IAC-22-B6.5.7.x68666

Future-Proof Mission Control Systems: Leveraging Agnostic Design for Autonomous and Event-Driven Satellite Operations

Lucas Brémond[a*], Brunston Poon[b], Gauthier Damien[c]

[a] *Loft Orbital Solutions, 321 11th Street, San Francisco, California 94103, USA, lucas@loftorbital.com*
[b] *Loft Orbital Solutions, 321 11th Street, San Francisco, California 94103, USA, brunston@loftorbital.com*
[c] *Loft Orbital Solutions, 321 11th Street, San Francisco, California 94103, USA, gauthier@loftorbital.com*

* Corresponding Author

Abstract

The number of small satellite missions launched over the past decade has increased by more than 150%. Driven largely by the expansion of constellations in low earth orbit (LEO), this number continues to accelerate, and thus the demand for innovative mission-enabling capabilities has surged as well. In particular, the development of autonomous, event-driven mission control systems (MCS) is integral to the successful scalability of this industry. For the operation of individual satellites and constellations, the challenges of architecting an agile MCS are considerable, and the MCS of the future must be built to adapt on the fly.

To address these challenges and meet the need for reactive and autonomous satellite operations, Loft Orbital has developed Cockpit, a unique MCS founded upon the principles of agnosticism and modularity. Capable of providing a simple yet comprehensive experience for payload control and tasking onboard dedicated or rideshare configured missions, Cockpit balances the degree of control authority provided to each payload—from separate customers and users—with the feasibility, safety, and optimality of the overall mission. Structured with a clear decoupling of back-end/front-end architectures, Cockpit comprises a set of micro-services connected via APIs to enable key benefits such as extensive scalability, focused testability, and reduced dependency on the underlying computing and satellite architectures. This design enables the same system to control multiple satellites from multiple vendors, each having multiple payloads from distinct customers, relying on various ground station networks—in a way that is completely abstracted away from the user and operator.

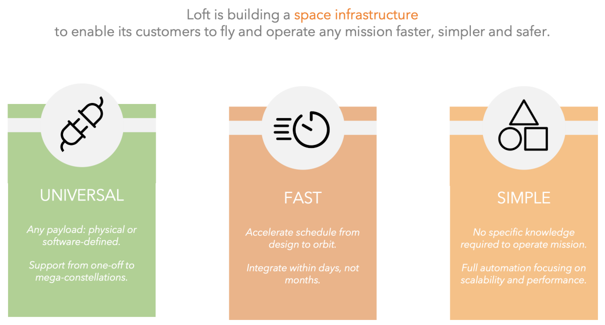
This paper presents the underlying frameworks and architectural approaches used in developing Cockpit as a future-proof MCS. It includes key innovations on partitioning the bus, ground stations, and payloads agnostically from the specific problem space, and leveraging GraphQL API technology to expose highly dynamic datasets. Together these capabilities provide a scalable, decentralized, and mission-agnostic system to better address the multi-node/constellation operations future smallsat missions will require.

Keywords: mission control software abstraction architecture constellation

1. Introduction

Loft Orbital is on a mission to build a scalable space infrastructure, enabling its customers to deploy their missions in space with minimal complexity and schedule constraints.

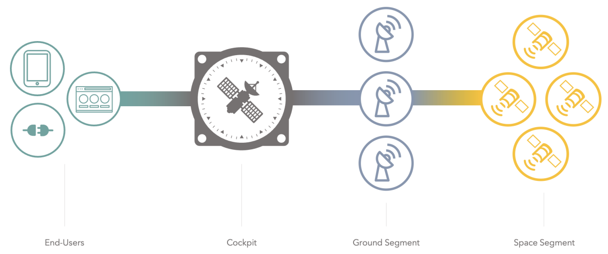
Leveraging existing cloud infrastructure paradigms, customer mission's interfaces, specifications, and concept of operations are used to configure and allocate resources of the infrastructure (which has both ground and space-based components). As such, resource utilization can be optimized—deploying multiple missions on the same satellite, for example. This optimization requires resource adaptation, abstraction, and partitioning, which is precisely what Loft's technology implements.


