18th International Conference on Space Operations, Montreal, Canada, 26 - 30 May 2025.

SpaceOps-2025, ID # 449

A Method for Rapid Autonomous Deployment of Software Updates to Satellite Constellations

Adam Paul
Lynk Global, Falls Church, Virginia, United States of America, apaul@lynk.world

Abstract

As modern software development practices begin to take hold in the space industry, updates to spacecraft software are becoming more frequent, with some spacecraft potentially seeing updates to core software on a near daily basis. These updates need to be validated and tested prior to deployment, in addition to ensuring that they can be deployed in a fault tolerant manner. Spacecraft mega constellations create the added challenge of requiring frequent updates to fleets of spacecraft, of which not all are the same hardware revision or configuration. This paper presents a method for orchestrating daily updates for a satellite mega constellation using an automated deployment pipeline based on technologies developed for the IoT industry. This approach includes an automated CI/CD pipeline to build and deploy code to simulated spacecraft and flatsats for automated test sequences, ground software to store releases and coordinate uplinks to the constellation, and a fault-tolerant method of installing updates on each spacecraft. The paper also details a method for the management of software configurations for different hardware revisions of the spacecraft. This process allows for software updates to be deployed fleetwide with fully autonomous operations, while providing methods for robust validation, configuration management, and fault recovery.

1. Introduction

Updating software on spacecraft during flight has historically been a fairly uncommon occurrence. A traditional satellite can expect to receive only a handful of updates over a service lifetime of a decade or more [1]. However, industry best practices are changing. New spacecraft startups are adopting more agile development methodologies and building small, cheap spacecraft with a focus on iteration and rapid execution [2]. Part of this new methodology involves iterating quickly on software, not just during development but during flight. Satellites can be launched with only the basic software needed for maneuvering and communication, and teams can continue iterating on software post-launch to add features and improvements to spacecraft.

Lynk, a company building direct-to-device communication satellites, currently operates a fleet of 5 operational spacecraft with plans to launch thousands in the coming years. With our current fleet, we perform fleetwide software updates at an approximately weekly cadence. With this cadence, typical methodologies for validating and deploying software updates are too slow and resource intensive. Thus, an automated system needed to be developed. This paper proposes such a system, and covers solutions to three main challenges for deploying software to a satellite fleet: automatic validation and verification of new updates, automatic upload and coordination of a deployment schedule, and automatic install with safe rollback for failed updates.

2. Ground Validation

Changes to the satellites software need to be validated before they can be deployed to the spacecraft. Typically, software updates go through both unit and integration testing before deployment. Extensive solutions already exist for automated unit testing, so only integration testing will be focused on here. The proposed method involves two opportunities for highly automated integration testing - release testing and feature development testing.

Release testing is performed when an update is complete and ready for deployment to the fleet. A proposed workflow for automated release testing is as follows

