SateLight: A Satellite Application Update Framework for Satellite Computing

Jinfeng Wen
Beijing University of Posts and Telecommunications
Beijing, China
jinfeng.wen@bupt.edu.cn

Jianshu Zhao
Beijing University of Posts and Telecommunications
Beijing, China
jianshuzhao@bupt.edu.cn

Zixi Zhu
Beijing University of Posts and Telecommunications
Beijing, China
zhuzixi.zzx@gmail.com

Xiaomin Zhang
Beijing University of Posts and Telecommunications
Beijing, China
zxm987626@bupt.edu.cn

Qi Liang
Beijing University of Posts and Telecommunications
Beijing, China
liangqi@bupt.edu.cn

Ao Zhou*
Beijing University of Posts and Telecommunications
Beijing, China
aozhou@bupt.edu.cn

Shangguang Wang
Beijing University of Posts and Telecommunications
Beijing, China
sgwang@bupt.edu.cn

***Abstract*—Satellite computing is an emerging paradigm that empowers satellites to perform onboard processing tasks (i.e., *satellite applications*), thereby reducing reliance on ground-based systems and improving responsiveness. However, enabling application software updates in this context remains a fundamental challenge due to application heterogeneity, limited ground-to-satellite bandwidth, and harsh space conditions. Existing software update approaches, designed primarily for terrestrial systems, fail to address these constraints, as they assume abundant computational capacity and stable connectivity.**

To address this gap, we propose *SateLight*, a practical and effective satellite application update framework tailored for satellite computing. *SateLight* leverages containerization to encapsulate heterogeneous applications, enabling efficient deployment and maintenance. *SateLight* further integrates three capabilities: (1) a content-aware differential strategy that minimizes communication data volume, (2) a fine-grained onboard update design that reconstructs target applications, and (3) a layer-based fault-tolerant recovery mechanism to ensure reliability under failure-prone space conditions. Experimental results on a satellite simulation environment with 10 representative satellite applications demonstrate that *SateLight* reduces transmission latency by up to 91.18% (average 56.54%) compared to the best currently available baseline. It also consistently ensures 100% update correctness across all evaluated applications. Furthermore, a case study on a real-world in-orbit satellite demonstrates the practicality of our approach.

***Index Terms*—satellite computing, software update**

I. Introduction

Satellite computing [1], [2] is an emerging paradigm that endows Low Earth Orbit (LEO) satellites with onboard computing capabilities, enabling them to process mission tasks directly in orbit. LEO satellites [3], [4], exemplified by leading constellations such as Telesat, OneWeb, and SpaceX, are undergoing rapid development [5]. Operating at orbital altitudes between 500 km and 1,000 km, these satellites have traditionally been limited to data collection and transmission. However, growing demands for real-time responsiveness and massive space-borne data have driven a shift towards autonomous, intelligent satellites with integrated processing capabilities. Modern satellites are now equipped with processors, memory, and accelerators, forming onboard infrastructure capable of executing complex tasks, i.e., *satellite applications*.

Satellite applications refer to the programs deployed on satellites to perform mission-specific computations, which can capture raw sensor data, analyze features of interest, and make decisions or transmit results back to the ground stations. Representative satellite applications [6]–[10] include image encoding, object detection, feature tracking, etc. However, once the satellite is launched, the functionalities of satellite applications are generally fixed, rendering it inflexible to accommodate evolving mission objectives or unforeseen requirements. This lack of adaptability poses a significant limitation in the dynamic and long-duration context of modern space missions. Thus, the capability to remotely update satellite applications from ground stations to satellites, whether for new features or bug fixes, has emerged as a critical and urgent requirement in satellite computing.

Despite its critical importance, effective application updating in satellite computing remains an unsolved problem. Exist-

*Corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00018

ing software update approaches in terrestrial environments are generally divided into three categories: pre-update, in-update, and post-update. Pre-update approaches have focused on recommending *what* to update [11]–[14], assuming updates can be directly applied through full replacements. In-update studies have involved dynamic software updates, which enable runtime changes without service interruption [15]–[17]. However, these methods have depended on heavyweight static analysis, language-specific instrumentation, and extensive test suites, which could make them unsuitable for resource-constrained satellite systems. Post-update research has explored user feedback and update behavior analysis [18]–[21], but focused on software evolution rather than specific update mechanisms.

Updating satellite applications introduces several unique and under-explored challenges. (1) *Heterogeneous applications and constrained resources.* Satellite systems host diverse software stacks, making language-specific update approaches infeasible. Limited onboard resources (e.g., CPU, memory, and energy) further restrict complex update logic. (2) *Low-bandwidth, intermittent communication.* Communication uplink capacity from the ground to the satellite is severely limited (tens to hundreds of kbps), with short, infrequent contact windows, leading to a high risk of incomplete or delayed application update delivery. (3) *Harsh and unreliable space environments.* Radiation and thermal shifts in space increase the likelihood of in-orbit faults, turning failed updates into potentially mission-critical events. To the best of our knowledge, limited research has addressed these challenges. A critical gap remains in designing satellite application update approaches that can effectively handle application heterogeneity, communication constraints, and reliability requirements.

To address these challenges, we present *SateLight*, a practical and effective satellite application update framework for satellite computing. At its core, *SateLight* adopts a container-based design to encapsulate heterogeneous satellite applications. This abstraction leverages the well-established benefits of containers (e.g., flexibility and isolation), thereby simplifying application deployment and maintenance in resource-constrained satellite systems. Containerization also mitigates the complexity of language-specific software update logic, enabling uniform update management based on containers.

Built on this foundation, *SateLight* introduces three core capabilities: (1) a communication-efficient upload strategy that minimizes uplink bandwidth usage through content-aware differential identification; (2) a fine-grained onboard update strategy that enables deterministic reconstruction of target application versions; and (3) a fault-tolerant recovery mechanism that ensures update reliability by leveraging the layered nature of containers for rapid recovery upon failure. Specifically, *SateLight* employs a content-aware differential algorithm to extract semantic deltas between the existing and updated application versions. These deltas are encoded and transmitted to the satellite, reducing the data transmission overhead. Upon reception, the onboard satellite system performs an application reconstruction by applying fine-grained updates at line or chunk granularity to the original containerized application.

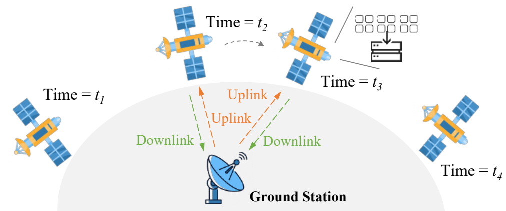

Fig. 1. Satellite-ground communication architecture.

To guarantee update reliability, *SateLight* integrates a layer-based rollback mechanism: upon detecting update failures, it instantly reverts to the last stable container layer, thereby eliminating the need for a full re-upload.

We implement *SateLight* and evaluate its effectiveness using 10 representative satellite applications written in diverse programming languages, within a satellite simulation environment. Results show that *SateLight* reduces uplink transmission latency by an average of 99.99%, 73.06%, and 56.54% compared to three baselines. Despite adopting a fine-grained onboard application reconstruction strategy, *SateLight* incurs only a negligible overhead, increasing execution time by around 2 seconds on average. Moreover, it consistently achieves 100% update correctness across all evaluated applications. We validate the practicality of *SateLight* through a case study on BUPT-2, a real-world in-orbit satellite equipped with onboard cloud-native computing. We release a public repository [22] containing data and code used in this study to support reproducibility and future research.

II. Background

A. Satellite-Ground System Architecture

As illustrated in Fig. 1, the interaction between LEO satellites and ground stations follows a highly time-sensitive and intermittent communication model, governed by orbital mechanics and visibility constraints. For instance, at t_1, the satellite is approaching the visibility zone but remains outside the effective communication range. At t_2, it enters the communication window, enabling both uplink (e.g., command transmissions, application updates) and downlink (e.g., telemetry, sensor data) operations. At t_3, the satellite exits the communication range and autonomously executes onboard tasks. At t_4, it passes beyond the ground station's reach and awaits the next contact opportunity.

