Unified Workflow Tools: Common Infrastructure for Portable Forecasting Workflows

Emily Carpenter and Paul Madden

Cooperative Institute for Research in Environmental Sciences at the University of Colorado Boulder

uwtools

Unified Workflow Tools (**uwtools**) provides a command-line interface (**uw**) and a Python API for automating tasks commonly needed in physical modeling, machine learning, forecast-verification, and other scientific workflows. Integrated tools support managing complex configuration files, preparing and executing component programs, staging files from a variety of sources, and setting up and driving the ecFlow and Rocoto workflow engines.

Portability

Easily installable with **conda** or **pip** on almost any system, **uwtools** supports workflow portability across HPC, cloud, and other environments. Base configs can be supplemented with platform-specific values, preserving application defaults. Component drivers (see below) provide consistent execution interfaces and support for multiple batch schedulers.

Why uwtools?

Modular
Reusable components enable flexibility and extension.

Composable
Combine component drivers to build complex applications.

Configurable
Common YAML configuration supports reproducibility.

Portable
Runs on Linux systems, from laptops to HPC and cloud.

Testable
Standardized structure and testing improve reliability.

Universal
Shared architecture supports multiple applications.

Get in touch & explore `uwtools`

Emily Carpenter: emily.carpenter-1@colorado.edu
Paul Madden: maddenp@colorado.edu
GitHub: https://github.com/ufs-community/uwtools

Configuration Management

The **uwtools** YAML language supports "Don't Repeat Yourself" sharing of values across blocks and files using Jinja2 expressions, YAML anchors/aliases, and composition. It translates between YAML, Fortran namelist, shell-script and other formats, generates ecFlow suite definitions and Rocoto XML documents from YAML, and can diff configs regardless of key order and (for namelists) case.

Here, the **config compose** tool combines multiple configs into a final app-ready config by selectively overriding values and realizing Jinja2 expressions:

```
uw config compose --realize --cycle 2026-11-23T00 base.yaml system.yaml user.yaml
```

base.yaml

```yaml
app:
  base: /path/to/app
  experiment_name: default
  rundir: '{{ app.base }}/run'
cycle: !datetime '{{ cycle }}'
leadtime: !timedelta 6
validtime: !datetime '{{ cycle + leadtime }}'
```

system.yaml

```yaml
app:
  partitions:
    netaccess: u1-service
```

user.yaml

```yaml
app:
  base: /home/myrepo
  experiment_name: nested
```

final.yaml

```yaml
app:
  base: /home/myrepo
  experiment_name: nested
  rundir: /home/myrepo/run
  partitions:
    netaccess: u1-service
cycle: 2026-11-23T00:00:00
leadtime: !timedelta '6:00:00'
validtime: 2026-11-23T06:00:00
```

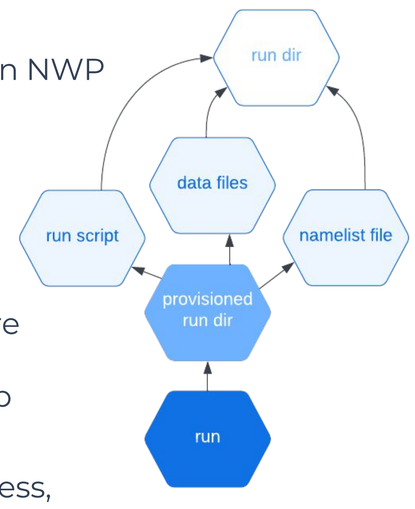
Drivers

A **uwtools** driver for a specific workflow component (often, but not always, an NWP executable) is responsible for

- Validating its configuration file against a schema
- Preparing run directories by staging data and configuration files
- Defining a component's execution resource requirements
- Executing the component locally or via a batch-system job

Drivers are implemented as graphs: Nodes are actions/assets, and edges are dependencies that imply execution order. Partial graph execution can be requested or can occur due to failure; subsequent driver execution picks up where it left off and does not repeat work.

