Shepherd: Seamless Integration of Service Workflows into Task-Based Workflows through Log Monitoring

Md Saiful Islam
University of Notre Dame
Notre Dame, IN, USA
mislam5@nd.edu

Douglas Thain
University of Notre Dame
Notre Dame, IN, USA
dthain@nd.edu

***Abstract*—Traditional workflow managers focus on coordinating discrete tasks: actions that run to completion. However, emerging workflows require persistent services that must be managed alongside traditional tasks. We introduce Shepherd, a local workflow manager that runs services as a task, enabling them to be seamlessly integrated into larger distributed workflows. By inferring service states through log outputs and file creations, Shepherd enables the coordinated startup and shutdown of dependent services without modifying their original code. We demonstrate Shepherd's effectiveness in large-scale drone simulations, where it enhances workflow flexibility, reliability, and comprehensive logging and visualization.**

***Index Terms*—Workflow Management, Service Orchestration**

I. Introduction

Workflow management is the coordinated execution of a series of tasks, often in a predefined order [1]–[4]. This concept is crucial for scientific research and many other fields where complex processes need to be managed efficiently [2], [5]. Traditional workflow managers are often data-centric, focusing on the movement and transformation of data between tasks, with task dependencies based on completion, where a dependent task can start only after the preceding task has returned an exit status [3], [4], [6], [7]. They excel at handling clear task dependencies, ensuring data consistency, and automating data processing pipelines [4], [7], [8]. However, this model is not well-suited for applications that require persistent services as part of a traditional task-based workflow.

For instance, consider a scenario where a web server should start only after the database service has completed its initialization and is ready to handle queries. The database does not perform a single action that completes; instead, it reaches an internal state that should dynamically trigger the launch of the web server. This highlights the need for workflow management that can accommodate such state-based dependencies.

Managing workflows with persistent services presents the additional challenge of ensuring a graceful shutdown. Unlike actions that terminate upon completion, services must be carefully terminated to prevent data loss and ensure system stability. This involves stopping the service itself, any subprocesses it may have started, and cleaning up temporary files. Failing to do so can cause resource leaks and instability, hindering reliability and efficiency. Coordinated startup and graceful shutdown allow services to integrate seamlessly into workflows, just like any other self-completing tasks.

To overcome these challenges, a workflow manager must track the internal states of services. This allows it to know when they have completed initialization or reached states that other processes depend on. Tracking these states without making code changes presents a significant challenge, especially if the services lack a dedicated API for this purpose. However, many modern applications extensively log their actions and execution status, providing an opportunity to infer the dynamic states of the application through log analysis. Similarly, monitoring files produced by services can help infer their states, which can then trigger the start or termination of other actions or services in the workflow.

Shepherd is a workflow manager that puts these concepts into practice. By monitoring standard output and specified files, Shepherd infers application states. This enables Shepherd to effectively manage workflows that include persistent services. For example, in the database and web server scenario, Shepherd monitors the database service's logs for a message indicating successful startup. Upon detecting this state, Shepherd triggers the initiation of the web server, ensuring efficient workflow execution. This state-based dependency management and state inference empowers Shepherd to handle complex service workflows without requiring code modifications. Moreover, Shepherd wraps the entire service workflow into a single task that terminates upon completion, making it easy to integrate into larger task-based workflows.

In this paper, we present the architecture of Shepherd, focusing on its novel method for integrating service workflows into task-based environments. We demonstrate Shepherd's capabilities through large-scale drone simulations, where it effectively manages dependencies and coordinates the startup and shutdown of services. Additionally, we show how Shepherd simplifies integration testing, a process often challenging with traditional bash scripts. By integrating services into traditional task-based workflows, Shepherd enhances workflow efficiency, reliability, and ease of management.

II. Service Workflows

Service workflows involve coordinating services that persist throughout the workflow's lifecycle and require dynamic state management. These workflows present unique challenges and opportunities, particularly in environments where service dependencies and state transitions are crucial.

*a) **Defining Service Workflows**:* A service workflow is a type of workflow that integrates both actions that terminate upon completion and persistent services into a unified execution plan. Services are processes that continue running until explicitly stopped, maintaining an operational state throughout the workflow. Examples include web servers, databases, and simulation engines.

*b) **Challenges of Service Workflows**:* Service workflows present several unique challenges compared to traditional task-based workflows:

- **Complex Dependency Management**: Services often depend on specific internal states of other services before they can start. For example, in a drone simulation, a physics engine must be ready before the drone autopilot can spawn a new model. This requires coordination based on internal states rather than task completion, necessitating more sophisticated management.
- **Graceful Shutdown:** Unlike actions that terminate upon completion, services require controlled shutdown processes to prevent data loss and ensure system stability. This involves stopping the service, terminating subprocesses, and cleaning up resources.
- **Indirect State Tracking:** Many services lack explicit interfaces for external monitoring of their internal states. This makes it difficult for workflow managers to determine when a service is ready, has successfully completed a task, or is experiencing errors.

