

August 10, 2026

To:

Jacopo Soldani, University of Pisa, Italy

Nour Ali, Brunel University London, UK

Vasilios Andrikopoulos, University of Groningen, the Netherlands

Cesare Pautasso, University of Lugano, Switzerland

Guest Editors, JSS Special Issue on “Impactful Software Architecture”

Dear Editors, Dear Reviewers,

We are grateful to the reviewers for the highly constructive feedback, which has genuinely helped us improve our work’s clarity and overall contribution. Please find enclosed the revised version of our manuscript, “On Architectural Analysis for Nanosatellite Flight Software: Deconstructing ERMIS3 per 4+1.”

We have carefully addressed all reviewer comments in this revision. Below is a summary of the most significant improvements:

- We significantly expanded the final section of the paper by restructuring it into a comprehensive, bulleted list that details an emerging research agenda for dependable and reusable flight software. We have also added a note in the Acknowledgments section to explicitly recognize the reviewers’ input in shaping this agenda.
- We updated the text to map our chosen architectural design drivers clearly to the ISO/IEC 25010 standard. We also added a clarifying paragraph detailing why other discussed quality attributes were not elevated to primary architectural drivers for this specific context.
- We have added a list in Section 4 defining the various architectural styles (e.g., layered, service-oriented, monolithic) as they specifically apply to the flight software domain.

A detailed, point-by-point response to all reviewer comments, alongside the corresponding changes made in the text, is enclosed in the accompanying “Response to Reviewers” document. We have corrected minor changes noted during the review process—core changes made have been highlighted in teal color.

We believe these revisions have substantially strengthened our experience report, making it a more comprehensive and instructive resource for the New Space community. We hope that the revised manuscript now meets the rigorous standards of your journal and is suitable for publication.

Kind regards,

Christoforos Vasilakis

Alexandros Tsagkaropoulos

Angelos Motsios

Panagiotis Karathanasis

Christos Tsigkanos

Dionysios Reisis

Response to Reviewers

Reviewer Comment 1: *My only major point of criticism is that I would have liked a more detailed account of the novel research agenda for FSW. The last sentence of the abstract mentions that the overall goal of the experience report is to “advance a novel research agenda for dependable and reusable flight software.” Currently, such an emerging agenda is only discussed in a single paragraph at the end of the Conclusions. I agree with the three points mentioned there. I believe however that they can be elaborated more and more points can and have to be provided to the readers and to the community to better help advance this novel agenda. To be constructive, some ideas:*

1. *How do you envision the process of creating a reference architecture for FSW?*
2. *How can the formal verification techniques be combined with existing testing methods for FSW?*
3. *Which other properties (e.g. runtime flexibility/adaptability) are or may be in the near future of interest for research?*
4. *Is there room for other frameworks apart from F' ? Are there are notable and important features that F' is currently missing?*
5. *Would it be reasonable to apply standard architecture evaluation methods (e.g. ATAM) in the evaluation of the proposed and of other architectures in the space domain?*

Authors' Response: We sincerely thank the reviewer for this highly constructive feedback. After the points raised by the anonymous reviewer, the novel research agenda for FSW can be further detailed and structured to benefit the community. We appreciate the concrete, five points provided as guidance and we recognized this contribution in the manuscript's Acknowledgments.

We have revised the part in question into a comprehensive, multi-point research agenda targeting the ECSA community. Below is a detailed breakdown of how we addressed each of your suggestions:

1. Regarding the vision for a reference architecture, we have expanded our manuscript to outline a multi-phase approach. We draw parallels to successful reference architectures in other domains such as embedded and IoT systems and anchor this vision to formal standardization practices, specifically addressing ISO/IEC/IEEE 42010.
2. While formal verification has a proven track record in the domain, indeed one may rethink a combination leveraging the merits of testing. While this is out of scope for this paper, we acknowledge the potential that this can bring. We indicated a pathway proposing the use of property-driven testing pipelines that may be feasible: formally verified properties (such as FSM safety conditions) could be translated into executable test oracles, which can then function as runtime monitors during dynamic system testing.
3. We highlighted the value of a possible evolution from static, FSM-driven logic toward runtime adaptability. The revised text specifically emphasizes the need for future research into dynamic Fault Detection, Isolation, and Recovery (FDIR) policy updates and autonomous operational planning that reduce reliance on continuous ground intervention.
4. Regarding the potential for other frameworks and current limitations in F' , we now explicitly discuss existing operational bottlenecks. The manuscript highlights missing features in current solutions, such as the lack of dynamic telemetry monitoring (which forces static threshold definitions) and the absence of native visual state machine integration. Consequently, we advocate for exploring alternative solutions, including dynamic, microkernel-style architectures and memory-safe languages for next-generation middleware.

5. Application of standard architecture evaluation methods like ATAM are certainly applicable; they represent a valuable avenue for future research. While such comprehensive frameworks provide systematic quantification of design trade-offs, their stringent procedural nature can be difficult to sustain in the space domain, especially when attempting to track non-functional requirements throughout an iterative engineering lifecycle. Therefore, as detailed in the revised manuscript, we suggest that future research investigate lightweight alternatives—such as the Lightweight Architecture Evaluation (LAE) method—capable of assessing architectural robustness without impeding the accelerated development velocity required by modern missions.

Reviewer Comment 2: *One doubt I had when reading the introduction was whether the authors actually developed the architecture and the FSW for the ERMIS3 mission themselves or not. It would be nice to clarify this early on.*

Authors' Response: We were the core team responsible for the onboard flight software of the ERMIS mission (launched in March aboard SpaceX's Transporter 16; currently at LEOP operations). We have updated the second point in the contributions list of the Section 1 to make this clear early on.

Reviewer Comment 3: *It would be nice to provide some more information and potentially some references regarding the ERMIS3 mission. Section 2 excludes mission particularities. I disagree with this decision, I think it would be instructive to also mention and briefly describe the mission-specific requirements, separately from the mission agnostic ones that are already present. This will provide a more holistic picture to the reader.*

Authors' Response: We agree that explicitly stating the functional, mission-specific requirements in Section 2 provides a much better holistic context for the reader before diving into the abstracted quality attributes; we revised the first paragraph of Section 2 explicitly outlining the core functional requirements of the ERMIS3 flight software.

Reviewer Comment 4: *"Key quality attributes were identified through consultations with mission partners, hardware-providers and stakeholders." → Which were the partners, providers and stakeholders? Again, more context would be appreciated here.*

Authors' Response: We thank the reviewer for pointing this out. To clarify, the ERMIS project is coordinated by the University of Athens, with mission partners including the University of Patras, the University of the Aegean, the National Observatory of Athens, and OQ Hellas. The hardware providers consulted included GomSpace (for the primary satellite bus hardware), CubeSpace (for the ADCS), and Astrolight (for the laser optical terminal). Primary stakeholders are the European Space Agency (ESA) and the Greek Ministry of Digital Governance. We have updated Step 1 in Section 2 of the manuscript to include these entities.

Reviewer Comment 5: *Were there other quality attributes considered/discussed apart from the chosen four?*

Authors' Response: During the stakeholder consultation process described in step 1 of Section 2, additional quality attributes were discussed, namely formulations of (notably) reliability

(including fault tolerance and recoverability) and *performance efficiency* (as specified in 25010 regarding time behavior and resource utilization), which are considered foundational in the space domain—virtually any major mission considers them. We opted for analysing and discussing major architectural drivers beyond typical reliability. We added a clarifying paragraph at the end of Section 2 that explicitly outlines the reasons why these attributes were not elevated to primary architectural design drivers.

Reviewer Comment 6: *Were there other architectural documentation frameworks considered apart from the 4+1?*

Authors' Response: During the early design phases, we did evaluate alternative frameworks, such as the arc42 and the C4 model. We selected Kruchten's 4+1 model because it provides dedicated views for the primary constraints of embedded flight software, specifically task concurrency (Process View) and hardware mapping (Physical View), all of which are tightly integrated by the overarching Scenario View. In C4, by contrast, these runtime and deployment concerns appear only as optional supplementary diagrams, with no dedicated view for concurrency or real-time scheduling. Meanwhile, arc42 is primarily a 12-section documentation template; it covers these topics, but as a comprehensive checklist of sections rather than a set of views unified by a single integrating scenario view, as in the 4+1 model. To this end, we have added a brief citation regarding the necessity of utilizing multiple architectural views in the first paragraph of Section 3.

Reviewer Comment 7: *"The flight software system was deconstructed and mapped to the views to clearly represent structure and behavior, and used for validating the design." → This was done before the implementation took place, correct? If so please consider rephrasing since the "deconstruction" part threw me off since you can only deconstruct something that is already in place.*

Authors' Response: We thank the reviewer for pointing out this semantic nuance. Indeed, the architectural mapping and validation process took place during the design phase, prior to implementation. We have revised the manuscript to use the term "decomposed" and adjusted the phrasing (step 4 in Section 2): "The flight software system as an early design concept was logically decomposed and mapped to the views to clearly represent its intended structure and behavior, serving as a basis for validating the design prior to implementation."

Reviewer Comment 8:

- *"Modifiability (D3) addresses the need for in-flight updates and feature enhancements, crucial for extending mission lifecycles or responding to unforeseen circumstances" → can the FSW really be updated at runtime? I think this attribute is a bit loaded since as far as I understand it covers both offline ease of changes and runtime flexibility/reaction to faults. Please consider splitting this into two attributes.*
- *The alignment of the mentioned design drivers to the quality attributes of the ISO 25010 standard is now partial and only mentioned at the end of Section 2. Please consider mapping each driver and also moving the discussion of the last paragraph in the bullet points above, I believe it would enhance the reading flow.*

Authors' Response: We thank the reviewer for these constructive observations regarding the design drivers. Because both comments pertain to the definition and presentation of these drivers in Section 2, we have addressed them together:

- Regarding the ambiguity raised regarding Modifiability (D3), we clarify now that the FSW does not support runtime hot-swapping without a reboot. Instead, a new compiled binary image is uploaded to the OBC filesystem, checked and if successful a reboot is scheduled. The OBC bootloader is then instructed to initialize the system using this new binary image, thereby completing the firmware update. A golden image is loaded if the process fails. Modifiability is thus a key driver requiring careful attention in the architectural design (FileUplink component et al). Per the 25010, Modifiability is defined as the capability of a product to be modified “effectively and efficiently” without introducing defects or degrading quality. The ease and safety of making these offline changes is what ultimately enables safe software updates during the mission lifecycle. We revised the (D3) driver description in Section 2 to better align with the standard’s definition, removing the loaded phrasing identified.
- Regarding the partial mapping, we clarify that the terms Modifiability and Testability were not explicitly mapped in the original draft because they are already direct sub-characteristics of the ISO 25010 standard (under Maintainability). By contrast, Deployability and Integrability are domain-specific architectural drivers that required explicit mapping to the standard’s Flexibility and Interoperability. However, we recognize that leaving this unstated caused ambiguity. We thus restructured the list (in Section 2) as requested and explicitly noted the ISO 25010 alignment for all four drivers.

Reviewer Comment 9: *Regarding the component diagram of the process view, is it true that passive components have no output ports?*

Authors’ Response: Passive components actually can and do possess output ports. In the F² framework, passive components do not enjoy having their own active execution thread and message queue. Therefore, while their input ports are limited to synchronous execution (operating on the caller’s thread), they are entirely capable of invoking output ports. This is also illustrated in the component diagram (Figure 5), where passive components such as Framer, Deframer, and FileUplink possess output ports (denoted by the white port boxes). To prevent any potential confusion for future readers, we have slightly expanded the explanation in the first paragraph of Section 3.4 to explicitly clarify that passive components can utilize output ports.

Reviewer Comment 10: *“As an active component, PrmDb processes the new mode data using its execution thread to update the parameter representing the current operational mode in FRAM storage; it then notifies the Controller thread of the result of the mode change.” → Such notification is not shown in Figure 6.*

Authors’ Response: We thank the reviewer for their attention to detail. You are entirely correct. The notification was inadvertently omitted from the sequence diagram during the drafting process. We have updated the sequence diagram in Figure 6.

Reviewer Comment 11: *What is the impact of the architectural workaround mentioned at the end of Section 3 in maintainability of the system and the reusability of the architecture?*

Authors’ Response: We are grateful to the reviewer for highlighting this important architectural consideration—a valid concern is indeed raised. In the design, the impact on maintainability and reusability is heavily mitigated through strict encapsulation. The original manuscript details in the second paragraph of Section 3.2, that both the standard CSP networking and the proxy functionality are encapsulated entirely within the CommCSP component. Because this solu-

tion is isolated in a single component, maintainability is preserved. Furthermore, by leveraging the Proxy pattern, the CommCSP component establishes a reusable architectural asset for future missions that must integrate black-box COTS components with similar routing or timing constraints. We have updated the final paragraph of Section 3.5 to explicitly address how this encapsulation preserves system maintainability and reusability.

Reviewer Comment 12: *Regarding Section 3 as a whole, I was wondering whether a brief overview of F' before that section (in a potential Background one) and a showcase of the code/pseudocode of an F' component (e.g. from Figure 5) would make the illustration even more complete and instructive.*

Authors' Response: We thank the reviewer for this suggestion. We carefully considered adding a dedicated background section and code artifacts. Ultimately, we chose to omit these elements to maintain a strict focus on the paper's primary objective: an architectural analysis using the 4+1 view model. Because F' is a mature, well-documented, open-source framework developed by NASA JPL, detailing its core mechanics or background would duplicate existing literature. Similarly, introducing concrete code or pseudocode shifts the focus away from structural patterns and design rationales into implementation details. Instead, we opted to cite core literature such as the foundational F' paper (Bocchino et al., 2018) that provides further context on the framework.

Reviewer Comment 13: *I understand that the choice of literature in Section 4 was not systematic. This is not ideal. At the very least, I would expect a better justification of why the particular papers were found and included in the analysis and not others.*

Authors' Response: We completely agree with the reviewer that a careful and systematic literature investigation is highly valuable and provides an excellent resource for the community. However, the guidelines for this specific Special Issue do not allow the submission of systematic literature reviews. Moreover, because this manuscript is fundamentally structured as an experience report, conducting a full systematic review falls outside its intended scope. As we noted in Section 4 of the manuscript, "This investigation does not aim for completeness; we aim to analyze key missions in a similar mission class as ERMIS, for which software architectural information is available. The list was compiled through extensive consultation with domain experts and flight software engineers; we defer a systematic literature study for future work." We have slightly expanded this text in the first paragraph of Section 4 to make this justification clearer.

Reviewer Comment 14: *The description of the drivers in Table 1 follows the ISO 25010 nomenclature, as also acknowledged later in the Related Work. I suggest to mention this in the caption of the Table, too.*

Authors' Response: We thank the reviewer for this excellent suggestion. We have updated the caption for Table 1 to explicitly state that the extracted requirements are mapped according to the ISO/IEC 25010 standard.

