**Software Architecture for Multi-Type Workflows:**

**Pillars of an Interoperability Framework**

Table of Contents

[About this Document 2](#_Toc118145729)

[Revision History 2](#_Toc118145730)

[Workflow Types & Site Pillars 3](#_Toc118145731)

[Motivations & Architectural Qualities 5](#_Toc118145732)

[Major Objects 8](#_Toc118145733)

[Subsystems & Major Components 9](#_Toc118145734)

[Auth 10](#_Toc118145735)

[Run 11](#_Toc118145736)

[Registrar / Event Handler 13](#_Toc118145737)

[DT4D Runner 14](#_Toc118145738)

[Repo 15](#_Toc118145739)

[Repo.MetaRepo 16](#_Toc118145740)

[Repo.RunRepo 18](#_Toc118145741)

[Repo.StorageDrivers 18](#_Toc118145742)

[Spin 18](#_Toc118145743)

[Interface: inC2 19](#_Toc118145744)

[Interface: DT4D API 20](#_Toc118145745)

[Interface: Py4DT4D 21](#_Toc118145746)

[Interface: J4DT4D 22](#_Toc118145747)

[Interface: DT4D GUI 22](#_Toc118145748)

[Interface: lwfm 24](#_Toc118145749)

[Use Case Examples 25](#_Toc118145750)

# About this Document

This conceptual design document will:

* Present, in thumbnail form with references to the longer form paper, an architecture for distributed workflow systems characterized by three types of workflows (in-situ, intra-site, inter-site) enabled by a workflow system comprised of four pillars (Auth, Run, Repo, Spin).
* Enumerate several interfaces for programmatic and human interaction with the workflow system.
* Describe in some additional detail specific implementations of these abstract subsystems and interfaces, though likely not enough to be considered an implementable specification nor manual for operations.

## Revision History

Major updates to the document are listed here.

October 2022 initial version

# Workflow Types & Site Pillars

To summarize our paper, we conclude there are three kinds or types of workflows:[[1]](#footnote-1)

* Type 1: in-situ workflows, within a Job and/or resource allocation (e.g., a coordinated multi-app / MPMD / multi-physics workflow running within a single user’s HPC Job allocation)[[2]](#footnote-2)
* Type 2: multi-Job workflows within a site / organization / enterprise (e.g., a typical HPC scheduler runs multiple Jobs on behalf of the same/different users concurrently and/or in sequence)
* Type 3: multi-Job / multi-site workflows (e.g., a MxN CI/CD system which builds and tests multiple applications on multiple target platforms with rollup reporting)

In order for *Sites* to interoperate in a workflow, we need to normalize their interface along four main functional pillars or subsystems:

* Auth: authenticate the user on the Site
* Run: run Jobs on the Site, chain Jobs together, get Job status, cancel Job
* Repo: put & get data to/from the Site, ideally using metadata decoration and search
* Spin: provision resources from a cafeteria of Site options, even from thin air (i.e., cloud) – spin up, spin down, track usage

Timeline

Description automatically generated

The tooling which supports this framework includes several key components:[[3]](#footnote-3),[[4]](#footnote-4)

* Auth: an enterprise authorization component, which knows how to authenticate Users and associate them with a user group or Tenancy.[[5]](#footnote-5)
* Run: a Job execution runtime, most likely batch. This might be a traditional HPC scheduler like PBS, or some other “runner” agent which can run a Job.[[6]](#footnote-6)
* Run: a “local” runtime, allowing Jobs which run on the User’s own machine(s) to be 1st class citizens in the ecosystem. Again, Jobs typically run asynchronously.[[7]](#footnote-7)
* Run: a component to track Job status and fire registered event handlers. This allows Job chaining, and workflow joins following branching. [[8]](#footnote-8)
* Repo: a persistent store of Job status information – call it RunRepo[[9]](#footnote-9)
* Repo: a persistent store of metadata – call it MetaRepo. MetaRepo can track data which is stored in a variety of locations in a variety of formats behind a variety of protocols (i.e., there is no monolithic “data Repo” – the MetaRepo tracks the data where it lives). We note that RunRepo + MetaRepo = the digital thread. Data is associated with the Job which created it and the Job(s) which used it. The thread is navigable from control flow or data flow perspectives.[[10]](#footnote-10)
* Spin: one or more 3rd party cloud vendor implementations, AWS, Azure, and the like. Most of the Spin subsystem is façade on 3rd party tooling.
* Interface: a programmatic interface for authoring workflows using the verbs provided by the Site subsystems. Python is popular, as is Java, both potentially fronting a language-neutral REST API.[[11]](#footnote-11)

Workflows across Sites require a simplifying normalization of interfaces – standard ways to run Jobs, normalize Job status, get and put data. These simple interfaces must have wide open argument lists to handle the complexity inherent in the Site. Thus, the Site framework presented here attempts to provide that simple interface, and it includes functions / methods which often take arbitrary arguments.

Thus, there is no effort to hide the complexity of the Site – we intend for users to be able to use every Site-specific feature of, for example, some very specific runtime in a heterogeneous ecosystem. We present here an interoperability framework, not a portability framework. There is no effort to provide tooling for authoring workflows which can be deployed anywhere / run anywhere. We instead acknowledge the vast array of workflow tools, runtimes, and workflow encoding schemes, and encourage the user to use what they like, as an “application”, then weave these multi-Site workflows together as “the workflow is the app”.

The perspective of inter-Site workflows is “me and my data”. The point of view of the User is their own local workstation. From this vantage they orchestrate workflows across a number of system Sites to which they have access. (e.g., workflows which use local processes and enterprise workflow systems and external systems like those at national laboratory leadership computing centers).

There is no assumption that a process on a remote Site can reach back to lwfm running locally. All communication is 1-way. Everything involving remote Sites involves polling, not callbacks. Implementations of type 2 tooling – intra-site workflow tooling – are of course free to use message buses or any other 2-way communication construct and can make assumptions about the state of connectivity within the enterprise.

