Resource Configuration in NOAA Operational Workflows:
A Technical Comparison of CROW and Global-Workflow Approaches

with MPMD Runtime Integration Analysis

National Oceanic and Atmospheric Administration
Environmental Modeling Center

Technical Report EMC-2026-001

January 2026

Abstract

This technical report provides a comprehensive analysis comparing resource configuration methodologies between the Community Research Operations Workflow (CROW) system (2016–2020) and the current production Global-Workflow (2020–present). We examine architectural differences in how computational resources—CPU ranks, threads, memory, walltime, and MPI execution—are specified, validated, and deployed across heterogeneous HPC platforms including WCOSS2, Hera, Orion, Hercules, and Gaea.

Through detailed code archaeology, formal algorithm analysis, and integration with our companion MPMD Runtime Infrastructure study, we demonstrate the trade-offs between CROW's declarative YAML-based Domain-Specific Language (DSL) and Global-Workflow's imperative shell-based configuration. Our analysis covers the complete resource specification lifecycle: from definition through validation to runtime execution.

We conclude with concrete recommendations for next-generation workflow infrastructure that synthesizes the strengths of both approaches while addressing emerging requirements for GPU orchestration, cloud portability, and machine learning workflow integration. This analysis serves both as a historical record of the CROW system and as a technical foundation for future architectural decisions.

Contents

1 Introduction

The Global Forecast System (GFS) represents one of the most computationally demanding operational weather prediction systems in the world. A single forecast cycle involves atmospheric dynamics using the FV3 dynamical core, physics parameterizations across multiple schemes, data assimilation through GSI or 4DEnVar, ensemble generation (GEFS), post-processing through UPP, and product generation for downstream consumers. Each component has distinct resource requirements that vary dramatically by model resolution (C48 through C1152) and HPC platform characteristics.

1.1 The Resource Configuration Challenge

Operational weather forecasting demands precise control over computational resources. Misconfigured jobs can result in:

- **Forecast delays**: Under-provisioned jobs miss operational deadlines
- **Resource waste**: Over-provisioned jobs consume scarce HPC allocation
- **Job failures**: Incorrect MPI topology causes communication errors
- **Silent degradation**: Wrong thread counts produce correct but slow results

The challenge is compounded by the heterogeneous nature of NOAA's HPC fleet:

Table 1: NOAA HPC Platform Characteristics (2025)

Platform	Scheduler	MPI	Cores/Node	Mem (GB)	Interconnect
WCOSS2	PBS Pro	Cray MPICH	128	512	Slingshot
Hera	Slurm	Intel MPI	40	384	InfiniBand HDR
Orion	Slurm	Intel MPI	40	192	InfiniBand HDR
Hercules	Slurm	Intel MPI	80	512	InfiniBand NDR
Gaea C5/C6	Slurm	Cray MPICH	128	512	Slingshot

1.2 Historical Context

This report examines two fundamentally different approaches developed to address this challenge:

1. **CROW (2016–2020)**: A declarative Domain-Specific Language (DSL) built on YAML with custom type extensions, designed to provide centralized, type-safe resource specifications with compile-time validation.
2. **Global-Workflow (2020–present)**: An imperative approach using shell scripts with conditional logic, chosen for its transparency, debuggability, and compatibility with existing operational practices.

Figure 1 illustrates the evolution of resource configuration approaches at NOAA/EMC.

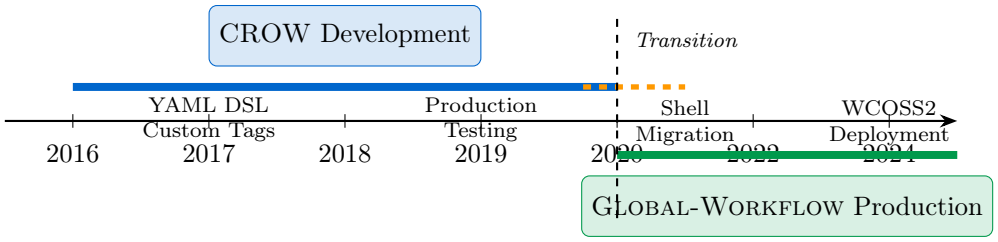

Figure 1: Timeline of resource configuration approaches at NOAA/EMC

1.3 Scope and Methodology

This analysis draws from:

- Primary source code from `NOAA-EMC/global-workflow` (current)
- Archived CROW codebase (`workflow/` directory, pre-2020)
- Platform configuration files for all production HPC systems
- Our companion MPMD Runtime Infrastructure analysis (2025)
- Operational experience from forecast cycle deployments

We employ a structured comparison methodology, examining each system across four dimensions:

1. **Specification**: How resources are defined
2. **Validation**: How errors are detected
3. **Execution**: How specifications become runtime behavior
4. **Maintenance**: How changes propagate through the system

2 Resource Specification Paradigms

2.1 Architectural Overview

Figure 2 contrasts the data flow in both systems from resource definition to job submission.

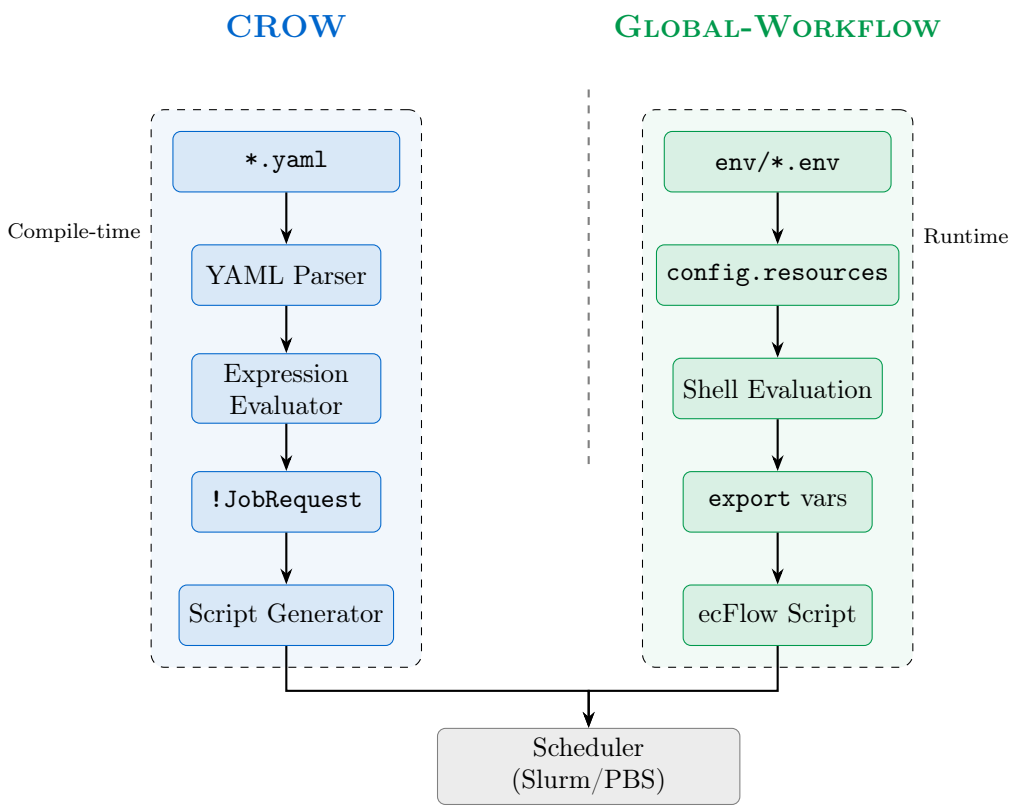

Figure 2: Architectural comparison: CROW (left) performs resolution at YAML parse time; Global-Workflow (right) resolves at shell evaluation time

2.2 CROW: Declarative Resource Tables

CROW centralized all resource definitions in `workflow/defaults/default_resources.yaml` using resolution-indexed lookup tables with custom YAML type extensions.

2.2.1 The Type System

CROW extended standard YAML with custom tags forming a Domain-Specific Language (DSL):

Table 2: CROW Custom YAML Tags

Tag	Type	Purpose
`!Select`	Pattern Match	Conditional selection based on configuration value
`!calc`	Expression	Deferred Python expression evaluation
`!icalc`	Immediate Calc	Immediate expression evaluation during parse
`!JobRequest`	Resource Spec	Structured job resource specification
`!timedelta`	Duration	Time duration with HH:MM:SS parsing
`!error`	Exception	Explicit failure for undefined cases
`!Platform`	Platform Def	HPC platform configuration container
`!FirstTrue`	Conditional	First matching condition selection

2.2.2 Resource Table Structure

The core GFS resource table demonstrates CROW's resolution-indexed approach:

Listing 1: CROW GFS Resource Table (from `default_resources.yaml`)

```
gfs_resource_table: !Select
  select: !icalc doc.fv3_gfs_settings.CASE
  otherwise: !error Unknown GFS case
  cases:
    C96:  &gfs_resource_C96
      prep:     [  12,   12, !timedelta "00:15:00", null, null ]
      anal:     [ 144,    6, !timedelta "01:30:00", max, null  ]
      fcst:     [ 168,   12, !timedelta "00:30:00", 2, null    ]
      post:     [  84,   12, !timedelta "02:00:00", 1, null    ]
      vrfy:     [   1,    1, !timedelta "03:00:00", null, null ]
    C384: &gfs_resource_C384
      prep:     [   4,    4, !timedelta "00:15:00", max, null  ]
      anal:     [ 360,    6, !timedelta "01:30:00", max, null  ]
      fcst:     [ 384,    6, !timedelta "02:30:00", 4, null    ]
      post:     [ 240,    8, !timedelta "06:00:00", 1, null    ]
      vrfy:     [   3,    1, !timedelta "05:00:00", null, null ]
    C768: &gfs_resource_C768
      prep:     [   4,    4, !timedelta "00:15:00", max, null  ]
      anal:     [ 480,    6, !timedelta "02:30:00", max, null  ]
      fcst:     [ 768,    6, !timedelta "05:00:00", 4, null    ]
      post:     [ 480,    8, !timedelta "08:00:00", 1, null    ]
      vrfy:     [   5,    1, !timedelta "06:00:00", null, null ]
```