Reviewer Comment 15: *Although a reader can intuitively grasp most of the differences between the architectural styles mentioned in Table 1, I suggest to define each term. Doing this for the FSW domain is interesting in itself, but would also help the reader grasp the differences better.*

Authors' Response: We thank the reviewer for this constructive suggestion. We have expanded the paragraph of the “Architectural Style” discussion in Section 4 that defines what an architectural style is, and a list at the end of the section that defines the remaining terms as they apply to the FSW domain based on both established software architecture literature and the reviewed publications themselves.

Reviewer Comment 16: *An overall question I had at this point in the paper is: doesn't the main motivation of the work, the lack of documented FSW architectures, contradict with the wealth of information shown in Table 1? An explanation is needed for this point.*

Authors' Response: We thank the reviewer for raising this point; while it is true that Table 1 presents a wealth of architectural data, we must point out that there exists a fundamental distinction between a publication briefly mentioning an architectural style and a publication providing comprehensive architectural documentation. As noted in Section 4 (in the Architectural Drivers paragraph), some of the data in Table 1 had to be “implicitly extracted” or inferred from the text of the publication. In the vast majority of these papers, software architecture aspects are minimally discussed. While there are a few notable exceptions that discuss FSW architecture in depth (such as Gonzalez et al., 2019 and de Souza et al., 2022), comprehensive, multi-view documentation remains exceptionally rare in the domain. To prevent this perceived contradiction for future readers, we have updated the first paragraph of Section 4.

Reviewer Comment 17: *In Section 5.1, the discussion of testability is too broad. I suggest to elaborate e.g. on the “F' primitives and workflows to efficiently support structured validation” and the “rigorous verification protocols inspired by common practice.”*

Authors' Response: We thank the reviewer for this feedback. To address this, we have expanded the last paragraph of Section 5.1 to elaborate on both points.

First, regarding the F' primitives, we clarified that the framework provides automated test scaffolding and mock interfaces. This allows to isolate components, inject commands, and programmatically assert expected telemetry and event outputs, efficiently streamlining unit-level validation.

Second, regarding verification protocols, we clarified that verification protocols of the European Cooperation for Space Standardization (2025) were followed, as interpreted within the mission context (e.g., structural code coverage tracking, phased progression from software-in-the-loop to hardware-in-the-loop, etc). Utilizing automated simulation environments alongside continuous integration streamlined testing across different stages of development.

Reviewer Comment 18: *In Section 5.2, I was wondering, was any other framework considered or was it a decision between using F' and not using any framework (as many of the works listed in Table 1)?*

Authors' Response: We thank the reviewer for this question, whose answer we did not make sufficiently clear in the original version. Other established flight software frameworks were indeed considered, most notably NASA's Core Flight System (cFS), as well as commercial ones. We explicitly selected the open source, JPL led and community contributed F' because its statically defined, type-checked port topology and component-based structure align directly with our design drivers. Regarding the decision against a bespoke (i.e., utilizing no framework) implementation, we note in the manuscript that this decision prioritized the long-term scalability of

a rigorous, component-based architecture over the “direct control” of a bare-metal implementation, also factoring the flight-proven, advanced features of F’ (which we consider as cutting-edge in the domain).

Reviewer Comment 19: *This part of the Lessons Learned was unclear to me, please consider rephrasing/elaborating: “This way, only the mission-critical parts of the ADCS client library were incorporated into the FSW, while integration of the remaining parts was deferred to the ground station software, respecting both time and storage budgets.”*

Authors’ Response: We thank the reviewer for pointing out the lack of clarity here – we revised accordingly. To clarify, the manufacturer’s ADCS client library provides an extensive set of APIs. Because the proxy pattern allows the ground station to communicate directly with the ADCS, the OBC does not need to implement every possible command. Therefore, we only needed to compile the absolute minimum ADCS APIs into the OBC firmware for automated on-board tasks. The vast majority of the API functionality was thus implemented exclusively within ground station software (and not flight software).

This division significantly reduced the compiled FSW binary size and accelerated FSW development. Furthermore, the proxy approach provides operational flexibility, as development on the ground station’s ADCS control software can continue even while the satellite is in orbit. We expanded Section 5.4 to make this rationale explicit.

Reviewer Comment 20: *Going through the code generation part of the Related Work I was wondering whether F’ was used in your case for code or interface generation (I assume the latter, but would be nice to comment on this).*

Authors’ Response: We thank the reviewer for highlighting this. F’ follows a model-driven engineering mindset: the system architecture is first captured as a high-level model of its components, ports, and the topology connecting them (see Figure 5 for an instance of this for the ERMIS FSW). Subsequently, an autocoder automatically generates the underlying component base classes, port interfaces, and the overall structural topology. This approach translates the architectural specification directly into concrete interface definitions and structural scaffolding, ensuring that the system’s structural foundation remains synchronized with the architectural design. We have updated the second bullet point of Section 5.2 to clarify this.

On Architectural Analysis for Nanosatellite Flight Software: Deconstructing ERMIS3 per 4+1

Christoforos Vasilakis^a, Alexandros Tsagkaropoulos^a, Angelos Motsios^b, Panagiotis Karathanasis^a, Christos Tsigkanos^{b,*},
Dionysios Reisis^a

^aDigital Systems Team, University of Athens, Athens, Greece

^bSpace Software Group, University of Athens, Athens, Greece

Abstract

The engineering of flight software for nanosatellite missions presents significant challenges due to constrained on-board computational resources, stringent reliability requirements, and complex, mission-specific operational demands. Despite the advancements of the New Space era, detailed architectural documentation is rarely available to the research community: this paper addresses this gap by illustrating a rigorous architecture for the ERMIS3 mission, a nanosatellite designed for high-throughput laser optical downlinking and hyperspectral Earth observation. Our architectural perspective is undertaken through the lens of Kruchten's 4 + 1 architectural view model. By deconstructing the system into its physical, logical, development, process, and scenario views, we provide an in-depth analysis of the architectural decisions, trade-offs, and design rationales that ensure the system satisfies critical design drivers of deployability, integrability, modifiability, and testability. We further position the design presented over architectural variation points elicited across more than 30 surveyed missions, categorizing the evolution of FSW frameworks and styles. This experience report intends to advocate rigorous software architecture principles in space software engineering, by sharing insights and providing detailed architectural documentation with the overall goal of advancing a novel research agenda for dependable and reusable flight software.

Keywords:

Flight Software Engineering, Architectural Views, Experience Report

1. Introduction

The engineering of flight software (FSW) architectures for nanosatellite missions represents a convergence of complex challenges inherent to the domain of high-reliability on-board systems. While the *New Space* (Martin, 2017) era has democratized access to powerful on-board computers and lowered barriers to entry, it has simultaneously increased software complexity and the scale of mission-specific operational demands (Dvorak, 2009). Indeed, the body of knowledge spanning architecture, requirements, specification and implementation of software on spacecraft and their payloads (European Cooperation for Space Standardization, 2013) is vast; however, the industry frequently suffers from a lack of disseminated architectural documentation e.g., due to intellectual property restrictions, which inhibits the growth of a collective body of knowledge within the research community.

We seek to address this gap by providing a comprehensive perspective of the FSW of the ERMIS3 nanosatellite mission from a software architecture lens. We utilize Kruchten's 4 + 1

architectural view model (Kruchten, 1995), to deconstruct the system into its Physical, Logical, Development, Process, and Scenario views. This multi-dimensional approach allows for a rigorous analysis of architectural decisions, trade-offs, and design rationales that ensure the system satisfies stringent quality attributes such as deployability, integrability, modifiability, and testability—which we treat as design drivers.

The ERMIS3 mission, a component of Greece's inaugural nanosatellite constellation, is designed to demonstrate high-throughput laser optical downlinking achieving data rates of up to 1 Gbps and hyperspectral imaging for Earth observation. By leveraging the principle of modularity and sticking to standardized interfaces, the FSW accomplishes efficient deployment and seamless integration with the various subsystems on-board (such as sensors, payloads, etc). We advocate loose coupling and high cohesion within the architecture to support straightforward code modifications, accommodating often evolving mission objectives with minimal impact on existing components. Comprehensive testability is achieved through embedded testing interfaces, utilization of automated build and simulation environments, and adherence to rigorous verification protocols, something typical in the aerospace domain. We work on top of F Prime (F') (Bocchino et al., 2018), a cutting-edge open-source framework developed by the Jet Propulsion Laboratory. Our choice is motivated by its notable real-world applications (Canham, 2022; Kolhof et al., 2021), and rigorous

*Corresponding author

Email addresses: christof-v@phys.uoa.gr (Christoforos Vasilakis),
atsagk@phys.uoa.gr (Alexandros Tsagkaropoulos),
aggelosmots@di.uoa.gr (Angelos Motsios), pkarath@phys.uoa.gr
(Panagiotis Karathanasis), christos.tsigkanos@unibe.ch (Christos
Tsigkanos), dreisis@phys.uoa.gr (Dionysios Reisis)

application of software engineering practices (e.g., components with typed port connections, and object-oriented design).

This paper extends previous work presented at ECSA (Vasilakis et al., 2026). While the preliminary work introduced the foundational software design, the current paper provides a significantly deeper exploration into the systemic flight software complexities encountered in this mission class. Specifically, we have expanded the architectural documentation to include a granular analysis of the Attitude Determination and Control System (ADCS) integration, detailed hardware-software mapping, and architectural patterns used—along positioning of the design presented over architectural variation points. ‘Specifically, our contributions are as follows:

- We advocate for a rigorous approach to applying established software architecture principles—such as separation of concerns, loose coupling, and high cohesion—to the specialized domain of nanosatellite software design;
- We report in-depth our experience in meticulously documenting the architectural decisions, technical trade-offs, and design rationales that informed the system’s evolution, derived from our experience engineering both the architecture and the flight software for the mission;
- We present detailed architectural documentation of the software design through the lens of Kruchten’s 4+1 view model, offering a multifaceted representation that bridges the gap between high-level mission requirements and low-level implementation details;
- We investigate architectural variation points in the design of nanosatellite flight software elicited across more than 30 surveyed missions, positioning the present work.

The remaining of this paper is structured as follows. We first distill fundamental design drivers in Section 2. Section 3 provides the core of our documentation, detailing the five architectural views to illustrate the system’s static structure and dynamic behavior. In Section 4, we analyze architectural variation points across surveyed missions to contextualize our design choices. Finally, Section 5 discusses lessons learned and provides the design rationale for the chosen architectural style. Related work is considered in Section 6, and Section 7 concludes the paper along with an outlook to a research agenda.

2. ERMIS3 Flight Software Design Drivers

Naturally, numerous software requirements are identified as part of any space mission, in typically extensive requirements processes involving high stakeholder engagement. To provide a holistic picture of the operational context for the ERMIS3 mission (National and Kapodistrian University of Athens, 2023), the flight software must satisfy several core mission-specific functional requirements to support its hyperspectral Earth observation and optical downlinking mission objectives. These primarily include the active monitoring of peripheral subsystem health, the reliable handling and routing of received telecommands, the deterministic control of the spacecraft’s modes of

operation, and the systematic logging of periodic telemetry values and system events. Alongside these functional mission particularities, we maintain a birds-eye view to distill the design goals that drive the architectural design. Those comprise essentially quality attributes, functionality and constraints, and were identified with the following methodological steps:

1. Key quality attributes were identified through consultations with mission partners¹, hardware-providers², and stakeholders³.
2. An iterative feedback loop was used to refine these to design drivers, ensuring alignment with institutional oversight and mission partners.
3. The 4+1 View Model was selected for representation due to its clarity and effectiveness in supporting both technical validation and team on-boarding.
4. The flight software system as an early design concept was logically decomposed and mapped to the views to clearly represent its intended structure and behavior, serving as a basis for validating the design prior to implementation.

The methodological steps above yielded design drivers (as architectural goals), selected to the stringent and particular operational demands of our working context—the FSW must not only function correctly but also adapt to evolving mission parameters and potential anomalies. To align with standardized software quality models, each driver is explicitly mapped to the ISO/IEC 25010 standard:

(D1) Deployability. The FSW design shall be deployable in terms of appropriate allocation of the software to on-board compute elements, as the runtime execution environments that support integration with various subsystems. This ensures efficient allocation across heterogeneous on-board computing resources, vital for resource-constrained environments and redundancy demands. (Aligns with the flexibility quality attribute in ISO/IEC 25010).

(D2) Integrability. The FSW design shall ensure that its software elements can interact and function together in a cohesive manner. This aims for seamless interaction between software elements, minimizing interface errors that can propagate system-wide failures (aligns with the interoperability quality attribute in ISO/IEC 25010).

(D3) Modifiability. The FSW design shall support software changes with minimal risk, ensuring that modifications can be implemented effectively and efficiently without degrading quality. This offline ease of modification facilitates the safe preparation of software updates or feature

¹University of Patras, University of the Aegean, National Observatory of Athens, and OQ Hellas.

²GomSpace for the primary bus, CubeSpace for the ADCS, and Astrolight for the optical terminal.

³The European Space Agency and the Greek Ministry of Digital Governance.

enhancements throughout the mission lifecycle while minimizing regression risks (maps to the modifiability sub-characteristic of maintainability in ISO/IEC 25010).

(D4) Testability. The FSW design shall support testability in order to ensure that functions behave as expected and meet the appropriate space mission reliability and safety requirements. This is naturally paramount for validating adherence to reliability and safety of a specific class of mission ([European Cooperation for Space Standardization, 2025, 2017](#)) (maps to the testability sub-characteristic of maintainability in ISO/IEC 25010).

Beyond D1–D4, the stakeholder consultations also considered other quality attributes prevalent in the domain (Table 1), notably reliability—including fault tolerance and recoverability—and *performance efficiency*—including time behavior and resource utilization. These were not selected as independent design drivers because they are either addressed through the four chosen ones or treated as system constraints. Reliability is an emergent outcome of the architecture: (a) deployability enables cold redundancy across compute hosts (Section 3.1); (b) integrability facilitates fault detection and isolation (Section 3.5); (c) testability validates adherence to safety standards; and (d) modifiability accelerates fault-response procedures. Performance constraints, imposed by the AVR32 hardware, are accommodated by specific design decisions (e.g., static buffer allocation, the active/passive component distinction) rather than driving the overall architectural structure.

