1. **Title**: A new “Cloud Ready” Command and Control Center for small satellites – Lessons learned from first experiments
2. **Proposed topic:** New Technologies .
3. **Full name of all author(s):** Christophe Pipo, Pierre-Alban Cros
4. **Indication of presenting author:** Christophe Pipo
5. **Institution addresses and e-mail, telephone contact details :**
   CS GROUP – France – ZAC de la Grande Plaine, 6 Rue Brindejonc des Moulinais – 31506 Toulouse Cedex 5
   Christophe Pipo, System Architect, christophe.pipo@csgroup.eu +33 5 61 17 62 81
   Pierre-Alban Cros, Project Manager, pierre-alban.cros@csgroup.eu, +33 5 61 17 65 04.
6. **Proposed type of presentation (oral or poster):** ORAL
7. **Summary, major results/ interests and novelties of the submission**

Abstract

For some years now, the NewSpace industry has been deploying nano/mini satellites and satellite constellations at an exponential rate. These new missions require more resiliency, high availability, size scaling, cost effectiveness and capabilities that legacy SCC systems lack to provide efficiently and that are inherent to Cloud systems.
Based on this analysis, CS GROUP is developing CSnano, a new Satellites Command and Control Center based on a microservices architecture and that leverages all the power of a “Cloud ready” system.
CSnano has been developed during two and half years and has passed its first steps toward an operational world. Several instantiations, in several contexts, have been implemented since last year.
This paper is focusing on the design of the solution, and on two first instantiations of the system in different contexts.
In the first experiment presented, CSnano was deployed in an existing ground segment of an in-flight mission and it has taken the responsibility of several satellite passes. This experiment was a good opportunity to test the scalability of the system and in particular its capability to downsize to a minimal set, since the entire control center software have been deployed on a single laptop.
The second experiment we talk about takes place in the context of a prototyping phase for a ground segment for a constellation of satellites. In that case, we have tested the deployment aspect in an existing cluster and we have pointed out the benefits of the Cloud architecture with respect to deployment standardization, resiliency and business continuity.
Through the presentation of these two case studies, we provide some feedback and lessons learned regarding the benefits and some drawbacks of Cloud technologies in the frame of a command and control system.
Finally, we conclude by addressing a potential evolution towards a satellite control center as a service.

1 Context

Satellite constellations have existed for many years. Among the best known are the GPS and GALILEO positioning systems (28 satellites) and the IRIDIUM constellation launched in the 2000s, which includes some 60 satellites. These constellations are operated by control centers with an infrastructure designed for each mission. They are proof that current control center architectures can meet the needs of this type of mission.

However, the growth of the broadband Internet market has led to the emergence of a new type of constellation, much larger in number of satellites. We are now talking about a mega-constellation of several hundreds (or even a thousand) satellites! The SpaceX company has already launched about 1600 satellites of its StarLink mission, since 2019. And the company plans to increase its constellation up to 4200 satellites in the next 18 months. This will represent approximatively the two thirds of all satellites currently in operation! We should also mention the OneWeb company, whose eponymous mission include more than 600 satellites. Other missions are appearing or being defined around the world (Planet Labs, Amazon, etc.).

We can therefore wonder about the adequacy of current control center solutions to the constraints of these mega-constellations.

2 Constellations challenges

The most important aspects of a mega-constellation of satellites are:

- **Performance**: a constellation drastically increases the volume of data to be processed and the data rates to be considered. Existing control center systems are simply not designed to handle this information overload. Nevertheless, information is inherently real-time in a control center, and it should be possible to have the most recent information at all times and without error.
- **Scaling up**: due to the large number of satellites to be launched, it is not possible to wait until the constellation is complete before operating it. The system must be able to adapt to the number of satellites, by having the capacity to process more and more information. In addition, we want to have a system capable of addressing any type of configuration, from a single-satellite mission to mega-constellations. Such a system must be natively scalable and not based on a fixed architecture.
- **Human/Machine Interface**: The multiplication of information sources requires rethinking how to present the user with a summary of the mission. It is necessary to have a means of aggregating information and to review the general ergonomics of software.
- **Automation**: the automation of routine operations seems essential to operate a mega-constellation: it seems unlikely to mobilize an operator per satellite as it is done in more traditional missions (including few satellites). Similarly, it is unlikely that a single operator can observe the behavior of the entire constellation. It is therefore essential that the final system provides a high degree of automation. It would probably be beneficial to rely on artificial intelligence technics to increase this level of automation.

- **Availability**: The notion of availability is not new in the world of space missions. Most missions require an availability of at least 90% (36.5 days/year of service interruption). The size of the constellations considered, however, greatly complicates the implementation of this availability. It is not sure that the current solutions are reusable in this context (too expensive or too complex to implement, or simply not adapted to the combinatorial).

These aspects appeared to us to be preponderant during the study of the characteristics of the mega-constellation. They include several technological locks that must be removed to define a system capable of commanding and controlling the satellites of a mega-constellation.

3 Microservices architecture

Before explaining how microservices can be applied to satellites constellation control centers, let us recall some key concepts of microservices architecture.

3.1.1 Key concepts

A **microservice** ideally matches the following key characteristics, or at least a subset of them:

- ✓ Small in size
- ✓ Messaging enabled
- ✓ Bounded by contexts
- ✓ Autonomously developed
- ✓ Independently deployable
- ✓ Decentralized
- ✓ Built and released with automated processes

3.1.2 Services in microservices

The term “microservice” can be fallacious as it can suggest that a microservice always exposes a service. It does not correspond to the reality.

The word “service” has not the same meaning whether we consider “microservice” and in “REST service” points of views. For “microservice”, the word “service” has a familiar meaning: the component playing its role *contributes* to the whole system. For “REST Service”, the word “service” is more technical as it is the set of operations that can be remotely called.

So, a microservice does not always provide a REST service. Without REST services, it can be internal without any input or output from and to the outside of the cluster.

But the system within the cluster needs to interact with the outside. For that, some microservices expose REST services. These services are only accessible through a Secure Gateway that provides authentication and authorizations.

In the figure below, we can see the two zones: the cluster zone protected by the gateway and the client zone. A yellow circle represents a service.

