# Enabling Command-and-Control in Advanced In Situ Workflows

Kshitij Mehta
Oak Ridge National Laboratory
Oak Ridge, TN, USA

Eric Suchyta
Oak Ridge National Laboratory
Oak Ridge, TN, USA

Frédéric Suter
Oak Ridge National Laboratory
Oak Ridge, TN, USA

Ana Gainaru
Oak Ridge National Laboratory
Oak Ridge, TN, USA

Norbert Podhorszki
Oak Ridge National Laboratory
Oak Ridge, TN, USA

Scott Klasky
Oak Ridge National Laboratory
Oak Ridge, TN, USA

## Abstract

Scientific discovery is progressing towards autonomous science with the combination of scientific instruments, high-performance computing, and artificial intelligence in complex workflows. This evolution introduces new requirements for managing scientific workflows, including feedback loops, near real-time constraints, and the ability to dynamically control workflow execution. In situ workflows that analyze and visualize data as it is generated are well-suited to satisfy stringent time constraints and their iterative nature offers greater opportunities for *command-and-control*. However, only a few of the many workflow management systems available have been specifically designed to manage in situ workflows and often lack support for automated feedback loops that allow analysis and visualization components to interact with the main scientific data producer. To address this need, we present in this paper how to add command-and-control capabilities to a workflow management system. We identify the functional design requirements of such a command-and-control system, detail its architecture, interface, and core mechanisms, and illustrate how advanced in situ workflows can leverage command-and-control in three use cases: graceful termination with checkpoint, dynamic and adaptive data reduction, and event-triggered analysis.

## CCS Concepts

• **Computing methodologies** → **Distributed computing methodologies**.

## Keywords

Command-and-Control, In situ Workflows, autonomous science

**ACM Reference Format:**
Kshitij Mehta, Eric Suchyta, Frédéric Suter, Ana Gainaru, Norbert Podhorszki, and Scott Klasky. 2025. Enabling Command-and-Control in Advanced In Situ Workflows. In *54th International Conference on Parallel Processing Companion (ICPP Companion '25), September 08–11, 2025, San Diego, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3750720.3757301


*ICPP Companion '25, San Diego, CA, USA*
2025. ACM ISBN 979-8-4007-2109-0/25/09
https://doi.org/10.1145/3750720.3757301

## 1 Introduction

Achieving new scientific discovery often requires combining several analyses and visualizations to examine complex phenomena emerging from the data produced by scientific instruments [3] or during the execution of a first principle physics simulation [18]. This capacity to analyze and visualize data as it is generated is termed *in situ* processing [5, 7], in opposition to the *post hoc* processing paradigm in which analyses are performed only after an entire dataset has been produced. The resulting in situ workflows allow researchers from various scientific domains to cope with the explosion in data volume and velocity combined with the growing discrepancy between storage subsystems performance and computing power in extreme-scale supercomputers. They also provide them with additional *command-and-control* capabilities and eventually reduce the time to science.

Command-and-control broadly refers to the ability to direct, coordinate, and regulate the behavior of a system or set of components, usually dynamically and in near real-time, to achieve certain high-level goals or enforce predefined rules [2]. In the specific context of scientific workflows, one of the most widespread forms of command-and-control is computational steering, which consists of breaking the traditional prepare-execute-analyze cycle and allows scientists to modify experiment parameters at runtime and receive immediate feedback on the effect of that change. This approach increasingly relies on Artificial Intelligence (AI) to automate the steering process and connect experimental and high-performance computing (HPC) facilities to form complex AI-coupled HPC workflow motifs [6] which are at the core of the autonomous science roadmap [9]. In situ simulation-analysis workflows add additional needs for command-and-control such as dynamically spawning a new analysis, changing the flow of data exported by the main simulation component, replacing a compute heavy component by a lighter surrogate model [8], or changing the level of accuracy at which data is exported thanks to data reduction techniques to accelerate the analysis process.

While a plethora of scientific workflow management systems (WMSs) has been proposed over the last two decades [1], only a few of these tools implement command-and-control features or have been specifically designed to manage in situ workflows. In particular, most WMSs lack support for automated feedback loops that allow the analysis and visualization components of advanced in situ workflows to interact with the main simulation component [20].

To address this need, we introduce in this paper an original WMS-agnostic command-and-control system and make the following contributions:

- We identify the functional requirements that drive the design of a command-and-control system interacting with an advanced in situ workflow;
- We detail the client-server architecture of a command-and-control systems and how it interfaces with both the WMS and individual workflow components;
- We introduce an API and built-in signals to implement the *monitor-detect-trigger* cycles at the core of the command-and-control system;
- We illustrate how advanced in situ workflows can leverage the proposed command-and-control system on three commonplace use cases.

The remainder of this paper is organized as follows. Section 2 details the proposed command-and-control system and a prototype implementation. Then, we provide preliminary results on different use cases in Section 3. Section 4 reviews the related work before we conclude and present future work directions in Section 5.

## 2 Enabling Command-and-Control in Advanced In Situ Workflows

In a typical execution, the activity of an advanced in situ simulation-analysis workflow is driven by the frequency at which the main simulation component produces data. Consumer analysis and visualization components are then deemed to act on the data when it becomes available. Moreover, users typically must wait for the end of the entire analysis to investigate results and errors, decide how to modify configuration parameters, or whether they must halt the whole workflow if an issue is detected. Existing science workflows usually lack the ability for a data consumer application, the workflow user, or the WMS to dynamically act on the workflow structure or data production and alter the current execution plan. Some WMSs offer the capability to declare conditional statements [4, 17] or even loops [15] thus allowing for the dynamic adaptation of workflow execution to runtime exceptions or specific task results. However, benefiting from such command-and-control-enabling features requires workflow practitioners to lock-in with a specific WMS that may be ill-suited to their scientific application on other aspects. In this paper, we advocate for the implementation of a standalone WMS-agnostic command-and-control system that brings the desired features with a greater flexibility than extending an existing WMS to enable command-and-control.

