name : sqd-local-cache (for naming consistency with code)
     - Basepath: .prefect_cache (for naming consistency with code)
4. **Run Workflow**:

```
python -m flows.batch_flow
#be sure to run in the root directory ,i.e., SQD_pipeline folder
```

5. **Monitor Execution**:
   - **Prefect UI**: Real-time task monitoring and retry management.
   - **Console Logs**: Quantum job IDs and execution progress.
   - **Backend Logs**: Detailed hardware status in `backend_logs/`.

---

10 Result Storage

File Naming Convention:

```
result_molecule1_20250831_141530.txt          # Normal execution
result_molecule2_20250831_141545_fallback.txt # Fallback execution
```

Result File Content:

```
Molecule: water.txt
Backend: ibm_brisbane
Quantum Job ID: ct9k2b4560bg008hvt8g
SQD Energy: -75.123456
Fallback Used: False
Timestamp: 20250831_141530
Full Result: SCIResult(energy=-76.234, ...)
```

---

11 Conclusion

This Quantum-Centric Supercomputing workflow executes **Sample based Quantum Diagonalisation** through:

- **Prefect Orchestration**: Robust task management, retries, and monitoring.
- **Custom Decorators**: Enhanced Prefect capabilities with interactive retries, caching, and checkpointing.
- **Fault Tolerance**: Multi-level error recovery from hardware to algorithmic fallbacks.
- **Load Balancing**: Intelligent distribution across IBM Quantum backends.
- **Scientific Robustness**: Maintains computational validity through basis set fallbacks.

This **Quantum-Centric Supercomputing** orchestrated workflow executes **Sample based Quantum Diagonalisation** with a custom resource management and allocation algorithm that can be replaced by **Machine Learning** for even smarter resource handling.

The future of research is backed by **AI for Quantum Centric Supercomputing**.