*c) **Shepherd's Approach**:* Shepherd addresses these challenges by treating services as first-class citizens within the workflow. It introduces the concept of ***services as a task***, where one instance of Shepherd itself constitutes a task. A service workflow is thereby encapsulated to function as a single task that can be seamlessly integrated into a task-based workflow. Shepherd services require specific attributes that differentiate them from regular services:

- **Start Conditions:** These conditions specify the states of other services or actions that must be satisfied before the service task can start.
- **Stop Conditions:** These conditions trigger the graceful shutdown of the service task when met.
- **Well-Defined Internal States:** Services must define states that can be tracked through log monitoring or file observations without modifying the original code.

Figure 1 illustrates a simplified service workflow for a drone simulation. It demonstrates how various services and actions interact, highlighting dependencies and state transitions.

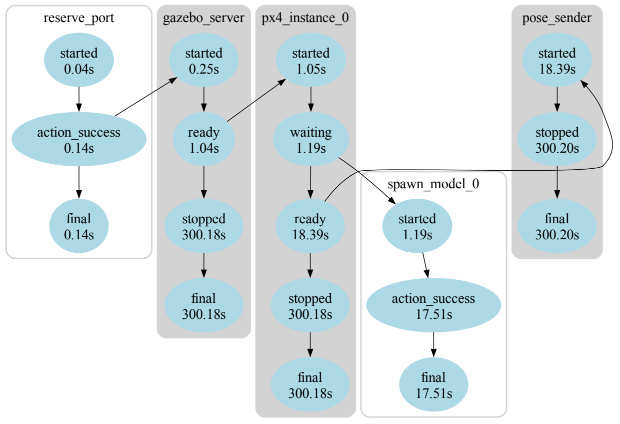


Fig. 1: Example of Service Workflow with State Transition
*In this workflow, **reserve_port** and **spawn_model_0** are actions that terminate upon completion, while **gazebo_server** and **px4_instance_0** are persistent services. Before starting a PX4 instance, the Gazebo server must be in the **ready** state; otherwise, PX4 will fail to connect. Shepherd tracks Gazebo's internal state through log messages and launches PX4 when it's ready. PX4 also transitions through multiple internal states, with other actions and services depending on some of them.*

III. Architecture of Shepherd

A. Overview

Shepherd is designed to manage and orchestrate local workflows that involve both types of tasks: actions that exit upon completion and services that run until they are stopped. At its core, Shepherd takes a YAML-based workflow description as input, which defines the tasks, their dependencies, and the conditions under which they should be executed.

Once provided with a workflow description, Shepherd evaluates the dependencies of each task. When the dependencies for a task are resolved, Shepherd invokes the task, whether it is an action or a service. The execution of these tasks is monitored in real-time through the logs they produce.

Shepherd tracks application states through user-defined messages or keywords found in logs, as well as default states introduced by Shepherd (discussed in Section III-D). These states can be used to define dependencies for other tasks in the workflow.

Shepherd also monitors for stop signals such as user intervention, success criteria, maximum run time, or keyboard interrupts. In these cases, it stops all actions and services, cleans up temporary files, and writes the workflow outcome. These outcomes can be used to visualize the Directed Acyclic Graph (DAG) and the timeline of execution.

B. Components

Shepherd has three primary components that work together to ensure the reliable execution of the described workflows:

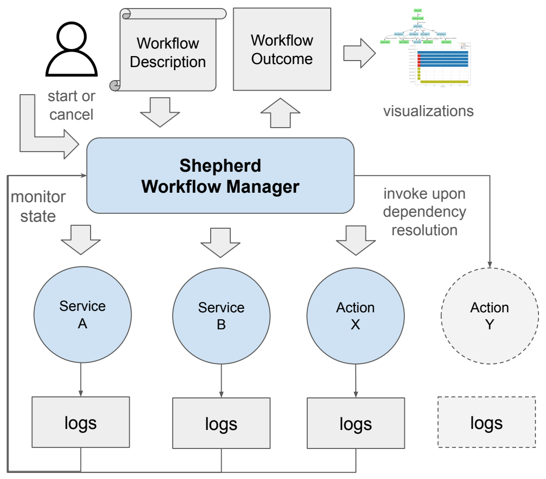


Fig. 2: Shepherd Architecture

The Shepherd Workflow Manager invokes actions and services when their dependencies are resolved. It then monitors their internal state through the logs they produce. At the end, Shepherd generates outcomes such as state transition times, stdout, and other logs, which can be used to generate visualizations and analysis.

the Service Manager, Program Executor, and Activity Monitor. Figure 3 illustrates these components and their interactions.

