The INTERSECT Open Federated Architecture for the Laboratory of the Future*

Christian Engelmann, Olga Kuchar, Swen Boehm, Michael J. Brim, Thomas Naughton, Suhas Somnath, Scott Atchley, Jack Lange, Ben Mintz, and Elke Arenholz

Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA

Abstract. A federated instrument-to-edge-to-center architecture is needed to autonomously collect, transfer, store, process, curate, and archive scientific data and reduce human-in-the-loop needs with (a) common interfaces to leverage community and custom software, (b) pluggability to permit adaptable solutions, reuse, and digital twins, and (c) an open standard to enable adoption by science facilities world-wide. The Self-driven Experiments for Science / Interconnected Science Ecosystem (INTERSECT) Open Architecture enables science breakthroughs using intelligent networked systems, instruments and facilities with autonomous experiments, "self-driving" laboratories, smart manufacturing and artificial intelligence (AI) driven design, discovery and evaluation. It creates an open federated architecture for the laboratory of the future using a novel approach, consisting of (1) science use case design patterns, (2) a system of systems architecture, and (3) a microservice architecture.

Keywords: software architecture; federated ecosystem; design patterns; system of systems architecture; microservice architecture

1 Introduction

The U. S. Department of Energy (DoE)'s Artificial intelligence (AI) for Science report [41] outlines the need for intelligent systems, instruments, and facilities to enable science breakthroughs with autonomous experiments, "self-driving" laboratories, smart manufacturing, and AI-driven design, discovery and evaluation. The DoE's Computational Facilities Research Workshop report [9] identifies intelligent systems/facilities as a challenge with enabling automation and eliminating human-in-the-loop needs as a cross-cutting theme.

* Research sponsored by the Laboratory Directed Research and Development Program's INTERSECT Initiative of Oak Ridge National Laboratory. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy.

Autonomous experiments, "self-driving" laboratories and smart manufacturing employ machine-in-the-loop intelligence for decision-making. Human-in-the-loop needs are reduced by an autonomous online control that collects experiment data, analyzes it, and takes appropriate operational actions to steer an ongoing or plan a next experiment. It may be assisted by an AI that is trained online and/or offline with archived data and/or with synthetic data created by a digital twin. Analysis and decision making may also rely on rule-based approaches, causal or physics-based models, and advanced statistical methods. Human interaction for experiment planning, observation and steering is performed through appropriate human-machine interfaces.

A federated hardware/software architecture for connecting instruments with edge and center computing resources is needed that autonomously collects, transfers, stores, processes, curates, and archives scientific data in common formats. It must be able to communicate with scientific instruments and computing and data resources for orchestration and control across administrative domains, and with humans for critical decisions and feedback. Standardized communication and programming interfaces are needed that leverage community and custom software for scientific instruments, automation, workflows and data transfer. Pluggability is required to permit quickly adaptable and deployable solutions, reuse of partial solutions for different use cases, and the use of digital twins, such as a virtual instrument, robot or experiment. This federated architecture needs to be an open standard to enable adoption.

This paper details the Self-driven Experiments for Science / Interconnected Science Ecosystem (INTERSECT) Open Architecture, which enables science breakthroughs using intelligent networked systems, instruments and facilities. It creates an open federated instrument-to-edge-to-center architecture for the laboratory of the future using a novel approach, consisting of (1) science use case design patterns, (2) a system of systems (SoS) architecture, and (3) a microservice architecture.

2 Related Work

There are about 300 workflow solutions for instrument science and data analysis [2]. However, only very few holistic automated solutions or research and development efforts exist. None offer a federated architecture standard.

The National Energy Research Scientific Computing Center (NERSC) Superfacility framework [33] integrates instruments with computational/data facilities for automation, such as connecting the SLAC National Accelerator Laboratory's Linac Coherent Light Source via ESnet to the Cori supercomputer for photosynthesis research [43]. The RESTful Superfacility API offers access to common supercomputer functions [32]. Oak Ridge National Laboratory (ORNL) offers federated environments for connecting instruments with computational/data resources, leveraging advances in software containerization and softwarization of hardware for processing data from ORNL's Spallation Neutron Source and High Flux Isotope Reactor [34, 1]. Data transfer and workflow tools developed at Ar-

gonne National Laboratory (ANL) and the University of Chicago, such as Globus Automate [17], Gladier [16] and Balsam [3], permit automated analysis of instrument data, such as by connecting ANL's Advanced Photon Source with the Theta supercomputer for real-time analysis [19]. Other solutions exist, such as the autonomous robot-controlled chemistry laboratory at the University of Liverpool [39], the FireCrest RESTful API at the Swiss National Supercomputing Centre [12], the design of experiments as a Cloud service by Kebotix [25], and robotic process automation using AI by UIPath [44].

Design patterns systematize software development using proven engineering paradigms and methodologies [6]. In object-oriented programming, design patterns provide methods for defining class interfaces, inheritance hierarchies and class relationships [15]. Pattern systems also exist for concurrent and networked object-oriented environments [40], resource management [26], and distributed systems [5]. Design patterns have been discovered in other domains, such as for natural language processing [42], user interface design [4], Web design [11], visualization [18], software security [10], high-performance computing (HPC) resilience [20, 21], and data processing for automation of business processes [14].