Communication with satellites is intermittent, with typical contact windows occurring 4–6 times per day, each lasting approximately 10 minutes [23]. Moreover, uplink bandwidth generally ranges from tens to a few hundred kbps [1], [23]–[25]. These impose strict requirements on data transmission to ensure reliable operations within each limited contact window.

B. Key Challenges in Satellite Application Updates

Ensuring effective and reliable satellite application updates is non-trivial due to the following challenges:

• *Challenge 1: Application Heterogeneity and Onboard Resource Constraints.* Satellites can host diverse applications built using different programming languages, e.g., C, C++, and Python. This application heterogeneity complicates the design of a unified software update mechanism. Moreover, satellite systems operate under stringent resource constraints, including limited CPU cycles, memory, and energy. These limitations preclude the use of heavyweight software update methods [15]–[17] (e.g., complex program analysis or large test cases). The limited storage capacity of satellites further constrains the retention of multiple versions of applications.

• *Challenge 2: Limited Communication Bandwidth.* Uplink channels in LEO satellites are fundamentally constrained by low bandwidth and short transmission windows with ground stations. Large application updates can easily exceed the available bandwidth within a single pass, leading to partial transmissions and incomplete deployments. These constraints place greater demands on maximizing communication efficiency to ensure the delivery of critical missions.

• *Challenge 3: Space Environment Risk in Orbit.* The space environment is subject to radiation, extreme temperature fluctuations, and mechanical stress, which could pose threats to application updates. Any failure in update execution or application logic can compromise mission-critical functions, and in extreme cases, lead to permanent satellite malfunction. Consequently, satellite systems impose stringent reliability requirements on any update mechanism.

III. *SateLight*: APPLICATION UPDATE FRAMEWORK

To systematically address the challenges of application updates in satellite-ground systems, we present *SateLight*, a unified framework that enables a lightweight, modular, and reliable software update tailored for satellite computing. The design of *SateLight* is guided by the following three principles.

Design Principle for Challenge 1: *Abstraction to Unify Heterogeneity.* Satellite applications span multiple languages and different configurations, complicating update mechanisms and maintenance logic. Rather than attempting to generalize over heterogeneous toolchains, *SateLight* sidesteps this complexity through application containerization as a unifying abstraction. Each application is encapsulated in a self-contained, portable unit that includes all dependencies (runtime, libraries, configurations), enabling uniform deployment and update workflows. This decouples update logic from application internals, reducing per-application engineering overhead.

Design Principle for Challenge 2: *Differentiation to Minimize Bandwidth Load.* Recognizing the limitations of uplink communication bandwidth, *SateLight* adopts a content-aware differential upload strategy. Instead of transmitting the complete application, *SateLight* analyzes and separates fine-grained semantic code changes between application versions, not file types. *SateLight* identifies these changes at the level of code lines or chunks. These differences are packaged into compact update payloads, reducing data volume and enabling efficient delivery within short communication windows.

Design Principle for Challenge 3: *Reliability Though Reversible Execution.* Space environments are sometimes unreliable, with radiation and power fluctuations posing risks to the update process. *SateLight* incorporates a fault-tolerant recovery mechanism. Instead of relying on fine-grained state reconstruction or re-downloading the entire application, *SateLight* leverages the inherent layered structure of containers, enabling it to rapidly track checkpoints for each application.

Guided by the aforementioned design principles, as illustrated in Fig. 2, *SateLight* implements an application update framework for satellite-ground systems through two core components. The *Application Upload Component*, deployed on the ground, is responsible for performing content-aware differential analysis between the updated and original containerized application versions (Step 1). It identifies semantic changes and encodes them into an expressive metadata representation (Step 2). The *Onboard Update Component*, residing in the satellite system, receives the update package and executes a fine-grained application reconstruction process to ensure update correctness (Step 3). It also integrates a fault-tolerant recovery mechanism based on container layering, enabling rapid application rollback in case of update failures (Step 4).

A. Application Upload Component

This component operates in two main stages: Content-Aware Differential Identification and Application Metadata Representation, as depicted in Step 1 and Step 2 of Fig. 2.

Step 1: Content-Aware Differential Identification. In this phase, *SateLight* conducts a hierarchical comparison between the original and updated container images. Since container images encapsulate applications and their dependencies within layered file systems, they can be unpacked into standard directory hierarchies for analysis. The corresponding workflow is illustrated in Algorithm 1. The process begins with a directory-level delta analysis, which categorizes changes as *added*, *removed*, or *modified* directories (line 1). For each *modified* directory, a recursive file-level delta analysis is triggered, similarly categorizing individual files (line 2). To determine whether a file has been modified, *SateLight* performs content hash comparisons (line 3). Files with identical names but differing hash values are labeled as *modified* and are subjected to type-specific, fine-grained content comparisons (lines 5-17). File types are broadly classified into textual and non-textual (binary) categories, each requiring a distinct analysis method.

For textual files such as Python scripts, we perform a line-level differential analysis, as illustrated in lines 5-9. Each file is abstracted as an ordered sequence of lines, and the differences between file versions are modeled as transformations from one sequence to another. To compute these transformations, we construct an edit graph where each node represents a position in the original and updated sequences, and each diagonal in the graph captures a possible sequence alignment (line 7). The objective is to identify the shortest edit path that converts the original version into the updated one, minimizing the number of insertions required. To optimize performance for large files, our line-level differential analysis adopts a bidirectional divide-and-conquer variant, which concurrently computes forward and reverse search paths until they converge at the midpoint of the edit graph (line 8). Our analysis outputs

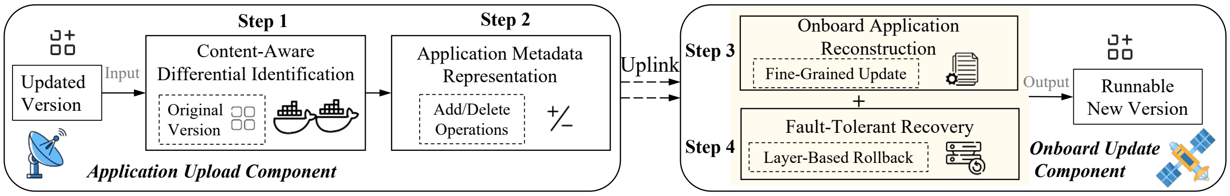

Fig. 2. The satellite application update workflow of *SateLight*.

a sequence of edit operations, indicating insertions, deletions, and retentions, captured as an edit path over line indices.

For non-textual files, e.g., compiled executables or bytecode, direct byte-wise comparison leads to overly fine granularity and inefficiency in delta generation. Even a single-byte insertion or deletion can shift subsequent byte offsets, leading to widespread, non-semantic changes across the entire file. Instead, we adopt a content-defined chunk-level differential analysis (lines 10-17), which computes rolling hash values over a sliding window to detect stable chunk boundaries based on content patterns. When the computed hash satisfies a predefined condition (e.g., a particular number of trailing zero bits, similar to the Rabin Fingerprint method [26]), a chunk boundary is declared (lines 12-13). Each chunk is individually hashed to produce a signature that uniquely represents its content. Our analysis then compares the hash sequences of chunks to identify changed chunks via the similar edit graph construction and path search (lines 14-15). Meanwhile, these chunks are annotated with their corresponding byte numbers, which serve as indicators of content modifications (line 16). This analysis is more effective than direct byte-wise comparison, thereby enabling stable segmentation and minimizing unnecessary recomputation. It outputs a list of changed content chunks, each specified by its byte offset range.

Our hybrid differential strategy, i.e., line-based for textual content and chunk-based for compiled binaries, balances semantic accuracy with computational efficiency, forming the basis for minimal and bandwidth-efficient updates.