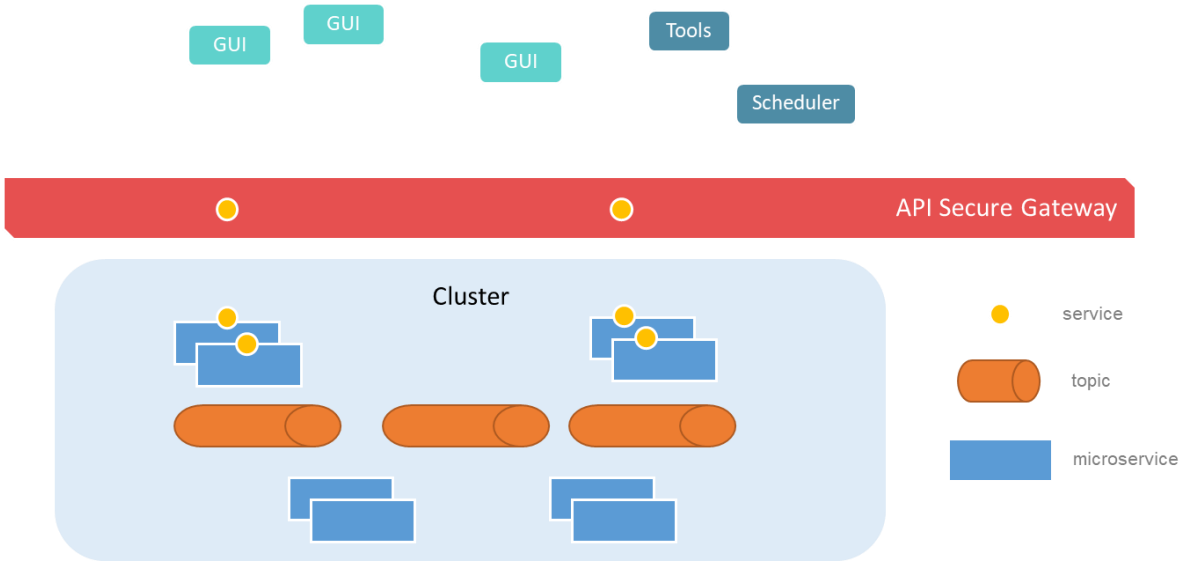


Figure 1 Services and microservices

3.1.3 Key patterns

The best **communication** between microservices relies on a MOM[1] so that components are not coupled.

Many patterns exist regarding microservices. The most structuring ones are illustrated here after in the context of a satellite control center.

3.1.3.1 Event Collaboration

A microservice does not know the whole system. It only knows its inputs and its outputs. Thus, the behavior of the system is an emerging behavior.

[1] Middleware-Oriented Messaging service

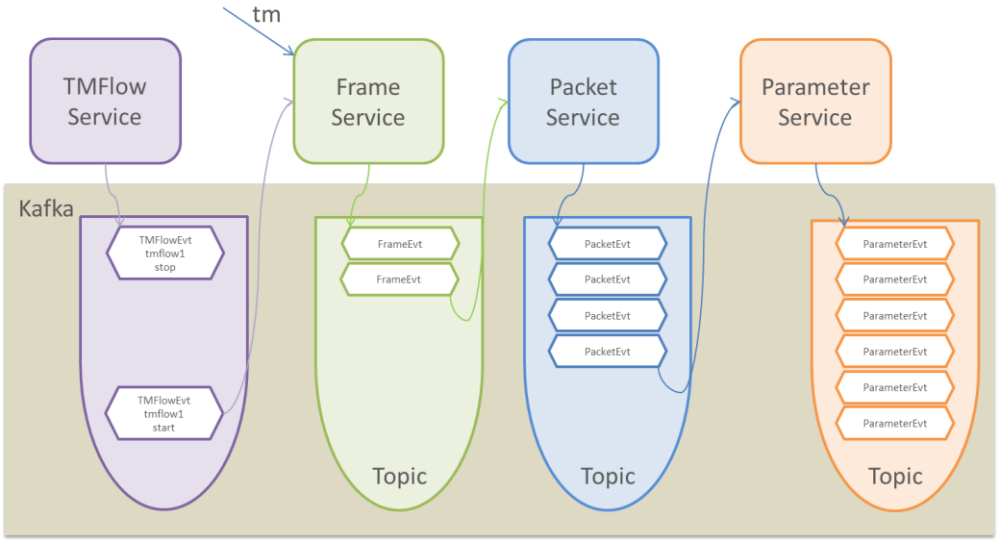


Figure 2 - Event collaboration for a telemetry chain processing

3.1.3.2 Event Sourcing

For stateful microservice, event sourcing consists in retrieving a state from replayed events which is called “re-hydration”.

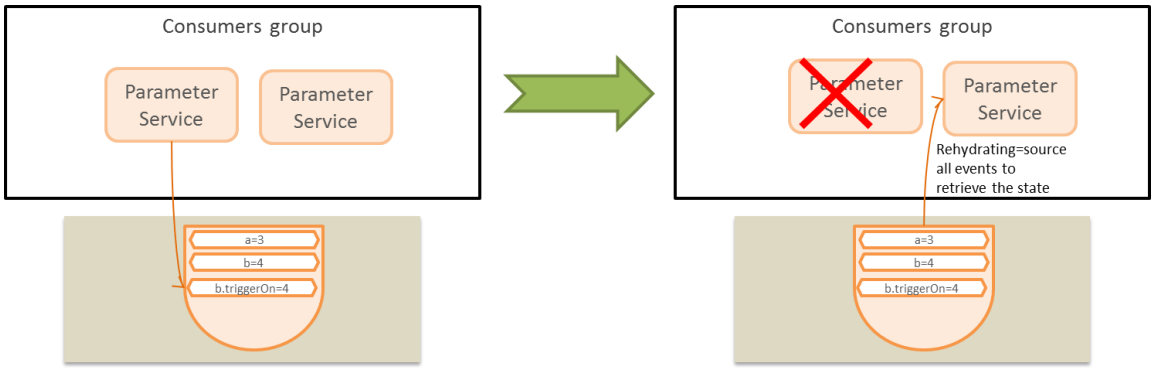


Figure 3 - Event Sourcing and re-hydration upon failover

3.1.3.3 Command Query Responsibility Segregation (CQRS)

A microservice that must answer to queries shall store its data on its own, in a way that is efficient for these queries. It can be shocking but the duplication of data that is avoided in classical architecture is a common practice in microservices.