### 2.1 Functional Design Requirements

The design of a command-and-control system that interacts with an advanced in situ workflow is influenced by several functional requirements and considerations:

1. A command-and-control system must establish *communication channels* between the workflow management system and the workflow components (i.e., the simulation that produces data and the analysis and visualizations that consume data). It must also define a set of signals for different events to react to and mechanisms (e.g., user-defined signal handlers and callback functions) to trigger the corresponding actions.
2. Define the *scope of execution* of a given control action. Advanced in situ workflows usually involve parallel applications using message passing to coordinate their execution across multiple processes. The command-and-control system must thus specify if control actions apply to a subset or all the processes that execute this parallel application.
3. *Control points* must be inserted in the workflow components where a runtime command has to be performed. A classical approach, used by operating systems for instance, is to use signal-based interrupts to trigger actions. The context switch caused by handling an interrupt acts as a control point at which a callback function can be executed. For parallel science applications, such control points must be consistent across all processes. If different processes are executing different instructions when an interrupt occurs, it could leave the application in an inconsistent and damaged state.
4. Create and maintain a *registry* for applications and signals. Users must register simulation, analysis, and visualization components with the WMS, enabling it to correctly launch and manage these tasks. An *ontology* of signals must also be defined, specifying the signal's source, recipient, and associated action. For example, an external analysis component (source) may send a signal to the main simulation component (recipient) instructing it to terminate its execution (action).
5. The command-and-control system must incorporate a *decision engine* capable of receiving, analyzing, and forwarding signals among workflow components. A decision engine provides the capability to execute complex control logic depending on the received signal, such as launching and terminating external applications. For example, if an external analysis code signals the main science application to terminate, the decision engine must instruct other dependent workflow components, such as an external visualization routine, to terminate as well.
6. Finally, the system must provide a *lightweight API* to initialize the communication channels, define control point(s), and send signals across workflow components. This API must allow workflow components, especially the main scientific data producer, to register "tunable knobs" to dynamically adapt their execution flow at runtime, while minimizing the instrumentation overhead.

### 2.2 Architecture

Figure 1 shows the architecture of the proposed command-and-control system. This client-server system is directly connected to the application launcher, a component of the workflow management system, to provide the connection points and communication channels to interact with the workflow components.

The server side of the command-and-control system handles and validates incoming signals before dispatching them to the corresponding workflow components. It is composed of a *decision engine* that executes actions based on runtime events. It also includes a *heartbeat monitor* that periodically checks for the continued progress of the workflow components.

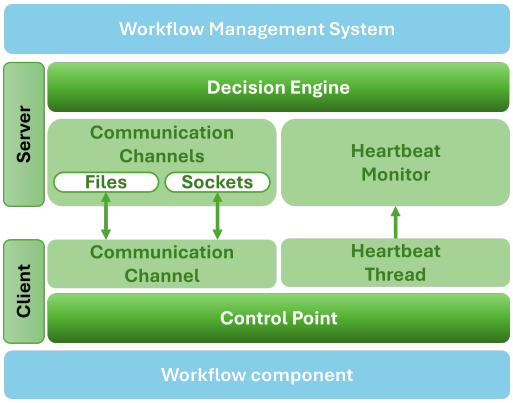


**Figure 1: Architecture of the command-and-control system.**

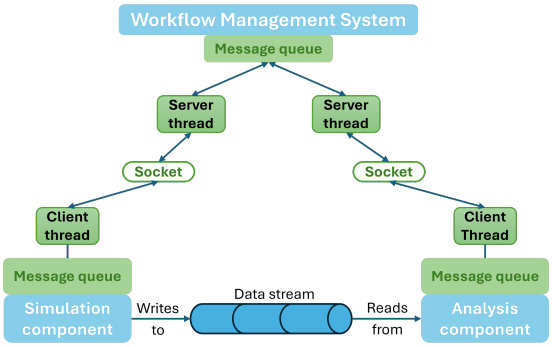


**Figure 2: Communication infrastructure of the command-and-control system. The workflow management system launches the workflow components and server threads that communicate with client threads spawned by the root process of these workflow components. The communication plane consists of socket communication and message queues.**

The clients of the command-and-control system are attached to individual workflow components. They manage the *control points* that define when command-and-control operations occur in the flow of a given workflow component. They also include a heartbeat thread that periodically sends heartbeat messages to the server module. The configurable timeout within which a heartbeat must be received is denoted as the component's 'heart rate'. Note that if the WMS with which the command-and-control system interacts already implements such a heartbeat monitor, or another feature provided by the command-and-control system, it can be either deactivated to prevent redundancy or kept active to increase resilience.

The interaction between the server and multiple clients is enabled by setting *communication channels*. These channels can either rely on a file-based mechanism or by using socket connections between processes. Figure 2 shows the communication infrastructure of the proposed command-and-control system, using sockets and message queues. First, the system creates a set of server and client threads for every component of the workflow. Server threads are launched by the server module associated with the workflow management system, while client threads are spawned by the root process (e.g., MPI process of rank 0) of each workflow component.