- **Service Manager:** Responsible for preprocessing and validating the YAML configuration, creating the execution DAG, initiating tasks according to the DAG order, monitoring stop conditions for graceful shutdown, and cleaning up temporary files.
- **Program Executor:** Starts services based on their defined commands, tracks their default states, and checks dependencies before execution to ensure all required conditions are met.
- **Activity Monitor:** Monitors each service's standard output and specified files for state changes, updating internal states based on user-defined criteria for accurate, real-time state management.

C. Actions vs. Services

Shepherd categorizes tasks as either actions or services:

- **Actions:** Tasks that execute specific commands and terminate upon completion, returning a result without external intervention. They are ideal for operations with a clear start and end, such as data processing jobs.
- **Services:** Tasks that continue running until explicitly stopped, performing ongoing functions like handling user requests or maintaining a database connection. Shepherd ensures services start only when their dependencies are met and are gracefully shut down when required.

By distinguishing between actions that terminate upon completion and services that require explicit termination, Shepherd effectively manages the unique requirements and lifecycles of each task type.

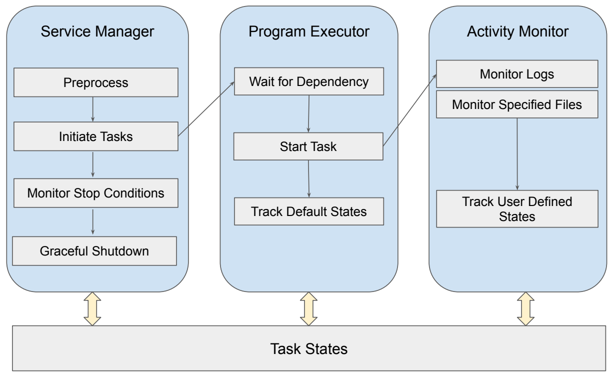


Fig. 3: Shepherd Internal Components

The figure illustrates Shepherd's three components: Service Manager, Program Executor, and Activity Monitor. The Service Manager handles configuration processing, execution DAG creation, service initiation, and graceful shutdown with cleanup. The Program Executor starts services, tracks their states, and ensures dependencies are met. The Activity Monitor monitors outputs and files for state changes, updating internal states in real time.

D. Task State Machine

Shepherd manages task execution through a set of well-defined states, ensuring dependencies are resolved before execution. Each task has default states (***initialized, started, and final***) and may include user-defined states reflecting specific workflow needs.

Tasks transition from ***initialized*** to ***started*** once all dependencies are resolved. After reaching ***started***, tasks can move through user-defined states based on execution progress. Actions return an ***action_success*** state when they complete with a zero return code, or an ***action_failure*** state if they terminate with a non-zero return code. Services may transition to a ***service_failure*** state if they stop unexpectedly.

Additionally, any task that receives a stop signal is marked as ***stopped***. Ultimately, all tasks, regardless of their type or execution path, transition to a ***final*** state, reflecting the completion and outcome of their execution.

Figure 4 illustrates the Shepherd state machine. This state machine ensures a structured and predictable workflow by accurately tracking the status and progression of each task within Shepherd.

E. Configuration Language

The power and adaptability of Shepherd lie in its YAML-based configuration file, which acts as the blueprint for any workflow. This file allows users to precisely define the services and actions involved, their dependencies on each other, the criteria used to monitor their states, and other crucial operational parameters.

Consider running a web application test that depends on a database service. The workflow involves the following steps:

- **Initialize Database:** Start the database service and wait for it to be ready.

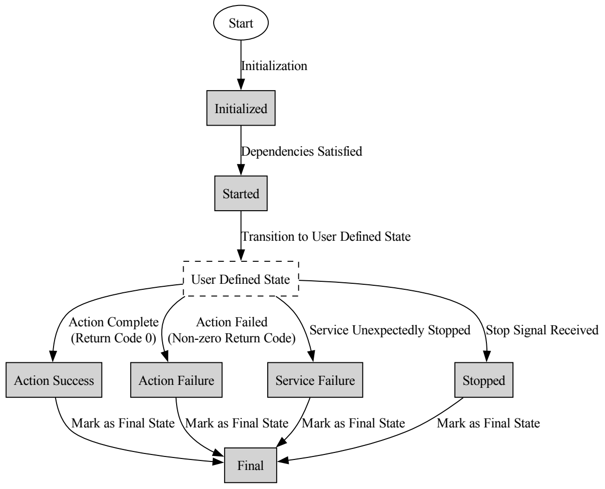


Fig. 4: Task State Machine

Tasks start in ***Initialized****, move to* ***Started*** *once dependencies are resolved, and transition through user-defined states based on execution progress. Actions end in* ***Action Success*** *if they complete with a zero return code or* ***Action Failure*** *with a non-zero return code. Services can transition to* ***Service Failure*** *if they stop unexpectedly. Any task that receives a stop signal is marked as* ***Stopped****. Ultimately, all tasks reach the* ***Final*** *state.*

- **Start Web Server:** Start the web server only after the database is ready.
- **Run Tests:** Execute a series of automated tests to verify the application's functionality.
- **Graceful Shutdown:** Stop everything and clean up after a successful test run.

