Updated on August 29, 2025

PI17: Plan for improving the UFS-WM software process.

ECC-1743

Date	PWS	Changes
08/29/25	4.7.2	Initial version

Per 4.2.4, An improved UFS-WM software process by shifting UFS-WM components toward unit testing, libraries and enforcing code quality in collaboration with the UFS System Architecture and Infrastructure Cross-Cutting team.

Due - A plan by the end of the first PI in the PoP. Process description and procedures in Confluence pages and available to stakeholders in the PoP following the milestones table specified in the PMP. - Online and selected cloud platforms resources

PI17: Plan for improving the UFS-WM software process

The Unified Forecast System (UFS) is a powerful and adaptable framework designed to simulate the coupled Earth physical processes using a suite of tightly coupled modeling configurations. At its foundation is a robust library of interoperable components that support precise, scalable, and flexible Earth system modeling.

UFS is built on a sophisticated architecture of interconnected repositories and subrepositories—many of which are shared across multiple applications—forming a modular ecosystem that promotes reuse and collaboration. The UFS Weather Model alone includes 14 submodules, some branching into deeper nested layers, underscoring the system's complexity and extensibility.

Most UFS applications are organized around a top-level workflow repository, hosted under the ufs-community organization on GitHub. This centralized structure enhances coordination, simplifies version control, and accelerates development across diverse teams.

To foster innovation, developers must be able to build and test components independently—without being constrained by infrastructure, libraries, or high-level dependencies. At the same time, the new workflow must deliver equal or greater functionality while remaining intuitive and user-friendly, reducing the barrier to entry and streamlining the software engineering experience.

1. Transformation of UFS Weather Model Development Workflow with Spack-Based Infrastructure

Using Spack within the UFS development ecosystem offers several key advantages:

- Developers can manage individual components independently, enabling flexible combinations of modules and dependencies. This modularity supports multiple versions derived from a single source, enhancing reproducibility and experimentation.
- Library dependencies can be resolved by referencing upstream Spack-stack environments, streamlining integration and reducing configuration overhead.
- Spack-stack site configurations and build options can be leveraged to ensure consistency across platforms and to simplify deployment.
- Component level unit testing can be incorporated into the build process, improving reliability and catching issues early.
- Developers retain the freedom to build and customize their own libraries—such as FMS—without being locked into centralized infrastructure.
- Spack integrates smoothly with the UFS CMake build system, making it easier to adopt without disrupting existing workflows.

Despite its advantages, integrating Spack with the UFS development environment presents several challenges:

- The UFS CMake build system does not automatically detect changes in individual modules. As a result, developers must manually instruct the build system to rebuild affected components, which can lead to inefficiencies and missed updates.
- This issue is mitigated when the entire build process is managed within the Spack framework. Adopting Spack as the primary build system could offer a long-term solution that supports community releases, operational deployments, and broader collaboration.
- However, using `spack install` for development packages is similar to running `make`—it does not always capture deeper code changes. In such cases, developers must perform additional steps, such as running `spack clean` or re-concretizing the environment, to ensure the build reflects the latest modifications.

2. Component-Level Unit Testing

Incorporating component-level unit testing into the UFS development workflow is a natural extension of Spack's modular design approach. By treating each UFS subcomponent—such as FV3, UPP, FMS, CDEPS, or stochastic physics—as an independently testable unit, developers can validate functionality early and isolate regressions before full system integration.

Benefits of Component-Level Testing with Spack:

- Early Detection of Errors: Unit tests catch bugs at the source, reducing the cost and complexity of downstream debugging.

- Modular Validation: Each component can be tested against its own interface contracts, ensuring internal consistency and robustness.
- CI/CD Integration: Spack environments can be paired with GitHub Actions or Jenkins CI to trigger tests automatically on pull requests or commits.
- Platform Consistency: By leveraging Spack-stack site configurations, unit tests can be executed uniformly across RDHPCS platforms or developer workstations.
- Customizable Test Suites: Developers can tailor tests to specific builds, compiler flags, or physics configurations, enabling targeted validation without running a full high level regression test suite.
- Example Workflow:
    - spack dev-build ufs-weather-model@develop +unit_tests
    - ctest -R fms_unit_tests

This approach allows developers to run focused tests (e.g., `fms_unit_tests`) without invoking the full model build or regression suite. Over time, this can evolve into a layered testing strategy—unit, integration, system—aligned with operational reliability goals.