The use of socket connections requires all the components of the in situ workflow to be executed concurrently on compute resources that can communicate over the network (e.g., within a single batch job on an HPC system). File-based mechanisms offer greater flexibility in terms of resource allocation as long as access to a shared storage space is granted to independent, isolated, batch jobs. However, concurrent execution is still needed, but not mandatory, to ensure the usefulness of command-and-control actions.

In this illustrative example, we consider an in situ workflow simply composed of two components: a parallel science simulation that uses MPI and a separate analysis component that processes data produced by this simulation. The workflow management system acts as a message broker between the workflow components and controls the entire workflow. The different server threads communicate with each other over shared message queues.

When the analysis component detects a pre-defined condition in the data stream it receives from the simulation component, its client thread triggers an action by sending the associated signal to its server thread. The server module then performs a series of checks on the signal and delivers it to the server thread connected to the simulation component, which in turn delivers the signal to its corresponding client thread. The simulation component can then execute the action associated with the signal.

## 2.3 API and Built-in Signals

To minimize the amount of modification required in application codes to enable command-and-control, we propose the following minimal set of API functions for setting up the environment and defining control points of the command-and-control system:

- `ccs_init` is called by every workflow component to set up the command-and-control system. It spawns a client thread on the root process of each workflow component and creates a message queue shared between this root process and the client thread for communication. The client thread also establishes a communication channel with the server thread spawned by the WMS. Client and server threads perform a simple handshake to complete the setup.
- `ccs_check` defines a control point within a workflow component. When called, the client thread of that component checks for pending signals and executes runtime actions by triggering the callback functions associated with the signals. Users can instrument their scientific application code with this function call as they see fit (e.g., once or multiple times every timestep loop iteration, or once every few timestep iterations). The scientific application must also provide references to the callback functions that are executed to service the signal (e.g., checkpointing and cleanup when the application must be safely terminated).
- `ccs_heartbeat` allows a workflow component to send a 'heartbeat' to the heartbeat monitor of the command-and-control system. It must send this heartbeat within every '`heart_rate`' seconds. The heart rate is implemented internally as a single integer that is sent by the client thread on the workflow component. The heart rate is configurable and is set by the user when the component is created. Calling

the ccs_check function automatically sends an additional heartbeat to the server. If a heartbeat is not received within the pre-defined heart_rate timeout interval, the system assumes that the workflow component has failed and terminates the associated process. This mechanism ensures that unresponsive or hung applications are killed and do not continue to consume compute resources.

- ccs_signal is called by the command-and-control client of a workflow component, typically by the one consuming data (i.e., analysis or visualization component) to signal the command-and-control server that a triggering event (e.g., the conditions of an imminent critical failure, an insufficient level of accuracy in a data stream) has been detected.
- ccs_finalize is called by all workflow components to perform a cleanup, terminate the client threads, and close the communication channels with the server module.

Figure 3 illustrates how to insert a control point in the time-stepping loop of an in situ workflow with a call to ccs_check. When entering this function, only the client thread of the root process checks for any pending signals delivered asynchronously from its corresponding server thread. Other processes running that component wait for a message from the root process. If a signal has been received, the client command-and-control thread of the root process broadcasts this signal to the client threads of all the other processes, which in turn execute the action associated with this signal. Otherwise, it broadcasts a message to notify all processes to continue with their execution.

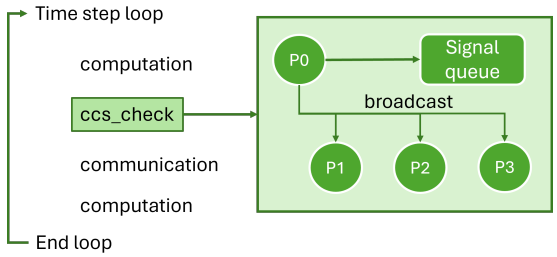


**Figure 3: Control flow of the ccs_check function. The root process checks the signal queue for a pending signal and broadcasts it to all the other processes.**

The broadcast operation at the core of a control point creates a synchronization point among the processes running the workflow component. Its overhead can become substantial and create a performance bottleneck in the execution flow of the in situ workflow if invoked too frequently (e.g., multiple times in a single loop iteration). To mitigate the effect of this overhead, we encode signals as short messages containing a 4-byte integer, thereby reducing the broadcast time. Nevertheless, user applications must control how frequently a control point is invoked.

The proposed command-and-control system comes with the following minimal set of built-in signals:

- CCS_CLIENT_READY: Used by an application's client thread for handshake while setting up a communication channel with its corresponding server thread.
- CCS_SIGTERM: Indicate a workflow component, i.e., the main simulation component, that it must take a checkpoint and terminate safely. This is useful in situations where an analysis component detects fatal conditions in output data, or when the desired goals has been achieved; two conditions that require the simulation component to gracefully end.
- CCS_SIGKILL: Immediately terminate a workflow component without a checkpoint.
- CCS_LAUNCH_NEW: Instruct the command-and-control system to launch a new external task or workflow component. Accompanying metadata includes a reference to the external task in the workflow's registry along with options such as launch arguments.

In addition to these built-in signals, workflow designers can define their own custom signals to react to different dynamic events and trigger corresponding callback functions. This is achieved by providing a map of user-defined signals and their associated callbacks to the ccs_check function. An example of a set of user-defined signals and callbacks is discussed in Section 3.2.

## 2.4 Control Flow

Figure 4 details the sequence of events and the messaging protocol in play when a WMS performs command-and-control. This figure only shows the interactions between the WMS and the simulation component that produces data in the workflow.

