**DataPY Engine/Execution Module Rules**

*Rules to be implemented in engine/execution modules during Phase 2+* *Runtime execution, control flow, and state management rules*

**Section 1: Execution Model (Runtime)**

**1.1 Component Execution**

* 🔄 **Async task isolation**: Each component runs in independent Python async task
* 🔄 **Topological execution**: Engine walks DAG in topological order
* 🔄 **Control edge acceleration**: Control edges can delay/accelerate downstream execution
* 🔄 **Deterministic execution**: Identical inputs produce identical data/control events

**1.2 Component State Machine (Runtime)**

WAIT ──► RUNNING ──► { OK | ERROR | SKIPPED }

▲ │

└─── cascade ────┘

* 🔄 **State transitions**: WAIT → RUNNING → {OK|ERROR|SKIPPED}
* 🔄 **WAIT**: Upstream dependencies not satisfied
* 🔄 **RUNNING**: execute() coroutine active
* 🔄 **OK**: Completed without uncaught exception
* 🔄 **ERROR**: Uncaught exception or die\_on\_error=True
* 🔄 **SKIPPED**: Engine marked inactive via cascade-skip

**1.3 Timing & Ordering (Runtime)**

* 🔄 **Control edge firing**: After execute() returns and state known
* 🔄 **No interleaving**: Control edges enqueued synchronously
* 🔄 **Global visibility**: Globals visible immediately after component finishes
* 🔄 **Atomic commits**: Globals committed atomically with control edges

**1.4 Dual Plane Execution**

* 🔄 **Data plane**: Push PipelineData objects along data edges
* 🔄 **Control plane**: Trigger nodes via control edges
* 🔄 **Coordination**: Component runnable when data ready AND control fired (or STARTABLE)

**Section 2: Control-Edge Runtime Behavior**

**2.1 Control Edge Firing Logic**

* 🔄 **ok edge**: Fires exactly once when state = OK
* 🔄 **error edge**: Fires exactly once when state = ERROR
* 🔄 **Mutual exclusivity**: Exactly one of ok/error fires per component
* 🔄 **if edge**: Fires when condition evaluates True after state settles
* 🔄 **Multiple if evaluation**: All True conditions fire, not just first
* 🔄 **parallelise edge**: Fires immediately after state = OK, creates N subjobs
* 🔄 **synchronise edge**: Fires when all upstream parallel branches finish (OK or SKIPPED)

**2.2 Expression Evaluation (Runtime)**

* 🔄 **Safe evaluation**: Use restricted eval() with whitelisted functions
* 🔄 **Timeout protection**: 100ms timeout, component ERROR if exceeded
* 🔄 **Missing globals**: Return False for missing global variables
* 🔄 **Evaluation error**: Component goes ERROR on syntax/runtime errors
* 🔄 **Re-evaluation limit**: Max 100 re-evaluations to prevent infinite loops

**2.3 Control Edge Coordination**

* 🔄 **Data edge independence**: Data edges never trigger downstream execution
* 🔄 **STARTABLE bypass**: STARTABLE components run without control edges
* 🔄 **subjob\_ok/subjob\_error**: Engine-fired when subjob aggregate state final

**Section 3: Subjob Lifecycle (Runtime)**

**3.1 Subjob State Management**

PENDING ──► RUNNING ──► { OK | ERROR }

▲ ▲

│ └─ finished when last component settles

└─ starts when first component starts

* 🔄 **PENDING**: Subjob ready to start
* 🔄 **RUNNING**: At least one component active
* 🔄 **OK**: All components OK/SKIPPED, no unhandled ERRORs
* 🔄 **ERROR**: Any component ERROR with no internal error edge handling

**3.2 Nested Subjob Execution**

* 🔄 **Chained parallelise**: Allow nested subjob spawning
* 🔄 **Error isolation**: Outer subjob ignores inner failures if error edge captures
* 🔄 **State aggregation**: Handle complex nested subjob state resolution

**Section 4: Parallel Execution (Runtime)**

**4.1 Fork/Join Implementation**

* 🔄 **asyncio.gather**: Use for Talend-style multi-threading simulation
* 🔄 **Copy-on-write context**: Each parallel branch gets isolated global map copy
* 🔄 **Write-back merge**: "Last-writer-wins" with timestamp tie-breaking
* 🔄 **Logical clock**: Use subjob\_index + event counter for deterministic ordering

**4.2 Parallel Error Handling**

* 🔄 **Sibling continuation**: Parallel branches continue if one ERRORs
* 🔄 **synchronise behavior**:
  + All branches OK/SKIPPED → synchronise state = OK
  + Any branch ERROR → synchronise becomes SKIPPED, cascade-skip follows
* 🔄 **Error propagation**: Unhandled branch ERROR propagates to subjob ERROR

