17th International Conference on Space Operations, Dubai, United Arab Emirates, 6 - 10 March 2023.


SpaceOps-2023, ID #233

# Lessons Learned from Automating Heterogeneous Spacecraft Systems

**Lucas Brémond[a]*, Etienne Vincent[b], Gauthier Damien[c], Caleb MacLachlan[d], Brunston Poon[e]**

[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, etienne@loftorbital.com*
[c] *Loft Orbital Solutions, 321 11th Street, San Francisco, California 94103, USA, gauthier@loftorbital.com*
[d] *Loft Orbital Solutions, 321 11th Street, San Francisco, California 94103, USA, caleb@loftorbital.com*
[e] *Loft Orbital Solutions, 321 11th Street, San Francisco, California 94103, USA, brunston@loftorbital.com*
* Corresponding Author

## Abstract

With the popularization of smallsat technology, an increasing number of "off-the-shelf" spacecraft platforms and ground segment networks from a variety of vendors are now available on the market. This opens up the possibility of leveraging systems of different kinds to support a wide range of applications and mission profiles, while being jointly operated "as a service" by a central coordination entity.

The lack of standardization of TT&C concepts and protocols remains, however, one of the (many) challenging aspects of operating at scale a set of heterogeneous spacecraft. Treating this as an overall infrastructure management problem, Loft has now integrated four kinds of satellite buses and four ground segment networks from various US and EU suppliers, operated using Cockpit, its system-agnostic and fully automated Mission Control System. What became apparent during the interfacing process was that each satellite platform is not only exposing varying messaging protocols, but also radically different higher-level command and control concepts. Although some systems do implement a subset of the CCSDS/ECSS standards – in every occurrence not implementing these protocols "to the letter" and therefore requiring custom deviations – some other microsat platforms expose ad-hoc messaging, exchange and interaction patterns. Another typical difference is on the temporality of the exchange: whether it is a request/response "synchronous" exchange flow, or "asynchronous" patterns with commanding acknowledgement and side-effects to be inferred from periodic and aggregated telemetry streams. Additionally, the notions of onboard value, file transfer or event have been consistently found to be platform dependent. On the ground segment side, each network vendor also exposes its own set of APIs and concepts for ground resource management, control and monitoring.

In order to operate and automate Loft's heterogeneous constellation, abstractions have been designed and implemented in Cockpit, exposing a unified and simplified set of command and control concepts to operators and to the auto-pilot system. This paper exposes and discusses these abstractions, the underlying generalization vs. specialization tradeoffs that are derived from them, and how Cockpit's unified command and control facade is serving as a foundational layer to support Loft's objective of managing its constellation in a fully automated manner.

**Keywords:** command & control, telemetry, auto-pilot, constellation, heterogeneous.

**Acronyms/Abbreviations**

| | | | |
|---|---|---|---|
| API | Application Programming Interface | OOP | Object-Oriented Programming |
| CI/CD | Continuous Integration / Continuous Deployment | MCS | Mission Control System |
| GUI | Graphical User Interface | YAM | Yet Another Mission |
| LEO | Low-Earth Orbit | | |

## 1. Introduction

Loft Orbital's core business is to offer infrastructure services to its customers: essentially allowing them to deploy and operate their space missions – as a service – on a pre-existing set of satellites (e.g., uplinking and executing their proprietary software on compute resources, and leveraging payloads already in orbit) or on an "about-to-be-launched" set of satellites (e.g., embarking their remote sensing, RF or compute payload on one of our 9available slots).

One of the key elements of Loft's approach is to leverage existing satellite buses, with extensive flight heritage, in order to benefit from a massive risk reduction. Similarly, Loft leases ground segment resources from already established network providers, which have specialized themselves into operating ground sites at scale. This, again, is done to break the "chicken and egg" problem of space system maturation and to let Loft focus on developing its core technologies. In addition, leveraging multiple satellite buses, and ground segment vendors, boosts the ability to pick and choose the right set of assets tailored for the right mission. For example, some of the missions deployed may be power-hungry but not necessarily require a lot of ADCS agility or pointing precision; or the opposite may occur where the mission requires fine pointing accuracy but does not consume a lot of mass. Being able to leverage multiple buses enables Loft to select the right tool for the job. And similarly, to select the right ground sites that are supporting the overall mission CONOPS best.

Another aspect of Loft's strategy is that missions are usually flown in a rideshare configuration, meaning that multiple users may end up leveraging common space and ground resources, without the need for direct coordination.