# Motivations & Architectural Qualities

DT4D was motivated by a desire to leverage commodity hardware for embarrassingly parallel engineering workflows. Design spaces can be explored in pseudo-random and guided walks in workflows involving up-front pre-processing, then utilizing HPC CFD simulations for physics-based data generation, then applying that data in the training of surrogate ML applications deployable to the field with their runtime cost savings. Throughout, the imperative to keep data separated by its (often legal) data classification was paramount. From this, the model of { Auth, Run, Repo, Spin } was developed, and with the exception of Spin, was implemented.

Subsequently, with separate customers, we became aware of a need to manage workflows within a running HPC Job, to provide for in-situ data extraction, analysis, routing to visualizations, routing to interactive users, with direct impact on the steerage of the running simulation. With modern heterogeneous hardware simultaneously useful for CFD and ML applications, co-launched applications are also possible to realize the above original DT4D use case. DT4D also provided much of the infrastructure necessary to achieve the wide-area aspects of the HPC workflow. A broadening of the concept of managed workflows in the enterprise began.

Further engagement with the national laboratories showed many things. At Sandia, we encountered researchers working on workflow systems similar to DT4D. At Oak Ridge and CINECA in the EU we explored an “MxN CI/CD” concept whereby M applications are kept buildable and validated in the face of code modifications and changes to N deployment platforms. This uncovered the need for a unifying interface over remote computing Sites so that cross-site workflows like the MxN CI/CD could be implemented scalable using a standard syntax which hides the per-Site complexity. At NERSC, we found the beginnings of this interface in their Superfacility API. An exploration of similar-to tools in the open source market[[12]](#footnote-12) let to the development of the “workflow type 1, 2, 3” concept and the lwfm implementation. As time went on and work on lwfm progressed, it became clear there was going to be strong if not literal convergence in the type 2 and 3 tooling.

We continue to publish on this refactoring and seek collaborators at the above and other institutions. In the meantime, GE as a corporation is breaking up. This results in one corporate entity becoming three. The GRC HPC service, once a corporate service, finds itself now owned by one of three companies, providing services back to the other two. Inter-enterprise, type 3 workflows between former GE entities are now quite plausible.

The design is always motivated by attention to certain architectural qualities:

* Security – data is owned by users and their teams. Access to system runtime and data resources is controlled, segregated by Tenant. Data classification is respected.
* Auditability – whatever the system does should leave a trail. The “digital thread” from both control and data flow perspectives should be navigable. It should always be answerable “how did this piece of data get here?”.
* Heterogeneity – there is a “Cambrian explosion” of hardware and a high barrier to entry for HPC systems. The complexity due to heterogeneity must be controlled.
* Loose Coupling – a means to control complexity, components are singular in purpose, publishing and adhering to contractual interfaces with other components in the ecosystem
* Testability, Modifiability, Reusability – atomicity, the basis of loose coupling, results in fine grained testability and composable reuse
* Scalability – HPC workflows often demand runtime extremes in terms of computing power, data size, etc. Systems must be ready to handle large Jobs.
* Performance – our engineering users often make this architectural quality of paramount importance, a view not necessarily shared by this author, but its undeniably important
* Availability & Reliability
* Usability & Supportability – the current design and implementation refactoring, as a simplification, results in lower barriers to entry and improved supportability. Several DT4D features also lend aid to incident troubleshooting and remedy – the existence of the RunRepo audit trail, cross-referenced by the MetaRepo, are excellent aids. The DT4D runtimes also capture Job log information and store it centrally – these are invaluable for assessing problems within the Job’s own user code, *even if that Job was run on the User’s local desktop*![[13]](#footnote-13)
* Immutability – data, and metadata, should be tracked to perpetuity. Disk is cheap. Historical data may someday (read: AI) become useful.

As noted in sections above, we do not endeavor to describe a workflow system where workflows are portable. We make no effort to force an implementation on a collaborating Site – Sites may implement, or not implement, the interfaces in as rich as manner as they desire. Methods / verbs for the subsystems provide for arbitrary arguments to be passed, permitting the addressing of those differences. We aim for interoperability, not portability.

As mentioned, we do not endeavor to write portable workflows. So how shall we express these workflows? Many mechanisms exist, usually around the idea of the workflow being fixed *a priori*, then scheduled in a maximally optimal way on the allocated resources. Workflows can also be rewritten by the tooling like code, optimized by the “compiler” as it were. Sometimes these workflows are focused on in-situ control. Sometimes they express type 2 workflows, like those executed by a traditional HPC scheduler. They can be authored in a markup language, or in a higher level language which produces some portable markup language. Or in a visual GUI which produces the same. Editing markup can be tedious, and GUIs are often restrictive to advanced users. Programming languages, especially Python, seemed most appropriate for expressing workflows.

Given the popularity of higher level languages like Python, and inter-operable Sites adhering to published interfaces, including one which schedules a Job for execution, it became clear that again, trading performance off for other architectural qualities, workflows need not be specified *a priori*. A Python script can call the “run Job” Site endpoint at will and have no expectation of a complete re-optimization of the workflow by the system.

However, a workflow which can create and modify its own future can be terribly difficult to contemplate and debug unless the system provides good interrogation functionalities, and likely also visualization features. Given the prioritization of auditability especially in the context of preserving the digital thread for our industrial users, the system proposed is already rich in interrogation capabilities. Visualization has been shown in two ways – in a columnar format (as is currently in DT4D production), and in prototype form as a navigable graph data structure.

# Major Objects

The model contains and code implements several major objects, some of which get extended by implementors of new Site drivers.

Graphical user interface

Description automatically generated

* The Site interface consists of four sub-interfaces.
* A Site Driver, like the Local Site Driver, implements the sub-interfaces, or borrows them from other Site implementations.
* A Job Defn represents the description of a Job. Once submitted to the Site Run subsystem, it becomes a Job running inside a Job Context, and emits Job Status messages each associated with the same Job Context.
* Site File Ref abstracts “file” objects on remote Sites. Those might be files on actual filesystems, or some other more exotic form of object storage, perhaps one which also tracks object metadata.
* Job Event Handler allows Sites which support it (like the lwfm local Site) to provide registration of a job to respond to upstream job status events.

# Subsystems & Major Components

For each subsystem / component (i.e., nouns), we’ll identify its purpose, its major verbs (i.e., functions, methods), then provide details to the point of pertinence, short of an implementable specification. We’ll end each section with some suggested topics for future work.

