Establishing Robust Software Quality Practices in Space Software Systems – An Engineering-Driven Perspective

Signatures and Approval

	Name and Role	Signature	Date
Prepared by	Laszlo Etesi		15.08.2025
Approved by	Laszlo Etesi, Simon Felix		15.08.2025
Release by	Laszlo Etesi		15.08.2025

Change Log			
Issue/Rev.	**Date**	**Author**	**Reason for Change**
I1R0	2023/24	L. Etesi, Team	First Issue
I1R1	16.04.2025	L. Etesi, Team	Integrated new information and documentation
I1R2	15.08.2025	L. Etesi, S. Felix, Team	Review throughout

Change Record				
Issue/Rev.	**Date**	**Description of Change**	**Paragraph**	**Page(s)**
I1R1	16.04.2025	Major rework and extension, integration of other documentation and sources, streamlined it all.	throughout	throughout
I1R2	15.08.2025	Minor refinements in the text, few clarifications	throughout	throughout

TABLE OF CONTENTS

Tables

Figures

1. INTRODUCTION

NB: Although this document focuses on embedded systems and software, it discusses many practices that apply to any software development project.

Developing reliable software for space systems presents a unique set of challenges. Space missions operate in remote, constrained, and non-maintainable environments, where even small software issues can lead to mission delays or failures. In this context, robust software quality practices are essential to ensure systems behave as expected throughout their operational lifetime.

This document provides an engineering-driven perspective on establishing and maintaining high-quality software practices for embedded space systems. The principles and workflows described here reflect practical experience and lessons learned from real-world space projects. These practices form the foundation of our internal development approach, aiming to help teams deliver functional but also maintainable, testable, and resilient software. The document is aimed at software architects and developers, product assurance/quality assurance managers, and project managers familiarizing themselves with software product assurance and development practices.

This document combines modern development techniques, including continuous integration and delivery (CI/CD), automated testing, virtualization, model-based design, and formal verification, while maintaining a strong focus on clarity and simplicity. Rather than proposing a one-size-fits-all methodology, it offers a flexible set of guidelines that can be adapted to different project contexts and criticality levels. The traceability of requirements and their reproducibility (criteria that code must meet before acceptance) are recurring themes, ensuring that quality is built-in at every step.

The document is organized as follows: Chapter 4 introduces an overarching software engineering framework that ties together agile processes, quality management, and verification strategies and how they map to formal standards, like those issued by the European Consortium for Space Standardization (ECSS). Chapter 5 delves into CI/CD pipelines, providing detailed workflows and open-source tooling guidelines that complement Chapter 4. Chapter 6 discusses virtualization and platform simulation, showing how virtual platforms and digital twins support continuous testing (and linking back to CI in Chapter 5 and verification in Chapter 4). Chapter 7 covers Model-Based System Engineering (MBSE), illustrating how modeling approaches (using tools like Capella, TASTE, ASN.1) enhance management and design, connecting to traceability and automation themes from earlier chapters. Finally, Chapter 8 provides case studies demonstrating these practices applied in actual missions, turning challenges into lessons learned.

2. APPLICABLE AND REFERENCE DOCUMENTS

2.1. Applicable Documents

Table 1: Applicable Documents

Ref.	Description	Doc-Nr.	Issue/Rev.
[AD1]	ECSS Software Engineering Standard	ECSS-E-ST-40C	Rev. 1, 6 March 2009
[AD2]	ECSS Software Product Assurance Standard	ECSS-Q-ST-80C	Rev. 1, 6 March 2009
[AD3]	ECSS Project planning and implementation	ECSS-M-ST-10C	Rev. 1, 6 March 2009
[AD4]	NASA Software Engineering Requirements	NASA NPR 7150.2D	Rev. D, 2019
[AD5]	Software Assurance and Software Safety Standard	NASA-STD-8739.8	Rev. B, 2022
[AD6]	DO-178C: Software Considerations in Airborne Systems	RTCA DO-178C	Rev. C, Dec 2011
[AD7]	NASA Appendix C: How to Write a Good Requirement	https://www.nasa.gov/reference/appendix-c-how-to-write-a-good-requirement/	Online
[AD8]	NASA Software Engineering Handbook	NASA-HDBK-2203	B, April 2020
[AD9]	ECSS Software Engineering Handbook	ECSS-E-HB-40A	11 December 2013

2.2. Reference Documents

Table 2: Reference Documents

Ref.	Description	Doc-Nr.	Issue/Rev.
[RD1]	Agile Handbook for ECSS Software Projects	ECSS-E-HB-40-01A	2020
[RD2]	TASTE Toolset User Manual (ASSERT Set of Tools for Engineering)	https://taste.tools/	Online
[RD3]	Capella MBSE Tool's Official Documentation	https://mbse-capella.org/	Online
[RD4]	ASN.1 Specification	ISO/IEC 8824-1:2021	2021
[RD5]	ASN1SCC Compiler	https://essr.esa.int/project/asn1scc-asn-1-space-certifiable-compiler	Online
[RD6]	QEMU Emulator's Official Project Documentation	https://www.qemu.org/docs/master/	Online
[RD7]	Core Flight System (cFS) User Guide	https://etd.gsfc.nasa.gov/capabilities/core-flight-system/	Online
[RD8]	Stainless Formal Verification Framework EPFL	https://epfl-lara.github.io/stainless/intro.html	Online
[RD9]	MISRA C:2012 Guidelines for the use of the C language in critical systems	MISRA:C2023	3rd Edition
[RD10]	Orion SysML Model, Digital Twin, and Lessons Learned for Artemis I	https://ntrs.nasa.gov/citations/20220017217	15.11.2022
[RD11]	Anthropic's Claude Code	https://docs.anthropic.com/en/docs/agents-and-tools/claude-code/overview	Online
[RD12]	ASN.1 PUS-C Types Library	https://github.com/n7space/asn1-pusc-lib	Online
[RD13]	From Verified Scala to STIX File System Embedded Code Using Stainless	Hamza, J., Felix, S., Kunčak, V., Nussbaumer, I., Schramka, F. (2022). From Verified Scala to STIX File System Embedded Code Using Stainless. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds) NASA Formal Methods. NFM 2022. Lecture Notes in Computer Science, vol 13260. Springer, Cham. https://doi.org/10.1007/978-3-031-06773-0_21	2022

[RD14]	NASA Operational Simulator for Small Satellites (NOS3)	https://github.com/nasa/nos3	Online
[RD15]	BFG Repor-Cleaner	https://rtyley.github.io/bfg-repo-cleaner/	Online
[RD16]	Awesome Safety Critical (Collection on guidelines on building safety critical software)	https://awesome-safety-critical.readthedocs.io/en/latest/	Online
[RD17]	Viper (Verification Infrastructure for Permission-based Reasoning)	https://www.pm.inf.ethz.ch/research/viper.html	Online
[RD18]	Tamarin Prover	https://tamarin-prover.com/	Online

3. TERMS, DEFINITIONS, AND ABBREVIATIONS

3.1. Terms and Definitions

Table 3: Terms and Definitions

Term	Definition
Definition	A statement of the exact meaning of a word, especially in a dictionary.
Definition of Ready (DoR)	Criteria a backlog item must meet before being worked on (e.g., clear acceptance criteria).
Definition of Done (DoD)	Criteria that determine when a backlog item is considered completed.
Agile	An iterative development approach focusing on collaboration, adaptability, and incremental delivery.
Backlog	A prioritized list of tasks, features, or requirements to be addressed in a project.
Sprint	A fixed period (usually 2 to 4 weeks) in agile during which specific tasks are completed.
Pull Request (PR)	A request to merge code changes into the main codebase, usually following a review.
Static Verification	Code analysis without executing it (e.g., using linters or static analysis tools).
Dynamic Verification	Runtime checks to detect unexpected or incorrect behavior during execution.
Formal Verification	Mathematical methods to prove the correctness of code against its specifications.
Digital Twin	A virtual replica of hardware used for simulation and validation during development.
Regression Testing	Tests ensure new changes don't break existing functionality.
Software Bus	A message-based communication system is used between different modules or apps (e.g., in cFS).
Telemetry	Data are sent from spacecraft systems to ground control for monitoring.
Telecommand	Commands sent from ground control to spacecraft systems.
Code Generator	A tool that automatically creates source code based on high-level specifications or models.
Atomic Interfaces	Interfaces where operations are either completed entirely or not at all (no partial updates).
Reproducible Builds	Builds that are bit-identical when repeated, ensuring reliability and consistency.
Version Control	A system that records code changes over time (e.g., Git).
Traceability Matrix	A document that maps requirements to corresponding design, code, and test artifacts.
TCL	Tool Command Language is a scripting language commonly used for automated testing and hardware interactions.
LEON3	A 32-bit processor core based on the SPARC V8 architecture is widely used in space applications.
TSIM	A simulator for LEON processors was developed by Gaisler and is used to test and debug embedded software.
FLASH	Non-volatile memory stores firmware and software and retains data without power.
Field Programmable Gate Array	A reprogrammable integrated circuit is used to implement custom hardware logic.
ASN.1	Abstract Syntax Notation One is a standard for defining data structures for reliable cross-platform communication.
Software Verification Facility	An environment for automated and repeatable software testing across multiple execution platforms.
Operating System Abstraction Layer	A layer that allows software to run unmodified across different OS platforms.
TASTE	The ASSERT Set of Tools for Engineering, an ESA toolchain for model-based real-time system development.
Model-Based System Engineering	An approach that uses models to support system requirements, design, analysis, and verification.
Electrical Ground Support Equipment	Hardware and software systems are used on the ground to test and interact with spacecraft.
ANTLR	ANother Tool for Language Recognition is a parser generator used to read, process, and translate structured text.
QEMU	Quick EMUlator is an open-source hardware emulator used for virtualization and embedded development.
NASA Core Flight System	A modular, reusable flight software framework developed by NASA for spacecraft systems.
FLOPS	Floating Point Operations Per Second, a measure of computational performance.

Consultative Committee for Space Data Systems	An international organization that develops standards for space data systems.
PUS-C	Packet Utilization Standard – Revision C, ESA's standard for spacecraft telemetry and telecommand services.
Software Development Model	A structured framework used to plan, control, and manage software development.
WSL	Windows Subsystem for Linux is a compatibility layer that runs Linux binaries natively on Windows.
Infrastructure as Code	Managing and provisioning infrastructure using machine-readable configuration files.
Qualification Model	A hardware or software model used for final testing and verification before flight.
Protoflight Model	A flight-ready system that is used for both qualification and the actual mission.
European Consortium for Space Standardization	A consortium that develops space engineering standards in Europe.
Test-driven Development	A software development method where tests are written before code implementation.
Verification and Validation (V&V)	Processes to ensure software meets requirements and performs intended functions.
Review Item Discrepancy	A documented issue identified during formal project reviews that must be resolved.
ASN1SCC	ASN.1 Space Certified Compiler, a code generator that produces safe and reliable code from ASN.1 specifications.
POLA	The Principle Of Least Astonishment is a design principle stating that systems should behave as users expect.
KISS	Keep It Simple, Stupid is a design principle emphasizing simplicity and avoiding unnecessary complexity.
Fail-Fast Methodology	A development approach where systems are designed to report failures, simplifying debugging and improving reliability immediately.

3.2. Abbreviations

Table 4: Abbreviations

Abbreviation	Definition
AADL	Architecture Analysis & Design Language
ANTRL	ANother Tool for Language Recognition
ASN.1	Abstract Syntax Notation One
ASN1SCC	ASN.1 Space Compiler
ASTE	The ASSERT Set of Tools for Engineering
ASW	Application Software
BER/DER/PER	Encoding rules (Basic/Distinguished/Packed Encoding Rules)
CCSDS	Consultative Committee for Space Data Systems
CD	Continuous Delivery
CDR	Critical Design Review
cFS	Core Flight System (NASA framework)
CI	Continuous Integration
CI/CD	Continuous Integration / Continuous Delivery
CPU	Central Processing Unit
DoD	Definition of Done
DoR	Definition of Ready
ECSS	European Cooperation for Space Standardization
EGSE	Electrical Ground Support Equipment
EO	Earth Observation
EPFL	Ecole Polytechnique de Lausanne
ESA	European Space Agency
FDIR	Failure Detection, Identification, and Recovery
FHNW	Fachhochschule Nordwestschweiz
FLOPS	Floating Point Operations
FPGA	Field Programmable Gate Array
HIL	Hardware-in-the-loop

IaC	Infrastructure as Code
IDPU	Instrument Data Processing Unit
KISS	Keep It Short and Simple
LARA	Lab for Automated Reasoning and Analysis
MBSE	Model-Based Software Engineering
ML	Machine Learning
NASA	National Aeronautics and Space Agency
NOS3	NASA Operational Simulator for Small Satellites
OBC	Onboard Computer
OSAL	Operating System Abstraction Layer
PDR	Preliminary Design Review
PFM	Proto Flight Model
PI	Principal Investigator
PKI	Public Key Infrastructure
POLA	Principle of Least Astonishment
PR	Pull Request
PUS-C	Packet Utilization Services revision C
QEMU	Quick Emulator
QM	Qualification Model
RID	Review Item Discrepancy
RTEMS	Real-Time Executive for Multiprocessor Systems
RTOS	Real-Time Operating System
SDL	Specification and Description Language
SDM	Software Development Model
SIL	Software-in-the-Loop
SRS	Software Requirements Specification
SSS	System/Subsystem Specification
STIX	Spectrometer/Telescope for Imaging X-rays
SuSW	Startup Software
SVF	Software Verification Facility
SysML	System Modelling Language
TDD	Test-Driven Development
TM/TC	Telemetry/Telecommand
TN	Technical Note
V&V	Verification and Validation
WSL	Windows Subsystem for Linux

4. SOFTWARE ENGINEERING FRAMEWORK

This chapter describes our overarching software engineering framework for embedded space systems. It guides a structured development process to ensure quality from day one. Key themes include agile methodologies, rigorous quality management, and comprehensive verification & validation (V&V). This framework acts as the backbone for subsequent chapters: the Continuous Integration/Delivery pipeline discussed in Chapter 5, the virtualization approaches in Chapter 6, and the model-based techniques in Chapter 7 all integrated into this framework to reinforce traceability, reproducibility, and quality at every step.

For general systems engineering guidance, also see the ECSS Software Engineering Handbook [AD9] and NASA's Systems Engineering Handbook [AD8], which advocate many of the practices detailed here (requirements tracking, verification planning, etc.) in a project's lifecycle.

4.1. Guiding Principles

Throughout the software engineering process, we adhere to a set of guidelines that emphasize simplicity, clarity, and collaboration. These principles help manage the inherent complexity of space software and ensure that design decisions are made with the end goals (reliability, maintainability, testability) in mind:

- **Early availability of concise specifications:** Aim to define the software's fundamental functionality and interfaces during the system/software requirements phases. By fixing core requirements and interfaces early (while expecting some changes later), you can provide a stable foundation for development. Any detailed changes can be managed in an agile way as new information becomes available (e.g., feedback from instrument teams or evolving mission needs).
- **Malleability over upfront complexity:** Because requirements can and will change, avoid overengineering hypothetical scenarios. Instead, design simple, modular, and easily modifiable components. This approach allows the software to evolve gracefully without unnecessary complexity.
- **Focus on essential requirements:** Distinguish between crucial vs. nice-to-have features. Nonessential requirements and gold-plating unnecessarily increase complexity and risk. Focusing on what is genuinely needed for mission success keeps the system lean and easier to verify. Start simple and build out later.
- **Outcomes over implementation details:** Emphasize achieving the required outcomes (meeting requirements and user needs) rather than defining particular implementation choices. This flexibility in design encourages innovation and easier adaptation if constraints change (e.g., needing to swap a library or algorithm). It also helps in contingency situations – the team can pivot to a different implementation as long as the publicly visible behavior stays the same.

- **Minimize concurrency unless needed:** Non-deterministic concurrency (multi-threading, parallel tasks) is a significant source of complexity and complicates testing exponentially. Adopt the following approach: Only use multiple threads or interrupts where necessary for performance or real-time needs. Wherever possible, choose simpler single-threaded designs or cooperative scheduling, which are easier to reason about and verify.
- **Ensure components are individually testable:** Each software component or module should have clearly defined responsibilities and interfaces. This modularity (loose coupling and high cohesion) makes writing unit tests for each isolated component feasible. Design for testability by exposing clear APIs and avoiding hidden dependencies.
- **Atomic interfaces:** Any operation that modifies system state should be atomic (all-or-nothing). Interfaces are designed such that a function call or service completes fully with a consistent outcome or has no effect if an error occurs, leaving the system state unchanged. This prevents partial updates that could leave the system inconsistent and simplifies error handling (rollback is inherent).
- **Reproducible builds:** Maintain the complete development under version control (for example, storing compiler binaries, build scripts, Docker container definitions, etc., in the repository). This practice guarantees that anyone can reproduce or create any software version dependent on their local environment. This greatly aids debugging (a bug can be tied to an exact build) and supports continuous integration since the build environment is consistent on all machines. It also aligns with ECSS standards for configuration control and ensures that the "it works on my machine" issues are eliminated. Lastly, problems like a rare buffer overflow may manifest much later. Keeping older builds will greatly support root cause analyses because older versions are available for analysis.
- **Pragmatic use of code generators:** Where appropriate, leverage code generation to reduce human error and enforce consistency. For example, using ASN.1 and an auto-code generator (ASN1SCC) to produce telemetry/telecommand packets from one specification ensures that all subsystems use identical data definitions. Code generators are used in well-defined areas (such as communication protocol interface stubs) to prevent manual coding errors while keeping the overall system understandable.
- **Strive for reliable, resilient, robust software:** This overarching goal aims to incorporate practices at every stage to improve reliability:
    - *Extensive, automated testing:* Plan for regular automated testing at every stage using both simulated environments and actual hardware when available. This includes regression tests to catch errors quickly.
    - *Defensive programming:* Write software assuming that errors can happen. Use defensive checks and handle error conditions gracefully (never assume a function will always succeed; always code the else/exception branch).
    - *Follow the fail-fast methodology:* During development, software should stop immediately when an anomaly is detected (e.g., via assertions during development) rather than

propagating incorrect data. This makes bugs straightforward to locate and forces developers to fix issues at their source.

- *Test-driven development (TDD) for bug fixes:* When a bug is found, create a test that reproduces it (if possible) to prevent regressions, then fix the bug so the test passes.
- *Regular code reviews:* Execute peer reviews of code regularly, especially via the pull request process (see Sections 4.2 and 4.4.4). This catches issues that automated tools might miss and spreads knowledge among the team.
- *Keep it simple (KISS principle):* Prefer simple solutions over complex ones as long as they fulfill the requirements. Simpler code is more straightforward to test and less prone to bugs. Specifically, implementations are preferred with as little state as possible.
- *Loose coupling:* Ensure modules have minimal dependencies on each other's internal details so a flaw in one module is less likely to affect others.
- *Avoid dynamic memory in critical code:* If at all reasonable, especially for flight software, use static memory allocation to avoid issues like memory fragmentation during runtime. To avoid dynamic allocation, consider using memory pools or allocating enough memory for the expected worst-case at startup to avoid runtime allocations.
- *Small incremental changes:* Encourage small, frequent code commits or updates to the codebase with continuous integration. This way, issues are detected faster.
- *Principle of Least Astonishment (POLA):* Design interfaces and code that behave in a way users (or other developers) would expect to reduce misuse and errors.
- *Write unit tests to break the code:* Don't write unit tests with the attitude (or hope) of getting a green test flag but with an adversarial approach to breaking the code.

- **Clear software architecture and layering:** Document the software architecture early, outlining tasks, threads (if any), their interactions, and resource usage. Proper layering (e.g., separation of application logic from low-level drivers and OS abstraction) is enforced. This also helps map requirements for implementation and identifies critical components that may need extra scrutiny or formal verification.
- **Invest in a great development environment:** Being able to step through code is key in complex software projects. Only relying on hardware typically makes debugging more challenging and bugs more difficult to reproduce consistently. Still, one should invest time early in the project to build good hardware debugging setups. NB: Hardware watchdogs may run into a starvation trap when debugging, so you may need to turn the hardware watchdogs off while debugging. Also, keep in mind that compiler optimizations may complicate debugging, thus you should strive to develop code with reasonable performance even without compiler optimizations.

