Intelligent Orchestration for Hosted Payload Architectures in Satellite-as-a-Service Missions

Riccardo Maderna
AIKO
Torino, Italy
riccardo@aikospace.com

Romeo Casesa
AIKO
Torino, Italy
romeo.casesa@aikospace.com

Christian Cardenio
AIKO
Torino, Italy
christian@aikospace.com

Federico Fontana
AIKO
Torino, Italy
federico@aikospace.com

***Abstract*—The rapid growth of the Satellite-as-a-Service market and In-Orbit Demonstration/Validation missions has increased demand for flexible hosted-payload architectures, where integrating third-party payloads onto a host bus raises challenges in resource contention, interface standardization, and operational risk. This paper presents orbital_OLIVER as an intelligent orchestration layer that decouples host-platform constraints from guest-payload objectives, enabling seamless onboard mission adaptation. orbital_OLIVER combines a dynamic planning engine, which generates payload-operation schedules based on resource availability, mission constraints, and request priority, with a conditional-execution framework that fuses payload and data-processing tasks into event-driven, real-time procedures. The user-centric architecture allows easy configuration of multi-stage conditional experiments, supports in-flight updates to reflect evolving requirements, and autonomously plans and schedules execution requests, reducing operator workload while improving scalability and real-time responsiveness.**

I. Introduction

The space industry is experiencing a shift from a product-based offering towards a service-based offering. An increasing number of "satellite-as-a-service" missions are being proposed, driving the trend towards space servitization [4]. Although the definition of different "-as-a-service" offerings varies considerably, these converge on the technological assets that enable these scenarios; intelligent payload orchestration plays a crucial role in allowing the transition from product-based offerings to services at scale. Current literature has identified resource allocation and fault tolerance [5] open research challenges which are critical to servitization of space. However, no integrated approach has been currently explored to tackle these challenges. This work discusses intelligent hosted payload orchestration as a critical enabling technology for space-as-a-service and introduces orbital_OLIVER as a platform for hosted payload orchestration within "-as-a-service missions".

Hosted payload orchestration refers to the coordinated management, scheduling, and operation of multiple payloads, often owned by different stakeholders, integrated on a shared satellite platform. In Earth observation missions, this typically involves aligning the needs of heterogeneous instruments (e.g., optical imagers, SAR, hyperspectral sensors) within the constraints of a single spacecraft's resources and operational concept. Orchestration goes beyond simple payload accommodation: it includes dynamic resource allocation (power, data handling, pointing), conflict resolution, prioritization of observation requests, and synchronization with platform-level functions such as attitude control and communications.

From a system perspective, hosted payload orchestration sits at the intersection of mission design and mission operations. During mission design, key challenges emerge from the need to reconcile competing requirements early in the lifecycle. Payloads may have incompatible observation geometries, duty cycles, thermal constraints, or data rate demands. Designing a platform that can flexibly support these without excessive over-dimensioning leads to complex trade-offs in power subsystem sizing, onboard data handling capacity, and attitude control agility. Additionally, interface definitions must anticipate a level of abstraction that allows payload independence while still enabling tight coordination, something that is difficult to standardize across payload providers [6], [7].

From the mission operations perspective, the orchestration problem becomes even more complex. Operators must handle competing tasking requests, often with different priorities, latencies, and service-level agreements. The traditional ground-centric approach, where scheduling is performed on the ground and uplinked as time-tagged commands, struggles to scale when the number of payloads and request frequency increases. It also limits responsiveness to dynamic events such as anomalies, natural disasters, transient phenomena, or changing environmental conditions (e.g., cloud cover). Furthermore, tightly coupled platform-payload interactions can create operational bottlenecks, where even small changes in one payload's plan propagate across the entire mission timeline.

This paper focuses on the second challenge by presenting orbital_OLIVER as an intelligent orchestration layer that supports and streamlines mission operations for flexible hosted-payload architectures. Thanks to its onboard autonomy functions, it decouples host-platform constraints from guest-payload objectives, reducing operator workload while improving scalability and real-time responsiveness and surpassing key challenges identified in literature hindering "-as-a-service" paradigms.

In the reminder of the paper, Section II details benefit of onboard autonomy and defines the required autonomy functions. Section III describes the autonomy functions provided by orbital_OLIVER, while Section IV highlight its user-centric design and configurability. Section V draws the final

conclusions.