Complexity Analysis of Algorithm 1. Let f be the number of changed files, with f_t and f_b denoting the number of modified textual and binary files, respectively. For a textual file with l lines, the bidirectional line-level comparison incurs a time complexity of $O(l \log l)$. For binary files, content chunking takes $O(b)$ time, and chunk comparison adds $O(c \log c)$, where b is the file size in bytes and c is the number of chunks. Thus, the total time complexity is $O(f_t * llogl + f_b * clogc)$. The space complexity is $O(f * \max(l, c))$, accounting for edit graphs, intermediate diffs, and metadata. As the algorithm processes files incrementally without loading full images into memory, it scales with the file number and supports delta extraction.

Step 2: Application Metadata Representation. This phase establishes a unified and expressive metadata representation for the upload package, which captures both structural and semantic changes derived from the differential identification phase (Step 1). Specifically, based on the outputs of Step 1, each change is encapsulated in a structural tuple format, enabling standardized processing. As shown in the Table I, each metadata entry is expressed as a tuple $\langle Type, FilePath, EditOps \rangle$. $Type$ encodes the category and granularity of the change. It includes: Type-D for added or deleted directories, Type-F for newly added or removed files, Type-T for modified textual files, and Type-B for modified binary files. $FilePath$ denotes the file or directory names, whose file locations are relative to the container image. $EditOps$ encodes the specific edit operations. For Type-D and

Algorithm 1: Content-Aware Differential Identification

Input:
I_{orig}: original container image
I_{upd}: updated container image
Output:
$changed\_dirs$: list of changed directories
$changed\_files$: list of changed files
$edit\_paths$: edit paths for same textual files
$chunk\_diffs$: changed chunks for same binary files

1 $changed\_dirs \leftarrow$ `CompareDir`(I_{orig}, I_{upd});
2 $changed\_files \leftarrow$ `CompareFile`(I_{orig}, I_{upd});
3 **foreach** $file$ *in* $changed\_files$ **do**
4   **if** $\mathtt{Hash}(file_{orig}) \neq \mathtt{Hash}(file_{upd})$ **then**
5   **if** $\mathtt{IsTextual}(file)$ **then**
6   `// Textual File Analysis;`
7             $edit\_graph \leftarrow$ `BuildEditGraph`$(file_{orig}, file_{upd})$;
8             $edit\_paths \leftarrow$ `BidirectionalEdit`$(edit\_graph)$;
9   `StoreEditPath`$(file, edit\_paths)$;
10   **else**
11   `// Binary File Analysis;`
12             $chunks_{orig} \leftarrow$ `Chunkify`$(file_{orig})$;
13             $chunks_{upd} \leftarrow$ `Chunkify`$(file_{upd})$;
14             $edit\_graph \leftarrow$ `BuildEditGraph`$(chunks_{orig}, chunks_{upd})$;
15             $chunk\_diffs \leftarrow$ `BidirectionalEdit`$(edit\_graph)$;
16   `ChunkWithByteNum`$(chunk\_diffs)$;
17   `StoreChunkDiffs`$(file, chunk\_diffs)$;
18 **return** $changed\_dirs$, $changed\_files$, $edit\_paths$, $chunk\_diffs$

TABLE I
THE APPLICATION METADATA REPRESENTATION.

Update Type ($Type$)	**Names** ($FilePath$)	**Operations** ($EditOps$)
Type-D	$Directory_1$	I
	$Directory_2$	D
Type-F	$File_1$	I
	$File_2$	D
Type-T	$TextualFile_1$	R l_1, D l_2, I l_3, ...
Type-B	$Non\text{-}textualFile_1$	R c_1, D c_2, I c_3, ...

Type-F, the operations are simplified to atomic Insert (I) and Delete (D) tags. For Type-T, the operations are represented at line-level granularity as an ordered list of instructions: $EditOps$ = { R l_1, D l_2, I l_3, ...}, where R, D, and I denote line retain, delete, and insert, respectively, and each l_i specifies the number of lines affected. For Type-B, the operations are represented at the chunk level { R c_1, D c_2, I c_3, ...}, where each c_i denotes the number of chunks involved in the corresponding operation. The size of each chunk, measured at the byte level, is determined during the chunkification process in Step 1. In particular, with the exception of edit-operation metadata, the actual data of inserted code is extracted and stored separately within the upload package using a naming scheme consistent with the original file. This design ensures deterministic reconstruction during the subsequent onboard update process. To further minimize uplink bandwidth consumption, the upload package is compressed using a standard lossless compression such as Gzip. This exploits the inherent redundancy in source code and textual diffs, reducing the data size and preserving the semantic structure.

B. Onboard Update Component

This component transforms the original application image into its updated version directly on the satellite, leveraging the structured metadata received in the upload package. It comprises Onboard Application Reconstruction and Fault-Tolerant Recovery, as shown in Step 3 and Step 4 of Fig. 2.

Step 3: Onboard Application Reconstruction: During this phase, *SateLight* processes the original application version by interpreting the associated edit operations and performing a fine-grained application reconstruction. For added or deleted directories and files, *SateLight* directly applies the corresponding addition or deletion operations. For each file to be updated, *SateLight* iteratively applies edit instructions, i.e., R (retain), D (delete), and I (insert), over lines or chunks. Insertions are performed using pre-extracted code segments, with file-level consistency preserved via unique identifiers and alignment metadata. Algorithm 2 outlines the onboard reconstruction logic for the updated file. The procedure consumes three inputs: (1) the original content of the file ($init\_file$), (2) the edit operations ($edit\_ops$), and (3) the delta code segment list (seg_list) containing the newly inserted content. It outputs the reconstructed target file ($updated\_file$). First, this algorithm uses two pointers to maintain and traverse the content of the original file and delta code segment list (lines 1–2). For each edit operation, the algorithm advances the file pointer in case of R (lines 4-5), deletes a block of content for D (lines 6-7), and inserts a new segment from seg_list at the current position for I (shown in lines 8-12).

Algorithm 2: Onboard Reconstruction for a File

Input:
$init\_file$: list of original lines/chunks (from the old file version)
$edit\_ops$: sequence of edit operations (op, len)
seg_list: list of added code segments (aligned with insert operations)
Output:
$updated\_file$: reconstructed target file version

1 $index\_init \leftarrow 0$; `// Pointer in` $init\_file$
2 $index\_seg \leftarrow 0$; `// Pointer in` seg_list
3 **foreach** (op, len) *in* $edit\_ops$ **do**
4   **if** $op =$ `'R'` **then**
5     $index\_init \leftarrow index\_init + len$; `// Retain original content`
6   **else if** $op =$ `'D'` **then**
7   delete $init\_file[index\_init : index\_init + len]$; `// Remove content`
8   **else if** $op =$ `'I'` **then**
9     $segment \leftarrow seg\_list[index\_seg]$;
10   insert $segment$ at $init\_file[index\_init]$;
11     $index\_init \leftarrow index\_init + len$;
12     $index\_seg \leftarrow index\_seg + 1$; `// Advance to next segment`
13 **return** $init\_file$ as $updated\_file$

<u>Complexity Analysis of Algorithm 2.</u> Let n denote the number of edit operations, m the number of lines/chunks in the original file, and k the number of inserted segments. The time complexity is $O(n + k)$, since each operation is processed sequentially, and insertions require copying new segments. Retain and deletion operations incur constant-time pointer adjustments. The space complexity is $O(m + k)$, as memory is needed to store the original file and the delta segments.

Step 4: Fault-Tolerant Recovery. This phase incorporates a fault-tolerant recovery mechanism to ensure application availability in the presence of failures. This mechanism automatically detects and responds to failures that may arise either during the update process or post-update execution. Failures during the update process are indicated by non-zero exit codes, which mark unsuccessful updates. Post-update execution failures are detected by an event-triggered mechanism in the container runtime interface that continuously monitors container status. Abnormal terminations with the associated non-zero exit codes enable timely identification of runtime failures. Upon detecting failure events, *SateLight* triggers an immediate rollback procedure without requiring remote assistance from the ground station.