Figure 5 shows the YAML configuration for this workflow. The database service logs ”Database is ready to accept requests” when prepared with the required data. This message is defined as the ***ready*** state. Shepherd monitors logs for this message to mark the database as ***ready***. The web server depends on this ***ready*** state and defines an ***up*** state when it starts successfully. Once the web server is ***up***, Shepherd runs the tests. The workflow initiates shutdown after meeting the success criteria.

A task can also depend on the existence of a file, allowing it to wait until a specific file is created. By specifying a minimum file size, users can ensure the file is fully generated as expected. Figure 6 shows an example.

Shepherd allows dependencies to be defined in two modes: the default ***all*** mode, where all specified conditions must be met before a task can start, and ***any*** mode, where a task begins as soon as any one condition is satisfied. For example, the configuration in Figure 7 triggers data analysis when either sensor data or historical data is ready.

Besides these options, Shepherd also allows users to choose the location of standard output and error files. Shepherd provides three different shutdown mechanisms. Users can specify a maximum run time, or a stop file. In the case of a stop file, if a user creates a file with the name specified in the configuration, Shepherd will treat it as a stop signal. Shepherd will also enter shutdown mode when the success criteria are met or if any tasks fail (if shutdown on failure is specified). A full list of configuration options is provided in the user guide.

```yaml
tasks:
  database:
    type: "service"
    command: "start_database.sh"
    state:
      log:
        ready: "Database is ready to ..."
  webserver:
    type: "service"
    command: "start_webserver.py --port 8080"
    dependency:
      items:
        database: "ready"
    state:
      log:
        up: "Webserver started at 8080"
  run_tests:
    type: "action"
    command: "pytest tests/"
    dependency:
      items:
        webserver: "up"
success_criteria:
    items:
        run_tests: "action_success"
output:
  state_times: "state_transition_times.json"
```

Fig. 5: Shepherd Configuration Example

```yaml
tasks:
  program-0:
  ...
  program-1:
    file_dependency:
      mode: "all"
      items:
        - path: "file-from-task-0.log"
          min_size: 1
```

Fig. 6: Configuration for File Dependency

```yaml
tasks:
    data_analysis:
        command: "python analyze_data.py"
        dependency:
            mode: "any"
            items:
                sensor_data: "ready"
                historical_data: "ready"
```

Fig. 7: Configuration for ”any” Dependency Mode

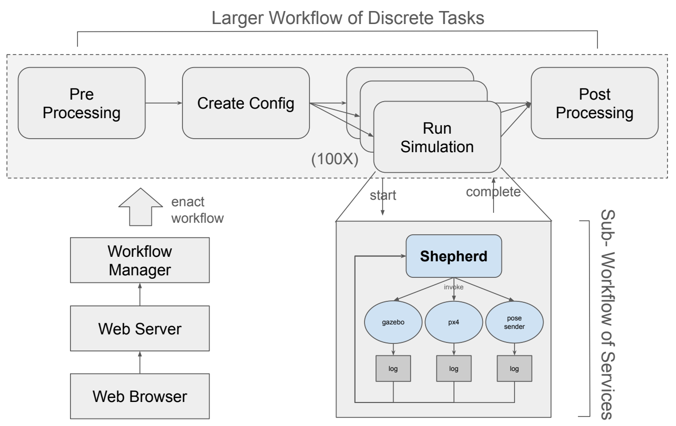


Fig. 8: Hierarchical Workflow for Drone Simulation
Shepherd converts a drone simulation, which comprises multiple persistent services, into a discrete task with defined internal dependencies, start, and end points. This task can then be seamlessly integrated into a task-based workflow manager such as Makeflow.

IV. Application in Large-Scale Drone Simulation: The SADE Project

A. *From Workflow of Services to Workflows of Discrete Tasks*

The SADE project aims to develop a UTM (Unmanned Traffic Management) system for safe, privacy-respecting sUAS flights, emphasizing automated, fair decision-making with speed, transparency, and scalability for large drone fleets [9]. It includes a simulation component that allows users to simulate SADE rules with a large number of drone fleets. In this platform, Shepherd functions as a node-level workflow manager, coordinating the launch of interconnected components required for these simulations.