These guiding principles form the ethos of our engineering approach. They are independent of the project to ensure we haven't deviated under schedule pressure or scope changes. By instilling these principles in the team, we create a quality and deliberate design culture crucial for successful space missions.

4.2. Agile Development Process

We implement an agile methodology throughout the entire software project lifecycle, emphasizing close collaboration between our development and the client's engineering teams. Importantly, our agile approach begins from day one, during initial co-engineering, requirements gathering, and design, and continues through implementation and testing. This end-to-end agile philosophy ensures continuous stakeholder engagement and frequent verification of progress against expectations. The Agile Handbook for ECSS Software Projects [RD1] provides an excellent guideline.

4.3. Why Agile?

Adopting an agile development process offers several key benefits:

- **Flexibility:** Agile methods accommodate minor changes to requirements or design even late in development without requiring heavy change-control overhead. This is critical in space projects where new findings (e.g., from hardware tests or scientific analysis) can necessitate requirement tweaks.
- **Adaptability:** Implementation priorities can be easily adjusted between iterations, allowing the project to respond to changing schedules or external dependencies (such as hardware or other software delays).
- **Enhanced Visibility:** Short iteration cycles (sprints) produce incremental software deliveries. This provides continuous validation opportunities, better progress tracking, and early identification of deviations or risks.
- **Continuous Validation:** By integrating testing into each sprint (including integration of simulators as early as possible), we ensure that the software grows by constantly checking against requirements.

At the start of the project, collaboratively gather and prioritize requirements and tasks into a structured backlog. Use an issue-tracking tool (like Jira, GitHub Issues, or GitLab) to manage the backlog, tagging each item with relevant metadata (requirement ID, priority, etc.). Development then proceeds iteratively in sprints, typically 3-6 weeks. Only pull an item into a sprint when it meets the Definition of Ready (DoR) (clear, actionable, and testable). Each backlog item (feature, improvement, or bug fix) is designed, implemented, tested, and documented according to our Definition of Done (DoD) criteria during the sprint.

Agile is not an excuse to run a project without focus until funds are depleted. We use our agile process to control the software lifecycle from vision to implementation within the budget, time, and specification.

1. First, create a shared vision with the customer and understand the need.
2. Then formulate a big picture (we often call this "the frame" or "the box"), which is defined by the available budget and identifies any cost- or risk-driving requirements (is it going to be a car, a bike, a ship, or a plane?).
3. Then start filling in that box with a design. Use mockups and process visualization tools.
4. Iterate over 2 and 3 until the Software Requirements Specification and the Detailed Software Design document are complete.
5. Start the agile development process.

Figure 1 illustrates the need for this approach. It shows that at the beginning of a project the cost of decision making and changes is low, and the uncertainty (degree of freedom) is high. Over the course of the project, decisions are made, and information is gained, all the while we lose the ability to easily make changes and the cost of changes increases. This is why sharing a vision and building a box (which defines the primary cost drivers) is essential.

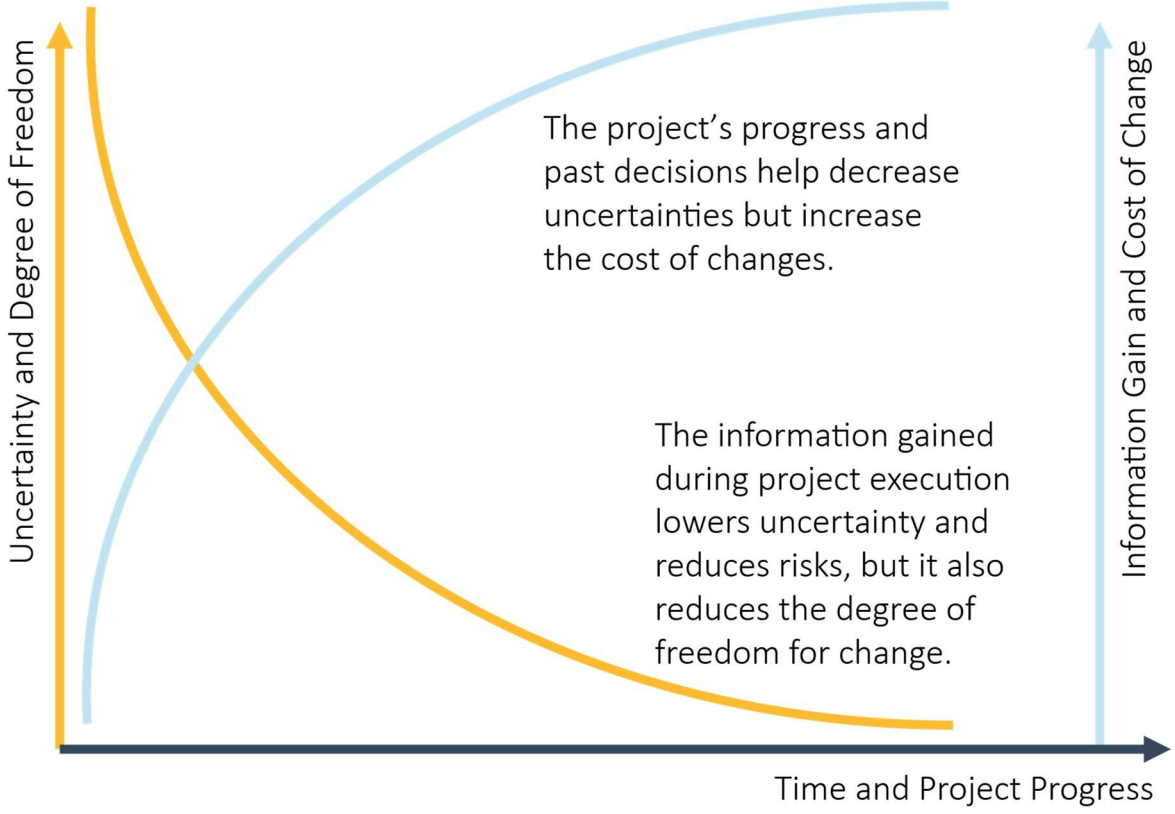

Figure 1: A project management illustration showing how typically information is gained during project execution, which helps make better decisions but increases the cost of change.

Figure 4 in the next Section illustrates how the agile approach spans requirements engineering, development, V&V, and delivery.

4.3.1. Typical Sprint Structure

This process starts once the Backlog has been prepared. **Ensure everyone on the team signs off on the process**. You can't organize and manage a process without buy-in from its members. Each sprint generally consists of the following steps:

1. **Sprint Planning:** The team establishes a clear sprint goal (what valuable increment will be delivered). Backlog items required to achieve this goal are selected based on priority and estimated effort. The team ensures the scope fits the sprint duration given their capacity (team velocity). Crucially, each candidate item is verified against the DoR at this stage – if an item isn't "ready" (unclear or missing info), it's not included. We document and make the sprint backlog visible (e.g., a sprint board in GitHub).

2. **Task Implementation:** Each selected backlog item (often broken into tasks) is developed on a dedicated feature branch in the version control system (e.g., Git on GitHub). Developers commit code to that branch frequently (at least daily), enabling continuous integration checks. Integrate often with the latest mainline to avoid drift, i.e., pull in changes from the main branch. Write unit tests for the new code during implementation and update relevant documents (requirements, design descriptions) as needed. Productivity tools, like the LLM-based solution Claude Code, can support unit test writing [RD11]. Once a task's code meets the DoD (see below) and all automated CI tests pass in the feature branch, the developer opens a pull request (PR) to merge into the main branch.

3. **Code Review & Integration:** The PR triggers a (more or less) formal code review by peers (at least one or two other developers must sign off). The reviewer checks code quality, style, and tests and ensures the changes fulfill the intended requirements. Any issues are captured as review comments. The developer addresses these in code or documentation. Once all comments are resolved and reviewers approve, the code is merged into the main branch (often called the main or development branch). The CI pipeline runs again on the merged code to double-check nothing was lost in integration.

4. **Deployment & Validation:** Deploy the new software increment to a test environment at the end of the sprint (or continuously for each feature as merged). In a space project context, "deploy" may mean running the software on an emulator or an engineering hardware model or simply packaging a release for stakeholder evaluation. Perform a sprint system demo or validation session, running through key use cases to show that new features work end-to-end. If possible, integrate with any available hardware or higher-level system to validate in a realistic context or run some longer-running mission or environment simulation (e.g., run the satellite software in a flatsat or simulated environment with other components).

5. **Sprint Review & Retrospective:** Present the sprint results to stakeholders, comparing what was achieved vs. the goal. Stakeholders verify that the increment is aligned with expectations. Immediately after, the team holds a retrospective meeting to discuss how to improve and make any process adjustments for the next sprint. Action items (e.g., "improve our unit

test coverage on driver code" or "coordinate earlier with the hardware team on interface changes") are recorded.

6. **Next Cycle Planning:** After the sprint closes, plan the next sprint with updated priorities (back to step 1). This iterative cycle repeats until the product backlog is completed or the project ends (e.g., launch date or delivery milestone).

4.3.2. Definition of Ready (DoR)

Use the DoR as a checklist before pulling an item into a sprint:

- The backlog item's **acceptance criteria** are clearly defined and unambiguous.
- All **dependencies or blocking issues are resolved** (e.g., if the item needs an interface spec, that spec is available; if it needs hardware, the hardware is ready).
- The item (story points or hours) is **estimated and fits** within sprint-given priorities.
- The **team understands** the item; do a brief backlog grooming where team members can ask questions to clarify an item.
- The item has **testability criteria**, so it is known how to verify it when done.

If any of these are false, the item stays in the backlog until the gaps are closed (ensuring we don't start work without requirements).

4.3.3. Definition of Done (DoD)

Consider a backlog item done only when it meets all of the following typical criteria:

- **All code implemented** for the item's requirements, and the code has been peer-reviewed.
- **Unit tests** and (where relevant) **integration tests** are written covering the new changes, with passing results. Use an "adversarial testing" mindset, i.e., writing tests not just for the happy path but also for edge cases and potential failure modes. Try to break the code.
- **Test coverage** is adequate (the internal target might be 80% code coverage for new code, though focus more on critical paths coverage rather than purely numeric targets).
- **No new compiler or static analyzer warnings** were introduced (a well-engineered CI pipeline runs static code analysis; any new issues must be fixed or justified).
- **All automated CI checks pass**, including unit tests, regression tests, static analysis, style/lint checks, etc.
- **Documentation is updated as needed:** implemented requirements are marked as such, design documents or user manuals are updated for the new feature, and code is commented on appropriately. Also here, LLMs may help with writing and updating documentation [RD11].

- **Integration tested**: If the change affects interfaces, perform a basic integration test in a dev environment or simulator to ensure it interacts well with other components.
- **DoD Review:** Optionally, review the checklist before closing the item in the tracking system to ensure nothing was skipped.

4.4. Quality Management

Quality management is integral to ensuring the successful delivery of a space software product or service. We implement robust document and code control processes, controlled agile workflows, and systematic reviews to guarantee quality at every project lifecycle phase. In our context, "quality" means compliance with requirements, reliability in operation, and maintainability over the mission life – all of which are objectives of both industry standards and our internal practices. The primary drivers for quality management processes are ECSS-M-ST-40 and ECSS-Q-ST-80C.

4.4.1. Document Control

Apply a rigorous document control approach suitable for a regulated environment (which demands high traceability, accountability, and control), including the blow items. Some document control is also beneficial for non-regulated environments!

- **Unique identification and versioning:** Every important project document (requirements specs, design descriptions, interface control documents, test plans, user manuals, etc.) is given a unique identifier (document number) and is version-controlled. The format can be project-specific. A general format for smaller collaborations could follow this format:

  ```
  <CLIENT>-<DOCUMENT_TYPE>-<NNNN>-<ORIGINATOR>-I<ISSUE>R<REVISION>-<NAME>
  ```

  For example, the Software Requirements Specification might be

  ```
  CST-RS-0001-ATS-I1R0-Software_Requirements_Specification_for_Component_X
  ```

  More extensive project collaborations may follow a more complex referencing scheme:

  ```
  <PROJECT>-<ORIGINATOR>-<PROJECT_ELEMENT>-<DOCUMENT_TYPE>-<NNNN>-I<ISSUE>R<REVISION>-<NAME>
  ```

  Table 5 and Table 6 provide an extensive list of sample project elements and document types. Document changes undergo a controlled process and result in new revisions, which are recorded in a change log table within the document. More significant changes lead to a new release.

- **Master Document List:** Maintain a master list of all project documents, which includes each document's ID, title, current revision, date, and owner. Formal processes often require this list (ECSS and other standards expect it). It helps anyone in the project find the latest approved version of any document. Figure 2 shows an example snippet of a master document list for a project.

List of Documents

Project: Green Space Logistics Analysis Tool
Number: 5

Code	Document Number	Document Name	Responsible Author	Company	Created On	Draft Issue	Released Issue	Released Date	Comment
RS	GSL-RS-0001-ATS	Software Requirement Specification Assessment and Comparison Tool ACT	O. Bühler	Ateleris	31.03.2022		I1R0	21.04.2022	
DD	GSL-DD-0001-ATS	Software Design Specification Assessment and Comparison Tool ACT	O. Bühler	Ateleris	05.05.2022		I1R1	06.05.2023	

Figure 2: Master document list for the lifecycle analysis project (ACT)

- **Access control and storage:** Official documents are typically stored in a central repository (like a document management system or a version control repository for docs) with appropriate access rights. Drafts and working copies might reside in a collaborative tool (like Confluence or SharePoint), but final releases are archived in PDF form in the central repository.
- **Review, approval, and Release:** High-impact documents undergo formal reviews. For instance, a System Requirements Review (SRR) will formally review the System Requirements Specificatin (SRS). Each document has an approval page (as seen at the start of this document) with signatures and dates for "Prepared by," "Reviewed by," and "Approved by." Do not consider a document baseline official until it's approved by the responsible authority (e.g., project manager or QA lead). Intermediate drafts might be marked as such (e.g., I1R0draft or "Draft for review"). Ideally, released documents highlight changes with track changes to indicate any changes made during the last iteration to the reviewer. Those changes will be applied once work on the next release version begins (see next element, "change control").
- **Change control:** Changes to baseline documents follow a change process. Minor changes might be made through an agile process (issue tracking tool linked to the doc section). In contrast, major changes may require a formal change request or even a review meeting if impacting external parties. Align this with the configuration management principles of ECSS, where changes in requirements or design documents are evaluated for impact, authorized, and traceable. Any change to a released document must be done with track changes activated and traced in the change control section ("Change Log"). To facilitate the review process, the documents are issued with the tracked changes visible (integrate view and use highlighted text only to improve legibility). The previous changes are applied when work on a new issue or release begins. This guarantees the traceability of changes across different versions of the same document.
- **Traceability in documentation:** Ensure requirements documents maintain bidirectional traceability (each requirement has a unique ID and can be traced to design elements and tests). This might be managed within the document via tables, spreadsheets, or an external tool. This traceability is crucial for verification (and is elaborated in Section 4.5).

Table 5: Project element codes used in space projects.

Project Element Code	Project Element
AIV	Assembly Integration & Validation
CON	Contractual
ENV	Environment
GS	Ground Segment
INST	Instrument
LV	Launch Vehicle

MAN	Management
MIS	Mission
MOC	Mission Operation Centre
PL	Payload
PF	Platform
PA	Product Assurance
SCI	Science
SOC	Science Operations & Data Centre
SC	Spacecraft
SYS	System Engineering
TDA	Technology

Table 6: Document type codes with often-used codes highlighted in orange.

Document Type Code	Document Type
MOU	Agreement/Memorandum of Understanding
AD	Assumption Document
AN	Analysis
AOO	Announcement of Opportunity
AR	Article
BR	Brochure
CE	Certificate (Certificate/Statement of Conformance, etc.)
CCN	Contract Change Notice
CO	Contract/Rider
CP	Change Proposal (Engineering/Document)
CR	Change Request (Engineering/Configuration)
CT	Cost Documents (Estimate/CaC/CtC, etc)
DEC	Declaration
DCR	Document Change Request
DD	Design Description/Document
DN	Delivery Notice/Release Notice/Transfer Notice
DP	Data Package
DRD	Document Requirements Definition
DW	Drawing/Diagram
EM	E-Mail
EX	Executive Summary
FI	File (Software/Configuration/Network)
FAX	Fax
HO	Handout/Presentation
IF	Interface Requirement/Specification/Interface Control Document/EID
INS	Instruction
ITT	Invitation to Tender
LB	Logbook
LE	Letter
LEG	Legal Text
LI	List
MAN	Manual/User Guide/Handbook
ML	Model
MN	Minutes of Meeting
MOU	Memorandum
MX	Matrix/Compliance
NC	Non-Conformance
OD	Operations Document
OJ	Agenda
POL	Policy Document
PG	Progress Report/Status Report
PL	Plan
PO	Proposal
PR	Procedure

PT	Product Tree
REG	Regulation
RD	Request for Deviation
REC	Record
RP	Report (Technical, Budget, Cost, Manpower, Travel, Audit, etc.)
RFQ	Request for Quotation
RS	Requirement Document/Specification (System, Subsystem, Unit, Equipment level)
RW	Request for Waiver
RES	Resolution
SC	Schedule/Network/Barchart/Chart
SP	Specifications
ST	Standards
SOW	Statement of Work
TC	Tender Conditions
TOR	Terms of Reference
TN	Technical Note
TP	Test Procedure/Test Plan
TR	Test Report/Test Result
TS	Test Specification
VCD	Verification Control Document
WBS	Work Breakdown Structure
WI	Work Instruction
WP	Working Paper
WPD	Work Package Description

4.4.2. Code Control

In parallel with document control, implement stringent code control practices:

- Use Git for **source code version control** (and related scripts/configurations). All project code resides in a Git repository (or multiple repositories for separate components, if needed), typically hosted on a platform like GitHub or GitLab for convenience.
- Every **code change is logged** with author and timestamp, and history cannot be altered without a trace (Git's immutable history). Enforce that controlled process (feature branch + PR as described below).
- The repository's main branches (e.g., main and perhaps a release branch) are **protected**; only merged commits via reviewed PRs can be updated. Direct commits to main are disallowed to ensure review and CI checks always happen.
- Leverage the platform's **automation features** for code control. For example, on GitHub, you can use GitHub Actions (CI) and branch protection rules; on GitLab, you can use similar pipelines and protected branch settings. Commits that fail CI or lack approvals cannot be merged.
- The repository is **backed up or mirrored** to prevent loss (especially important for long missions). Use tags to tag versions (using Git tags) for significant milestones (e.g., a tag v1.0 for the software delivered at CDR).
- **Branching strategy:** We typically use a simple branching strategy. Sometimes, it's a variant of GitFlow or a trunk-based approach with short-lived feature branches. The goal is to minimize long-lived branches (to reduce merge complexity) and ensure everyone integrates

frequently. For mission software, it is advisable to maintain a long-term branch for each major release (for bug fixes on that release), and any development is applied to the main branch. Short-lived branches are deleted after they have been moved back into the main development branch.

- **Issue tracking link:** Integrate issue tracking with commits. For example, each commit message references an issue ID or requirement ID it addresses. This gives traceability from code commits back to requirements or problem reports.

Using platforms like GitHub/GitLab also gives us **audit logs** and insights (like code frequency and contributions), which, while not directly a quality measure, provide transparency. It's worth noting that these practices align with ECSS-Q-ST-80C configuration management and NASA's expectations for software configuration control – the source code is a configuration item that must be controlled, and our process ensures that.