In a nutshell, Loft is building, operating and orchestrating a constellation of heterogeneous spacecraft, accessed via a heterogeneous ground segment, and supporting the physical and virtual missions of its customers in a rideshare configuration. Sourcing concepts from cloud computing platforms, we treat each spacecraft as a "node", part of a "cluster". These nodes are named YAMs – for *Yet Another Mission*.

We have identified that each mission relying on a unique combination of satellite(s) / ground site(s), operating them at scale cannot rely on a satellite operations team, as the sheer combinatorial complexity would largely surpass the capabilities of individual operators, or would require scaling the team to unrealistic proportions. As a result, Loft's approach is to go automation-first and be fully operator-less for nominal operations, and to rely on the SatDevOps approach [1] [2] whenever manual interventions are required.

To this end, Loft has developed its own Mission Control System (MCS) – Cockpit – built around agnosticism, flexibility and scalability principles [3]. Cockpit has been successfully interfaced with four bus providers, and four ground segment providers so far, from the US and from Europe.

## 2. Protocols

Throughout our interfacing work became apparent that the level of protocol standardization across the space industry remains relatively limited. Some bus manufacturers either rely on ad-hoc protocols, or loosely interpret existing ones (like CCSDS or ECSS), especially when it comes to the network layer and above. As for ground segment networks, there is no standardization at all, as this is an emerging market with only a handful of suppliers. The modems are often sourced from established vendors, and therefore commonalities can emerge, but other interfaces (reservation, ephemeris, availability, metrics) are usually vendor-specific.

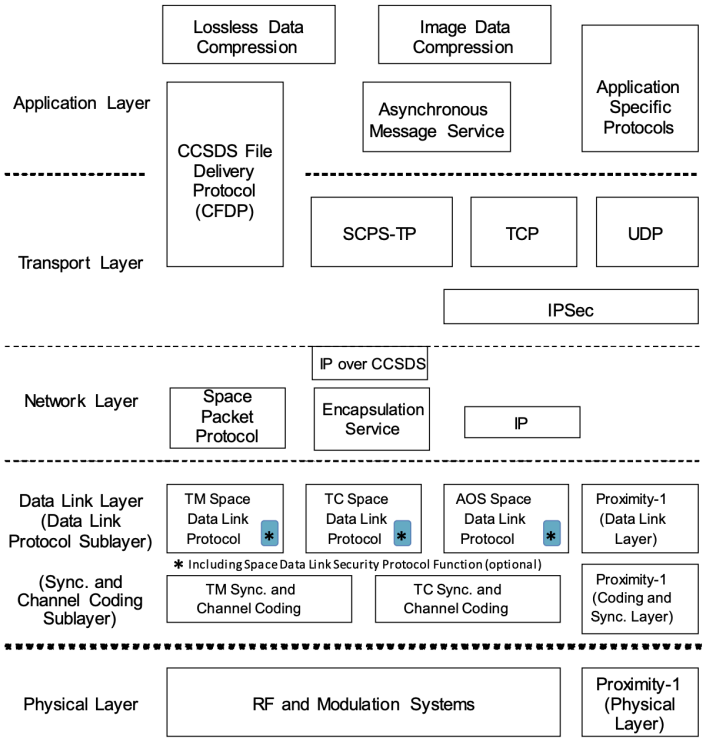



*Figure 1: Space Communications Protocols Reference Model [4].*

As mentioned previously, Cockpit has been interfaced with four satellite buses, including systems from Airbus OneWeb, LeoStella, Blue Canyon Technologies, but also with a cubesat provider. Some of these systems are following ECSS PUS [5] "to the letter", some others are sourcing concepts from CCSDS or CSP [6] without adhering to the spec in a fully compliant manner, while some others have purely and simply designed their own protocols. Not only this makes the interfacing work challenging, as all these protocols have to be somehow implemented and supported by Cockpit, but it also makes mapping these to operational abstractions a challenging process. Indeed, protocols do not only define how messages should be structured (e.g., a telecommand format, or a telemetry packet), but also define the "interaction lifecycle" that each system expects to be operated upon.

As an illustration: some systems expose a request/response interface, where each ground/space interaction is made of a succession of acknowledgements. Other systems are broadcast-based, with decorrelated telecommand and telemetry streams, necessitating the ground system to "infer" side-effects from telemetry.

File transfer protocols are also varying quite drastically across vendors, as we've seen CFDP [7] implementations ranging from "on spec" to "off spec", or even non-space-specific file transfer protocols being used (ad-hoc, or even UDPcast [8]).