**Graphical user interface

Description automatically generated**

## Auth

**Purpose**: Authenticate users and identify their scope within one or more *tenancies* (a collection of users with a common need-to-know interest and data classification(s)).

**Verbs**: Login. Check login.

**Description**:

A given Site need not implement any concept of tenancy other than the bootstrap case of 1 user = 1 tenant. A Site may however permit the grouping of similar Users. Typically, a Tenancy is comprised of Users with some common interest in some subject (e.g., a project) and which have rights over the resulting work products of the Tenancy – i.e., the Data Classification of the team or project. Data Classification examples include “US Export Control”, where all User members of the Tenancy group must be verifiably US Citizens.

A Tenancy may have sub-tenancies – a group of groups. Theoretically the org tree can be arbitrarily deep, but practically, the Site can choose whatever depth is appropriate, or no depth at all (1 user = 1 tenancy). For example, a Site might implement group / tenancy membership using an LDAP or Active Directory, with their Site.Auth driver providing the normalizing interface to this enterprise directory.

A call to Site.Auth.login() might result in an interactive exchange of user login credentials, and the return of some secure token with finite time-to-live. The Site implementation would then utilize this token when calling Site services to realize the implementation of the Site driver.

Such Tenant-sensitive Site services might include, for example, returning a subset of all Site computing resources when a call on Site.Run.listComputeTypes() is made, showing the calling User only those resources available to it in its member Tenancies.

Validating Users as having the proper governance paperwork for membership in, for example US EC Tenancies, is not in the scope of this design.

**Future Work**:

* “Bi-Directional Zero Trust” – Consider the case of a User of a type 3 workflow tool who wishes to deploy their critical IP data to a remote Site and make use of its compute capabilities. The User will authenticate with the Site. Will the Site authenticate with the User, in a way that the User can directly validate they are indeed talking to the Site before moving their critical IP? This same bi-directional authentication construct might also apply to the distributed portions of the Site or Site Compute Type runtime. (In DT4D, this is our remote Runner agents, identifying themselves to Site.Auth and being granted permissions to run batch jobs on behalf of a Tenancy. Similarly, the DT4D System Runner concept is authorized to run jobs on behalf of any Tenancy, so long as the code is authorized as such by a process outside the scope of the workflow software.)
* Hierarchical Tenancy & RBAC – In a Site which implements hierarchical tenancies (i.e., groups of groups of users), there might be the need for certain “super users” who have jurisdiction over all of an enterprise’s Tenancies, or some portion of the subtree. This is akin to the super user being a member of the tenancy. Use cases include the “data steward” who might groom metadata for an enterprise, or an admin who manages tenancy membership, or the Tenancy structure itself. At the moment, this design is mute on such topics – tenancy membership management is delegated to the Site to do as they wish.
* Modal vs. non-modal Tenancy membership – i.e., having a single Tenancy for the lifetime of the Job – is an implementation detail. DT4D currently implements modal Tenancy – the User may be a member of more than one Tenancy but selects a single one modally. This is a safety feature (e.g., a user would never be able to accidentally copy EC controlled data to a non-EC Tenant to which they were a member, thus exposing the data). However, it also makes certain tasks more difficult, such as maintaining metadata for more than one Tenancy, or copying data to another Tenancy on purpose.

## Run

**Purpose**: Auditably execute arbitrary Jobs on behalf of users. Allow Job chaining – the triggering of a Job’s execution based on an upstream event (e.g., Job completion).

**Verbs**: Run job. Check job status. Terminate job. List Compute Types. Set / unset event trigger, list triggers.

**Description**:

A Run subsystem consists of two major components:

* A runtime, which can execute a user application / script within the context of a Job
* An event handler apparatus – a separable piece of middleware – which can hold registered event handlers and fire them when the triggering event occurs

We will not include the User’s workflow scripting language in the Run subsystem – it can vary (Python, Java, etc.) without altering the base API nor semantics. We’ll call the scripting language an “Interface” – see that component separately below. We’ll also delegate storage of all Run subsystem and Job status logging to the Repo.

The runtimes fronted by a Run interface can vary. An HPC scheduler is a runtime. A driver which implements the Run interface using PBS as the scheduler can be implemented. DT4D has its own remote “Runner” agent which implements a Job runtime. There are also local runtimes – these can make tasks which run locally on the User’s desktop or department server true Jobs and therefore 1st class citizens of the pan-Site ecosystem.

Compute Type is a device or computing facility within the Site. It might be an HPC cluster. It might be a commodity node painstakingly pre-prepared with a particular licensed engineering application. It might be a deployable Docker container.

Compute Types can be modeled as Sites, or as named resources within the Site, whichever is easier to conceptualize for the implementor of the driver which implements the interfaces described in this document. If Compute Types can be instantiated ephemerally (see discussion of the Spin subsystem below) then modeling them as elements of a single Site might be more practical. In either case, it’s the implementor of the Site.Run driver which must understand the per-Site or per-Compute Type variations and implement them while hiding those complexities from the invoker of the Site.Run interface.

Sites and Compute Types might naturally have extensive metadata. If desired by the Site driver implementor, the Site.Repo.MetaRepo component provides a means to store and retrieve that metadata about the computing platform. We note in the discussion of Spin below the ability to see a cafeteria of Compute Type options. We will consider these to be just a list of cafeteria menu options – their descriptors, not their instantiations. This Spin.showCafeteria() would return a menu of options, while Run.showComputeTypes() would return the resources that the User currently has at their disposal (though those resources, such as in the example of an HPC cluster, might be fronted by a scheduler – this is irrelevant, and nearly assumed to exist). We also note that Run.showComputeTypes() might also return information about static, not “spun” resources. The list might be filtered by the Site to be per-Tenant.

We can conceptualize a very broad set of Compute Types – not just computers, but devices of many kinds. A microscope may have a concept of “running” an experiment – what we can call a Job. It takes configuration data as input and produces data which a workflow would intend to liberate – see the discussion of Repo below. A Site such as a research institution may have many unusual Compute Types.