Challenges and Considerations:

- Test Discovery: Legacy components may lack standardized test entry points. Refactoring or wrapping legacy code may be required.
- Build Coupling: Some components (e.g., stochastic physics) are tightly coupled to FV3 and may require mock interfaces or stubs for isolated testing.
- Maintenance Overhead: Unit tests must evolve alongside the codebase. Spack's versioning helps manage this, but discipline in test upkeep is essential.

3. PI-18/PI-19 MVP Plan: UFS UPP Spack-Stack Conversion

Enable UPP to be built, tested, and deployed via Spack-stack as a modular component of the UFS ecosystem, supporting both standalone and coupled workflows. This MVP will lay the groundwork for broader UFS component Spack-stack adoption. Key technical tasks are:

- Spack Package for UPP: Create a modular Spack package for UPP with support for NetCDF, MPI, and optional graphics libraries.
- CMake Integration: Refactor UPP's build system to support Spack-driven CMake flags and external dependency resolution.
- Unit Test Harness: Introduce component-level unit tests for GRIB decoding, interpolation, and formatting routines using `ctest`.
- Baseline Comparison Tool: Develop a lightweight Python tool to compare GRIB2 outputs against known baselines for regression validation.
- Platform-Agnostic Build Validation: Ensure builds work across Hera, Orion, Jet, and containerized environments using Spack-stack site configurations.
- Documentation & CI Hooks: Provide developer documentation and integrate GitHub Actions or Jenkins for automated Spack builds and test execution.

2. Strengthen Agile Testing and Validation Process

To meet the rigorous demands of operational forecasting, the Unified Forecast System (UFS) employs a robust continuous regression testing framework. Every code integration—whether a minor tweak or a major overhaul—is subjected to a comprehensive suite of tests that evaluate:

- Scientific accuracy
- Computational performance
- System robustness
- Code reliability across platforms

These tests are validated using bit-for-bit reproducibility across a wide spectrum of parameters, including:

- Application types and coupled model configurations
- Spatial resolutions and physics suites
- Compiler versions and build options
- Computing platforms and threading models
- Parallel decomposition strategies and I/O restart configurations

Figure 1 illustrates the stacked wall-time required to complete a full UFS Weather Model regression test run across NOAA Tier-1 RDHPCS platforms (Hera, Ursa, GaeaC6, Orion, Hercules, Derecho) and WCOSS2 operational systems. This visual highlights the computational load and time distribution across systems. Figure 2 illustrates the substantial number of regression test cases executed on each NOAA RDHPCS and WCOSS2 platform, including Hera, Orion, Derecho, and others. The consistently high test counts across platforms reflect the extensive validation efforts required to ensure model reliability and performance in diverse computing environments.

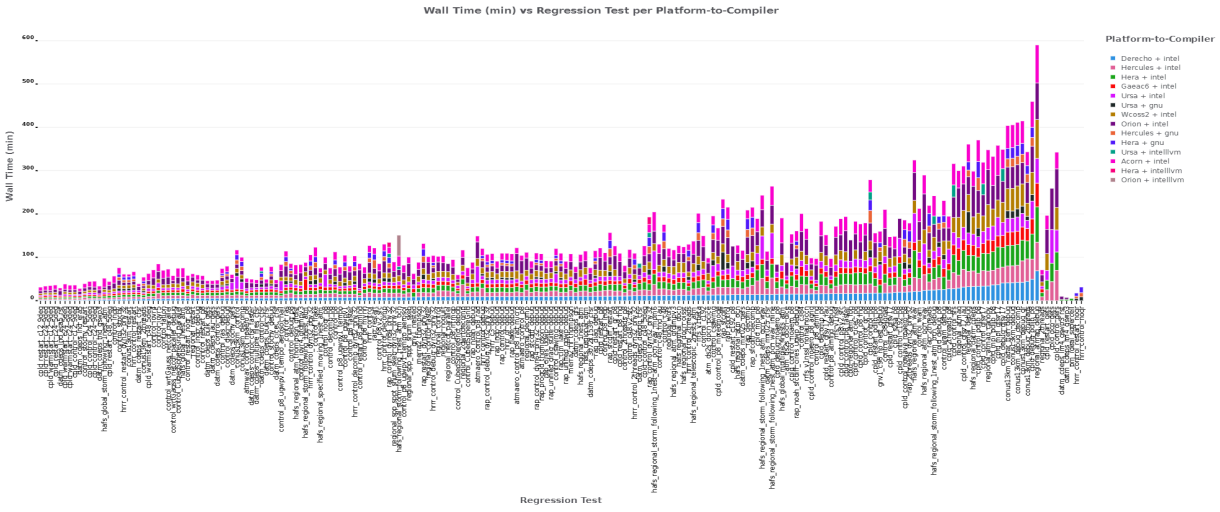

Figure 1. *Stacked wall-time for UFS Weather Model regression testing across NOAA RDHPCS and WCOSS2 platforms.*