The simulation is set up through a web interface that sends the simulation description to a distributed workflow manager. The manager initiates a workflow consisting of preprocessing, configuration creation, run simulation, and post-processing. The run simulation task executes Shepherd on HPC cluster nodes using the configuration from the previous step. Shepherd starts a local workflow on each node, encapsulating all service runs into a single task. This task is then integrated into the larger distributed workflow. Figure 8 illustrates this workflow.

Besides converting services into tasks, Shepherd manages dependencies among them. To simulate drone operations, tasks must be executed in order. First, network ports are reserved to ensure each process has a dedicated communication port. Next, the Gazebo server is started to create the simulation environment. It must be ready and accepting connections before any drone instances can operate. Once Gazebo is ready, PX4 instances, which control drone behavior, are started. Each instance waits for a connection to Gazebo before drone models are spawned. Finally, the Pose Sender begins transmitting the drones' position and rotation to a central server.

The simulation stops when it exceeds the maximum runtime or receives a stop signal. Shepherd then gracefully shuts down all services and actions, cleans up temporary files, and writes the simulation outcome to the shared filesystem, making it available for the next step in the distributed workflow.

B. *Configurations for Drone Simulation*

This section outlines key components of Shepherd's configuration used in the drone simulation. The Gazebo server starts only after permissions on ***ports.config*** have been changed. Shepherd tracks this with its default state, ***action_success***, when an action is successful. Figure 9 shows the YAML configuration for the Gazebo server. We also define a custom state, ***ready***, marked when the server logs "Connected to gazebo master."

```yaml
gazebo_server:
  type: "service"
  command: "./start_gazebo_server.sh"
  state:
    log:
      ready: "Connected to gazebo master"
  dependency:
    items:
      chmod_port_config: "action_success"
```

Fig. 9: Configuration for Gazebo

```yaml
px4_instance_0:
  type: "service"
  command: "./start_px4_instance.sh 0"
  state:
    log:
      waiting_for_simulator: "Waiting ..."
      ready: "Startup script returned ..."
  dependency:
    items:
      gazebo_server: "ready"
```

Fig. 10: Configuration for PX4

```yaml
spawn_model_0:
  type: "action"
  command: "./spawn_model.sh 0"
  dependency:
    items:
      px4_instance_0: "waiting_for_simulator"
```

Fig. 11: Configuration for Model Spawn

Next, each PX4 instance starts only after the Gazebo server is ready, ensuring a proper connection. This is tracked by the custom ready state in the Gazebo server. Figure 10 shows the YAML configuration for a PX4 instance. PX4 custom states include ***waiting_for_simulator*** and ***ready***, triggered by logs "Waiting for simulator to accept connection" and "Startup script returned successfully."

```
pose_sender:
  type: "service"
  command: "./start_pose_sender.sh"
  dependency:
    items:
      px4_instance_0: "ready"
      px4_instance_1: "ready"
      px4_instance_2: "ready"
      px4_instance_3: "ready"
```

Fig. 12: Configuration for Pose Sender

Each PX4 instance requires a corresponding model to be spawned in the Gazebo environment. This spawning action occurs only after the PX4 instance reaches the ***waiting_for_simulator*** state. Figure 11 illustrates the YAML configuration for spawning a model for a PX4 instance.

Finally, the Pose Sender service, which sends drone position data to a central server, starts only after all PX4 instances are in the ready state. Figure 12 shows the YAML configuration for Pose Sender.

C. Logs and Visualization

Shepherd not only manages task execution but also generates logs for detailed analysis and visualization. Each task produces standard output, error logs, and state transition times, which can be used to create insightful visualizations. Shepherd Viz uses these outputs to produce three key visualizations:

- **Execution DAG (Figure 13a):** Illustrates the execution order of tasks based on dependencies.
- **Timeline (Figure 13b):** Shows task execution duration, highlighting start times, states, and completion times for performance analysis.
- **State Transition Diagram (Figure 13c):** Combines workflow structure with state transition times, detailing when state changes occurred.

These visualizations provide a comprehensive view of the workflow, aiding in performance analysis and debugging. Figure 14 shows Shepherd integrated into a larger workflow of 100 drones running in 25 nodes, where each Shepherd instance manages 4 simulated drones.

These figures are generated by the Shepherd tool and have been resized for space constraints. Full-resolution versions are available on the project's GitHub repository.