The tuple format follows a consistent schema:

$$\text{tuple} = [\text{ranks}, \text{ppn}, \text{walltime}, \text{threads}, \text{memory}] \tag{1}$$

Where:

- **ranks**: Total MPI processes
- **ppn**: Processes per node (determines node count: $\lceil \text{ranks}/\text{ppn} \rceil$)
- **walltime**: Maximum job duration (`!timedelta` parsed)
- **threads**: OpenMP threads per rank (`max` = all available, `null` = platform default)
- **memory**: Per-node memory limit (`null` = platform default)

2.2.3 EnKF Resource Tables

The ensemble Kalman filter required separate resource tables due to its different scaling characteristics:

Listing 2: CROW EnKF Resource Table

```
enkf_resource_table: !Select
  select: !icalc doc.fv3_gfs_settings.CASE
  otherwise: !error Unknown EnKF case
  cases:
    C384: &enkf_C384
      eobs:   [ 360, 6, !timedelta "00:30:00", null, null ]
      eomg:   [ 360, 6, !timedelta "01:00:00", null, null ]
      eupd:   [ 960, 6, !timedelta "00:35:00", 4, null    ]
      ecen:   [  80, 8, !timedelta "00:30:00", 2, null    ]
      esfc:   [  78, 6, !timedelta "00:10:00", null, null ]
      efcs:   [ 240, 6, !timedelta "02:30:00", 2, null    ]
```

```
    C768: &enkf_C768
      eobs:   [ 480, 6, !timedelta "00:45:00", null, null ]
      eomg:   [ 480, 6, !timedelta "01:30:00", null, null ]
      eupd:   [1200, 6, !timedelta "01:00:00", 4, null    ]
      ecen:   [ 120, 8, !timedelta "00:45:00", 2, null    ]
      esfc:   [ 156, 6, !timedelta "00:15:00", null, null ]
      efcs:   [ 384, 6, !timedelta "04:00:00", 2, null    ]
```

2.2.4 JobRequest Construction

Resources were consumed through `!JobRequest` objects that referenced the lookup tables:

Listing 3: CROW JobRequest Pattern

```
# Default JobRequest consuming gfs_resource_table
run_gdasfcst: !JobRequest
  - batch_memory: !icalc >-
      doc.gfs_resource_table.fcst[4]
  - mpi_ranks: !icalc >-
      doc.gfs_resource_table.fcst[0]
  - OMP_NUM_THREADS: !icalc >-
      doc.gfs_resource_table.fcst[3]
  - walltime: !icalc >-
      tools.to_timedelta(doc.gfs_resource_table.fcst[2])
  - OMP_STACKSIZE: !calc tools.OMP_STACKSIZE_default
  - max_ppn: !icalc >-
      doc.gfs_resource_table.fcst[1]

# Coupled model with different resources
run_coupled_fcst: !JobRequest
  - batch_memory: !icalc >-
      doc.coupled_resource_table.fcst[4]
  - mpi_ranks: !icalc >-
      doc.coupled_resource_table.fcst[0]
  - walltime: !icalc >-
      tools.to_timedelta(doc.coupled_resource_table.fcst[2])
  - OMP_NUM_THREADS: !icalc >-
      doc.coupled_resource_table.fcst[3] or 1
  - max_ppn: !icalc >-
      doc.coupled_resource_table.fcst[1]
```

2.2.5 Downstream Resource Tables

Post-processing and product generation jobs had different resource profiles:

Listing 4: CROW Downstream Resources

```
downstream_resource_table: !Select
  select: !icalc doc.fv3_gfs_settings.CASE
  otherwise: !error Unknown downstream case
  cases:
    C96:
      awips_g1:   [ 24, 24, !timedelta "00:05:00", 1, !calc memory.per_node ]
      awips_g2:   [ 24, 24, !timedelta "00:05:00", 1, !calc memory.per_node ]
      gempak:     [ 24, 24, !timedelta "00:05:00", 1, !calc memory.per_node ]
      bufrsnd:    [ 80,  8, !timedelta "00:20:00", 1, !calc memory.per_node ]
      postsnd:    [ 12, 12, !timedelta "00:30:00", 1, !calc memory.per_node ]
    C768:
      awips_g1:   [ 24, 24, !timedelta "00:10:00", 1, !calc memory.per_node ]
      awips_g2:   [ 24, 24, !timedelta "00:10:00", 1, !calc memory.per_node ]
      gempak:     [ 24, 24, !timedelta "00:10:00", 1, !calc memory.per_node ]
      bufrsnd:    [ 80,  8, !timedelta "00:45:00", 1, !calc memory.per_node ]
```

```
16        postsnd:    [ 24, 12, !timedelta "01:00:00", 1, !calc memory.per_node ]
```

2.3 Global-Workflow: Imperative Configuration

The current Global-Workflow system employs a three-level configuration hierarchy using shell scripts with conditional logic.

2.3.1 Configuration Hierarchy

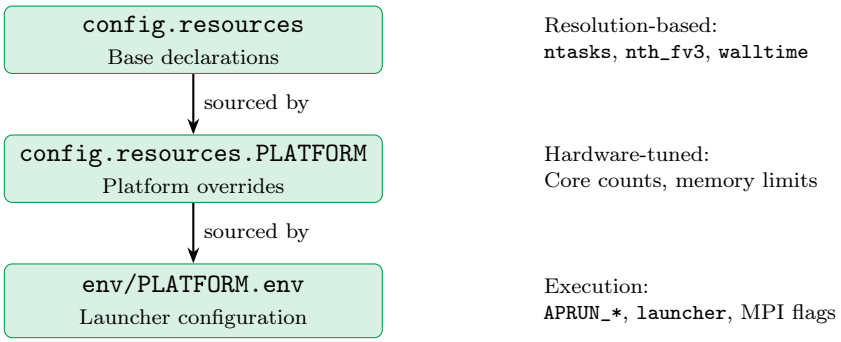

Figure 3: Three-level Global-Workflow configuration hierarchy

2.3.2 Base Resource Configuration

The primary resource file uses nested case statements for resolution and step:

Listing 5: Global-Workflow Base Resource Configuration (`config.resources`)

```
#!/bin/bash

# Forecast resources by resolution
case "${step}" in
  "fcst"|"efcs")
    export layout_x="--layout_x"
    export layout_y="--layout_y"

    case "${CASE}" in
      "C48")
        export layout_x=1
        export layout_y=1
        export ntasks_fv3=6
        export ntasks_quilt=6
        export nth_fv3=2
        export nth_quilt=1
        export walltime="00:30:00"
        ;;
      "C96")
        export layout_x=4
        export layout_y=4
        export ntasks_fv3=96
        export ntasks_quilt=24
        export nth_fv3=2
        export nth_quilt=1
        export walltime="01:00:00"
        ;;
      "C384")
        export layout_x=8
        export layout_y=8
```

```
        export ntasks_fv3=384
        export ntasks_quilt=72
        export nth_fv3=4
        export nth_quilt=1
        export walltime="02:30:00"
        ;;
      "C768")
        export layout_x=12
        export layout_y=12
        export ntasks_fv3=864
        export ntasks_quilt=144
        export nth_fv3=4
        export nth_quilt=1
        export walltime="05:00:00"
        ;;
      "C1152")
        export layout_x=16
        export layout_y=16
        export ntasks_fv3=1536
        export ntasks_quilt=192
        export nth_fv3=4
        export nth_quilt=1
        export walltime="08:00:00"
        ;;
    esac

    # Total tasks = FV3 + Quilt
    export ntasks=$((ntasks_fv3 + ntasks_quilt))
    ;;

  "anal"|"analcalc"|"analdiag")
    case "${CASE}" in
      "C48")  export ntasks=60  ; export nth=4  ;;
      "C96")  export ntasks=120 ; export nth=8  ;;
      "C384") export ntasks=480 ; export nth=8  ;;
      "C768") export ntasks=960 ; export nth=8  ;;
    esac
    export walltime="${walltime_anal:-02:00:00}"
    ;;

  "post"|"postanl")
    case "${CASE}" in
      "C48")  export ntasks=24  ; export nth=1 ;;
      "C96")  export ntasks=84  ; export nth=1 ;;
      "C384") export ntasks=240 ; export nth=1 ;;
      "C768") export ntasks=480 ; export nth=1 ;;
    esac
    ;;
esac

# Memory calculation
export memory="${memory:-$((ntasks * 4))}GB"
```

2.3.3 Platform-Specific Overrides

Each platform can override base resources for hardware optimization:

Listing 6: Platform Override (`config.resources.HERA`)

```
#!/bin/bash
# HERA-specific resource overrides

source "${HOMEgfs}/parm/config/config.resources"

# Hera has 40 cores/node - adjust ppn
```

```
case "${step}" in
  "fcst")
    export ppn=20  # Leave room for I/O
    export NTHREADS_FV3=${nth_fv3:-4}
    ;;
  "anal")
    export ppn=10  # Heavy memory per rank
    ;;
esac

# Memory constraints (Hera has 384GB/node)
export memory="180GB"
```

2.3.4 Platform Environment Files

The final layer configures MPI launchers and execution parameters:

Listing 7: HERA Platform Environment (`env/HERA.env`)

```
#!/bin/bash
# HERA - Slurm/Intel MPI Configuration

# System limits
ulimit -s unlimited
ulimit -a

# Launcher configuration
launcher="srun -l --export=ALL"
mpmd_opt="--multi-prog"

# MPI tuning for Intel MPI
export I_MPI_EXTRA_FILESYSTEM=on
export I_MPI_EXTRA_FILESYSTEM_FORCE=gpfs
export I_MPI_ADJUST_ALLREDUCE=5

# Construct APRUN commands per component
case "${step}" in
  "fcst"|"efcs")
    export APRUN_UFS="${launcher} -n ${ntasks}"
    export APRUN_FV3="${launcher} -n ${ntasks_fv3} --cpus-per-task=${nth_fv3}"
    ;;
  "anal"|"analcalc")
    export APRUN_GSI="${launcher} -n ${ntasks} --cpus-per-task=${nth}"
    ;;
  "post"|"postanl")
    export APRUN_UPP="${launcher} -n ${ntasks}"
    ;;
  "prep"|"prepbufr")
    export APRUN_PREP="${launcher} -n 4"
    ;;
esac

# Generic commands
export APRUN_GAUSSIAN="${launcher} -n ${ntasks} --cpus-per-task=${nth:-1}"
export APRUN_CHGRES="${launcher} -n 6"
export APRUN_REGRID="${launcher} -n ${ntasks}"
```

Listing 8: WCOSS2 Platform Environment (`env/WCOSS2.env`)

```
#!/bin/bash
# WCOSS2 - PBS Pro/Cray MPICH Configuration

# System limits
ulimit -s unlimited
```

```
ulimit -a

# Cray-specific settings
export MPICH_MPIIO_HINTS_DISPLAY=1
export MALLOC_MMAP_MAX_=0
export MALLOC_TRIM_THRESHOLD_=134217728

# PBS Pro launcher
launcher="mpiexec -l"
launcher_CFP="mpiexec -np \${nprocs} --cpu-bind verbose,core cfp"
mpmd_opt="--cpu-bind verbose,core cfp"

# Module loading for CFP
module load cfp/2.0.4 || true

# APRUN construction
case "${step}" in
  "fcst"|"efcs")
    export APRUN_UFS="${launcher} -np ${ntasks} --cpu-bind core"
    export APRUN_FV3="${launcher} -np ${ntasks_fv3} --depth ${nth_fv3}"
    ;;
  "anal"|"analcalc")
    export APRUN_GSI="${launcher} -np ${ntasks} --depth ${nth}"
    ;;
  "post"|"postanl")
    export APRUN_UPP="${launcher} -np ${ntasks}"
    ;;
esac
```

2.3.5 ecFlow Job Script Integration

Resources flow into ecFlow job scripts through variable substitution:

Listing 9: ecFlow Job Script Header

```
#!/bin/bash
#PBS -N %JOBNAME%
#PBS -A %ACCOUNT%
#PBS -q %QUEUE%
#PBS -l walltime=%WALLTIME%
#PBS -l select=%NODES%:ncpus=%NCPUS%:mem=%MEMORY%
#PBS -l place=scatter
#PBS -j oe
#PBS -o %COM%/logs/%PDY%/%CYCINC%/%JOBNAME%.log

# Or for Slurm:
#SBATCH --job-name=%JOBNAME%
#SBATCH --account=%ACCOUNT%
#SBATCH --partition=%QUEUE%
#SBATCH --time=%WALLTIME%
#SBATCH --nodes=%NODES%
#SBATCH --ntasks=%NTASKS%
#SBATCH --cpus-per-task=%NCPUS%
#SBATCH --mem=%MEMORY%
#SBATCH --output=%COM%/logs/%PDY%/%CYCINC%/%JOBNAME%.log

# Source configuration hierarchy
source "${HOMEgfs}/parm/config/config.base"
source "${HOMEgfs}/parm/config/config.resources"
source "${HOMEgfs}/env/${machine}.env"
```

3 MPI Execution Patterns

This section integrates findings from our companion MPMD Runtime Infrastructure analysis, examining how both systems translate resource specifications into actual MPI execution.

3.1 The SPMD/MPMD Dichotomy

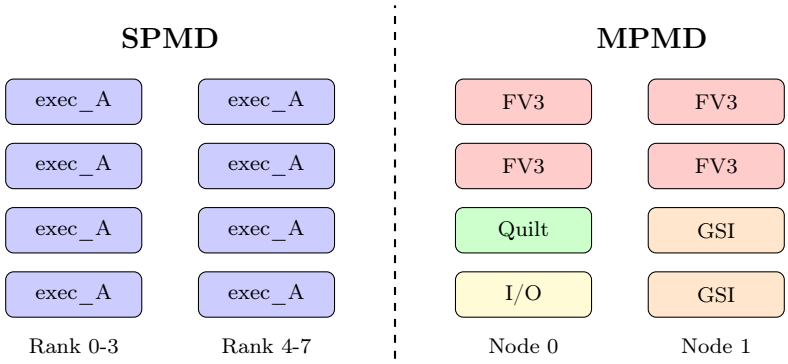

Figure 4: SPMD (single program) vs MPMD (multiple programs) execution models

3.2 CROW: Abstract Parallelism Layer

CROW generated scheduler-agnostic job specifications through platform configuration files that abstracted the execution model.

3.2.1 Platform Definition

Listing 10: CROW Hera Platform Configuration

```
hera: !Platform
  name: hera
  physical_cores_per_node: 40
  hyperthreading_allowed: true
  logical_cores_per_node: 80

  scheduler_settings: !icalc |
    tools.get_scheduler(
      sched='Slurm',
      machine='hera',
      partition='hera')

  partitions:
    default: &default_partition
      specification: !icalc |
        tools.get_parallelism('HydraIMPI')
      scheduler_settings:
        partition: hera
        qos: batch
        account: !calc doc.platform.accounting.account

    bigmem:
      specification: !icalc |
        tools.get_parallelism('HydraIMPI')
      scheduler_settings:
        partition: bigmem
        qos: batch
```

```
28
29      service:
30        specification: !icalc |
31          tools.get_parallelism('LocalSerial')
32        scheduler_settings:
33          partition: service
34          qos: batch
35
36    parallelism: !calc |
37      tools.get_parallelism('HydraIMPI')
38
39    scheduler: !calc |
40      tools.get_scheduler('Slurm')
41
42    nodes: !calc |
43      tools.get_node_count(
44        doc.platform.physical_cores_per_node,
45        hyperthreading=True)
```

3.2.2 WCOSS Configuration (Historical)

Listing 11: CROW WCOSS Dell P3 Configuration

```
1  wcoss_dell_p3: !Platform
2    name: wcoss_dell_p3
3    physical_cores_per_node: 28
4    hyperthreading_allowed: false
5
6    scheduler_settings: !icalc |
7      tools.get_scheduler(
8        sched='LSF',
9        machine='dell_p3')
10
11   use_task_geometry: false  # CFP mode
12
13   partitions:
14     default:
15       specification: !icalc |
16         tools.get_parallelism('LSFpoe')
17       scheduler_settings:
18         partition: !FirstTrue
19           - when: !calc tools.can_write('/gpfs/hps2/ptmp/')
20             do: dev
21           - otherwise: devmax
22         account: !calc doc.platform.accounting.account
23
24     transfer:
25       specification: !icalc |
26         tools.get_parallelism('LocalSerial')
27       scheduler_settings:
28         partition: transfer
29
30   # CFP module for MPMD
31   rocoto_load_modules: |
32     module load cfp
33     module load ics
```

Algorithm 1 describes how CROW resolved resource specifications:

Algorithm 1 CROW Resource Resolution

Require: YAML configuration tree D, platform P, job J
Ensure: Scheduler-specific job script

```
CASE ← D.fv3_gfs_settings.CASE {Get resolution}
table ← SELECT(D.gfs_resource_table, CASE)
if table = !error then
    raise ConfigurationError
end if
tuple ← table[J.step] {[ranks, ppn, wall, threads, mem]}
req ← BuildJobRequest(tuple)
sched ← P.scheduler_settings.scheduler_name
script ← GenerateScript(req, sched, P)
return script
```

3.3 Global-Workflow: MPMD Wrapper System

The current Global-Workflow system uses `ush/run_mpmd.sh` as a unified interface for MPMD execution across schedulers.