Loft is building a space infrastructure
to enable its customers to fly and operate any mission faster, simpler and safer.
UNIVERSAL
Any payload: physical or software-defined.
Support from one-off to mega-constellations.
FAST
Accelerate schedule from design to orbit.
Integrate within days, not months.
SIMPLE
No specific knowledge required to operate mission.
Full automation focusing on scalability and performance.

One of the main principles of Loft's approach is to leverage ground and space segment elements tapped from existing supply chains with significant flight heritage. This is done to dramatically reduce risk while providing more control over overall program schedule and quality. System diversification is key not only to avoid vendor lock-in, but also to be able to select and assemble a set of space and ground elements that will best support a given space mission.

Practically speaking, this means that Loft's space infrastructure is built upon satellite buses from multiple space systems vendors, and by leveraging ground sites from multiple ground segment networks. It is crucial that specifics of the various elements used are not exposed to the customer mission being deployed—Loft develops abstractions in the space- and ground-segments to present a universal set of space and ground interfaces to its customers, regardless of the underlying hardware stack.

This paper will focus on Cockpit, Loft's Mission Control System, which implements operational abstractions allowing the management of a heterogeneous set of satellites at scale.



2. Requesting & Scheduling

Once a customer mission has been deployed onto the infrastructure (the deployment process itself being outside the scope of this paper), *Requesting* is the primary mechanism that customers leverage to task their assets, whether *physical* (e.g., a multispectral Earth observation sensor), or *virtual* (e.g., a detection algorithm fed with data collected by other sensors).

Requests issued by customers are handled by Cockpit's Requesting system. Requests are a "declaration of intent", defined using a constraint-based formulation. These constraints are grouped into the following categories: *what*, *where*, *when*, and *how*.

The *what* defines the desired operational target:
- Is it a specific *physical* Payload?
- Is it a *virtual* one?
- Is it a collection of identical Payloads to be co-orchestrated? (this concept is called *Swarm*)

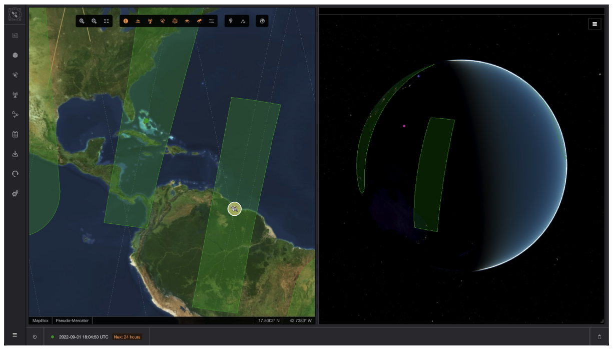
The *where* defines geospatial constraints:
- Over a certain point?
- Over a set of regions?
- On a certain phase of the orbit only?
- *etc.*

The *when* defines temporal constraints:
- As soon as possible?
- Within the next three days?
- Only on Tuesdays?
- *etc.*

The *how* places additional constraints on top:
- What range of (forecasted) cloud coverage is acceptable?
- How much off-nadir maneuvering is acceptable?
- What should be the Sun elevation during the execution?
- *etc.*

As such, when formulated in natural language, Requests are equivalent to:
- Take a picture using My Multi-Spectral Sensor over San Francisco sometime next week, only during daytime, if the expected cloud coverage is less than 20 % and not exceeding an off-nadir angle of 5 degrees.
- Collect RF samples using My SDR Swarm over California either on Tuesday or on Friday, only within an elevation angle greater than 40 degrees.
- Activate My Processing Payload whenever flying over Continental USA.
- *etc.*