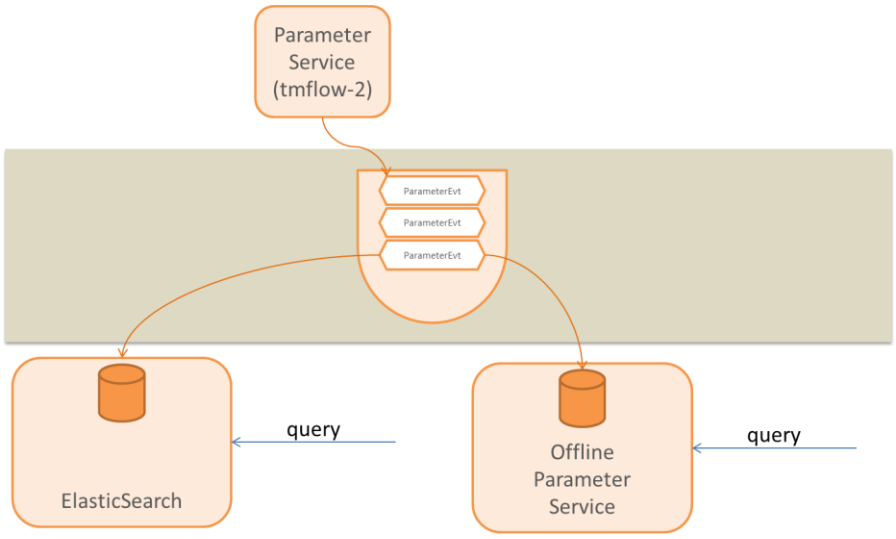


Figure 4 - CQRS applied to offline analysis of telemetry

3.1.3.4 Transfer State

An event is a notification that contains information. So, a microservice can easily share its state with other microservices via events subscription or playback.

In the figure below, the parameter service subscribes to system parameter service to store the values in

a local storage. Thus, the parameter service has always access to the system parameters event if the system parameter service is down and the topic of system parameters deleted.

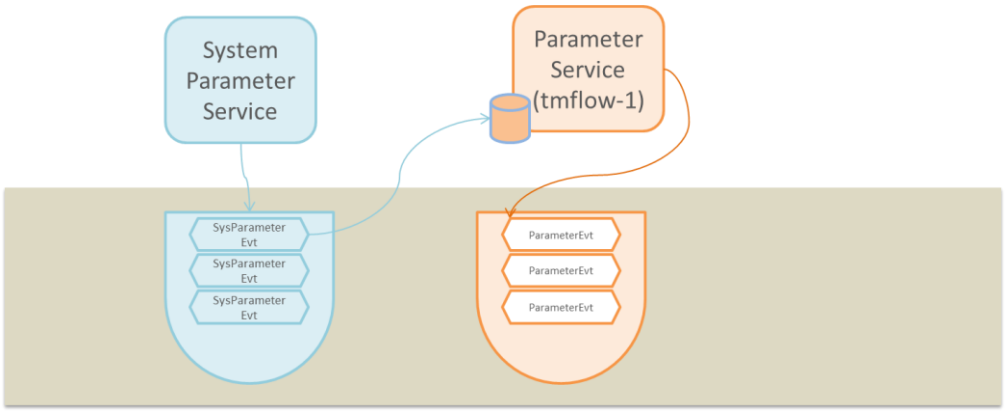


Figure 5 - Transfer State

4 Microservices Key features

Thanks to the concepts described in the previous paragraph, a microservices architecture can natively and gracefully provides:

- **Hot Scalability**
  While running, it is possible to increase/decrease the number of machines, the number of replicas for each service for which the infrastructure performs load-balancing.

- **Replaceability**
  Historically, replaceability was the main purpose to make *micro* services. A microservice can be hot replaced, hot upgraded without impacting the whole system. During the operation, no messages are lost thanks to the retention in the MOM. As a result, DevOps can naturally be applied to microservices.

- **Resiliency**
  If a microservice crashes it is automatically re-started by the infrastructure, for example Kubernetes or Docker Swarm.

- **High Availability**
  REST services exposed by microservices, mainly consumed by GUI client applications or external customer services, are still available thanks to the resilience and the mesh routing implemented by virtual networks in the cluster (for example, Docker Swarm or Kubernetes).

5 CSnano

CSnano is a CS GROUP software package to develop satellites control centers. In this article, the objective is not to describe the features of CSnano but to focus on how its architecture can solve the challenges raised by constellations.

5.1 CSnano chains

The chain concept is a key concept of CSnano, the one that makes possible the scalability at applicative level.

A chain is a set of microservices collaborating to communicate, directly or indirectly, with the satellite. The most used built-in chains are the “telemetry chain” and the “telecommand chain”.

5.1.1 Telemetry chain

The telemetry processing chain is composed of the following microservices:

- **acquisition.app**: acquisition of raw telemetry frames from an external source.
- **packetizer.app**: extraction of raw telemetry packets (e.g., CCSDS packets) from the telemetry frames.
- **decom.app**: decommutation of telemetry packets according to a telemetry packet descriptor, transforms raw (i.e., binary) telemetry packets to physical telemetry packets. Physical telemetry packets are provided in JSON format.
- **tmbroker.app**: allows external clients to subscribe to physical telemetry packets using WebSockets.

The following figure illustrates how the different microservices work together to acquire, to process, to monitor and to distribute the telemetry.