II. Benefit of onboard autonomy operations

Onboard autonomous operations offer a compelling way to address these challenges by shifting parts of the orchestration function from ground to space. One of the primary benefits is platform-mission decoupling: instead of embedding payload-specific logic in the platform or ground segment, the onboard autonomy application can expose abstracted services (e.g., ways to post data acquisition requests and downlink opportunities), hiding orchestration and constraints satisfaction behind the scenes. This reduces interdependencies and allows new payloads or updated mission objectives to be accommodated with minimal redesign.

A second major benefit is enabling event-driven concepts of operations. Rather than executing a fixed timeline, the spacecraft can react in near real time to triggers such as failed acquisitions, onboard detections, external alerts, or changes in resource availability. For example, a payload acquisition over clouds could be rescheduled at the next opportunity while accommodating the operations of other hosted payloads, reallocating power and change data-downlink prioritization, all without ground intervention. This paradigm significantly improves responsiveness, resource utilization, and overall mission value.

To realize these benefits, a set of onboard autonomy functions is required. At the core is an onboard planning and scheduling capability that can generate and update activity plans based on high-level goals and current system state. This must be coupled with resource management functions capable of monitoring and allocating power, data storage, processing capacity, and thermal margins in real time. Equally important is a goal- and policy-based decision layer that can resolve conflicts between payload requests according to predefined priorities and constraints.

State awareness and health monitoring are also essential, as autonomy depends on accurate knowledge of the spacecraft's status and environment. This includes not only platform subsystems but also payload states and external factors such as orbital position and observation conditions. Event detection and processing capabilities are required to support event-driven operations, including the ingestion of onboard and offboard triggers.

III. orbital_OLIVER's autonomy functions

orbital_OLIVER [1] is an onboard autonomy platform developed by AIKO, which serves as an intelligent orchestration layer providing all required autonomy functions that enable satellite-as-a-service paradigms. First, it provides a flexible planning engine that dynamically generates payload-operation schedules by jointly accounting for resource availability, mission constraints, and request priority. Second, it embeds a conditional-execution framework that fuses payload and data-processing tasks into event-driven, real-time procedures, supporting complex concepts such as tip-and-cue and adaptive

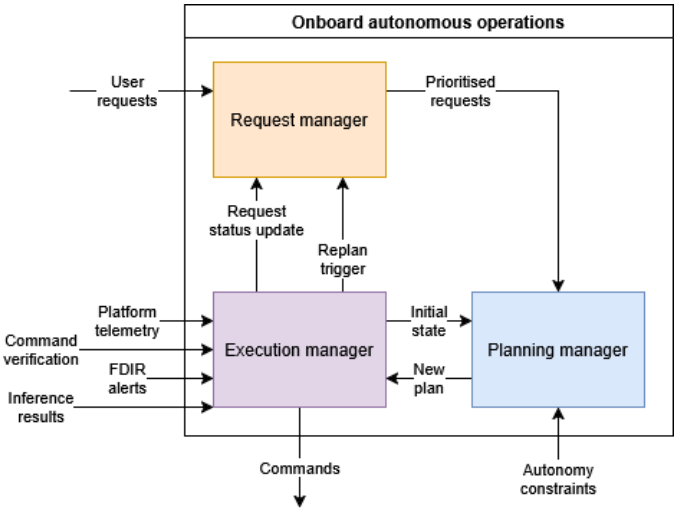

Fig. 1. High-level system architecture.

command sequences. To support event-driven operations, orbital_OLIVER can also provide advanced health monitoring functions to complement the spacecraft's FDIR system, and exposes standard interfaces for edge data processing application integration.

A. Autonomous planning and execution

The architecture of the planning and scheduling application (Figure 1) is composed of three primary components: Request Manager, Planning Manager, and Execution Manager. The Request Manager is responsible for managing incoming user requests. It monitors the life cycle of each request from submission to completion and assigns priority levels based on mission rules and operational constraints to ensure that the system processes requests in an optimal order.

The Planning Manager provides the proactive component: it generates long-term flexible plans of activity using a timeline-based approach. It incorporates user requests, mission goals, operational constraints, and mandatory activities received from operators. The resulting flexible plan can be seen as an envelope of deterministic schedules, each one representing a fixed-in-time sequence of actions or commands. This flexibility allows for absorbing uncertainties during execution without the need for re-planning, which strongly reduces the computational load of the whole planning pipeline and increases the predictability of the satellite's behavior.