4.4.3. Controlled Agile Development Process

We sometimes refer to our approach as a "controlled agile" process, combining agile best practices with the rigor and traceability required in space projects. The controlled agile approach enhances product quality through the following:

- **Traceability and transparency:** As mentioned, we use an issue tracker to log every feature, bug, or change request. Each of these, when implemented, is linked to a Git commit (via commit messages or PR descriptions). This creates a chain: requirement, issue, code commit, and test results. Anyone can trace why a change was made or which bug it fixes. This satisfies standards that require linking code to requirements and problem reports, like ECSS.
- **Incremental implementation and validation:** Short sprints and frequent deliveries can facilitate continuous user/stakeholder feedback. Early prototypes can be shown to scientists or systems engineers to validate that the team is on the right track, reducing the risk of late rework. This also supports the ECSS principle of early and iterative verification – rather than waiting until all code is done to test, we can test as we go.
- **Disciplined integration:** You can't merge half-working code; everything must build and pass tests. Code review acts as a peer audit, often catching missing test cases or non-conformance to coding standards (like MISRA C rules in critical components). This improves stability by effectively treating the main branch as always shippable (when something is in the main, it's been tested and reviewed).
- **Continuous integration & delivery (CI/CD):** Automation is key to the controlled process. Prepare CI pipelines (Chapter 5 details this) that compile the code, run static analysis, run tests, and even package the software for deployment (for example, creating a firmware image or installer). This means every change is automatically largely verified, enforcing quality gates (like no static analysis regressions and no test failures). Continuous delivery ensures a deployable artifact at any point – valuable for frequent demos or even emergency patches.

- **Quality metrics:** The CI generates metrics (test coverage, complexity, etc.). In some cases, it might be advisable to treat thresholds on these metrics as quality gates to maintain code health (for instance, maintaining at least X% coverage or ensuring the cyclomatic complexity of functions stays within set bounds).
- If possible, it is good practice to incorporate **Digital Twin** concepts for functional software verification of embedded software. This means investing in a virtual platform or high-fidelity simulator of the system on which the software can be run. Integrating this into CI makes testing the software on near-real hardware on each commit possible. For example, one can use a CPU/system emulator (like QEMU, TSIM or T-EMU) in CI to boot the software and run some operational scenarios. This continuous V&V catches hardware-specific issues early. It also complements formal test campaigns by continuously testing between them.

Figure 3 and Figure 4 below illustrate the main phases of our controlled agile lifecycle in a typical project. It shows how upfront co-engineering (ideation, prototyping) is followed by a development cycle and how continuous integration and feedback loop back into refinement.

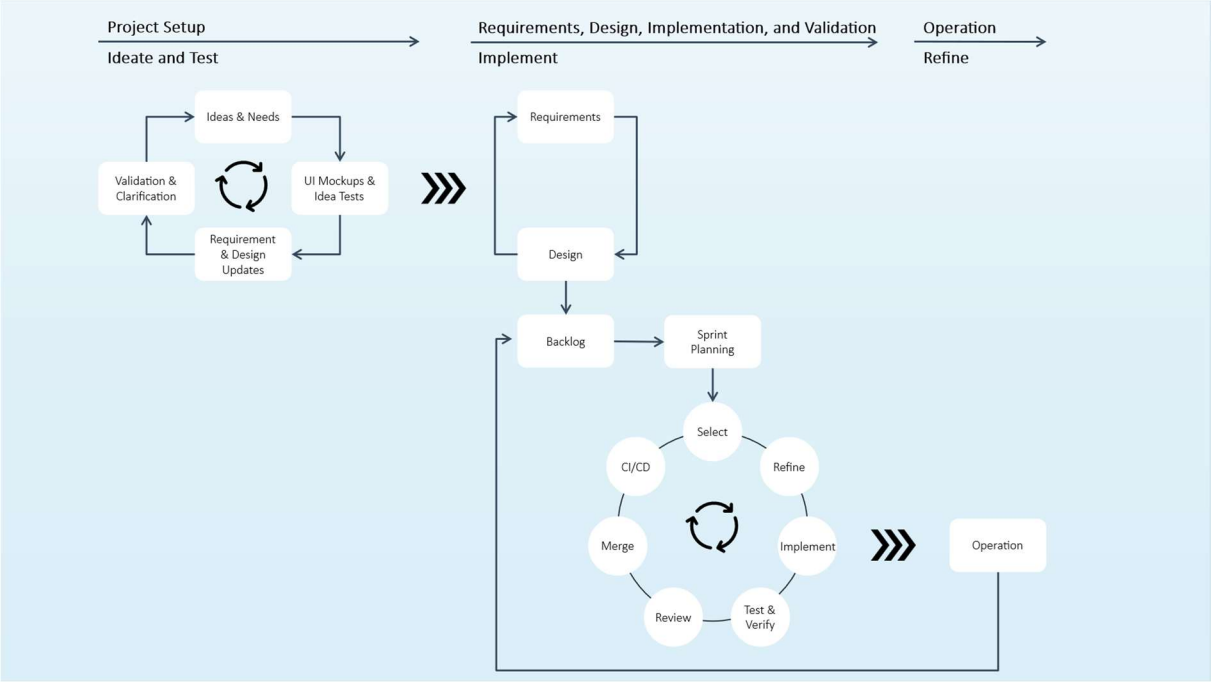

Figure 3: A more generic illustration of our agile approach, encompassing the entire software development lifecycle.

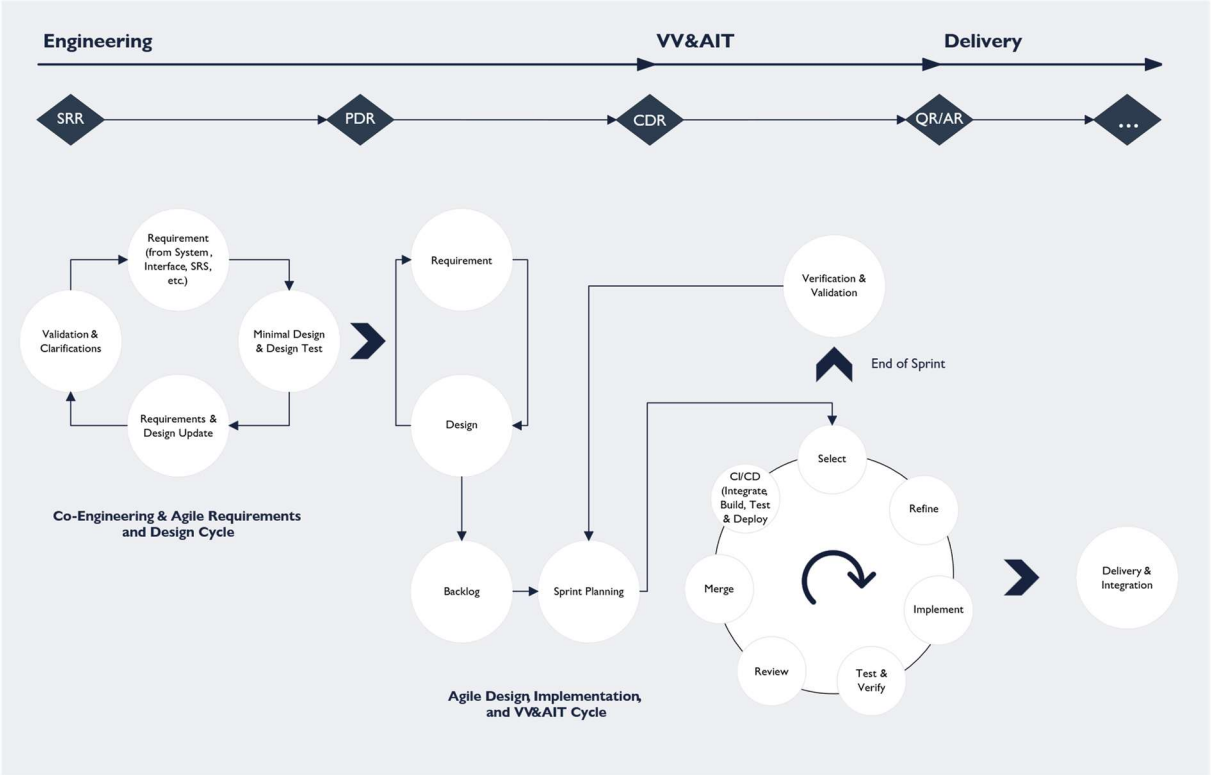

Figure 4: Phases of our controlled agile lifecycle – from initial Project Setup (capturing ideas and needs with prototyping) to a repeating cycle of Requirements, Design, Implementation, and Validation in sprints, and finally to Operation/Refinement. This approach merges agile iteration with formal milestone checkpoints (SRR, PDR, CDR, QR/AR, etc.) as required by standards, using CI/CD and continuous V&V at each step.

4.4.4. Reviews and Audits

Depending on the project's required rigor (set by the client or mission classification), incorporate formal reviews and audits at key points while keeping our process agile and responsive. For large, highly regulated projects (e.g., ESA missions or NASA projects), standard milestone reviews are mandatory: SRR (System Requirements Review), PDR (Preliminary Design Review), CDR (Critical Design Review), and possibly others like QR (Qualification Review), AR (Acceptance Review). Comprehensive documentation and software artifacts are delivered to an independent review board for scrutiny during these.

Prepare for these reviews by ensuring that:

- All relevant documents (from the document list) are up to date and reviewed internally beforehand.
- The software is at a maturity level appropriate for the review (e.g., by CDR, we should have a working prototype of every major component with testing evidence).
- Traceability matrices (requirements to design, code, and tests) are populated.
- Known issues and risks are documented (with mitigation plans).

- All deliverables are packed and can be easily shipped.
- All necessary stakeholders and reviewers have been identified and informed.
- The review process and RID tracking spreadsheets (see below) have been agreed on.

During formal reviews, reviewers raise Review Item Discrepancies (RIDs). Tracking each RID in a RID log (e.g., a spreadsheet), assigning responsible engineers, and systematically addressing them is possible; no special software or tool is needed. A follow-up review or "RID closure" meeting is often held to verify fixes before proceeding. Figure 6 illustrates a typical ESA project lifecycle with these reviews and Figure 5 shows the RID information flow in the ECSS process, which we emulate for formality.

Typically, a RID log contains the following items:

- **RID Reference:** A unique identifier assigned to each RID for traceability.
- **Originator:** The individual or organization who raised the RID.
- **Classification:** The type or severity of the issue (e.g., major, minor, editorial).
- **Reviewer Comment:** A description of the concern or issue raised.
- **Contractor Position:** The response or justification provided by the contractor or author of the reviewed document or code.
- **Action:** The agreed resolution or change to be implemented.
- **Actionee**: The person or team responsible for implementing the action.
- **Status**: The current state of the RID (e.g., open, closed with action, closed).

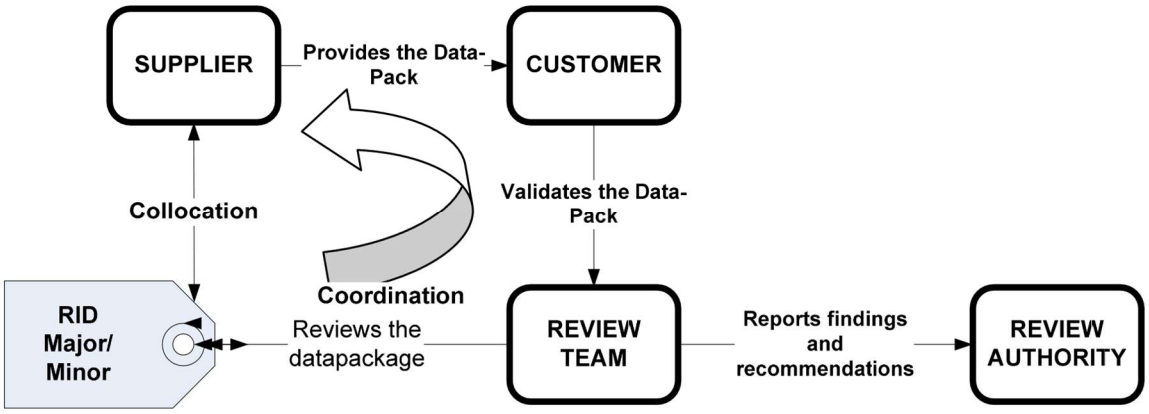

Figure 5: Review information flow as per [AD3]

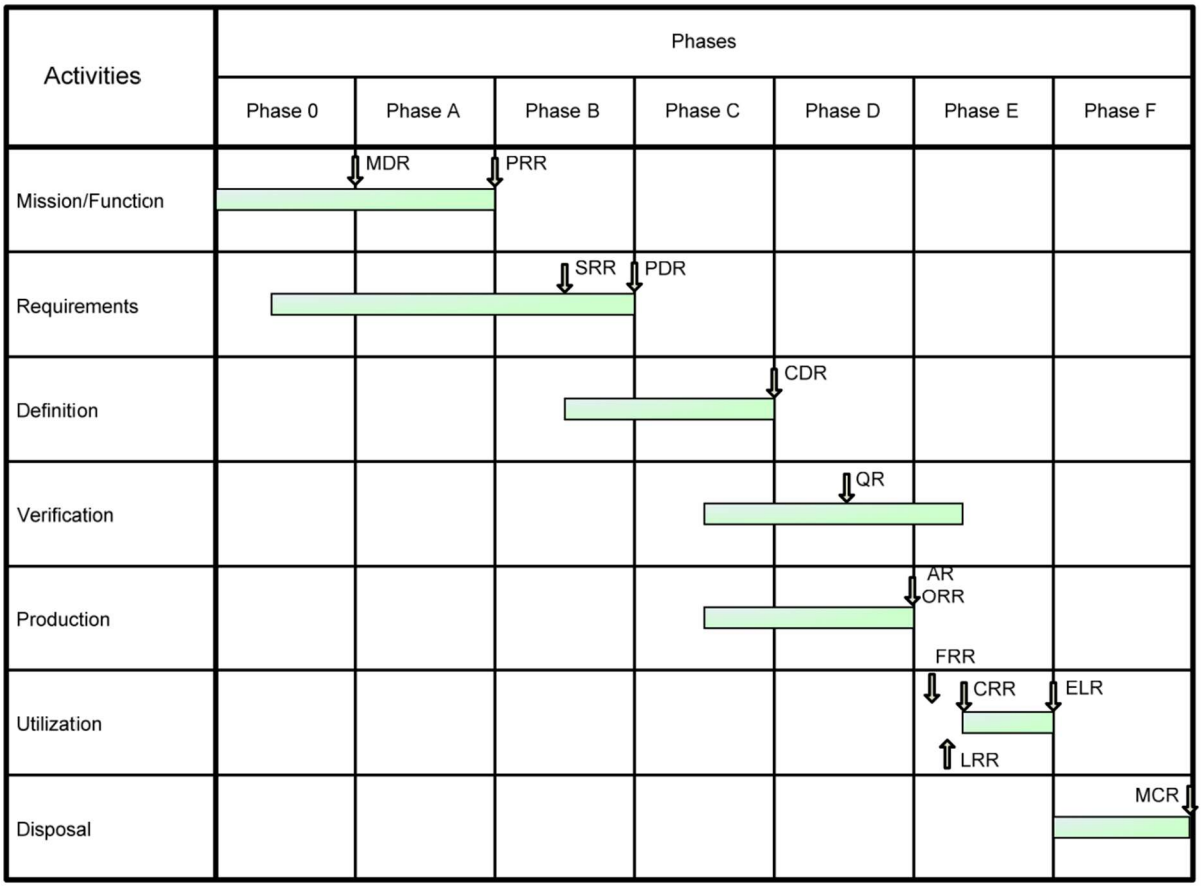

Figure 6: Typical project life cycle of an ESA project as per [AD3]

The review process can be tailored for smaller or less formally regulated projects. Instead of big-bang reviews, incremental or rolling reviews may be more appropriate:

- For example, a sprint review series with stakeholders that covers all PDR topics by the time the project reaches that stage.
- It might still be advisable to do an internal documentation audit at a midpoint.
- The key is "right-sizing" the process: ensure enough rigor (no critical aspect goes un-reviewed) but avoid unnecessary paperwork that doesn't add value. For instance, CubeSat payload software might not need a full formal SRR (think "New Space") if the team is small and in constant communication, but it is still advisable to do a structured walkthrough of requirements to catch omissions.

We integrate reviews within the agile workflow: every code PR is a mini-review (by peers). Requirements and design discussions happen in backlog grooming (which is like a continuous SRR/PDR in pieces). By the time we get to a formal review, there should be no surprises – it's more of a formality to satisfy external governance since, internally, we've been reviewing all along.

Benefits of embedded reviews:

- They provide early feedback on deliverables – catching issues when they are cheaper to fix.

- They enforce discipline continuously rather than just at gates – quality is maintained throughout.
- They familiarize stakeholders (like the client) with progress, reducing the chance of significant changes being requested later.
- As a concrete practice, it is good to maintain a "review checklist" at the end of each sprint or major feature: "Is there a test for every new requirement? Did we run a static analysis? Are the docs updated?" This is like a mini-audit that keeps everyone ready for formal audits.

Figure 7 through Figure 10 illustrate an example of this robust yet streamlined review and quality assurance approach, suitable for projects with less stringent regulatory constraints.

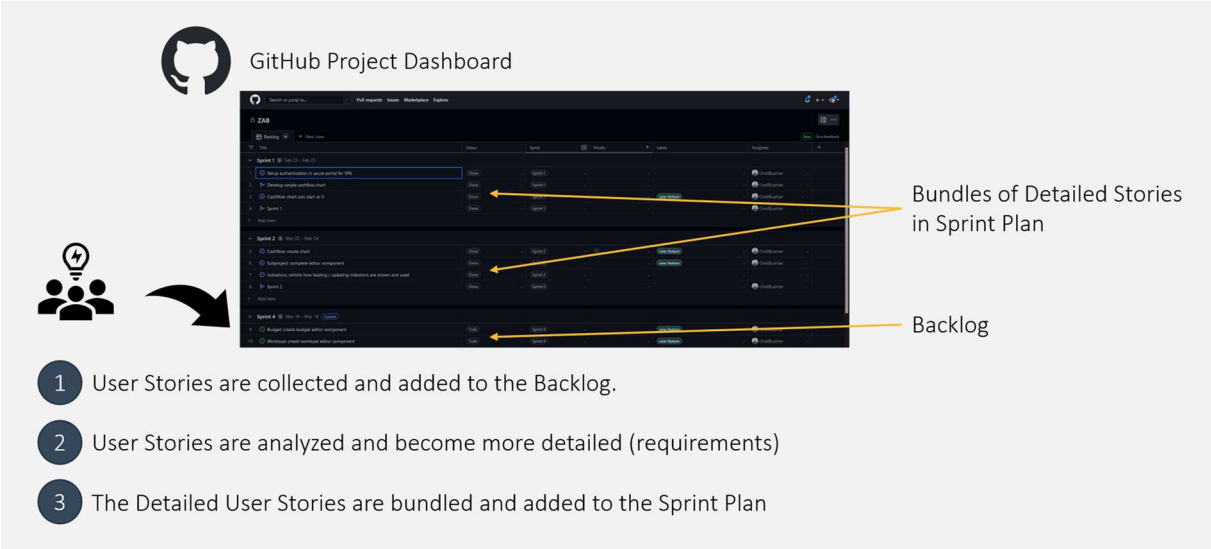

Figure 7: Requirement analysis, design, and planning process with GitHub dashboards.