Compute Types conceptually extend into type 1 workflows – being able to park different applications / processes on different parts of an HPC allocation, with fine-grained control (e.g., some nodes have GPUs, some don’t). See the discussion of the “Interface: inC2” below.

A Job may have two identities – a native id known to the Site, and a portable id which can be used in type 3 inter-Site workflows. A Job can have a parent, which is its immediate predecessor in a chain of Jobs, and a Job can have an originator, its eldest grandparent. We track the originator as a graph traversal optimization, and It provides a mechanism to associate a set of Jobs. An additional job set id can be used to create named subsets within the originator’s full set. A seminal Job has no parent and is its own originator. Together these attributes are considered the runtime “Job context”.

A Job goes through a sequence of statuses / states, from “about to be queued” to “Job done” with many states in between. Some status messages can be purely informational - Job status messages can usually also, and in our implementations do, allow an arbitrary payload. This is especially useful for information status messages which aren’t state transitions per se, and DT4D uses this payload to carry information such as references to the data elements being put or gotten from the Repo, or info from a running HPC job, or a declarative GUI which can be displayed by the DT4D GUI permitting Job steerage.

Some statuses may be terminal (e.g., “cancelled”, “completed”, “timed out”, “failed”). Sites, and perhaps even Compute Types, will have their own set of status names. At the type 3 level, and perhaps even in some cases at the type 2 level, we need a mechanism to normalize / map these native status names to a canonical name – see “Interface: lwfm” below.

**Future Work**:

* The list of “Interfaces” below includes several local runtime implementations (e.g., Py4DT4D, lwfm, DT4D Runner). It would be preferable to have just one implementation, and for it to operate at the type 3 level, perhaps with pluggable drivers if needed. See the comments below on those components.
* Implement a single Registrar / Event Handler which fully implements the triggering model and can be used in either type 2 or type 3 tooling.

### Registrar / Event Handler

Running Jobs causes a sequence of Job status messages to be emitted. All the runtimes do this, not just the ones described in this document but generally too. Status messages, if coming from an internal but different system (e.g., an internal HPC) or an entirely external Site in the case of type 3 workflows, may need to be normalized to a standard set of names. As mentioned above, status messages can also carry a payload.

Once they are normalized, they can be used to trigger downstream workflow events, specifically, the running of a Job. The Job triggered will be a child of the triggering Job in the digital thread.

Conceptually there are two kinds of triggers: “job triggers” and “data triggers”. These map to control flow and data flow, respectively. Both are based on events from the Job status sequence, data triggers using the status payload to determine if a data move of interest occurred, and if so, to fire the registered Job. Job triggers are based on the states themselves – “when Job A finishes, run Job B”, or “when the set of Job S finishes, run Job T”.

The Registrar in DT4D, called the Event Handler in lwfm, exposes three major verbs – set handler, unset handler, list handlers. Handlers are in scope for a Tenant. Thus, inter-User workflows or departmental hand-offs are possible.

A range of conceptual logical arrangements are possible in a full-blown event handler implementation, including:

* Fire when state >= name (can be used for any state, for success or failure or interim state handlers)
* Fire when set reaches state
* Fire on partial set completion
* Fire when satisfied or time T elapses
* Fire on the event recurring *N* times then evict
* Fire as many times for TTL then evict
* Fire on a schedule
* Fire when Job puts or gets data with metadata M (e.g., “fire when design model reaches approved state”)

The Registrar persists its registrations over restarts.

### DT4D Runner

(See the sections below on “Interfaces: Py4DT4D” and “Interfaces: lwfm” for discussion of local runtimes. Here we’ll discuss some details of the DT4D Runner implementation, and as noted above, we may have too many runtimes and a consolidation is possible.)

Authored in Java (as compared to Python for Py4DT4D and lwfm), the Runner is installed under a functional account as an agent on a compute node – any node of any kind – and listens on the internal DT4D message bus for incoming Job messages which are targeted at itself – this being defined by the Tenancy and the Compute Type. Thus, a Runner never runs Jobs for more than one Tenant, giving the node exclusivity to the Tenant. This can be seen as useful for data classification separation as the Job can be arbitrary user script.

A System Runner, in contrast, runs Jobs for any Tenant, but the script is from a curated set of system-approved scripts. An example is a Runner which fronts an entire HPC scheduler, preparing Jobs from the canonical DT4D system for the specifics of that HPC cluster. (We note this view would be significantly different in the presence of type 3 workflow tooling – such tooling didn’t exist at the time.)

In either case, the script is most likely not authored in Java – most likely (and to this point, always) authored in Python. So, the Runner is effectively a shell managing *N* concurrent Jobs per Compute Type on the host, as defined by the Runner’s deployment configuration.

(We note that a more efficient Runner implementation might be authored in Python, as in fact one already practically exists. Direct contact with the bus can be removed in favor of a poll against the public API.)

Runners are fitted with a command process, which send out heartbeat pings and independently listen for commands (note: could poll an API) which include lifecycle events like pause, resume, halt, restart, and update, which causes the Runner to download a new copy of its own code and bounce itself, thus updating. The Runner waits until Jobs it is running to complete the update or can be forced. Also, when a Job is run, the Runner downloads a fresh copy of Py4DT4D and notes the version used in the digital thread.

## Repo

**Purpose**: Store and serve up data on behalf of tenants. Track metadata about the data. Adhere to FAIR principles for data.[[14]](#footnote-14)

**Verbs**: Put / edit metadata. Get metadata. Find data by metadata. Put data. Get data. Write and read Job status info.

**Description**:

In a generalized metadata system, there may not be any data – the *target* of the metadata might not exist in the digital world but in the real one – a real person, a computer. When you consider groupings of data (simply using the metadata fields themselves), you end up with something that looks a lot like LDAP or Active Directory – an arbitrary collection of name=value sets, with relationships between sets defined by the metadata.

An LDAP browser, thus shows the individual *metadata sheet* rendered in its place in a hierarchical tree, as defined by an ordered list of the metadata. A similar GUI was built for DT4D (the “Meta Tree View”) which allows the User to order the list of metadata fields themselves, permitting multiple and ad hoc hierarchical views.