The SoS approach designs a highly complex system by decomposing it into many smaller and easier to design systems [31, 37]. The set of systems interact to provide a unique capability that none of the individual systems can accomplish on its own [22]. A SoS has five key characteristics [29]: operational independence of systems, managerial independence of systems, geographical distribution, emergent behavior, and evolutionary development. Systems are individually developed and evolved, as the architecture of a SoS is the system interfaces [38, 30]. A recent example is Defense Advanced Research Projects Agency (DARPA)'s System of Systems Integration Technology and Experimentation (SoSITE) [8] System-of-systems Technology Integration Tool Chain for Heterogeneous Electronic Systems (STITCHES) [13, 7]. The U. S. Department of Defense Architecture Framework (DoDAF) [46] is an overarching, comprehensive framework for the development of architectures from different viewpoints. It is used across the U. S. Department of Defense (DoD) for developing SoS architectures.

Microservice architectures emerged from service-oriented architectures, initially realized with Web services [47]. They have since become the modern approach to decompose complex software systems. For example, Netflix created an open source microservice architecture for their internal applications [35, 36]. Kubernetes uses a microservice architecture for automating deployment, scaling, and management of containerized applications [28]. Cray/HPE is working on a management software solution for supercomputers using microservices [24].

3 The INTERSECT Open Architecture

The INTERSECT Open Architecture approach roughly follows the DoDAF [46] with its different architectural viewpoints, such as (i) operational scenarios, (ii) composition, interconnectivity and context, (iii) services and their capabilities, (iv) policies, standards and guidance, and (v) capability. The major difference is

that the INTERSECT Open Architecture splits these views over three different parts: (1) science use case design patterns, (2) a SoS architecture, and (3) a microservice architecture.

Science use cases for autonomous experiments, "self-driving" laboratories, smart manufacturing, and AI-driven design, discovery and evaluation are described as design patterns that identify and abstract the involved components and their interactions in terms of control, work and data flow. The SoS architecture clarifies used terms, architectural elements, the interactions between them and compliance. The microservice architecture maps the patterns to the SoS architecture with loosely coupled microservices and standardized interfaces.

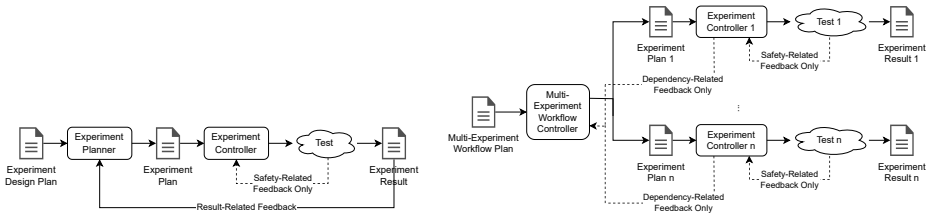
This approach permits separating (a) coarse-grain architectural decisions, such as what objective a particular "self-driving" laboratory has and how that objective is being achieved, from (b) mid-grain architectural decisions, such as which instruments, robots, networks and computing systems are part of this "self-driving" laboratory and how do they communicate with each other, and from (c) fine-grain architectural decisions, such as which particular experiment control, data transfer and compute microservices are being used and how. The science use case design patterns, SoS architecture and microservice architecture complement each other, just like the different viewpoints of the DoDAF. Additionally, the SoS architecture itself offers complementary viewpoints, such as user, data, operational, logical, physical and standards view.

3.1 Science Use Case Design Patterns

Machine-in-the-loop capabilities with connected scientific instruments, robot-controlled laboratories and edge or center computing and data resources that enable autonomous experiments, "self-driving" laboratories, smart manufacturing, and AI-driven design, discovery and evaluation is an inherent open or closed loop control problem. Therefore, the basic template for a science use case design pattern is defined in a loop control problem paradigm (Fig. 1). The abstract science use case design pattern consists of a behavior and a set of interfaces in the context of performing a single or a set of experiments in an open or closed loop control. Such an abstract definition creates universal patterns that describe solutions free of implementation details.

Design Pattern Format Design patterns for science use cases are expressed in a written form and in a highly structured format, which permits quick identification of relevant patterns given a certain problem to be solved and easy comparison of patterns regarding their applicability and capabilities. The format for describing science use case design patterns consists of individual descriptions of pattern properties, including text, diagrams, and mathematical models. It can be extended over time by adding more pattern properties and their descriptions. The current science use case design pattern format is as follows:

– **Name:** A name that distinctly identifies the pattern and permits thinking about designs in an abstract manner and communicating design choices.



(a) Design of experiments (b) Multi-experiment workflow

Fig. 1: Science use case anatomy as design of experiments (closed) loop control problem and as multi-experiment workflow (open) loop control problem

- **Problem:** A description of the problem, providing insight on when to apply the pattern. Multiple patterns may address the same problem differently.
- **Context:** The preconditions under which the pattern is relevant, including a description of the system before the pattern is applied.
- **Forces:** A description of the relevant forces/constraints, and how they interact or conflict with each other and with the intended goals and objectives.
- **Solution:** A description of the solution that defines the abstract elements that are necessary for the composition of the design solution as well as their relationships, responsibilities, and collaborations.
- **Capabilities:** The specific capabilities provided by this pattern in terms of the control problem it solves.
- **Resulting context:** A description of the post-conditions arising from the application of the pattern. There may be trade-offs between competing parameters due to the implementation of a solution using this pattern.
- **Related patterns:** The relationships between this and other patterns, which may be predecessors or successors. This pattern may complement or enhance others. There may also be dependencies between patterns.
- **Examples:** A description of one or more examples, including their properties, that illustrate the use of the pattern for solving concrete problems.
- **Known Uses:** A list of known applications of the pattern in existing systems, including any practical considerations and limitations.