3. Architectural Views as per 4+1

In the following, we elaborate on the architecture using the 4+1 architectural view model ([Kruchten, 1995](#)). As [Perry and Wolf \(1992\)](#) note, “multiple views are needed to emphasize and to understand different aspects of the architecture.” Our selection of 4+1 stems from its long-recognized ability to comprehensively address the multifaceted nature of complex software systems. This structure supports thorough analysis and promotes clear communication across interdisciplinary teams, which are typical in scientific missions. Additionally, it aids onboarding by presenting the system in accessible, role-specific views. Although the model’s merits have been long-recognized in the community, we note the absence of a comprehensive FSW architectural design in the public domain. For elaborating a FSW architecture, we believe that it is highly appropriate in its provision of multiple perspectives. In our case, perspectives are tailored to specific concerns (from on-board compute elements, to development, to fault management techniques), ensuring that the architecture is understood and validated from various angles. Accordingly, we detail the architecture according to the following views:

1. **Physical:** Representing the system engineer’s perspective, this view describes the on-board hardware topology and the various execution environments of the FSW.
2. **Logical:** Representing the end-user perspective, this view describes the organization of on-board software components and their functionalities.
3. **Development:** From the programmer’s perspective, this view concerns the arrangement of modules in a topology as well as the development workflow.
4. **Process:** From the integrator’s perspective, this view illustrates task interactions and execution flows within the FSW system.

The Scenario View serves as the “+1” component, integrating the four primary views through a series of scenarios that demonstrate key system functionalities. To illustrate this, we select two representative cases: first, a fragment of the *fault detection and isolation* mechanism ([Hanmer, 2007](#)), often deemed cross-cutting as it involves activations of different parts of the software architecture; and second, the proxy functionality, which demonstrates the architecture’s capacity to facilitate transparent interoperability between the ground station and proprietary subsystems. We adopt UML2 diagrams to illustrate the 4+1 views ([Muchandi, 2007](#)).

3.1. Physical View

The Physical View illustrated in Fig. 1 consists of two primary components: the ground segment and the space segment, reflecting the typical perspective of space system engineers. Communication between the ground station and the nanosatellite is accomplished by using each segment’s transceivers, which ensure data exchange through the ground and satellite antennas, establishing a communication link between the segments during 5-minute windows that occur every 12 hours.

In the ground segment, the primary interface for operators comprises the ground software⁴. Its core functions include monitoring the satellite’s health by collecting telemetry beacon data and issuing operational commands controlling the spacecraft. Within the space segment, onboard communication between subsystems takes place over a shared Controller Area Network (CAN) bus. The mission’s FSW operates on two processing units: the On-Board Computer (OBC), specifically the GomSpace Nanomind A3200, which utilizes an Atmel AVR32 microcontroller unit (MCU), and the Payload Data Processing Unit (PDPU), the Nanomind HP MK3, which integrates a Xilinx Zynq 7000 system-on-chip.

The FSW deployed on these processing units is responsible for executing mission specific tasks, including system configuration, task scheduling, command execution, event logging, telemetry management and health monitoring. Due to the limited duration of the communication windows with the ground station, the software is designed for autonomous operation, transmitting collected data only on request when the satellite is within range of the ground station’s antennas. Additionally, the PDPU handles the compression of data acquired from the

⁴We treat ground software as out of scope for this paper, as it involves axiomatically different architectural drivers, decisions and operational context.

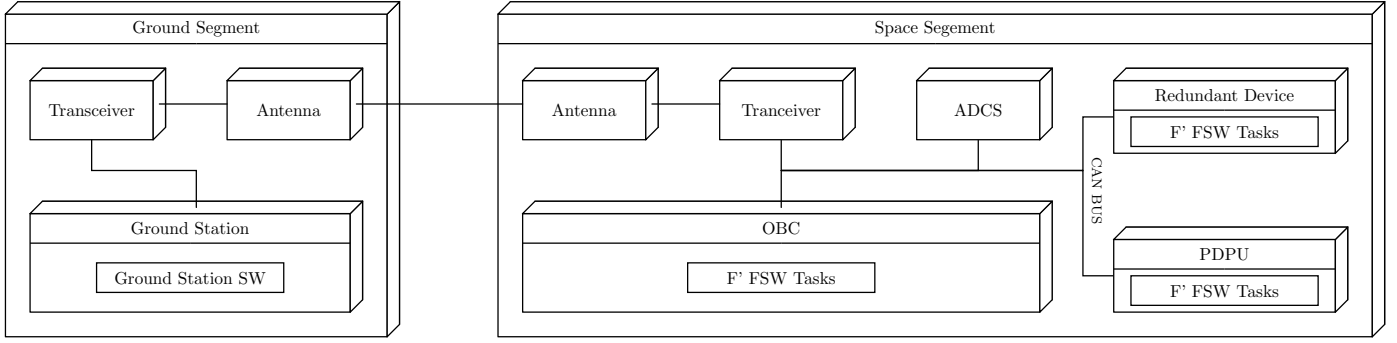


Figure 1: Deployment diagram illustrating the hardware aspects of the system and software modules running on each hardware component.

mission’s on-board hyperspectral camera and transmits the processed output to a high-throughput laser communication system.

Observe that in the design presented, both the OBC and the PDPU function in a multi-master configuration, where the OBC serves as the primary system controller. While the OBC is the primary execution context of the FSW as is typical in missions, we highlight an important deployment aspect of the developed architecture: All FSW components are also deployed on the PDPU to ensure continued operation in the event of an OBC failure, demonstrating the system’s deployability and fault tolerance. This redundancy is illustrated in Fig. 1, where the (cold) redundant spare (Bass et al., 2023; Ingeno, 2018) represents the OBC functionalities that the PDPU can assume, in addition to its nominal role in managing payload operations.

In addition to the processing units described above, the satellite incorporates third-party subsystems such as the CubeSpace ADCS, which operates proprietary closed-source firmware and thus behaves as a black-box component within the overall architecture. Interaction with the ADCS occurs exclusively through a well-defined API over the shared CAN bus. This integration demonstrates the versatility of the system, enabling interoperability with closed-source, Commercial-off-the-shelf (COTS) components through a client library integrated with the FSW.

3.2. Logical View

The Logical View is aligned on its functional requirements. We opted for the class diagram of Fig. 2 to describe the objects of the architecture and the static relationships that exist among them. It is divided into functionality areas (Muchandi, 2007) that encompass classes providing the same functionality.

The design of the *Communications* area organizes the communication with other satellite subsystems and the ground station uniformly (D2). The CubeSat Space Protocol (CSP) is a proven low-footprint and modular choice (Latachi et al., 2020) that encapsulates networking information (D3). The CommCSP class encapsulates the networking functionality of the network and transport layers (International Organization for Standardization and International Electrotechnical Commission, 1994). Specifically, it supports pre-programmed, multi-hop routing and two datagram protocols for reliable and unreliable communication between subsystems. Also, it extends

the routing capabilities of the CSP in order for the OBC to act as a proxy between the ADCS and the ground segment (D2), utilizing the remote proxy pattern Gamma et al. (1995). Each subsystem that participates in the CSP network is assigned a unique CSP address and is referred to as a CSP node. The CSP network is partitioned into two segments (see Fig. 1): (a) the ground segment, that aggregates the CSP nodes of the ground station, and (b) the space segment, which consists of the satellite’s subsystems. In the case of inter-segment communication, the CommCSP class forwards (or receives) packets’ content to (or from) the Deframer class (or the Framers class). The Framers class undertakes the encryption of a packet’s payload, which results from the issuing of a command or file transfer. The Deframer decrypts the packet’s payload and then, either routes commands to the CmdDispatcher class or forwards the file contents to the FileUplink class. The FileUplink class assembles parts to complete a file, facilitating the transfer of mission files (e.g., new binary images for in-flight firmware updates), while the FileDownlink class fragments the file’s contents into CSP packets.

Following the 3rd rule proposed by Hinchey (2018), the BufferMng class manages memory statically to mitigate the unpredictability of dynamic allocation, something typical in critical flight software. The centralized memory management allows: (a) straightforward modification of memory allocation strategy without impacting other classes of the FSW (D3); (b) controlled testing scenarios and more robust verification of memory usage (D4).

The *Command Handling* area consists of two classes, the CmdDispatcher and the CmdSequencer. The CmdDispatcher handles the routing of issued commands to their destination and the return of their issued to the source following their completion. Moreover, it encapsulates how class instances communicate with commands in the runtime environment using the mediator design pattern (Gamma et al., 1995). The CmdSequencer class complements the functionality of the CmdDispatcher class by supporting in-order execution and scheduling of command groups.

Certain classes provide functionality globally accessed by all components. The *Periodic Tasks* sub-area consists of three classes: Time, RateGroupDriver and ActiveRateGroup. The Time class shares the system’s time with other classes and can

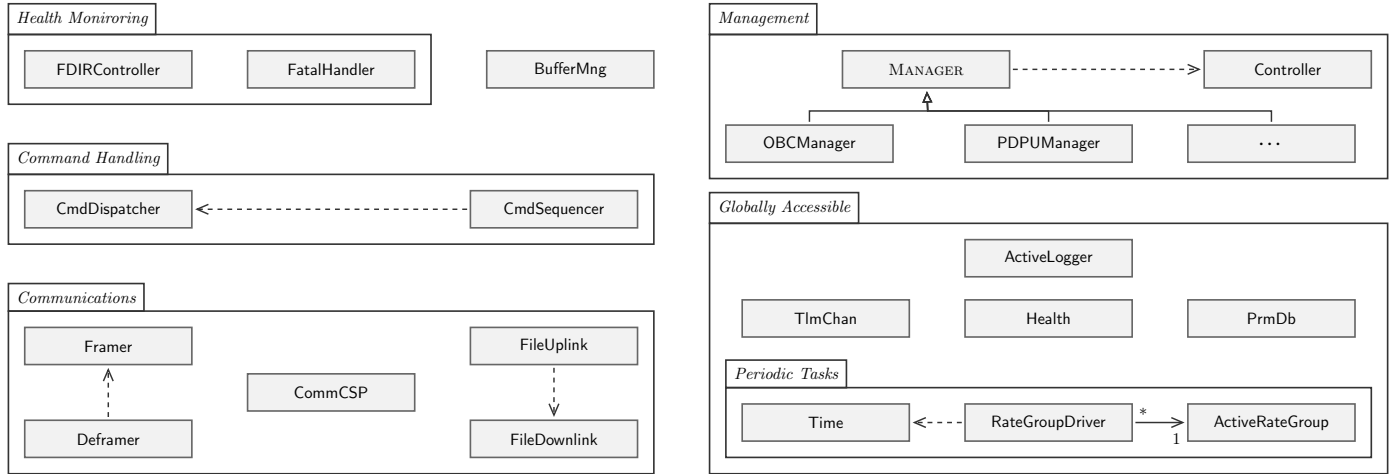


Figure 2: Class diagram representing the Logical View of the FSW architecture—it categorizes classes based on their functional areas and explicitly illustrates significant relationships among them. Dashed arrows express dependency, solid arrows denote association and solid hollow arrows indicate generalization, following UML2 notation.

synchronize the system’s time with that of the ground station. The time can be retrieved from either GPS, FRAM RTC or the MCU’s system clock. To enhance robustness, the system periodically commits the latest system timestamp to non-volatile FRAM memory. This storing action preserves a proximate time reference mitigating the risk of the system reverting to the UNIX epoch during a critical failure of primary timing sources (RTC and GPS). Maintaining valid system time is paramount, as reliance on an invalid timestamp compromises the temporal continuity of housekeeping telemetry data and the synchronization of subsystem operations; for instance, the ADCS requires precise timing to compute the pointing vector for the ground optical terminal. A synchronization error in this domain would prevent the ADCS from accurately aligning the onboard optical payload, thereby severely degrading the CubeSat’s high-speed communication capabilities. The RateGroupDriver handles periodic signaling of the ActiveRateGroup by sourcing its timing information from the Time class. When the ActiveRateGroup gets the corresponding signal, it will perform sequentially the actions with which it has been configured. Although, the RateGroupDriver is a singleton (Gamma et al., 1995) class, the ActiveRateGroup supports multiple instances. Through the interaction of these two classes, the system is able to schedule and execute periodic tasks. A notable application of this capability is the collection of telemetry data from peripheral subsystems. The PrmDB class utilizes non-volatile FRAM memory to store configuration parameters, while the TlmChan class stores telemetry data in non-volatile memory (ROM) in a serialized form suitable for downlink transmission. The ActiveLogger class stores the events generated from the FSW components in a non-volatile memory (ROM/FRAM); it uses configurable filtering functionality (D3) to distinguish the fatal events and forward these to the FatalHandler, while the remaining to the FDIRController. The Health class implements a software watchdog timer tailored for FSW components, effectively realizing the Ping/Echo architectural tactic described by Bass et al. (2023). If a periodic ping sent from the Health

class is not returned in a timely fashion, a fatal event will be raised. Note that both telemetry and events are optionally and periodically downlinked.

The *Health Monitoring* area comprises classes that handle and respond to events from the ActiveLogger class. The FDIRController class reacts to filtered events with pre-configured and tested procedures. It enables early mitigation of predefined high-risk events to ensure quick system response and prevent mission failure. The FatalHandler class manages fatal events and reboots the system after a configurable amount of time if one is received. Section 5 elaborates on the reasoning behind these two classes that handle events.

The satellite’s *Management* area includes the subsystem’s managers and the operational mode management. Spacecraft operation typically involves several so-called *modes*, which encompass the various planned phases of the mission. The Controller class encapsulates the spacecraft’s operational modes and manages transitions between them through a hierarchical structure of Finite State Machines (FSMs). Fig. 3 illustrates this hierarchical structure, where a top-level FSM governs the primary mode of operation and each mode is further decomposed into substates implemented as nested sub-FSMs, to capture more granular system behavior. This hierarchical organization ensures that the Controller provides a centralized interface for mode changes. For this purpose, the design employs guard functions that evaluate whether requested state transitions are permissible based on the current system state. The top-level FSM defines a distinct sequence of states designed to satisfy mission operational requirements:

Powerup Mode: This initial state is responsible for the initialization of critical peripheral subsystems, including the ADCS, radio transceiver, and GNSS receiver.

Post Ejection Mode: Following the initialization, this mode enforces a mandatory wait period (e.g., 45 minutes) post-deployment from the launcher carrier, ensuring compliance with launch provider requirements before transmis-

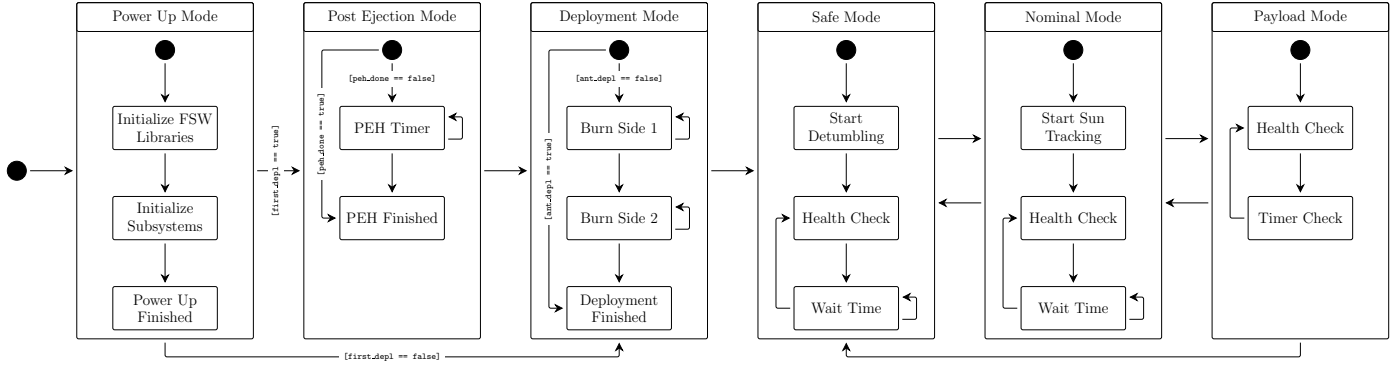


Figure 3: FSM diagram illustrating the CubeSat’s operational modes. Each mode is decomposed into a sub-FSM. The directed arrows represent transitions between states, while the text labels (enclosed in brackets) specify the guard conditions required to execute a transition.

sion begins. Crucially, this wait period is cumulative; the system stores the remaining duration in non-volatile FRAM to allow the countdown and resume from the last recorded state in the event of a system reboot. After the timer elapses, the system sets a completion flag, disabling this mode for all subsequent reboots.