Only two components are required to exist in this subsystem – a *MetaRepo,* simply the collection of metadata sheets, and the *RunRepo*, which is the running log of all Job status messages. We could have delegated RunRepo to the Run subsystem, but the RunRepo in conjunction with the MetaRepo represents the digital thread, and thus we decided to group it logically in the Repo subsystem.

While only metadata storage is required, the Repo subsystem *might* include its own actual managed storage. In DT4D this is primarily in S3, but the metadata sheets may reference its target in any storage medium given a driver in the system exists for that storage. So, we consider Repo.Storage an optional component in the model, though a system without such managed data stores would be unlikely. A Repo.Storage driver might for example be an HTTP URL of a data service, or a departmental server filesystem reference, or HDFS, or generally anything. Given the wide variety of options, we intentionally separate metadata from data, since not all storage mechanisms, or Sites, support metadata (though for example, S3 does).

The MetaRepo can also maintain an index of keywords – the unique metadata field names for a given Tenant – to facilitate user experiences such as GUI auto-fill.

It is assumed that the MetaRepo is a singleton in the type 2 or 3 system[[15]](#footnote-15), though it may be implemented as a distributable database. MetaRepo, like its owning Repo subsystem, exposes an interface which can be implemented in a number of ways from Elasticsearch to budget flat file implementations. An interface and driver architecture are used here and nearly everywhere in this system design.

Since these data movements are potentially time consuming, it would be expected that Repo implementations would wrap them as Jobs, run them asynchronously, and utilize workflow event handling. It also raises the question of transactions with the MetaRepo – does the metadata sheet get created before or after the data object (if one exists) is put under management? What if something bad happens? We conclude that since some data elements targeted by the metadata sheet are beyond the scope of the system, that the data object should be known to exist prior to writing a metadata sheet to front it. Thus, there is only a trivial two-step transaction and no transactional rollback – if the data writing step, should one be needed, fails, there is no reason to move to the writing of the metadata sheet. A failure of writing the metadata sheet (which should be avoided, with retries, etc.) would not cause the data write to be reverted. Thus, the put of data and the put of the metadata are separate verbs in the subsystem. A utility method could be created which would effectively just fire the first write, and if successful, fire the second. This might be in tandem with a Run subsystem helper which treats the data write as an asynchronous job.

### Repo.MetaRepo[[16]](#footnote-16)

The MetaRepo contains a collection of metadata sheets[[17]](#footnote-17), each describing a target, which may be a real physical or digital world entity, or entirely imaginary. The sheet can represent anything. The user is free to consider the target to be of whatever type they want and use the metadata to express the target type.[[18]](#footnote-18) There is a 1-to-1 relationship between metadata sheets and targets – a copy of a target object would have a second metadata sheet.

The sheet is organized into a few logical sections or namespaces:

* Framework – this is where this architecture requires a few fields, but not many. The sheet must have a unique identifier – this will be important in the digital thread. It must record its creation date, its tenancy (tenants own their data[[19]](#footnote-19) and their metadata), its Site, and it must provide for version control.[[20]](#footnote-20) In the spirit of immutability, an original metadata sheet should remain in the system while edits create new instances which point back to the origin in a version history. Thus, rollbacks via soft delete of sheets are possible.
* Driver – this is where the specific implementation of the MetaRepo interface can put information which is specific to the details of that MetaRepo driver implementation.
* Target – different targets may have their own metadata. A filesystem target is referred to probably by path, while other storage mediums have their own identification schemes, URIs, and the like – S3, HTTP, HDFS, DTR, etc. There is no logical reason to consider these targets to be part of the system per se, but rather, part of the IT infrastructure of the enterprise – in fact, some of these targets might be decidedly outside the scope of the system, owned by external teams or agencies.
* User – arbitrary metadata fields invented by the User, maintained by the User.

While metadata editing is permitted, the implementation should only allow users to edit sheets owned by their tenancy, and also, to only edit User metadata. The implementation should handle edits to other namespaces itself. It’s an implementation detail if the system wants to handle the edit as a Job and could thus issue Job status containing information which could be used by a Run subsystem event handler. Bulk edits should be permitted – to find a set of sheets by their metadata, then edit them in one step.

Within the user metadata section, Users such as a data steward for a Tenancy might invent further namespaces within this section by using a dotted notation for names or leveraging other metadata fields for namespace identification.

Each of these namespaces can be defined by a *schema*. The MetaRepo should implement a validate() method which applies the schema to a sheet namespace. This method can check for the existence of certain fields, perhaps their ranges, or any other validating criteria desired by the MetaRepo implementation.

Searches of the MetaRepo are expressed as conjunctions of search clauses, each clause being a name=value pair to match, fully or partially. Wildcards are permitted, as are “fuzzy” searches, where the term can be no more than some set number of rewrites away from a match (e.g., “today” and “toady” are two rewrites apart). Each clause can be assigned a relative priority, which allows searches to, for example, cast a net widely and then narrow the scope. These are all however implementation details as not all MetaRepo implementations will support, for example, fuzzy search. The method signature on MetaRepo.search() must permit for implementation specific expression of the search. Exporting of search results should also be supported.

Since Run subsystem runtimes can be widely distributed (e.g., both on-prem and in the cloud for a given enterprise, perhaps modeled as two Sites, perhaps as one with two Compute Types, whichever), it might make sense to keep copies of certain data files local to the runtime (in the above example, we might have two S3 repos, one on-prem and one in the cloud, and selectively copy certain data objects from one to the other). Each copy would have its own metadata sheet, which might notate a “location” metadata field which a runtime Job can later use to query “find me the data which is local to ‘my’ location”. When located, the metadata sheet’s target metadata would include the callable reference to the data in that location (e.g., the local S3 URL).

### Repo.RunRepo

The RunRepo is simply a running log of all Job status messages. The Registrar component, being a listener on Job status messages for the purposes of assessing event handler firing, can call a put() method on Repo.RunRepo. This component should also provide for a “find by Job id” method.

### Repo.StorageDrivers

The target referred to by the MetaRepo might not be digital data, but it might be. And it might have existed prior to the workflow, or it might even be on an external system. We conceptualize a Repo.StorageDrivers component which provides a simple put/get interface and a set of driver implementations for different storage mediums – S3, filesystem, and the like.