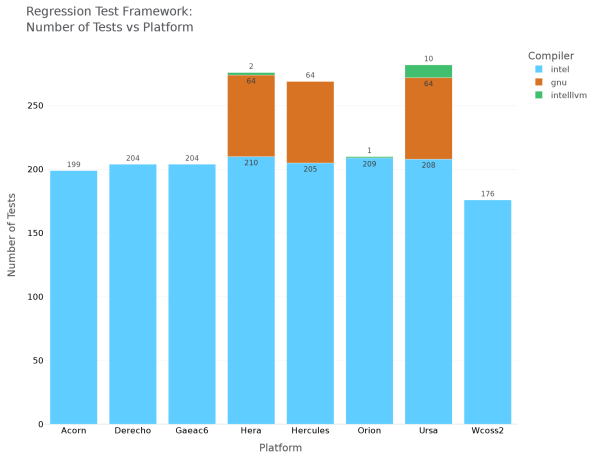

Figure 2. *High-volume distribution of UFS Weather Model regression test cases across platforms.*

Despite the thoroughness of the current testing strategies, its uniform application across all platforms can introduce bottlenecks. Delays often arise from:

- System instability
- Incomplete readiness of submitted code
- Failures triggered by platform-specific issues

These challenges slow down pull request processing and reduce development velocity. To mitigate this, the team is exploring more adaptive testing strategies that prioritize critical paths and isolate platform-specific failures.

We propose a platform-specific optimization strategy for daily and nightly regression runs, aiming to enhance testing efficiency and accelerate feedback cycles. By analyzing the test case distribution presented in Figure 1, we plan to reorganize and allocate test cases across platforms to maximize parallel execution and minimize overall runtime.

To achieve this, future test strategy needs to be guided by the following objectives:

- Limit total test cases and test time to under two hours per platform to ensure rapid turnaround.
- Maximize platform parallelism and resource utilization by tailoring test distribution to each system's capabilities.
- Conduct platform profiling to assess compute capacity—including core count, memory, and queue behavior—in order to estimate optimal test throughput.
- Categorize test cases based on runtime, resource demand, and priority:
    - *Short tests* (less than 5 minutes): Ideal for quick validation cycles.
    - *Medium tests* (5 to 20 minutes): Provide balanced coverage across platforms.
    - *Long tests* (over 20 minutes): Scheduled less frequently, prioritized for nightly runs or distributed across multiple platforms.

- Implement threshold monitoring to track nightly run performance, flag test overruns, and dynamically adjust test execution plans based on observed behavior.

4. Consolidated Testing Environment and Containerization *(Long-Term Goal)*

The UFS Weather Model currently maintains over 25 distinct `modulefiles`, each customized for a specific platform and compiler configuration—examples include `ufs_hera.intel.lua`, `ufs_ursa.intelllvm.lua`, and `ufs_derecho.intel.lua`. While this approach accommodates diverse HPC environments, it introduces several long-term challenges:

- Platform-specific build logic that complicates cross-system development
- Inconsistent compiler and MPI versions leading to variability in performance and results
- Limited reproducibility, especially when migrating workflows between systems

Containerization offers a strategic path forward by encapsulating the build and runtime environment into a unified, portable image. This abstraction eliminates the need for machine-specific configurations and enables a more streamlined development and testing process.

Key advantages include:

- Consistency: A single container image ensures reproducible builds and outputs
- Portability: Seamless execution across HPC clusters, cloud platforms, and local machines
- Simplified maintenance: Reduces the overhead of managing numerous modulefiles

The consolidated approach of the EPIC container transition plan (https://docs.google.com/document/d/1QRo6NFuHCxpwatff9nhb6vwcZj5foO9zgyoSgtdKYs4/edit?tab=t.0#heading=h.csvio0e6xmvm) will lay the foundation for a more scalable and maintainable UFS Weather Model and application development ecosystem.