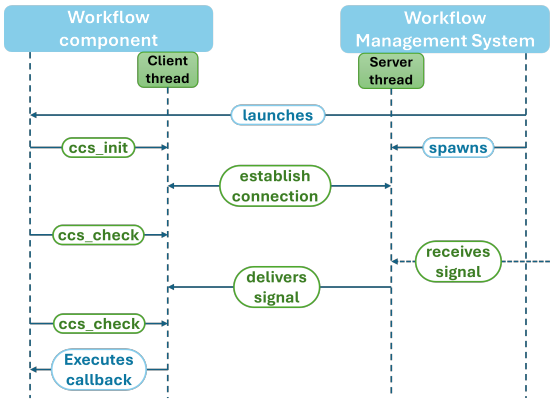


**Figure 4: Messaging protocol between the WMS and a workflow component. A combination of socket communication and message queues between server and client threads are used to signal commands between workflow components.**

First, the WMS launches the simulation workflow component that produces data and an external analysis component (not shown) that consumes data and analyzes it. It then spawns a server thread for each of these two components. These server threads write their connection information to a file on the shared file system. The simulation component can now initialize the command-and-control environment by calling ccs_init which spawns the client thread on its root process. This client thread reads the connection information from the shared file and opens a socket connection to the server thread. The client and server threads establish a communication channel, which sets up the command-and-control environment.

During its execution, the simulation component periodically calls `ccs_check` to check for signals. At some point of the workflow execution, the analysis component sends a signal via the WMS, which delivers it to the simulation component's client thread. When the simulation component calls `ccs_check` next, the client thread executes the callback associated to the signal.

## 2.5 Implementation

A proof-of-concept implementation of the command-and-control system presented in the previous section has been developed as a Python library[1]. This system is composed of modular components designed to coordinate, monitor, and control workflows using a lightweight API. The key components include client and server threads for signal exchange, a decision engine for runtime logic execution, a centralized launcher for application orchestration, and the API for using the system.

The API provided in `api.py` includes an initialization routine that registers the application name and optional callbacks for checkpointing and cleanup, a function to signal the WMS, a function to send a heartbeat to the WMS, and a function to check for signals when a control point is reached. A dictionary of user-defined signals and callbacks can be provided to this function as optional arguments. After checking for in-built signals, the system checks whether it has received any user-defined signals and executes the associated callback function. Signal semantics are defined in `signals.py`, which enumerates symbolic constants representing various control signals. These signals are exchanged between workflow components to coordinate actions such as starting, terminating, or confirming the readiness of applications.

The `client.py` and `server.py` modules implement the communication layer. The client thread is launched on a workflow component, whereas its corresponding server thread resides on the WMS. The server thread listens to or sends signals using a shared message queue, depending on whether it is assigned to a simulation or an analysis component. The messaging protocol is built on top of Python's `multiprocessing.connection` module that uses Unix Pipes for communication. The `decision_engine.py` executes centralized runtime control logic. It monitors signals and invokes policy-based decisions, such as terminating an entire workflow or escalating signals to other components. It ensures that system-wide commands, such as graceful shutdowns, are enforced in a coordinated manner.

All components are orchestrated through the `app_launcher.py`. It launches application processes, spawns their corresponding server and heartbeat threads, and maintains internal state via a registry of running applications. The launcher also communicates with the decision engine via a dedicated queue. The `apps` directory consists of a set of benchmark simulation and analysis codes for the use cases discussed in the following section.

# 3 Command-and-Control Use Cases

To highlight the benefits of command-and-control capabilities and illustrate how they can augment existing in situ simulation-analysis workflows, we consider three representative use cases. The first use case simulates a fatal condition that is detected at runtime by analyzing the produced data. The main simulation is then gracefully terminated, i.e., it checkpoints its current state and cleans up the allocated memory before exiting. In the second use case, a predefined threshold on a given quantity of interest has been reached which indicates that a certain transient physics phenomenon is expected to occur. It thus becomes necessary to export additional scientific data to capture information about this phenomenon for its duration, and then one can resume the initial data production scheme. The third use case highlights how the detection of a pre-defined event in output data can be used to dynamically trigger the launch of a new external analysis component.

In each use case, we use a synthetic benchmark that performs some computations and writes data out in a stream as the data producer. A consumer application then reads this data stream in situ and performs some analysis. When the predefined condition is detected (i.e., the fatal condition or reaching a threshold), it signals the WMS about the observed event, which in turn triggers an action to alter the workflow.

## 3.1 Graceful Termination with Checkpoint

In this scenario, the synthetic code executes a time stepping loop typical of numerical simulations. In each time step, it performs some computations, internal data exchanges, and outputs a stream of data. A loosely coupled external analysis component reads from the stream asynchronously. If it detects a predefined fatal condition in the data, it sends a signal through the command-and-control system to produce a fresh checkpoint and safely halt the simulation.

Figure 5 shows the modification made to the simulation component flow by triggering a new execution path upon the reception of a signal from the command-and-control system. It amounts to adding a call to `ccs_check` in the time step loop, which serves as the control point. Note that it is assumed the developer of the simulation component provides the command-and-control system with callbacks to the existing checkpointing and cleanup functions of the simulation code as arguments.