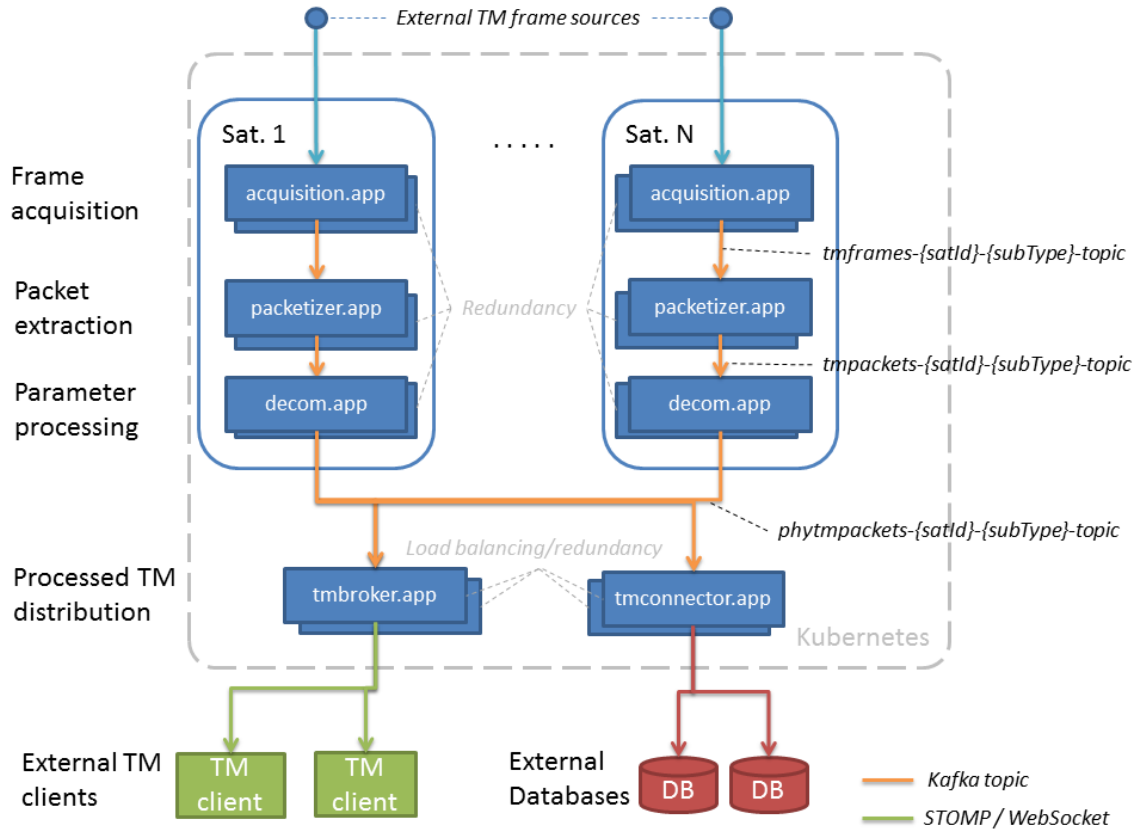


Figure 6 The built-in telemetry chain

5.1.2 Telecommand chain

A command chain encompasses all the microservices required to send a command to the satellite. It is composed of the following microservices:

- **encoder.app**: providing a REST API for encoding a command **and** building the TC packet containing this command.
- **segmentation.app and cltu.app**: providing the services to build the structures required to carry the commands up to the satellite according to the board to ground protocol in use.
- **tchistory.app**: providing a REST API allowing to retrieve execution reports for each Telecommand sent.
- **tcbroker.app**: which is responsible of handling the communication with the external target to which the commands are sent (i.e. the station network or a simulator for instance).

The following diagram illustrates how the microservices interact with each other.

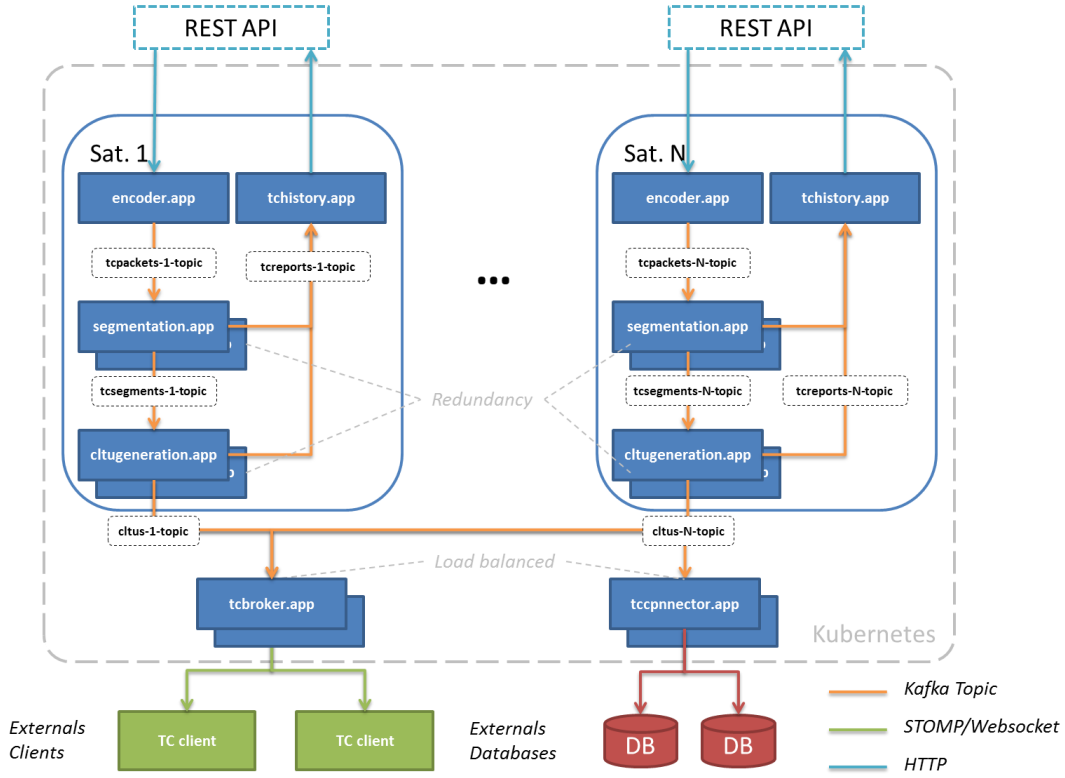


Figure 7 The built-in telecommand chain

5.2 Rational microservices development

Developing microservices can quickly become very cumbersome without rules, without methodology. To develop CSnano properly, it was important to rationalize how the system can grow.

To formalize the development of microservices, we have identified the different kinds of microservices we need in a system. It is always comforting to know that a new service belongs to a well-known, identified category. Complex off-piste skiing is suspicious in computers science… Once the type of a microservice is well defined, the problems can be anticipated and gracefully solved in a standard way and, above all, we can check that our microservice is really a microservice.

For that, from different point of view, we are going to bring to light some types and some recurrent characteristics of microservices.

5.2.1.1 Functional programming point of view

From a functional programming view, external events must be considered as service operations, in other words a call to a function. The REST services map very well to this kind of abstraction.

As to Bus events, you may have guessed that they are input and output data of a service. This obvious fact makes us think of functional programming. If you are familiar with Java language you may know the

package `java.util.function` in which we can find `Consumer`, `Supplier` and `Function` functional interfaces. If you are familiar with functional programming theory, you also know these concepts.

Let us apply functional programming at service level.

Supplier Service

This type of service produces events without consuming any. For example, a microservice based on a "crontab" is a supplier service. Imagine a monitoring service that monitors the state of the host and regularly produces results. In a supplier service, the stimulus that causes the production of the events is intrinsic.