Using constraints, a customer can define in precise terms what conditions are or are not acceptable for a given Request, without needing exposure to the actual intricacies of the underlying systems. This is quite important, as some of these satellites may support multiple missions, and therefore the fully consolidated onboard schedules must remain private. The formulation of these Request constraints is platform-independent, which means that the actual type of satellite, ground station, *etc.* involved for the execution of a given Request is irrelevant (although it is always exposed for awareness). These abstractions are the basis for the orchestration of Swarms, where the execution of a given Request may be fragmented and dispatched onto multiple satellites (as exemplified in the second example above), potentially each leveraging differing satellite buses without any operational nor configuration impact.



Request AOIs displayed on Cockpit UI.

2.1. Simulation

To process the high-level input collected via the Requesting mechanism into actionable Tasks that the various systems involved must execute, system modelling is used to guarantee the feasibility and safety of each operation, but also to estimate the *side-effects* that each Task would impart:

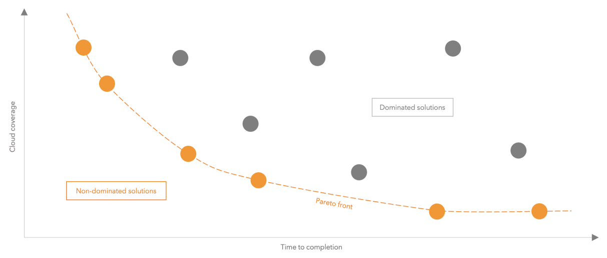
- How long would each Task take?
- What resources would be allocated?
- Are these resources exclusively accessed (e.g., ADCS) or shareable (e.g., onboard storage)?
- Will this Task affect any Tasks already scheduled?
- *etc.*

Loft is therefore developing a general-purpose Simulator which aims to model the complete spacecraft with mid-fidelity accuracy. The purpose here is to model ADCS maneuvers, data transfers, link budgets, power and thermal profiles, subsystem states, *etc.* to assess:

- The *feasibility* of the Schedule.
- The *safety* of the Schedule.
- The *optimality* of the Schedule.

Only Operational Scenarios that have been identified as *feasible* and *safe*, and compatible with the set of the customer-defined constraints, are exposed for selection and confirmation.

The Operational Scenarios which are returned to the customer can be ranked using various metrics, revealing the fact that the absolute notion of optimality is not defined. Cockpit provides several scenarios that balance Request input (temporal and/or geospatial constraints, cloud coverage, *etc.*). Offering multiple options that satisfy a Request allows a customer to choose which constraint is paramount while keeping the system generally agnostic to that selection criteria. This flexibility is critical to building an infrastructure which can scale across customers and missions with varying concepts of operation. Selecting from a list of feasible and safe Operational Scenarios prevents customers from interfering with each other (e.g., for missions which are deployed on the same satellite).



Operational Scenario ranking and selection (Pareto front visualization).

2.2. Event-Driven Requesting

In the previous section, emphasis was placed on user-triggered Requesting, in which a given customer must be intentional about defining and submitting a Request at a given moment in time, for a given purpose. Although the Requesting interface greatly simplifies the problem definition and allows an automated and optimized dispatching of how resources of various systems are allocated on a timeline (i.e., the Schedule) without exposing any platform-specifics to the end-user, there is still a human decision maker in the loop that is initiating the Requesting flow and assessing whether to proceed or not with the proposed Operational Scenario.

Event-Driven Requesting aims at streamlining the flow of Requesting, leading to an increased autonomy and faster response times. Instead of requiring user input and selection, a customer can define a Pipeline, connected to event sources, and its "activation logic".

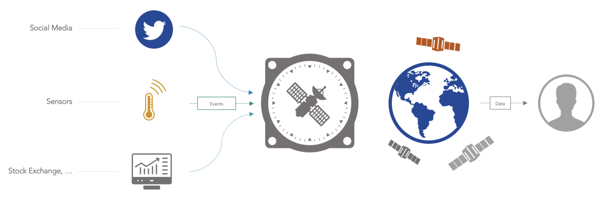


Illustration of Event-Driven Requesting flow.