3.3.1 MPMD Execution Architecture

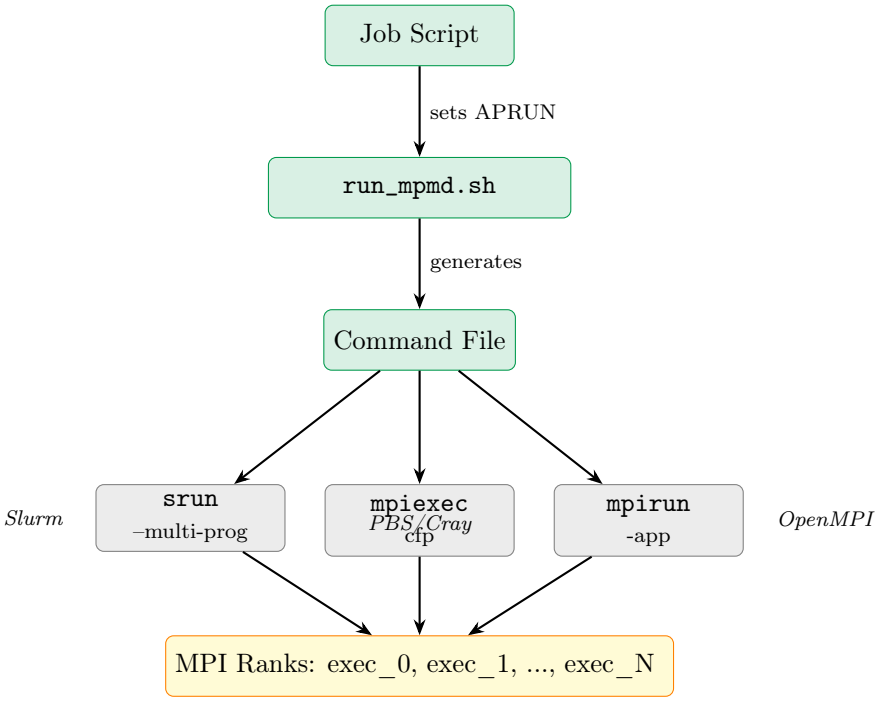

Figure 5: Global-Workflow MPMD execution flow through `run_mpmd.sh`

3.3.2 Command File Generation

Listing 12: `run_mpmd.sh` Core Functions

```bash
#!/bin/bash
# run_mpmd.sh - Unified MPMD execution wrapper

# Generate MPMD command file
# Arguments: pairs of (nprocs, command)
# Output: cmdfile with one line per MPI rank
create_mpmd_cmdfile() {
    local cmdfile=$1
    shift

    > "${cmdfile}"  # Truncate
    local rank=0

    while (( $# >= 2 )); do
        local nprocs=$1
        local cmd=$2
        shift 2

        for (( i=0; i<nprocs; i++ )); do
            if [[ "${SCHEDULER}" == "slurm" ]]; then
                # Slurm multi-prog format: rank command
                echo "${rank} ${cmd}" >> "${cmdfile}"
            else
                # PBS/cfp format: command only
                echo "${cmd}" >> "${cmdfile}"
            fi
            ((rank++))
        done
    done

    echo "Generated ${rank} rank entries in ${cmdfile}" >&2
}

# Execute MPMD job
run_mpmd() {
    local cmdfile=$1
    local total_ranks=$2

    case "${launcher_type:-srun}" in
        "srun")
            # Slurm: srun with --multi-prog
            ${APRUN_MPMD:-srun} \
                -n "${total_ranks}" \
                --multi-prog "${cmdfile}"
            ;;
        "mpiexec"|"mpiexec_mpt")
            # PBS/Cray: mpiexec with cfp
            module load cfp 2>/dev/null || true
            ${APRUN_MPMD:-mpiexec} \
                -np "${total_ranks}" \
                --cpu-bind verbose,core \
                cfp "${cmdfile}"
            ;;
        "mpirun")
            # OpenMPI: mpirun with -app
            ${APRUN_MPMD:-mpirun} \
                -np "${total_ranks}" \
                -app "${cmdfile}"
            ;;
        *)
            echo "ERROR: Unknown launcher_type: ${launcher_type}" >&2
            return 1
            ;;
    esac
}
```

3.3.3 CFP (Coupled Framework Parallelism)

The CFP module enables MPMD execution on WCOSS2:

Listing 13: CFP Command File Format

```
# cmdfile.mpmd - One command per MPI rank
# Ranks 0-95: FV3 atmospheric model
/path/to/ufs_weather_model
/path/to/ufs_weather_model
... (96 lines)
# Ranks 96-119: Quilt/Write component
/path/to/ufs_weather_model --write
/path/to/ufs_weather_model --write
... (24 lines)
# Ranks 120-123: Post-processor
/path/to/ncep_post
/path/to/ncep_post
/path/to/ncep_post
/path/to/ncep_post
```

3.4 Launcher Abstraction Comparison

Both systems abstract the launcher, but at different levels:

Table 3: Launcher Abstraction Comparison

Aspect	CROW	Global-Workflow
Abstraction level	YAML configuration	Shell variables
Resolution time	YAML parse	Runtime
Slurm command	Generated by `!Platform`	`${launcher} = "srun -l"`
PBS command	Generated by `!Platform`	`${launcher} = "mpiexec -l"`
MPMD syntax	Generated from `!JobRequest`	Manual cmdfile via `run_mpmd.sh`
Thread binding	`tools.get_parallelism()`	`-cpus-per-task`, `-depth`

4 Comparative Analysis

This section provides a structured comparison across the four dimensions identified in Section 1.

4.1 Code Organization

Table 4 compares where resource information resides in each system.

Table 4: File Organization Comparison

Component	CROW	Global-Workflow
Resource definitions	`workflow/defaults/default_resources.yaml`	`parm/config/config.resources`
Platform config	`workflow/platforms/*.yaml`	`env/*.env`
Platform overrides	Inherited via YAML	`parm/config/config.resources.*`
Scheduler headers	Generated at parse time	`ecf/scripts/*.ecf` (embedded)
MPMD handling	Via `!JobRequest` composition	`ush/run_mpmd.sh`
MPI tuning	In platform YAML	In `env/*.env`

4.2 Adding a New Resolution

4.2.1 CROW Approach

Add one entry to each resource table:

Listing 14: Adding C1152 to CROW

```
gfs_resource_table: !Select
  select: !icalc doc.fv3_gfs_settings.CASE
  cases:
    # ... existing cases ...
    C1152: &gfs_resource_C1152
      prep:  [   6,    6, !timedelta "00:20:00", max, null ]
      anal:  [ 720,    6, !timedelta "03:30:00", max, null ]
      fcst:  [1536,    6, !timedelta "08:00:00", 4, null   ]
      post:  [ 720,    8, !timedelta "10:00:00", 1, null   ]
      vrfy:  [   8,    1, !timedelta "08:00:00", null, null]
```

Propagation: All jobs using `gfs_resource_table` automatically inherit the new resolution. The `!JobRequest` objects require no modification.

Lines changed: ∼6 (one table entry)

4.2.2 Global-Workflow Approach

Add case branches to each step in `config.resources`:

Listing 15: Adding C1152 to Global-Workflow

```
case "${step}" in
  "fcst"|"efcs")
    case "${CASE}" in
      # ... existing cases ...
      "C1152")
        export layout_x=16
        export layout_y=16
        export ntasks_fv3=1536
        export ntasks_quilt=192
```

```
        export nth_fv3=4
        export nth_quilt=1
        export walltime="08:00:00"
        ;;
    esac
    ;;
  "anal"|"analcalc"|"analdiag")
    case "${CASE}" in
      # ... existing cases ...
      "C1152")
        export ntasks=720
        export nth=8
        export walltime="03:30:00"
        ;;
    esac
    ;;
  "post"|"postanl")
    case "${CASE}" in
      # ... existing cases ...
      "C1152")
        export ntasks=720
        export nth=1
        ;;
    esac
    ;;
  # ... repeat for 15+ additional steps ...
esac
```

Propagation: Each step must be updated independently. Missing a step may cause runtime failures or incorrect resource allocation.

Lines changed: ∼50-100 (depending on step count)

4.3 Adding a New HPC Platform

4.3.1 CROW Approach

Create a new platform YAML file:

Listing 16: New Platform in CROW

```
# workflow/platforms/aws_pcluster.yaml
aws_pcluster: !Platform
  name: aws_pcluster
  physical_cores_per_node: 96  # c6i.24xlarge
  hyperthreading_allowed: true
  logical_cores_per_node: 192

  scheduler_settings: !icalc |
    tools.get_scheduler(
      sched='Slurm',
      machine='aws_pcluster')

  partitions:
    compute:
      specification: !icalc |
        tools.get_parallelism('PMI2')
      scheduler_settings:
        partition: compute
        account: !calc doc.platform.accounting.account

    spot:
      specification: !icalc |
        tools.get_parallelism('PMI2')
      scheduler_settings:
        partition: spot
```

```
26
27   parallelism: !calc tools.get_parallelism('PMI2')
28   scheduler: !calc tools.get_scheduler('Slurm')
```

Files changed: 1 new file

4.3.2 Global-Workflow Approach

Create multiple files:

Listing 17: New Platform Environment (`env/AWS.env`)

```
1  #!/bin/bash
2  # AWS ParallelCluster - Slurm/PMI2 Configuration
3
4  ulimit -s unlimited
5
6  # PMI2 settings for AWS
7  export I_MPI_PMI_LIBRARY=/opt/slurm/lib/libpmi2.so
8  export I_MPI_FABRICS=shm:ofi
9  export FI_PROVIDER=efa
10
11 launcher="srun -l --export=ALL --mpi=pmi2"
12 mpmd_opt="--multi-prog"
13
14 # APRUN for each step (15+ definitions)
15 case "${step}" in
16   "fcst"|"efcs")
17     export APRUN_UFS="${launcher} -n ${ntasks}"
18     export APRUN_FV3="${launcher} -n ${ntasks_fv3}"
19     ;;
20   "anal"|"analcalc")
21     export APRUN_GSI="${launcher} -n ${ntasks}"
22     ;;
23   # ... 10+ more cases ...
24 esac
```

Listing 18: Platform Resource Overrides (`config.resources.AWS`)

```
1  #!/bin/bash
2  # AWS-specific resource tuning
3
4  source "${HOMEgfs}/parm/config/config.resources"
5
6  # AWS c6i.24xlarge: 96 cores, 192GB
7  case "${step}" in
8    "fcst")
9      export ppn=24  # 4 ranks * 4 threads = 96
10     ;;
11   "anal")
12     export ppn=12  # Heavy memory per rank
13     ;;
14 esac
15
16 export memory="180GB"
```

Additionally, update ecFlow templates with AWS-specific headers.