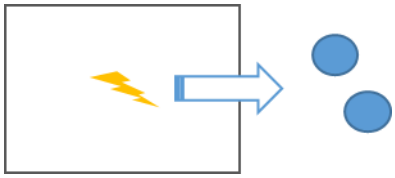


Figure 8 Supplier service

Consumer Service

Contrary to the supplier service, the consumer service consumes events but not produce any. We could call it "sink service". This is a "terminal" service. Indeed, because the service is not useless, it necessarily produces something (data in a database or in memory etc.), but this something is not an event. Aa an example: a service in charge of sending SMS or a service in charge of printing some events on a printer are both consumer services.

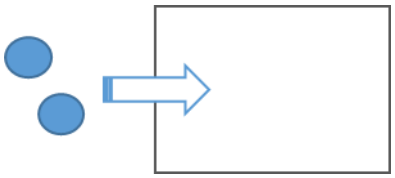


Figure 9 Consumer service

Function Service

A Function service is the canonic form and the more frequent form with input events that are processed to produce output events. A decom process is a good example of a Function Service.

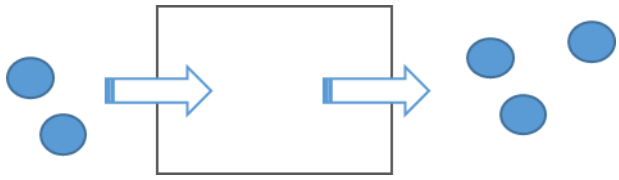
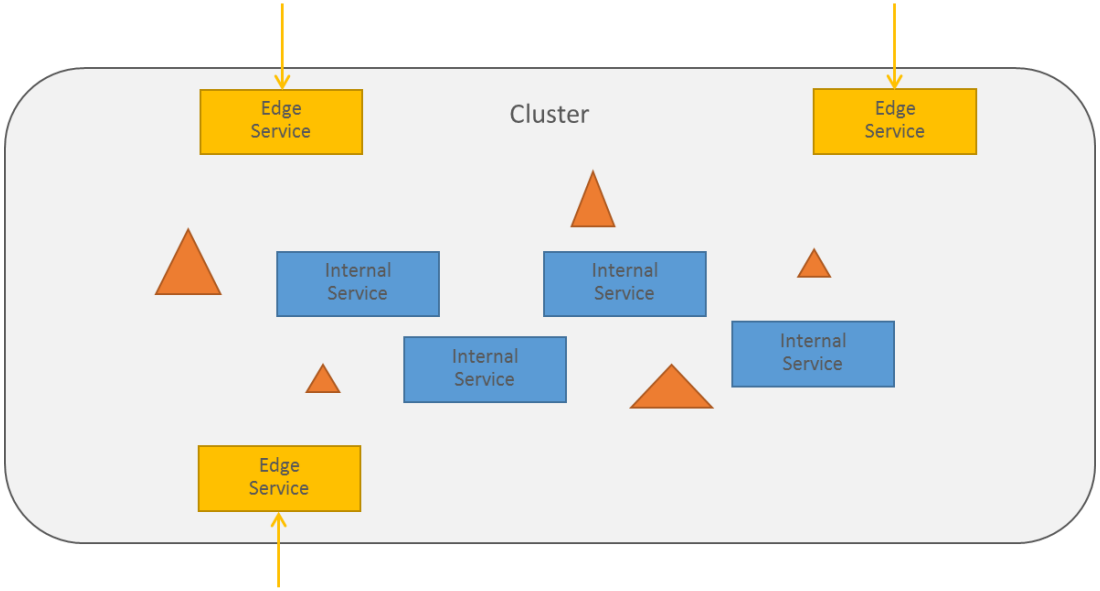


Figure 10 Function service

5.2.1.2 Topology point of view

At a lower level of abstraction, where bus events and external events are well distinguished, it is possible to create two categories of service. Some services are not exposed to the outside while others are. The first ones are internal to the cluster whereas the others can communicate with the outside. Naturally, on a figure, we are inclined to draw services on the edge of the cluster if they communicate with the outside. That is why they are usually called "**Edge Services**". Other hidden services in the cluster are called "**Internal Service**".



5.2.1.3 Domain point of view

Now, let us look at which domain a service works for.

Kubernetes has the concept of the **namespace** in order to group and isolate services related to a same domain. For example, all services that are started by Kubernetes itself to make the cluster operational are in the "system" namespace; the services installed by the user are by default added to the "default" namespace. It is possible to create your own namespaces.

This shows that in a microservices application, all the microservices do not contribute to the same domain.

In the case of CSnano, we can distinguish at least 4 domains:

- **Infrastructure**
  It consists in the services installed by the bus middleware.

- **Supervision**
  Supervision services usually run on each node of the cluster.

- **Business**
  The business services are the different services of the telemetry chain. In general, within the business domain, the services are grouped into “stacks” (an application).

- **Meta business**
  The chaincontroller service can be considered as a meta service since it manages the lifecycle of other services.

5.2.1.4 Microservice classification

As illustrated in the figure below, finally, the previous paragraphs have highlighted a three-dimensional classification of services.

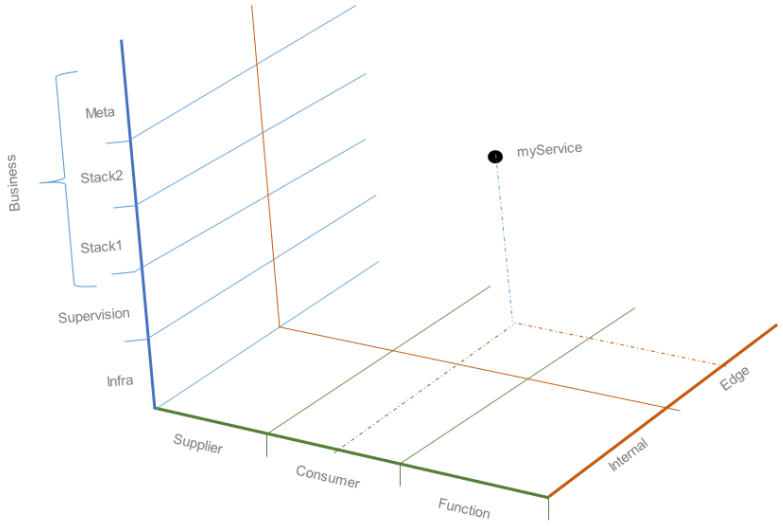


Figure 11 Microservice classification

No doubt that the classification that is presented in this document is not exhaustive. The purpose is not really to classify a microservice. Indeed, the real goal is to ask ourselves the right questions:

- What my microservice does?
- How does it behave regarding events?
- Which events does it deal with?
- Which domain does it address?
- Is it private or public?

- …

Without this questioning, microservices architecture have only drawbacks.