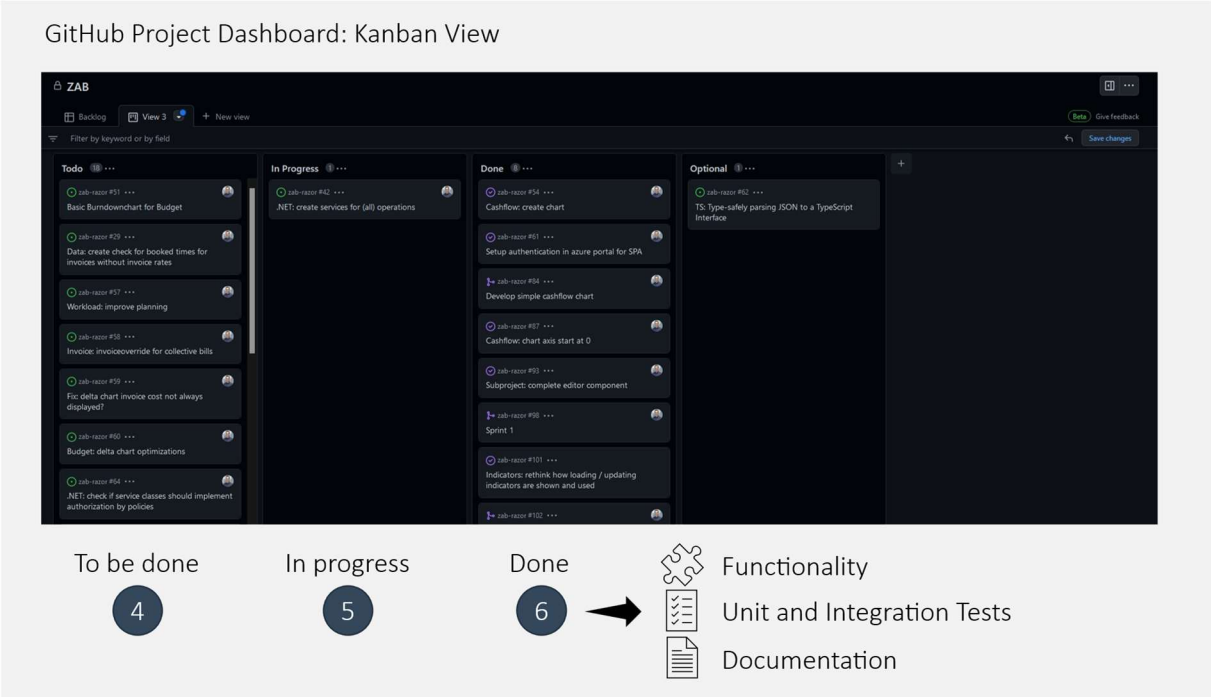

Figure 8: Implementation and the definition of "Done" in the project.

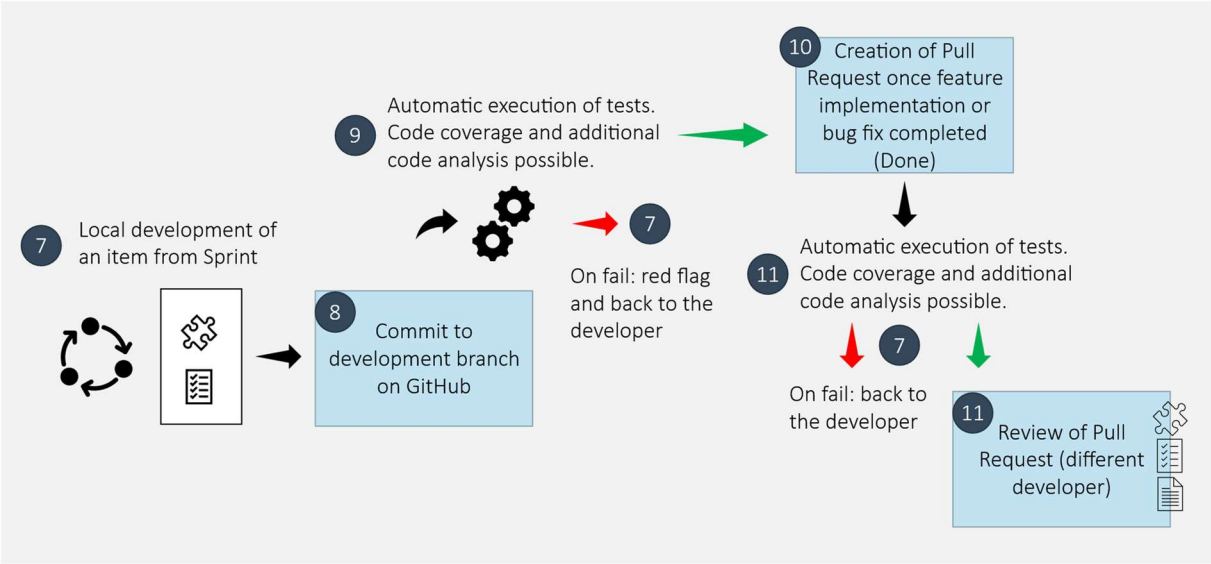

Figure 9: Source code control and review (continuous V&V) in the project.

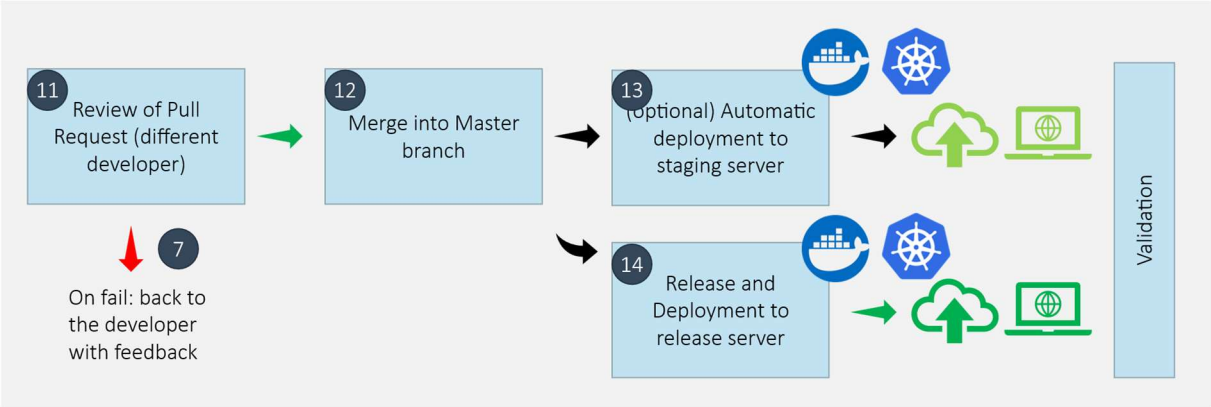

Figure 10: Continuous deployment in the project.

4.4.5. Guidelines for Writing Effective Requirements

The following guidelines are adapted from NASA's "Appendix C: How to Write Good a Requirement" [AD7] a widely recognized reference for creating clear, precise, and verifiable system requirements. Incorporating best practices from NASA helps ensure your project's requirements are well-defined, consistent, and effective, ultimately leading to successful system development and validation. This approach is in line with ECSS-Q-ST-80C requirements on traceability and verification coverage.

4.4.5.1. Terminology

- **"Shall"** – Indicates a mandatory requirement.
- **"Will"** – States a fact or declares intent.
- **"Should"** – Describes a goal or recommendation.

4.4.5.2. Structure and Style

- Use active voice and clearly state the subject acting (e.g., "The system shall measure...").
- Clearly identify the responsible party for personnel requirements ("Party X shall perform Y").
- Keep product requirements focused on what must be achieved, not how to achieve them.
- Use consistent, defined terminology throughout your document.
- Provide precise tolerances for quantitative or performance specifications.
- Avoid operational details or implementation specifics (describe "what" is needed rather than "how" to do it).
- State requirements positively whenever possible (avoid negative phrasing like "shall not").

4.4.5.3. Clarity and Completeness

- Write concise, simple, and unambiguous statements.

- Each requirement should express a single thought; avoid combining multiple requirements into one statement.
- Clearly define all assumptions and provide a rationale where needed.
- Minimize the use of "To Be Determined" (TBD). Instead, use best estimates and mark them as "To Be Resolved" (TBR), with clear actions, responsible persons, and deadlines to resolve them.

4.4.5.4. Validation Checklist

Ensure requirements meet these key criteria:

- **Clarity:**
  - Avoid ambiguous terms ("as appropriate," "and/or," "etc.").
  - Clearly specify subject and predicate; avoid indefinite pronouns ("this," "these").
- **Completeness:**
  - Confirm that all relevant requirement areas (functional, performance, interface, environment, safety, security, maintainability, reliability, etc.) have been addressed.
  - Explicitly state and verify all assumptions.
- **Compliance:**
  - Requirements should be at the correct level (system, subsystem, component).
  - Avoid implementation details, operational procedures, or personnel assignments within requirements.
- **Consistency:**
  - Ensure no contradictions exist between the requirements and related systems/documents.
  - Maintain consistent terminology aligned with the project glossary.
- **Traceability:**
  - Each requirement must clearly trace back to higher-level requirements, mission goals, or operational needs.
  - Confirm each requirement is necessary; distinguish between "needs" and "wants."
- **Correctness:**
  - Verify technical feasibility and accuracy of all requirements and assumptions.
- **Functionality and Performance:**

- Confirm that the requirements fully define the necessary functions to meet system objectives.
- Provide realistic and justified performance specifications and tolerances.

- **Interfaces:**
  - Clearly define all internal and external interfaces.
- **Maintainability and Reliability:**
  - Specify maintainability and reliability requirements clearly and measurably.
  - Include error detection, handling, recovery, and responses to undesired events.
- **Verifiability/Testability:**
  - Requirements must be testable via inspection, demonstration, analysis, testing, etc.
  - Avoid vague, unverifiable terms (e.g., "user-friendly," "easy," "robust," "fast," "adequate").
- **Data Usage:**
  - Clearly state "don't care" conditions when applicable to improve design clarity and portability.

4.4.6. Requirements Identification and Traceability

Clear and structured requirement management is crucial for effective project execution and quality assurance. We follow a pragmatic yet rigorous approach to defining, naming, and tracing requirements, aligning closely with established industry standards such as ECSS and NASA's recommendations. As every requirement must be verifiable, well-managed requirements can be traced to their verification method and result.

4.4.6.1. Requirements Identification and Naming Convention

Use a hierarchical identifier for each requirement to ensure uniqueness and context. A typical format is:

```
DOC-SECTION-COMPONENT-NNN
```

Where:

- **DOC** indicates the document or source. For example:
  - SSS (System/Subsystem Specification) for system-level requirements.
  - SRS (Software Requirements Specification) for software-specific requirements.
  - ICD (Interface Control Document) for interface requirements.

- **SECTION/GROUP** is an optional category or grouping, often based on functionality or subsystem. For example, "COMM" for communications, "CTRL" for the control system, etc.
- **COMPONENT** is optional for projects with multiple distinct components or modules. It's useful to denote which component a requirement is for. For instance, in a combined software subsystem, we might prefix it with the subsystem name.
- **NNNN** is a zero-padded sequence number (usually 3 or 4 digits) unique within that document or group.

Example: A high-level system requirement might be SSS-COMM-0001 ("The spacecraft shall support X Mbps downlink rate"). A derived software requirement in the communications software module might be SRS-COMM-COMMSOFT-0010 ("The Comm software shall implement CCSDS telemetry frames for downlink"). This way, by looking at the ID, you can tell where the requirement lives and roughly what it concerns.

Maintain this convention in requirement management tools or even in spreadsheets. It helps in discussions ("Requirement SRS-COMM-0100 is not met by the current design, what do we do?" clarifies what you're referring to).

4.4.6.2. Requirements Traceability and Management

We ensure every requirement is tracked from origin to verification:

- Maintain a Requirements Traceability Matrix (RTM), which often transforms into a Verification Control Document (VCD), often as a table or a set of linked tables. This matrix maps each high-level requirement (like system requirements in SSS) to one or more lower-level requirements (in SRS or derived requirements) and further to design elements (in design docs) and test cases (in test plans).
- If the project is small, a spreadsheet can manage traceability. For larger projects, it might be advisable to use a tool like IBM DOORS, Jama, or even just a well-structured Jira project with links, depending on client preferences. We have even built internal lightweight tools to track requirements and their status.
- Each requirement in the RTM has fields: unique ID, description (abbreviated), origin (which higher-level requirement or use case it is derived from), status (draft, validated, implemented, tested, etc.), owner (who is responsible), and verification method (inspection, test, analysis, demo).
- Regularly update this matrix as development progresses. For example, when a test case is written to verify requirement X, we update the matrix to link X to that test case identifier and later mark it verified when the test passes in a formal test campaign.
- During design and code reviews, refer to the matrix to ensure no requirements are forgotten. Conversely, if someone proposes a new feature, we ask, "Which requirement does this satisfy?" If none, it might be out of scope unless a new requirement is added and approved.

Though it sounds heavyweight, this traceability approach is lightweight and tool-supported not to impede agility. It is aligned with ECSS and NASA guidelines, which mandate bidirectional traceability (you can trace from requirement to implementation and vice versa). Handling it continuously rather than at the end isn't an overwhelming task.

This approach ensures:

- **Full transparency and control:** At any point, it is clear how much of the system is implemented and tested regarding coverage requirements.
- **Change management:** If a requirement changes or is added, the matrix highlights what design and tests need updating.
- **Simplified verification compliance:** When it is time to test, we have ready lists of tests needed for each requirement, satisfying ECSS-Q-ST-80C and NASA's expectations for V&V traceability.

4.5. Testing and Verification

Testing and verification are the primary means by which we ensure the software meets all requirements and performs reliably. Follow a multi-level testing strategy integrated with development, meaning testing isn't a phase at the end but a continuous activity that accompanies coding and a distinct set of activities for formal verification. NASA's software engineering requirements [AD4] similarly require projects to perform unit, integration, and system testing; our multi-level testing approach ensures compliance with such stringent criteria.

We typically produce two build configurations for the software:

- The **DEBUG build** includes extra checks, logs, and assertions enabled. Used in development and internal testing (e.g., ELF file). Make sure to keep any other debug-related files like memory maps, etc. to help analysis. DEBUG builds can be slower because they are build with fewer compiler optimization levels. Evaluate which compiler flags provide a good balance between speed and debugging performance (e.g., inlining can mess with breakpoints).
- The **RELEASE build** is optimized, with debug assertions disabled or reduced, and used for final delivery. Careful to remove any dynamic verification (see Section 4.5.3) from the code to avoid accidentally triggering assert statements and causing a system reset. NB: Certain compiler optimizations may not be allowed depending on the mission requirement (can impede patching strategy).

Both builds are based on the same code base and implement the same functionality. Thus, a bug found in RELEASE can be diagnosed by reproducing it in DEBUG with more diagnostics. This is important for space: if something goes wrong in orbit (with the optimized build), we must reproduce it on the ground with debug instrumentation.

The extent of testing (how deep we go into formal methods and how much hardware testing is needed) is decided early in the project based on criticality and resources. Table 7, which had been produced for flight software technical documentation, summarizes which types of tests apply in which environments (e.g., unit tests are easily done in pure software; hardware-in-the-loop tests need hardware).

Table 7: Sample applicability matrix of different testing methods in different execution environments. X = Easily applicable, (X) = Applicable

	Unit Tests	Static Verification	Dynamic Verification	Formal Verification	HW Tests	Performance Testing	Integration Testing	Regression Testing	Continuous Integration
Code		X		(X)					
Digital Twin	X		X			X	X	X	X
SDM	(X)		X		X	X	(X)	(X)	(X)
(QM)	X		X		X	X	X	X	
(P)FM	X		X		X	X	X		

4.5.1. Unit Tests

Unit testing is the backbone of our verification strategy. Each software unit (typically a function, class, or module) is tested in isolation with a set of unit tests. The goal is to verify that each unit's logic is correct for nominal and edge cases. Write adversarial tests, i.e., try breaking the code and don't try to make the test succeed. Productivity tools like LLMs can be used to create basic unit tests [RD11].

- Depending on the language, **frameworks** like CppUTest, catch2, Google Test for C/C++ code, JUnit for Java, etc. can be used. The tests are coded and thus integrated into CI for automation.
- **Automation:** Unit tests run on each commit via CI. They are meant to be fast (a suite should run in seconds to a few minutes) to give quick feedback.
- We differentiate **white-box vs black-box** unit tests:
    - **White-box** unit tests are written with knowledge of the code internals, often by the developer (e.g., testing internal functions or using dependency injection to force specific paths). We encourage TDD to write these tests as you write the code.
    - **Black-box** unit tests treat the unit as a "black box," testing only the public interface against its specification. These might be written by someone else or later to validate that the unit meets its requirements.

- Strive for high coverage in critical modules via unit tests. While 100% code coverage is not always practical or meaningful, aim to achieve a meaningful level of 70-90% on core algorithms. Pay attention to covering all branches of critical decision logic.
- Consider parameterized tests and random (fuzz) inputs for robustness testing at the unit level, especially for parsing or math functions.
- Unit tests often run in a host environment (e.g., x86 Linux), so simulate embedded aspects where needed, using stubs for hardware access.

The benefit of unit tests is immediate bug detection. If a recent change causes a previously passing unit test to fail, it is caught in a regression. This drastically reduces debugging time, as issues are detected close to their source and introduction time.

Finally, Large Language Models (LLM) can assist in the development of unit tests, like Anthropic's Claude Code [RD11]. Special care must be taken to ensure that generated code is meaningful and exercises the code under test properly.

4.5.2. Static Verification

Static verification involves analyzing the source code **without executing it** to find potential issues. This is extremely valuable in embedded systems, where specific bugs (like null pointers or overflows) can be catastrophic but might not easily manifest in tests.

- **Compiler warnings as errors:** Compile with a high warning level (e.g., -Wall -Wextra in GCC for C/C++) and treat warnings as errors. This catches many issues (unused variables, type conversions, etc.).
- **Static analysis tools**, like **PVS-Studio, Qodana**, **Cppcheck,** or **Clang-Tidy:** These tools analyze code paths and find uninitialized variables or violations of coding standards. Integrate these into CI so every commit gets analyzed. If new warnings are introduced, the CI, a quality gate, fails, and we require it to be fixed or marked as a false positive with justification.
- **Static stack usage analysis:** Knowing stack usage is essential for embedded systems (especially with RTOS tasks having fixed stack sizes). Use static analysis or linker features to determine the upper bound stack usage of each thread. To make this tractable, avoid recursion and unbounded stack allocations.
- **Coding standards compliance:** Adhere to standards like MISRA C recommendations and use static analysis to check compliance (some tools have MISRA checkers). This ensures that no dangerous type casts, use of goto, direct pointer arithmetic, etc., slip in, which comply with, e.g., NASA Power of 10 rules for C. You may also want to have a look at JPL's C Coding Standard (found here [RD16]).
- **Security/static vulnerability analysis:** Though less of a concern for offline embedded code, still check for things like buffer overflows or injection risks, especially if the software has

any interface that processes external input (telecommands are essentially external inputs, so we treat those parsing routines with care).

By integrating static analysis into the CI pipeline (Chapter 5 covers this), we ensure every build is scrutinized. This significantly reduces runtime errors because many are eliminated before the code runs. It also helps satisfy standards: e.g., NASA's 8739.8 standard strongly encourages static analysis, and ECSS-Q-ST-80C mandates static verification be done.

4.5.3. Dynamic Verification

Dynamic verification means runtime checks and tests that catch issues during program execution when the software is running (especially those related to timing, memory usage, or interactions of multiple components). You may employ several dynamic verification techniques: Throughout the code, especially in debug builds, you can use assert() or custom macros to validate assumptions – for instance, after computing a number to check it's within an expected range or assert that a pointer returned from a function is not null before use. These assertions will immediately stop the program (in debug mode) if violated, alerting us to a problem. During operations, e.g., during flight (release builds), it is paramount to either remove them or convert them to error logs/telemetry, depending on criticality (for critical must-not-fail assumptions, sometimes they can be left active even in flight but handle the failure by resetting the component or switching to safe mode).

- **Error handling paths:** Design the software to log it and either recover or go to a safe state if something abnormal happens. Induce errors during testing (especially integration and system testing) to ensure these paths work (like unplugging a sensor or feeding malformed data).
- **Runtime tools:** Use tools like Valgrind (for memory checking, if running the code on a PC) or address sanitizers (ASan, if we can run code with sanitization enabled) to catch memory misuses like leaks or out-of-bounds at runtime. Thread sanitizers (TSan) can catch data races during tests for threads and concurrency. Depending on the compiler, not all tools are available (e.g., when compiling an RTEMS-based system with the rtems-gcc).
- **Fuzz testing:** Use fuzz testing (tools that generate random input variations) to break the code for inputs like file parsers or communication packet handlers. This is dynamic because it runs the code with many inputs to see if any crash or misbehavior occurs.
- **Resource monitoring:** In long-running or HIL tests, monitor CPU usage, memory usage, etc., to spot leaks or performance issues.
- **Recovery and restart tests:** Test how the system behaves if a process is restarted or an MCU is power-cycled (to ensure no persistent bad state across reboots).

Dynamic verification complements static analysis – where static might not prove something fully or doesn't know actual values, dynamic verification checks catch them in action. A trivial example: static analysis might not flag a division by zero if it's not apparent, but a dynamic assert can catch if a denominator ever becomes zero during tests. Formal verification, however, may also catch the division by zero (see next section).