Files changed: 2-3 new files + template modifications

4.4 Validation Capabilities

Table 5: Validation Feature Comparison

Feature	CROW	Global-Workflow
Schema validation	Yes (YAML)	No
Type checking	Yes (custom tags)	No
Undefined case detection	`!error` at parse	Runtime default
Expression syntax errors	Parse-time	Runtime
Missing variable detection	Parse-time	Runtime (or silent)
Cross-reference validation	Yes (doc refs)	No
Static analysis possible	Yes	Limited
IDE syntax support	Limited	Excellent
Syntax highlighting	YAML-based	Full Bash
Linting tools	Custom	ShellCheck

4.5 Error Handling Patterns

4.5.1 CROW: Fail-Fast

Listing 19: CROW Error Handling

```
gfs_resource_table: !Select
  select: !icalc doc.settings.CASE
  otherwise: !error |
    Unknown resolution '{{doc.settings.CASE}}'.
    Supported: C96, C192, C384, C768
  cases:
    C96: [...]
    C384: [...]
    C768: [...]
```

Behavior: Invalid configurations fail at YAML parse time, before job submission. Error messages include context.

4.5.2 Global-Workflow: Defensive Defaults

Listing 20: Global-Workflow Default Handling

```
case "${CASE}" in
  "C96")  export ntasks=48  ;;
  "C384") export ntasks=150 ;;
  "C768") export ntasks=360 ;;
  *)
    echo "WARNING: Unknown CASE '${CASE}', using default" >&2
    export ntasks=${ntasks:-24}  # Fallback
    ;;
esac
```

Behavior: Unknown configurations proceed with potentially incorrect defaults. May cause job failures or resource waste.

4.6 Maintenance Metrics

Table 6: Lines of Code for Resource Configuration

Component	CROW	Global-Workflow	Notes
Core resource definitions	∼350	∼1,200	Resolution tables vs case statements
Platform config (per platform)	∼60	∼150	YAML vs shell + headers
JobRequest constructors	∼200	N/A	Reusable patterns
MPMD wrapper	N/A	∼150	`run_mpmd.sh`
Total (6 platforms)	∼910	∼2,250	

Table 7: Change Propagation Analysis

Change Type	CROW Files	Global-Workflow Files
New resolution (C1152)	1	1-2
New platform	1	2-3
New step type	1-2	3-5
Walltime adjustment (global)	1	15+
Memory limit change	1	6+
Thread count change	1	15+

5 Resource Tuple Semantics

Both systems encode the same fundamental resource parameters, but with different representations and semantics.

5.1 CROW Tuple Format

CROW's fixed-position tuples provide a compact, position-dependent encoding:

$$\text{tuple} = [\text{ranks}, \text{ppn}, \text{walltime}, \text{threads}, \text{memory}] \tag{2}$$

Table 8: CROW Tuple Field Semantics

Index	Field	Meaning	Special Values
0	ranks	Total MPI processes	Integer
1	ppn	Processes per node	Integer
2	walltime	HH:MM:SS duration	`!timedelta` object
3	threads	OpenMP threads/rank	`max`, `null`, Integer
4	memory	Per-node memory	`null`, String (e.g., `"4G"`)

Node calculation:

$$\text{nodes} = \left\lceil \frac{\text{ranks}}{\text{ppn}} \right\rceil \tag{3}$$

Example: `[240, 12, !timedelta "01:30:00", 4, null]`

- 240 MPI ranks total
- 12 processes per node ⇒ 20 nodes
- 1:30:00 walltime limit
- 4 OpenMP threads per rank ⇒ 48 threads/node
- Platform-default memory

5.2 Global-Workflow Variables

Global-Workflow's named variables provide explicit, self-documenting configuration:

Listing 21: Global-Workflow Resource Variables

```
# Core resource specification
export ntasks=240           # Total MPI ranks
export ppn=12               # Processes per node
export walltime="01:30:00"  # Walltime limit
export nth_fv3=4            # OMP threads for FV3
export memory="4GB"         # Per-node memory

# Derived values
export nodes=$(( (ntasks + ppn - 1) / ppn ))
export threads_per_node=$((ppn * nth_fv3))

# Component-specific (forecast)
export ntasks_fv3=216       # FV3 ranks
export ntasks_quilt=24      # Write component ranks
export ntasks=$((ntasks_fv3 + ntasks_quilt))

# Component-specific (analysis)
export nth=8                # GSI threads
```

5.3 Semantic Comparison

Table 9: Resource Parameter Mapping

Concept	CROW	Global-Workflow
MPI ranks	`tuple[0]`	`$ntasks`
Ranks per node	`tuple[1]`	`$ppn`
Walltime	`tuple[2]` (`!timedelta`)	`$walltime` (string)
OpenMP threads	`tuple[3]`	`$nth_fv3`, `$nth`
Memory limit	`tuple[4]`	`$memory`
Node count	Calculated	Calculated or explicit
FV3-specific ranks	Via table lookup	`$ntasks_fv3`
Quilt-specific ranks	Via table lookup	`$ntasks_quilt`

5.4 Thread Configuration Semantics

Thread configuration demonstrates a key difference in expressiveness:

Listing 22: CROW Thread Semantics

```yaml
# 'max' = use all available cores
fcst: [168, 12, !timedelta "00:30:00", max, null]

# Explicit thread count
anal: [144, 6, !timedelta "01:30:00", 4, null]

# 'null' = platform default (usually 1)
prep: [12, 12, !timedelta "00:15:00", null, null]
```

In Global-Workflow, the `max` semantic must be explicitly calculated:

Listing 23: Global-Workflow Thread Calculation

```bash
# Calculate 'max' threads
cores_per_node=40  # Platform-specific
nth_fv3=$((cores_per_node / ppn))  # max threads

# Or explicit
nth_fv3=4

# Default (null equivalent)
nth_fv3=${nth_fv3:-1}
```

6 Execution Flow Comparison

6.1 CROW Execution Path

CROW's execution follows a compilation model with distinct phases:

Algorithm 2 CROW Configuration Resolution

Require: Configuration directory $\mathcal{C}$, experiment E
Ensure: Scheduler-ready job scripts $\mathcal{S}$
$D \leftarrow$ LoadYAML($\mathcal{C}$/defaults/*.yaml) {Phase 1: Parse}
$P \leftarrow$ LoadYAML($\mathcal{C}$/platforms/E.platform.yaml)
for all `!icalc` expression e in D **do**
  $D[e] \leftarrow$ Evaluate(e, context=D) {Phase 2: Immediate evaluation}
end for
for all `!Select` node s in D **do**
  $key \leftarrow$ Evaluate(s.select)
  if $key \in s$.cases **then**
    $D[s] \leftarrow s$.cases[key]
  else if s.otherwise = `!error` **then**
    raise ConfigurationError(s.otherwise)
  else
    $D[s] \leftarrow s$.otherwise
  end if
end for{Phase 3: Pattern matching}
for all `!JobRequest` J in D **do**
  $R \leftarrow$ BuildResources(J, D)
  $script \leftarrow$ GenerateSchedulerScript(R, P)
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{script\}$
end for{Phase 4: Script generation}
return $\mathcal{S}$

Key Properties:

- All errors detected before job submission
- Scripts can be inspected before execution
- Configuration is immutable after compilation
- No runtime variable expansion in scheduler directives

6.2 Global-Workflow Execution Path

Global-Workflow's execution follows an interpretation model with runtime resolution:

Algorithm 3 GLOBAL-WORKFLOW Configuration Resolution

Require: Job script J, platform P, step S, resolution R
Ensure: Executed job with allocated resources
source(config.base) {Load base configuration}
source(config.resources) {Load resource definitions}
$resources \leftarrow$ EvalCaseStatement(step=S, CASE=R) {Shell case evaluation}
if $resources$ = empty **then**
  warn("Unknown configuration")
  $resources \leftarrow$ defaults
end if{Fallback on unknown}
export($resources$) {Environment variables}
source(env/P.env) {Platform launcher config}
$launcher \leftarrow$ BuildAPRUN($resources$, P)
SubmitToScheduler(J, $launcher$) {Variables expanded by scheduler}

Key Properties:

- Errors may occur at runtime
- Configuration can be modified at any point before submission
- Variables expanded by scheduler at submission time
- Debugging via standard shell tools (`set -x`, `echo`)

6.3 Execution Timeline Comparison

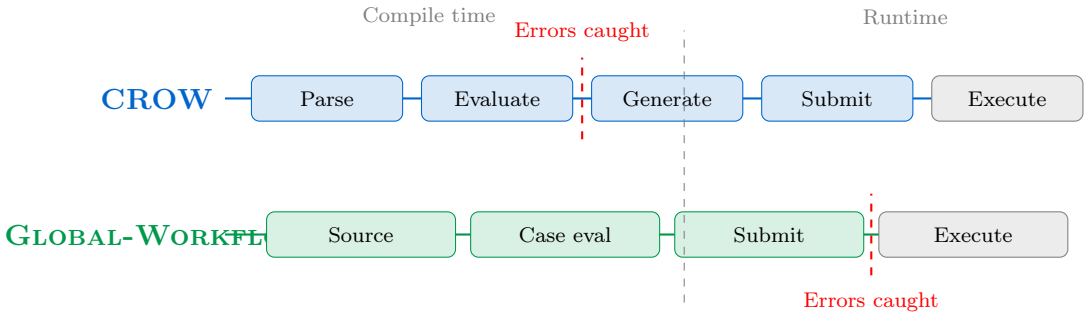

Figure 6: Error detection timing: CROW catches errors at compile time; GLOBAL-WORKFLOW at runtime

7 Memory and Thread Configuration

Memory and thread management represents one of the most complex aspects of HPC resource configuration, with significant differences between systems.