Design Pattern Classification A pattern classification helps to identify groups of patterns that address similar problems in different ways or that describe solutions at different levels of granularity or from different view points. A classification scheme codifies these relationships between patterns and enables designers to better understand individual pattern capabilities and relationships. It also helps to understand how patterns rely on each other and can be composed to form a complete solution.

At this point, there are two classes of science use case design patterns (Fig. 2): (1) strategy patterns that define high-level solutions using control architecture features at a very coarse granularity, and (2) architectural patterns that define more specific solutions using hardware and software architecture features at a

finer granularity. While the architectural patterns do inherit the features of certain parent strategy patterns, they also address additional problems that are not exposed at the high abstraction level of the strategy patterns.

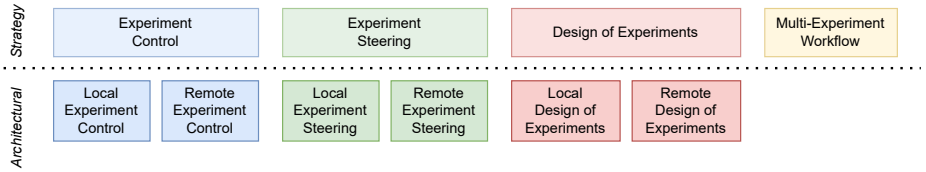


Fig. 2: Classification of the science use case design patterns

Strategy Design Patterns The science use case strategy design patterns define high-level solutions using control architecture features at a very coarse granularity. Their descriptions are deliberately abstract to enable architects to reason about the overall organization of the used techniques and their implications on the full system design. The features of these patterns and their relationships are compared in Table 1, where Fig. 1a shows the components of the Design of Experiments pattern and Fig. 1b of the Multi-Experiment Workflow pattern. The strategy patterns solve the following problems:

- **Experiment Control:** Certain predetermined actions need to be performed while running an experiment.
- **Experiment Steering:** Certain predetermined actions need to be performed while running an experiment, depending on experiment progress.
- **Design of Experiments:** Certain predetermined actions need to be performed to run a set of similar experiments with different experiment plan parameters, depending on experiment results.
- **Multi-Experiment Workflow:** Certain predetermined actions need to be performed to run a set of experiments in serial and/or parallel.

Table 1: Features and relationships of the science use case strategy patterns

Feature	**Experiment Control**	**Experiment Steering**	**Design of Experiments**	**Multi-Experiment Workflow**
# of experiments	1	1	Multiple	Multiple
Control type	Open loop	Closed loop	Closed loop	Open loop
Operation type	Automated	Autonomous	Autonomous	Automated
Extends		Experiment Control		
Uses			Experiment Control	Experiment Control
May also use or use instead			Experiment Steering	Experiment Steering, Design of Experiments

Architectural Design Patterns The science use case architectural design patterns define more specific solutions using hardware and software architecture features at a finer granularity. They offer more detailed descriptions, conveying different design choices for implementing strategy patterns and their abstract architectural features. Architectural patterns inherit the features of their parent strategy patterns. However, they also address additional problems through specific design choices that are not exposed at the high abstraction level of the

parent strategy patterns. The architectural patterns provide abstractions for different hardware and software architecture choices of implementing control and workflow, such as using experiment-local or experiment-remote computing and data resources. Fig. 3 shows the Remote Design of Experiments architectural pattern as an example. Table 2 shows the science use case architectural design patterns, their relationships to the strategy design patterns and their features.

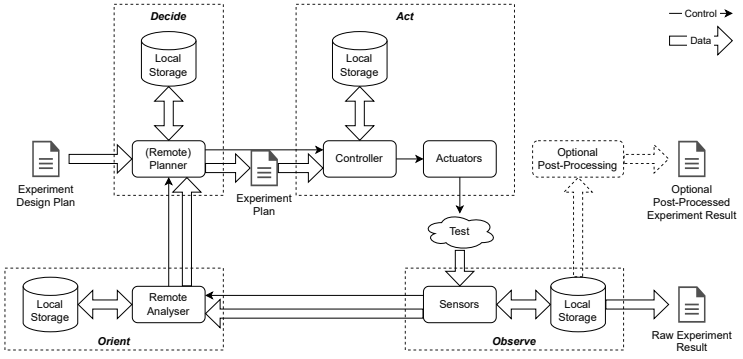


Fig. 3: Remote Design of Experiments architectural pattern

Table 2: Features and relationships of the science use case architectural patterns

Architectural Pattern	**Related Strategy Pattern**	**Remote Components**
Local Experiment Control	Experiment Control	None
Remote Experiment Control	Experiment Control	Controller
Local Experiment Steering.	Experiment Steering	None
Remote Experiment Steering	Experiment Steering	Analyzer and Controller (optional)
Local Design of Experiments	Design of Experiments	None
Remote Design of Experiments	Design of Experiments	Analyzer and Planner (optional)

3.2 System of Systems Architecture

The SoS architecture decomposes the federated hardware/software ecosystem into smaller and less complex systems and components within these systems. It permits the development of individual systems and components with clearly defined interfaces, data formats and communication protocols. This not only separates concerns and functionality for reusability, but also promotes pluggability and extensibility with uniform protocols and system/component life cycles. Instead of developing individual monolithic solutions for each science use case, the SoS architecture provides one solution that can be easily adapted to different use cases using different compositions of systems. It offers operational and managerial independence of systems and of components within systems, geographical distribution with a physically distributed and federated ecosystem, emergent behavior based on the interplay between systems and components, and evolutionary development through pluggability and extensibility.