D. How Shepherd Helps

Shepherd significantly enhances the efficiency and reliability of large-scale drone simulations by offering several key advantages:

- **Service as Task:** Shepherd enables workflows that include services to be run as part of a larger distributed workflow by encapsulating these services into a single task with a defined start and end. Without Shepherd, it would be challenging to manage services within traditional task-based workflows.
- **Dependency Management:** Shepherd ensures services and actions start only when prerequisites are met, enabling proper synchronization, which is critical for drone simulations.
- **Automated State Tracking:** By monitoring logs, Shepherd dynamically tracks states, allowing real-time workflow management without manual intervention.
- **Graceful Shutdown and Cleanup:** Shepherd provides controlled shutdown mechanisms for persistent services. This ensures services are terminated in an orderly manner, resources are released, and temporary files are cleaned up, preventing data loss and maintaining system stability.
- **Robust Logging and Visualization:** Shepherd generates comprehensive logs and state transition data, which can be used with tools like Shepherd Viz to create visualizations that optimize simulations.
- **Simplified Configuration Management:** Shepherd's YAML-based configuration simplifies the setup and maintenance of complex workflows, making it easy to visualize, modify, and debug intricate scenarios.
- **Enhanced Reliability:** With robust error handling and dependency checks, Shepherd improves simulation reliability, reducing errors and ensuring predictable execution.

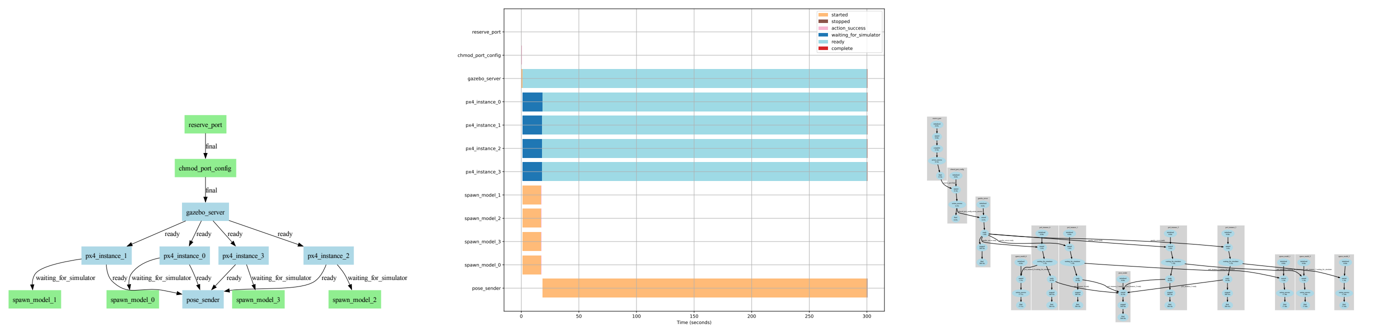
V. Application in Integration Test

Integration testing is a crucial phase in software development, ensuring that different components of an application work seamlessly together. Traditional test frameworks or bash scripts can become cumbersome and error-prone, especially when dealing with numerous configurations, datasets, and setup variations. Shepherd addresses these challenges by providing a structured and automated approach to integration testing.

Consider the YAML configuration example shown earlier in figure 5. In a scenario where we need to run the web application test 100 times with 100 different setups and datasets, manually managing these variations can be impractical. Shepherd simplifies this process by allowing users to define each test setup as a service or action within a unified workflow. The configurations, dependencies, and state monitoring are clearly defined, ensuring consistent and reliable execution across all test runs.

By using Shepherd, each iteration of the integration test can be automatically orchestrated, from initializing the database and starting the web server to running the tests and performing graceful shutdowns. This automation not only saves time but also reduces the risk of human error, ensuring that all tests are executed under the specified conditions. Additionally, Shepherd's robust logging and visualization tools provide detailed insights into each test run, aiding in debugging and analysis.

In summary, Shepherd offers a clear and concise way to automate large-scale integration tests, providing a reliable framework for managing complex test scenarios and enhancing the overall efficiency and effectiveness of the integration testing process.



(a) Execution DAG

(b) Timeline

(c) State Transition Diagram

Fig. 13: Visualizations Produced by Shepherd for Drone Simulations
Three key visualizations generated by Shepherd (a) The Shepherd Workflow Diagram, illustrating the execution order of tasks based on dependencies; (b) The Shepherd Timeline Diagram, showing task execution duration and states for performance analysis; (c) The Shepherd State Transition Diagram, detailing state changes and the logical flow of the workflow. Together, these visualizations provide a comprehensive view of the workflow. ***Note:*** *Full-resolution figures are available at the project's GitHub repository.*