Dynamic verification can also be used, e.g., to ensure the detection of array overflows by adding an extra byte at the end of an array or after the array inside a struct and using that variable as a guard with a known pattern, like 0xDEADBEEF. Then, at regular intervals, use asserts() to evaluate if that (otherwise unused) memory location is still unchanged or at the top of the program execution. This technique has been used extensively when looking for issues during the program execution of STIX (see Section 8.1).

For safety-critical parts, we may leave dynamic checks in the production code. For example, avionics software might continuously monitor its outputs; if something goes out of bounds, it triggers a safe mode. This is dynamic checking, ensuring the system stays within safe operation.

4.5.4. Formal Verification

Formal verification uses mathematical methods to prove the software's properties. Unlike testing (which can show the presence of bugs, not their absence), formal methods can guarantee that specific bugs are not present in all possible executions. Due to the high effort involved, we selectively apply formal verification to the most critical algorithms or modules.

Approaches we have used:

- **Property specification:** Formally state what the code should do (e.g., "This function's result will always be within this range," "This routine will always terminate within N steps," or "No memory overflow occurs in this module"). This can be done with pre/post conditions and invariants in the code (using annotations or a formal spec language). Some properties are implicitly defined, in cases like divisions by zero.
- **Automated theorem proving or model checking:** We have experience with tools like Stainless (EPFL's framework for formal software verification on Scala), where we write specific modules in a subset of a language the tool can handle (Scale in the case of Stainless) and then specify correctness properties. The tool then tries to prove those properties. In a project with EPFL, we translated critical C code (parts of the STIX file system; see Section 8.1) to Scala, used Stainless to prove the absence of runtime errors and certain functional correctness properties, and then translated it back to C. This gave firm confidence in the file system – and we found subtle issues that testing hadn't.
- **Formal analysis of state machines:** If you are using state machine formalisms (like SDL or TLA+ for concurrency), you might model a protocol or critical sequence in a formal tool and verify its liveness/safety properties (e.g., a command handshake will never deadlock).
- **Absence of runtime errors:** Some static analysis tools (like Astree or Polyspace) can prove the absence of specific runtime errors for C code (standard in the avionics industry). Use such tools in critical routines to verify no division-by-zero, out-of-bound array access, etc.

Other tools used for program verification are Viper [RD17] and Tamarin [RD18] (though the latter is more used for security protocol verification).

The output of formal verification is often a report that states which properties were proven for which code. This is great for assurance arguments (e.g., we can say, "The control law is proven stable by design" or "The memory manager is proven never to leak memory").

However, formal methods require **expertise and effort**:

- You need someone who understands formal specifications and can model the problem correctly.
- The code may need to be written in a restricted subset or annotated heavily.
- It can be time-consuming to get complex properties to verify (often an iterative process tweaking the proof or adding lemmas).

Therefore, we apply it **where the payoff justifies it:** typically, a small piece of highly critical code (like a scheduler, a core math algorithm for guidance, etc.). Combining tests and static analysis is usually sufficient and more cost-effective for less critical parts.

One practical outcome is that the code can be treated with high confidence after formally verifying a module. Test it still, but perhaps reduce the testing needed elsewhere because we trust the formally verified part (e.g., treat it as a black box with guaranteed behavior).

In summary, formal verification is part of a great toolbox, used selectively to bolster confidence where traditional testing might not be exhaustive enough and where errors would have a very high impact. It aligns with the highest levels of integrity in standards like ECSS or DO-178C (which, at Level A, encourages formal methods to comply).

4.5.5. Hardware Tests

Hardware testing is crucial because software behavior can differ on real hardware compared to simulations. Timing, memory layout, and hardware peripherals can all cause issues that won't appear in a simulated environment.

Our hardware testing approach (often referred to as Hardware-in-the-Loop, HIL):

- If available, use a **Breadboard, Engineering Model**, or prototype of the actual flight hardware. If not, use development boards as closely as possible (same CPU, similar peripherals).
- Develop **automated test scripts** to run on the hardware. For example, a Python script on a PC that interacts with the board via a serial port or JTAG to deploy the software, send test commands, and read telemetry.
- Many unit and integration tests from a well-through-out **CI can be re-used on hardware**. You might compile the test code to run on the board or wrap tests in special telemetry that the test script can interpret.
- Specifically test **hardware-dependent aspects**: device drivers ("Can we read sensor data correctly?"), real-time performance ("Does an interrupt meet its deadline?"), and robustness (power cycling the device repeatedly, checking for proper startup).

- **Environmental testing:** When possible, involve the hardware in environmental tests (thermal vacuum, vibration). The software is exercised during these to ensure, for example, that boot sequences work at extreme temperatures or that no memory corruption occurs due to radiation (if we can do radiation testing).
- **Regression on hardware:** You might set up nightly tests for critical systems where the latest build is flashed onto a spare board and a suite of tests run overnight. This catches issues like memory leaks (long-duration tests) or rare glitches.

One challenge is limited hardware availability. Often, we have only one or two EM boards, so we can't use them for every CI commit (hence virtualization, Chapter 6, is essential). Instead, we schedule hardware tests periodically (like daily or for each release candidate build). Another challenge is that hardware is stateful (e.g. flash memory), and it might be difficult to reset hardware to a known, consistent state. Ideally, there are hardware features available to support state resets, e.g. by allowing externallycontrolled power cycling.

Hardware tests bridge the gap between continuous virtual tests and the real world, increasing confidence that "What we test is what we fly" – a NASA mantra. They also often uncover issues with hardware interfaces (e.g., a sensor giving unexpected values causing the algorithm to misbehave or a difference in floating-point precision on the target CPU).

Document each hardware test result and incorporate it into the traceability: e.g., if the requirement says, "The system shall acquire sensor data at 10 Hz," have a hardware test that measures that on the actual board and link that test back to the requirement.

4.5.6. Performance Testing

Performance testing ensures the software meets non-functional requirements such as timing, memory usage, and throughput, which are vital in embedded systems with constrained resources.

Perform performance testing at multiple stages:

- **Unit micro-benchmarks:** For key algorithms, you might have micro-benchmarks that run the function with sample data and measure execution time or memory usage. This can be part of unit tests (e.g., assert that it runs under X milliseconds for Y-sized input).
- **Profiling on target:** Measure CPU utilization of each thread or major cycle on the actual hardware or a cycle-accurate simulator. Detect and optimize if something is over budget (like a control loop missing its 50 Hz deadline).
- **Memory usage:** Track static memory (by analyzing map files) and dynamic memory (by instrumenting allocators or using built-in RTOS features to check heap usage). Ensure you have sufficient margin as per requirement (e.g., no more than 70% of RAM used, so the remainder is margin for fragmentation or future growth).
- **Throughput tests:** For example, feed the maximum expected data rate through the system (telemetry packets, images, etc.) and see if any backlog occurs or data is dropped.

- **Power consumption (if relevant):** Sometimes software can impact power (e.g., by keeping CPU at 100%). We might test power usage in different modes.

Performance tests can be integrated into CI to some extent (especially on a simulator or using QEMU with instruction counters). But final performance tuning is done on hardware with proper instrumentation.

Results from performance tests are compared against requirements or targets. If there are deviations, treat them as bugs to fix (either optimize code, adjust the requirement if it is unrealistic, and get stakeholder buy-in or upgrade hardware if feasible – though in space, that's rare, and hardware is fixed early).

Performance testing ties into our quality gates. For instance, if new features are creeping up worst-case execution time, watch it and take corrective action before it jeopardizes the mission schedule (finding a performance issue late can require major refactoring; our continuous approach avoids that).

4.5.7. Integration, System, and Scenario Testing

Beyond individual units, we test the interactions of components (integration testing) and the whole system's behavior under realistic scenarios (system testing). Scenario testing often involves end-to-end sequences that the actual system will perform (mission scenarios).

- **Integration testing:** Gradually integrate modules and test their interaction. For example, test the software's communication stack end-to-end by sending a telecommand from the ground software to the onboard software (in a lab setup) and verifying the correct telemetry response. Integration tests might still be automated and run in CI (especially with virtual setups).
- **System testing:** Once the entire software stack is integrated on a representative platform, run system tests, which treat the whole software (and possibly hardware) as a black box to validate requirements. This could include a nominal mission timeline (boot, deploy components, perform operations, etc.) and off-nominal scenarios (component failure simulation to see if FDIR works).
- Use test scripts and possibly test frameworks (e.g., using **EGSE software and hardware** to send commands and verify telemetry).
- **Scenario examples:** A calibration sequence, a science data collection pass, a safe-mode entry, and recovery are tested. Larger setups allow for entire mission simulations, like NASA's NOS3 [RD14].
- Involve **domain experts** (e.g., ops team members) in scenario testing to ensure it meets operational needs.

System testing is often formally done during validation phases (e.g., system validation tests) and witnessed by the customer or independent verification teams. Ensure all mission requirements and use cases have corresponding system tests.

Traceability: Every requirement from the system requirements (especially those relating to functionality or performance) is verified by one or more system tests, which we mark in the RTM or VCD.

It is also advisable to include **stress testing** in system tests, e.g., continuously running the system for 72 hours to see if any issues like memory leaks or counter overflows occur (important for long-duration missions).

4.5.8. Regression Testing

Regression testing is not a separate type of testing but rather the practice of re-running all relevant tests whenever changes occur to ensure new changes haven't broken anything that worked before.

Our approach to regression testing is:

- **Automate it in CI:** Every time code is merged, the CI runs the full suite of unit tests and often a subset of integration tests on the simulator. This catches regressions immediately (e.g. if a math library change causes a control test to fail).
- **Regular complete test cycles:** Schedule full regression test campaigns at key points (e.g., before a major release or delivery). This might include re-running all system tests, including those that are too manual or hardware-involved to run on every commit.
- **Baseline comparison:** For performance metrics or outputs, it can be helpful to store known good outputs (baseline data) and compare current test outputs to those. Differences can indicate regressions.
- **Continuous regression tracking:** If a test fails, it is advisable not to merge the code but to fix it first. For long projects, maintain a dashboard of test results over time.

One good practice is that for any bug found (either during testing or, heaven forbid, in operation), write a new test case that would have caught that bug and include it in the regression suite. This way, a specific problem can never reoccur without us noticing.

By the end of development, you will have a comprehensive regression suite that gives confidence for maintenance updates and software reuse in future projects (with modifications).

4.5.9. Continuous Integration (CI)

NB: Continuous integration and related topics are so important that we dedicate Chapter 5 entirely to CI/CD. Here, we give a brief overview in the context of testing to show how it fits into the verification strategy.

Continuous integration frequently merges all developers' changes to a shared mainline, and automated build and test steps verify each merge. In our framework:

- **CI ties together many of the above activities**: When code is pushed, the CI server automatically compiles the code, runs static verification (section 4.5.2), runs unit tests (4.5.1),

maybe some integration tests (4.5.7) in a simulated environment, and perhaps even package the software.

- **CI provides rapid feedback:** Developers are alerted within minutes if their change breaks something. This ensures issues are addressed immediately, not weeks later.
- **Treat the CI status as a criterion for code integration**: A change must pass CI (all tests green), or it's not done. This enforces a culture of keeping the building "green" (always passing). Still, write adversarial tests that try to break the code rather than returning "OK."
- **Configure the CI to generate artifacts** (e.g., compiled binaries, test reports, coverage reports) archived for traceability. For example, we can later retrieve the exact binary that passed all tests at CDR.

Notably, the CI pipeline shall be set up to mirror the verification stages of the V-model continuously. This is like doing a mini-V&V cycle on each commit, significantly reducing the integration effort because you constantly integrate and verify.

Some specifics on what to do in CI (which will be expanded in Chapter 5):

- Build in a **clean environment** (like a Docker container) to ensure no dependency issues.
- **Cross-compile for the target**, then run tests in a simulator (like QEMU for the target CPU).
- **Possibly run hardware tests** in CI if hardware is available and automation is set (often as nightly due to time).
- Use **coverage analysis** in CI to see if a commit drops test coverage (as a quality gate).
- Use **linting and formatting checks** to keep code style in relevant parts consistent.

The CI system itself is under configuration control (the "Pipeline as code" concept), meaning changes to CI scripts are tracked and reviewed as part of our process.

In essence, CI is the glue that binds your development and testing together, ensuring verification is continuous, not just a phase. It supports agile (fast iteration) and standards compliance (evidence generation).

4.5.10. Independent Software Verification

As an additional measure towards software quality, Independent Software Verification (ISV) is commonly employed in space software projects to provide an impartial and rigorous assessment of the software product. ISV involves verification activities conducted by a team independent from the original software developers, thereby reducing biases and ensuring a fresh perspective in identifying software defects, inconsistencies, or deviations from specified requirements. By performing independent reviews, formal inspections, static analysis, and dedicated testing, ISV helps ensure compliance with stringent standards and enhances the overall reliability and robustness of the space software system.

This practice is explicitly recommended by established standards such as ECSS-E-ST-40C and is recognized as a best practice within the aerospace software community.

5. CONTINUOUS INTEGRATION AND CONTINUOUS DELIVERY (CI/CD) FOR SPACE SOFTWARE

Continuous Integration and Continuous Delivery (CI/CD) form the backbone of modern software development practices, enabling teams to detect issues early and deliver updates quickly and reliably. In the context of embedded space systems, CI/CD must be carefully tailored to handle cross-compilation, hardware-in-the-loop testing, strict reliability requirements, and the traceability demands of space industry standards. This chapter provides a detailed look at implementing CI/CD in our projects, complementing the framework described in Chapter 4 by automating and enforcing many of its practices.

5.1. Overview

Continuous Integration (CI) involves frequently merging all developers' code changes into a central repository (often multiple times per day). Each merge triggers an automated build and test process. The goal is to find and address integration issues (like conflicting changes or failing tests) as early as possible. In effect, CI turns integration – which used to be a big bang at the end of a project – into a routine, incremental activity.

Continuous Delivery/Deployment (CD) extends CI by automatically deploying (delivering) the integrated changes to an environment (or releasing them to users) after passing the pipeline. In the case of flight software, "deployment" might mean flashing to a test device or packaging a flight software build for delivery to a spacecraft or another team.

For an instrument software team, a well-designed CI/CD pipeline yields several benefits:

- **Early and Continuous Verification:** Automated builds and tests provide rapid feedback on each code change, helping to detect integration issues, requirement mismatches, and regressions early when they are easier and cheaper to fix. This supports ECSS-Q-ST-80C's principle of iterative, multi-level verification.
- **Reproducible, Controlled Builds:** By running in isolated environments (e.g., Docker), pipelines ensure identical results for identical inputs, eliminating host-specific inconsistencies and supporting configuration control.
- **Increased Reliability Through Automation:** Replacing manual steps (e.g., compiler setup, packaging, signing) with automated scripts reduces human error and enforces consistent execution, which is critical for complex embedded systems.
- **Traceability and Compliance:** CI/CD systems generate detailed logs that map code commits to test results and requirements. This facilitates automatic traceability reporting and supports compliance with ECSS and NASA verification traceability requirements.

- **Rapid Iteration and Collaboration:** Developers can integrate more confidently, knowing CI catches regressions. Frequent merges promote collaboration, reduce integration conflicts, and accelerate development cycles.
- **Operational Readiness:** The software always remains in a deployable state. While actual spacecraft deployments occur at defined milestones, this readiness enables fast reaction to late-breaking issues and supports frequent internal demos or system tests.
- **Process Transparency and Insight:** CI dashboards provide real-time visibility into building health, test coverage, and compliance metrics, fostering team accountability and informed project management.

5.2. Embedded CI/CD Considerations

Unlike pure cloud or app software, embedded CI/CD faces:

- Hardware dependencies (you can't spin up a satellite in the cloud to test on; though, having said that, clever cloud virtualization services are becoming available for platforms). We address this with virtualization (Chapter 6) and a careful HIL testing strategy.
- Cross-compilers and toolchains must be managed so that a consistent environment includes GCC for ARM or LLVM for RISC-V, etc. As discussed later, this can be handled via containerization.
- Hardware tests have long test times. Mitigate this by splitting fast tests (run on each commit) into slow tests (nightly or on demand).
- Test the reliability of your tests because flaky tests can reduce confidence. Invest in making tests deterministic, using simulators that you control.
- Consider the pipeline environment's security. For example, if the code is ITAR/export controlled or proprietary, you might be unable to use public cloud CI and must rely on self-hosted solutions.

5.3. Common Challenges and Mitigations

- **Initial setup effort:** Setting up a robust CI/CD for embedded projects can be non-trivial. It requires writing build scripts, Docker images, possibly hardware interfacing scripts, etc. Mitigate this by maintaining templates and scripts from past projects (a CI "starter kit").
- **Maintenance:** CI pipelines can grow complex and slow if not managed. Continually refine the pipeline (as noted in Chapter 4, treating it as its product to improve).
- **Flaky tests:** Tests that sometimes fail due to timing or environmental issues erode CI trust. Test results must be made deterministic. In the worst case, quarantine flaky tests so they don't block the pipeline but get flagged for someone to fix.

- **Resource usage:** Running a complete pipeline for every commit can be resource-intensive (many CPU hours). Optimize by using caching (so rebuilds are faster) and scaling runners.
- **Cultural change:** Team members must adapt to writing tests and relying on the pipeline. Some may be initially resistant if not used to it. Address this via training and showing the benefits (and sometimes by mandate: "The pipeline must be green to merge").

5.4. Streamlined and Reproducible CI/CD Environment

A key prerequisite for effective CI/CD is having a reproducible, automated build and test environment. Any developer or CI runner can set up the environment and get identical results. We achieve this through cross-platform build systems, containerization, and scripting of all processes.

- **Cross-Platform Build Tools:** Use portable build systems (like CMake for C/C++, Meson, etc.) to generate platform-specific builds. This allows the software to be built on various host OS (Windows, Linux, Mac) and to target different architectures (via toolchain files). For example, CMake scripts can produce a Linux x86 build for host testing and an ARM Cortex-M build for a flight from the same source. This addresses ECSS-E-ST-40C guidance to maintain portability and avoid environment-dependent errors. It also means developers can reproduce CI builds locally by invoking the same CMake commands.
- **Containerization:** Encapsulate the entire build environment in a Docker container (or similar container technology). Create a Dockerfile that installs the cross-compiler, specific library versions, Python (if needed for scripting), etc. The CI pipeline then pulls/builds this container and runs the build inside it. This ensures the environment on the CI server is the same as that of a developer's machine using that container. If a new dependency is needed (say, you added a new library), update the Dockerfile (which is in version control), and everyone gets it. Also, by versioning the Docker image (tagging it with a version or commit hash), you can fulfill configuration management expectations of ECSS-Q-ST-80C, even if the build environment is under configuration control.
- **Hardware Emulation & Simulated Environments:** As detailed in Chapter 6, set up emulators like QEMU in the CI environment to run the compiled binary in a simulated target environment. For example, after cross-compiling, the CI launches QEMU with an emulated CPU/board to execute integration tests. Also, simulate peripherals where possible. This allows early detection of issues specific to the target (like endianness or alignment problems) as part of CI. It's not full HIL, but it covers a lot of ground with zero hardware. Table 7 shows that virtualization gets an "X" for many test types precisely because of this capability.
- **Automation and Orchestration:** All CI steps are scripted (in Jenkinsfile, GitHub Actions YAML, or GitLab CI YAML, etc.). There is no manual step. For example, a typical job in our pipeline might be:

    1. Checkout code
    2. Set up Docker (or use a pre-built CI image)

3. Run cmake .. -DCMAKE_TOOLCHAIN_FILE=... for target
4. Run cmake --build . to compile
5. Run static analysis tool script
6. Boot QEMU and run tests
7. Archive results (binaries, logs)
8. If all is good, mark build successful.

Everything above is defined in code, so it's repeatable. If a new developer joins, you can hand them a README that says, "Install Docker and run these two commands to replicate the CI." That quick start lowers onboarding time.