To give a practical example of this concept, using this feature a customer can define a Pipeline that can derive events from a wildfire API (e.g., data provided by NOAA), cross the information with social media (e.g., Twitter) to derive a "wildfire event" of high likelihood. This logic then serves as the basis for an automated Request generation, submission, and selection process (where selection of an Operational Scenario can be defined based on parameter importance), providing an end-to-end data collection plan involving no human decision maker in the loop during its execution, reducing overall collection latency.

3. **Operations**

To truly enable end-to-end automation of the system, not only must the previously-described Task generation be automated, but so must the rest of the chain—uploading, execution, monitoring, recovery, and artifact downlink of Tasks.

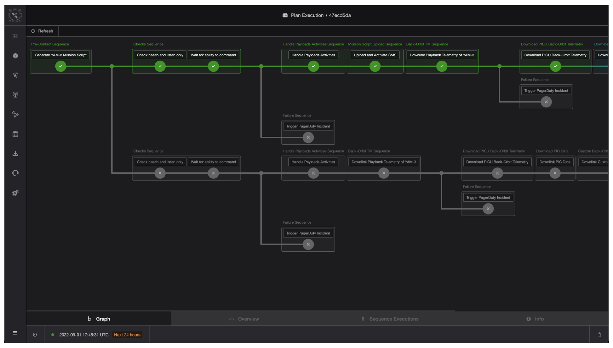
For this purpose, operational abstractions exist in Cockpit that also reduces the discrepancies between the various systems involved at operational level, promoting harmonized satellite operations paradigms and tools (even though the underlying satellite platforms are completely different, since they are coming from different vendors with no overlap in terms of protocols, command & control concepts, telemetry, *etc.*).

The way this problem has been addressed is by modelling each system (both space and ground) as a tree of Components, whose paradigms closely parallel common object-oriented programming (OOP) patterns. As a result, only a handful of interactions must be leveraged at operational level, greatly reducing the overall command & control complexity, on top of unifying the "control plane" of the various systems.

Leveraging these abstractions, an Auto-Pilot has been developed which takes care of orchestrating the various parts of the system automatically:

- Performing in-pass operations.
- Performing out-of-pass tasks.
- Raising alerts for off-nominal telemetry or system status.
- Recovering from anomalies, when necessary.

The approach taken by Loft in this instance is closely modelled after common CI/CD patterns, which describe sequences of jobs. Jobs typically consist of atomic flight Procedures and are aimed at accomplishing a certain Objective. Depending on the result of the execution of a given Job, the Auto-Pilot decides what Job to execute next. If a Job has failed, the running Sequence is cancelled and a recovery Sequence starts, so that the Objective can still be met. If no recovery Sequence is defined, the status of the Activity will be set to "Error", and an Operator will receive a notification to address the problem. By incorporating this sequential logic and conditional execution of modular and flexible components, the Auto-Pilot can manage complex operational Plans.


Plan Execution ▸ 47ecd5da
Refresh
Pre-Contact Sequence
Generate YAM-3 Mission Script
Checks Sequence
Check health and listen only
Wait for ability to command
Handle Payloads Activities Sequence
Handle Payloads Activities
Mission Script Upload Sequence
Upload and Activate SMS
Back-Orbit TM Sequence
Downlink Playback Telemetry of YAM-3
Download PICU Back-Orbit Telemetry
Download PICU Back-Orbit Telemetry
Failure Sequence
Trigger PagerDuty Incident
Failure Sequence
Trigger PagerDuty Incident
Checks Sequence
Check health and listen only
Wait for ability to command
Handle Payloads Activities Sequence
Handle Payloads Activities
Back-Orbit TM Sequence
Downlink Playback Telemetry of YAM-3
Download PICU Back-Orbit Telemetry
Download PICU Back-Orbit Telemetry
Download PIC Data
Downlink PIC Data
Failure Sequence
Trigger PagerDuty Incident
Failure Sequence
Trigger PagerDuty Incident
Graph
Overview
Sequence Executions
Info
2022-09-01 17:45:31 UTC
Next 24 hours

Auto-Pilot Sequences shown in the Cockpit Web App.