The Execution Manager translates planned activities into a sequence of commands via conditional procedures. Flexibility of the plan and conditional branches are exploited to optimize the mission return and to adjust the schedule in response to just-in-time information (like edge data processing results) and contingencies. The Execution Manager also monitors the validity of the plan and triggers a re-planning process in case of: violation of resource availability envelopes, indicating a mismatch between forecast and actual resource use; occurrence of anomalies that invalidate the plan; reception of

new information, including new user requests or mandatory activities from ground operators.

Both Planning and Execution Managers implement generic engines that are highly configurable to mission requirements ensuring high re-usability, as described in Section IV.

The Planning Manager is based on timeline-based planning [3], a formalism well suited for mission planning thanks to its expressiveness and natural alignment with how space mission constraints and resources are modeled. In this approach, the world is represented through state variables whose values evolve over time. These variables may be external (e.g., ground station visibility) or internal (e.g., spacecraft orientation). Their evolution is encoded in timelines, which are sequences of tokens representing flexible time intervals during which each variable holds a specific value. A chronicle aggregates all timelines along with constraints and resource information.

The Execution Manager implements reactivity at two levels. First, it executes the flexible plan by assigning fixed times at time-points in the STNU. Firings of controllable time-points are chosen to optimize mission return; uncontrollable ones are based on sensed events or the conclusion of procedures.

B. Event-driven operations

Onboard orchestration beats ground mission planning as it can exploit complete and real-time information. The availability of accurate and real-time system knowledge of the spacecraft's status and environment provides the required information for optimal decision making, allowing not only increased mission efficiency, but also unlocking novel, highly dynamic mission concepts, such as multi-user payload utilization [4]

Alongside its autonomous operations application described in Section III-A, orbital_OLIVER natively provides an health monitoring application able to complement alert raised by the spacecraft FDIR with advanced anomaly detection and classification functions. The Detector module is powered by machine-learning algorithms trained to spot anomalies in telemetry data, whereas the Classifier module attempts to classify the detected anomaly based on a knowledge base extracted from experts and examples. The health monitoring application produces two types of data that are exploited to guarantee robust spacecraft operations and better orchestrate payloads: i) alerts that inform the autonomous operations application to avoid usage of unavailable resources or operating modes; ii) suggested recovery plans that are inserted in the plan as autonomy constraints.

Differently, applications able to process payload data to produce actionable information on board, like data quality measures (e.g., cloud coverage [2]) or event detections, are tightly related to the requirements of specific missions or hosted payload users. As such, it is difficult for orbital_OLIVER to provide a general purpose set of edge data processing applications. Therefore, orbital_OLIVER provides tailorable APIs to interact with external applications. Applications' results are exploited by the autonomous operations agent in two ways: they are input to evaluate conditional branches of procedure (for instance, discarding low quality data and abort further processing) and they are used to generate requests onboard (for instance, follow-up observations following fire detection).

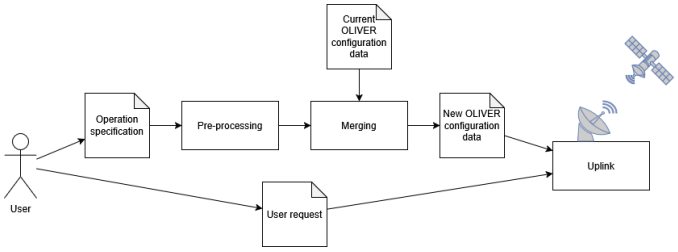

Fig. 2. Experiment definition and request process.

IV. User-centric configurability

The architecture is designed for user-centric configurability: users can easily configure experiments by defining payload and data processing tasks and combining them into multi-stage conditional procedures. The system automatically assembles them into a unified orbital_OLIVER configuration, which incorporates static information about the mission and the satellite platform and can be updated in-flight to reflect evolving user requirements. Execution requests can be submitted at any time; the onboard autonomy module plans and schedules them, ensuring continuous alignment with mission goals. Figure 2 depicts the overall process.