The SoS architecture consists of various architectural views of the INTERSECT system that connects scientific instruments, robot-controlled laboratories,

computational facilities, data centers, and edge computing devices to enable autonomous experiments, "self-driving" laboratories, smart manufacturing, and AI-driven design, discovery, and evaluation. The architecture specification incorporates the IEEE 42010 Standard [23], entitled "Systems and software engineering – Architecture description." It uses the concepts of multiple, concurrent views to describe a complex system. The views are used to describe the system from the viewpoint of different stakeholders, such as end-users, developers, and project managers. There are many examples of view-based architectural descriptions, such as "the 4+1 Architectural View Model" [27], the DoDAF [46], and the UK Ministry of Defence Architecture Framework (MoDAF) [45]. We use a hybrid of the well-known 4+1 view model and the DoDAF.

User View The user view is a representation of a SoS that illustrates different human interactions with the system. It does not include interactions between systems themselves. This view highlights the human facing functionality required from the overall system. A person's view changes depending on their role, which is specific to a context. We identify the following five roles:

- **User:** This is the default role for a person interacting with a given resource in the system. Persons with this role do not own, administer, provide, or maintain the resource. Users leverage the interfaces provided by the resources in the system to compose and run scientific experiment campaigns.
- **Maintainer/Operator:** This person maintains or operates a given resource in the system. Examples include experts who configure instruments, such as a microscope, or those who maintain computational clusters.
- **Administrator:** This person performs most administrative tasks associated with a given resource, including assigning and managing maintainers/operators, granting users access to a resource, and ensuring that a resource complies with membership requirements for the system.
- **Owner:** This person is fiscally responsible for a given resource and assigns corresponding administrators.
- **Provider:** This person is a creator of software infrastructure underpinning the system or an application, such as a visualization widget or simulation module. They could alternatively be a representative of the manufacturer of resource, such as a compute cluster or an instrument.

A given person can hold multiple roles. For example, they could be the owner of a compute cluster, the administrator of three edge compute resources and a user of all other resources in the system. Roles can also be temporary and expire after a predetermined time.

The user view also provides a basic graphical representation or views of every possible interaction a person could have with the system for all relevant roles to serve as guidelines for implementors of the system. For example, the user view includes graphical representations of users logging into the system, applying for an account, viewing the catalog of resources available in the system, composing a new campaign, and monitoring or steering a running campaign. Fig. 4 shows

an example of graphical representations for a user configuring a dashboard for a running campaign.

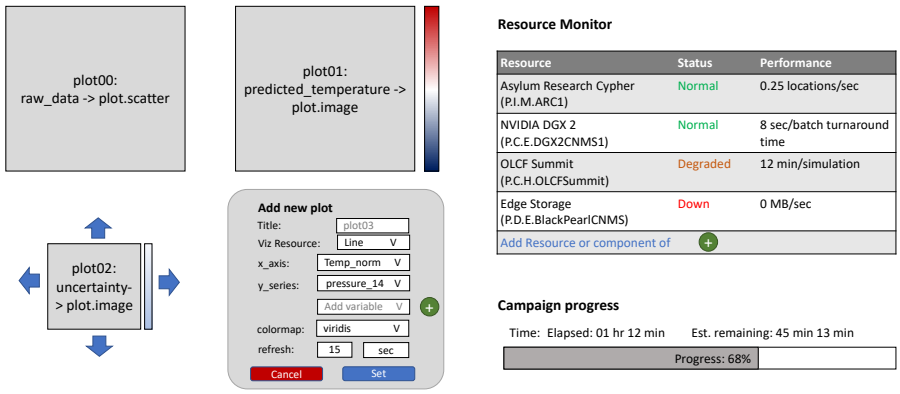


Fig. 4: Graphical representation of a user view

Logical View The logical view addresses the logical composition and interactions between the different components in the overall system. By decomposing the overall architecture into systems, subsystems, services, capabilities and activities, it simplifies the overall design and makes it easier to architect the interactions between the different components. It contains the definition of system concepts and of system options, the system resource flow requirements, capability integration planning, system integration management, and operational planning. The logical view uses the term agent to categorize any internal or external actor that is interacting with a system or subsystem, such as humans or systems (as system agents). Additionally, it describes the resource structure of the overall system and identifies the primary systems, subsystems, performers (agents) and activities (functions), and their interactions.

Fig. 5 shows an example of the system level functions and their interactions which are required for a user to configure and schedule an experiment and viewing of the experiment results after the experiment was run. Interactions are defined through interfaces, which capture the data and resource flow required to execute the different activities that comprise a capability. The data exchanged between the different system functions is captured in exchange items, which are also visible in the diagram in Fig. 5. By doing do, the logical view is tying together the data view and the operational view.

Physical View The physical view provides a mapping of the architecture onto the physical infrastructure. This view of the environment enables system designers to determine how to decompose and place the various system components onto the resources that make up the overall system. This view provides the architecture with an understanding of the attributes of the environment, and allows