On top of differences on the command, control and telemetry aspects, security varies also quite significantly across vendors, some implementing security at packet level, at frame level or else, using a variety of encryption and authentication schemes, key rotation policies, replay attack prevention rules, etc.

## 3. Abstractions

As with any form of abstraction and "commonization", some decisions have to be made, which will inevitably reduce the spectrum of available interactions as a means to simplify and streamline them. What this paper introduces is Loft's "opinionation" on the matter, and the rationale behind why some decisions have been made. There are of course other systems of abstractions that can exist, with their own advantages and disadvantages.

Cockpit's approach is to use a layered model, with each layer specializing in providing a specific set of functions. Another important aspect that was taken into consideration during the design process was the distinction between code and configuration, between data structure and logic. These two aspects can often be seen as the two sides of the same coin. In Cockpit, we use a Resolver concept, which allows developers to implement behavior using code, keeping the configuration complexity to a minimum while retaining the ability to operate any system. We believe this is a nice trade-off between committing to a future-proof structure and data model (which can scale to a number of systems we have no awareness of, at the present time) while still providing a skeleton in order to "compare apples to apples".

Another point to mention is the fact that these abstractions are used to operate *Systems*, which can be:

- Satellite buses
- Satellite sub-components (e.g., payload computers)
- Ground stations (e.g., SDRs)
- EGSE components (e.g., power supplies, spectrum analyzers, …)

At the bottom of the stack is the Connection layer, whose sole purpose is to abstract away the various ways that are necessary to interface with other systems: TCP, UDP, NNG [9], … One of the specializations that has been made at this stage is to consider all these interfaces to be message-oriented (including TCP, via the addition of a framing system). This is because these abstractions are made to support a command and control interface, which by nature is about exchanging discrete "messages". Stream interfaces are also very important, but mostly exist to deal with payload data, which is not covered in this paper. In essence, the Connection layer allows to send and receive messages across a large number of interfaces and framing systems. Configuration at this level is quite minimal, mostly consisting of protocol type, host and port and optionally of a framing type. In addition, and optionally, a Resolver can be used to extend the behavior of the receiving or sending end, whenever more advanced message manipulation is required.

The next layer is called the Transaction layer, and is specialized into defining the *structure* and *routing* of the messages: telecommand, telemetry items, header, data fields, … Messages incoming from the Connection layer need to be parsed and dissected prior to being handled by the upper layers. And when dealing with telecommands, structured datasets need to be serialized according to serialization/packetization rules, which are defined here as well. Finally, this layer is named Transaction because it handles the association between cause and effect, request and response, or transactions. The configuration of this layer is quite straightforward: it is all about message structures, serialization/deserialization rules. The Transactions themselves are only a time-bounded execution context for Resolver-backed logic to occur, each Resolver being instantiated following rules dictated by upper layers. We believe that the dichotomy between configuration and code here is optimal, as it allows developer to reuse functionalities that most implementations will require (data structure parsing/construction, …) and provides a skeleton within which Transaction can occur, while retaining a significant amount of flexibility thanks to the "write your code here" approach promoted by the Resolver paradigm.

On top of the Transaction layer sits the Interaction layer, which is about defining the *behavior* of the various interactions exposed by Cockpit. An important trade-off at this level exists between operational fitness vs operational simplicity. Trivially speaking, do we want to expose a control panel with hundreds of different concepts, all different from one system to another, or do we greatly reduce complexity and cross-system differentiation by

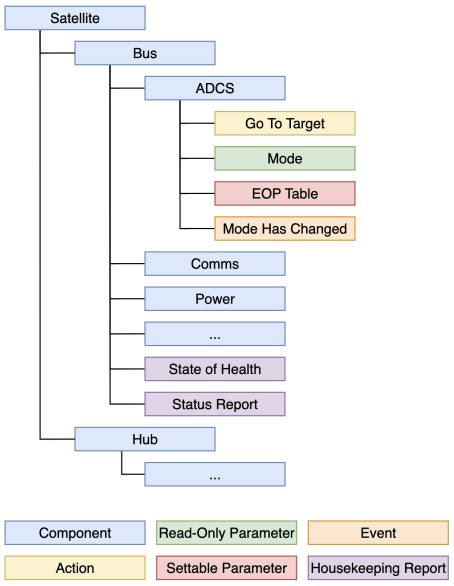
remapping all these to a handful of concepts, with the caveat that sometimes the remapping is quite artificial or not as optimal as the underlying concept? Loft has elected the second approach: we traded exposing the system interfaces in all their subtleties, as designed by the various vendors, in favor of a general facade that may provide only 90% of "optimality" but which greatly simplifies operator's onboarding but also, and mostly, procedure scripting.