5.3 Microservices inter-communication

All microservices asynchronously communicate between each other through the HA[2] Kafka middleware. The retention of messages makes possible the resiliency when a microservice crashes. The Kafka brokers are replicated on different nodes in the cluster to ensure the high availability of the messaging system.

The Kafka bus used by CSnano is private, but it can be exposed to the outside if needed.

5.4 Architecture picture

Below, the diagram illustrates the (simplified) architecture of CSnano in which the automation tool calls the *chaincontroller* services that control the telemetry/telecommand chains lifecycle. Outside of the cluster, the client applications interface with the backend through exposed REST APIs.

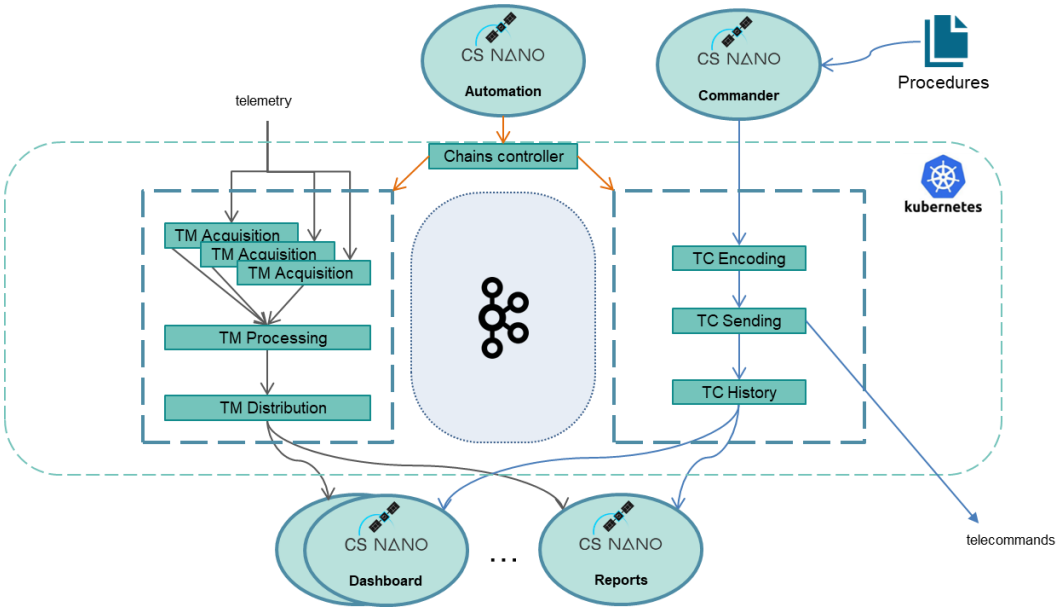


Figure 12 CSnano architecture

6 Is CSnano a solution for constellations?

Based on the overview on CSnano above, we can now check if the CSnano product does solve the problems highlighted at the beginning of this article (cf. §2) with legacy systems.

[2] High Availability

- **Scalability / High throughput**
  CSnano is scalable at two levels. On the one hand, deployed on a Kubernetes cluster, CSnano can benefit from the scalability at infrastructure level. For example, if the machines of the cluster have too many running processes, it is possible to add some additional machines to increase the computing power. On the other hand, CSnano provides the scalability at application level thanks to the "chain" concept (cf. § CSnano chains). Basically, the number of chains must be increased when the number of satellites grows.

- **Low coupling middleware**
  All microservices always communicate together through Kafka topics. When it is not appropriate, they communicate via REST API.

- **DevOps**
  The microservices architecture is very well adapted to DevOps as a single microservice can be modified without service interruption. For example, the decom process of a telemetry chain can be replaced without stopping the chain. The mechanisms involved while upgrading at runtime a microservice are the Kubernetes state-keeper mechanism, the Kafka buffering and the election algorithm.

- **Automation**
  For now, the automation function is still under development but CSnano is ready to be controlled by an automation system.

7 First Instantiation

For this first instantiation, the objective was to replace an existing and operating control center with CSnano during some dedicated satellite passes on an operational in-flight mission.

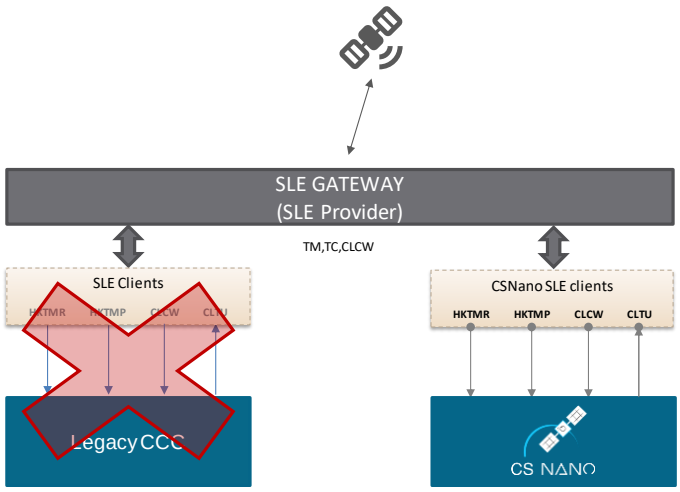


Figure 13 Satellite handling handover to CSnano scenario

CSnano was deployed in the existing ground segment infrastructure, and it has taken the responsibility of several satellite passes in replacement of the legacy system. This experiment was a good opportunity to test the downward scalability of the system and in particular its capability to downsize to a minimal set, since the entire control center software has been deployed on a single laptop.

7.1 Running on a single laptop

CSnano is a cloud-ready system but also a “laptop-ready” system. The possibility to deploy CSnano on a single laptop to operate one or two satellites is a very important feature to address small satellite control centers without available a large infrastructure. It is also a key feature to address different utilization targets like during the Satellite Assembly Integration and Test phases.

Technically speaking, the deployment on a single laptop does not differ from a deployment on a cluster. Indeed, on a single laptop, CSnano is deployed into minikube[3] or k3os[4] that are tiny Kubernetes clusters running on a single machine.

The target laptop has 32 Gbytes of memory and a 7cores CPU. In such a configuration, all microservices had just one replica to minimize memory and CPU consumption.

7.2 Running on obsolete OS

To be compliant with security constraints on this first mission, we had to deploy on an old CentOS release. Fortunately, we could find a version of Docker compatible with this release. This point was crucial as all microservices are packaged as Docker images.

The power of containerization avoids many issues related to installing different software.