Our recovery strategy leverages the native layered architecture of containers, in which each application version is encapsulated as a discrete image layer. This structure enables version

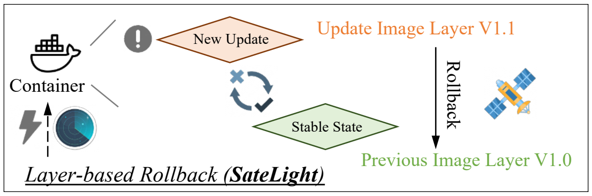

Fig. 3. Fault-tolerant recovery mechanism of *SateLight*.

tracking and facilitates instant restoration. As illustrated in Fig. 3, when an application update (e.g., image layer V1.1) is identified as faulty, *SateLight* rollbacks to the last stable image layer (e.g., V1.0) by replacing the uppermost layer. The rollback is automatically performed and with minimal latency, eliminating the need for re-downloading or reinstallation from the ground station. Compared to image, file, or patch recovery strategies, discussed in experimental evaluation in Section V-C, our layer rollback strategy offers a balanced trade-off between recovery overhead and storage overhead.

C. Prototype Implementation

We implement *SateLight* as a prototype in Python. The design principles of *SateLight* are language-agnostic and applicable to satellite applications across different programming languages. It is built upon the foundational ideas, including dynamic programming, divide-and-conquer strategy, hash computation, and lossless compression, which are extended and adapted to support the analysis requirements of *SateLight*.

IV. Experimental Evaluation

We first present our research questions, followed by the baselines used for comparison and the satellite application dataset. We then describe the experimental settings.

A. Research Questions

RQ1 (Upload Transmission Overhead): *How much can SateLight reduce the upload transmission overhead of satellite applications compared to baselines?* We assess the effectiveness of *SateLight*'s content-aware upload strategy through data size and transmission latency comparisons with baselines.

RQ2 (Onboard Update Overhead): *How is the onboard update overhead of SateLight relative to baselines?* We measure the onboard update latency introduced by the application reconstruction strategy of *SateLight*, relative to baselines.

RQ3 (Recovery Overhead): *How effective is the fault-tolerant recovery mechanism of SateLight during update failures?* We assess the recovery performance overhead of *SateLight*'s layer-based rollback design under update failures.

RQ4 (Real-World Case): *How practical is SateLight for real-world in-orbit satellite deployment?* We validate the onboard update process of *SateLight* via a case study, involving their deployment and execution on an in-orbit satellite.

TABLE II
The representative satellite applications used in this paper.

AppID	Applications	Task description	Languages
App1	Object Detection [6]	It detects the specific objects in an image.	Python
App2	Core Network [7]	It includes network functions for AMF, SMF, and UPF.	C
App3	Image Encoding [6]	It transforms and encodes images.	Python
App4	Client Cache [8]	It implements a read-write cache.	Python, Go
App5	Multi-Stage Image Compression [9]	It implements a five-stage image compression.	MATLAB
App6	Ship Detection [10]	It detects ships from satellite images.	Python
App7	Tracking Algorithm [27]	It implements a feature tracking task.	C++
App8	Data Compression [4]	It reduces data volume via neural-based autoencoders.	Python
App9	Attitude Determination [28]	It estimates the satellite's attitude.	JavaScript
App10	Change Detection and Encoding [29]	It lightly detects and encodes changes.	C, C++

B. Compared Baselines

To the best of our knowledge, no existing application update approaches have specifically addressed the challenges of satellite computing. To evaluate *SateLight*, we compare it against the following potential baselines in this scenario:

- ***Baseline 1 (Image-level approach):*** The entire container image including the updated application version is uploaded and deployed on the satellite. This baseline performs full image transmission without any onboard reuse or processing.
- ***Baseline 2 (Application-level approach):*** Only the application layer within the container image that contains the updated application version is uploaded. This approach reuses unchanged layers, and the new image is reconstructed onboard with the updated layer.
- ***Baseline 3 (File-level approach):*** Within the container image, changed files related to the application are identified. Only these changed files from the new version are collected, compressed, uploaded, and used to replace the original version onboard. This approach avoids the upload of unchanged files.

C. Satellite Application Dataset

We collect 10 representative satellite applications mainly from the published relevant papers, encompassing tasks such as object detection [6], core network functionality [7], and attitude determination [28]. Their detailed descriptions are provided in Table II. The implementations span a range of commonly used satellite-related programming languages, including Python, C, Go, C++, Matlab, and JavaScript. All applications are designed to be stateless, meaning that previous computations and results do not influence the current processing. This design is both reasonable and well-suited for handling raw data generated by satellites [2].

Our software update scope encompasses functional and non-functional changes to satellite application code. Due to the absence of satellite application update datasets, we construct synthetic variants of representative applications exhibiting varying degrees of code modification. In practice, software evolves through version updates that introduce new features or address defects. Empirical studies have quantified the extent of such changes across diverse systems. These studies have shown that version updates generally affect less than approximately 10% or 20% of source components [30]–[32], while more substantial releases may modify more of the codebase with upper bounds generally remaining below 50% [33]. Grounded in these empirical findings, we model application updates using approximately 10% and 20% of code modifications relative to the original application, respectively, and simulate large-scale changes with a modification level of around 50%.

To isolate update effectiveness from application-level bugs, all variants are *task-goal intent equivalent*, preserving high-level task objectives while varying internal implementations. Specifically, we first determine the change ratio, compute the modification size, and then randomly select files and code regions for manual modification. Code updates involve insertions and deletions. Insertions include semantically neutral constructs such as comments, logging statements, inactive conditionals, and unused variables. Deletions target non-functional elements like redundant comments, unused imports, and logs. For compiled languages, comment modifications are excluded from consideration, as they are eliminated during preprocessing and have no effect on the binary. This generation process ensures that the modified applications remain task-goal intent equivalent to their original versions and execute without errors. We validate the executability of these variants through real execution.

To measure the extent of code modifications between two versions of a satellite application, we present a quantitative metric. This metric captures the proportion of the new version's content that differs from the previous version, thereby offering an interpretable indication of update magnitude. It is formally defined as $1 - \frac{S_{preserved}}{S_{upd}} = 1 - \frac{\sum_{i=1}^{N} len_{byte}(u_i)}{S_{upd}}$. $S_{preserved}$ denotes the total size, in bytes, of code segments that are preserved across both the old and new versions of the application. These segments are identified by detecting common subsequences through our differential analyses, which support granularity at both the line and chunk levels of source code. The size is computed as the sum of the byte lengths of the preserved units u_i, with each unit's size given by $len_{byte}(u_i)$. S_{upd} denotes the total size in bytes of the updated version of the application. By computing the relative complement of the retained content within the new version, it effectively captures the extent of code modifications.

D. Experimental Settings

Experimental Environment. To support onboard sensing, computation, and control tasks, the aerospace field increasingly adopts Commercial Off-the-Shelf (COTS) computing devices in satellite systems [6], [34]. The Raspberry Pi 4B industrial module [2], as a representative COTS computing device, has been widely used in satellite research [2], [6], [34] and successfully deployed in multiple in-orbit missions [6], [35], demonstrating its feasibility and operational stability. Thus, to answer RQ1-RQ3, we use the Raspberry Pi 4B as the satellite simulation environment. On top of the hardware, the evaluated applications are containerized and executed in isolated environments using Docker [36], the industry and open-source standard for container deployment. Prior studies [1], [8] have demonstrated the practicality of deploying containerized applications on satellites. To answer RQ4, we deploy the onboard functionality of *SateLight* on a real-world satellite (called BUPT-2) equipped with onboard computing capabilities and operating in a Sun-synchronous orbit. Satellite operations are supported by two ground stations: a telemetry, tracking, and control (TT&C) station, which manages command uplink and satellite control functions; and a data transmission station, responsible for uploading application update packages and downlinking payload data generated onboard.