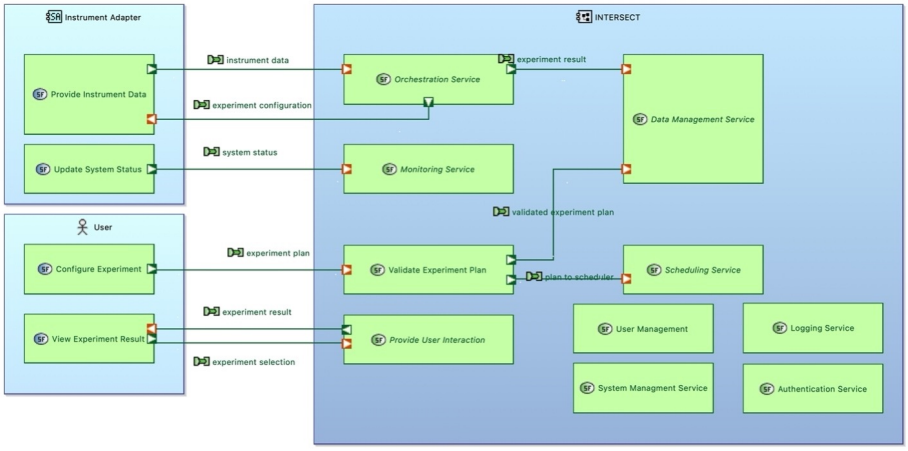


Fig. 5: Example of a high-level system diagram with services and connections between a user and an instrument

the system to configure its services based on the constraints and capabilities of the underlying system components. The elements in this view consist of physical resources that provide services to the architecture as well as the network topology that connects them together. Types of physical resources include computational resources, data storage services, data sources, and network connections.

The physical view enables the enumeration of constraints placed on the overall system. These constraints can consist of capacity constraints (e.g., available storage capacity or computational elements), network constraints (e.g., available bandwidth between elements in the architecture), policy constraints (e.g., firewall rules or access control policies), and availability constraints (e.g., ability to allocate resources within necessary time frames). These constraints limit the configuration space of the architecture, and enumerate the necessary interfaces and processes required to configure the physical infrastructure to support the operations of the overall system.

Operational View The operational view describes the tasks, activities, procedures, information exchanges/flows from the perspective of the real-world operations stakeholders, i.e., systems administrators, maintenance, facility engineers, system managers, instrument scientists. The operational view captures restrictions that may be necessary to reflect facility constraints and procedures. The intent of the view is to capture the elements needed for the operation and usage of the distributed resources in the SoS environment.

The operational view captures activities like the creation and connection of SoS services, and subsequent monitoring of services (Fig. 6), e.g., availability, health monitoring. The overall system control tree is another key component of the operational view. These control connections provide the basis for performing

operations across the distributed system. The addition and removal of services within the control channel need to be clearly defined in order to maintain a coherent control network. Additionally, the registration and coordination of services must adhere to security policies. These are the types of details captured within the operational view.

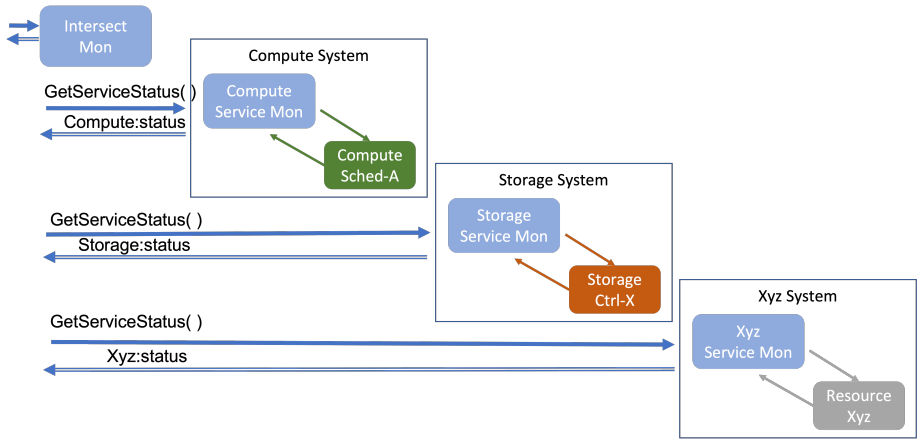


Fig. 6: Example high-level operational view diagram of service monitoring, using service adapters to connect resources to the INTERSECT environment (blue).

Data View The data view of a SoS architecture is a representation of the system from the perspective of data needs, and the data framework that needs to exist to support the INTERSECT Open Architecture. The data view is a specification for all data aspects of the system as a whole, and shall include the conceptual, logical, and physical data models. The conceptual data model provides the high-level data concepts and their relationships that are important to the SoS's operations that meet its intended purpose. Fig. 7 depicts one such conceptual concept regarding building and executing a workflow in the SoS. The logical data model bridges the conceptual and physical data models and introduces the data structure for needed components. The physical data model is the actual data schema and specifications for SoS services and applications.

Standards View The SoS architecture incorporates various versioned standards, including instrument-specific standards, messaging standards, and other external standards. The standards view provides a list of supported standards and the corresponding views or architecture elements that are impacted by each standard (see Table 3 as an example). The standards view also provides block diagrams to illustrate exactly where each standard impacts a given system.

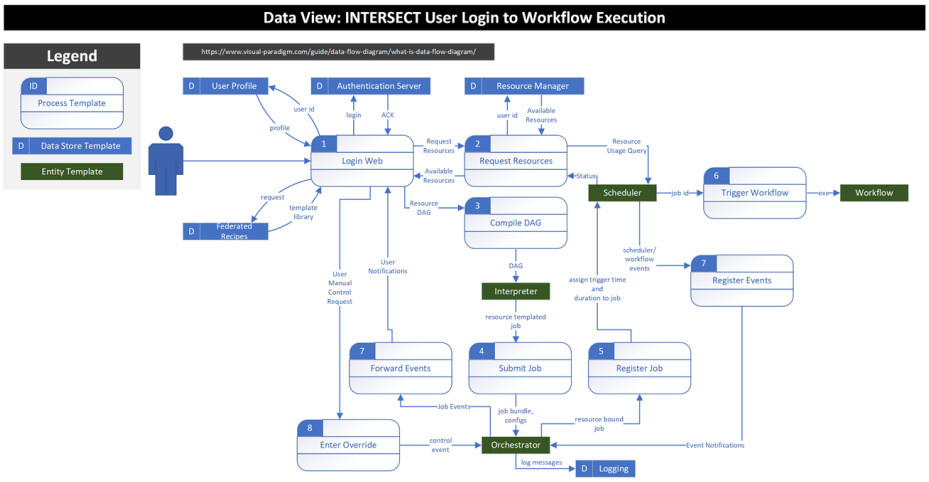


Fig. 7: Example data view of user logging in and building/executing a workflow.