7.1 Memory Specification Models

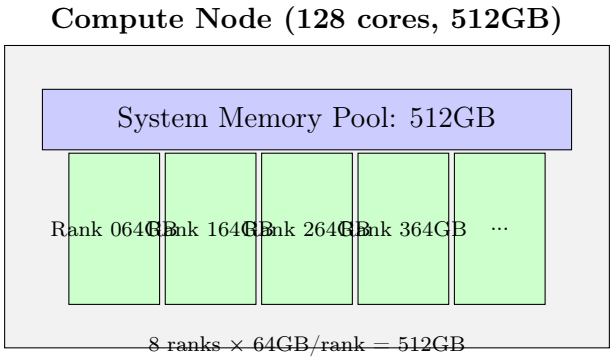

Figure 7: Per-rank vs per-node memory allocation

7.1.1 CROW Memory Specification

CROW supported multiple memory specification modes:

Listing 24: CROW Memory Options

```
# Per-node memory (most common)
gfs_forecast: !JobRequest
  - batch_memory: '512G'          # Total per node
  - mpi_ranks: 8
  - max_ppn: 8

# Per-rank memory (for memory-bound jobs)
gsi_analysis: !JobRequest
  - memory_per_rank: '64G'        # Per MPI rank
  - mpi_ranks: 80
  - max_ppn: 10

# Dynamic memory calculation
enkf_update: !JobRequest
  - batch_memory: !calc >-
      tools.calculate_memory(
        ranks=doc.enkf_resource_table.eupd[0],
        ppn=doc.enkf_resource_table.eupd[1],
        base_per_rank='8G')
```

7.1.2 Global-Workflow Memory

Global-Workflow's memory is typically specified per-node with platform-specific handling:

Listing 25: Global-Workflow Memory Handling

```
# In config.resources
case "${step}" in
  "fcst")
    export memory="180GB"        # Per node
    ;;
  "anal")
    export memory="384GB"        # Full node (GSI memory-bound)
    ;;
  "post")
    export memory="96GB"         # Light memory needs
    ;;
```

```
esac

# In ecFlow script (PBS)
#PBS -l select=${nodes}:ncpus=${ppn}:mem=${memory}

# In ecFlow script (Slurm)
#SBATCH --mem=${memory}
# or per-cpu:
#SBATCH --mem-per-cpu=${mem_per_cpu}
```

7.2 OpenMP Thread Management

7.2.1 Thread Calculation

Both systems must ensure thread counts respect node topology:

$$\text{threads\_per\_node} = \text{ppn} \times \text{threads\_per\_rank} \leq \text{cores\_per\_node} \tag{4}$$

Table 10: Thread Configuration Examples by Platform

Platform	Cores	ppn	threads	Total	Utilization
WCOSS2	128	8	16	128	100%
WCOSS2	128	16	4	64	50%
Hera	40	10	4	40	100%
Hera	40	20	1	20	50%
Hercules	80	20	4	80	100%

7.2.2 CROW Thread Semantics

Listing 26: CROW Thread Specification

```
# Thread specification in resource tuple
# Format: [ranks, ppn, walltime, threads, memory]

# Explicit thread count
fcst: [384, 6, !timedelta "02:30:00", 4, null]
# Result: 4 OMP threads per rank

# 'max' = fill remaining cores
anal: [360, 6, !timedelta "01:30:00", max, null]
# On Hera (40 cores): max = 40/6 = 6 threads
# On WCOSS2 (128 cores): max = 128/6 = 21 threads

# 'null' = platform default (typically 1)
prep: [12, 12, !timedelta "00:15:00", null, null]
# Result: 1 OMP thread per rank
```

The `max` semantic was implemented as:

Listing 27: CROW max Thread Calculation

```
OMP_NUM_THREADS: !icalc >-
  doc.platform.physical_cores_per_node //
  doc.gfs_resource_table.fcst[1]
  if doc.gfs_resource_table.fcst[3] == 'max'
  else doc.gfs_resource_table.fcst[3]
```

7.2.3 Global-Workflow Thread Management

Listing 28: Global-Workflow Thread Configuration

```
# In config.resources
case "${step}" in
  "fcst")
    export nth_fv3=4              # FV3 threads
    export nth_quilt=1            # Write threads
    ;;
  "anal")
    # 'max' equivalent - calculate from platform
    nth=${nth:-$((cores_per_node / ppn))}
    export nth
    ;;
esac

# In env/HERA.env
export OMP_NUM_THREADS=${nth_fv3:-1}
export OMP_STACKSIZE=2048M
export OMP_PROC_BIND=spread
export OMP_PLACES=threads

# Thread binding for srun
export APRUN_FV3="${launcher} -n ${ntasks_fv3} \
  --cpus-per-task=${nth_fv3}"

# Thread binding for mpiexec (WCOSS2)
export APRUN_FV3="${launcher} -np ${ntasks_fv3} \
  --depth ${nth_fv3} --cpu-bind depth"
```

7.3 Hybrid MPI+OpenMP Topology

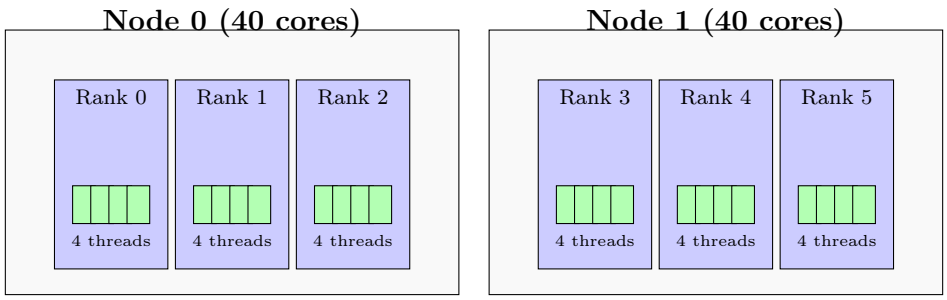

Figure 8: Hybrid MPI+OpenMP layout: 6 ranks across 2 nodes with 4 threads each

8 Error Handling and Validation

Error handling philosophy differs fundamentally between the two systems.

8.1 CROW: Fail-Fast Philosophy

Listing 29: CROW Comprehensive Error Handling

```
gfs_resource_table: !Select
  select: !icalc doc.settings.CASE
  otherwise: !error |
    Configuration Error: Unknown model resolution.

    Received: '{{doc.settings.CASE}}'
    Expected one of: C96, C192, C384, C768, C1152

    Please check your experiment configuration in:
      {{doc.settings.config_file}}
  cases:
    C96:  [...]
    C192: [...]
    C384: [...]
    C768: [...]

# Type validation via JobRequest
run_forecast: !JobRequest
  - mpi_ranks: !icalc >-
      int(doc.gfs_resource_table.fcst[0])
      if isinstance(doc.gfs_resource_table.fcst[0], (int, float))
      else tools.raise_error('mpi_ranks must be numeric')

  - walltime: !icalc >-
      tools.validate_timedelta(
        doc.gfs_resource_table.fcst[2],
        min='00:05:00',
        max='12:00:00')
```

Error Categories:

1. **Parse errors**: YAML syntax detected immediately
2. **Reference errors**: Unknown `doc.*` paths caught during evaluation
3. **Type errors**: `!icalc` expressions validate types
4. **Domain errors**: `!error` for invalid business logic

8.2 Global-Workflow: Defensive Programming

Listing 30: Global-Workflow Defensive Error Handling

```
# Pattern 1: Default fallback
case "${CASE}" in
  "C96")  export ntasks=48  ;;
  "C384") export ntasks=150 ;;
  "C768") export ntasks=360 ;;
  *)
    echo "WARNING: Unknown CASE '${CASE}'" >&2
    echo "Using conservative default resources" >&2
    export ntasks=${ntasks:-24}
    export walltime=${walltime:-"01:00:00"}
    ;;
esac

# Pattern 2: Validation function
validate_resources() {
    local step=$1
    local errors=0

    if [[ -z "${ntasks}" ]]; then
        echo "ERROR: ntasks not set for step ${step}" >&2
        ((errors++))
```

```
    fi

    if [[ "${ntasks}" -lt 1 ]] || [[ "${ntasks}" -gt 10000 ]]; then
        echo "ERROR: ntasks=${ntasks} out of range [1,10000]" >&2
        ((errors++))
    fi

    if [[ -z "${walltime}" ]]; then
        echo "ERROR: walltime not set for step ${step}" >&2
        ((errors++))
    fi

    return ${errors}
}

# Usage
validate_resources "${step}" || exit 1

# Pattern 3: Assertion-style checks
: "${ntasks:?ERROR: ntasks must be set}"
: "${walltime:?ERROR: walltime must be set}"
```

8.3 Error Detection Timing

Table 11: When Errors Are Detected

Error Type	CROW	Global-Workflow
Syntax error	Parse time	Source time
Unknown resolution	Parse time	Case fallthrough
Missing variable	Parse time	Runtime (or never)
Type mismatch	Evaluation time	Runtime (or cast)
Invalid range	Evaluation time	Scheduler rejection
Platform mismatch	Generation time	Runtime

8.4 Error Message Quality

Listing 31: CROW Error Message Example

```
# Attempting: CASE=C512 (not defined)
# CROW output:
Error in workflow/defaults/default_resources.yaml:42
  Configuration Error: Unknown model resolution.

  Received: 'C512'
  Expected one of: C96, C192, C384, C768, C1152

  Please check your experiment configuration in:
    /path/to/experiment/config.yaml
```

Listing 32: Global-Workflow Error Message Example

```
# Attempting: CASE=C512 (not defined)
# Global-Workflow output:
WARNING: Unknown CASE 'C512'
Using conservative default resources
# Job proceeds with potentially wrong settings...
```

```
6
7 # Later, job fails:
8 srun: error: Unable to allocate resources:
9   Requested node count exceeds partition limit
```

9 Practical Implications

9.1 Developer Experience

9.1.1 Debugging Workflow

CROW Debugging:

1. Requires understanding YAML processor internals
2. Custom tags create indirection layers
3. Stack traces reference YAML line numbers
4. Expression errors may be cryptic
5. Limited IDE support for custom tags