In order to define behavior, we first need to define a model onto which various behaviors can be mapped. The top level entrypoint concept here is called System. A System represents the entity which Cockpit *interacts* with. What is important to note is that Systems may evolve over time, for example whenever their onboard software gets updated (which is something Loft does routinely, especially post launch), which may expose new interfaces and new behaviors. In order to track the evolution of the model associated with the System, Cockpit keeps a record of the topology of the model and never overwrites them: this is called a Configuration. This also provides the benefit of tracking the evolution of a given value onboard across various Configurations, keeping the underlying context non-ambiguous.

Another trait of Cockpit is in its immutability principle: data is never modified but only added and linked appropriately. This allows for backwards tracing: in case forensics have to be performed on the data, it is available and context can be derived clearly without ambiguity on whether "the environment was the same back then".

For a given Configuration, we have elected to reduce all system interfaces to the following constructs, some being derived from common Object-Oriented Programming (OOP) principles:

- Component (OOP equivalent: class): a part of the model, divided according to semantics designed to map onto operational decisions (and not necessarily 1-1 mapped to the associated system ICDs). A Component may contain other interactions, or other Components.
- Parameter (OOP equivalent: property): represents an onboard value, optionally provides constructs to fetch / update these values, transforming the data from a raw representation to a calibrated one with units.
- Action (OOP equivalent: method): represents an onboard function that can be executed, with or without arguments, either synchronously or asynchronously, within a given context.
- Event (OOP equivalent: signal): represents a message that can be sent asynchronously by a Component, either to provide an awareness of an ongoing process, or to convey warnings and errors.
- Housekeeping Report: represents an asynchronous message containing a set of Parameter States (i.e., values) received asynchronously from the System.
- File: represents an unstructured (from Cockpit's point of view), but bounded, dataset that can be uplinked to, or downlink from, a given Component (if the Component exposes such ability).




Interactions are under the hood powered by Resolvers, executed at Transaction level. One useful analogy we use here is the "tape player": interactions are selecting/building the tape, given to the Transaction layer to "play" (without the Transaction layer having to understand how a given tape was constructed in the first place, or what the

"song" is about). Trivially speaking, we call these various interactions the "keys of the piano": using each independently is akin to pressing a key: on its own it's not very useful but does produce a given side-effect.

Parameter values (post decommutation, parsing and calibration), are streamed in real-time to a time-series database for archival and retrieval purposes. Visualizing the evolution of values over time is a very common task which has benefited from a lot of great development lately, this is why Cockpit is leveraging Grafana [6], an open-source platform for creating, analyzing, and visualizing metrics and logs. It allows users to create and share dynamic and interactive dashboards that display real-time data from various sources. Grafana can be used for monitoring, troubleshooting, and alerting on various metrics. It is widely used in the tech industry, particularly in the field of DevOps and observability, and is supported by major cloud providers.


*Figure 2: One of the Grafana dashboards of Cockpit, displaying a subset of spacecraft telemetry.*

The last and top-most layer is called the Operation layer. This is where the various "piano keys" are played together and orchestrated to form a melody, "to do something useful". This is possible because all System interfaces have been reduced and mapped to piano keys, all sharing a similar signature and comparable semantics (despite the fact that each real-life System may actually understand completely different protocols and not even "know" what interactions are!). These are purely Cockpit-mapped constructs which do not necessarily make any sense from a system perspective (and this is the purpose behind using abstractions).

The operation layer is modeled after common Continuous Integration / Continuous Deployment (CI/CD) concepts, in order to further formalize the sequencing of the various interactions. This means defining Jobs that can run concurrently within the scope of Stages, themselves executed sequentially within Sequences that are defining the "branching" structure of the logic flow. At the top sits the Plan, which consolidates all these concepts within a given configuration umbrella.