The Auto-Pilot executes exactly the same flight Procedures that have been written and validated via manual operations during the LEOP phase, reducing the overall complexity and avoiding code and logic duplication. These Procedures can be dynamic; they contain logic which adapts to the state of the system (e.g., dynamically requesting missing back-orbit telemetry). Further, these flight Procedures are tested prior to being executed by the Auto-Pilot in development/integration environments which operate under a test-like-you-fly paradigm, reducing operational risk.

As a result, the infrastructure is today fully auto-piloted, with team members being on-call in case of an off-nominal scenario. This is quite fundamental in supporting the "SatDevOps" approach coined by Loft, which does not have any Satellite Operations (SatOps) team by design; but rather dispatches SatOps knowledge and responsibilities across the various engineering teams. Engineers are trained and rotate on-call responsibility as Satellite Operators and Flight Directors, granting requisite authority to respond to anomalies. With automated telemetry monitoring and alerting, safety of the space and ground infrastructure is assured while enabling an on-call rather than on-console paradigm for operations.

Cockpit interfaces with multiple ground segment networks and is capable of automatically managing Contacts and Reservations. It abstracts away the differences between networks; all ground stations are similarly exposed to the rest of the system, even though the underlying networks may have vendor-specific interfaces.

4. **Flight Dynamics**

In Cockpit, flight dynamics are divided into four topics:

- State Estimation and Prediction.
- Guidance & Navigation.
- Constellation Geometry Management.
- Conjunction Assessment and Avoidance.

Each topic is handled by a different service.

The state estimation and prediction services are critical aspects of the operations infrastructure within Cockpit. The computation of ground station access windows and elevation profiles, aided by orbital data provided by third-party data sources (e.g., 18th SDS), allows the microservice which orchestrates the ground segment to automatically reserve passes with ground station providers, and to cancel passes when scheduled payload activities preclude the ability to take a pass. These services also provide information to the Simulator and Requester services, to enable the use of geospatial constraints in Requests. Customers can use the flight dynamics data in their own Pipelines and external systems.

The other services (for constellation geometry management and conjunction assessment/avoidance) are designed to alleviate the operational burden of maintaining and managing a constellation so that the customer can focus on payload operations. For particularly high-risk events, the same alerting system described in the Operations section which handles Operator notifications will notify a SatDevOps operator to take further action.

By building these flight dynamics services into Cockpit, it allows the entire system to use a single source of truth for synchronicity while keeping payload operations separate from satellite management.

5. Data Processing & Delivery

The various collected artifacts are automatically downlinked from the spacecraft, and need to go through an automated process to:

- Detect missing/corrupted fragments (and potentially re-downlink them)
- Isolate the various data streams
- Package multiple files into consolidated deliverables

Once the deliverables have been created, they are automatically delivered to customer-specified endpoints, closing the Requesting loop and guaranteeing an end-to-end traceability from Request to Deliverable.

The Data Processing and Delivery services are designed so that users of their interfaces do not need to consider the space segment when developing software that relies on the delivery of those artifacts. The data deliveries can be treated like any ground-based data pipeline, using common interfaces. The re-downlink and consolidation of artifacts allows those users to achieve their objectives without requiring familiarity with spacecraft operations, the delivery mechanism of the ground station providers, or the transmission methods of the satellites.

Data Processing and Delivery services are also responsible for forwarding relevant telemetry to customers—payload telemetry, bus pointing vectors, and more; these processes are similarly designed to be as mission-agnostic as possible to enable the scalability of these systems to constellation- and infrastructure-scale.