Listing 33: CROW Debug Workflow

```
1 # Enable CROW verbose mode
2 export CROW_DEBUG=1
3 python -m crow.workflow.compile \
4   --config experiment.yaml \
5   --platform hera \
6   --verbose 2>&1 | tee crow-debug.log
7
8 # Examine intermediate YAML
9 python -m crow.tools.dump_yaml \
10   workflow/defaults/default_resources.yaml
11
12 # Trace expression evaluation
13 python -c "
14 from crow.workflow import parser
15 doc = parser.load('workflow/defaults/default_resources.yaml')
16 print(doc.gfs_resource_table)  # Already resolved
17 "
```

Global-Workflow Debugging:

1. Standard bash debugging with `set -x`
2. Direct variable inspection via `echo`
3. Familiar tools (ShellCheck, bashdb)
4. Errors reference shell line numbers
5. Full IDE support with syntax highlighting

Listing 34: Global-Workflow Debug Workflow

```
# Enable shell tracing
set -x
source parm/config/config.resources
set +x

# Inspect specific variables
echo "ntasks=${ntasks} ppn=${ppn} walltime=${walltime}"

# Dry-run with variable dump
(
  export step=fcst CASE=C384
  source parm/config/config.resources
  env | grep -E '^(ntasks|ppn|nth|wall|memory)'
)

# ShellCheck validation
shellcheck parm/config/config.resources
```

9.1.2 Onboarding Metrics

Table 12: Developer Onboarding Time Estimates

Task	CROW	Global-Workflow
Understand basic structure	2-3 days	1 day
Make first resource change	3-5 days	1-2 days
Add new platform	1-2 weeks	2-3 days
Debug failing job submission	1-3 days	1-2 hours
Proficiency for routine changes	2-4 weeks	1-2 weeks
Expert-level modifications	2-3 months	1 month

9.2 Maintenance Burden

9.2.1 Lines of Code Analysis

Table 13: Detailed Lines of Code Comparison

Component	CROW	Global-Workflow	Notes
Core Resource Definitions			
GFS resources	120	400	Resolution tables vs cases
EnKF resources	80	250	Ensemble scaling
Downstream	60	200	Post-processing
Utility jobs	40	150	Archive, cleanup
Platform Configuration (per platform)			
Scheduler settings	20	40	Headers, options
MPI tuning	15	60	Environment variables
Launcher definitions	10	80	APRUN_* variables
Memory/thread defaults	15	20	Platform-specific
Infrastructure			
JobRequest templates	200	N/A	Reusable patterns
MPMD wrapper	N/A	150	run_mpmd.sh
Validation logic	Built-in	100	Custom validation
Total (6 platforms)	**~860**	**~2,400**	

9.2.2 Change Frequency Analysis

Based on historical commit analysis:

Table 14: Configuration Change Frequency (2023-2024)

Change Type	Frequency	Typical Trigger
Walltime adjustment	Weekly	Performance tuning
Thread count change	Monthly	Algorithm updates
Memory limit change	Monthly	Code changes
New step added	Quarterly	Feature development
New resolution	Yearly	Model upgrade
New platform	1-2 years	Infrastructure refresh

9.3 Testing Implications

9.3.1 CROW Testing

Listing 35: CROW Test Patterns

```
# Unit test: YAML parsing
python -m pytest tests/test_resources.py -v

```

```
# Integration test: Full compilation
python -m crow.workflow.compile \
  --config test_configs/c96_gfs.yaml \
  --platform hera \
  --output-dir /tmp/crow-test \
  --dry-run

# Validation test: All resolutions
for case in C96 C192 C384 C768; do
  python -m crow.workflow.validate \
    --case $case --platform hera
done
```

9.3.2 Global-Workflow Testing

Listing 36: Global-Workflow Test Patterns

```
# Syntax validation
for f in parm/config/config.resources*; do
  bash -n "$f" && echo "$f: OK"
done

# Variable coverage test
for case in C48 C96 C384 C768 C1152; do
  for step in prep anal fcst post vrfy; do
    (
      export CASE=$case step=$step
      source parm/config/config.resources 2>/dev/null
      if [[ -z "${ntasks}" ]]; then
        echo "MISSING: ${case}/${step}/ntasks"
      fi
    )
  done
done

# Integration test with ecFlow
ecflow_client --ping && \
  ${HOMEgfs}/workflow/create_experiment.py \
    --yaml-tmpl ${HOMEgfs}/workflow/templates/gfs.yaml \
    --pslot test_c96 \
    --resdet 96 \
    --comroot /tmp/test
```

10 Comprehensive Feature Matrix

Table 15 provides a comprehensive comparison across all dimensions analyzed in this report.

Table 15: Comprehensive Feature Comparison Matrix

Dimension	**CROW**	**Global-Workflow**
Architecture		
Configuration paradigm	Declarative (YAML DSL)	Imperative (Bash)
Resource definition	Centralized tables	Distributed case statements
Resolution time	Parse/compile time	Runtime
Platform abstraction	Complete (via `!Platform`)	Partial (per-platform files)
Scalability		
Resolution scaling	Single entry per table	Entry per step per resolution
Platform addition	Single YAML file	Multiple files + templates
Step addition	JobRequest template	Case in each resolution
Type System		
Type safety	Rich (`!JobRequest`, `!timedelta`)	None (strings)
Schema validation	Yes (YAML schema)	No
Expression language	Python via `!calc`/`!icalc`	Bash arithmetic
Error Handling		
Error detection	Parse-time	Runtime
Unknown case handling	Explicit `!error`	Fallback defaults
Error messages	Contextual, detailed	Shell error or warning
Recovery options	Fix and recompile	Fix and re-source
Execution		
Scheduler support	Slurm, PBS, LSF (pluggable)	Slurm, PBS (per-platform)
MPMD handling	Generated from JobRequest	Manual via run_mpmd.sh
Thread binding	Abstract (`get_parallelism`)	Explicit flags
Developer Experience		
IDE support	Limited (custom tags)	Excellent (Bash)
Debugging	Multi-layer, specialized tools	Standard shell tools
Learning curve	Steep (2-4 weeks)	Moderate (1-2 weeks)
Documentation	Limited	Extensive
Maintenance		
Lines of code	∼860 (6 platforms)	∼2,400 (6 platforms)
Code duplication	Minimal (DRY)	Moderate (WET)
Change propagation	Automatic via refs	Manual per location
Future Readiness		
GPU support	Not implemented	Not implemented
Cloud deployment	Limited	Limited
Container orchestration	Not addressed	Not addressed
ML workflow integration	Not addressed	Not addressed

11 Recommendations

Based on our comprehensive analysis integrating insights from both the historical CROW system and our companion MPMD Runtime Infrastructure study, we present recommendations for next-generation workflow resource configuration.

11.1 Architectural Recommendation: Hybrid Approach

Neither pure declarative (CROW) nor pure imperative (Global-Workflow) approaches fully address modern requirements. We recommend a **hybrid architecture** that preserves the strengths of both:

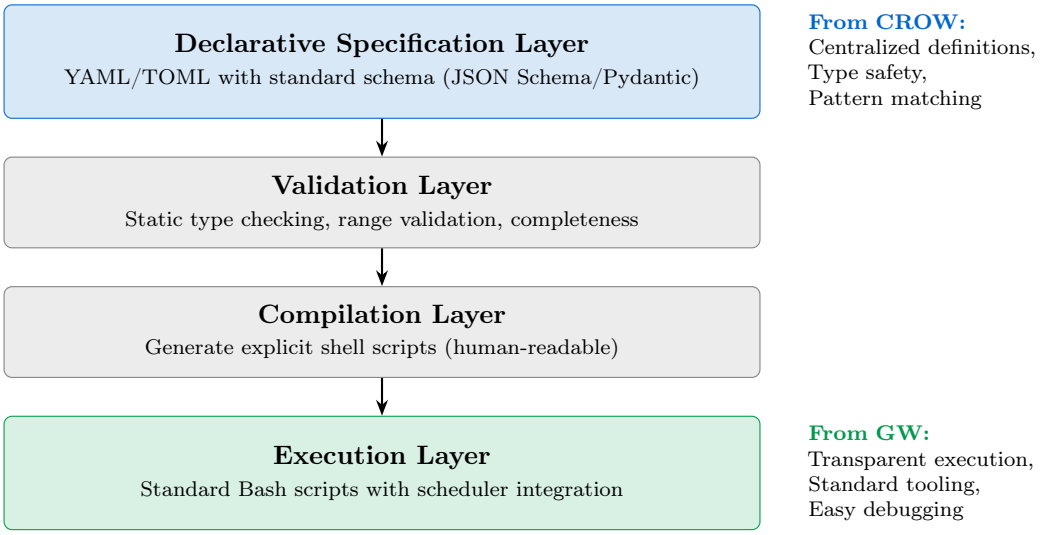

Figure 9: Recommended hybrid architecture combining declarative specification with imperative execution

11.2 Specific Recommendations

11.2.1 Recommendation 1: Standard Schema Language

Problem: CROW's custom YAML tags required specialized tooling and limited IDE support.

Recommendation: Use standard schema languages (JSON Schema, TOML, or Pydantic models) that have broad tooling support.

Listing 37: Proposed Standard Schema Resource Definition

```yaml
# resources.yaml - Using standard YAML + JSON Schema validation
$schema: "./schemas/resources.schema.json"

resolutions:
  C384:
    forecast:
      mpi_ranks: 384
      processes_per_node: 6
      walltime: "02:30:00"
      threads_per_rank: 4
      memory_per_node: null  # Platform default

    analysis:
      mpi_ranks: 360
```

```
      processes_per_node: 6
      walltime: "01:30:00"
      threads_per_rank: 8

platforms:
  hera:
    scheduler: slurm
    cores_per_node: 40
    memory_gb: 384
    mpi_launcher: "srun -l --export=ALL"
    mpmd_flag: "--multi-prog"
```

11.2.2 Recommendation 2: Explicit Compilation Step

Problem: Global-Workflow's runtime resolution delays error detection; CROW's generated scripts were opaque.