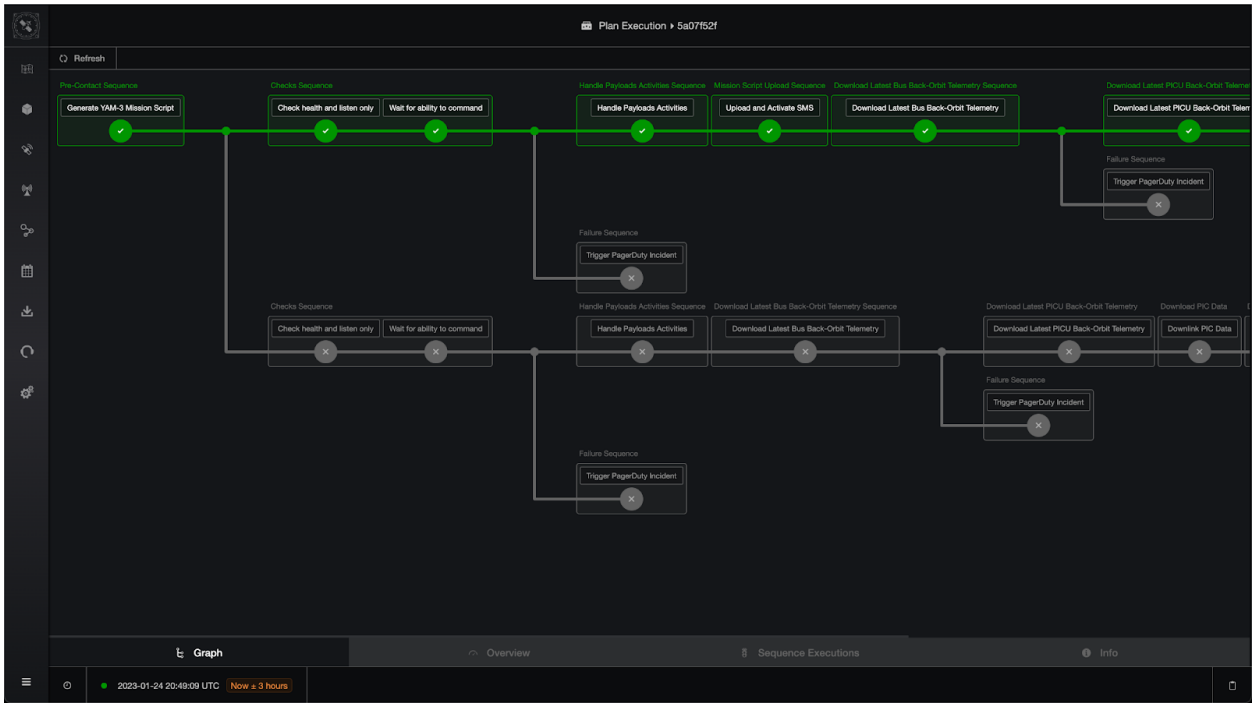



*Figure 3: Auto-Pilot Plan Execution viewer.*

This is also at the operation layer that leveraging Kubernetes [7] as the underlying orchestration system brings a lot of value. Kubernetes is an open-source platform for automating the deployment, scaling, and management of containerized applications. It provides a way to manage and orchestrate containers across multiple hosts, making it easier to deploy and scale applications in a cloud-native environment. These operational Jobs are spawned in and out of existence in a dynamical fashion, executed by ephemeral Kubernetes pods, allowing to operate multiple systems concurrently without any cross-talk nor dependency conflicts. The number of systems that can be operated concurrently using this approach is mostly capped by the resources of the cluster itself (which can scale elastically to great proportions, especially when hosted in the cloud) and by the ability of the databases to cope with the data flow (Cockpit leverages different kind of databases optimized for various tasks, e.g. relational and time-series).