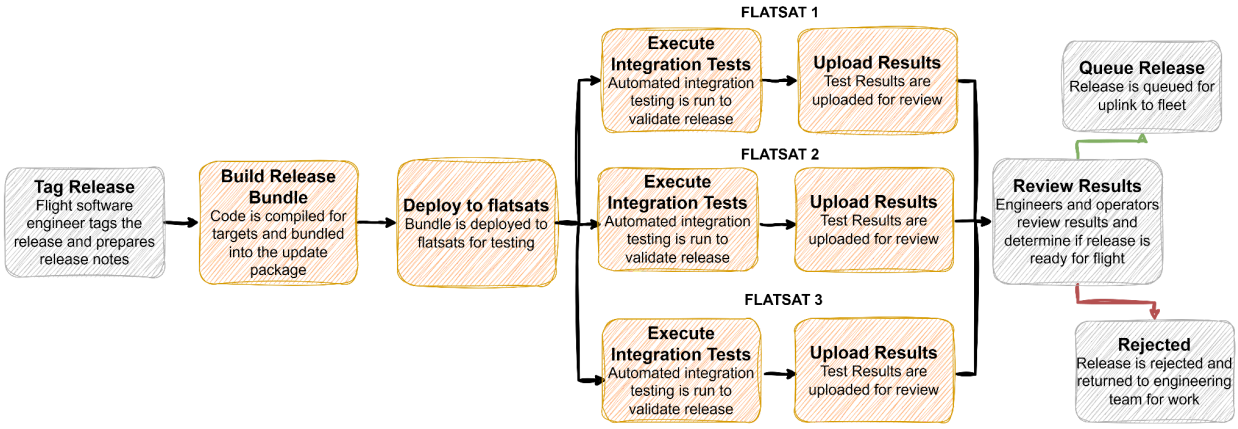

Fig. 1. Release testing and deployment workflow. Automated steps are in orange

In this workflow, the flatsat simulator is heavily used. The flatsat is a perfect hardware copy of the flight hardware, laid out across a table for easy access to components. Simulated spacecraft dynamics are fed to its sensors and a simulated ground station is used to communicate with it. Multiple flatsats are used, each representing a different generation of satellite hardware, and the release must be tested on all of them. When a new release is ready, a flight software engineer tags the code with a release version, which sets off an automated build process. The process compiles the software, packages it, and deploys it using the flight deployment method over a simulated ground contact radio link to the flatsat. Once deployed, automated integration testing is run. This is a set of tests built up over time that exercises all of the basic functionality of the satellite, ensuring that it can power on, perform maneuvers, communicate with the ground, and operate its payload. Specific tests can be added to exercise the new features added as part of the software update. If the tests fail, the test report is sent back to the flight software engineering team to review, revise, and re-attempt. If successful, the update is monitored on the flatsat for 24 hours to ensure it doesn't interfere with day-in-the-life operations, at which point a member of the operations team greenlights it for deployment to the fleet.

Unit and integration testing is also performed at the feature development stage. However, with multiple developers and a fast release cadence, there is not enough time to test each feature on the flatsats due to resource contention. Instead, a software simulator is used. The software simulator runs the full spacecraft flight software with simulated hardware interfaces, and is capable of running the integration test suite. While the software simulator is a lower fidelity simulation of the real spacecraft, multiple simulators can be run at the same time either on a central server or on the developer's laptop, enabling parallel testing of

features. The lower fidelity is of little concern as the features will be tested again on the flatsat once bundled into the release.

The proposed workflow for feature development is as follows

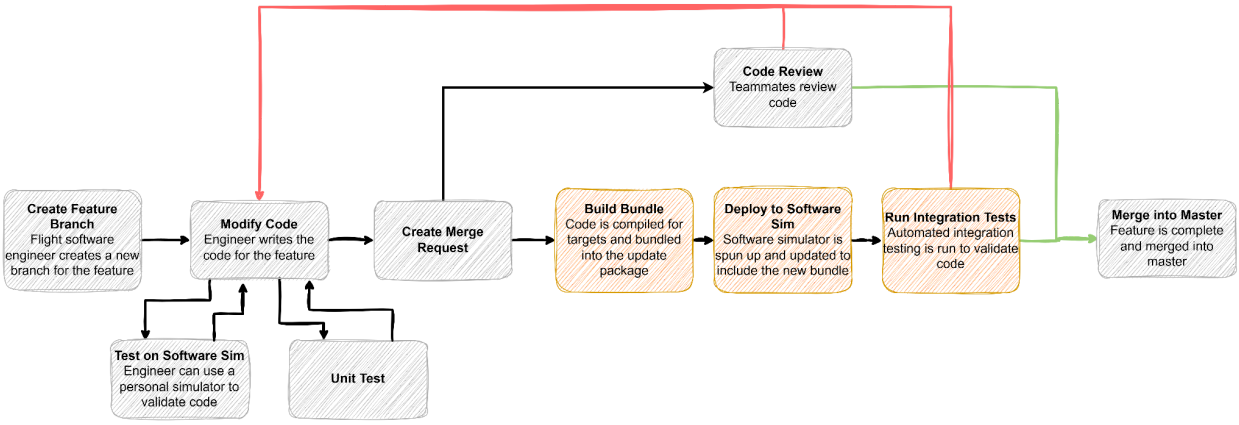

Fig. 2. Feature development workflow. Automated steps are in orange

By testing at both the release and feature development levels, a high level of confidence can be achieved in the functionality of new updates while managing the physical resources available for testing.

3. Upload Coordination

Once updates have been validated on the ground, they need to be deployed to the entire fleet. For a typical spacecraft mission, an operator may manually upload a package to the spacecraft, send a series of commands to load and execute the update, and monitor to ensure a successful deployment. However, for a fleet of hundreds of spacecraft, there is simply not enough time for operators to do this weekly.

For solutions to this problem, we can turn to the IoT industry for inspiration. Modern satellite constellations can be thought of as expensive IoT devices for the purpose of update delivery. Like IoT devices, they are physically unreachable for service and have limited connectivity. Multiple products exist in the world for deploying updates to fleets of updates to IoT such as Balena, Mender, and SWUpdate. They typically work in the following way [3]:

- A server on the ground stores update files and keeps a database of the update status on each device
- Each IoT device runs a service that handles the installation process
- The ground server exposes a web interface where operators can upload artifacts, and select devices for deployment
- When the device is able to contact the server, it will download and install any available updates.