This process ensures mission operations carried out by ground operators are minimally impacted by changing user requirements, as hosted payload experiments are fully managed by orbital_OLIVER. Instead, the ground segment can focus on spacecraft and orbit maintenance.

A. Static configuration

Autonomous operation functions of orbital_OLIVER are configured via files and operated via telecommands or files. Most of configuration data is static and can be defined before mission starts: they encode base platform and mission characteristics such as resources, operational constraints, and available payloads. It is also possible to update configuration data during operations.

The actual configuration files are generated from a set of entities storing the configuration knowledge, namely:

- Mission Tree. Defines all elements that make up the mission in a structured manner, their properties, available telemetry, and associated commands.
- Planning Domain. Defines executable activities by the satellite, relevant external events for planning operations, and constraints between activities and events (causal and temporal). It also connects user requests to planner objectives, configuring how they will be interpreted and executed on board.
- Execution Domain. Defines on-board procedures for executing each activity. This information allows concrete translation of planner decisions into operational actions, defining the conditions and parameters required for each procedure.

- Platform State. Defines how the on-board system should interpret data received from sensors to build the internal mission state. Maps raw values into more abstract representations managed by orbital_OLIVER.

When an entity is updated, it is verified against a set of rules to ensure internal consistency and consistency with other entities. Consistency criteria include type checks, valid references between elements, completeness of representation.

1) Planning domain specification: It is composed of different sections:

- Resources: definition of relevant resources. It includes the list of resources, their type, and characteristics, such as upper and lower bounds. Supported resource types are: $capacity$ resources (that have a maximum capacity that limits simultaneous usage, like instantaneous electrical power usage) and $consumable$ resources (that are consumed or generated over time, like propellant or data storage space).
- States: definition of the timelines and their state values in the planning scenario. Timelines can model either external events or satellite activities. Activities are defined as possible state values of appropriate timelines. In defining each state value, one can specify the effects on the mission state (resource consumption) and feasible transitions to other states (causal constraints).
- Synchronization Rules: synchronization rules among timeline state values in the planning scenario, which enforce temporal and causal constraints among state values (that is, events and activities).

The overall structure of the file is shown in Listing 1.

As part of the static mission configuration, basic tasks of the satellite are specified, such as slewing, payload activation, and data transmission. For each task, resource usage must be defined, as well as some additional information consumed by the planning engine.

Tasks are often not independent but are components of macro operations. For example, the activity of downlinking data to the ground could be composed of: data preparation, point antenna to earth, data transmission. Besides, tasks may have operational constraints, such as specific allowed spacecraft attitude. Both information types are encoded via synchronization rules among state values.

2) Execution domain specification: It is composed of a set of procedure specifications and a map associating procedures to tasks. In particular, each procedure is defined in a dedicated file through a custom scripting language. The overall structure of a procedure is shown in Listing 2 and consists in separate bodies:

- Main (required): the primary execution body that runs by default.
- Interruption (required): executed when the procedure must be interrupted upon execution failure.
- Conclusion (optional): only present for procedures with controllable duration, to end the procedure nominally when planned.

Procedures accept input parameters to provide context and parametrize execution. Supported statement types are:

- Variable manipulation: definition, assignment...
- Command dispatching: post a command to be executed by the spacecraft.
- Control flow: if/else and loops. Conditional evaluations can be made on mission state values, results of data processing, and dispatched command states.
- Wait: procedures can wait on time intervals, mission state values or dispatched command states.

B. User experiment specification

Planning and execution domain specification allow the inclusion of user-defined experiments. These are special operations that can make use of available tasks and mostly relates to payload operations. The user defines operations characteristics so that they are known to orbital_OLIVER, which can then properly plan for them upon receiving requests (see Section IV-C). Structure of user inputs is shown in Listing 3, which includes:

- Experiment duration, with possibility to specify lower and upper bound for variable length operations.
- Resource usage.
- Synchronization rules, to specify operations sequences and operational constraints.
- Link to a procedure definition, provided in the dedicated file.

Insertion of user-defined operations into the configuration file of orbital_OLIVER can be performed automatically and it is mostly straightforward. With reference to the structure shown in Listing 1, user-defined info extends the static content of the file and supporting information, such as relationships between basic tasks and user-defined ones are automatically generated.