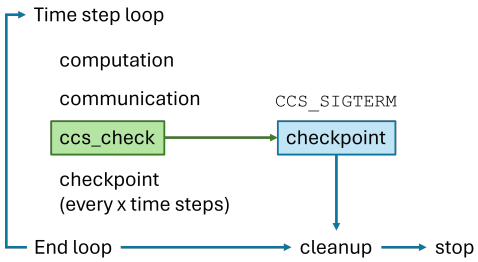


**Figure 5: Change in program flow when a checkpoint is taken dynamically.**

At each time step, the simulation's root process calls `ccs_check` for pending signals in its signal queue. If no signal has been received, it broadcasts a message to all the other processes, indicating them to continue with their normal execution. If the analysis application detects a predefined fatal condition, it sends the `CCS_SIGTERM` signal to its server thread, which in turn delivers it to that of the simulation component. At the subsequent call to `ccs_check`, this

[1] https://github.com/kshitij-v-mehta/effis-cmd-ctrl

signal is broadcast to all simulation processes, which then execute the checkpointing and cleanup functions and terminate.

Enabling such a command-and-control feature in an in situ workflow offers several advantages. First, it requires minimal changes to the science application code to benefit from this feature. Second, it removes the need for manual intervention to analyze output data and automates workflow termination in the presence of fatal conditions, which can lead to significant savings in terms of resource and energy consumption.

To evaluate the overhead and reaction time of the proposed command-and-control system, we conducted an experiment on 100 nodes of a cluster equipped with two 16-core AMD EPYC 7302 processors per node. Ninety-nine compute nodes were dedicated to the simulation that spawned 3,168 MPI ranks (32 per node) while one node ran the analysis with 32 MPI ranks. We ran the time stepping loop 200 times. A synthetic error was inserted into the data stream at time step 100. This error was detected by the analysis code, which then emitted the `CCS_SIGTERM` signal. As the simulation component progresses independently of the analysis component, it only captured the signal when invoking `ccs_check` at time step 102. The measured overhead of the command-and-control system was less than 0.012%.

Further research is needed to study the impact of disparities that can occur when the external analysis component lags behind the data producer in processing output time steps. This depends on several factors, such as the rate at which the simulation produces data, the cost of data analysis, and the resources allocated to the analysis component. In such cases, the command-and-control system could be leveraged to implement back pressure management and load balancing strategies.

## 3.2 Dynamic and Adaptive Reduction of Produced Data

In typical scientific experiments and numerical simulations, the focus on specific quantities or regions of interest evolves over time. Depending on the analysis or visualization that scientists want to perform at a given moment, only a very specific subset of the entire data set produced, sometimes at an accuracy much lower than the best accuracy available, is sufficient. Conversely, occurrences of transient instabilities in a numerical simulation require inspecting high-fidelity data, whose need disappears once the simulation returns to a more stable state. Strong time constraints usually prevent the analysis of the entire data set at full accuracy.

In this use case, we show how the proposed command-and-control allows users to dynamically adapt reduction techniques applied to data streams. This example also shows how applications can define their own set of signals and callback functions in addition to the built-in signals currently provided. In this workflow, the user adds a `START_STREAM` signal to indicate that the simulation must start producing a new data stream using a preset reduction operator and accuracy setting, while the `STOP_STREAM` signal indicates terminating the data stream. The science simulation encapsulates starting and stopping the new stream in a separate function and provides an ON/OFF "knob" in a callback function to start and stop the stream. Figure 6 illustrates the effect of such a command-and-control action on the data production of the simulation code.

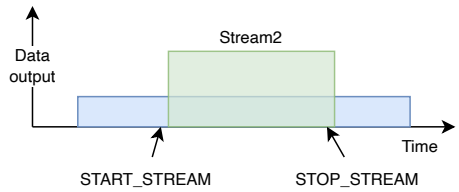


**Figure 6: Timeline of the simulation that shows how writing of a new data stream is enabled when a signal is received. The blue region shows data that is regularly output by the simulation, and the green region represents a stream that is started and terminated dynamically.**

To further tune such a workflow, a `SELECT_REGION` signal can be added to indicate to the simulation to focus on a specific region of interest while a `SET_ACCURACY` signal can allow to modify the parameter of a lossy compression operator applied to the data stream. To this end, the simulation must provide a mapping between these user-defined signals and callback functions to `ccs_check`. When one of these signals is received, the control point looks up the table of mappings and executes the associated callback.

Note that this use case requires the analysis code to process data in near real-time and signal the simulation to adapt the data reduction in a timely way. A delay in sending the signal may cause the simulation to have lost the temporal data required to analyze the necessary physics.

## 3.3 Event-triggered Analysis

This use case addresses scenarios where a new external task or analysis component must be launched dynamically in response to specific events detected in the output data produced by a scientific simulation. The external application may be a diagnostic routine, a visualization tool, or a machine learning (ML) model used for classification or prediction. This dynamic, event-driven triggering mechanism enables the construction of responsive workflows that adapt to changing conditions in a simulation or experimental pipeline.

In this setup, a monitoring task is launched alongside the main science application. This task continuously observes one or more data streams produced by the science simulation. It scans the data for pre-defined patterns or anomalies that represent events of scientific interest. Upon detecting such an event, the monitoring task signals the command-and-control system to spawn the external task or application.

A notable feature of this setup is the decoupling of the science application from the dynamic control logic. All runtime decisions and signaling occur strictly between the monitoring task and the workflow system. The science application remains agnostic to these interactions and requires no modification. This enables working with legacy applications or codes that cannot be modified. This separation of concerns ensures broader applicability and easier integration into existing HPC workflows. This workflow is shown in Figure 7.