Table 3: Example of messaging standards maintained in the standards view

Name	Version	Affected Views	Affected Elements
INTERSECT Core Messages	1.0	Data, Logical, Operational	Microservice Capabilities: All
Compute Allocation Capability	1.0	Data, Logical	Microservice Capabilities: Application Execution, Container Execution, Host Command Execution
Compute Queue Capability	1.0	Data, Logical	Microservice Capabilities: Compute Queue Reservation
NION Swift API	0.16.3	Logical, Operational	Systems: Electron Microscopes
Robot Operating System (ROS)	2.rolling	Logical, Operational	Systems: Additive Manufacturing

3.3 Microservice Architecture

Within the INTERSECT Open Architecture, the microservice architecture specification provides a catalog of infrastructure and experiment-specific microservices that may be useful within an interconnected science ecosystem (Fig. 8). All microservices are defined to facilitate composition within federated SoS architectures, where each subsystem corresponds to one or more coordinating microservices. INTERSECT infrastructure microservices represent common service functionality and capabilities, such as data management, computing, messaging, and workflow orchestration that are likely to be generally useful across many science ecosystems without the need for customization. Experiment-specific microservices, on the other hand, represent services whose implementation may require detailed application knowledge, such as experiment planning or steering services that require knowledge of experiment-specific control parameters and their associated constraints. The INTERSECT science use case design patterns help identify the relevant infrastructure and experiment-specific microservices for a given science ecosystem.

Each microservice provides a well-defined set of functions that is domain-scoped to ensure separation of concerns between differing microservices, avoid duplicate functionality, and encourage reuse. The supported functions are defined by the microservice contract, which describes the purpose for each service function and associated data (e.g., request parameters and response types). A microservice may have several different implementations, where each implementation provides the same contract but uses different underlying technologies or supports a particular deployment environment. Where multiple implementations exist, an application can choose the implementation most suitable for its environment or application needs.

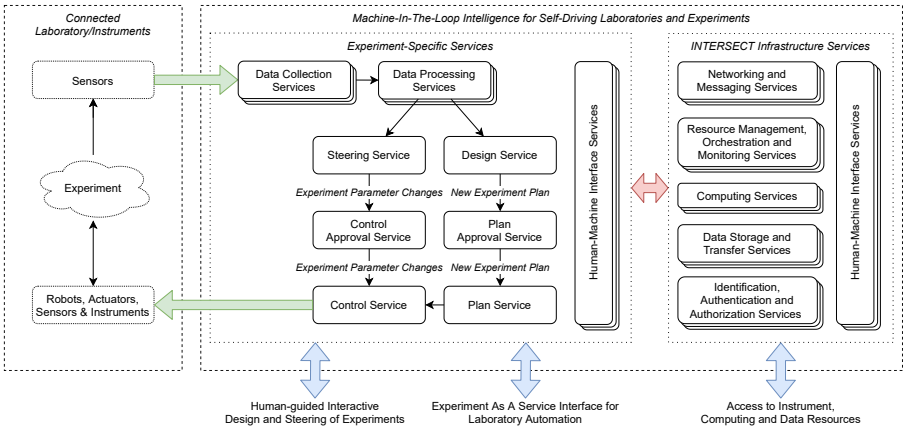


Fig. 8: Classification of INTERSECT microservices

Interaction Patterns To enable federation of microservices, it is useful to understand the types of interactions a given microservice may reasonably expect from one of its clients. Fig. 9 shows three common patterns that substantively cover the expected interactions: *Command*, *Request-Reply*, and *Asynchronous Status*. The *Command Interaction Pattern* involves the client asking the microservice to do something. The microservice typically responds immediately with a simple acknowledgement that the command has been received successfully or some error status indicating why the command was not acceptable. The *Request-Reply Interaction Pattern* has the client making a request of the microservice that includes an expected reply containing pertinent information or data related to the request. Finally, the *Asynchronous Status Interaction Pattern* represents cases where the microservice generates status or event information as a result of its internal actions or state, and sends that information to one or more of its clients.

These common interaction patterns form the basis for definition of a limited set of core message types (i.e., `Command`, `CommandResponse`, `Request`, `Reply`, `Event`, and `Status`). Each message type is easily mapped to a wide variety of messaging protocols used in both RESTful client-server communication and

asynchronous messaging via message brokers. The microservice architecture specifies a consistent yet flexible message structure across all message types that incorporates two sections: (1) a generic header containing information necessary for message routing and tracing, and (2) a data content section that is specific to the particular microservice function being exercised. For instance, the data contents of a `Command` or `Request` message may include function-specific parameters, while an `Event` message would include details of the associated event occurrence.