7.3 Satellite database

CSnano has its own satellite database format: JSOC. JSOC is a straightforward YAML[5] format that concisely describes all the telemetry parameters and all the telecommands. Some dynamic parts of this format, like the monitoring functions, the calibration functions etc. are very flexible as they are written in JavaScript.

Because the operational database was in XTCE[6], we had the opportunity to use some tools to perform the conversion from XTCE to JSOC.

7.4 SLE gateway

The little story about SLE is interesting as it shows how a microservices architecture can be flexible, evolutive and heterogeneous.

At the time of this experiment, the SLE protocol that ensures the communication with the station was not yet integrated into CSnano. As a result, we had to quickly integrate SLE to CSnano. For that, some SLE

[3] https://github.com/kubernetes/minikube
[4] https://k3os.io/
[5] https://yaml.org/
[6] https://www.omg.org/xtce/

components from an old system were reused, with their own protocol (CORBA), and quickly wrapped into Docker containers and Kubernetes PODs (processing unit in Kubernetes) with some adaptative services around. The telemetry chain and the telecommand chain were configured to address the new SLE microservice instead of the stations directly.

Finally, the new SLE microservice was added into CSnano without any deep and structural changes.

7.5 Use case

Once adapted, deployed, configured, and connected, CSnano was ready to be operated. The telemetry was decoded, monitored, and displayed on the visualization dashboard and some command procedures written in Python could be executed to send some commands in BD mode and AD mode (CCSDS COP-1 protocol).

8 Second Instantiation on-premises

This second experiment was realized in the context of a prototype phase for constellation of about thirty satellites. In this experiment, CSnano was deployed as part of a bigger system in the existing infrastructure provided by our customer. It was also the opportunity to test the deployment flexibility of our solution since only a subset of the CSnano services were required and deployed.

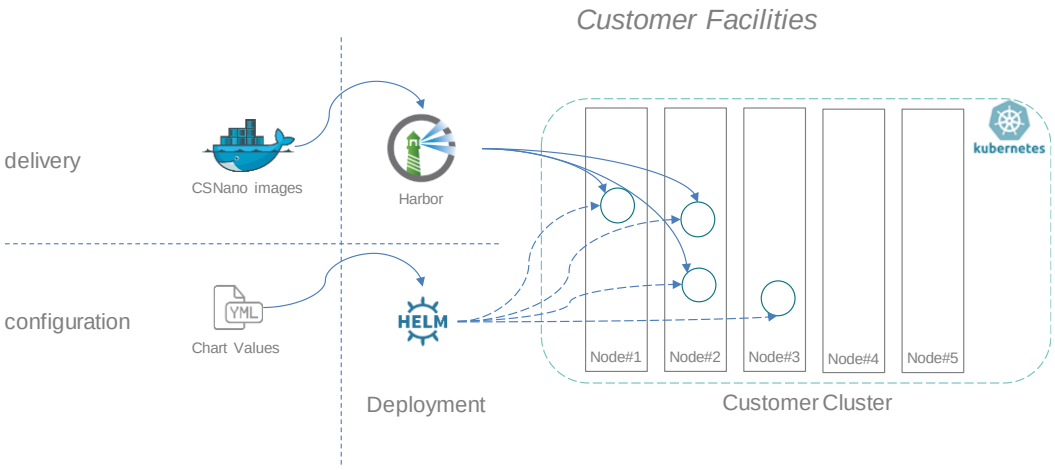


Figure 14 CSnano deployment principles on a private local cluster

Deploying on a cluster on the customer's site was a very good opportunity to test our solution on a cluster that is not managed by us. The main differences, we had to deal with, were:

- The cluster has no access to Internet
- The cluster is reachable through a VPN access
- The cluster has security constraints on entering traffic

8.1 Docker images

CSnano uses two kinds of Docker images:

- Public Docker images available in public Docker repositories on Internet
- Private Docker images that CSnano builds

The target cluster uses a private Docker registry that has no access to Internet. This implies that all Docker images used by CSnano must be pushed into this registry before. The CSnano installation provides the scripts to push CSnano private Docker images but not for public Docker images.

So, we wrote some scripts to automatically find all public Docker images used by CSnano. This task is not so obvious as there is no native tool to do that from Helm charts. Fortunately, some command tools like 'jq[7]' help to semantically "grep" YAML files… Once we got all public Docker images references, we pushed them into the private registry.

8.2 Ingress security

From the outside of the cluster, it is not possible to directly call a CSnano service exposed inside the cluster. Instead, we need to address to an HTTP reverse proxy deployed by Kubernetes. This reverse proxy is called "Ingress". It parses HTTP URLs and routes the calls to the right services inside the cluster.

The Ingress component has generally security constraints. For example, in our case, it limits the size of an HTTP payload to 20 MB.

This limit was a problem during the install when all the configuration data and satellite telemetry and telecommand descriptor files are pushed by a single "git push" command into a git server microservice. As a result, we had to modify the installation to split the single 'git push' into many little 'git push' commands.

We also had to remove some configuration snippets in the ingress configuration that were not accepted due to security constraints.

In short, the lesson learnt is that the Helm charts configuration must be very flexible to take into account the possible constraints of a cluster.

8.3 DevOps

An interesting use case was the demonstration of the DevOps capabilities of CSnano. The scenario was the following:

1. A telemetry chain is running
2. We ask Kubernetes to replace the decom process with a new one
3. Kubernetes starts the new process and stops the old one

[7] https://stedolan.github.io/jq/

4. Meanwhile Kafka is buffering messages
5. The new decom is running
6. No message has been lost

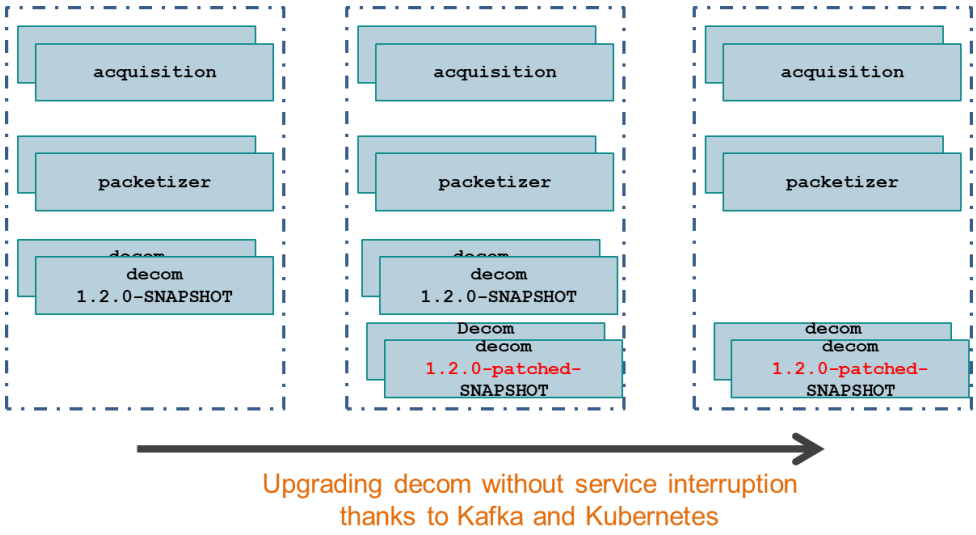


Figure 15 Business Continuity scenario

8.4 High Availability

The High Availability operates at two level. First, the applicative level with the collaboration of Kafka and Kubernetes. Second, at infrastructure level, with Kubernetes that is resilient to machine crashes.