**Future Work**:

* Implementation of the above, which is an extension of the current DT4D MetaRepo implementation. Develop a solution which can be applied in both type 2 and 3 systems. Thus. a User armed with a type 3 workflow tool including MetaRepo can track data on Sites which have no MetaRepo implementation – the target is on a remote Site.
* Application of MetaRepo to type 1 workflows, with specific attention to performance.

## Spin

**Purpose**: Allow the user to provision fixed or instantiate ephemeral resources such as cloud computing nodes or complete turnkey clusters. Expose costing information to the user.

**Verbs**: Spin up. Spin down. Show cafeteria. Show costs.

**Description**:

The Spin subsystem relies heavily on the APIs provided by cloud computing vendors and their aggregators. For example, Amazon AWS, MS Azure, Google Cloud – these services all provide programmatic mechanisms to (de)instantiate cloud services (“make me a 16 core VM with 64 GB RAM and 1 TB storage”). Some cloud services provide entire turnkey systems with just a few simple method calls. These cloud services also provide programmatic ways to administrate the platform, including cost reporting.

The implementation of Site.Spin for AWS is therefore a mostly trivial invocation of the AWS API. A Site.Spin driver for Azure would be similar. The implementation of a corporate Site might, in order to provide a form of hybrid cloud, utilize an AWS Site driver as a Compute Type within the Site.

The cafeteria of available compute options which are able to be provisioned can be Tenant specific, as desired by the Site. The Site might also return the list of available options decorated with detailed metadata for each, e.g., cost in USD per compute hour, cost per TB per month, flops or other performance metrics, etc.

**Future Work**:

* The entire Site subsystem is future work. Most of it is line of sight to implementation given the existence of rich cloud vendor APIs.
* The above includes the implementation of the Site.Spin.showCosts() function. A call to Site.Spin.spinUp() would instantiate a Compute Type from a cafeteria of identified options. This Compute Type would have a cost per unit time. The Site.Spin.spinUp() method would run as a Job, and thus be trackable in time by the framework. The Site.Spin.spinDown() is similar, and the delta is the unit of time for billing. Rollup reporting is also possible by noting the Jobs associated with a User, and a Tenant.

## Interface: inC2

**Purpose**: Provide a messaging interface for orchestrating intra-Job / in-situ workflows.

**Verbs**: Spawn child. Get parent. Send message. Check messages.

**Description**:

This library, available in either Python or C++, permits bi-directional messaging passing between distributed processes, like those distributed across nodes in an HPC cluster. The message passing is implemented with MPI, though that is insulated from the invoker.

A typical scenario: an HPC C++ application is fitted with the inC2 library permitting it to receive messages which steer its behavior (e.g., “alter the combustion temperature”, “halt gracefully”) or to emit messages containing interim results. Next, a Python script uses inC2 to launch the child C++ app across nodes in the allocation. The Python controller script and the running simulation can now exchange information in real time with performance equal to that of MPI.

Using the wide-area type 2 connection, we’ve shown an in-situ workflow communicating with the type 2 (DT4D) system to emit interim results to a human consumer, display a workflow-specific declarative GUI, accept input from an interactive User, and issue steerage back to the running simulation.

We mention inC2 being used both in the controller and within the application itself. Naturally the latter is only possible if source is available – then the app is a “white box” to inC2 and the app can be fitted with a wide range of controlling functions. If the source is not available, then inC2 will handle the app as a “black box” and do with it whatever the app *a priori* exposes as possible.

**Future Work**:

* Intra-Job messaging is implemented with MPI. Messaging beyond the Job, for inter-Job type 2 workflows, utilizes the Py4DT4D library. One could bury the Py4DT4D library under inC2 or provide it as a pluggable “wide-area communication” driver. Thus, the author of the controller script would have a messaging interface where the invoker was more insulated from the details of the messaging protocols and drivers. ADIOS2 suggests this kind of abstraction over a specific messaging transport.
* We notice that the type 1 workflow tooling is limited to elements of the Run subsystem. The Auth, from a type 1 perspective, is assumed – the Job is already running, the User is already authenticated and authorized to run the Job. However, the Job might invoke wide-area communications, interacting with an Auth component at the type 2 level. More thought is needed to see if it’s useful to have a type 1 Auth construct, similar to abstracting the type 1 and type 2 messaging suggested above.
* A similar discussion could be had over the Repo component – the inC2 controller might desire to store interim results or pull data for feeding to the running simulation. Type 2 constructs could be introduced in a façade / driver implementation in the type 1 tooling.
* Compute Type – a type of resource within a Site – is a concept which given heterogeneity of today’s computing nodes extends into type 1 workflows – within an allocation, inC2 can permit the specific parking of apps / processes on specific portions of the allocation.

## Interface: DT4D API

**Purpose**: The DT4D API is the language-neutral REST interface over the Auth, Run, Repo, Spin subsystems of the DT4D implementation of a type 2 workflow system.

**Verbs**: The collective verbs of Auth, Run, Repo, Spin.

**Description**:

At its best, this interface is nearly or completely identical to the interface specified at the type 3 level. Practically speaking, being at this point a “legacy” system, it will be bent to the type 3 standard as closely as possible. In so much as it can bend to comply with the standard interface, the DT4D components which implement these interfaces are then reusable in type 3 tooling, and vice versa (implementations of type 3 subsystem drivers would be pluggable into DT4D).

**Future Work**:

* Further align the signatures of the interface with the type 3 subsystem standards. This activity would be sensitive to any layered libraries, would likely need to preserve the older versions of the API until those dependent libs are updated.[[21]](#footnote-21)

## Interface: Py4DT4D

**Purpose**:

* Provide a native Python convenience interface to the DT4D REST API for the implementation of type 2 workflows within the GE enterprise.
* DT4D wraps the API signatures with native pythonic methods.
* It also implements a local job runtime, meaning, that Py4DT4D can run jobs locally which are first-order citizens of the Run ecosystem, i.e., Py4DT4D implements aspects of the Run subsystem for local use. This includes runtime management such as the retention of job log files.