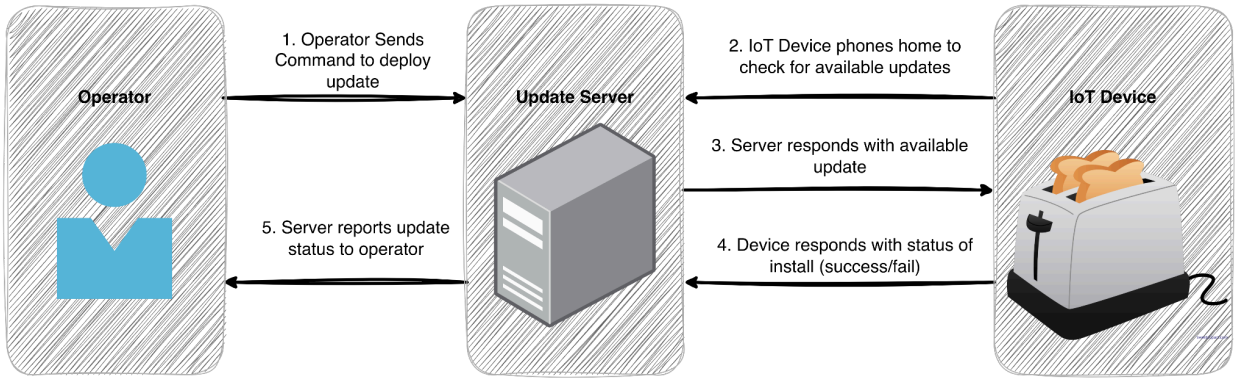

Fig. 3. Typical deployment process for IoT applications

For spacecraft, we can follow a similar workflow with some minor changes. One change regards the timing of deploying the update. Typically, IoT devices will perform the update as soon as it's received, or notify the owner of the device that an update is available and ask if they want to postpone (both options often causing much frustration to the owner). With spacecraft, the timing of the deployment is critical to ensure that no damage is caused to the system and that the service provided by the payload is not interrupted. The proposed workflow suggests two different types of deployment that the operator can choose from: queued or direct execution.

Direct execution allows for the update to be commanded at a specific time. The time can be selected to coordinate with other spacecraft activities and ensure that there is no service disruption. When paired with an automatic spacecraft activity scheduler, this can be done fleetwide autonomously. Operators can set specific rules for each update they wish to deploy and the scheduler will use these to find an opportunity for deployment and inform the spacecraft of the deployment time.

Queued execution can be used for updates that are simpler or affect fewer critical systems. These updates are sent to the spacecraft and placed in a queue. After queuing, a command is sent to the spacecraft to start processing the queue. Once sent, all updates in the queue will be processed in order of insertion. Spacecraft can be given a daily "maintenance window" where this queue processing can occur alongside other regular maintenance activities.

When deploying updates to the fleet, operators can also take advantage of a canary release structure. In the event that a bug makes it through release testing, it is important that it does not have the potential to take down the entire fleet at once. The canary release schedule involves deploying the update first to a small subset of the fleet, often older satellites or ones with degraded functionality. If there is an uncaught bug, it will appear on those satellites and allow operators to diagnose and fix before deploying fleetwide. Canary releases can be managed through the software system that handles deployment of updates to the fleet.

4. Spacecraft Install

Once the update has been commanded to deploy on the spacecraft, it must update the software in a reliable manner. If there are any errors or interruptions during the installation process, it must not leave

the software in an undefined state, as this could render the spacecraft inoperable. To do this, updates can utilize a rollback structure.

In this structure, update bundles are prepared with an install file that specifies what files go where on the spacecraft, as well as any custom scripts that must run to complete the install. As the install progresses, any files that will get overwritten as part of the installation are backed up, and the new files are installed. If the installation enters an error state, the process runs in reverse. The newly installed files are deleted and all backed up files are restored. Any custom scripts to complete the install have a counterpart that performs the operations in reverse, which are run here.