Deployment Mode: Upon completion of the Post Ejection timer, the system transitions to this mode for executing the deployment sequence of the radio antennas and solar panels. By design, the system enters this mode upon each system reset to guarantee deployable release. Upon receiving a telecommand from an operator verifying successful deployment, the system deactivates this mode to prevent future attempts.

Safe Mode: This constitutes the system’s fallback mode of operation (Harvey, 2020; El Allam et al., 2021). It initializes the detumbling operations to stabilize the CubeSat after carrier ejection or following a loss of attitude control (Markley and Crassidis, 2014). In this state, the spacecraft prioritizes for energy conservation and reliability of the communication link. Note that significant anomalies detected during other modes of operation will trigger a transition to Safe Mode; exiting Safe Mode requires explicit manual intervention via a ground operator telecommand.

Nominal Mode: This represents the standard operational state where the spacecraft performs its nominal housekeeping duties and positioning procedures.

Payload Mode: This mode is dedicated to scientific/payload operations. Specifically for the ERMIS3 mission, this mode includes the capturing of ground images using the hyperspectral imager, on-board data compression using the PDPU, and data transmission via the high-throughput laser optical terminal.

Critical Mode: This state signifies a severe failure triggered by specific critical event types. It serves as a transient indicator that the FDIRController is actively executing isolation procedures. Upon completion of the predefined iso-

lation routine, the FDIRController automatically issues a command to revert the system to Safe Mode.

The *Manager* abstract class provides the basic common interface, used by concrete manager classes to realize their own (as Subsystem Managers in Bjørnsen (2023)). Each satellite subsystem—OBC, ADCS, Electrical Power System (EPS), etc.—is assigned a dedicated manager. The basic common interface includes: (a) tracking the subsystem’s state, (b) managing peripherals, and (c) validating the operational mode to execute the corresponding command that was received.

To illustrate the concrete realization of the above abstract class, we highlight the *ADCSManager* class, which controls the ADCS black-box component by integrating the manufacturer’s client library. It also encapsulates the ADCS control logic using a dedicated FSM, designed to monitor and control the ADCS state in coordination with the spacecraft’s mode of operation. Moreover, it implements the necessary Telemetry and Telecommand (T&TC) API functions, which enable the *RateGroupDriver* class to schedule the periodic reception of telemetry, and facilitating the handling of commands targeting the ADCS from the ground or other onboard software components. Operating in a similar fashion, the *EPSManager* controls the spacecraft’s power state. It monitors critical health metrics—including battery voltage, and temperature—and serves as the execution point for the application of the power profiles requested by the spacecraft’s mode of operation.

3.3. Development View

The ERMIS3 FSW design team divided the development process into two phases: the initial project setup and the component-based decomposition. The setup phase establishes foundational elements including compilers, build systems, hardware drivers, and the Operating System Abstraction Layer (OSAL). Subsequently, the development process decomposes the system into modular components. The design of these components targets their generic functionality and their reusability. We opted to present the Development View using two UML diagrams: (i) a package diagram (Fig. 4) illustrating the fundamental building blocks, and (ii) a component diagram (Fig. 5) detailing the interconnection of the framework

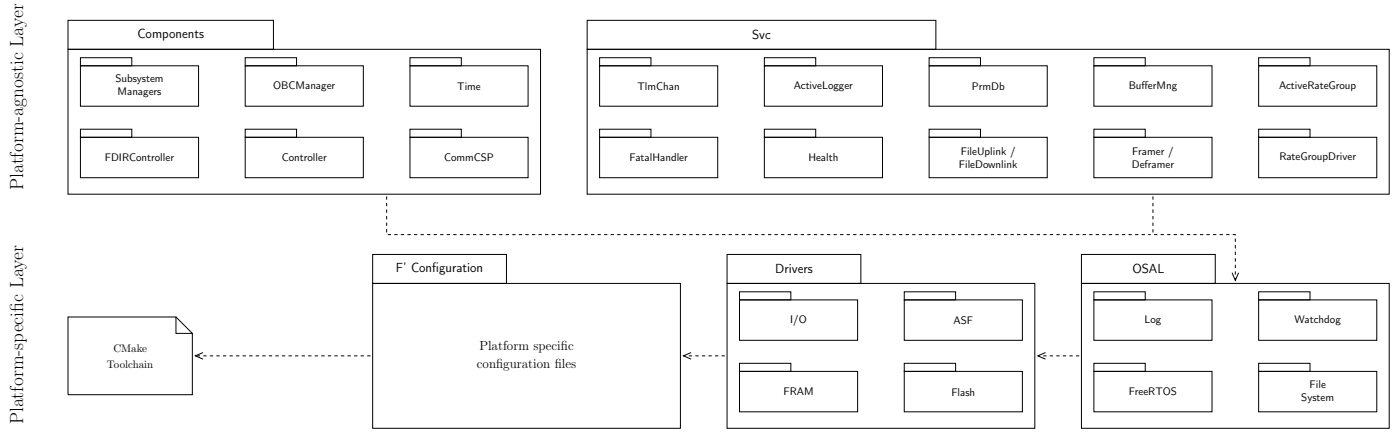


Figure 4: Package diagram illustrating the platform-specific layer (bottom) and platform-agnostic layer (top). Arrows denote dependency.

components. These are known as Service Components (Svc) and mission-specific Components and are designed to meet the functionality specifications of the OBC FSW. To maximize reuse of flight-proven code we leverage F' facilities as much as possible (Bocchino et al., 2018).

3.3.1. Platform Porting

Porting is essential and requires adaptation of the component-based workflow, involving cross-compilation of the framework in the AVR32 toolchain. Platform-specific constraints and capabilities such as the number of buffers, their size and the maximum number of concurrent commands, are specified in the F' Configuration. The F' platform layer also includes Drivers needed for the hardware of the A3200 board like the NOR Flash, FRAM and I/O interfaces, while the platform specific development concludes with the OSAL, which consists of the RTOS, the File System, Watchdogs and Log format of the system. After configuring the platform-specific layer, later component development is hardware-agnostic, enabling the same FSW codebase to be deployed across different platforms (D1); observe that components such as Health and FatalHandler are connected to hardware and the OSAL of the architecture, pointing to additional platform-specific adaptation. Fig. 4 distinguishes the lower layer as platform-specific and the upper layer as platform-agnostic.

3.3.2. Component Architecture

For an effective modular system, the FSW design relies on building blocks termed components, which encapsulate discrete portions of the system's functionality. This component-based architectural principle significantly enhances the testability by enabling independent unit testing and integration testing through scenario-based evaluation, typically employed at the final stages of the process in the domain (D4). We use F' ports to create communication channels between components; F' ports are base classes that represent well-defined interfaces (D2) and are categorized as either input (receiving data) or output (invoking an input port by sending data to it). Components and ports collectively form the core functionality of the FSW, en-

abling the system to execute its intended mission objectives, as illustrated in Fig. 5.

Following the logical analysis of Section 3.2, in the following we outline inter-component communication particularities. The CommCSP component functions as the primary communication endpoint for the On-Board Software (OBSW). Using Send and Recv ports, it manages inbound and outbound data packets. Incoming data packets are first routed to the Deframer component. Upon processing, Deframer separates the packets: command packets are routed to the CmdDispatcher via the CmdSend port, while file segments are directed to the FileUplink component via the FrameRecv port. To support the reconstruction of large files, the BufferMng component serves buffer allocation and deallocation requests through the BufferAlloc and BufferDealloc ports. The component Framer encapsulates packets intended for downlink (i.e. outbound data). It receives file packets from the FileDownlink component (via the FilePkt port) and telemetry and event logs from the TlmChan and ActiveLogger components (via the PktSend ports). Once framed, the data is passed to the CommCSP component, which routes the packets to the appropriate CSP node. The CmdDispatcher accepts commands from the uplink (via Deframer) or the CmdSequencer—which manages autonomous, time-tagged command execution—and dispatches them to the target components using the CmdSend and CmdRecv ports. The component Controller encapsulates the satellite's operational mode logic (similar to Bjørnsen (2023) Mode Manager), providing access to the current mode of operation via the GetMode port. For peripheral control, dedicated components such as the ADCSManager control specific subsystem operations. A key architectural feature of these managers is their direct interfaces with the CommCSP component. This design bypasses the standard Framer and Deframer components and enables raw data exchange with hardware that does not natively support the on-board communication protocols—exemplified by the CubeSpace ADCS integration within the ERMIS3 mission. This design strategy is replicated through subsystem-specific managers such as the EPSManager and PDPManager to manage their corresponding subsystems. Finally, the

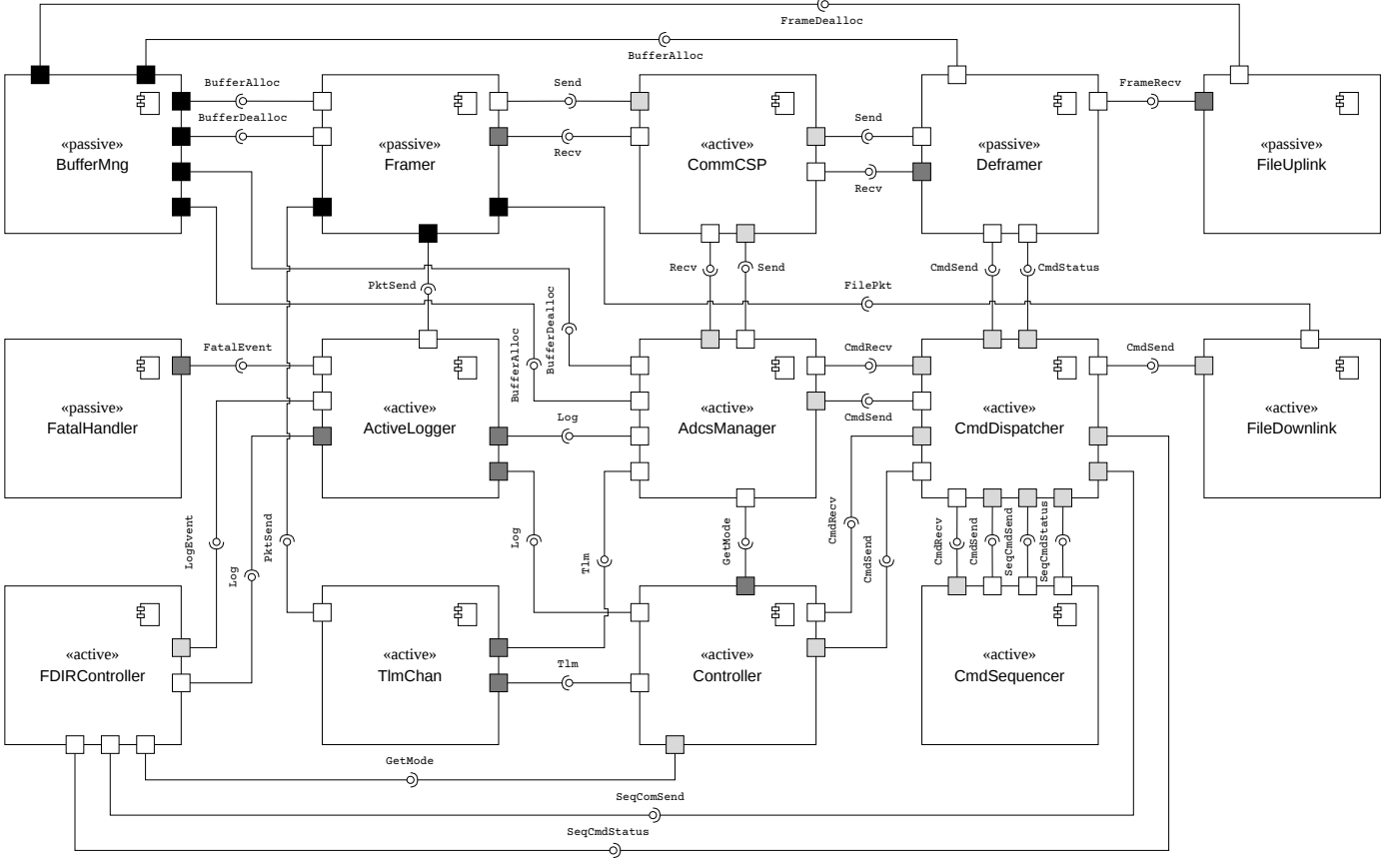


Figure 5: Component diagram of the FSW architecture. The ports are type-designated: (a) output ports in white; (b) asynchronous ports in light gray; (c) synchronous ports in dark gray; (d) guarded ports in black.

FDIRController, receives filtered events from the ActiveLogger via the LogEvent port and interfaces with the CmdDispatcher to trigger recovery procedures.

3.4. Process View

From the integrator’s perspective, FSW components can be classified as either active or passive. Active components include their own execution thread and contain a queue to store incoming data prior to processing. In contrast, passive components operate within the context of the caller’s thread. The communication ports between the components are categorized as either asynchronous or synchronous. Asynchronous ports enable non-blocking communication by operating independently on the thread of execution of their component. Conversely, synchronous ports function akin to traditional function calls, executing their functionality within the thread of the invoking component. A subtype of synchronous ports, known as guarded ports, ensures single-threaded access, thereby preventing race conditions among calling threads. As a result, active components may include both synchronous and asynchronous input ports, whereas passive ones which do not correspond to a thread of their own, are limited to synchronous input ports. However, both component types can invoke output ports. Fig. 5 illustrates the classification of ports within our architecture alongside the corresponding component types.

The BufferMng component exemplifies the practical utility of guarded ports in ensuring thread-safe operations. This component is responsible for managing the system’s limited pool of statically allocated buffers. To prevent race conditions during buffer transmission and reception—which could otherwise compromise system stability—the component exclusively utilizes guarded ports for synchronization and safe access control.