Evaluation Metrics. We evaluate *SateLight* and baselines using the collected satellite applications with code modifications (about 10%, 20%, and 50%). Each application has three update variants. Evaluation metrics are as follows:

- *Transmission latency*: We estimate this latency based on the specific uplink bandwidth from the ground station to the satellite. Consistent with prior studies that commonly assume uplink bandwidth in the range of hundreds of kbps [1], [23]–[25], we adopt a representative value of 200 kbps, as suggested in the work [25]. The uplink bandwidth is a constant, as the uplink leverages the low-frequency S-band to communicate [2], [37]. It is computed by dividing the total upload package size (in bits) by the uplink bandwidth (in bits per second).
- *Onboard update latency*: This latency is computed as the duration between the initiation of the onboard update process and the completion of the update. It reflects the internal processing time required for onboard application reconstruction.
- *Correctness*: This metric evaluates whether the application has been updated accurately. It identifies consistency between the onboard updated application and the corresponding version validated for successful execution at the ground station.

Experimental Repetitions. Each experiment is repeated ten times, and the mean is reported to mitigate the random impact.

V. Experimental Results

A. RQ1 Results (Upload Transmission Overhead)

To evaluate the transmission overhead of *SateLight*, we compare it against three baselines across satellite applications, each with three levels of code modifications. Table III reports the data size and transmission latency. Results show that *SateLight* consistently outperforms all baselines in both metrics. It achieves transmission latency reduction of up to 100.00%, 99.97%, and 91.18% compared to baselines 1, 2, and 3, respectively. On average, *SateLight* reduces transmission latency by 99.99%, 73.06%, and 56.64%, showing the effectiveness across varied update granularities and application types.

We further analyze the causes of the poor performance observed in the baselines. Baseline 1 incurs the highest overhead, as it uploads the entire container image for every update, irrespective of the actual code change. This results in near-constant data sizes (e.g., about 127,000 KB across all update

TABLE III
RQ1: TRANSMISSION OVERHEAD RESULTS (DATA SIZE AND **TRANSMISSION LATENCY**) OF *SateLight* AND BASELINES.

ID	**Level**	Baseline 1 (B1)		Baseline 2 (B2)		Baseline 3 (B3)		***SateLight***		**Improvement of *SateLight* ↑**		
		Data Size (KB)	Latency (s)	Data Size (KB)	Latency (s)	Data Size (KB)	Latency (s)	Data Size (KB)	Latency (s)	(vs. B1) Latency	(vs. B2) Latency	(vs. B3) Latency
App1	10%	942,701.00	38,613.03	188,845.50	7,735.11	146.96	6.02	48.64	1.99	99.99%	99.97%	66.94%
	20%	943,308.00	38,637.90	189,061.38	7,743.95	331.91	13.60	154.34	6.32	99.98%	99.92%	53.53%
	50%	945,335.00	38,720.92	189,819.63	7,775.01	905.87	37.10	709.26	29.05	99.92%	99.63%	21.70%
App2	10%	127,516.50	5,223.08	11.50	0.47	11.01	0.45	2.24	0.09	100.00%	80.85%	80.00%
	20%	127,517.00	5,224.10	11.69	0.48	11.01	0.45	2.24	0.09	100.00%	81.25%	80.00%
	50%	127,516.50	5,223.08	11.49	0.47	10.82	0.44	3.65	0.15	100.00%	68.09%	65.91%
App3	10%	62,805.50	2,572.51	0.88	0.04	0.67	0.03	0.29	0.01	100.00%	75.00%	66.67%
	20%	62,805.50	2,572.51	0.91	0.04	0.69	0.03	0.34	0.01	100.00%	75.00%	66.67%
	50%	62,805.50	2,572.51	1.00	0.04	0.77	0.03	0.45	0.02	100.00%	50.00%	33.33%
App4	10%	386,661.50	15,837.66	9.77	0.40	8.71	0.36	1.47	0.06	100.00%	85.00%	83.33%
	20%	386,664.00	15,837.76	11.78	0.48	9.79	0.40	1.96	0.08	100.00%	83.33%	80.00%
	50%	386,665.00	15,837.80	12.83	0.53	13.03	0.53	3.08	0.13	100.00%	75.47%	75.47%
App5	10%	2,310,128.50	94,622.86	11.51	0.47	9.97	0.41	1.93	0.08	100.00%	82.98%	80.49%
	20%	2,310,130.50	94,622.95	13.44	0.55	4.46	0.18	3.68	0.15	100.00%	72.73%	16.67%
	50%	2,310,141.00	94,623.38	23.79	0.97	22.62	0.93	13.38	0.55	100.00%	43.30%	40.86%
App6	10%	743,251.50	30,443.58	205.08	8.40	77.76	3.19	32.41	1.33	100.00%	84.17%	58.31%
	20%	743,294.00	30,445.32	247.67	10.14	145.92	5.98	72.12	2.95	99.99%	70.91%	50.67%
	50%	743,540.00	30,455.40	493.39	20.21	412.86	16.91	316.20	12.95	99.96%	35.92%	23.42%
App7	10%	392,131.00	16,061.69	8.61	0.76	16.50	0.68	2.26	0.09	100.00%	88.16%	86.76%
	20%	392,131.00	16,061.69	18.61	0.76	16.52	0.68	4.41	0.18	100.00%	76.32%	73.53%
	50%	392,131.00	16,061.69	18.58	0.76	16.52	0.68	1.35	0.06	100.00%	92.11%	91.18%
App8	10%	265,721.00	10,883.93	2.77	0.11	2.29	0.09	0.98	0.04	100.00%	63.64%	55.56%
	20%	265,721.50	10,883.95	3.16	0.13	2.68	0.11	1.34	0.05	100.00%	61.54%	54.55%
	50%	265,723.50	10,884.03	4.90	0.20	4.65	0.19	3.19	0.13	100.00%	35.00%	31.58%
App9	10%	54,557.50	2,234.68	17.76	0.73	4.13	0.17	2.99	0.12	99.99%	83.56%	29.41%
	20%	54,557.50	2,234.68	17.75	0.72	6.85	0.28	5.57	0.23	99.99%	68.06%	17.86%
	50%	54,575.00	2,235.39	35.14	1.44	21.01	0.86	20.22	0.83	99.96%	42.36%	3.49%
App10	10%	489,989.00	20,069.95	28.95	1.19	26.12	1.07	3.05	0.12	100.00%	89.92%	88.79%
	20%	489,989.00	20,069.95	29.04	1.19	26.19	1.07	4.92	0.20	100.00%	83.19%	81.31%
	50%	489,989.00	20,069.95	28.98	1.19	26.13	1.07	16.01	0.66	100.00%	44.54%	38.32%
Mean										**99.99%**	**73.06%**	**56.54%**

levels for App2) and high transmission latency (up to 5,224.10 seconds). Baseline 2, which uploads only the application layer, reduces some redundancy by avoiding full-image transfer. However, it still sends the entire layer regardless of how minor the internal code edits are. For instance, in App1 (20%), it transmits 189,061.38 KB, a reduction from 943,308.00 KB of baseline 1, resulting in latency saving (7,743.95 vs. 38,637.90 seconds). Baseline 3 improves further by uploading only changed files. It performs well on small updates, for example, in App1 (10%), it transmits 146.96 KB, with a latency of 6.02 seconds. However, its file-level granularity leads to performance degradation as code changes increase. In App1 (50%), it transmits 905.87 KB, which is more than *SateLight*. This is due to its lack of semantic awareness, causing even minor changes to trigger whole-file transfers. In contrast, *SateLight* employs a fine-grained, context-aware differential update, which isolates and transmits only semantically meaningful changes. This results in consistently minimal data transfer across all settings. For instance, in App1 (10%), it uploads only 48.64 KB, achieving 99.97% latency reduction over baseline 2, and 66.94% over baseline 3. In App1 (20%), despite increased update scope, it keeps latency at 6.32 seconds, still outperforming all baselines. Even in large updates, e.g., App1 (50%), the latency of *SateLight* is 29.05 seconds, better than 37.10 seconds of baseline 3.