**4.3 Multiple Synchronise Points**

* 🔄 **Branch-set tracking**: Each synchronise listens to specific (branch-set, id) tuple
* 🔄 **No ambiguous joins**: Engine prevents unclear synchronization points

**Section 5: Global Variable Management (Runtime)**

**5.1 Global Variable Lifecycle**

* 🔄 **Read/write access**: Components get read/write dict during execution
* 🔄 **Staged writes**: All writes staged, committed atomically on component finish
* 🔄 **Immutable snapshots**: After commit, globals become immutable snapshot
* 🔄 **Job lifetime**: Persist until job completion or fatal abort

**5.2 Conflict Resolution (Runtime)**

* 🔄 **Parallel write conflicts**: Newest commit\_ts wins (monotonic clock)
* 🔄 **Logical clock ordering**: Use logical clock for deterministic tie-breaking
* 🔄 **Audit logging**: Record old/new values and writer component

**5.3 Cross-Subjob Visibility**

* 🔄 **Downstream visibility**: Subjob globals visible after subjob\_ok/subjob\_error
* 🔄 **Export restrictions**: Read-only unless explicit global\_export in metadata
* 🔄 **Checkpoint serialization**: Serialize globals for resume reproducibility

**5.4 Joblet Global Scoping (Runtime)**

* 🔄 **Prefix enforcement**: Runtime enforces caller.\_\_internal\_\_GLOBAL naming
* 🔄 **Parent visibility**: Parent components see joblet globals as regular globals
* 🔄 **Isolation**: Joblet globals don't leak outside intended scope

**Section 6: Error Propagation (Runtime)**

**6.1 Error Propagation Matrix**

| **Upstream State** | **Downstream Edge Fires** | **Downstream Default State** |
| --- | --- | --- |
| OK | ok, if (true), parallelise | WAIT |
| ERROR | error, subjob\_error | SKIPPED |
| SKIPPED | none | SKIPPED |

**6.2 Cascade-Skip Algorithm (Runtime)**

* 🔄 **Recursive skipping**: When node reaches SKIPPED, recursively SKIP purely-dependent descendants
* 🔄 **Alternate edge check**: Don't skip if alternate inbound edge exists
* 🔄 **Component vs subjob failure**: Component ERROR → subjob ERROR → subjob\_error edges fire

**Section 7: Checkpoint/Resume (Runtime)**

**7.1 Checkpoint Persistence**

* 🔄 **Subjob granularity**: Serialize at end of every subjob
* 🔄 **Global snapshots**: Save global map state
* 🔄 **Component hashes**: Per-component output hashes for idempotency
* 🔄 **Context preservation**: Save context variables

**7.2 Resume Logic**

* 🔄 **Last checkpoint detection**: Find last OK checkpoint
* 🔄 **State restoration**: Re-hydrate globals & context from snapshot
* 🔄 **Subjob marking**: Mark completed subjobs as SKIPPED, resume from failure point
* 🔄 **Parallel resume**: Skip finished branches, restart unfinished from fork point

**Section 10: Data-Flow Runtime Behavior**

**10.1 Data Edge Execution**

* 🔄 **Queue on OK**: Sender queues output port when component reaches OK
* 🔄 **PipelineData references**: Queue PipelineData objects, not raw data
* 🔄 **Memory management**: Large frames may use shared memory or disk spill

**10.2 Multiple Input Handling**

* 🔄 **allow\_multi\_in=False**: Exactly one inbound connection per port
* 🔄 **allow\_multi\_in=True**: Accumulate arrivals into inputs[port] = [PipelineData...]
* 🔄 **Component decision**: Downstream component decides concatenate vs incremental processing

**10.3 Data/Control Coordination**

* 🔄 **Scheduler blocking**: Block downstream until data ready AND control fired
* 🔄 **Early data arrival**: Data can arrive before control, but execution waits
* 🔄 **STARTABLE exception**: STARTABLE components don't need control edges

**10.4 Back-pressure Management**

* 🔄 **Unbounded queues**: Default local runner uses unbounded async queues
* 🔄 **Enterprise limits**: Configurable max in-flight bytes/rows for enterprise runner
* 🔄 **Sender blocking**: Sender awaits consumer dequeue when limits exceeded

**Section 11: Joblet Runtime Execution**

**11.1 Joblet Runtime Behavior**

* 🔄 **Transparent execution**: Engine sees flat DAG, no special joblet logic needed
* 🔄 **Prefixed component execution**: Execute callerName.\_\_internal components normally
* 🔄 **Global variable handling**: Handle prefixed globals as regular globals

**11.2 Joblet Error Propagation**

* 🔄 **Internal error handling**: If internal error handler also fails → subjob ERROR
* 🔄 **Parent propagation**: Joblet subjob ERROR fires parent subjob\_error edges
* 🔄 **Error isolation**: Internal subjob\_ok/subjob\_error stay within joblet scope