The delivery path from Cockpit is cloud-agnostic, again allowing for flexibility in supporting customers with different technology stacks and needs.

6. Interfaces

Cockpit exposes two kinds of interfaces: programmatic interfaces, mostly used for scripted access, machine-to-machine interfacing; and a graphical user interface, to support human-computer interaction.

The system is designed to be API-centric: the API is the first-class citizen, and the Web App only acts as a graphical façade which leverages API-exposed features. It also means that every interaction of a user with the graphical interface can be scripted and automated.

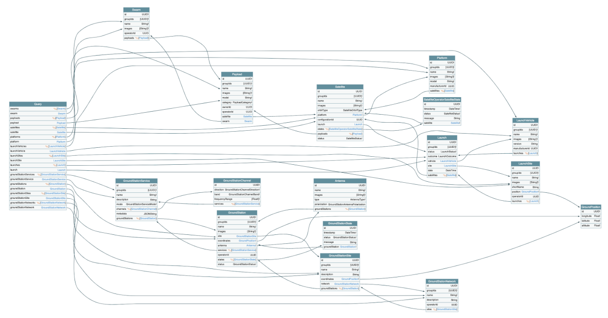
6.1. Programmatic Interface

Cockpit exposes a Web API following the GraphQL standard [5], which sits on top of HTTP(S).

A GraphQL query to the Cockpit API.

In Cockpit, models are heavily interconnected, hence using a graph data structure is a logical choice. We selected GraphQL as our API specification and query language. It simplifies exploration of complex datasets and makes the Web App (and customer-facing API accesses) more efficient, as GraphQL lets it query for exactly the data it needs in a single query.
Cockpit also leverages the Schema Federation feature of GraphQL, which consolidates the APIs of its various sub-services into a global (“federated”) API, exposing a

single endpoint to the user. Consequently, both internal and external consumers of these APIs are relieved from being tightly coupled to the internal microservice architecture and its access patterns.



A subset of data model relationships in Cockpit.

Additionally, a Python software development kit (SDK) has been developed, which makes scripted interactions with Cockpit's objects (via the API) easy using native Python.

```
from cockpit.asset import Satellite


yam_3 = Satellite.get(name=”YAM-3”)

simera = yam_3.payloads.get(name=”Simera”)
```

6.2. Graphical Interface

The Cockpit Web App is useful from both an operational and visibility perspective. It can be used to manually perform many of the same operations which are available via the API, like presenting a unified view of payload activities, generating Requests, selecting Operational Scenarios, viewing Flight Dynamics data, monitoring individual satellite Schedules, or checking Ground Station status (among many other capabilities).

Various Cockpit Web App displays.

The Web App allows operations at multiple levels of abstraction—it is designed both for customers (to manage Requests) and for SatDevOps Users (to monitor low-level connections and status). Further, visual telemetry monitoring is available for operational awareness and critical manual operations.

In combination with the authorization/authentication model of Cockpit, the Web App allows for secure operation of the satellite without necessarily requiring operations to occur at a specific physical mission control center. This creates operational flexibility and is also a key part of Loft's SatDevOps approach to satellite operations.

7. Security

Cockpit has an Identity & Access Management (IAM) system, providing authentication and authorization mechanisms. It is designed to be secure while allowing operation and monitoring of payload and satellite activities to be location independent. It enables secure access to payload telemetry, data, and operation flows. Access to the API is also controlled by this system through access delegation.

7.1. Authentication

For authentication, a JWT-based mechanism [1] is used. Tokens are obtained via an OAuth2 [2] mechanism, powered by third-party platforms (Google Firebase and Okta are currently supported). This allows interoperability with existing identity providers, reducing the attack surface.

7.2. Authorization

Each User can be given different Roles, each Role providing various access privileges into the system. In this manner, fine-grained control and high-level requesting can co-exist in the same MCS platform.