**Verbs**: All the verbs implemented by the DT4D API are represented in their pythonic forms, plus elements of the Auth and Run subsystems. Py4DT4D provides a login() method which interacts with the GE internal security service (GE IDM), and a “run local Job” mechanism. It does not implement any local Repo verbs for local storage, rather, uses the Repo services of DT4D.

**Description**:

Py4DT4D does not provide a local Repo subsystem – it relies on the enterprise DT4D for Auth and Repo facilities and simply provides a local runtime, creating Jobs and emitting Job status to the intra-Site type 2 DT4D communication mechanisms (a message bus).

A tutorial for new Py4DT4D users is provided elsewhere, with examples.

It would be appropriate for specific Tenants in their specific domains to author convenience libraries which wrap their common use cases in terms of the basic subsystem verbs.[[22]](#footnote-22) It would also be common and appropriate to author scripts which wrap complex application executions with a convenience “shim” which simplifies its use. These tools can all be stored under Repo management and their use in workflows tracked.

**Future Work**:

* The differences between Py4DT4D in its role as a provider of a local Job runtime and the DT4D Runner component in Java are minimal. Additionally, the lwfm local runtime is very similar to that of Py4DT4D. A unification of these similar implementations into one, with drivers based on the Site model, is suggested.

## Interface: J4DT4D

**Purpose**:

The purpose of J4DT4D is to provide a native Java convenience interface to the DT4D REST API. It is like Py4DT4D in that it wraps the API signatures with native methods but is unlike Py4DT4D as it does not provide for a local job runtime.

**Verbs**: Same as the DT4D API.

**Description**:

J4DT4D is simply a convenience layer on the native REST API for the benefit of Java programmers who want to incorporate DT4D into their code. It is automatically kept in sync with its underlying REST API version. It also provides a small set of utility functions for aiding the user in common tasks such as Auth security token management, as each API call will of course require an authentic token.

Unlike Py4DT4D, but like the bare API, J4DT4D does not provide a local job runtime. It technically could be extended to do so, but we do not intend to unless there is clear need – we will use Py4DT4D for local jobs, which are of course free to incorporate Java apps into their workflows as/if desired. Java code can also shell out to run Python.

**Future Work**:

* Improvements to the DT4D login mechanism are desired, but those likely wait in the project plan until a broader relook at a replacement for the GE IDM.

## Interface: DT4D GUI

**Purpose**: Provide a GUI on the REST API.

* Show the User a view of their Job status, with traversal of the digital thread to associated data, parent / child Jobs, etc.
* Show the User their available Compute Types.
* Show their registered Job event triggers.
* Show type 1 / 2 workflow declarative GUIs. (See the section on inC2, above.)
* Search the MetaRepo and download associated data entities.
* Provide a place to hang prototype interfaces.

**Verbs**: A subset of the REST API. Login. Get Job status. Show Compute Types. Show event handlers. Search MetaRepo. Get data.

**Description**:

A typical Web application providing the above features. In theory, this GUI can be rendered on mobile devices too.

**Future Work**:

* The biggest criticism people have with the GUI is the cosmetics. Improve it.
  + Improve the layout of the search. Implement certain User conveniences (e.g., saved searches, shared / published searches, limit search to my User SSO by default, export search results to file). Improve result paging.
  + Improve rendering of data object metadata – currently shown as a long string when it’s clearly a dictionary / 2-column grid. Improve / promote display of the file download button. Promote use of Job set id.
* Feature flags by tenant – new features can be shown in the navigation panel based on the Tenant. Not all Tenants might get all features. This customization could be extended within a feature page too, if needed. Such feature flags could also be used for page branding (e.g., “show GE Vernova logo”).
* Consider the overlap between type 2 and type 3 GUIs needed – they are nearly if not completely identical. Consider what a client-side type 3 GUI app looks like, one which is also deployable securely within a type 2 enterprise.
* A declarative GUI may include a rich 3rd party GUI, popped on the client side, e.g., a 3D model viewer like ParaView.
* The Meta Tree View is a rendering of data under management as seen through the lens of an ordered set of metadata fields. This renders the distributed sea of data objects as a file tree, a browser. Improve the cosmetics of the current implementation to help promote the concept.
* Workflows may be visualized and navigated in their graph representation. In a control flow rendering, nodes are Jobs, and edges parent-child relationships between Jobs. Nodes can refer to the data objects under management which were produced or consumed by that Job. Workflows can also be rendered by their data flow view, in which nodes are data objects and edges represent downstream usage of the data, and the data which results from Jobs which used the data. Since workflows also consist of triggers, i.e., Job futures, the rendering can express that visually (e.g., color coding) to the user.

## Interface: lwfm

**Purpose**: Demonstrate an implementation of a type 3 inter-Site workflow system. Provide a programming interface for type 3 workflows.

**Verbs**: All the verbs of the Auth, Run, Repo, Spin subsystems.

**Description**:

Each subsystem exposes a small number of verbs – half a dozen or less. Each subsystem is implemented in Python by one or more drivers for specific Sites and their constituent services. Workflows are then authored in Python using the library.

Drivers are intended to be shared across Sites. For example, many Sites implement Auth in their own way, yet Sites owned by a single organization (e.g., NERSC) might use the same Auth mechanism. Thus, the same Auth driver is usable in more than one Site implementation.

Run drivers will likely be similar and reusable when the underlying runtimes are similar, for example, a driver for the PBS scheduler might have common elements across all Sites which use PBS.

**Future Work**:

* Continue to harden the implementation, especially around persistence.
* Position the type 3 tooling as the “next-gen” DT4D with a streamlined implementation and reusable components permitting a phased transition, and cross-pollination of the type 2 and 3 efforts, the proverbial “two birds one stone” as it pertains to implementations and their reuse.
* Seek collaborators on the open source and reusable components, such as researchers at the national labs.

# Use Case Examples

Example 1: in-situ workflow (red box) including a (parallelizable) inC2 controller script (black), co-launches two parallelizable HPC apps (yellow and pink). With inC2, MPI is used to communicate between these collaborators a workflow authored in Python. The inC2 controller also calls out to type 2 workflow constructs (brown, outside the HPC allocation). This enables use cases such as interactive steerage, visualization, etc. Note the inC2 script can run arbitrary user code, including little kernels of analysis code, perhaps auto generated via GUI tools like Paraview.