Moreover, the Controller component demonstrates the application of both synchronous and asynchronous communication models within an active component context. The GetMode interface is a synchronous port that achieves immediate state retrieval. By executing the query within the caller’s thread, the architecture minimizes latency and overhead. Conversely, the CmdSend port operates asynchronously to enforce temporal decoupling between command dispatching and execution. When the CmdDispatcher triggers this port, the command is serialized into the Controller’s queue for subsequent processing by its dedicated thread. This non-blocking mechanism prevents command execution from stalling the CmdDispatcher thread, thereby preserving the responsiveness of the system’s command router.

The process architecture of the mode management FSM, encapsulated within the Controller component, is event-driven: transitions between states occur strictly in response to messages/events requesting a state change. These events are classi-

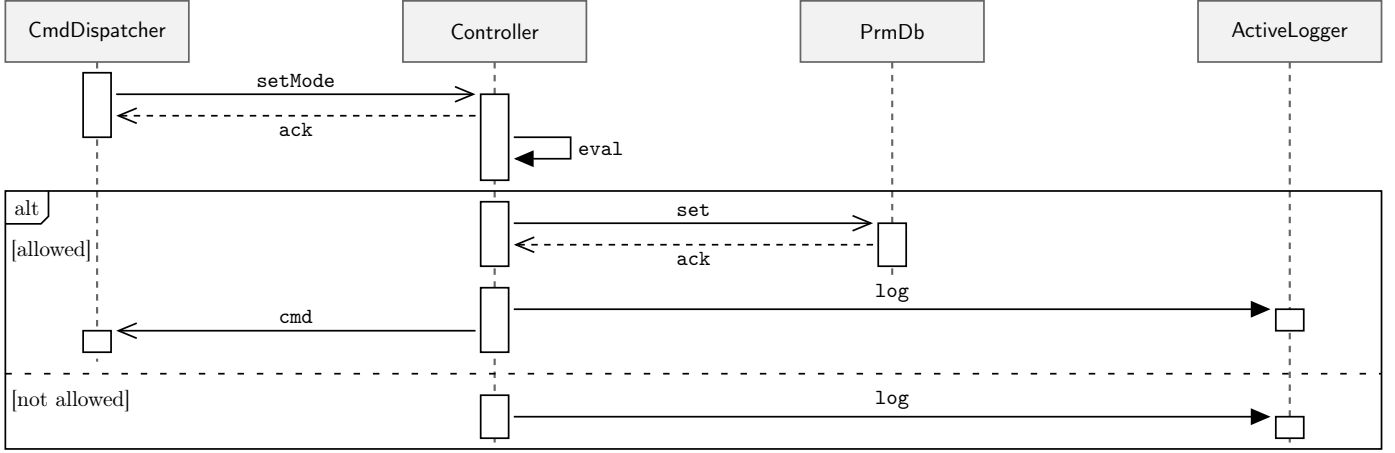


Figure 6: Sequence diagram presenting the Process View of the mode change mechanism. Filled arrows indicate synchronous calls, unfilled arrows denote asynchronous calls and alternative (alt) frame models conditional execution, following UML2 notation.

fied as either Internal—generated by on-board software (e.g., a transition to Safe Mode triggered by the FDIRController), or External—issued manually by an operator via telecommand (e.g., a transition from Safe Mode to Nominal Mode). To accomplish the intended flow of events, the Controller, being an active component, utilizes a single execution thread paired with a First-In-First-Out (FIFO) message queue. This thread operates on a blocking event loop architecture, based on the Active Object computing model (Samek, 2009, chap. 7): it continuously attempts to dequeue pending events. If the queue is empty, the FSM thread suspends execution and waits indefinitely until a new event arrives, thereby allowing the scheduler to allocate more CPU cycles to other active components.

Upon the retrieval of an event, the FSM evaluates the guard conditions. This evaluation step is crucial for ensuring system stability, preventing unsafe mode transitions, thus maintaining operational constraints. The assessment considers factors such as the current subsystem states, ongoing mission tasks, and predefined mode transition rules—all of which are encapsulated within the Controller’s logic. Upon the validation of a transition, the FSM enters the new mode and executes the corresponding output function.

Fig. 6 depicts a representative example of process communication; it illustrates the Process View of the mode change mechanism. In short, inherent functionality in Fig. 6 entails the following. The CmdDispatcher thread is responsible for forwarding mode change commands to the Controller component, which controls the satellite’s operational mode. The CmdDispatcher thread places the command in the queue of the Controller component. As an active component, the Controller has a dedicated execution thread. Once the command is dequeued, the Controller thread first issues an acknowledgment of its reception. It then evaluates whether the requested transition is permissible. If the transition is evaluated as valid, the Controller thread enqueues the new mode data into the queue of the PrmDb component. As an active component, PrmDb processes the new mode data using its execution thread to update the parameter representing the current operational mode

in FRAM storage; it then notifies the Controller thread of the result of the mode change. Subsequently, the Controller thread suspends execution until the ActiveLogger task completes logging the event in the ROM and FRAM. Since the input port of ActiveLogger is a synchronous port, it operates within the execution context of the calling thread—in this case, the Controller thread. Upon a successful mode transition, any additional commands required for execution are forwarded to the CmdDispatcher thread, which routes them to the appropriate components as part of the new mode’s output function. Conversely, if the mode change request is evaluated as invalid, the Controller component logs an event and takes no further action.

3.5. Scenario View

To complete the Kruchten’s view model and showcase the combination of the primary 4 views, in the following paragraphs we present the critical scenario of Fault Detection, Isolation, and Recovery (FDIR) and the use case scenario of the FSW’s proxy functionality.

The FDIR mechanism onboard a spacecraft is a critical autonomous system that identifies anomalies, pinpoints their source, and executes pre-programmed actions to restore nominal operations (Hanmer, 2007). It ensures mission survival by mitigating effects of hardware or software failures through rapid response and corrective measures, minimizing downtime and potential damage. As such, we select a fragment of FDIR, particularly because it involves different parts of the overall architecture.

The (use case) scenario in Fig. 7 demonstrates a high-level view on the system’s autonomous fault management capability. The procedure begins when a subsystem manager (OBCManager, EPSManager, etc.) component raises an event. Such events are collected by the ActiveLogger component, which categorizes them based on severity. Lower-severity events (trace, debug, info) are logged for later inspection by ground operators, whereas higher-severity events (warning, error) are both logged and forwarded to the dedicated FDIRController component. The response of the FDIRController component

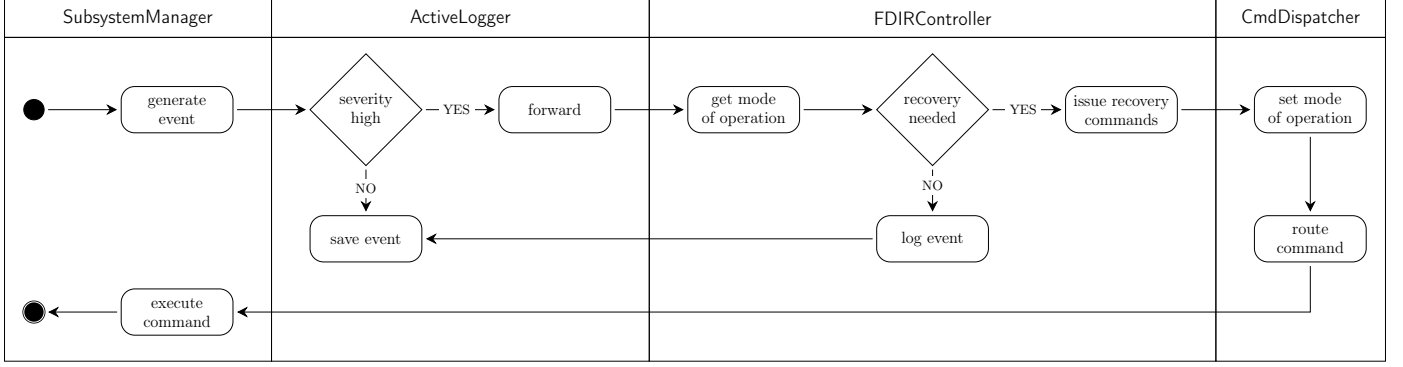


Figure 7: Activity diagram illustrating the FDIR use case scenario, following UML2.

depends on the satellite’s mode of operation. As an example, if the satellite operates in `image_capture` mode and the EPS battery level falls below a critical threshold, the system will deactivate the hyperspectral camera unit. In the other case, if the same battery event occurs in `compression` mode, the PDPU unit will be deactivated (instead of the hyperspectral camera). To determine the appropriate action, the `FDIRController` component queries the `Controller` component for the current operational mode and sends the required command to the `CmdDispatcher` for routing to the relevant component. In the case of the previous example, the `EPSManager` component receives a command to cut the power of the activated payload. Additionally, if a change in the spacecraft’s mode of operation is required, the corresponding command will be sent to the `Controller` to save the new mode. The monitoring and response process operates continuously at runtime, ensuring that the system detects and reacts to critical conditions.

In addition to FDIR, the Scenario View serves to illustrate how the system satisfies the Integrability design driver (D2) when interfacing with black-box hardware. The proxy scenario depicted in Fig. 8 addresses specific timing and routing incompatibilities between the transceiver and the ADCS. By configuring the OBC to act as an intermediary router to compensate for ADCS timing constraints (uplink) and CAN bus routing limitations (downlink), the architecture ensures reliable data exchange enabling transparent Ground-to-ADCS communication.

The sequence initiates when the Ground Station (CSP address 1) transmits a command destined for the ADCS (CSP address 20). This transmission is received by the AX100 transceiver (CSP address 21). However, a hardware limitation necessitates an intermediary step: the closed-source firmware of the ADCS requires a 1 ms delay between incoming CAN frames, a constraint violated by the transceiver’s native higher transmission rate. To mitigate this, the transceiver is configured to route the message “via” the OBC (CSP address 16). Upon reception, the OBC retransmits the data frames to the ADCS, while injecting the required 1ms inter-frame delay. This software-defined latency enables the system to forward messages from the Ground Station to the ADCS while satisfying the strict ADCS timing requirements.

The return path presents a routing challenge. The ADCS attempts to respond directly to the message originator (the

Ground Station). However, the AX100 transceiver filters out CAN traffic not explicitly addressed to itself, and the ADCS lacks the routing table functionality to designate the transceiver as a gateway (a CSP “via” node). To overcome this, the FSW team configured the OBC’s CAN interface to operate in a promiscuous mode by disabling hardware filtering. Consequently, the OBC intercepts the ADCS response traffic on the bus. Recognizing the destination as the Ground Station, the OBC acts as a transparent router, forwarding the packet to the AX100 transceiver.

This architectural solution effectively bypasses the high-level FSW stack—specifically the Deframer, `CmdDispatcher`, and `ADCSManager` components—for these specific interactions. By avoiding the overhead of high-level command processing, this proxy mechanism provides operators with a direct, low-latency tunnel to the proprietary ADCS, facilitating commanding and telemetry retrieval as if the ground station were directly connected to the satellite’s bus. Because this proxy functionality is encapsulated entirely within the `CommCSP` component, any required configuration changes remain strictly localized. This isolation preserves the maintainability of the overall system while the underlying proxy pattern establishes the `CommCSP` component as a reusable architectural template for future missions facing similar COTS hardware constraints.

4. Architectural Variation Points

To contextualize the design choices of the ERMIS3 flight software architecture within the broader domain of nanosatellite software engineering, this section analyzes the software architectural variation points identified in the literature. This investigation does not aim for completeness; rather, in the spirit of an experience report, we aim to analyze key missions in a similar mission class as ERMIS, for which software architectural information is available. The list was compiled through extensive consultation with domain experts and flight software engineers; we defer a systematic literature study for future work. As the “New Space” era introduces increasingly complex operational demands, mission architects face diverse design options, ranging from centralized, monolithic implementations (Lamichhane et al., 2015; Kekez, 2006) to modular, distributed systems (Yao et al., 2024; Eshaq et al., 2023). Three distinct architectural

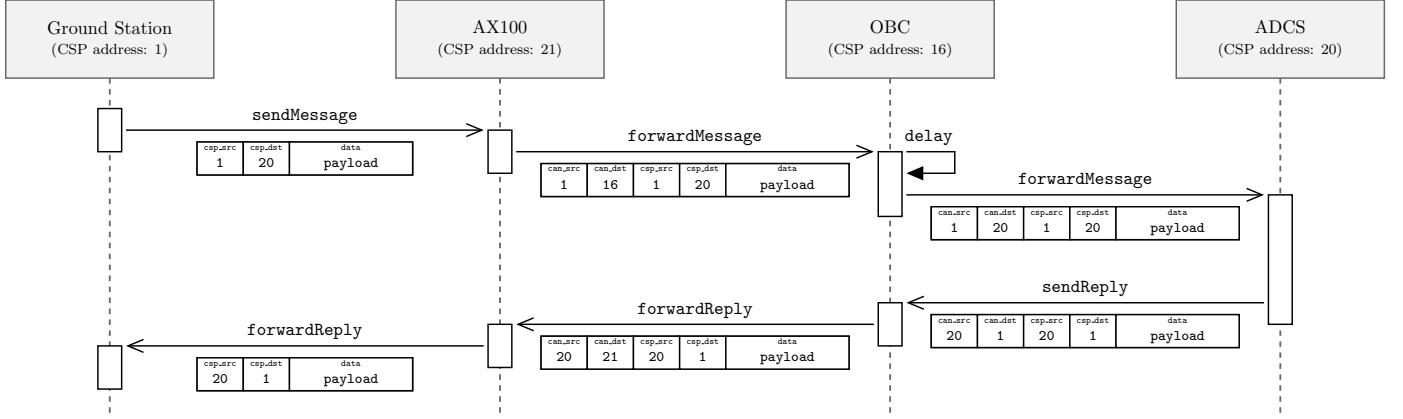


Figure 8: Sequence diagram illustrating the flight software proxy functionality. Beneath each transmission arrow, the diagram details the protocol encapsulation: all messages include a CSP header specifying source and destination addresses, while communication limited to the space segment additionally includes a CAN header denoting the hardware CAN IDs.

variation points generally influence the system design: Architectural Drivers, Framework Selection, and Architectural Style. Table 1 maps notable missions to these variation points and the following paragraphs highlight their characteristic features. While the table aggregates a substantial amount of architectural data, this compilation does not indicate a general availability of FSW documentation. In the majority of literature, architectural choices are only minorly discussed. Although a few publications extensively document individual architectural decisions, such as [de Souza et al. \(2022\)](#) and [Gonzalez et al. \(2019\)](#), comprehensive, multi-view architectural descriptions remain scarce.