We also analyze the mean transmission latency reductions achieved by *SateLight* under varying levels of code modification. As illustrated in Fig. 4, *SateLight* consistently outperforms all three baselines across all levels, with particularly significant improvements under small-scale updates. At the 10% update level, *SateLight* achieves mean transmission latency reductions of 100.00%, 83.32%, and 69.63% over baselines 1, 2, and 3, respectively. These reductions highlight the effectiveness of *SateLight*'s fine-grained, content-aware approach, which minimizes unnecessary data transmission by identifying code differences. As the update scope increases, the performance gain gradually decreases, which is expected due to the growing amount of actual content that must be transmitted. Nonetheless, *SateLight* still maintains a considerable advantage. At the 50% modification level, it achieves mean reductions of 99.98% (vs. baseline 1), 58.64% (vs. baseline 2), and 42.53% (vs. baseline 3). This trend also reveals an insight. Baseline 1 is highly sensitive to any update, due to its full-container upload strategy, making the advantage of *SateLight* nearly absolute regardless of update size. Baseline 2 shows moderate resilience at smaller update levels, but suffers from fixed-layer limitations. Baseline 3 benefits from file-level granularity, yet fails to capture similarities in larger changes, leading to degraded performance under heavier updates.

Overall, these results confirm that *SateLight* provides consistent transmission efficiency. This makes it suitable for bandwidth-constrained satellite computing, where reducing

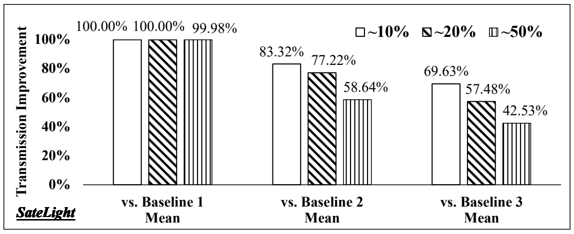

Fig. 4. Mean transmission performance improvement of *SateLight* under varying code modification levels.

uplink time is crucial for timely application updates.

> **Ans. to RQ1:** *SateLight* significantly reduces upload transmission overhead. On average, it outperforms three baselines by 99.99%, 73.06%, and 56.54%, respectively. The most pronounced improvements occur at lower code modification levels, with at least 69.63% mean reductions. These results underscore the effectiveness of *SateLight* in bandwidth-constrained satellite computing.

B. RQ2 Results (Onboard Update Overhead)

To assess the onboard update overhead introduced by *SateLight*, we compare its onboard update latency against three baseline strategies. The results, presented in Table IV, include both the absolute update latency and the difference relative to each baseline. Note that entries marked as "–" indicate no onboard update processing is required, as in baseline 1, which directly executes the full uploaded container image. Despite employing a fine-grained, content-aware update strategy, *SateLight* incurs only minimal onboard overhead, comparable to or slightly higher than baselines 2 and 3, which rely on coarser update mechanisms. On average, *SateLight* increases onboard update latency by 2.67 seconds relative to baseline 2 and 1.27 seconds compared to baseline 3. This marginal overhead confirms the efficiency of the onboard reconstruction mechanism of *SateLight*. For example, in App6 (10%), the latency of *SateLight* is 17.41 seconds, while baselines 2 and 3 report 16.65 seconds and 17.39 seconds, respectively, indicating a negligible difference. Even in high-latency scenarios such as App5 (10%), *SateLight* incurs 136.67 seconds, which is within 16.43 seconds of baseline 2 and 7.33 seconds of baseline 3.

Particularly, *SateLight* achieves 100% update correctness across all cases, as indicated in the "Correctness" column. This ensures that the reconstructed onboard applications are equivalent to their ground-tested counterparts, meeting a fundamental requirement for execution. The use of fine-grained application construction of *SateLight* introduces only minimal overhead while preserving the integrity of the updated content.

> **Ans. to RQ2:** Despite employing a fine-grained reconstruction process, *SateLight* incurs onboard update overhead comparable to the baselines, introducing only an average increase of about 2 seconds. It achieves 100% update correctness across all evaluated applications.

C. RQ3 Results (Recovery Overhead)

The recovery mechanism of *SateLight* is a layer-based rollback, which preserves only the previous application layer and performs quick revision. Leveraging the original application version and fine-grained upload package available onboard, we compare it with three alternative recovery strategies: (1) Image recovery, which retains a complete container image of the previous version and restores it directly without additional computation; (2) File recovery, which reconstructs the prior version by identifying and reversing file-level changes introduced during the update; and (3) Patch recovery, which utilizes recorded reverse edit paths to perform fine-grained recovery. To evaluate their recovery capability in the context of update failures, we analyze their recovery overhead using the satellite application. Recovery overhead consists of two parts: backup time, the duration required to prepare necessary recovery data, and recovery processing time, the time needed to restore the application after a failure. For this evaluation, App1 with a 10% code modification level is selected as the evaluation case.

As shown in Table V, our layer rollback of *SateLight* strikes an effective balance. Specifically, image recovery does not have recovery overhead, but at the cost of high storage demand (942,701.00 KB), which is infeasible in resource-constrained satellites. File and patch recoveries significantly reduce storage usage (82.74 KB and 6.50 K), but suffer from long backup time (over 3,300 ms) and high recovery processing time (over 43,800 ms). Thus, file and patch recoveries are prone to recovery failure, due to the extended duration required for their backup phases. In contrast, our layer recovery uses moderate storage while achieving ultra-fast backup (0.48 ms) and reduced recovery processing time (36,131.41 ms), outperforming file- and patch-based methods in responsiveness and reliability. The efficiency stems from the immediate preservation of only the relevant application layer, enabling timely rollback for most failures. Overall, *SateLight* provides a practical and efficient recovery solution that avoids the high storage of image recovery and the computational burden and unreliability of file and patch recoveries.

> **Ans. to RQ3:** *SateLight* achieves a balanced and efficient fault-tolerant recovery mechanism.

D. RQ4 Results (Real-World Case)

This section presents a real-world case study on an operational LEO satellite called BUPT-2 equipped with onboard cloud-native computing. BUPT-2 supports containerized application deployment and remote communication with ground stations, offering a real-world environment to initially verify the practicability of the onboard update process of *SateLight*.

Due to the limited time we have available to experiment with this satellite, we conduct only one test case. We select a data encryption task, commonly used on satellites [38], [39]. Its initial version is containerized, uploaded, and deployed on the satellite. Subsequently, a corresponding upload package, containing the differential code, is prepared on the ground.

TABLE IV
RQ2: ONBOARD UPDATE OVERHEAD RESULTS (**ONBOARD UPDATE LATENCY**) OF *SateLight* AND BASELINES.