- **Hardware Interface Scripts:** Include scripts to interact with hardware for pipelines that involve actual hardware testing (say nightly). For example, you might use a lab PC connected to the board via USB. A Python script uses OpenOCD or ST-Link to flash the firmware and then uses a serial port to communicate and run tests. Keep these scripts in the repository as well. Over time, you will accumulate a library of such scripts (e.g., a power switch control script to power-cycle boards, a script to measure current from a connected multimeter, etc.). By automating hardware interaction, you can include hardware in CI (even if not every commit, it could be scheduled) and avoid manual testing errors.
- **Secret Management:** The CI sometimes needs secrets (maybe credentials to deploy artifacts to a server or access a proprietary compiler). Never store secrets in code! Also, keep your Git history clean with secrets. NB: Should you ever accidentally commit a secret (or even public staging or testing API endpoints or any other information that the public should not know, use tools to clean your history. Some platforms, like GitHub, have scanners to help you find leaked secrets and confidential information, like BFG Repo-Cleaner [RD15]). Instead, inject these securely at runtime using CI's secret store (like GitHub Actions Secrets, GitLab CI variables, or Hashicorp Vault integration). This ensures that our repository can remain public or shared without leaking sensitive info and meets security best practices.
- **Documentation of CI Setup:** Maintain a "CI/CD guide" or integrate it into your Configuration Management Plan documentation. It describes how the pipeline is structured and how to run it locally. This serves both the team and any external auditors or new team members. It also aligns with ECSS-E-ST-40C, which expects that development infrastructure configuration is documented and controlled.

By investing in this reproducible environment, you essentially "infrastructure as code" your development process. Newcomers can get the environment up quickly, and the CI can run reliably on different runners (self-hosted or cloud). You also gain the ability to resurrect any past build. Since you have the container (by version) and the code, you can recreate an old build even years later, which might be needed for anomaly investigation on orbit.

5.5. CI/CD Pipeline Workflow for Embedded Systems

With the environment in place, a CI/CD pipeline typically follows an extended version of the classic software pipeline, adding steps specific to embedded development. A generic workflow is:

1. **Code Commit & Merge:** A developer pushes code to a feature branch. When ready, a pull request is opened to merge to the main (or main development) branch. The CI pipeline triggers on the PR (and again on merge). Adopt a branching strategy (like GitFlow or trunk-based) to isolate development until checks pass. Only code that has passed all pipeline stages and review gets merged (quality gate).

2. **Automated Build & Code Analysis:** The pipeline compiles the code using the cross-compiler inside the container environment. It is advisable to build multiple configurations (Debug, Release). After building, run static analysis tools (like clang-tidy, cppcheck, and MISRA checker). Any issues cause pipeline failure with a report (for developers to fix). This step ensures you enforce coding standards right after coding and catch easy-to-miss bugs. It directly maps to ECSS-Q-ST-80C requirements for static verification and coding rules compliance. Many tools available can analyze the code during this step, like SonarCube or GitHub Dependabot (which, in fact, runs in parallel to the CI/CD pipeline directly on your code – make use of those tools).

3. **Unit and Integration Testing (Software-in-the-Loop):** Next, the pipeline runs the suite of unit tests (which don't need the target hardware) on the built binaries (some might run on a host or an emulator). Then, run integration tests that can be executed in a simulated environment, for example, using a QEMU-emulated board or a special test harness. Gather code coverage during these tests (using gcov or similar). The pipeline fails if any test fails or coverage drops below the threshold. This step gives you confidence that the software logic is correct and meets the requirements (since tests are derived from the requirements). It also aligns with standards: e.g., ECSS-Q-ST-80C expects verification at unit and integration levels with coverage analysis.

4. **Package & Deploy to Emulator:** If basic tests pass, take the compiled artifact (e.g., a firmware .elf or .bin) and deploy it to a virtual platform environment. For example, spin up a QEMU instance that emulates the satellite's CPU and possibly some peripherals. If possible, run a series of higher-level tests – maybe even the whole system test suite – in this emulator. For flight software, you might boot the RTOS and run a scripted sequence: send a telecommand, get a response, check a sensor simulation, etc. This step verifies that the software can run on the target architecture and handle realistic scenarios within CI. It's like a digital twin test run each time. If issues are found here (e.g., it crashes on startup in QEMU), the pipeline fails, and you can debug the low-level issues. This step is unique to embedded projects and is very powerful if set up (Chapter 6 covers how it can be done with QEMU, TASTE, etc.). NB: Invest time designing a good end-to-end debugging setup, e.g., using GDB!

5. **Hardware-in-the-Loop (HIL) Testing (as applicable):** Not every pipeline run will do this (due to hardware availability), but you might have a stage that waits for a hardware slot (maybe nightly or on a special runner with hardware attached). This stage would load the new

build onto an actual board or testbed and run a regression test suite. If you include this in CI, you should mark it as manual or scheduled (not every commit, but every night or merge to main). Failures here indicate hardware-specific problems. While not all projects can have this (some hardware can't be automated easily or is too scarce), regularly plan for at least one round of automated hardware testing.

6. **Results & Reporting:** Logs and results are recorded at each stage. The CI aggregates these and can even produce summary documents. For example:
    - A test report (perhaps in JUnit XML format or a custom dashboard).
    - A coverage report (which files have low coverage).
    - Static analysis report (list of warnings).
    - If all stages pass, the pipeline can automatically create a release artifact. This could be a firmware image zip with a naming convention (including git hash and build number), and maybe even push it to a repository or storage (so that you will have an archive of every "green" build).
    - If any stage fails, the pipeline immediately flags it (red status) and notifies developers (via email or chat integration).
    - Also integrate requirement traceability: e.g., your test cases can be tagged with requirement IDs, and the pipeline can generate a matrix of requirements vs test results, which is excellent for proving compliance in reviews.

This workflow ensures that integration problems are caught early and continuously. In essence, try to constantly exercise the right side of the V-model (verification and validation) within minutes or hours of changes rather than waiting for designated times. This continuous verification approach keeps the project compliant and shippable. It fully aligns with ECSS-E-ST-40C's recommendation of iterative testing throughout the lifecycle and NASA's [AD4] focus on continuous risk management.

5.6. Integrating CI/CD with Project Workflows

For CI/CD to deliver maximum value, it must be part of the team's daily routine and the project's overall systems engineering approach, not a parallel process. Here's how you can integrate CI/CD into our workflows and project management:

- **Alignment with V-Model & Reviews:** In Chapter 4 we discussed formal reviews (like PDR and CDR). Use CI/CD as evidence to support those reviews. For instance, by CDR, you might extract from CI a report showing that all requirements have at least one passing test case, proving verification progress. You might also freeze a build at CDR and present the CI artifact of that build as the "CDR delivery." CI outputs like coverage reports and static analysis trends can show improved quality. This way, CI/CD isn't just a dev tool; it's part of the formal qualification data.

- **Continuous Risk Management:** Treat persistent CI issues as risks. For example, suppose a test is flaky or fails repeatedly. In that case, it indicates an underlying uncertainty ("Is there a design flaw?", "Is a requirement not understood?", "Is resource margin too low?" etc.). In line with NASA [AD4] emphasis on risk management, log these in your risk register and ensure the team addresses them. CI can even provide metrics that feed into risk analysis, like "Module X always has lower test coverage – the risk of insufficient testing."
- **Developer Workflow Integration:** Encourage developers to run parts of the CI pipeline locally before pushing. Since they have containers and scripts, a developer can do "docker run your_image make test" to catch issues. Also, you could set up quick test targets, like "make quick_test," that run a fast subset (e.g., build and run tests on a native platform) so they catch apparent mistakes. Some teams integrate CI with IDEs (like running static analysis on file save, etc.). Keep it simple with a rule: don't push code that you expect will break the build. If the build breaks, fixing it is a top priority (no new features until green). Over time, developers internalize the CI checks.
- **Feedback & Continuous Improvement of CI:** Gather team feedback on the pipeline. If it's too slow, improve it (e.g., more parallel jobs, better caching). If false positives happen in static analysis, tune or suppress them responsibly (with justification in code). Set goals like "CI must finish in under 30 minutes" and "Keep master green 100% of the time". Track metrics such as average build time, most extended wait, success rate, etc. Many CI tools have dashboards that can be displayed on a monitor in the office or team chats. It gamifies quality – e.g., if coverage goes down, someone will notice and ask why. ECSS's Agile Handbook [RD1] even suggests tracking test "pass" trends.
- **Documentation and Handover:** The CI pipeline is documented so everyone knows how to run builds if the project is handed over to a different team or operations. This becomes part of the project's technical baseline. It's essential for long missions; maybe a patch is needed five years later. With your CI documentation and preserved environment, any future maintainer can rebuild the software precisely as you did, reducing the risk of mistakes.

5.7. Best Practices and Lessons Learned in CI/CD

Based on our experience and industry lessons (as reflected in ECSS's Agile Handbook [AD1] and other sources), here is a checklist of CI/CD best practices for embedded projects:

- **Start CI Early:** Set up a simple CI pipeline from the project's inception. Even if initially it just builds the code and runs one test on a simulator, it forces you to structure the project for automation. Incrementally expand it. This also surfaces integration issues when they are easiest to handle (early).
- **Keep Builds Fast:** Developers will shy away from CI if each run takes hours. Optimize the pipeline:
    - Use ccache or similar to cache compilation results between runs.

- Build only the changed components (split builds into multiple jobs, possibly).
- Run tests in parallel (most CI systems allow multiple executors).
- If you use Docker, cache the Docker layers for dependencies so you don't rebuild them each time. A rule of thumb: unit tests + build under 10 minutes for a typical commit. Longer-running tests can be less frequent. When pipelines are slow, developers begin to merge less often, which hurts integration.
- Run a subset of tests during development, and the full set on an asynchronous schedule. This way, common bugs are found quickly during development, but rare bugs are still found with additional latency.

- **Prioritize Reproducibility:** If a developer or auditor cannot easily run the build and tests on their machine, it's a red flag. Ensure a one-command setup (like "./setup_env.sh" to get Docker and run tests).
- **Automate Testing at All Levels:** Incorporate unit, integration, system (simulated), static analysis, and maybe property-based tests into CI. For example, if you have a requirement to "process telemetry packet correctly," write an integration test feeding a sample packet and checking output, and include that. If you have a requirement regulating code complexity, enforce a static metric. ECSS-Q-ST-80C requires a strategy for each test level – ensure each is represented in CI, even at different frequencies.
- **Embrace Simulated Environments:** As Chapter 6 details use simulators heavily. Have at least one job that runs the software in an emulator on each push. It catches integration issues sooner and reduces dependence on hardware (which may be limited). This would have saved us in a project where the hardware wasn't ready for a long time; we could have kept developing and testing in QEMU, and things mostly just worked when the hardware finally arrived. Sadly, the first development team did not invest in an emulated environment (see Section 8.1).
- **Implement Quality Gates:** Define the pipeline's clear pass/fail criteria beyond "tests pass." For example:
  - No increase in static analysis warnings.
  - Code coverage is not decreasing (or requirement coverage is 100% for new features).
  - Memory leak check must pass.
  - Timing tests within limits.
- **Blame-Free Culture for Failures:** A failing pipeline is not a personal failure; it's the process working. Promote a culture where fixing a red build is the team's collective priority, and no one is "at fault" – the goal is product quality. This is important to keep morale high and avoid hiding problems. Managers should reinforce this by praising the quick recovery from red to green rather than blaming why it went red.

- **Pipeline as Code & Versioned:** Keep the CI configuration in the repository (most modern CI does this by design, like ".gitlab-ci.yml" in the repo) and subject it to code review. This means any team member can propose improvements to CI via the same PR process. It also means changes to CI (like adding a new job for a new test tool) are tracked.
- **Archive Results and Artifacts:** For critical builds (like a build that goes to a formal test or the spacecraft), archive all logs and artifacts. Store them in an artifact repository (like Nexus or as attachments in a ticket). This allows post-mortem and root cause analyses if something goes wrong in the mission – you can retrieve the exact binary and test logs that went with it. ECSS standards emphasize the configuration of delivered software. In our STIX flight software project, we were able to decompile an old binary to evaluate what potential root causes an event may have had.
- **Security in CI:** Use the least privilege for CI runners. If using cloud CI, ensure credentials are secure. To avoid code exposure, you can use self-hosted runners on an internal network for mission code. Also, ensure the CI doesn't accidentally deploy secrets (you can scan logs to ensure the scripts don't print secret values).

Adhering to these practices makes a CI/CD pipeline a powerful tool for automating building and testing and enforcing quality and reliability. It's like having an additional team member (an automated one) who is tireless and pedantic, checking every detail every time.

In summary, CI/CD for embedded space software enables a project to be both fast and right – achieving the reliability expected in space missions without sacrificing the agility and speed of modern development cycles. It allows you to sustainably meet rigorous standards (ECSS, NASA) by baking compliance into everyday workflow rather than treating it as separate paperwork.

6. VIRTUALIZATION AND PLATFORM SIMULATION

Testing space software on actual hardware can be challenging due to limited hardware availability, high costs, and difficulty recreating ground space conditions. Virtualization and platform simulation provide powerful solutions by creating virtual models of the hardware and environment in which the software can run. This chapter details best practices for using virtualization in embedded space software development, complementing the CI/CD practices from Chapter 5 and fulfilling verification needs early in development (as encouraged by standards like ECSS-E-ST-40C and ECSS-Q-ST-80C).

6.1. Purpose and Benefits

Virtual platforms (or virtual testbeds) are software representations of the target hardware (and sometimes even the environment or other systems) on which the flight software runs. By using virtualization, teams can:

- **Start software development early:** You can begin developing and testing software before the actual hardware exists. For example, if a new test board is only available in six months, using an emulator allows the software to progress.
- **Run tests at scale:** With virtual platforms, you can run many instances in parallel (e.g., dozens of virtual satellites) or overnight tests, something impossible with a single physical board. This vastly improves testing throughput and regression testing. It also allows many developers to code and test in parallel without waiting on a hardware board they may need to share.
- **Inject faults and edge cases safely:** You can simulate sensor failures, extreme data rates, or invalid inputs in a virtual environment without risking hardware damage. This is great for testing fault handling (like how the software reacts to a failed component) or stress tests. Maybe you want to know how your system deals with suddenly dying flash cells or a burst of single event upsets in your memory.
- **Integrate into CI pipelines:** As seen in Chapter 5, virtualization allows automated tests on each commit. You can include system-level tests in CI by running the software on a virtual platform within the pipeline.

In short, virtualization is a force multiplier for testing and development, improving feedback speed and software quality. Recognizing this, space standards and best practices endorse simulation:

- ECSS-Q-ST-80C explicitly lists “model simulation” as a verification technique to be used where possible (to verify requirements cost-effectively).
- NASA missions frequently develop high-fidelity simulators (digital twins) for verification and mission rehearsal. For instance, a NASA JPL project used a testbed called “Software Testbench," essentially a virtual spacecraft for continuous testing.

6.2. Levels of Simulation and Fidelity Trade-offs

There are different levels of fidelity in simulation, each with trade-offs between accuracy and speed/effort:

- **Instruction-Level Emulation:** Tools like QEMU (open-source), T-EMU or TSIM emulate CPUs and allow integration of virtualized hardware components, like memory or other devices. This means you can run the actual compiled binary as-is.

  Pros: reasonably high fidelity for CPU and basic peripherals, catching low-level issues (e.g., alignment, endianness, OS context switches).

  Cons: needs a model of the board and is slower than functional simulation.

  We used QEMU, e.g., to emulate an ARM Cortex-M board, running our RTOS and application, which lets us test the whole software stack in CI. TSIM is used for LEON (SPARC) processors typical in ESA instruments, like STIX (see section 8.1), to do similar things.

- **Functional Simulation (Stubs/Mocks):** Instead of emulating hardware, simulate specific components. For instance, write a function that generates synthetic sensor data for testing algorithms instead of an actual sensor driver. Or simulate a bus with a task that sends messages to the software. This doesn't run the actual binary on a simulated CPU, but instead runs a specialized configuration of the function under test with mock interfaces on a developer machine.

  Pros: more straightforward to implement, very fast (no need to simulate every instruction).

  Cons: might miss issues that would occur during real integration (timing, concurrency).

  We use this for testing logic in isolation, e.g., testing the control algorithm by feeding it recorded sensor inputs via a harness program without involving the actual RTOS or hardware drivers.

- **Full System Digital Twins:** These are high-fidelity simulations of entire spacecraft or systems, often including physics. For instance, a simulator that not only runs the flight software but also simulates the spacecraft attitude dynamics, orbit propagation, and environment (sun sensors, magnetometers, etc., giving realistic signals).

  Pros: can accurately validate end-to-end mission scenarios (almost like a dress rehearsal).

  Cons: highly complex and heavy – often separate big projects to develop these, and they might run slower than real-time or require HPC resources.

  You might typically see these at big space agencies for mission validation. You might not build a full twin for smaller projects, but you could simulate a few key environmental aspects (like sensor noise models).

Choosing the right level: Use the simplest model for the test objective. For example, suppose you are testing a data handling algorithm. In that case, you don't need a full CPU emulator, but you could

call the algorithm in a simple program with test inputs (functional simulation). If you want to test the integration of our software with an RTOS, then an instruction-level emulator is appropriate. Reserve full digital twin usage for final system validation or specific analyses.

Also, mix levels: early in development, functional sims for speed; later, run some tests on the QEMU emulator for realism; and for final validation or debugging weird issues, maybe run on a high-fidelity simulator or actual hardware.

Example trade-off: To test compliance of a software telemetry implementation, you could initially simulate an instrument sending packets to the communication handler by feeding crafted packet bytes (functional sim). This quickly verifies logic for framing, etc. Later, you run the same test on QEMU with the software reading from a simulated UART to ensure the byte-by-byte handling and interrupts work (instruction-level). You likely don't need a complete spacecraft simulator for that particular feature.

6.3. Integration in Development Workflows

We integrate virtualization deeply into our workflows:

- **During Development:** Developers often run the flight software on their PC using QEMU or even a high-level simulator to do quick checks. Make targets or scripts like "run_in_qemu.sh" to make this easy. This avoids needing a development board on every engineer's desk; they can test most things virtually (and we reserve boards for when hardware-specific issues need attention).
- **CI Pipelines:** As covered, virtualization is used in CI for automated testing on each commit. Set up at least one job for "Software-In-the-Loop" (SIL) testing. Over time, add more scenarios to this as you develop them.
- **Continuous System Integration:** Maintain a virtual "flat satellite" where your software and possibly other components' software (like payload or ground segment simulators) run together. For instance, using containers or VMs, simulate a ground station sending telecommands to the virtual satellite software and getting telemetry back, maybe even processing it with a ground software component. This helps test interfaces between teams without requiring both to have hardware in the loop.
- **Operations Rehearsal & Debugging:** Later in the project, operations teams might use the software simulator to practice procedures or to verify telecommand sequences. If an anomaly happens on an actual spacecraft, engineers can try replicating it in the virtual platform to investigate (assuming the simulation fidelity is high enough for that scenario).
- **Reuse across projects:** Treat models and simulation setups as assets. For example, if you have modeled a generic AOCS (Attitude and Orbit Control System) sensor in one project, reuse it with minimal tweaks in another. This library of simulation components grows and makes future virtualization easier.

Essentially, the virtual platform becomes a core part of the "Software Verification Facility" – a concept some agencies formalize (e.g., having a reference facility where software is tested extensively in simulation before going to the actual hardware).

6.4. Limitations and Validation Strategy

While virtualization is powerful, it's not a panacea. Remain aware of its limits:

- **Timing differences:** An emulator might not reproduce exact real-time timings (e.g., instruction execution times, cache behaviour, interrupt latencies). Race conditions or performance issues might not appear until you run on real hardware. You may mitigate this by doing hardware tests and artificially constraining or randomizing timing in simulation (for example, adding random delays in simulated interrupts to try different timings).
- **Peripheral fidelity:** Simulating complex peripherals (like RF radios or high-speed DMA controllers) is hard. Our tests might bypass those, by stubbing them, or use simplified models. Thus, we must still test the real integration with those devices when the hardware is available.
- **Partial coverage of physical phenomena:** Unless explicitly modeled, a digital twin might simulate some physics (like orbital motion) but not others (like radiation effects or thermal noise). You cannot rely on simulation alone for things like "Does the sensor work in a vacuum at -20° C?" that need physical testing.
- **Simulator bugs:** Simulators themselves can have bugs or incorrect models. Thus, when something odd happens, also question, "Could it be the sim?" and validate the simulator by cross-checking with hardware results for a set of test cases.