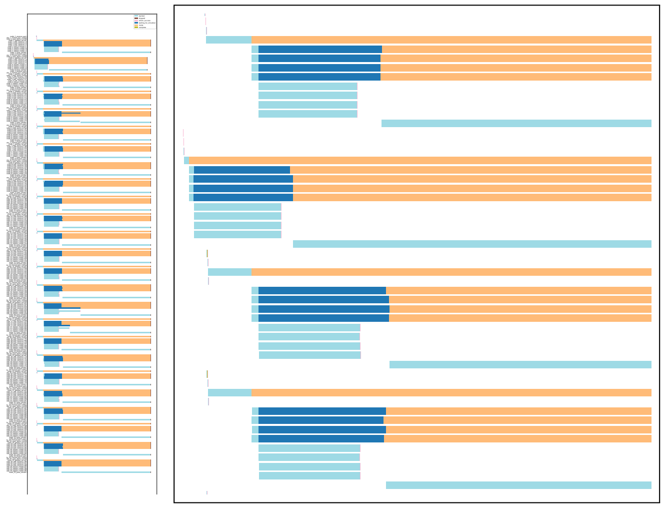


Fig. 14: Timeline of 100 Drone Run
Left: Full timeline of complete workflow with 25 Shepherds running 4 simulated drones each. Right: Detail view of 4 Shepherd nodes.

VI. Related Work

a) ***Workflow Management:*** Several workflow managers have been developed to orchestrate complex computational pipelines. Apache Airflow [7], for instance, provides a platform for creating, scheduling, and monitoring workflows as directed acyclic graphs (DAGs). Nextflow [4], another popular choice, excels at executing data-intensive pipelines across diverse computing environments. Tools like Snakemake [10], [11] and Makeflow [12], inspired by the Unix make utility, offer a declarative approach to workflow definition. While these systems effectively manage task-based workflows, they focus on task completion and data movement or transformation as triggers for subsequent steps, and are not designed for orchestrating persistent services with dynamic state dependencies. Workflow description schemes like CWL [13] and YAWL [14] are powerful, but Shepherd's YAML-based configuration is simpler and more human-readable, ideal for flexible and rapid deployment.

Shepherd complements these workflow managers by introducing a novel approach to service orchestration. By monitoring service states through log analysis and file creation, Shepherd enables seamless integration of persistent services into existing workflow paradigms.

b) ***Service Orchestration:*** Docker Compose allows control over the order of service startup and shutdown, but it doesn't wait for a container to be 'ready'—only that it's running [15]. Shepherd allows users to control startup through log monitoring, enabling service startup to depend on readiness and other internal states. Docker Swarm ensures high availability with service replication and load balancing [16], [17], while Kubernetes offers advanced features like automated rollouts, self-healing, and detailed monitoring [18], [19]. Shepherd complements these tools by providing granular control over service lifecycles and dependencies on a single machine. Using log-based state tracking and dynamic dependency management, Shepherd enables complex service interactions and workflows, ensuring coordinated execution and enhanced reliability.

Service management tools like systemd [20] are effective at starting and maintaining system-level services, managing basic service dependencies and lifecycles. Shepherd builds upon this foundation, offering a similar approach tailored for the execution of services as tasks within a workflow. It incorporates features like log-based state tracking and dynamic dependency management. This allows Shepherd to handle complex service interactions within a workflow, enhancing flexibility and adaptability.

c) ***Log Monitoring and Real-Time Analysis:*** Log analysis is a powerful tool in software engineering, used for anomaly detection [21], [22], event extraction [23], [24], performance and security auditing, and many other applications.

This technique helps in identifying patterns, detecting issues, and gaining insights into system behavior, often in real time which can be used to trigger other system behavior. Shepherd employs simple pattern matching to identify dynamic states in the service execution lifecycle. However, this concept can be extended by integrating more advanced event matching algorithms such as SLCT [25] and LogCluster [26], further enhancing Shepherd's capabilities.

VII. Future Work and Conclusions

Future work for Shepherd includes extending state tracking to external sources like database queries or custom script executions, which would enhance versatility and accommodate more workflows. Another area for improvement is robust failure handling. Currently, Shepherd performs a graceful shutdown of the entire workflow if any task fails. Introducing mechanisms to restart only the failed task could enhance resilience and uptime. These enhancements will involve balancing additional functionality with system performance, making Shepherd more powerful for managing complex workflows in various simulations and real-world applications.

Shepherd provides a robust framework for integrating service workflows into traditional task-based workflows. It effectively manages dynamic state dependencies among services by inferring internal states via log monitoring. This allows Shepherd to run services as tasks, which can be plugged into any traditional workflow. Shepherd is especially useful when a local workflow requires orchestrating complex dependencies between multiple services and actions that depend on each other's dynamic states. This local workflow can then easily be used as part of a distributed workflow. This approach has proven valuable in managing complex, large-scale drone simulations, demonstrating Shepherd's practical application and utility in real-world scenarios.