When the monitoring task detects an event of interest, it sends the `CCS_LAUNCH_NEW` signal to the command-and-control system

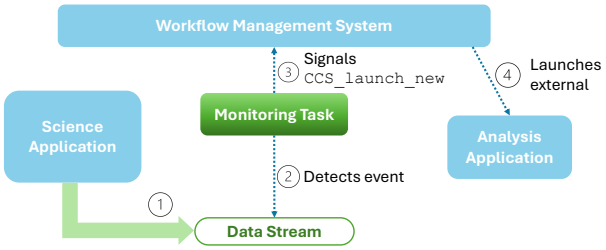


**Figure 7: Figure showing the sequence of interactions between different components of the event-triggered workflow. (i) The simulation outputs a data stream, (ii) a monitoring task reads data to scan for pre-defined events, (iii) it signals the command-and-control system when an event is detected, (iv) the system launches an external application at runtime.**

along with accompanying metadata to launch the external application. This metadata includes the name of the application, a set of launch arguments, and any resource constraints or preferences. For this mechanism to function in a reliable way, all external applications must be pre-registered with the WMS. This registration abstracts system-specific launch details, such as environment setup, MPI configuration, or resource binding, allowing the monitoring task to issue simple, declarative requests encoded in signals without additional complexities or platform-dependent configurations. This design promotes modularity and enables a microservice-like architecture, where independent components can be developed, deployed, and scaled separately while interacting through the command-and-control system.

Looking further, once a triggering condition is met and an external task is launched, the resulting analysis or visualization can feed back into the scientific process. Researchers can apply it to ML workflows in which simulation parameters are adjusted in real time. This event-driven design represents a shift from static workflows towards adaptive, intelligent control systems that can respond to data as it is generated.

## 4 Related Work

This section reviews the main characteristics of WMSs that implement command-and-control-related features or have been specifically designed to manage in situ workflows and position our contribution with regard to these related works.

The Managing Event Oriented Workflows (MEOW) framework [11] follows an original approach in which file-related events (e.g., the creation of a new file by a compute component, accessing a specific file path, or getting an input file with a certain extension) can trigger the execution of new data processing. This event-driven orchestration of a workflow makes MEOW an interesting candidate to implement command-and-control features.

Workways [14] is an interactive science gateway that supports *human-in-loop-workflows*. It allows its users to interact with their workflows to monitor or steer their execution. It relies on a dynamic I/O model that allows data to be inserted into or exported out of a running workflow. This model exposes I/O actors that manage data exchanges between the workflow and the scientific gateway user interface. Human intervention is enabled by blocking these actors that prompt users to provide new input data (e.g., application configuration parameters, or additional information about control flow) before the workflow can resume its execution.

Wilkins [21] is a WMS specifically designed to execute in situ workflows in the context of large scientific campaigns. It enables the static composition of in situ workflows and ensembles of workflows through a flexible YAML interface and relies on the LowFive high-performance data transport layer [16] to provide efficient communications between workflow components. Wilkins also implements a flow control mechanism to handle back pressure in a producer-consumer system and throttles data production rate when consumers are too slow. However, in its current implementation, Wilkins does not support dynamic changes in the requirements of workflow components, which limits its command-and-control capabilities.

The Exascale Framework for High Fidelity Coupled Simulations (EFFIS) [19] is a workflow management system designed to efficiently execute tightly coupled high-performance science applications on multiple leadership class supercomputers. EFFIS enables the composition of in situ workflows in YAML or their programming in Python. Once the workflow has been described, EFFIS delegates the back-end composition process and the execution of the workflow to Cheetah [12], a runtime engine specifically designed to efficiently execute ensembles [13]. EFFIS is also highly synergistic with the ADIOS high-performance I/O framework [10]. Users can instrument their applications with pragmas to couple workflow components with ADIOS. This offers both flexibility and performance for all data movements during workflow execution and offers ways to implement data-driven command-and-control operations.

The work presented in this paper does not extend an existing WMS with new command-and-control features but instead proposes to enable these features through an external WMS-agnostic command-and-control system. As detailed in the previous sections, this system has been specifically designed for in situ simulation-analysis workflows. However, we illustrated how users can easily extend it with additional signals. Thus, we believe that other categories of scientific workflows, such as the AI-coupled HPC motifs described in [6] could leverage the proposed command-and-control system.

## 5 Conclusion and Future Work

This paper presents a framework for enabling command and control in advanced in situ workflows. We discuss the design requirements of such a system and describe its architecture and API for dynamic coordination between components of a science workflow. A set of built-in and user-defined signals facilitates dynamic feedback and adaptive behavior through a centralized workflow system that handles resource management and decision making. The proposed system provides the ability to define events and trigger associated actions upon detection of such events in science data.

We demonstrate the utility of this system across three representative use cases: graceful termination with checkpointing, dynamic data reduction, and event-triggered analysis. We use a benchmark code to show how a science application can alter its program flow or its data production by dynamically responding to events that are

detected in data at runtime, or launch external applications when pre-defined events are detected in data.

Our future work includes extending the set of signals and callbacks supported by the system and evaluating its usefulness and benefits in real-world science workflows. We plan to adopt more sophisticated messaging middleware such as ZeroMQ to improve scalability and performance over the current socket-based implementation. Finally, we will expand the system with a library of pre-built analysis and data monitoring tasks that can be deployed by a WMS for enabling data analysis and event detection.

## Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

## References

[1] Peter Amstutz, Maxim Mikheev, Michael R. Crusoe, Nebojša Tijanić, Samuel Lampa, et al. 2024. Existing Workflow systems. [Online] https://s.apache.org/existing-workflow-systems. updated 2024-08-18, accessed 2024-08-18.