Our strategy:

- Use virtualization for **what it's best at** (functional logic, early integration, non-intrusive testing).
- Gradually introduce **more accurate hardware tests** for things where virtualization might be weak (timing, device drivers).
- **Validate the simulator itself**. Once the hardware is available, run known tests on both and compare outputs. If they differ, improve the sim or be aware of those differences.
- **Document assumptions of simulations**. For example, if our power system is not simulated for a test, we note that no one assumes we validated a power dropout scenario when we didn't.

We often layer tests:

1. Virtual tests run on each commit (fast, broad coverage).
2. Daily or weekly hardware tests run key cases (slower, limited by hardware).

3. Formal qualification tests on hardware in the final phase (slow, thorough).

The combination catches most issues.

6.5. Summary of Virtualization Best Practices

To conclude the virtualization chapter, here are best practices to consider:

- **Leverage Open-Source Tools:** QEMU is our go-to for CPU/board emulation (it supports many architectures). Contribute device models if needed for your boards or use ones provided by vendors. For higher-level simulations, use frameworks like TASTE (which allows co-simulation of components) or even Unity/Python to visualize scenarios.
- **Integrate with CI:** As stated, ensure the simulation is part of automated testing. On each commit, run at least one scenario virtually. This also provides the virtual platform is always up-to-date (if code changes, our simulation adapters might need updating, and CI catches that).
- **Maintain a library of models:** Build a collection of simulated devices and environments (sensor models, actuators, etc.). For example, an extensive library of PUS-C models of telemetry packets in ASN.1 [RD12] feeds into a tool that generates code and a simulation to produce packets (as in our turn-key protocol case study). Reuse these to avoid redoing work.
- **Know when to stop simulating:** Identify tests that must be on real hardware, and don't spend excessive time trying to emulate those that are easier to test on real hardware. For example, don't simulate the exact thermal noise on an ADC if you can test the code with recorded data from a lab thermal test.
- **Use Digital Twins for operations:** If resources allow, involve the operations team in building a high-fidelity simulator (digital twin) that can run parallel with the mission operations. This can be invaluable for training and anomaly resolution (e.g., the Orion capsule's flight software had a digital twin for the ground to practice on [RD10]).
- **Document simulation configuration:** Just like code, treat the configuration of simulations (what version of QEMU, what models loaded, etc.) as part of your baseline. If someone reruns a simulation a year later, results should be comparable (given the same seed inputs, etc.).
- **Simulate failures and stresses:** Proactively use the simulator to push the software. Inject out-of-spec inputs, simulate sensor freezes or random resets, etc., to see how the software handles them. This often reveals resilience issues. You can simulate some things you can't do easily on real hardware, like simulating bit-flips.

7. Model-Based System Engineering (MBSE) for Embedded Space Software

Modern space missions are increasingly complex, with tight integration between hardware and software and rigorous interface definitions. Model-based System Engineering (MBSE) is a methodology for managing this complexity using formalized models rather than traditional document-centric approaches. Regarding software quality, MBSE offers benefits in requirements clarity, design consistency, and automated generation of artifacts (like code or test cases). This chapter provides an overview of MBSE concepts and how we apply them in our software projects, ensuring strong links between system models, software implementation, and verification.

7.1. What is MBSE?

MBSE refers to using digital models to support system specification, design, analysis, verification, and validation. Instead of writing extensive textual documents that describe the system, MBSE encourages creating interconnected graphical and/or textual models that capture all aspects: requirements, behavior, architecture, data, interfaces, etc.

Key ideas in MBSE:

- **Central System Model:** There is a single source-of-truth model (or a set of integrated models) that all stakeholders contribute to and use. This model can be visual (diagrams) and semantic (machine-readable).
- **Multi-View Consistency:** Different views (e.g., a functional flow, a physical component diagram, a state machine) are all part of the same model, ensuring consistency with each other through construction or automated checks.
- **Executability:** Some MBSE tools allow you to execute or simulate the model (for example, run a state machine or generate a sequence diagram from a scenario) to validate behavior early.
- **Traceability built-in:** Requirements can be linked to model elements, and model elements can be linked to each other and test cases. The model repository effectively maintains the traceability you would otherwise do in documents or spreadsheets.

MBSE uses notations and languages such as SysML (Systems Modeling Language), UML, AADL (Architecture Analysis & Design Language), and SDL (Specification and Description Language), which are often supported by tools like Capella [RD3], MagicDraw, Enterprise Architect, etc.

7.2. Relevance to Space Systems

For embedded space software and satellite missions, MBSE provides several concrete advantages:

- **Clear definition of interfaces:** Space systems involve many interfaces (between onboard subsystems and between spacecraft and ground). Using MBSE, you create interface models that precisely define data exchanged, timing, protocols, etc., in a unified way. This reduces team miscommunication (e.g., the software expects a sensor to send data in one format, but the hardware sends another – a model would catch that discrepancy).
- **Traceability from mission objectives to code:** The model allows for tracking high-level mission objectives (like "The satellite shall provide X data. ") down to system functions, software requirements, and code or simulation elements. This is aligned with ECSS and NASA expectations for end-to-end traceability, and MBSE tools often provide built-in trace matrices or queries to obtain this information.
- **Early design validation:** The team can validate system behavior before any code is written or hardware is built using executable models or simple simulations (like state machines or functional chains). For instance, check that the power subsystem model can support the duty cycles of the payload as modeled.
- **Improved multi-discipline communication:** Space projects have system engineers, software engineers, mechanical engineers, thermal engineers, etc. A well-done system model (especially using SysML or Capella with the Arcadia method) creates a common language for these disciplines. For example, the system engineer can see how software components are distributed and communicated, and software engineers can see how their part fits into the bigger picture (and what the hardware expects).

Space agencies have recognized these benefits:

- ESA promotes MBSE and provides tools like TASTE [RD2] (which integrates AADL, SDL, and ASN.1 modeling for embedded systems) to streamline development.
- NASA has used MBSE on projects like Orion [RD10] to integrate flight software design with the overall vehicle models.

We incorporate MBSE proportionally to project needs, mainly to generate end-to-end packet services (see Section 8.5).

7.2.1. Practical Application in Software Projects

In practice, MBSE in our software projects often involves a combination of system-level modeling, interface definition, and code generation:

- **High-Level System Modeling with Capella (Arcadia):** You can use Capella [RD3], an open-source MBSE tool for system and architecture modeling. Following the Arcadia methodology, you define:
    - Operational scenarios (how the system is used in various modes).
    - Functional breakdown (functions the system must perform and how they interact).

- Physical architecture (subsystems, equipment, software components, and connections).
- Interfaces between components (data flows).

Capella provides a graphical interface to do this. We can use Capella for projects where we must design interactions between multiple software and hardware components. The benefit is that once the model is built:

- It serves as a blueprint that both software and system engineers can refer to.
- It ensures we don't forget any components or interactions (the tool can check completeness).
- Changes in architecture (say we split a component into two) are easily updated and ripple through diagrams.

Capella also allows requirements to be documented and linked to model elements (e.g., linking a requirement to the function that satisfies it). We used Capella in a system design process to model use cases and functional changes across multiple partner contributions.

- **Real-Time and Embedded Modeling with TASTE:** You can use ESA's TASTE framework for real-time embedded systems, especially ones that will eventually run on an OS or use middleware. TASTE lets you model tasks (called Functions in SDL), interfaces between them (using ASN.1 for data modeling and automatically generated communication code), and deployment (which function runs on which CPU, on which bus, etc.). Key aspects:
  - You can draw state machines in SDL for component behavior, define interface data structures in ASN.1, and then TASTE will generate code (in Ada, C, or others) for the communication glue (like message passing or bus comms).
  - It integrates with simulation: you can run the system on a "virtual platform" where each component might be a process on a host machine, communicating via a simulated bus, which is excellent for testing.
  - We see TASTE as particularly useful in ensuring all interface assumptions are explicit and that we could swap implementation languages for components (e.g., one component in C, another in Ada, and TASTE handles their interaction).

  By using TASTE, you can effectively apply MBSE principles at the software architecture level – the model is the code to some extent or at least generates a lot of the repetitive code.
- **Interface Definition with ASN.1 and Auto-generation:** A very pragmatic MBSE practice we follow is using ASN.1 to formally define data structures for communications (telemetry/telecommand or internal messages). ASN.1 is a modeling language for data that can be automatically compiled into code for different languages using tools like ASN1SCC [RD5] (ESA's ASN.1 compiler). Here's how we use it:

- The communication interfaces (e.g., all the telemetry packets and telecommands or network messages between subsystems) are defined once in ASN.1 files.
- ASN1SCC is run to generate C code for those definitions (structures, encoding/decoding functions) and maybe also for other languages (we've generated Ada and soon Python from the same ASN.1).
- This ensures that every part of the system (flight SW, ground SW, test tools) uses the same data format, eliminating mismatches.
- The ASN.1 is a formal spec – we consider that part of our MBSE artifacts, as it is machine-readable and documents interfaces.

The combination of ASN.1 modeling and auto-code generation is a concrete example of MBSE providing consistency and saving time (less manual coding, fewer bugs). We will see this in the case study 8.5 where the protocol generation is described.

- **Simulation and Validation of Models:** After building models (Capella, TASTE, etc.), you can perform simulations on them:
  - In Capella, simulating a scenario (step through a functional chain to see data flow) is possible.
  - With state charts, some tools allow the execution or generation of test cases from state models.
  - With TASTE, you can compile and run an entire system model on a host for validation.

This doesn't replace testing the final code, but it can catch integration logic issues or early interactions. It's also a way to validate requirements: e.g., if a requirement says, "Component A shall send data to B at 10 Hz", you ensure in the model that there is a functional path for that and simulate timing.

7.2.2. Challenges and Considerations

MBSE isn't a silver bullet and introduces its challenges:

- **Tool maturity and interoperability:** Different teams might prefer different tools, but not all integrate. We have seen models exported from one tool to import into another (using exchange formats like XMI or OSLC), which sometimes works but sometimes loses information. Custom code generators or scripts may be needed to bridge gaps.
- **Learning curve:** Engineers need training to use MBSE tools effectively. Drawing good models is a skill – otherwise, you risk "diagramming" without much value. Start with modest MBSE and grow as the team gains skill – remember Section 4.1 advice and use MBSE pragmatically and where it makes sense.
- **Keeping models in sync with reality:** Updating the model when things change requires discipline. If the model becomes outdated, people stop trusting it. You may mitigate this by

tying work products to the model (e.g., you cannot change code for an interface unless the ASN.1 is updated first). But it's an ongoing effort to avoid divergence.

- **Deciding level of detail:** One can model down to a very low level (every thread, every variable), but that may not be worth it. Choose what to model: usually architecture and interfaces, sometimes key algorithms abstractly, but not every line of code. The aim is to model enough to get the benefits (clarity, code gen) but not try to model everything (which could be as time-consuming as coding itself).

The key to successful MBSE adoption is pragmatism:

- Use it where it adds clear value (complex interfaces, cross-team understanding, safety-critical logic).
- Don't force it where a simple document or script would do.
- Ensure management supports the initial overhead because MBSE might initially seem slower (drawing diagrams vs hacking code) but pays off later in integration and maintenance.

8. CASE STUDIES

8.1. STIX Flight Software – A Validation of Our Guiding Principles

The Spectrometer/Telescope for Imaging X-rays (STIX) is one of ten instruments on ESA's Solar Orbiter spacecraft, launched in 2020. STIX's flight software had to manage detector data processing on an Instrument Data Processing Unit (IDPU), which included an FPGA and a LEON3 processor running RTEMS (a real-time operating system).

8.1.1. Challenges

The initial software development contractor faced multiple challenges:

- The testing hardware was unavailable during the early flight software development, limiting testing opportunities.
- Early development was done without a strong CI or virtualization setup. Much testing was done manually or at a very high-level interaction (integration test level instead of unit tests). No reproducible build or well-automated testing setup was created.
- A lot of effort went into the boot loader ("Startup Software," SuSW), which did not leave much time or budget for the actual Application Software (ASW).
- Many of the onboard algorithms for data analysis kept changing and were difficult to "pin down."

Due to these factors, the project encountered issues late in the development (during instrument testing and even during the spacecraft integration):

- The PI institution had to take over software development late in the project because the supplier stopped developing it before completing the work. The SuSW development was completed, but the state of the Application Software was very preliminary.
- Because of the lack of automation (virtualization and with the test hardware), a lot of effort has to be invested in creating a pragmatic testing environment. Because of time constraints, no full automation was achieved (or even attempted), which impeded testing and lowered
- A lot of effort had to be spent creating visibility of the project state and prioritizing development. Some issues were only detected during the flight. Luckily, no serious problems were found.
- Fixing these issues under time pressure was challenging and risky.

8.1.2. Approach

Five core developers rewrote 90% of the STIX Application Software before launch. They also supported stabilizing and improving the STIX software post-delivery:

- We cleaned up the project and focused on creating reproducible builds with clear versions and a history of older binaries.
- We improved the development environment, for example by enabling real-time debugging on hardware with gdb.
- We started extending EGSE scripts to allow reproducible and automated testing, combining telemetry parsing and telecommand sending, controlling test hardware (like the detector hardware emulator), running sequences to stimulate the test hardware, analyzing results in the science analysis environment, etc.
- We applied the engineering principles laid out in Chapter 4. For example:
    - We added asserts to the code to ensure we find coding and programmatic issues during testing (dynamic verification).
    - We ran static checkers on the code (static verification).
    - We extracted the file system into an external project and mocked the FLASH system to run extensive file system operations.
    - We simplified the functional chains and introduced simpler but robust processes, e.g., limiting telemetry and telecommand buffers, which would throw away extra TM or TC in case of flooding but would otherwise not fail.
    - We introduced a clear development and issue-tracking process using GitHub.
- Later, in the frame of a research project, we formally verified parts of the STIX file system [RD13] and found and fixed several minor bugs.
- Later, we built a prototypical STIX emulator based on QEMU.

8.1.3. Lessons Learned

The STIX experience underscored:

- If CI/CD and virtualization had been used, many issues would likely have been caught earlier (e.g., an automated integration test could have found a particular telemetry encoding error rather than during spacecraft commissioning).
- Early investment in test infrastructure pays off massively later. The effort to create the virtual platform and tests after the instrument was built was much higher than it would have been to make it incrementally during development.

- **Traceability is key:** When issues arose, the first question asked was, "What requirement was this code supposed to fulfill?" If that wasn't clear, it caused delays. Aligning code to requirements (with IDs in code comments and tests referring to them) would have clarified intent.
- This case reinforced our conviction to always push for CI, even if the project timeline is tight. The argument "no time for CI/tests now, we'll test later" did not hold – it ended up costing more time later. We explicitly allocate time for test infrastructure in proposals or project plans and explain that it is as important as coding.

8.2. NASA cFS – Reliability through Modular Architecture

Embedded flight software for space missions must be robust and reliable, operating autonomously under harsh conditions without opportunities for physical intervention after deployment. A single software malfunction can jeopardize an entire mission. For this project, we adopted NASA's Core Flight System (cFS) [RD7] as the foundation for developing a new platform to test quantum-safe encryption modules intended for space use.

NB: While cFS's decoupled message-based architecture offers clear advantages described here, monolithic payload and platform software systems (such as the one developed for STIX, described in Section 8.1) may sometimes be more suitable depending on mission requirements.

8.2.1. Challenges

Key challenges included:

- **Ensuring reliability under resource constraints**: The flight software needed to run reliably on limited hardware resources without compromising performance.
- **Managing complexity and compliance**: The software had complex functionality and had to adhere strictly to aerospace standards and specified interfaces.
- **Adapting to changing requirements**: Project requirements and hardware interfaces evolved during development, requiring the architecture to accommodate updates quickly and safely.
- **Early defect detection**: Traditional late-stage testing would be inadequate. Early identification and resolution of software defects were critical to preventing costly problems later in development.

  To address these challenges, we required an architecture supporting modularity, portability, rigorous testing, and continuous integration and validation across multiple platforms.

8.2.2. Approach

We selected NASA's open-source Core Flight System (cFS) as our base architecture for flight software development. The cFS framework offers a modular architecture, dividing software into independent applications that communicate via a software bus. This modularity simplifies development, debugging, and testing. Additionally, cFS includes an Operating System Abstraction Layer (OSAL), enabling the same codebase to run unmodified on multiple platforms, such as a Linux-based simulator for development and RTEMS on the actual spacecraft processor.

Figure 11 illustrates this software architecture, highlighting application communication over the cFS message bus.

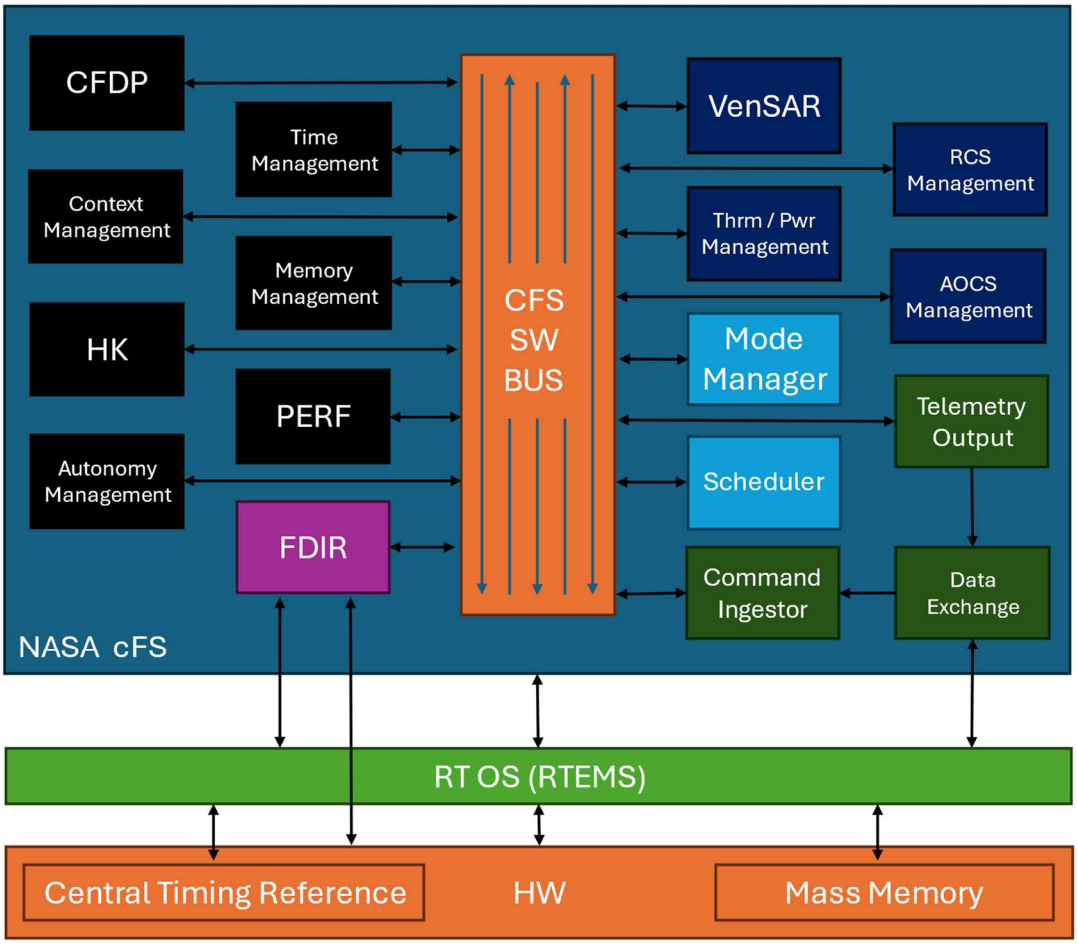

Figure 11: Illustration of the overall flight software system, tailored to a specific mission.

Key aspects of our approach include:

- **Modular Software Architecture:** Clearly defined, independent cFS applications simplify parallel development and reduce complexity.