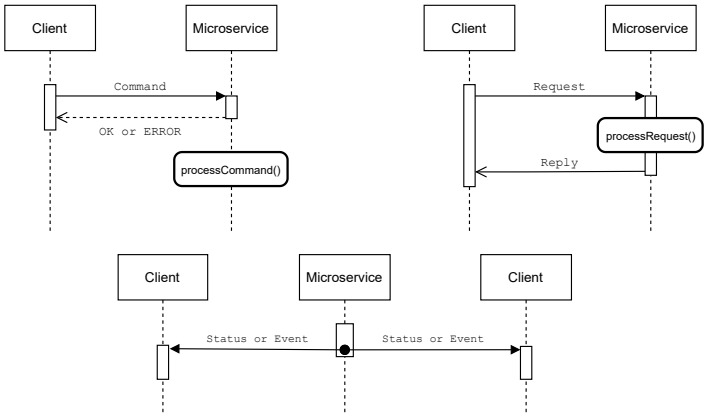


Fig. 9: Interaction patterns of INTERSECT microservices

4 Conclusion

This paper detailed the INTERSECT Open Architecture, which enables science breakthroughs using intelligent networked systems, instruments and facilities with autonomous experiments, "self-driving" laboratories, smart manufacturing and AI driven design, discovery and evaluation. The proposed open federated instrument-to-edge-to-center architecture for the laboratory of the future uses a novel approach, consisting of (1) science use case design patterns, (2) a SoS architecture, and (3) a microservice architecture.

Science use cases are described as design patterns that identify and abstract the involved components and their interactions. The SoS architecture clarifies used terms, architectural elements, the interactions between them and compliance. The microservice architecture maps the patterns to the SoS architecture with loosely coupled microservices and standardized interfaces. While there are about 300 workflow tools and only very few holistic automated scientific workflows, this is the first published work in a federated architecture standard for automated and autonomous instrument science and data analysis.

Ongoing work focuses on refining the INTERSECT Open Architecture and applying it to the following six initial science use cases at ORNL: (1) automation

for grid interconnected-laboratory emulation, (2) autonomous additive manufacturing, (3) autonomous continuous flow reactor synthesis, (4) autonomous microscopy, (5) an autonomous robotic chemistry laboratory, and (6) an ion trap quantum computing resource. Detailed architecture specification documents are currently under development for the science use case design patterns, the SoS architecture, and the microservice architecture.

In the near future, we seek to publish the specification documents and reach out to research institutions world-wide for feedback, collaboration and adoption. There is also a significant amount of work needed to develop the software environment that implements this architecture, in part using existing tools, and performing the necessary research in certain critical areas, such as (1) algorithms for autonomy and AI-driven design, discovery and evaluation, (2) codesign of "self-driving" laboratories and smart manufacturing facilities, (3) data management for scientific computing and instrument science, (4) cybersecurity, and (5) distributed operating and runtime systems as enabling technology.

References

1. Al-Najjar, A., Rao, N., Imam, N., Naughton, T., Hitefield, S., Sorrillo, L., Kohl, J., Elwasif, W., Bilheux, J.C., Bilheux, H., Boehm, S., Kincl, J.: Virtual framework for development and testing of federation software stack. In: 2021 IEEE 46th Conference on Local Computer Networks (LCN). pp. 323–326 (2021). https://doi.org/10.1109/LCN52139.2021.9524993
2. Amstutz, P., Mikheev, M., Crusoe, M.R., Tijanić, N., Lampa, S., et al.: Existing workflow systems (Jun 2022), https://s.apache.org/existing-workflow-systems
3. Balsam workflows (Jun 2022), https://www.alcf.anl.gov/support-center/theta/balsam
4. Borchers, J.: A Pattern Approach to Interaction Design. John Wiley & Sons, Inc., New York, NY, USA (2001)
5. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern-Oriented Software Architecture - Volume 4: A Pattern Language for Distributed Computing. Wiley Publishing (2007)
6. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture - Volume 1: A System of Patterns. Wiley Publishing (Aug 1996)
7. Defense Advanced Research Projects Agency, U.S. Department of Defense: Creating cross-domain kill webs in real time (Jun 2022), https://www.darpa.mil/news-events/2020-09-18a
8. Defense Advanced Research Projects Agency, U.S. Department of Defense: System of systems integration technology and experimentation (sosite) (Jun 2022), https://www.darpa.mil/program/system-of-systems-integration-technology-and-experimentation
9. DOE national laboratories' computational facilities – Research workshop report. Tech. Rep. ANL/MCS-TM-388, Argonne National Laboratory, Lemont, IL, USA (Feb 2020), https://publications.anl.gov/anlpubs/2020/02/158604.pdf
10. Dougherty, C., Sayre, K., Seacord, R., Svoboda, D., Togashi, K.: Secure design patterns. Tech. Rep. CMU/SEI-2009-TR-010, Software Engineering Institute, Carnegie Mellon Univer-