In a typical scenario, a Satellite Operator on shift may be (temporarily) granted the Satellite Admin Role (with command & control authority), while a Satellite Operator in training may only be given the Satellite Operation Viewer Role, only allowing read-only access into the state of the system, without any possibility of altering it. Similarly, customers have a specific set of associated Roles which are granting them Requesting-level access without allowing any infrastructure-level management authority.

Cockpit provides the concept of Teams, complementary to User-level granularity, allowing group management of Users. For example, a User on the Satellite Operator Admin Team is granted the associated Role.

The intention here is to implement a Zero Trust [3] security model where no distinction is being made between "insiders" and "outsiders", therefore greatly reducing the attack surface of the system, and also better partitioning duties and escalation paths. The Roles, Teams, and Users grant tiered access to the API depending on whether they need access to the full low-level operational abstractions of Cockpit or only limited access to high-level Requesting and Data Delivery flows.

8. **Infrastructure**

Cockpit was designed to be cloud-first. It leverages the scalability of modern cloud architectures by running its services in Kubernetes [4]. Cockpit extends its agnosticism to the cloud infrastructure on which it is deployed, enabling its deployment (supporting customer requirements) and connection (via its Data Delivery systems) to multiple cloud providers. It lowers operating costs (and carbon emissions) by starting containers on-demand as they are needed.

Containerizing its services enables long-term scalability and modularity. It allows different services to be written in different languages, deployed on different hardware platforms, and takes advantage of existing software tooling to improve availability and reliability.

Loft's ground infrastructure also extends to the ground station networks supported by Cockpit. Services within our infrastructure maintain a network backbone that allows interoperability between cloud providers and ground station providers, in a manner that is transparent to both Cockpit and any User.

Loft also contributes to the open-source infrastructure tooling community with Cuebe [6], a Kubernetes release manager powered by CUE (open-source data validation language and inference engine).

9. **Conclusion**

The concepts of operation proliferating in the new space age: operating a heterogenous satellite constellation, ridesharing multiple payloads on a single satellite, flying satellites without dedicated operators, rapid payload development and launch timelines—all pose considerable challenges, but also offer many advantages in the rapidly changing space environment. By creating a mission control system that is architected to be modular and system-agnostic, Cockpit enables the same velocity and agility to be applied to the operational aspects (payload, satellite, ground segment, or data management) of any space mission.

Cockpit is more than a mission control system—it is a mission enablement system. Decoupling concepts which were historically intrinsically coupled—separating payload operations from satellite bus operations, maintaining unified interfaces for both low-level operations and high-level user requests, keeping operational concepts (requests, data deliveries, *etc.*) untethered to the satellite platforms or ground station networks they rely on—enables the type of scalable, mission- and platform- agnostic operations that future multi-node smallsat missions require.

The use-case for such an MCS is analogous to the transformative impact that cloud services have had on software development. The aerospace industry (and other engineering disciplines more broadly) benefits from the commoditization of hardware and software; it encourages competition, drives down cost, and sparks the creation of better products. It is that sea change which creates the need for a similarly flexible, capable, and innovative mission control system that Cockpit fulfils.

Fundamentally, Cockpit is what the foundations of space infrastructure as a service looks like. As more organizations consider the total lifetime cost of operating their own space assets or incorporating space into their portfolio, they must consider the implications of the central role that their mission control system plays in operations, data management, expansion opportunities, and, ultimately, mission success.

References

[1] JWT
https://datatracker.ietf.org/doc/html/rfc7519

[2] OAuth2
https://datatracker.ietf.org/doc/html/rfc6749

[3] Zero Trust
https://ldapwiki.com/wiki/Zero%20Trust

[4] Kubernetes
https://kubernetes.io/

[5] GraphQL
https://graphql.org/

[6] Cuebe
https://github.com/loft-orbital/cuebe