Availability

Shepherd is open-source and available at: https://github.com/cooperative-computing-lab/shepherd

Acknowledgment

This work was supported by NASA grant 80NSSC23M0058, "A Safety-Aware Ecosystem of Interconnected and Reputable sUAS"

References

[1] B. Sly-Delgado, T. S. Phung, C. Thomas, D. Simonetti, A. Hennessee, B. Tovar, and D. Thain, "TaskVine: Managing In-Cluster Storage for High-Throughput Data Intensive Workflows," in *18th Workshop on Workflows in Support of Large-Scale Science*, 2023.

[2] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[3] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock, "Kepler: An extensible system for design and execution of scientific workflows," in *Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM*, vol. 16, 07 2004, pp. 423 – 424.

[4] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nature biotechnology*, vol. 35, no. 4, pp. 316–319, 2017.

[5] S. Bowers and B. Ludäscher, "Actor-oriented design of scientific workflows," in *International Conference on Conceptual Modeling*. Springer, 2005, pp. 369–384.

[6] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the kepler system," *Concurrency and computation: Practice and experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[7] A. S. Foundation, "Apache airflow documentation," https://airflow.apache.org/docs/apache-airflow/stable/, 2024, accessed: 2024-07-25.

[8] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the tenth european conference on computer systems*, 2015, pp. 1–17.

[9] S. Project, "Sade: Safety-aware ecosystem of interconnected and reputable suas," https://sites.nd.edu/uli-drone-reputations/, 2024, accessed: September 16, 2024.

[10] J. Köster and S. Rahmann, "Snakemake—a scalable bioinformatics workflow engine," *Bioinformatics*, vol. 28, no. 19, pp. 2520–2522, 2012.

[11] F. Mölder, K. P. Jablonski, B. Letcher, M. B. Hall, C. H. Tomkins-Tinch, V. Sochat, J. Forster, S. Lee, S. O. Twardziok, A. Kanitz *et al.*, "Sustainable data analysis with snakemake," *F1000Research*, vol. 10, 2021.

[12] M. Albrecht, P. Donnelly, P. Bui, and D. Thain, "Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids," in *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, ser. SWEET '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2443416.2443417

[13] P. Amstutz, M. R. Crusoe, N. Tijanić, B. Chapman, J. Chilton, M. Heuer, A. Kartashov, D. Leehr, H. Ménager, M. Nedeljkovich *et al.*, "Common workflow language, v1. 0," 2016.

[14] W. M. Van Der Aalst and A. H. Ter Hofstede, "Yawl: yet another workflow language," *Information systems*, vol. 30, no. 4, pp. 245–275, 2005.

[15] Docker, Inc., "Docker compose," https://docs.docker.com/compose/, 2024, accessed: 2024.

[16] D. Merkel *et al.*, "Docker: lightweight linux containers for consistent development and deployment," *Linux j*, vol. 239, no. 2, p. 2, 2014.

[17] Docker, "Docker swarm," Online; accessed on [date], 2014. [Online]. Available: https://docs.docker.com/engine/swarm/

[18] C. N. C. Foundation, "Kubernetes," Online; accessed on [2024-07-23], 2014. [Online]. Available: https://kubernetes.io/docs/

[19] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[20] L. Poettering, K. Sievers *et al.*, "systemd," 2010. [Online]. Available: https://www.freedesktop.org/wiki/Software/systemd/

[21] A. Vervaet, "Monilog: An automated log-based anomaly detection system for cloud computing infrastructures," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 2739–2743.

[22] W. Xiong, W. Chen, J. Liu, and K. Zhao, "An anomaly detection framework for system logs based on ensemble learning," in *Pacific Rim International Conference on Artificial Intelligence*. Springer, 2023, pp. 52–65.

[23] N. Algiriyage, R. Prasanna, K. Stock, E. E. Doyle, and D. Johnston, "Dees: a real-time system for event extraction from disaster-related web text," *Social Network Analysis and Mining*, vol. 13, no. 1, p. 6, 2022.

[24] R. Yang, D. Qu, Y. Qian, Y. Dai, and S. Zhu, "An online log template extraction method based on hierarchical clustering," *EURASIP Journal on Wireless Communications and Networking*, vol. 2019, no. 1, p. 135, 2019.

[25] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*. Ieee, 2003, pp. 119–126.

[26] A. Woodard, M. Wolf, C. Mueller, N. Valls, B. Tovar, P. Donnelly, P. Ivie, K. H. Anampa, P. Brenner, D. Thain, K. Lannon, and M. Hildreth, "Scaling Data Intensive Physics Applications to 10k Cores on Non-Dedicated Clusters with Lobster," in *IEEE Conference on Cluster Computing*, 2015.