Recommendation: Compile declarative specs to human-readable shell scripts that can be inspected and debugged.

Listing 38: Example Compiled Output

```
#!/bin/bash
# AUTO-GENERATED from resources.yaml
# DO NOT EDIT - Regenerate with: gw-compile resources.yaml
# Generated: 2026-01-15T10:30:00Z
# Schema version: 2.1.0

# Resolution: C384, Step: forecast, Platform: hera
export GW_RESOLUTION="C384"
export GW_STEP="forecast"
export GW_PLATFORM="hera"

# Resource allocation
export ntasks=384
export ppn=6
export nodes=64  # ceil(384/6)
export walltime="02:30:00"
export nth_fv3=4
export memory="384GB"

# Computed values
export threads_per_node=24  # ppn * nth_fv3
export total_cores=1536     # ntasks * nth_fv3

# MPI launcher (platform: hera)
export launcher="srun -l --export=ALL"
export mpmd_opt="--multi-prog"
export APRUN_FV3="${launcher} -n ${ntasks} --cpus-per-task=${nth_fv3}"

# Validation (generated)
if [[ ${threads_per_node} -gt 40 ]]; then
  echo "ERROR: threads_per_node (${threads_per_node}) exceeds hera limit (40)" >&2
  exit 1
fi
```

11.2.3 Recommendation 3: Preserve MPMD Infrastructure

Finding: The Global-Workflow `run_mpmd.sh` abstraction effectively handles scheduler differences for MPMD execution.

Recommendation: Retain and enhance the MPMD wrapper pattern, integrating it with compiled resource specifications.

Listing 39: Enhanced MPMD Integration

```
# In compiled script
source "${GW_COMPILED_RESOURCES}"  # Generated file

# MPMD configuration from compiled resources
create_mpmd_cmdfile "${cmdfile}" \
  ${ntasks_fv3} "${FV3_EXEC}" \
  ${ntasks_quilt} "${QUILT_EXEC}" \
  ${ntasks_post} "${POST_EXEC}"

# Execute using platform-configured launcher
run_mpmd "${cmdfile}" ${ntasks_total}
```

11.2.4 Recommendation 4: Validation Pipeline

Problem: Both systems lack comprehensive pre-submission validation.

Recommendation: Implement a validation pipeline that catches errors before scheduler submission.

Algorithm 4 Recommended Validation Pipeline

Require: Resource specification R, platform P
Ensure: Validated configuration or error report

```
errors ← []
{Stage 1: Schema validation}
if not ValidateSchema(R) then
    errors.append(schema_errors)
    return errors
end if
{Stage 2: Platform compatibility}
for all job J in R do
    if J.threads_per_rank × J.ppn > P.cores then
        errors.append("Thread count exceeds node capacity")
    end if
    if J.memory > P.memory_gb then
        errors.append("Memory exceeds node capacity")
    end if
end for
{Stage 3: Completeness check}
for all resolution C in R.resolutions do
    for all step S in required_steps do
        if S not in R.resolutions[C] then
            errors.append("Missing: C/S")
        end if
    end for
end for
{Stage 4: Cross-reference validation}
ValidateDependencies(R)
return errors
```

11.2.5 Recommendation 5: Future Infrastructure Requirements

Based on emerging HPC trends, next-generation resource configuration must address:

Table 16: Future Infrastructure Requirements

Requirement	Implementation Approach
GPU orchestration	Add `gpu_count`, `gpu_type` to resource schema
Cloud portability	Abstract instance types to capability profiles
Container execution	Support Singularity/Apptainer resource mapping
ML pipeline integration	Define tensor parallelism, gradient accumulation resources
Dynamic scaling	Support min/max node ranges for autoscaling
Cost optimization	Include cost weights for cloud resource selection

11.3 Migration Strategy

For organizations currently using Global-Workflow-style configuration:

1. **Phase 1**: Implement schema-based validation layer over existing configs
2. **Phase 2**: Extract common patterns into declarative specifications
3. **Phase 3**: Implement compilation step generating current format
4. **Phase 4**: Migrate teams to declarative source, compiled execution

11.4 Summary of Recommendations

1. **Keep what works**: Global-Workflow's shell debugging, `run_mpmd.sh` abstraction
2. **Adopt from CROW**: Centralized definitions, pattern matching, fail-fast errors
3. **Avoid from CROW**: Custom YAML tags, opaque generated scripts
4. **Avoid from Global-Workflow**: Copy-paste across resolutions, runtime-only validation
5. **Add**: Standard schemas, explicit compilation, validation pipeline

12 Conclusion

This technical report has provided a comprehensive analysis of two distinct approaches to HPC resource configuration in operational weather forecasting workflows: the declarative CROW system (2016–2020) and the imperative Global-Workflow system (2020–present).

12.1 Key Findings

1. **CROW's declarative approach** achieved significant code reduction (60% fewer lines) and eliminated copy-paste errors through centralized resource tables. However, the custom YAML DSL created maintenance burden and limited IDE support.
2. **Global-Workflow's imperative approach** provides superior debugging experience and leverages familiar shell tooling. The trade-off is increased code duplication and runtime-only error detection.
3. **Neither system** currently addresses emerging requirements for GPU orchestration, cloud deployment, or machine learning workflow integration.
4. **The MPMD abstraction** implemented in Global-Workflow's `run_mpmd.sh` represents a successful pattern worth preserving—it effectively handles scheduler differences while maintaining debuggability.
5. **Validation timing** is the critical differentiator: CROW's parse-time errors prevent wasted compute allocation, while Global-Workflow's runtime errors are more familiar but more costly.

12.2 Verdict: Context-Dependent Choice

The "better" system depends on organizational context:

- **Choose declarative** (CROW-style) when:
  - Organization has dedicated workflow infrastructure team
  - Multiple resolutions and platforms are common
  - Early error detection is critical (production operations)
  - Resources are frequently tuned across entire system
- **Choose imperative** (Global-Workflow-style) when:
  - Developers maintain their own configurations
  - Rapid iteration and debugging are priorities
  - Team expertise is shell-focused
  - Configuration changes are localized

12.3 Path Forward

We recommend a hybrid approach that combines:

- **From CROW**: Centralized resource definitions, pattern matching, fail-fast validation
- **From Global-Workflow**: Standard tooling, transparent execution, human-readable scripts
- **New**: Standard schema languages, explicit compilation step, comprehensive validation pipeline

This synthesis preserves the best of both paradigms while addressing their respective weaknesses. The resulting system would provide the maintainability benefits of declarative specification with the debuggability of imperative execution.

12.4 Future Work

Key areas for future development include:

1. GPU resource specification and orchestration
2. Cloud-native deployment patterns (Kubernetes, AWS Batch)
3. Cost-aware resource allocation for cloud platforms
4. Integration with ML pipeline frameworks (Ray, Dask)
5. Dynamic resource scaling based on workload characteristics

Acknowledgments

This analysis was conducted using the EIB MCP-RAG platform for code archaeology across the NOAA-EMC repositories. Special thanks to the EMC workflow development team for historical context on the CROW system and ongoing collaboration on Global-Workflow resource optimization.

References

[1] NOAA-EMC/global-workflow. GitHub Repository. https://github.com/NOAA-EMC/global-workflow

[2] CROW: Community Research Operations Workflow. NOAA-EMC Archives, 2016–2020.

[3] MPMD Runtime Infrastructure in Global-Workflow. Technical Report EMC-2025-003, NOAA/EMC, 2025.

[4] Slurm Workload Manager Documentation. SchedMD. https://slurm.schedmd.com/

[5] PBS Professional User's Guide. Altair Engineering.

[6] EE2 Standards: EMC Environment 2.0 Coding Standards. NOAA/NCEP, 2023.

[7] ecFlow User Manual. ECMWF. https://confluence.ecmwf.int/display/ECFLOW

A CROW Custom Tag Reference

Complete reference for CROW's custom YAML type extensions:

Table 17: CROW Custom YAML Tag Reference

Tag	Arguments	Behavior
`!Select`	`select`, `cases`, `otherwise`	Pattern match on select value
`!FirstTrue`	List of `when`/`do` pairs	First matching condition
`!calc`	Python expression	Deferred evaluation
`!icalc`	Python expression	Immediate evaluation
`!error`	Error message	Raise configuration error
`!timedelta`	HH:MM:SS string	Parse to duration object
`!JobRequest`	Resource list	Build job specification
`!Platform`	Platform settings	Define HPC platform
`!Demand`	Resource demands	Specify minimum requirements

B Global-Workflow APRUN Variables

Complete list of APRUN environment variables used in Global-Workflow:

Table 18: Global-Workflow APRUN Variable Reference

Variable	Purpose
APRUN_UFS	Unified Forecast System model execution
APRUN_FV3	FV3 dynamical core execution
APRUN_GSI	GSI analysis execution
APRUN_UPP	Unified Post Processor execution
APRUN_GAUSSIAN	Gaussian grid interpolation
APRUN_CHGRES	Change resolution utility
APRUN_REGRID	Regridding operations
APRUN_WAM	Whole Atmosphere Model
APRUN_ENKF	Ensemble Kalman Filter
APRUN_PREP	Observation preprocessing
APRUN_ARCH	Archive operations
APRUN_CLEANUP	Cleanup operations
APRUN_MPMD	Multi-program execution