Architectural Drivers. Architectural drivers are the set of critical factors that influence the system’s design. These drivers serve as the primary inputs for the architectural process, effectively specifying the rationale behind specific design decisions. According to [Ingeno \(2018\)](#), these drivers encompass design objectives, core functional requirements, quality attribute scenarios, constraints, and architectural concerns. The design drivers presented in Table 1 were derived through two methods: explicit extraction, where the drivers were directly stated by the authors in the text or implicit extraction, where drivers were inferred based on the publication content, aligned with the ISO 25010 standard ([International Organization for Standardization and International Electrotechnical Commission, 2023](#)).

Framework Selection. Fundamentally, a flight software framework serves as a reusable middleware layer that mediates the interaction between low-level hardware drivers and mission-specific application logic. It abstracts the complexities of the underlying avionics while offering standardized implementations for core flight services such as telemetry, command handling, and scheduling.

A significant trend in recent literature is the adoption of NASA JPL’s F’ framework. Utilized in missions such as Mars Ingenuity ([Canham, 2022](#)), Lunar Flashlight and NEA Scout ([Rizvi et al., 2022](#)), GT-1 ([Kolhof et al., 2021](#)), and RECO-FSCA ([Herndon et al., 2023](#)), F’ facilitates a paradigm

shift toward modular, reusable component-based architecture. Another characteristic example of a framework is found in missions utilizing formal methods to enforce correctness-by-construction. For instance, [Ivanov and Bliudze \(2017\)](#) describe the application of the Behavior-Interaction-Priority (BIP) framework for the CubETH mission. However, the accelerated timelines characteristic of the “New Space” era often necessitate a different approach. Rather than adopting these frameworks, a significant number of missions continue to rely on bespoke solutions. This preference is often driven by the view that the steep learning curve of established frameworks imposes a prohibitive overhead compared to the immediate control offered by custom implementations ([Doyle et al., 2020](#)).

Architectural Style. Beyond the selection of the design drivers and framework, high-level architectural styles influence handling of spacecraft operations and state management. An architectural style is a pattern of system organization that provides an abstract framework for the family of FSW ([Garlan et al., 1995](#)). [Perry and Wolf \(1992\)](#) note that architectural styles promote both descriptive and prescriptive decisions about architectural elements and enforcement of constraints on their structure and interaction.

Legacy CubeSat designs, such as those described in LightSail-1 ([Manyak and Bellardo, 2011](#)) and UWE-2 ([Schmidt and Schilling, 2008](#)), employ centralization. In this model, a single OBC functions as the master node, responsible for polling sensors and executing control loops. The advantage of this centralization is a straightforward verification process and the disadvantage that it introduces a single point of failure and can create performance bottlenecks during concurrent high-load operations.

To mitigate the risks inherent to centralization, modern missions have moved toward distributed architectures. For example, [Asundi and Fitz-Coy \(2013\)](#) utilized a topology where an low-power MSP430 microcontroller operated continuously for Command and Data Handling (C&DH), while a high-performance Texas Instruments DSP was powered intermittently to execute complex attitude determination and control

Reference Publication	Project/Mission	Architectural Drivers	Framework	Architectural Style
Yao et al. (2024)	Unspecified	Modularity, reusability, modifiability	—	Distributed, layered, service-oriented
Rosemurgy et al. (2024)	Unspecified	Interoperability, reliability	F'	Layered, component-based, command pattern
Wang et al. (2023)	GF-3	Resource utilization, operability, reliability	—	Distributed, layered, state machine
Eshaq et al. (2023)	PHI	Modularity, reusability	—	Layered, service-oriented, command pattern
Bjørnsen (2023)	BioSat	Resource utilization, reliability, maintainability	—	Layered, service-oriented
Rizvi et al. (2022)	LF & NEA	Modularity, reusability, testability	F'	Layered, component-based, command pattern
de Souza et al. (2022)	Canadian CubeSat	Reliability, reusability, flexibility	—	Layered, service-oriented, command pattern
Canham (2022)	Ingenuity Helicopter	Functional correctness, reusability, testability	F'	Layered, component-based, command pattern
El Allam et al. (2021)	MeznSat	Time behavior, learnability, reliability, modularity, reusability, flexibility	—	Layered, component-based, state machine, command pattern
Latachi et al. (2020)	Masat-1	Reliability, modularity, reusability, modifiability, flexibility	—	Layered, service-oriented, state machine, client-server, command pattern
Doyle et al. (2020)	EIRSAT-1	Learnability, modularity, adaptability, safe integration	Bright Ascension	Layered, component-based
Quiros-Jimenez and d'Hemecourt (2019)	GWU CubeSat	Modularity, flexibility	—	Layered, service-oriented
Gonzalez et al. (2019)	SUCHAI Missions	Reliability, modularity, reusability, modifiability, flexibility	—	Layered, command pattern
Araguz et al. (2018)	³ Cat-1	Time behavior, modularity, reusability	—	Layered, service-oriented, command pattern
Ivanov and Bludze (2017)	CubETH	Functional suitability, modularity, flexibility	BIP	Correct-by-construction, component-based
Benson et al. (2017)	CSUNSat1	Time behavior, availability, fault tolerance	—	Layered, state machine, command pattern
Arslan et al. (2017)	HAVELSAT	—	—	State machine, service-oriented
Choi et al. (2016)	KARI	Compatibility, maintainability	cFS	Layered, service-oriented, publish-subscribe
Lamichhane et al. (2015)	STUDSAT-2	Time behavior, modularity, modifiability	—	Monolithic
Mitchell et al. (2014)	KySat-2	Modularity, reusability, adaptability	SDM	Distributed, Space Plug-and-play Avionics (SPA), layered
Johl et al. (2014)	RACE, Bevo-2 & ARMADILLO	Recoverability, maintainability, flexibility	—	Centralized, component-based, state machine, command pattern
Straub et al. (2013)	OpenOrbiter	Interoperability	—	Control loop
Mahmood et al. (2013)	ICUBE-1	Fault tolerance, recoverability, flexibility	—	Service-oriented, state machine
Gundy-Burlet (2013)	LADEE	Testability	cFS & Simulink	Layered, model-based, service-oriented, publish-subscribe
Fiala and Vobornik (2013)	PilsenCUBE	Reliability	—	Layered, state machine, event-driven
Asundi and Fitz-Coy (2013)	Unspecified	Resource utilization, modularity	—	Distributed
Ferreira (2012)	ISTNanosat-1	Resource utilization, interoperability, operability, faultlessness, analysability, fail safe	—	Distributed, layered
Manyak and Bellardo (2011)	LightSail-1 & CP7	Fault tolerance, recoverability, integrity, modularity, modifiability	—	Centralized, layered, service-oriented
Schor et al. (2009)	WinCube	Fault tolerance, recoverability, modularity	—	Layered, service-oriented
Hishmeh et al. (2009)	KySat-1	Operability, fault tolerance, recoverability, modularity, reusability, analysability, testability	—	Layered, service-oriented, command pattern
Schmidt and Schilling (2008)	UWE-2	Resource utilization, reliable, modularity, reusability, modifiability, fail safe	—	Centralized, layered, service-oriented
Kekez (2006)	CanX-2	Resource utilization, operability, reliability, testability	—	Centralized, service-oriented
Dabrowski (2005)	ION	Fault tolerance, recoverability, reusability, analysability, fail safe	—	Layered, service-oriented

Table 1: Variation points between mission FSW. The first column cites the publication on which the corresponding FSW is described, the second column quotes the Project or mission name, the third column extracts the requirements that drove the FSW design mapped to the ISO/IEC 25010 standard ([International Organization for Standardization and International Electrotechnical Commission, 2023](#)), the fourth column notes the framework—if applicable—and the last column summarizes the architectural styles utilized.

algorithms. Similarly, KySat-2 (Mitchell et al., 2014) implemented a fully distributed architecture where each subsystem (radio, power, payload) possessed a dedicated “interface processor” connected via an I2C bus. In this configuration, the flight software functioned effectively as a distributed application coordinated by the Satellite Data Model (SDM) middleware.

While the software topological distribution varies between these approaches, the logic governing satellite behavior often relies on FSMs (Wang et al., 2023; Latachi et al., 2020; Benson et al., 2017; Arslan et al., 2017; Fiala and Voborník, 2013). This approach defines system behavior through discrete states—such as Detumble, Idle, or Transmit—and transition events, providing a deterministic mechanism for managing complex mission phases.

The remaining architectural styles based on which the Table 1 classifies the reviewed mission’ FSWs are defined below in order of prevalence in the surveyed literature:

Layered organizes software into hierarchical layers where each layer provides services to the layer above and consumes services from the layer below; in FSW, typical layers include a hardware abstraction, an operating system abstraction, middleware services and application logic (Gonzalez et al., 2019; Eshaq et al., 2025).

Service-oriented structures the system as a collection of loosely coupled services—such as telemetry, command handling, and housekeeping—where each service encapsulates an operational capability; units are decomposed by function and communicate at runtime (Quiros-Jimenez and d’Hemecourt, 2019; Latachi et al., 2020; Bjørnsen, 2023).

Command pattern encapsulates operational requests as first-class objects, enabling their queuing, logging, filtering and execution (Gamma et al., 1995).

Component-based composes the system from self-contained, interchangeable components with explicitly defined interactions and interfaces; units are decomposed by structure and wired at design time, as exemplified by F’ (Bocchino et al., 2018) and Bright Ascension (Doyle et al., 2020) frameworks.

Publish-subscribe decouples components by routing communication through a message bus to which producers publish and consumers subscribe, as implemented notably e.g., by NASA’s cFS software bus (Choi et al., 2016).

Client-server dedicated server processes manage shared resources and respond to requests from client processes.

Event-driven triggers component execution in response to asynchronous events rather than periodic scheduling (Fiala and Voborník, 2013).

Monolithic integrates all functionality into a tightly coupled executable.

Control loop structures execution as iterative cycles of sensing, processing, and actuating (Straub et al., 2013).

Model-based derives flight software from system models, including automated code generation (Gundy-Burlet, 2013).

Correct-by-construction paradigm employs formal methods to guarantee specified system properties by design (Ivanov and Bludze, 2017).

Space Plug-and-play Avionics (SPA) standard defines interchangeable hardware and software interfaces for modular subsystem integration (Mitchell et al., 2014).

5. Discussion & Lessons Learned

The ERMIS3 FSW architecture presented in this paper is the result of a series of deliberate design decisions and trade-offs intending to meet the stringent requirements of the mission at hand. Moreover, the design team followed the “New Space” era norms for developing modular, component-based architectures. This section discusses our design rationale in the context of the variation points identified in the literature: architectural drivers, framework selection, and architectural style. We further distill key lessons learned that can benefit future mission architects.

5.1. Architecture (Design) Drivers

Among the primary design drivers was the need for *deployability* in order to ensure efficient allocation of software across the heterogeneous on-board compute hosts (OBC/PDPU) and meet the redundancy demands of the mission. Beyond the flight environment, deployability was also critical to the development workflow, particularly in supporting continuous integration (CI) processes, where build and testing (at unit, integration, and hardware-in-the-loop levels) are supported across various pipeline stages. Given that these activities occur on different hardware architectures—i.e., development on x86, testing on x86/AVR/hardware-in-the-loop, and deployment on AVR—deployability was deemed important to be seamless, despite the inherent complex technicalities.

Integrability was another key focus, aiming to minimize interface errors that could propagate system-wide failures, as exemplified in the FDIR use case. This was achieved by limiting dependencies, promoting encapsulation, and defining narrow interfaces with ports serving as the sole communication channels between components. To dictate the execution flow, these components are categorized into active and passive types. This separation allows certain operations to progress concurrently, while others executed on the caller’s thread of control lead to a finer-grained management and monitoring of the system’s state. Additionally, the architecture emphasizes loose coupling and high cohesion, e.g., as exemplified by the CmdDispatcher; this component employs the mediator pattern to prevent others from sending direct commands to each other, thereby ensuring that command dispatching is mediated through a centralized entity. Finally, careful resource management was achieved through static allocation with the BufferMng serving as an intermediary entity enforcing, prioritization, safety, and fairness in memory—a critical and scarce resource.

Modifiability was also a crucial consideration, as in-flight updates and feature enhancements are often necessary to extend the mission lifecycle or respond to unforeseen circumstances. By adhering to loose coupling and high cohesion principles,

we ensured that modifications could be implemented with minimal risk of regression. For instance, the `Time` class encapsulates the system’s time source, allowing straightforward time source modifications. Additionally, the proposed architecture balances flexibility and efficiency in fault handling by incorporating both a `FatalHandler` and an `FDIRController`. While the `FDIRController` is an active component requiring a thread context switch and incurring greater overhead; the `FatalHandler` is passive and executes on the caller’s thread with minimal delay. More broadly, the modularity of the proposed architecture enhances modifiability by enabling component updates and replacements with minimal effort, thereby streamlining both development and maintenance.

Finally, *testability* was paramount, as it is essential for validating adherence to the stringent reliability and safety standards of space missions. This was achieved by incorporating dedicated testing interfaces and leveraging F’ primitives and workflows to efficiently support structured validation. Specifically, the framework automates the generation of test scaffolding and mock interfaces, enabling engineers to stimulate isolated components and programmatically assert expected telemetry and event outputs. Furthermore, utilizing automated simulation environments alongside continuous integration streamlined testing across different stages of development. Specifically, the rigorous verification protocols of the [European Cooperation for Space Standardization \(2025\)](#) were followed, as interpreted within the mission context.

5.2. Framework Selection

Regarding the variation point of framework selection, the adoption of NASA JPL’s F’ aligns the ERMIS3 architecture with the growing category of missions leveraging flight-proven, open-source middleware. Furthermore, this decision prioritizes the long-term scalability of a rigorous, component-based architecture over the “direct control” offered by bare-metal implementations. The architectural rationale for selecting F’ was:

- **Heritage and Reliability:** The framework provides a suite of flight-proven, reusable components for standard functions (e.g., command handling, telemetry), ensuring that critical operational logic relies on established design patterns.
- **Architecture-Centric Tooling:** The framework enforces a component-based architecture with typed port connections. Specifically, we utilize the framework’s Domain Specific Language (DSL) to model component interfaces, from which F’ automatically generates both the concrete interface definitions and the corresponding structural template code. This supports the Integrability (D2) and Modifiability (D3) drivers of the ERMIS3 mission.
- **Portability and Verification:** The framework’s testing primitives (D4) and the architectural abstraction provided by the OSAL facilitates the cross-platform Deployability (D1) required by our design drivers.

5.3. Architectural Style

The ERMIS3 architecture integrates a component-based, layered style with FSM-driven logic and a hybrid distribution model.

- **Component-Based & Layered:** The layering separates platform-specific drivers from platform-agnostic logic (Fig. 4), enabling the portability required to run the FSW on both the OBC and the PDPU, while supporting CI/CD pipelines on x86 hardware. The layered architecture also promotes maintainability and portability, as analyzed by [Harrison and Avgeriou \(2007\)](#).
- **Distributed with Centralized Control:** While literature shows a divergence between centralized and distributed architectures, our design adopts a hybrid approach. Logically, the architecture is centralized around the Controller for deterministic state management. However, the software design itself is deployable, allowing the PDPU to act as a redundant master if the OBC fails. This satisfies the mission’s redundancy requirement without the complexity of a fully distributed algorithm.
- **FSM-Driven Logic:** The Controller component utilizes hierarchical FSMs to manage mission modes (Fig. 3). This approach is consistent with widespread domain practices, as FSMs provide the deterministic behavior necessary for safety-critical transitions, such as the fallback to Safe Mode.