[2] Michael Athans. 1987. Command and Control (C2) Theory: A Challenge to Control Science. *IEEE Trans. Automat. Control* 32 (April 1987), 286 – 293. Issue 4. doi:10.1109/TAC.1987.1104607

[3] Anakha Babu, Tekin Bicer, Saugat Kandel, Tao Zhou, Daniel Ching, Steven Henke, Siniša Veseli, Ryan Chard, Antonio Miceli, and Mathew Cherukara. 2023. AI-assisted Automated Workflow for Real-time X-ray Ptychography Data Analysis via Federated Resources. *Electronic Imaging* 35, 11, Article 232 (2023), 6 pages. doi:10.2352/EI.2023.35.11.HPCI-232

[4] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel Katz, Ben Clifford, Rohan Kumar, Luksaz Lacinski, Ryan Chard, Justin Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. 2019. Parsl: Pervasive Parallel Programming in Python. In *Proceedings of the 28th ACM International Symposium on High-Performance Parallel and Distributed Computing*. 25–36.

[5] Andrew C. Bauer, Hasan Abbasi, James Ahrens, Hank Childs, Berk Geveci, Scott Klasky, Kenneth Moreland, Patrick O'Leary, Venkatram Vishwanath, Brad Whitlock, and E. Wes Bethel. 2016. In Situ Methods, Infrastructures, and Applications on High Performance Computing Platforms, a State-of-the-art (STAR) Report. *Computer Graphics Forum* 35, 3 (June 2016), 577–597. doi:10.1111/cgf.12930

[6] Wes Brewer, Ana Gainaru, Frédéric Suter, Feiyi Wang, Murali Emani, and Shantenu Jha. 2025. AI-coupled HPC Workflow Applications, Middleware and Performance. arXiv:2406.14315 [cs.DC] doi:10.48550/arXiv.2406.14315

[7] Hank Childs, Sean Ahern, James Ahrens, Andrew Bauer, Janine Bennett, E. Wes Bethel, Peer-Timo Bremer, Eric Brugger, Joseph Cottam, Matthieu Dorier, Soumya Dutta, Jean Favre, Thomas Fogal, Steffen Frey, Christoph Garth, Berk Geveci, William Godoy, Charles Hansen, Cyrus Harrison, Bernd Hentschel, Joseph Insley, Chris Johnson, Scott Klasky, Aaron Knoll, James Kress, Matthew Larsen, Jay Lofstead, Kwan-Liu Ma, Preeti Malakar, Jeremy Meredith, Kenneth Moreland, Paul Navrátil, Patrick O'Leary, Manish Parashar, Valerio Pascucci, John Patchett, Tom Peterka, Steve Petruzza, Norbert Podhorszki, David Pugmire, Michel Rasquin, Silvio Rizzi, David Rogers, Sudhanshu Sane, Franz Sauer, Robert Sisneros, Han-Wei Shen, Will Usher, Rhonda Vickery, Venkatram Vishwanath, Ingo Wald, Ruonan Wang, Gunther Weber, Brad Whitlock, Matthew Wolf, Hongfeng Yu, and Sean Ziegeler. 2020. A Terminology for in situ Visualization and Analysis Systems. *International Journal of High Performance Computing and Applications* 34, 6 (2020), 676–691. doi:10.1177/1094342020935991

[8] Francesco Di Natale, Harsh Bhatia, Timothy S. Carpenter, Chris Neale, Sara Kokkila-Schumacher, Tomas Oppelstrup, Liam Stanton, Xiaohua Zhang, Shiv Sundram, Thomas R. W. Scogland, Gautham Dharuman, Michael P. Surh, Yue Yang, Claudia Misale, Lars Schneidenbach, Carlos Costa, Changhoan Kim, Bruce D'Amora, Sandrasegaram Gnanakaran, Dwight V. Nissley, Fred Streitz, Felice C. Lightstone, Peer-Timo Bremer, James N. Glosli, and Helgi I. Ingólfsson. 2019. A Massively Parallel Infrastructure for Adaptive Multiscale Simulations: Modeling RAS Initiation Pathway for Cancer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, Denver, CO, Article 57, 16 pages. doi:10.1145/3295500.335617

[9] Rafael Ferreira da Silva, Robert G. Moore, Ben Mintz, Rigoberto Advincula, Anees Al-Najjar, Luke Baldwin, Craig Bridges, Ryan Coffee, Ewa Deelman, Christian Engelmann, Brian D. Etz, Millie Firestone, Ian T. Foster, Panchapakesan Ganesh, Leslie Hamilton, Dale Huber, Ilia Ivanov, Shantenu Jha, Ying Li, Yongtao Liu, Jay Lofstead, Anirban Mandal, Hector Garcia Martin, Theresa Mayer, Marshall McDonnell, Vijayakumar Murugesan, Sal Nimer, Nageswara Rao, Martin Seifrid, Mitra Taheri, Michela Taufer, and Konstantinos D. Vogiatzis. 2024. *Shaping the Future of Self-Driving Autonomous Laboratories Workshop*. Technical Report ORNL/TM-2024/3714. Oak Ridge National Laboratory. doi:10.5281/zenodo.14430233

[10] William F. Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, Axel Huebl, Mark Kim, James Kress, Tahsin Kurc, Qing Liu, Jeremy Logan, Kshitij Mehta, George Ostrouchov, Manish Parashar, Franz Poeschel, David Pugmire, Eric Suchyta, Keichi Takahashi, Nick Thompson, Seiji Tsutsumi, Lipeng Wan, Matthew Wolf, Kesheng Wu, and Scott Klasky. 2020. ADIOS 2: The Adaptable Input Output System. A Framework for High-Performance Data Management. *SoftwareX* 12 (2020), 100561. doi:10.1016/j.softx.2020.100561

[11] David Marchant, Mark Blomqvist, Philip Jensen, Iben Lilholm, and Martin Nørgaard. 2023. Delivering Rules-Based Workflows for Science. In *Proceedings of the 18th Workshop on Workflows in Support of Large-Scale Science*. ACM, Denver, CO, 2000–2008.