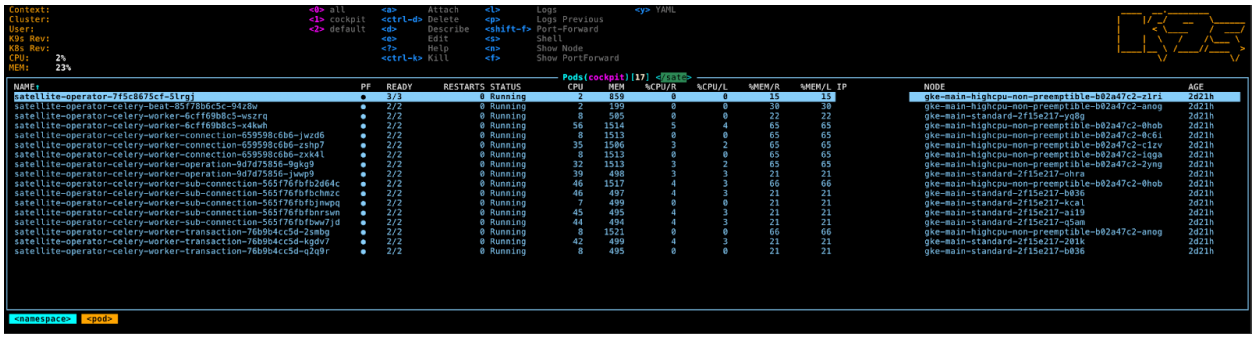



*Figure 4: Command & control pods orchestrated by Kubernetes.*

## 4. Use Cases

So far, Cockpit has been used operationally to support three programs, YAM-2, YAM-3 and YAM-5, handling varying satellite platforms and configurations, is de-facto compatible with YAM-4, YAM-6 and YAM-7, and is being actively developed to extend its capabilities to support our ongoing activities from YAM-8 to YAM-19.

YAM-2 was the first program that Cockpit had to support, leveraging a bus provided by Blue Canyon Technologies (BCT) and a ground segment network provided by KSAT. While being in active and initial development, the YAM-3 program came to life, this time leveraging a bus from LeoStella and relying on a radically different CONOPS. Both satellites ended up being launched on the same Transporter-2 mission (SpaceX), which meant that our MCS had to perform its operational debut by handling two drastically different spacecraft, at the same time, operated jointly.

Fortunately, first contact was obtained on the first try on both spacecraft, which meant that Cockpit gained flight heritage on both platforms right away. Commissioning activities continued the following weeks, although at that time most of the work was performed using manual operations while space and ground system maturity was assessed and perfected. Over the course of the weeks following launch, more and more automations were turned on, with the culmination being the auto-pilot, flying the spacecraft automatically and autonomously, changing the SatOps dynamics from on-shift to on-call.

This year, the YAM-5 spacecraft has been successfully launched, and this time benefited from the automations that were put in place, allowing the auto-pilot to be engaged only two days after launch!

**5. Lessons Learned**

Deriving generalities and abstractions from a small set of systems usually leads to overfitting if not careful, this is why it took Loft around 9 months to perfect its abstractions during the work to interface Cockpit with its first space and ground systems (within the YAM-2 program).

However, this work paid off as subsequent systems started to be integrated into Cockpit: adding a second satellite bus to the mix took about 2 weeks, and supporting a third one took about 2 days for the initial compatibility test to be performed. Although these timelines are indicative, as the actual time to integrate a whole new system depends on the system complexity itself, but also on the overall experience of the staff and expected integration coverage, they still indicate that the abstractions put in place are paying off as the speed to integrate gets reduced in a compounding manner every time a new system is mapped to Cockpit. Which is ultimately the goal of the company: to significantly reduce the “time to orbit” while streamlining the overall process.

**5. Conclusion**

Cockpit is still a relatively young system, maintained by a small team, which is expected to mature over the years. For example, the underlying databases could be better distributed to enable a massive increase in operational concurrency. Also, the overall system performance can still be improved, by relying on more optimized services developed using CPU and memory efficient programming languages.

The version currently used in production has been developed with operational breadth in mind: it was expected, and actually encouraged, to go sub-optimal at service level, in order to reach automation on all fronts as fast as possible, with the expense of sometimes yet-to-be-optimized areas. This approach is embraced by Loft, as automating globally liberates development resources than can then be allocated to address the lower-level performance tuning and improvements, rather than being stuck in an operational on-shift mode long term, or necessitating a dedicated SatOps team that would eventually become obsolete once automations are fully in place.

**References**

[1] B. Poon, L. Brémond, C. MacLachlan and L. Stepan, SatDevOps: A Novel Automated Satellite Operations Methodology, SpaceOps 2023.
[2] Loft Orbital Solutions Inc., SatDevOps™, 2023.
[3] P.-D. Vaujour and L. Brémond, System and Method for Providing Spacecraft-Based Services, U.S. Patent No. 10,981,678, Washington, DC: U.S. Patent and Trademark Office, 2019.
[4] Overview of Space Communications Protocols, Informational Report, CCSDS 130.0-G-3.
[5] Telemetry and telecommand packet utilization, ECSS-E-ST-70-41C.
[6] Cubesat Space Protocol (https://github.com/libcsp/libcsp)
[7] CCSDS File Delivery Protocol (CFDP), Recommended Standard, CCSDS 727.0-B-5.
[8] UDPcast (https://www.udpcast.linux.lu/).
[9] NNG: Lightweight Messaging Library (https://nng.nanomsg.org/).
[10] Grafana (https://grafana.com/).
[11] Kubernetes (https://kubernetes.io/).