5.4. Lessons Learned

The trade-offs of the proposed design reflect the balance between the four competing drivers and the constraints of development time and resources. These time limitations are inherent in the overall endeavor given a relatively short mission timeline of a few months from the project’s start to launch typical in *New Space* nanosatellite missions ([De et al., 2022](#); [Osman and Mohamed, 2017](#); [Weston et al., 2025](#)). Our design rationale was guided by a combination of mission requirements, industry best practices, and lessons learned from hitherto space software development efforts. These lessons include:

- Although meeting the mission’s redundancy demands introduced a level of managerial and technical complexity that was initially underestimated, the adoption of standardized interfaces and modularity as principles were essential for realization.
- The interdisciplinary nature of the mission requiring collaboration among different fields (e.g., physics, computer science, aerospace engineering) highlighted the need for architecture and process phases capable of accommodating diverse domain-specific concerns and various approaches to solutions. The architecture’s inherent modifiability was instrumental in supporting this adaptability, especially given the mission’s accelerated development timeline.

- Due to the mission-critical implications of fatal events, the software system was designed to preserve error state information promptly and initiate timely reboots to prevent further damage. This capability was made possible through the use of a dedicated passive component (*FatalHandler*), to enable error state preservation and system recovery with minimized overhead.
- The decision of following component-based design was also instrumental. While loose coupling introduces additional complexity in terms of inter-component communication, it supports key non-functional requirements such as modifiability and integrability.
- Similarly, although loose coupling supports integrability and modifiability, it can impact performance at runtime, e.g., due to increased function calls and higher memory usage which requires careful design and assessment.
- The integration of the proprietary ADCS subsystem was facilitated by the modifiability that characterizes the FSW architecture. The introduction of the proxy pattern in the CommCSP circumvented the limitation of the ADCS's CSP implementation—namely the absence of multi-hop routing capability. Crucially, because this proxy tunnel enables direct ground-to-ADCS communication, only the absolute minimum, mission-critical ADCS APIs were compiled into the OBC firmware to support automated on-board tasks. The extensive remainder of the manufacturer's client library functionality was integrated into the ground station software. This strategic division yielded three major benefits: it significantly reduced the compiled FSW binary size; it accelerated FSW development to meet tight time budgets; and it decoupled ground software development, allowing operators to continue building new ADCS control functionalities on the ground station even while the satellite is in orbit.
- Operational experience confirms that running the OBC CAN interface in promiscuous mode to support transparent routing incurs negligible computational overhead. This demonstrates the feasibility of centralized packet inspection as a low-cost solution for interconnecting incompatible subsystems or bypassing hardware routing limitations.

Ultimately, we underline that the approach of following a rigorous architectural design through the 4+1 model provided a comprehensive and multifaceted representation of the system, supported stakeholder communication and enabled architecture understanding and validation from various angles.

6. Related Work

We advocated a FSW architecture and described our experience in its design; consequently, we classify related work into three major categories. First, we discuss software architectures in other spacecraft contexts, positioning our work within

this major application area⁵. Subsequently, we consider other, architecturally-relevant works from a wider software engineering perspective. Finally, we examine comparative studies of FSW frameworks and surveys of CubeSat software development, which provide essential context for the variation points discussed.

Forms of layered and component-based architectures are typical in FSW architectures, often largely comprising of the operating system abstraction, communication/interfaces, middleware, and functional software (Yao et al., 2024; de Souza et al., 2022; El Allam et al., 2021). Rosemurgy et al. (2024) detail the integration of the Interplanetary Overlay Network mission with F', demonstrating its adaptability and test-related component-level features. Eshaq et al. (2023) propose a ground-up FSW design emphasizing modularity and reusability through an app-based, service-oriented architecture with a command-line interface and script engine, an approach that aligns with the principle of abstraction as a cornerstone of robust FSW design. Heunis (2014) comments on the design of a generic FSW for a CubeSat, where the modularity, reusability and fault tolerance guide the implementation. Latest endeavors have employed model-based engineering and adoption of UML (or variants, such as SysML)—Halvorson and Dale Thomas (2022) present a notable approach. A taxonomy of architecture styles is outlined by Mavridou et al. (2017) with respect to the CubETH satellite.

Advanced software engineering techniques have also seen use in spacecraft missions—in the following, we highlight key ones that we deem architecture-relevant for our context. Wang et al. (2023)'s decomposition of system requirements into distinct architectural layers, coupled with the implementation of heterogeneous backup modes, exemplifies a systematic approach to addressing complex mission objectives. Similarly, the LADEE mission FSW (Gundy-Burlet, 2013) highlights MBSE from requirements to automated code generation—we highlight the emphasis on performance-related test drivers and formal verification techniques. Advanced requirements techniques such as goal modeling are developed in GOPRIME (Li et al., 2024), a framework for monitoring a goal model (from individual requirements to satisfaction of higher-level goals) integrated within the FSW architecture with according executable instrumentation. Due to the modularity inherent in the architecture presented and use of F', integration of such reasoning could be a conceptual next step in our design and highly relevant to FDIR activities. Gonzalez et al. (2019) apply software visualization techniques for FSW quality monitoring illustrated over the SUCHAI satellite series along with a FSW architecture based on the command design pattern, while also adopting fuzz testing (Gutierrez et al., 2021). Fuzz testing is also employed for automated vulnerability analysis by Willbold et al. (2024) with an empirical analysis over the FSW for ESTCube-1, OPS-Sat, and Flying Laptop, without over-reliance on extensive human expertise.

⁵For related work which addresses the variation points discussed in this mission class, please refer to Section 4.

Finally, comparative studies and surveys provide essential context for understanding current trends in FSW development. [Eshaq et al. \(2025\)](#) highlight the growing prominence of the field, noting that despite the diversity of available frameworks, scientific payloads often necessitate custom solutions—a trend evident in Table 1. In addition, [Eshaq et al. \(2025\)](#) highlight the essential features of a FSW design and underscore the importance of layered architecture and modularity, principles central to our design. Regarding architectural drivers, [El Allam et al. \(2021\)](#) outline that common goals exist across missions, though definitions vary significantly. To resolve this ambiguity, we opted for the adaption of requirements to the ISO/IEC 25010 in Table 1, to facilitate standardized comparison. Other studies narrow their scope to specific implementation strategies: [Quiros-Jimenez and d’Hemecourt \(2019\)](#); [Miranda et al. \(2019\)](#) provide comprehensive comparisons of FSW frameworks, explicitly excluding custom architectures, while [Gonzalez et al. \(2019\)](#) classify published architectures according to their technical details, development tools, and hardware platforms.

7. Conclusions and an Emerging Research Agenda

In this paper, we illustrated the FSW architecture for the ERMIS3 nanosatellite mission as per the 4+1 architectural view model. By deconstructing the on-board software system into its Physical, Logical, Development, Process and Scenario Views, we highlighted the architectural description and distilled architectural decisions, trade-offs, and design rationales that guided development. This experience report intends to advocate rigorous software architecture principles in software engineering for space software, by sharing insights and providing detailed architectural documentation with the overall goal of advancing a novel research agenda within the community. We believe the architectural variation points identified further help in this regard. Thereupon, we identify several key directions that comprise an emerging research agenda:

- A highly desirable outcome will be the development of a comprehensive reference software architecture as per ISO/IEC/IEEE 42010 ([International Organization for Standardization et al., 2011](#)), similarly to those that have been proposed in similar contexts (e.g., for embedded systems ([Spray and Sinha, 2018](#))). This process can follow a multi-phase approach: (1) systematically analyzing domain variation points to map the operational envelope of modern CubeSats; (2) defining hardware-agnostic interfaces to satisfy domain constraints ([El Allam et al., 2021](#); [Araguz et al., 2018](#); [Quiros-Jimenez and d’Hemecourt, 2019](#)); (3) prototyping a minimal, formally verified middleware core for critical capabilities like fault tolerance; and (4) building community consensus through standardization bodies (e.g., CCSDS) to drive universal adoption.
- Second, formal verification at the FSW architecture layer, with techniques such as model checking that can be embedded at the architectural (i.e., versus code) level can aim for guaranteeing correctness and reliability especially

with respect to component interactions and interfacing (which yield runtime behaviors) over a variety of properties ([Bögli et al., 2025](#); [Mavridou et al., 2017](#)). Building upon this foundation, an important research direction lies in combining such mathematical proofs with empirical testing methods ([Gundy-Burlet, 2013](#)). This could be achieved through property-driven testing pipelines, where formal properties—such as safety conditions in mode-management FSMs—are translated into executable test oracles. These oracles can act as runtime monitors during dynamic system testing, ensuring that observed operational behaviors and telemetry remain within formally verified architectural constraints.

- Third, as mission complexities escalate, static FSM-driven logic may evolve to target runtime adaptability. This direction has also been identified by [Vashev and Hinchey \(2014\)](#) for avionics, and approaches targeting it have emerged from the community ([Kucinskis and Ferreira, 2013](#); [Lowe et al., 2024](#); [Li et al., 2024](#); [Yao et al., 2024](#)). As an example, future research could enable flight software to dynamically update FDIR policies and execute autonomous operational planning without continuous ground intervention.
- Fourth, although frameworks such as NASA JPL’s F² ([Bocchino et al., 2018](#); [Canham, 2022](#)) are widely utilized, there is ample room to address existing operational bottlenecks through framework extensions ([Rosemurgy et al., 2024](#); [Herndon et al., 2023](#)) or alternative solutions ([Eshaq et al., 2023](#)). For instance, current frameworks often do not provide facilities for common spacecraft operations such as dynamic telemetry monitoring and handling, forcing developers to adopt bespoke case-by-case solutions e.g., to statically define threshold checks rather than updating parameters at runtime. Additionally, while modeling languages effectively support component generation, the absence of integration of native state machine facilities often necessitates external tools. Future research could address these gaps and explore the viability of dynamic, microkernel-style architectures or memory-safe languages for next-generation spaceflight middleware.
- Fifth, while comprehensive architecture evaluation methods, such as the Architecture Tradeoff Analysis Method (ATAM) ([Kazman et al., 1998](#)), provide systematic frameworks for quantifying design trade-offs, their implementation within the space domain presents unique challenges. According to [Gonzalez et al. \(2019\)](#), the fluid development environments and agile methodologies characteristic of CubeSat engineering make the ongoing tracking of non-functional requirements particularly challenging. Although lightweight evaluation frameworks already exist in broader software engineering—such as the Lightweight Architecture Evaluation (LAE) method ([Bass et al., 2023](#)), which streamlines core ATAM concepts for regular, internal peer reviews—their systematic application within the space domain remains largely underexplored. Conse-

quently, future research could investigate how these existing low-overhead methodologies can be tailored to assess architectural robustness in contemporary space missions.

- Finally, regarding educational and knowledge transfer aspects more widely, we believe disseminating open-source artifacts, documentation and exemplars (Motsios et al., 2026) can have substantial merit for the greater community.

Acknowledgments

Partially supported by HFRI Project 15706/RV4THINGS and the ERMIS project 4000140732/23/NL/ND. We are grateful to the anonymous reviewers, whose constructive feedback significantly enriched the discussion of the future research agenda presented.