**Section 12: Iterator Runtime Execution**

**12.1 Iterator Execution Logic**

* 🔄 **Collection processing**: Turn single input into N iterations
* 🔄 **Subjob cloning**: Clone iterator body DAG for each iteration
* 🔄 **Iteration naming**: Name iterations as iteratorName[#i]

**12.2 Iterator Control Edges**

* 🔄 **iteration\_ok**: Fire when iteration subjob state = OK
* 🔄 **iteration\_error**: Fire when iteration subjob state = ERROR
* 🔄 **iterator\_complete**: Fire after all iterations finish

**12.3 Iterator Global Handling**

* 🔄 **accumulate mode**: Merge iteration globals into parent dict
* 🔄 **reset mode**: Clear globals before each iteration
* 🔄 **override mode**: Standard last-writer-wins
* 🔄 **Error isolation**: Discard globals from failed iterations

**12.4 Iterator Failure Strategy**

* 🔄 **halt strategy**: First iteration\_error aborts remaining iterations
* 🔄 **continue strategy**: Record failures but continue, final state ERROR if any failed
* 🔄 **Empty input**: Zero elements → iterator\_complete fires immediately, state = OK

**12.5 Iterator Checkpointing**

* 🔄 **Iteration checkpoints**: Checkpoint each finished iteration
* 🔄 **Resume logic**: Skip completed iterations, restart earliest failed
* 🔄 **Global consistency**: Apply globals only from successful iterations

**Section 13: Component Startup (Runtime)**

**13.1 Startup Execution**

* 🔄 **Ready queue**: Place all STARTABLE components in initial ready queue
* 🔄 **Concurrent startup**: STARTABLE components may run concurrently
* 🔄 **Data readiness**: Still respect data-input readiness requirements

**13.2 Execution Ordering**

* 🔄 **Topological scheduling**: Purely topological, no priority weights
* 🔄 **Fair scheduling**: Don't starve branches when multiple nodes runnable
* 🔄 **Data dependency respect**: Never violate data-dependency readiness

**13.3 Context vs Global Runtime**

* 🔄 **Context immutability**: Components can read context, must not modify
* 🔄 **Global mutability**: Components can read/write globals during execution
* 🔄 **Lifetime management**: Context constant for job, globals until job end

**Addendum: Runtime-Specific Clarifications**

**A1. Joblet Error Handling (Runtime)**

* 🔄 **Unhandled failure propagation**: Silent failures inside joblet bubble to parent via subjob\_error
* 🔄 **Error state aggregation**: Multiple internal failures aggregate to single subjob ERROR

**A2. die\_on\_error Runtime Behavior**

* 🔄 **Component-level enforcement**: Respect die\_on\_error only at component level
* 🔄 **No inheritance**: No die\_on\_error inheritance between parent and joblet
* 🔄 **Cascade evaluation**: Failed component may allow subjob to continue if downstream doesn't require output

**A3. SKIPPED Component Runtime**

* 🔄 **No edge firing**: SKIPPED components fire no control edges
* 🔄 **Debug logging**: Log trace events explaining SKIPPED transitions and causes

**A4. Iterator Global Consistency**

* 🔄 **Success-only merge**: Apply globals only from OK iterations in accumulate mode
* 🔄 **Failure isolation**: Discard globals from ERROR iterations
* 🔄 **Resume consistency**: Re-execute failed iterations from last consistent state

**A5. Logical Clock Implementation**

* 🔄 **Deterministic ordering**: Use subjob\_index + event counter for global conflict resolution
* 🔄 **Tie-breaking**: Lexicographic component name order for logical clock ties
* 🔄 **Replay consistency**: Ensure deterministic global state across replays

**Implementation Priority for Engine**

**Phase 2A: Basic Execution (Days 1-3)**

1. Component state machine
2. Basic control edge firing (ok/error)
3. Data edge execution
4. STARTABLE component execution

**Phase 2B: Advanced Control Flow (Days 4-6)**

1. if condition evaluation
2. Expression evaluator integration
3. Cascade-skip algorithm
4. Error propagation matrix

**Phase 2C: Parallel Execution (Days 7-9)**

1. parallelise/synchronise implementation
2. Subjob state management
3. Global variable conflict resolution
4. Async task coordination

**Phase 2D: Special Components (Days 10-12)**

1. Iterator execution logic
2. Iterator global handling
3. Iterator checkpointing
4. Complex iteration scenarios

**Phase 3: Checkpointing (Days 13-14)**

1. Subjob boundary checkpointing
2. Global state serialization
3. Resume logic implementation
4. Parallel resume handling

**These rules focus on runtime behavior and execution semantics. They ensure the engine executes the DAG according to Talend-compatible semantics with proper state management, error handling, and parallel execution.**