C. User experiment requests

Static configuration data are defined once to configure orbital_OLIVER behavior. Then, user can request the execution of an experiment included in the planning domain specifications by sending a request to orbital_OLIVER. Multiple requests can be sent to perform the same experiment multiple times and they specify the desired operation, its target, and the desired data processing operations. They can be sent to the spacecraft as telecommands or via files, and their ingestion does not require reboot of orbital_OLIVER, but are part of nominal operations. User request structure is shown in Listing 4.

V. Conclusion

The rise of satellite-as-a-service missions demands complex payload orchestration capabilities, including dynamic resource allocation and conflict resolution. The proposed approach, leverages the capabilities of orbital_OLIVER to enable autonomous operations and transfer the burden of ConOps from human operators on ground to an autonomous orchestration

```
<model>
  <resources>
    <resource name="resource_name" type="cap/cons" init="xxx" lb="yyy" ub="zzz"/>
    ...
  </resources>
  <states>
    <state name="timeline_name" type="timeline_type">
      <value name="value_name" ctrl="ctrl_value" rsrv="rsrv_value" lb="xxx" ub="yyy">
        <use resource="resource_name" amount="xxx" />
        ...
      </value>
      ...
      <transition source="source_value_name" target="target_value_name"/>
      ...
    </state>
    ...
  </states>
  <synchrules>
    <rule operator="operator_type" delay="xxx">
      <source name="source_state_name"/>
      <target name="target_state_name"/>
    </rule>
    ...
  </synchrules>
</model>
```

Listing 1: Structure of the planning domain specification configuration file.

```
procedure <name>(
    <param1>: <type1>,
    <param2>: <type2>,
    ...
)
main
    <main_statements>
    ...
interruption
    <interruption_statements>
    ...
conclusion
    <conclusion_statements>
    ...
end
```

Listing 2: Structure of a procedure specification.

layer. Its ability to dynamically plan, schedule, and conditionally execute payload activities allows satellite-as-a-service missions to respond in real time to evolving conditions and objectives without continuous ground intervention, while global mission value and end-user quality of service. This paradigm not only reduces operator workload but also unlocks more sophisticated experimental workflows, all while maintaining strict adherence to host-platform constraints and ensuring overall system stability.

```
<experiment name="name">
    <duration lb="xxx" ub="yyy">
    <use resource="name" amount="xxx"/>
    ...
    <synchrules>
        <rule operator="type" delay="xx">
          <source name="state_name"/>
          <target name="state_name"/>
        </rule>
        ...
      </synchrules>
    <procedure name="procedure_name">
</experiment>
```

Listing 3: Structure of experiment specification inputs.

References

[1] AIKO Srl. orbital_OLIVER. https://aikospace.com/products/oliver [Accessed: 2026-04-24].

[2] AIKO Srl. Onboard intelligence for Earth Observation. https://aikospace.com/products/clear [Accessed: 2026-04-24].

[3] M. Ghallab, D. Nau, P. Traverso. 2016. Deliberation with Temporal Models. Cambridge University Press.

[4] A. Hein, C. Rosete. 2022. Space-as-a-Service: A Framework and Taxonomy of -as-a-Service Concepts for Space. IAC-22-D1.2.12.

[5] Karteris, Antonis, et al. 2025. Satellite-as-a-Service Architecture for ML Edge Computing on Heterogeneous Processing Platforms. 2025 European Data Handling & Data Processing Conference (EDHPC). IEEE.

[6] V. Matricardi, L.C. Ripley. 2005. Pros and Cons of Standard Interfaces.

```
{
    "request_id": <id>,
    "arrival_time": <timestamp>,
    "range_of_validity":
        [<timestamp>, <timestamp>],
    "requested_activity": enum,
    "periodicity": float,
    "roi": <region_of_interest>,
    "client_id": <id>,
    "priority": enum
}
```

Listing 4: Structure of a user request.

[7] K. Reese, D. Acton, V. Moler, B. Landin, J. Deppen. 2013. Rapid accommodation of payloads on the Standard Interface Vehicle through use of a standard payload interface, 2013 IEEE Aerospace Conference, Big Sky, MT, USA, pp. 1-14.

[8] A. Wander, R. Förstner. 2013. Innovative Fault Detection, Isolation and Re- covery Strategies On-Board Spacecraft: State of the Art and Research Challenges. Deutsche Gesellschaft für Luft- und Raumfahrt (2013).