ID	Level	Baseline 1	Baseline 2	Baseline 3	***SateLight***	***SateLight***	**Difference of *SateLight***		
		Onboard Update Latency	Onboard Update Latency (second)	Onboard Update Latency (second)	Onboard Update Latency (second)	Correctness	(vs. B1) Latency	(vs. B2) Latency	(vs. B3) Latency
App1	10%	–	32.26	43.42	44.55	✓	–	+ 12.29	+ 1.13
	20%	–	34.06	46.03	46.47	✓	–	+ 12.41	+ 0.44
	50%	–	34.92	54.29	58.54	✓	–	+ 23.62	+ 4.25
App2	10%	–	1.64	1.69	1.74	✓	–	+ 0.10	+ 0.05
	20%	–	1.64	1.68	1.74	✓	–	+ 0.10	+ 0.06
	50%	–	1.64	1.68	1.90	✓	–	+ 0.26	+ 0.22
App3	10%	–	0.86	0.90	0.89	✓	–	+ 0.03	- 0.01
	20%	–	0.86	0.90	0.90	✓	–	+ 0.04	0.00
	50%	–	0.86	0.90	0.90	✓	–	+ 0.04	0.00
App4	10%	–	5.13	5.18	5.17	✓	–	+ 0.04	- 0.01
	20%	–	5.11	5.16	5.17	✓	–	+ 0.06	+ 0.01
	50%	–	5.21	5.19	5.32	✓	–	+ 0.11	+ 0.13
App5	10%	–	120.24	129.34	136.67	✓	–	+ 16.43	+ 7.33
	20%	–	121.79	117.15	128.23	✓	–	+ 6.44	+ 11.08
	50%	–	122.74	116.02	128.41	✓	–	+ 5.67	+ 12.39
App6	10%	–	16.65	17.39	17.41	✓	–	+ 0.76	+ 0.02
	20%	–	16.60	16.66	16.86	✓	–	+ 0.26	+ 0.20
	50%	–	16.07	16.25	16.65	✓	–	+ 0.58	+ 0.40
App7	10%	–	4.95	5.04	5.10	✓	–	+ 0.15	+ 0.06
	20%	–	5.02	5.12	5.11	✓	–	+ 0.09	- 0.01
	50%	–	5.03	5.07	5.11	✓	–	+ 0.08	+ 0.04
App8	10%	–	3.40	3.45	3.46	✓	–	+ 0.06	+ 0.01
	20%	–	3.44	3.47	3.48	✓	–	+ 0.04	0.00
	50%	–	3.47	3.51	3.51	✓	–	+ 0.04	0.00
App9	10%	–	0.76	0.81	0.81	✓	–	+ 0.05	0.00
	20%	–	0.81	0.82	0.82	✓	–	+ 0.01	0.00
	50%	–	0.77	0.82	0.82	✓	–	+ 0.05	0.00
App10	10%	–	6.36	6.29	6.30	✓	–	- 0.06	+ 0.01
	20%	–	6.45	6.49	6.48	✓	–	+ 0.03	- 0.01
	50%	–	6.37	6.49	6.82	✓	–	+ 0.45	+ 0.33
Mean								**+ 2.67**	**+ 1.27**

TABLE V
A COMPARATIVE ANALYSIS OF RECOVERY MECHANISMS.

Recovery Mechanisms	**Storage Size**	**Backup Time**	**Recovery Processing Time**
Image Recovery	942,701.00 KB	–	–
File Recovery	82.74 KB	3,331.36 ms	43,879.51 ms
Patch Recovery	6.50 KB	3,320.74 ms	43,845.99 ms
Ours	188,845.49 KB	**0.48 ms**	36,131.41 ms

Along with this package, we upload the application reconstruction script implementing *SateLight*'s onboard update strategy, as well as the necessary control script. Upon receipt, the satellite executes the reconstruction script to perform a fine-grained application reconstruction process for the data encryption task entirely onboard. During in-orbit testing, we collaborate with satellite operators to collect and analyze telemetry data. The "successCount" field, tracking successfully executed remote control commands, served as an indirect yet reliable indicator of application correctness and executability post-update. Furthermore, operators explicitly confirm the update's success and verify normal satellite system operation. The results demonstrate the practicality of *SateLight* for satellite application updates in real satellite environments.

> **Ans. to RQ4:** *SateLight* shows the practicality of updating satellite applications on real-world in-orbit satellites.

VI. THREATS TO VALIDITY

Measurement Bias. Performance evaluation, including transmission latency and onboard update latency, may be influenced by communication variability and onboard resource fluctuations. To reduce this, we assume a constant uplink bandwidth, as satellite S-band communication is stable and predictable [2], [37], making transmission latency effectively deterministic. For onboard update latency, we report the mean over multiple runs [40], [41] as the final result of each experiment to reduce the impact of transient variations and improve result reliability.

Conclusion Generalizability. The generalizability of conclusions drawn from *SateLight* and baselines may be influenced by variations in satellite application types and languages. To address this, we evaluate representative applications covering a broad spectrum of satellite tasks, e.g., object detection and tracking, core network services, and image encoding. These applications are implemented in diverse languages, e.g., Python, C, Go, C++, JavaScript, and mixed languages, thereby reflecting the application heterogeneity. Thus, our conclusions are grounded in diverse empirical evidence and are reasonably generalizable across satellite application scenarios.

VII. RELATED WORK

Satellite Computing Recent advances in satellite computing have focused on enhancing onboard computational capabilities to support increasingly complex satellite applications.

Komet [8] was a serverless edge computing platform tailored for LEO satellites, enabling on-demand function execution and dynamic service migration through hardware abstraction. Xing *et al.* [6] analyzed how thermal control and power constraints affect task scheduling on COTS devices in orbit. SECO [9] was a collaborative edge computing framework to coordinate real-time tasks, such as image compression and object detection, across multiple satellites for low-latency Earth observation. Phoenix [10] was an energy-ware scheduling framework that leveraged sunlit orbital periods to opportunistically perform onboard processing. To alleviate bandwidth constraints, Maskey and Cho [42] designed a lightweight convolutional neural network for onboard image classification, reducing unnecessary data transmission. The Tiansuan constellation [43] employed KubeEdge to enable satellite-ground collaborative inference, improving application responsiveness. While these efforts advance various aspects of satellite computing, little attention has been given to the approaches for updating and evolving satellite applications. In this work, we address this gap by presenting *SateLight*.

Software Update Research on software updates has been categorized into three primary phases: pre-update, in-update, and post-update. Prior research on pre- and post-update phases has primarily focused on update planning and impact assessment. In the pre-update stage, efforts have centered on recommending *what* to update. Techniques include mining user feedback [11], [12], recommending configuration updates [13], and leveraging large language models for code change suggestions [14]. However, these approaches focus on update recommendation rather than execution. Most assume a ground-based context, where direct version replacement is feasible, ignoring the operational constraints present in resource-limited satellites. In the post-update stage, studies have addressed user behavior and update effectiveness. This includes analyzing the influence of automated pull requests [18], detecting faulty updates [19], evaluating update interval strategies [20], and characterizing update patterns [21]. While valuable, these studies address different concerns and do not tackle the unique constraints of software evolution in satellites.

Studies on in-update have explored dynamically how to apply updates safely and efficiently, i.e., dynamic software update (DSU). For instance, Zhao *et al.* [15] leveraged test cases to identify safe update points in Java programs, albeit with high computational cost. Cazzola and Jalili [16] proposed an annotation-driven framework to detect unsafe update points and dynamically avoid them. MCR [17] supported live updates in C applications by automating control transfer and state migration through static instrumentation and thread coordination. However, DSU techniques pursue incremental, in-place updates during program execution, fundamentally differing from our goal of stable application replacement on satellites, which are infrequent, planned, communication-limited, and ground-controlled. Additionally, DSU relies on language-specific analysis, runtime monitoring, and extensive test suites, making it impractical for resource-constrained satellites. In contrast, *SateLight* employs a lightweight content-aware update framework that avoids such overhead and addresses unique challenges to support diverse satellite applications.

VIII. Conclusion

This paper presented *SateLight*, an effective framework for updating satellite applications under the constraints of satellite computing. *SateLight* adopted a container-based design for heterogeneous applications. Moreover, it enabled content-aware differential updates, onboard application reconstruction, and fault-tolerant recovery. Evaluation in a satellite simulation shows that *SateLight* significantly reduces transmission latency by up to 91.18%, with an average reduction of 56.54%, compared to the best-performing baseline. It also ensures 100% correctness in application updates. A real-world deployment on an operational in-orbit satellite confirms the applicability of *SateLight* for satellite software maintenance and evolution.

Acknowledgment

This work is supported by the China Postdoctoral Science Foundation under Grant No. 2025M771560, Postdoctoral Fellowship Program (Grade B) of China Postdoctoral Science Foundation under Grant No. GZB20250400, National Natural Science Foundation of China under Grant Nos. U21B2016, 62425203, and 62032003, and Fundamental Research Funds for the Central Universities under Grant No. 2024ZCJH11.

References

[1] S. Wang and Q. Li, "Satellite computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 10, no. 24, pp. 22 514–22 529, 2023.