[12] Kshitij Mehta, Bryce Allen, Matthew Wolf, Jeremy Logan, Eric Suchyta, Swati Singhal, Jong Y. Choi, Keichi Takahashi, Kevin Huck, Igor Yakushin, Alan Sussman, Todd Munson, Ian Foster, and Scott Klasky. 2022. A Codesign Framework for Online Data Analysis and Reduction. *Concurrency and Computation: Practice and Experience* 34, 14 (2022), e6519. doi:10.1002/cpe.6519

[13] Kshitij Mehta, Ashley Cliff, Frédéric Suter, Angelica M. Walker, Matthew Wolf, Daniel Jacobson, and Scott Klasky. 2022. Running Ensemble Workflows at Extreme Scale: Lessons Learned and Path Forward. In *Proceedings of the IEEE 18th International Conference on e-Science*. IEEE, Salt Lake City, UT, 284–294. doi:10.1109/eScience55777.2022.00042

[14] Hoang Anh Nguyen, David Abramson, Timoleon Kipouros, Andrew Janke, and Graham Galloway. 2015. WorkWays: Interacting with Scientific Workflows. *Concurrency and Computation: Practice and Experience* 27, 16 (2015), 4377–4397.

[15] Hilary Oliver, Matthew Shin, David Matthews, Oliver Sanders, Sadie Bartholomew, Andrew Clark, Ben Fitzpatrick, Ronald van Haren, Rolf Hut, and Niels Drost. 2019. Workflow Automation for Cycling Systems. *Computing in Science & Engineering* 21, 4 (2019), 7–21. doi:10.1109/MCSE.2019.2906593

[16] Tom Peterka, Dmitriy Morozov, Arnur Nigmetov, Orcun Yildiz, Bogdan Nicolae, and Philip Davis. 2023. LowFive: In Situ Data Transport for High-Performance Workflows. In *Proceedings of the 37th IEEE International Parallel and Distributed Processing Symposium*. IEEE, St. Petersburg, FL, 985–995. doi:10.1109/IPDPS54959.2023.00102

[17] Giovanni Pizzi, Andrea Cepellotti, Riccardo Sabatini, Nicola Marzari, and Boris Kozinsky. 2016. AiiDA: A–utomated Interactive Infrastructure and Database for Computational Science. *Computational Materials Science* 111 (2016), 218–230. doi:10.1016/j.commatsci.2015.09.013

[18] Eric Suchyta, Jong Youl Choi, Seung-Hoe Ku, David Pugmire, Ana Gainaru, Kevin Huck, Ralph Kube, Aaron Scheinberg, Frédéric Suter, Choongseock Chang, Todd Munson, Norbert Podhorszki, and Scott Klasky. 2022. Hybrid Analysis of Fusion Data for Online Understanding of Complex Science on Extreme Scale Computers. In *Proceedings of the 24th IEEE Cluster Conference*. IEEE, Heidelberg, Germany, 218–229. doi:10.1109/CLUSTER51413.2022.00035

[19] Eric Suchyta, Scott Klasky, Norbert Podhorszki, Matthew Wolf, Abolaji Adesoji, Choong-Seock Chang, Jong Choi, Philip Davis, Julien Dominski, Stéphane Ethier, Ian Foster, Kai Germaschewski, Berk Geveci, Chris Harris, Kevin Huck, Qing Liu, Jeremy Logan, Kshitij Mehta, Gabriele Merlo, Shirley Moore, Todd Munson, Manish Parashar, David Pugmire, Mark Shephard, Cameron Smith, Pradeep Subedi, Lipeng Wan, Ruonan Wang, and Shuangxi Zhang. 2022. The Exascale Framework for High Fidelity coupled Simulations (EFFIS): Enabling Whole Device Modeling in Fusion Science. *International Journal of High Performance Computing and Applications* 36, 1 (2022), 106–128.

[20] Frédéric Suter, Tainã Coleman, Ílkay Altintas, Rosa M. Badia, Bartosz Balis, Kyle Chard, Iacopo Colonnelli, Ewa Deelman, Paolo Di Tommaso, Thomas Fahringer, Carole Goble, Shantenu Jha, Daniel S. Katz, Johannes Köster, Ulf Leser, Kshitij Mehta, Hilary Oliver, J.-Luc Peterson, Giovanni Pizzi, Loïc Pottier, Raül Sirvent, Eric Suchyta, Douglas Thain, Sean R. Wilkinson, Justin M. Wozniak, and Rafael Ferreira da Silva. 2025. A Terminology for Scientific Workflow Systems. *Future Generation Computer Systems* 174 (2025), 107974. doi:10.1016/j.future.2025.107974

[21] Orcun Yildiz, Dmitriy Morozov, Arnur Nigmetov, Bogdan Nicolae, and Tom Peterka. 2024. Wilkins: HPC In Situ Workflows Made Easy. *Frontiers in High Performance Computing* 2 (2024), 14 pages. doi:10.3389/fhpcp.2024.1472719