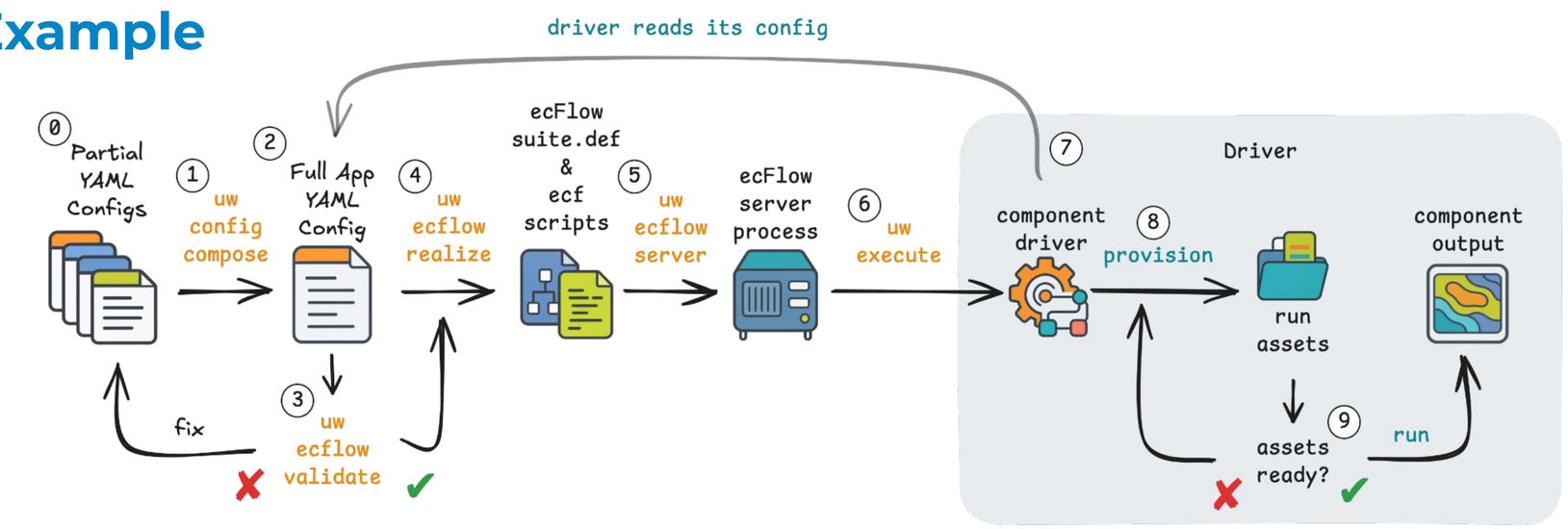
Example use case: Execute **provisioned run dir** on a node with Internet access, then execute **run** on a batch node with many cores. The **run** task does not repeat work already done by **provisioned run dir**.



Custom Drivers

The **execute** tool allows applications to define their own drivers while retaining the **uwtools** interface for configuration, validation, logging, and execution. This supports integration of application-specific components without existing **uwtools** support.

End-to-End Example



⓪ The app config is assembled from a base config with universal values, a platform config for a specific system, and user preferences. ① Partial configs are composed. ② The full config has an **ecflow** block defining an ecFlow workflow, a block configuring a component driver, and app-global values like cycle, run directory location, etc. Jinja2 expressions and YAML aliases are realized to final values. ③ The **ecflow** block is validated against an internal schema; the user can fix any reported errors. ④ The **ecflow** block guides realization of an ecFlow **suite.def** and **.ecf** scripts. ⑤ The user asks **uwtools** to start an ecFlow server, then loads the suite definition. **uwtools** selects a free TCP port, creates SSL certificates if needed, and enables ecFlow server security. ⑥ ecFlow issues **uw execute** commands to run component drivers, locally or via batch jobs. **uwtools** translates driver-config resource requirements into Slurm or PBS directives as needed. ⑦ The driver reads its config block from the full app config, which allows value sharing between drivers, workflow engines, etc. ⑧ Guided by its config, the driver provisions its run directory with data, config files, or other assets. If not all assets are ready, the driver can be retried later without repetition of work. ⑨ When the driver's assets are all ready, it runs, produces its expected output, and informs the ecFlow server that it is complete.

This research was supported in part by NOAA cooperative agreement NA22OAR4320151 and the OCIO Software Engineering for Novel Architectures (SENA) program.