The figure below illustrates two cases of failure to show the resiliency and the high availability of CSnano:

- Process failure
- Node failure

From the user point of view, the failures are not visible with just a few seconds of latency. It is the service continuity.

8.4.1 A process failure

A process failure is simulated by killing a running process. As soon as Kubernetes detects the missing process, it restarts it on the same node or another node. In the figure, the down process of tm chain #2 is restarted on node #1.

8.4.2 A node failure

Before the crash of node #6 (a machine of the cluster), the tm and tc chains are running their microservices on node #5 and node #6. When node #5 is down (the power supply is cut off), Kubernetes restarts the impacted processes on node #4. Meanwhile, the other microservices are still working and their messages are buffered in Kafka topics.

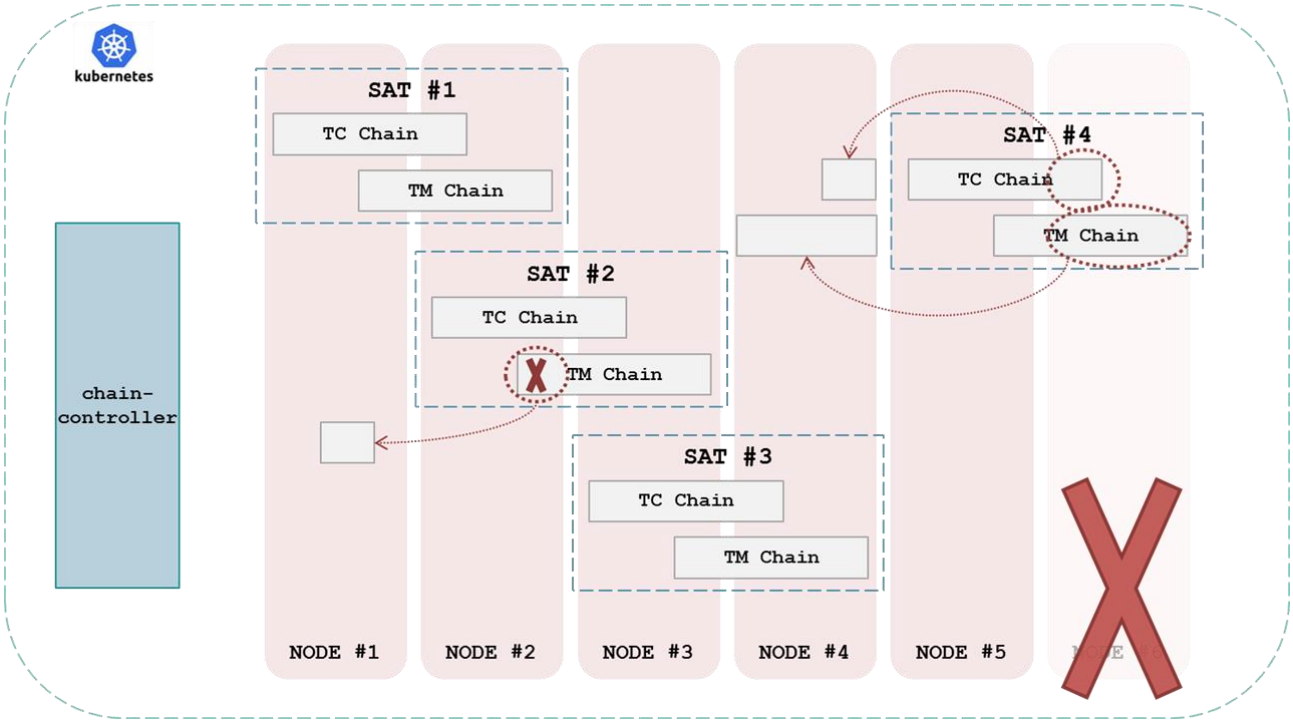


Figure 16 High Availability scenario

8.5 Cluster resources management

The cluster on which CSnano was deployed was also used by other teams. The workload was sometimes important so that there was latency on the network traffic and on CPU availability… This situation highlighted the fact that resources on a cluster are not unlimited.

Fortunately, the design of CSnano allows to apply two resources management policies:

- "**Booking**" policy: the resources are booked before and never released
- "**On-demand**" policy: the resources are booked when needed and released just after

The following table compares the two approaches:

	On-demand Policy	**Booking Policy**

Infrastructure overhead	A lot since processes keep starting and stopping	No
Reactivity	Micro-service has to start fast	Better
Scalability	Dynamic	Static
Number of processes	Dynamic	Static
Resources allocation	Adjusted, optimized	Maximal
System stability	Many variations	Better

For a “On-demand policy”, the CSnano chain is created just before the satellite flight-by and released at the end of the flight-by.

On the contrary, with a “Booking policy”, all chains are started at the start of the system and never stopped.

We could experiment that on a shared cluster, the “On-demand” policy is the best one to process many satellites without overloading the cluster. It would be also the best strategy on a cloud provider with no free resources.

9 Conclusion

We have seen CSnano fully benefit from the microservices architecture. The two different instantiations on two very different targets clearly showed that. The new technologies like Kubernetes and Kafka make possible a true High Availability of the system to be eligible to DevOps processes.

It is crucial to remember that microservices architecture has many pitfalls that can ruin all benefits. But thanks to a rational and quite simple design, we have succeeded to make the most of such an architecture.

CSnano is still a young product but it will soon be deployed and used to operate the ASTROID satellite which should be launched by the end of 2023. For this mission, CS GROUP will take the responsibility of the routine operations and will setup the Control Center room in its own facilities. A very exciting challenge that our CSnano product has made possible!