Diagram

Description automatically generated

Example 2: MxN CI/CD – a type 3 workflow, whereby a local workflow script interacts with N sites to keep M applications built and tested and in a state of continuous readiness.

Graphical user interface

Description automatically generated

Example 3: a Spin workflow, perhaps type 2, perhaps type 3, where a set of nodes are created, data moved to that location, a set of Jobs run, and the nodes destroyed controlling the cost function.

Diagram

Description automatically generated

Example 4: “Surrogate U.” – models are kept under management, each backed by a MetaRepo sheet. The sheet contains fields representing the capabilities of the model – what it can do, what it can’t, its fidelity, its cost to train (how “deep” is the model?). Some models have BS degrees, some PhDs, and their cost to train and/or operate is usually proportional. When models are used in workflows, their results are assessed, and the model metadata is updated with metrics which indicate that success or failure. Over time, the best models are identified. Ensembles of models can also be used, the ensemble modeled in its own metadata sheet. Models which fail to provide good results can be sent back to school for retraining.

* Major Objects
  + Stacked diagram showing the type 3 interfaces w DT4D and the impls, such as MetaRepo, target & driver

1. From the proceedings of the Oak Ridge National Lab Smoky Mountains Conference 2022, a pre-print available here: <https://drive.google.com/file/d/1c3YEVmEAUjbI5urj4PiV2TtjzBUzLlws/view?usp=sharing> [↑](#footnote-ref-1)
2. Nouns in capital letters are proper nouns. Job means “a Site runtime entity which has an id”. It might be a single process, or it might be thousands of processes distributed across an HPC cluster. It means what most people would normally think it means in the context of HPC computing. [↑](#footnote-ref-2)
3. Leaping ahead a bit in this document, we will name the implementations of these components in the GE DT4D and lwfm codebases. DT4D is a type 2 workflow system which manages workflows across multiple machine types within the GE enterprise. lwfm is a very similar but type 3 open source project which can track inter-Site workflows across multiple enterprises. [↑](#footnote-ref-3)
4. Consider the benefits of an “Admin” subsystem along with Auth, Run, Repo, Spin. The NERSC Superfacility API includes many such endpoints – are any worth including in a generalized type 3 Site interface? [↑](#footnote-ref-4)
5. At GE, this is the legacy GE IDM with its management of User SSO ids, group membership, notation of the User’s data classification access parameters, and the dispensing upon valid authentication of a timed access token usable for system API calls. [↑](#footnote-ref-5)
6. In GE DT4D, this is both HPC schedulers and a Runner agent which sits on commodity machines awaiting Jobs to run on behalf of a User / Tenant. [↑](#footnote-ref-6)
7. In GE DT4D, there is a local Job runtime implemented in Python. Same for lwfm. [↑](#footnote-ref-7)
8. In GE DT4D, this is the Registrar component. In lwfm there is a similar component which likewise listens for registrations, then fires Jobs when triggered by upstream Job events. [↑](#footnote-ref-8)
9. Both GE DT4D and lwfm have an appendable log of all Job status messages issued by executing workflows. [↑](#footnote-ref-9)
10. Both GE DT4D and lwfm have a MetaRepo. In DT4D, the Elasticsearch-based database tracks data objects stored on one or more S3 services within the enterprise. In lwfm, the MetaRepo can track data entities which are local to the User (e.g., on the user’s filesystem or department server) , or which are not specifically managed by the lwfm system but rather managed by some external Site. For example, consider a Site which doesn’t implement a MetaRepo concept – the lwfm implementation of MetaRepo can be used to track data on that Site for the purposes of the User’s workflows. Admittedly, the data on the remote Site is not under management per se, and therefore might be altered or even removed. C’est la vie. [↑](#footnote-ref-10)
11. DT4D has a REST API, a Java façade on that API, and a Python façade which also includes a local Job runtime. lwfm has a Python interface. [↑](#footnote-ref-11)
12. Notably Flux from Lawrence Livermore and Balsam from Argonne. [↑](#footnote-ref-12)
13. We will shamelessly use the exclamation point on this one. [↑](#footnote-ref-13)
14. Findability, accessibility, interoperability, usability. [↑](#footnote-ref-14)
15. Conceptually, a type 1 workflow system might utilize a localized MetaRepo with a lightweight implementation. [↑](#footnote-ref-15)
16. The author invites readers to point us to academic papers which describe an adoptable (i.e., readily understood and implementable) metadata model better than the one naively presented here. [↑](#footnote-ref-16)
17. Sheet, like a manifest or material property sheet. A “model card” describing the characteristics and capabilities of an ML object under management. It’s a document by another name. [↑](#footnote-ref-17)
18. Additionally, the system can consider the target to be of any type. For example, DT4D currently stores Job logs in a location separate from other data and does not notate them in the MetaRepo. This could be altered to consider the Job log just another piece of trackable data. [↑](#footnote-ref-18)
19. This is a potential complication as it pertains to target-specific metadata. Different tenants might have different metadata for the same target type. It would be up to the implementation driver to handle these variations if they exist. [↑](#footnote-ref-19)
20. For navigability, associations would ideally be bi-directional, but we also notice that with rollbacks a given version can be branched from more than one time, so while a given sheet will have only one parent, it may have many children. And (in theory) updating its list of children is an edit! (That’s not practical of course.) We delegate this problem to detailed design and specific implementation. [↑](#footnote-ref-20)
21. The non-technical reality here is its going to be very difficult if not impossible to ever change the older signatures, nor modify their behaviors. DT4D, the production system as we know it today, will ultimately be abandoned to the customer and maintained minimally on an on-demand basis. We might end up forking it, and thus permit its growth along the lines of the unifications described in this document. Or more likely, we’ll implement type 3 tooling which contains all the same elements as DT4D, subsuming it, but adding multi-Site awareness. [↑](#footnote-ref-21)
22. In fact, we prefer that type 2 systems like DT4D write their own semantic layer if there are additional concepts the enterprise chooses to elevate to top-level status, vs. treating them, as they always can, as arbitrary metadata. [↑](#footnote-ref-22)