References

- Araguz, C., Marí, M., Bou-Balust, E., Alarcon, E., Selva, D., 2018. Design Guidelines for General-Purpose Payload-Oriented Nanosatellite Software Architectures. *Journal of Aerospace Information Systems* 15, 107–119.
- Arslan, İ., Arslankoz, K., Telli, A., Karabulut, B., Aslan, A.R., 2017. On-board software for HAVELSAT cubesat, in: 2017 8th International Conference on Recent Advances in Space Technologies (RAST), IEEE, Istanbul, Turkey. pp. 229–233.
- Asundi, S.A., Fitz-Coy, N.G., 2013. Design of Command, Data and Telemetry Handling System for a Distributed Computing Architecture CubeSat, in: 2013 IEEE Aerospace Conference, IEEE, Big Sky, MT, USA. pp. 1–14.
- Bass, L., Clements, P., Kazman, R., 2023. *Software Architecture in Practice*. SEI Series in Software Engineering. 4 ed., Addison-Wesley.
- Benson, I., Kaplan, A., Flynn, J., Katz, S., 2017. Fault-Tolerant and Deterministic Flight-Software System For a High Performance CubeSat. *International Journal of Grid and High Performance Computing* 9, 92–104.
- Bjørnsen, T., 2023. *Software Architecture Design for the BioSat CubeSat*. Master's thesis. Norwegian University of Science and Technology.
- Bocchino, R., Canham, T., Watney, G., Reder, L., Levison, J., 2018. F Prime: An Open-Source Framework for Small-Scale Flight Software Systems, in: AIAA/USU Conference on Small Satellites.
- Bögli, R., Rohani, A., Studer, T., Tsigkanos, C., Kehrer, T., 2025. Temporal Logics Meet Real-World Software Requirements: A Reality Check, in: 2025 IEEE/ACM 13th International Conference on Formal Methods in Software Engineering (FormalSE), IEEE, Ottawa, ON, Canada. pp. 74–85.
- Canham, T., 2022. The Mars Ingenuity Helicopter - A Victory for Open-Source Software, in: 2022 IEEE Aerospace Conference (AERO), IEEE, Big Sky, MT, USA. pp. 01–11.
- Choi, W.s., Kim, J.H., Kim, H.d., 2016. A Study on developing Flight Software for Nano-satellite based on NASA cFS. *Journal of the Korean Society for Aeronautical & Space Sciences* 44, 997–1005.
- Dabrowski, M.J., 2005. *The Design of a Software System for a Small Space Satellite*. Master's thesis. University of Illinois at Urbana-Champaign. Urbana, Illinois.
- De, R., Abegaonkar, M., Basu, A., 2022. Enabling Science With CubeSats—Trends and Prospects. *IEEE Journal on Miniaturization for Air and Space Systems* 3, 221–231.
- de Souza, K.V.C.K., Bouslimani, Y., Ghribi, M., 2022. Flight Software Development for a CubeSat Application. *IEEE Journal on Miniaturization for Air and Space Systems* 3, 184–196.
- Doyle, M., Gloster, A., O'Toole, C., Mangan, J., Murphy, D., Dunwoody, R., Emam, M., Erkal, J., Flanagan, J., Fontanesi, G., Okosun, F., Nair, R.R., Reilly, J., Salmon, L., Sherwin, D., Thompson, J., Walsh, S., de Faoite, D., Javaid, U., McBrean, S., McKeown, D., O'Callaghan, D., O'Connor, W., Stanton, K., Ulyanov, A., Wall, R., Hanlon, L., 2020. Flight Software Development for the EIRSAT-1 Mission, in: *Proceedings of the 3rd Symposium on Space Educational Activities*, pp. 157–161.
- Dvorak, D., 2009. NASA Study on Flight Software Complexity, in: AIAA Infotech@Aerospace Conference, American Institute of Aeronautics and Astronautics, Seattle, Washington.
- El Allam, A., Jallad, A.H., Awad, M., Takruri, M., Marpu, P., 2021. A Highly Modular Software Framework for Reducing Software Development Time of Nanosatellites. *IEEE Access* 9, 107791–107803.
- Eshaq, M., Al-Midfa, I., Al-Shamsi, Z., Atalla, S., Al-Mansoori, S., Al-Ahmad, H., 2023. Flight Software Design and Implementation for a CubeSat, in: 2023 Advances in Science and Engineering Technology International Conferences (ASET), IEEE, Dubai, United Arab Emirates. pp. 1–6.
- Eshaq, M., Zitouni, M.S., Atalla, S., Al-Mansoori, S., Macdonald, M., 2025. CubeSat Flight Software: Insights and a Case Study. *Journal of Spacecraft and Rockets* 62, 1328–1345.
- European Cooperation for Space Standardization, 2013. *Space Engineering – Software Engineering Handbook*. Technical Report ECSS-E-HB-40A. ECSS.
- European Cooperation for Space Standardization, 2017. *Space Product Assurance – Software Dependability and Safety*. Technical Report ECSS-Q-HB-80-03A Rev.1. ECSS.
- European Cooperation for Space Standardization, 2025. *Space Engineering – Software*. Technical Report ECSS-E-ST-40C Rev.1. ECSS.
- Ferreira, J.A.H., 2012. *ISTNanosat-1 Heart – Processing and Digital Communications Unit*. Master's thesis. Instituto Superior Técnico.
- Fiala, P., Voborník, A., 2013. Embedded microcontroller system for Pilsen-CUBE picosatellite, in: 2013 IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), IEEE, Karlovy Vary, Czech Republic. pp. 131–134.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, Addison-Wesley.
- Garlan, D., Allen, R., Ockerbloom, J., 1995. Architectural Mismatch or Why It's Hard to Build Systems Out of Existing Parts, in: 17th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA. pp. 179–185.
- Gonzalez, C., Rojas, C., Bergel, A., Diaz, M., 2019. An Architecture-Tracking Approach to Evaluate a Modular and Extensible Flight Software for CubeSat Nanosatellites. *IEEE Access* 7, 126409–126429.
- Gundy-Burlet, K., 2013. Validation and Verification of LADEE Models and Software, in: 51st AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition, American Institute of Aeronautics and Astronautics, Grapevine (Dallas/Ft. Worth Region), Texas.
- Gutierrez, T., Bergel, A., Gonzalez, C., Rojas, C., Diaz, M., 2021. Systematic Fuzz Testing Techniques on a Nanosatellite Flight Software for Agile Mission Development. *IEEE Access* 9, 114008–114021.
- Halvorson, M., Dale Thomas, L., 2022. Architecture Framework Standardization for Satellite Software Generation Using MBSE and F Prime, in: 2022 IEEE Aerospace Conference (AERO), IEEE, Big Sky, MT, USA. pp. 1–20.
- Hanmer, R., 2007. *Patterns for Fault Tolerant Software*. Wiley Series in Software Design Patterns. 1 ed., John Wiley & Sons, Ltd, Chichester, England.
- Harrison, N.B., Avgeriou, P., 2007. Leveraging Architecture Patterns to Satisfy Quality Attributes, in: Oquendo, F. (Ed.), *Software Architecture*, Springer, Berlin, Heidelberg. pp. 263–270.
- Harvey, R., 2020. Flight Software and Software-Driven Approaches to Small Satellite Networks, in: Pelton, J., Madry, S. (Eds.), *Handbook of Small Satellites: Technology, Design, Manufacture, Applications, Economics and Regulation*. Springer International Publishing, Cham. pp. 295–329.
- Herndon, R., Banta, B., Milan, G., Zhang, Y., Walker, M., Moussa, A., Chiu, D., Karthikeyan, D., El-Hadedy, M., 2023. RECO-FSCA: Reconfigurable Low-Power Implementation of Fprime-Software for CubeSats Applications, in: Gad, A.A., Elfiky, D., Negm, A., Elbeih, S. (Eds.), *Applications of Remote Sensing and GIS Based on an Innovative Vision*, Springer Nature Switzerland, Cham. pp. 11–19.
- Heunis, A.E., 2014. *Design and Implementation of Generic Flight Software for a CubeSat*. Master's thesis. Stellenbosch University.
- Hinchey, M., 2018. The Power of Ten—Rules for Developing Safety Critical Code, in: *Software Technology: 10 Years of Innovation in IEEE Computer*. 1 ed., Wiley-IEEE Press. pp. 188–201.
- Hishmeh, S.F., Doering, T.J., Lumpp, J.E., 2009. Design of Flight Software for the KySat CubeSat Bus, in: 2009 IEEE Aerospace Conference, IEEE, Big Sky, MT, USA. pp. 1–15.
- Ingenio, J., 2018. *Software Architect's Handbook: Become a Successful Soft-*

- ware Architect by Implementing Effective Architecture Concepts. 1 ed., Packt Publishing.
- International Organization for Standardization, International Electrotechnical Commission, 1994. Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model. ISO/IEC 7498-1:1994.
- International Organization for Standardization, International Electrotechnical Commission, 2023. Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuARE) — Product quality model. ISO/IEC 25010:2023.
- International Organization for Standardization, International Electrotechnical Commission, Institute of Electrical and Electronics Engineers, 2011. Systems and Software Engineering — Architecture Description. ISO/IEC/IEEE 42010:2011.
- Ivanov, A., Bliudze, S., 2017. Robust Software Development for University-Built Satellites, in: IEEE Aerospace Conference, IEEE, Big Sky, Montana, USA.
- Johl, S., E. Glenn Lightsey, Horton, S.M., Anandayavaraj, G.R., 2014. A Reusable Command and Data Handling System for University CubeSat Missions, in: 2014 IEEE Aerospace Conference, IEEE, Big Sky, MT, USA. pp. 1–13.
- Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., Carriere, J., 1998. The Architecture Tradeoff Analysis Method, in: Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No.98EX193), IEEE, Monterey, CA, USA. pp. 68–78.
- Kekez, D.D., 2006. Development of Flight Software and Communication Systems for the CanX-2 Nanosatellite. Master's thesis. University of Toronto.
- Kolhof, M., Rawson, W., Yanakieva, R., Loomis, A., Lightsey, G., Peet, S., 2021. Lessons Learned from the GT-1 1U CubeSat Mission, in: AIAA/USU Conference on Small Satellites.
- Kruchten, P., 1995. The 4+1 View Model of architecture. IEEE Software 12, 42–50.
- Kucinskis, F.D.N., Ferreira, M.G.V., 2013. On-Board Satellite Software Architecture for the Goal-Based Brazilian Mission Operations. IEEE Aerospace and Electronic Systems Magazine 28, 32–45.
- Lamichhane, K., Kiran, M., Kannan, T., Sahay, D., Ranjith, H.G., Sandya, S., 2015. Embedded RTOS implementation for Twin Nano-satellite STUDSAT-2, in: 2015 IEEE Metrology for Aerospace (MetroAeroSpace), IEEE, Benvenuto, Italy. pp. 541–546.
- Latachi, I., Rachidi, T., Karim, M., Hanafi, A., 2020. Reusable and Reliable Flight-Control Software for a Fail-Safe and Cost-Efficient Cubesat Mission: Design and Implementation. Aerospace 7, 146.
- Li, J., Tsigkanos, C., Li, N., Tei, K., 2024. Instrumenting Runtime Goal Monitoring for F' Flight Software, in: 2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC), IEEE, Osaka, Japan. pp. 1300–1309.
- Lowe, S., Fitzpatrick, D., Buynovskiy, A., Shoemaker, L., Palo, S., D'Amico, S., 2024. Concept of Operations for SWARM-EX: A Three CubeSat Formation-Flying Mission, in: 2024 IEEE Aerospace Conference, IEEE, Big Sky, MT, USA. pp. 1–13.
- Mahmood, R., Khurshid, K., ul Isalm, Q., 2013. Design Techniques for On-Board Software of Nano-Satellites, in: 2013 6th International Conference on Recent Advances in Space Technologies (RAST), IEEE, Istanbul, Turkey. pp. 517–522.
- Manyak, G., Bellardo, J.M., 2011. PolySat's Next Generation Avionics Design, in: 2011 IEEE Fourth International Conference on Space Mission Challenges for Information Technology, IEEE, Palo Alto, CA, USA. pp. 69–76.
- Markley, L., Crassidis, J., 2014. Fundamentals of Spacecraft Attitude Determination and Control. Space Technology Library. 1 ed., Springer, New York, NY.
- Martin, G., 2017. NewSpace: The Emerging Commercial Space Industry. Conference Paper (ISU Master Class Lecture) ARC-E-DAA-TN38624. NASA Ames Research Center. Strasbourg, France.
- Mavridou, A., Stachtari, E., Bliudze, S., Ivanov, A., Katsaros, P., Sifakis, J., 2017. Architecture-Based Design: A Satellite On-Board Software Case Study, in: Kouchnarenko, O., Khosravi, R. (Eds.), Formal Aspects of Component Software, Springer International Publishing, Cham. pp. 260–279.
- Miranda, D.J.F., Ferreira, M., Kucinskis, F., McComas, D., 2019. A Comparative Survey on Flight Software Frameworks for 'New Space' Nanosatellite Missions. Journal of Aerospace Technology and Management 11, e4619.
- Mitchell, C., Rexroat, J., Rawashdeh, S.A., Lumpp, J., 2014. Development of a Modular Command and Data Handling Architecture for the KySat-2 CubeSat, in: 2014 IEEE Aerospace Conference, IEEE, Big Sky, MT, USA. pp. 1–11.
- Motsios, A., Kehrer, T., Tsigkanos, C., 2026. CUBEX: A cubesat exemplar for teaching software architecture principles, in: Bianculli, D., Sartaj, H., Andrikopoulos, V., Pautasso, C., Mikkonen, T., Pérez, J., Bures, T., Sanctis, M.D., Muccini, H., Navarro, E., Soliman, M., Zdun, U. (Eds.), Software Architecture. ECSA 2025 Tracks and Workshops - Limassol, Cyprus, September 15-19, 2025, Proceedings, Springer. pp. 29–37.
- Muchandi, V., 2007. Applying 4+1 View Architecture with UML 2. White Paper. FCG Software Services (FCGSS).
- National and Kapodistrian University of Athens, 2023. ERMIS Project Press Release.
- Osman, D., Mohamed, S., 2017. Hardware and software design of Onboard Computer of ISRASAT1 CubeSat, in: 2017 International Conference on Communication, Control, Computing and Electronics Engineering (ICCC-CEE), IEEE, Khartoum, Sudan. pp. 1–4.
- Perry, D.E., Wolf, A.L., 1992. Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes 17, 40–52.
- Quiros-Jimenez, O.D., d'Hemecourt, D., 2019. Development of a flight software framework for student CubeSat missions. Revista Tecnología en Marcha 32, 180–197.
- Rizvi, A., Ortega, K., He, Y., 2022. Developing Lunar Flashlight and Near-Earth Asteroid Scout Flight Software Concurrently using Open-Source F Prime Flight Software Framework, in: AIAA/USU Conference on Small Satellites.
- Rosemurgy, P., Gao, J., DeBaun, S., Starch, M., Levison, J., Castano, R., 2024. Enabling DTN in Spaceflight Systems: Integration of ION with the F Prime Flight Software Framework, in: 2024 IEEE Aerospace Conference, IEEE, Big Sky, MT, USA. pp. 1–8.
- Samek, M., 2009. Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems. 2 ed., Newnes.
- Schmidt, M., Schilling, K., 2008. An Extensible On-Board Data Handling Software Platform for Pico Satellites. Acta Astronautica 63, 1299–1304.
- Schor, D., Scowcroft, J., Nichols, C., Kinsner, W., 2009. A Command and Data Handling Unit for Pico-Satellite Missions, in: 2009 Canadian Conference on Electrical and Computer Engineering, IEEE, St. John's, NL, Canada. pp. 874–879.
- Spray, J., Sinha, R., 2018. Abstraction Layered Architecture: Writing Maintainable Embedded Code, in: Cuesta, C., Garlan, D., Pérez, J. (Eds.), Software Architecture, Springer, Cham. pp. 131–146.
- Straub, J., Korvald, C., Nervold, A., Mohammad, A., Root, N., Long, N., Torgerson, D., 2013. OpenOrbiter: A Low-Cost, Educational Prototype CubeSat Mission Architecture. Machines 1, 1–32.
- Vasilakis, C., Tsagkaropoulos, A., Motsios, A., Tsigkanos, C., Reisis, D.I., 2026. Nanosatellite flight software: A rigorous software architecture perspective, in: Andrikopoulos, V., Pautasso, C., Ali, N., Soldani, J., Xu, X. (Eds.), Software Architecture - 19th European Conference, ECSA 2025, Limassol, Cyprus, September 15-19, 2025, Proceedings, Springer. pp. 127–143.
- Vassey, E., Hinchey, M., 2014. Autonomy Requirements Engineering for Space Missions. NASA Monographs in Systems and Software Engineering, Springer International Publishing, Cham.
- Wang, W., Tian, H., Lei, Y., Li, X., Fan, D., Jiang, Y., Zhang, Q., Li, Y., 2023. Software System Design and Implementation of Remote Sensing SAR Satellite, in: 2023 IEEE 11th Joint International Information Technology and Artificial Intelligence Conference (ITAIC), IEEE, Chongqing, China. pp. 1623–1627.
- Weston, S., Burkhard, C., Stupl, J., Ticknor, R., Yost, B., Austin, R., Galchenko, P., Newman, L., Soto, L.S., 2025. State-of-the-Art Small Spacecraft Technology. Technical Report NASA/TP—20250000142. National Aeronautics and Space Administration (NASA). Ames Research Center Moffett Field, CA United States.
- Willbold, J., Schloegel, M., Göhler, F., Scharnowski, T., Bars, N., Wörner, S., Schiller, N., Holz, T., 2024. Scaling Software Security Analysis to Satellites: Automated Fuzz Testing and Its Unique Challenges, in: 2024 IEEE Aerospace Conference, IEEE, Big Sky, MT, USA. pp. 1–12.
- Yao, Y., Chang, L., Yu, X., Zhang, H., 2024. Application-Oriented On-Board Software Management System for Micro-Nano Satellite, in: IGARSS 2024 - 2024 IEEE International Geoscience and Remote Sensing Symposium, IEEE, Athens, Greece. pp. 6900–6903.