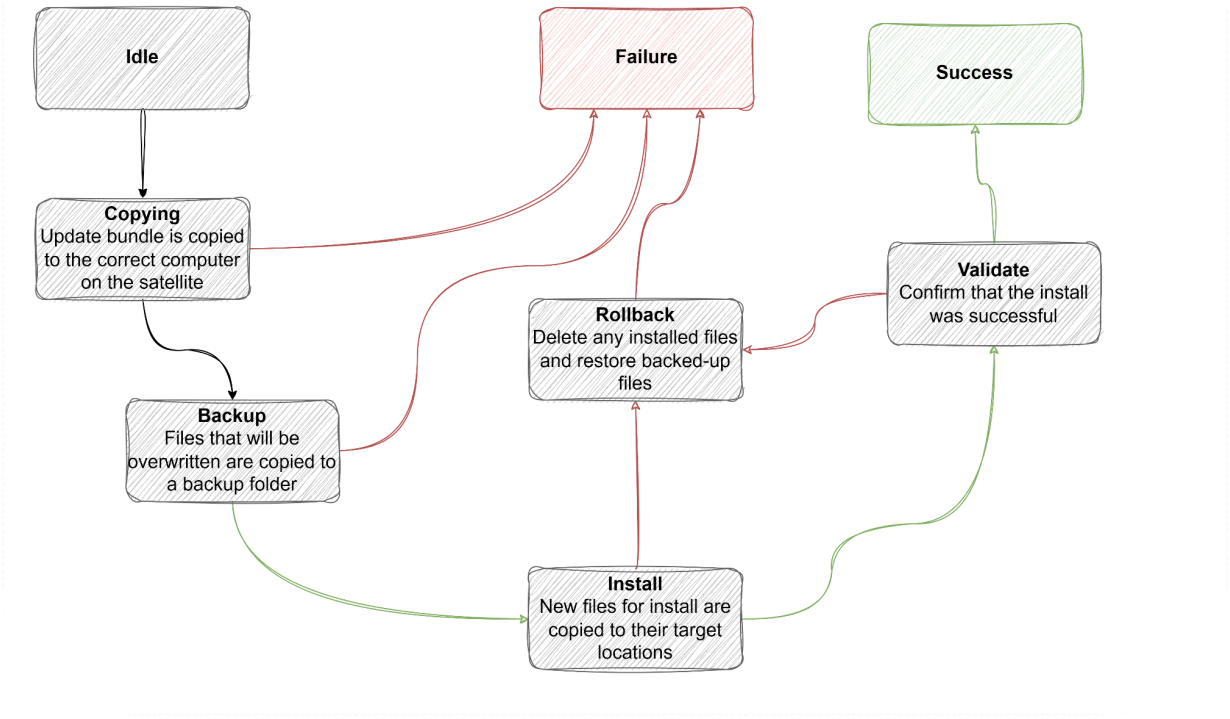

Fig. 4. Update installation workflow

But how can an error be detected? Errors can occur for a multitude of reasons, but can be grouped into three broad categories:

- An error with the installer process (missing file, improper logic)
- An external interruption to the satellite during install (power loss, restarting computer, etc)
- An error with the software itself (uncaught bugs in testing)

The first is fairly straightforward to handle. Programs return error messages that can be parsed and handled to trigger the rollback process. Regarding the second, recovery from power loss can be handled by storing state in persistent storage as the update is underway, allowing the system to restart, recover state, and rollback the update.

Errors with the software itself can be handled with a self-test and rollback mechanism. When the update is created, a validation key is created along with it. The key is a unique hash that identifies the update. When the update is installed, this key is passed both to a validation server and to the program being installed. The program is then free to implement whatever self-test mechanism it wishes to determine that it was installed correctly. This could be as simple as ensuring that it initializes properly, or as complex as running through all features of the program. Once it is satisfied, it sends the validation code to the validation server, confirming the update is successful. If the validation server does not receive the code within a certain timeout period, it automatically begins the rollback process.

5. Conclusion

In this paper, we have outlined a comprehensive methodology for orchestrating software updates across satellite mega constellations. By integrating automated CI/CD pipelines, high-fidelity flatsat and software simulation testing, and robust upload coordination mechanisms, our approach addresses the key challenges inherent in updating spacecraft software in a fault-tolerant and timely manner. Drawing inspiration from the IoT industry, we have adapted update strategies—such as queued and direct execution, as well as canary releases—to the unique operational constraints of space systems, ensuring that updates can be deployed safely across a heterogeneous fleet.

The hybrid testing strategy, which encompasses both feature development and release validation, provides a robust framework for detecting potential issues early in the development cycle while maintaining the agility needed for rapid iteration. Coupled with a rollback mechanism that safeguards against installation errors and unforeseen operational interruptions, our deployment process ensures that the satellite remains in a well-defined and recoverable state even in the face of anomalies.

As the satellite industry evolves toward more agile and responsive operations, the techniques presented here lay the groundwork for future advancements in spacecraft software management. Future work will focus on further refining these processes—enhancing telemetry feedback, incorporating advanced fault detection, and integrating more sophisticated scheduling algorithms—to continue improving the resilience and reliability of in-orbit software updates.

Ultimately, our framework not only enables fleetwide, autonomous software deployment but also establishes a scalable, secure, and efficient process essential for the next generation of space operations.

[1] Strout, N. (2021, July 22). *Space Force delivers software upgrades to satellite communications system*. C4ISRNet. https://www.c4isrnet.com/battlefield-tech/space/2021/07/22/space-force-delivers-software-upgrades-to-satellite-communications-system/

[2] Sweeting, M. N. (2018). Modern Small Satellites-Changing the Economics of Space. *Proceedings of the IEEE*, *106*(3), 343–361. https://doi.org/10.1109/jproc.2018.2806218

[3] *How mender works*. Mender. (n.d.). https://mender.io/engineers/how-mender-works