[2] Q. Zhang, X. Yuan, R. Xing, Y. Zhang, Z. Zheng, X. Ma, M. Xu, S. Dustdar, and S. Wang, "Resource-efficient in-orbit detection of earth objects," in *Proceedings of the IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2024, pp. 551–560.

[3] Y. Ren, A. Gamage, L. Liu, M. Li, S. Chen, Y. Dong, and Z. Cao, "Sateriot: High-performance ground-space networking for rural iot," in *Proceedings of the Annual International Conference on Mobile Computing and Networking*, 2024, pp. 755–769.

[4] Q. Ouyang, N. Ye, J. Gao, A. Wang, and L. Zhao, "Joint in-orbit computation and communication for minimizing download time from leo satellites," *IEEE Transactions on Mobile Computing*, vol. 23, no. 5, pp. 3950–3963, 2023.

[5] I. Del Portillo, B. G. Cameron, and E. F. Crawley, "A technical comparison of three low earth orbit satellite constellation systems to provide global broadband," *Acta astronautica*, vol. 159, pp. 123–135, 2019.

[6] R. Xing, M. Xu, A. Zhou, Q. Li, Y. Zhang, F. Qian, and S. Wang, "Deciphering the enigma of satellite computing with cots devices: Measurement and analysis," in *Proceedings of the Annual International Conference on Mobile Computing and Networking*, 2024, pp. 420–435.

[7] R. Xing, X. Ma, A. Zhou, S. Dustdar, and S. Wang, "From earth to space: A first deployment of 5g core network on satellite," *China Communications*, vol. 20, no. 4, pp. 315–325, 2023.

[8] T. Pfandzelter and D. Bermbach, "Komet: A serverless platform for low-earth orbit edge services," in *Proceedings of the ACM Symposium on Cloud Computing*, 2024, pp. 866–882.

[9] Z. Zhai, L. Zeng, T. Ouyang, S. Yu, Q. Huang, and X. Chen, "Seco: Multi-satellite edge computing enabled wide-area and real-time earth observation missions," in *Proceedings of the IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2024, pp. 2548–2557.

[10] W. Liu, Z. Lai, Q. Wu, H. Li, Q. Zhang, Z. Li, Y. Li, and J. Liu, "In-orbit processing or not? sunlight-aware task scheduling for energy-efficient space edge computing networks," in *Proceedings of the IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2024, pp. 881–890.

[11] A. Di Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall, "What would users change in my app? summarizing app reviews for recommending software changes," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 499–510.

[12] Y. Zhou, Y. Su, T. Chen, Z. Huang, H. Gall, and S. Panichella, "User review-based change file localization for mobile applications," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2755–2770, 2020.

[13] F. Hassan, R. Rodriguez, and X. Wang, "Rudsea: recommending updates of dockerfiles via software environment analysis," in *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 796–801.

[14] Y. Liu, C. Tantithamthavorn, Y. Liu, P. Thongtanunam, and L. Li, "Automatically recommend code updates: Are we there yet?" *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–27, 2025.

[15] Z. Zhao, X. Ma, C. Xu, and W. Yang, "Automated recommendation of dynamic software update points: An exploratory study," in *Proceedings of the Asia-Pacific Symposium on Internetware*, 2014, pp. 136–144.

[16] W. Cazzola and M. Jalili, "Dodging unsafe update points in java dynamic software updating systems," in *Proceedings of the IEEE International Symposium on Software Reliability Engineering*. IEEE, 2016, pp. 332–341.

[17] C. Giuffrida, G. Tamburrelli, and A. S. Tanenbaum, "Automating live update for generic server programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 207–225, 2016.

[18] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?" in *Proceedings of the International Conference on Automated Software Engineering*. IEEE, 2017, pp. 84–94.

[19] I. Saidani, A. Ouni, M. Ahasanuzzaman, S. Hassan, M. W. Mkaouer, and A. E. Hassan, "Tracking bad updates in mobile apps: A search-based approach," *Empirical Software Engineering*, vol. 27, no. 4, p. 81, 2022.

[20] G. Di Tizio, M. Armellini, and F. Massacci, "Software updates strategies: A quantitative evaluation against advanced persistent threats," *IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 1359–1373, 2022.

[21] P. Kotzias, L. Bilge, P.-A. Vervier, and J. Caballero, "Mind your own business: A longitudinal study of threats and vulnerabilities in enterprises." in *Proceedings of the Symposium on Network and Distributed System Security*, 2019, pp. 151–164.

[22] "Supplemental material," https://github.com/WenJinfeng/SateLight.

[23] B. Tao, M. Masood, I. Gupta, and D. Vasisht, "Transmitting, fast and slow: Scheduling satellite traffic through space and time," in *Proceedings of the Annual International Conference on Mobile Computing and Networking*, 2023, pp. 1–15.

[24] K. Devaraj, M. Ligon, E. Blossom, J. Breu, B. Klofas, K. Colton, and R. Kingsbury, "Planet high speed radio: Crossing gbps from a 3u cubesat," in *Proceedings of the Small Satellite Conference*, 2019.

[25] D. Vasisht, J. Shenoy, and R. Chandra, "L2d2: Low latency distributed downlink for leo satellites," in *Proceedings of the ACM SIGCOMM 2021 Conference*, 2021, pp. 151–164.

[26] "Rabin fingerprint," https://en.wikipedia.org/wiki/Rabin_fingerprint.

[27] N. Tsog, M. Behnam, M. Sjödin, and F. Bruhn, "Intelligent data processing using in-orbit advanced algorithms on heterogeneous system architecture," in *Proceedings of the IEEE Aerospace Conference*. IEEE, 2018, pp. 1–8.

[28] "Attitude determination," https://github.com/MieszkoGulinski/infrared-camera-orientation-sensing.

[29] K. Du, Y. Cheng, P. Olsen, S. Noghabi, and J. Jiang, "Earth+: On-board satellite imagery compression leveraging historical earth observations," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2025, pp. 361–376.

[30] "Released code percentage," https://lwn.net/Articles/374574/.

[31] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, pp. 2146–2189, 2016.

[32] S. Koch, "Software evolution in open source projects—a large-scale investigation," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 6, pp. 361–382, 2007.

[33] I. Neamtiu, G. Xie, and J. Chen, "Towards a better understanding of software evolution: an empirical study on open-source software," *Journal of Software: Evolution and Process*, vol. 25, no. 3, pp. 193–218, 2013.

[34] Y. Chen, Q. Zhang, R. Xing, Y. Li, X. Ma, C. Yu, Y. Zhang, A. Zhou, and S. Wang, "Energy-aware satellite-ground co-inference via layer-wise processing schedule optimization," in *Proceedings of the Asia-Pacific Symposium on Internetware*, 2024, pp. 303–312.

[35] "Gaspacs overview," https://www.usu.edu/physics/gas/projects/gaspacs.

[36] "Docker," https://www.docker.com/.

[37] "Esa - sentinel-2 operations," https://www.esa.int/Enabling_Support/Operations/Sentinel-2_operations.

[38] I. Ingemarsson and C. Wong, "Encryption and authentication in on-board processing satellite communication systems," *IEEE Transactions on Communications*, vol. 29, no. 11, pp. 1684–1687, 2003.

[39] R. Banu and T. Vladimirova, "Fault-tolerant encryption for space applications," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 45, no. 1, pp. 266–279, 2009.

[40] X. Liu, J. Wen, Z. Chen, D. Li, J. Chen, Y. Liu, H. Wang, and X. Jin, "Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–29, 2023.

[41] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proceedings of the USENIX annual technical conference*, 2018, pp. 133–146.

[42] A. Maskey and M. Cho, "Cubesatnet: Ultralight convolutional neural network designed for on-orbit binary image classification on a 1u cubesat," *Engineering Applications of Artificial Intelligence*, vol. 96, p. 103952, 2020.

[43] S. Wang, Q. Li, M. Xu, X. Ma, A. Zhou, and Q. Sun, "Tiansuan constellation: An open research platform," in *Proceedings of the IEEE International Conference on Edge Computing*. IEEE, 2021, pp. 94–101.