sity, Pittsburgh, PA (2009). https://doi.org/10.1184/R1/6583640.v1, https://dx.doi.org/10.1184/R1/6583640.v1
11. Duyne, D.K.V., Landay, J., Hong, J.I.: The Design of Sites: Patterns, Principles, and Processes for Crafting a Customer-Centered Web Experience. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
12. FireCrest RESTful API (Jun 2022), https://firecrest.readthedocs.io/en/latest/ index.html
13. Fortunato, E.: STITCHES - SoS technology integration tool chain for heterogeneous electronic systems (Sep 2016), https://ndiastorage.blob.core.usgovcloudapi.net/ndia/2016/systems/18869_Fortunato_SoSITE_STITCHES_Overview_Long_9Sep2016_.pdf
14. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
15. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Professional (1994)
16. Gladier experiment steering (Jun 2022), https://labs.globus.org/projects/ gladier.html
17. Globus automation services (Jun 2022), https://docs.globus.org/globus-automation-services
18. Heer, J., Agrawala, M.: Software design patterns for information visualization. IEEE Transactions on Visualization and Computer Graphics **12**(5), 853–860 (Sep 2006). https://doi.org/10.1109/TVCG.2006.178, https://dx.doi.org/10.1109/TVCG.2006.178
19. Heinonen, N.: Argonne researchers use Theta for real-time analysis of COVID-19 proteins (Jul 2020), https://www.alcf.anl.gov/news/argonne-researchers-use-theta-real-time-analysis-covid-19-proteins
20. Hukerikar, S., Engelmann, C.: Resilience design patterns: A structured approach to resilience at extreme scale. Journal of Supercomputing Frontiers and Innovations (JSFI) **4**(3), 4–42 (Oct 2017). https://doi.org/10.14529/jsfi170301, https://dx.doi.org/10.14529/jsfi170301
21. Hukerikar, S., Engelmann, C.: Resilience design patterns: A structured approach to resilience at extreme scale (version 1.2). Tech. Rep. ORNL/TM-2017/745, Oak Ridge National Laboratory, Oak Ridge, TN, USA (Aug 2017). https://doi.org/10.2172/1436045, https://dx.doi.org/10.2172/1436045
22. ISO/IEC JTC 1/SC 7 Software and systems engineering: ISO/IEC/IEEE 21839:2019 (Jul 2019), https://www.iso.org/standard/71955.html
23. ISO/IEC/IEEE: ISO/IEC/IEEE 42010 - A Conceptual Model of Architecture Description (Jul 2019), http://www.iso-architecture.org/42010/cm/
24. Kaplan, L.: Hpe cray supercomputer modernized system management and compute environment. Presentation at the 10th Accelerated Data Analytics and Computing Institute Workshop (May 2021)
25. Kebotix (Jun 2022), https://www.kebotix.com
26. Kircher, M., Jain, P.: Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management. Wiley Publishing (2004)
27. Kruchten, P.: Architectural blueprints - The "4+1" view model of software architecture. IEEE Software **12**(6), 42–50 (Nov 1995), http://www.cs.ubc.ca/ gregor/teaching/papers/4+1view-architecture.pdf
28. Kubernetes (Jun 2022), https://kubernetes.io
29. Maier, M.W.: Architecting principles for system-of-systems. Systems Engineering **1**(4), 267–284 (Nov 1998)

30. Maier, M.W., Rechtin, E.: The Art of Systems Architecting (Systems Engineering). CRC Press, Boca Raton, FL, USA (2009)
31. Manthorpe Jr., W.H.J.: The emerging joint system of systems: A systems engineering challenge and opportunity for apl. John Hopkins APL Technical Digest **17**(3), 305–313 (Jul 1996)
32. National Energy Research Scientific Computing Center (NERSC): Superfacility API (Jun 2022), https://api.nersc.gov
33. National Energy Research Scientific Computing Center (NERSC): Superfacility project (Jun 2022), https://www.nersc.gov/research-and-development/superfacility
34. Naughton, T., Hitefield, S., Sorrillo, L., Rao, N., Kohl, J., Elwasif, W., Bilheux, J.C., Bilheux, H., Boehm, S., Kincl, J., Sen, S., Imam, N.: Software framework for federated science instruments. In: Nichols, J., Verastegui, B., Maccabe, A.B., Hernandez, O., Parete-Koon, S., Ahearn, T. (eds.) Driving Scientific and Engineering Discoveries Through the Convergence of HPC, Big Data and AI. pp. 189–203. Springer International Publishing (2020)
35. Netflix: Netflix OSS (Jun 2022), https://netflix.github.io
36. Netflix: Spring Cloud Netflix (Jun 2022), https://spring.io/projects/spring-cloud-netflix
37. Pei, R.S.: System of systems integration (sosi) - a smart way of acquiring army c4i2ws systems. In: Proceedings of the Summer Computer Simulation Conference 2000. pp. 574–579 (2000)
38. Rechtin, E.: Systems Architecting: Creating & Building Complex Systems. Prentice Hall (1990)
39. Sanderson, K.: Automation: Chemistry shoots for the Moon. Nature **568**, 577–579 (Apr 2019). https://doi.org/10.1038/d41586-019-01246-y, https://www.nature.com/articles/d41586-019-01246-y
40. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects. Wiley Publishing (2000)
41. Stevens, R., Taylor, V., Nichols, J., Maccabe, A.B., Yelick, K., Brown, D.: AI for science report (Mar 2020), https://www.anl.gov/ai-for-science-report
42. Talton, J., Yang, L., Kumar, R., Lim, M., Goodman, N., Měch, R.: Learning design patterns with bayesian grammar induction. In: Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology (UIST) 2012. pp. 63–74. ACM, New York, NY, USA (2012). https://doi.org/10.1145/2380116.2380127, https://dx.doi.org/10.1145/2380116.2380127
43. Troutman, K.: Superfacility framework advances photosynthesis research (May 2019), https://www.nersc.gov/news-publications/nersc-news/science-news/2019/superfacility-framework-advances-photosynthesis-research
44. UIPath (Jun 2022), https://www.uipath.com
45. UK Ministry of Defense: MOD architecture framework (Dec 2012), https://www.gov.uk/guidance/mod-architecture-framework
46. U.S. Department of Defense: The DoDAF architecture framework version 2.02 (Aug 2010), https://dodcio.defense.gov/Library/DoD-Architecture-Framework
47. Wolff, E.: Microservices: Flexible Software Architectures. Addison-Wesley Professional (2016)