- **Operating System Abstraction Layer (OSAL):** OSAL allows consistent execution of flight software on development machines, simulators, and spacecraft hardware without modifications, significantly increasing test coverage.
- **Continuous Integration/Continuous Delivery (CI/CD):** With CI/CD, we can automate pipelines to build, test, and analyze code changes continuously. Each commit triggers a suite of unit and integration tests executed on Linux servers, catching defects early.
- **Software Validation Facility (SVF):** The SVF, as shown in Figure 12, performs repeatable automated testing, including nominal and off-nominal scenarios. The SVF conducts closed-loop tests against simulated devices by employing software models of hardware components. Tests are executed across multiple environments (PC-based simulation or hardware-in-the-loop setups), and results are logged and analyzed automatically. Usually, an SVF is a "shippable" framework that can be used by the customer, e.g., ESA, to run their own tests.

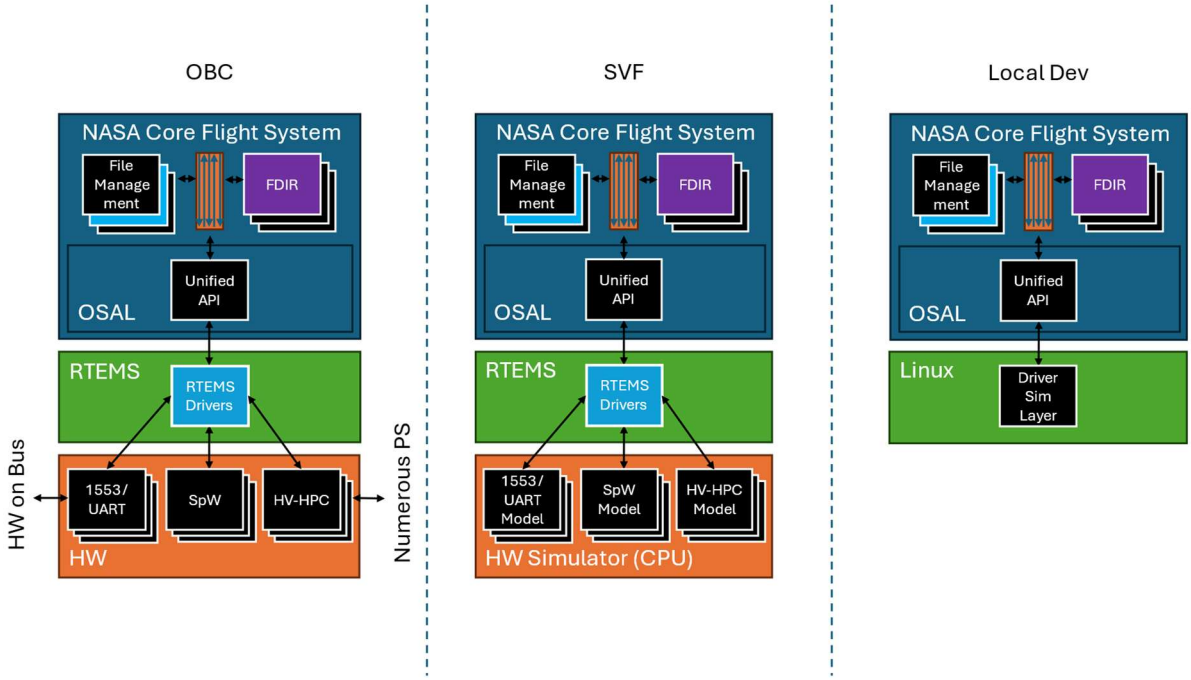

Figure 12: Visualization of the Flight Software with the three runtime layers.

- **Static Analysis and Formal Methods:** To complement testing, we integrate static analysis tools (e.g., PVS-Studio) into development workflows to automatically flag potential issues. Formal verification can be selectively applied to prove the correctness of critical algorithms when feasible.
- **Model-Based Development:** We use ASN.1 and ASN1SCC to auto-generate consistent and correct code from high-level definitions to generate protocols.

Figure 13 illustrates the testing infrastructure, showing how requirements and interface definitions feed into SVF test suites. Tests run in different environments (development PCs, high-fidelity simulators, onboard computers), with faster tests executed on each commit and more intensive tests scheduled periodically.

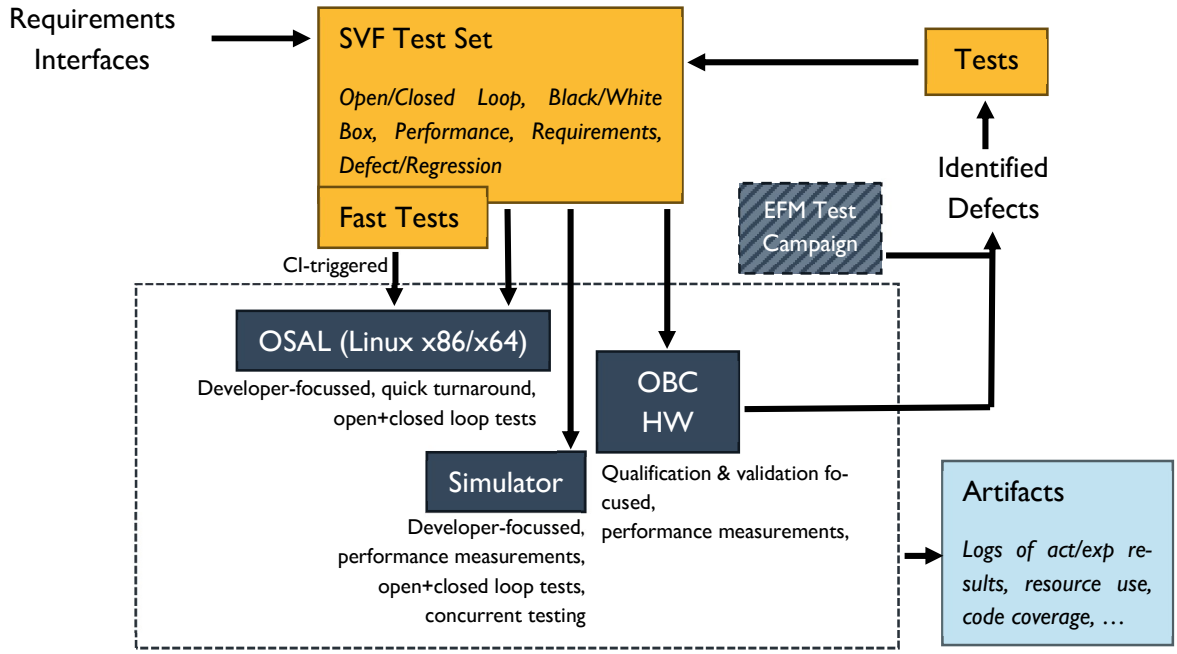

Figure 13: System diagram for the Software Verification Facility and testing approaches.

8.2.3. Benefits

Adopting cFS and these software quality practices provides several concrete benefits:

- **Improved Reliability:** The modular structure allows thorough testing of individual components, making it easier to isolate and correct issues early. Extensive testing across simulation, hardware-in-the-loop setups, and SVF provides high confidence in integration.
- **Increased Development Efficiency:** OSAL allows developers to code and debug on standard Linux machines without waiting for flight hardware. Parallel development and frequent integration significantly reduce development cycles.
- **Rigorous Testing Capabilities:** Automated testing through the SVF ensures regressions and integration issues are identified quickly, reducing late-stage debugging time and costs.
- **Flexibility to Adapt:** The abstraction provided by cFS and OSAL enables straightforward adaptation to changing requirements or hardware interfaces, minimizing costly redesigns.
- **Continuous Quality Assurance:** Combining static analysis, model-based code generation, and formal checks allows the early detection and resolution of defects, ensuring software maturity earlier in the project lifecycle.

- **Reduced Risk and Cost:** Early and automated testing, proven frameworks, and rigorous analysis significantly reduce overall project risk. Potential mission-impacting software defects are identified and resolved well before deployment, avoiding costly late-stage fixes.

8.2.4. Lessons Learned

Our experience with the chosen approach reinforces several vital lessons:

- **Invest Early in Infrastructure:** Establishing automated CI/CD and testing infrastructure early in a project pays off significantly by catching defects early and improving confidence in software quality. Delaying this investment often leads to increased risk and cost later.
- **Leverage Proven Frameworks:** Adopting a validated framework such as cFS reduced development risk and effort compared to creating a custom architecture from scratch. Proven frameworks provide a robust starting point and enable quicker development.
- **Modularity Facilitates Testing:** Designing software as modular components simplified testing and debugging, enabling developers to isolate and correct problems efficiently. While modularity often involves message-based decoupling (as in cFS), a well-designed monolithic architecture can also achieve similar modularity.
- **Continuous Integration Catches Issues Early:** Frequent integration and automated testing significantly improved software stability. Immediate feedback prevented integration problems from accumulating into more significant issues later.
- **Holistic Quality Approach:** High-assurance software requires multiple complementary quality practices. Combining static analysis, formal verification of critical modules, model-based code generation, and comprehensive testing provided a robust quality assurance strategy. Relying on a single technique is insufficient; effective software quality results from employing multiple reinforcing methods.

8.3. EGSE Scripting Language – Inherently Safe Execution

Although specific project details cannot be disclosed, this brief case study highlights valuable practices to ensure software reliability and correctness within an embedded testing environment. The project's primary objective was to develop a robust, inherently safe scripting language for Electrical Ground Support Equipment (EGSE). The language is needed to generate accurate telemetry and telecommands to interact and test simulated or actual hardware instruments. Unexpected errors in execution could potentially leave sensitive instruments in a harmful or undefined state, making predictability, reliability, and correctness paramount.

8.3.1. Challenges

The EGSE project confronted a few unique challenges:

- **Inherently Safe and Correct Scripting:** We had to ensure that any script written by a user could not accidentally harm the connected hardware. Errors in the script (incorrect loop definition, missing language structure elements like curly brackets, etc.) must be detected before beginning script execution.
- **Integration with Existing Tools:** The new EGSE scripting environment had to integrate with existing testing workflows, including the stack of real telemetry/telecommand messages. Ensuring compatibility with hardware interfaces and legacy test procedures was a non-trivial task.
- **Flaky Deployment Pipeline:** The CI/CD pipeline was outdated and had to be adjusted to run our new scripting environment. However, interactions with the IT department were slow and not straightforward.

8.3.2. Approach

We developed the scripting language using ANTLR to achieve inherent safety and reliability. This approach explicitly mapped every language operator, such as mathematical operations, loops, and conditional statements, to clearly defined, safe execution paths. Rather than individually verifying each script, the scripting language architecture itself ensured correctness and safety by design.

Rigorous unit tests were created directly in the scripting language. These test scripts were translated into telecommands and executed against a separate loopback test class implemented directly in C++. Each operation was independently evaluated using these "raw" C++ commands. Comparing results from the scripting language implementation and the independent C++ implementation provided high confidence in correctness, significantly reducing the risk of identical interpretation errors appearing in both implementations simultaneously.

To allow testing of the telemetry/telecommand system and later simplify integration with the real TM/TC system of the encompassing EGSE framework, we designed an external interface class, like an adapter class, that can abstract any external command and make it available to the scripting language. Ideally, the existing TM/TC messages can be auto-generated and mapped to the external interface to reduce implementation efforts and user coding errors.

The team improved the existing Continuous Integration and Continuous Delivery (CI/CD) pipeline to use Docker images and GitLab Registries to ensure repeatable builds. The entire build, test, and deployment process was containerized using Docker, with container images hosted securely within GitLab Registries. Secure secret management was introduced to allow Docker containers to access GitLab infrastructure securely. Additionally, configuration parameters were externalized and securely injected into containers from dedicated, well-managed configuration stores. This approach ensured a consistent, secure, and repeatable integration and deployment process.

8.3.3. Benefits

Key benefits realized from these practices included:

- **Inherent Reliability:** The scripting language architecture and rigorous testing approach significantly reduced the likelihood of runtime errors or unsafe instrument states.
- **Consistent and Repeatable Testing:** Containerizing the build and test processes ensured repeatable and consistent results across development and deployment environments.
- **Enhanced Security:** Secure secret management and externalized configuration parameters improved overall security and reduced risks associated with sensitive data handling.
- **Increased Confidence:** Independent verification of command execution through separate implementations significantly increased confidence in correctness and reliability.
- **Clean and Modular Code:** We replaced the existing scripting approach with our new strategy, simplifying and decoupling the entire setup (EGSE framework vs. scripting) to be more flexible and robust.

8.3.4. Lessons Learned

This case study reinforced the following lessons:

- **Safety by Design:** Architecting scripting languages with inherent correctness significantly reduces runtime risks.
- **Independent Verification:** Comparing independent implementations of the same functionality provides high assurance of correctness.
- **Effective CI/CD Practices:** Containerization and secure management of build and deployment processes enhance software reliability and security.

8.4. CI/CD Automation of Onboard Image Processing Pipeline

Modern earth observation satellites are moving towards using onboard machine learning (ML) algorithms to reduce the amount of data transmitted to ground stations. In this first of three studies, our team designed and developed an onboard image processing pipeline suitable for embedded satellite payload hardware. The goal was to preprocess optical imagery onboard in real-time using a TensorFlow-based processing graph, including demosaicing, image corrections, and projections. We built and integrated this TensorFlow pipeline directly into our embedded OS and control software, verifying functionality and performance automatically through a Continuous Integration/Continuous Delivery (CI/CD) pipeline.

Subsequent studies built upon this initial effort: Study 2 added object detection models to this platform. Study 3 culminated in an integrated real-time demonstrator that successfully performed onboard object detection using drone-mounted cameras.

Figure 14 illustrates the TensorFlow-based preprocessing pipeline we implemented.

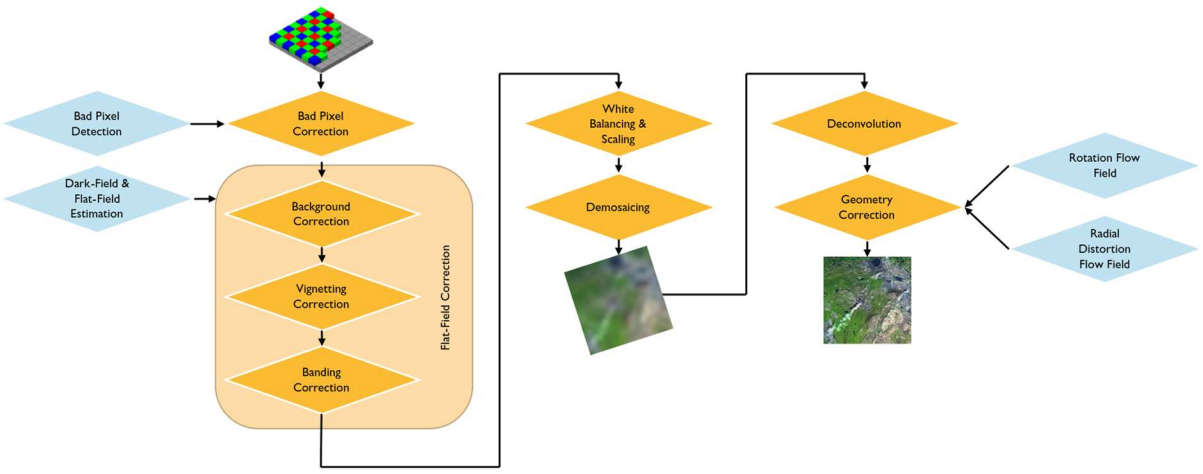

Figure 14: Schematic of the final pre-processing pipeline, as previously shown.

8.4.1. Challenges

Integrating TensorFlow-based image processing into embedded payload software posed several challenges:

- **Hardware Constraints:** Target hardware (embedded Linux boards running Buildroot and microcontrollers running Zephyr RTOS, initially evaluated but ultimately not selected) had limited CPU power and memory. The TensorFlow models developed on desktop computers needed significant optimization to fit these constraints.
- **Complex Integration Workflow:** Models were developed in TensorFlow/Python, but flight software was written primarily in C/C++. Manual integration of these distinct workflows risked human error and inconsistency.
- **Constant Performance Evaluation:** One of the study's objectives was to evaluate the performance of the chosen methods. Consistently evaluating all the different and changing processing steps would be tedious, so they had to be reliably automated.
- **Slow Feedback Loop:** Without automation, building, deploying, and testing model changes on embedded hardware was slow and tedious. Integration issues were often discovered late, causing rework.
- **Testing and Reliability:** Mission-critical software required extensive testing to ensure correctness and avoid regressions after each change.
- **Toolchain Complexity:** Multiple tools and frameworks (TensorFlow Lite, Zephyr, Buildroot, QEMU virtualization, and GitHub Actions) had to be integrated seamlessly.

8.4.2. Approach

From the beginning of the project, we wanted to integrate all the different study pieces into a CI/CD pipeline. Thus, we implemented an automated CI/CD pipeline combining TensorFlow model

integration, embedded software builds, virtualized hardware testing, and continuous performance evaluation to overcome these challenges. The entire system, from model to embedded OS to flight software, was developed and integrated within a single version-controlled repository.

Key elements of our approach:

- **TensorFlow as an Execution Graph:** We modeled the image processing pipeline using TensorFlow as an execution graph, which allows rapid iteration, a clear structure, and built-in optimization tools.
- **Automated Model Integration:** Each TensorFlow model update triggered automated conversion to TensorFlow Lite (TFLite), verifying compatibility with embedded runtimes and resource constraints.
- **Integrated OS and Software Builds:** The CI pipeline built the embedded Linux OS (Buildroot, see Figure 15), along with our flight control software, automatically integrates the latest TFLite model, ensuring reproducibility and consistency across builds.

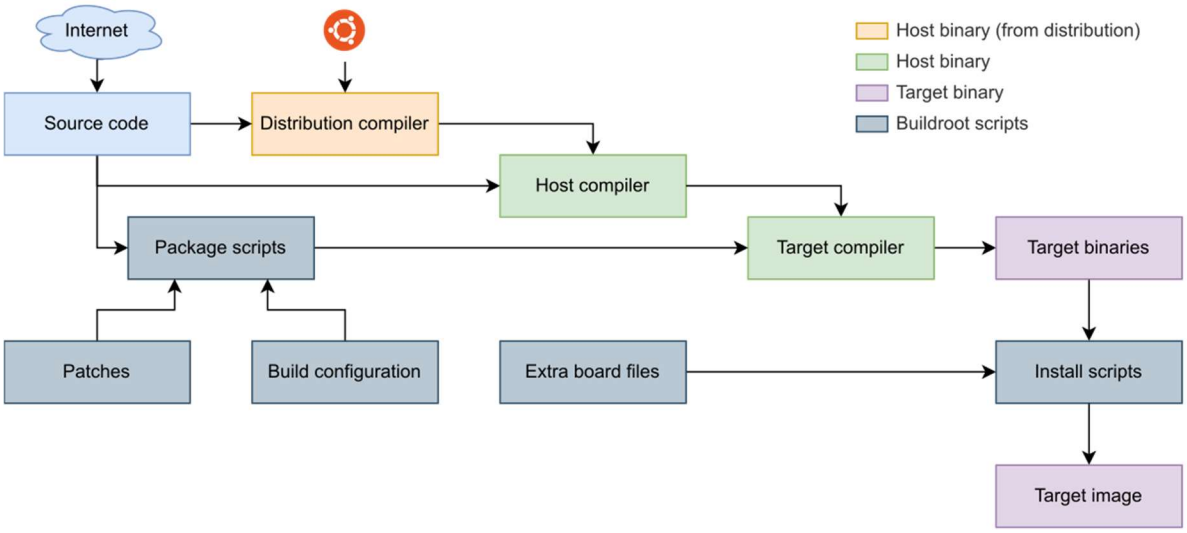

Figure 15: Buildroot build process

- **Virtualized Hardware Testing (QEMU):** We used QEMU to emulate the embedded hardware, enabling automated hardware-in-the-loop style testing in a virtual environment after every software or model change.
- **Automated Functional and Performance Testing:** The pipeline executed functional tests (verifying the correctness of image preprocessing) and performance tests (CPU utilization, memory footprint, processing speed), immediately flagging regressions or resource violations.

Figure 16 shows an overview of our automated CI/CD workflow.

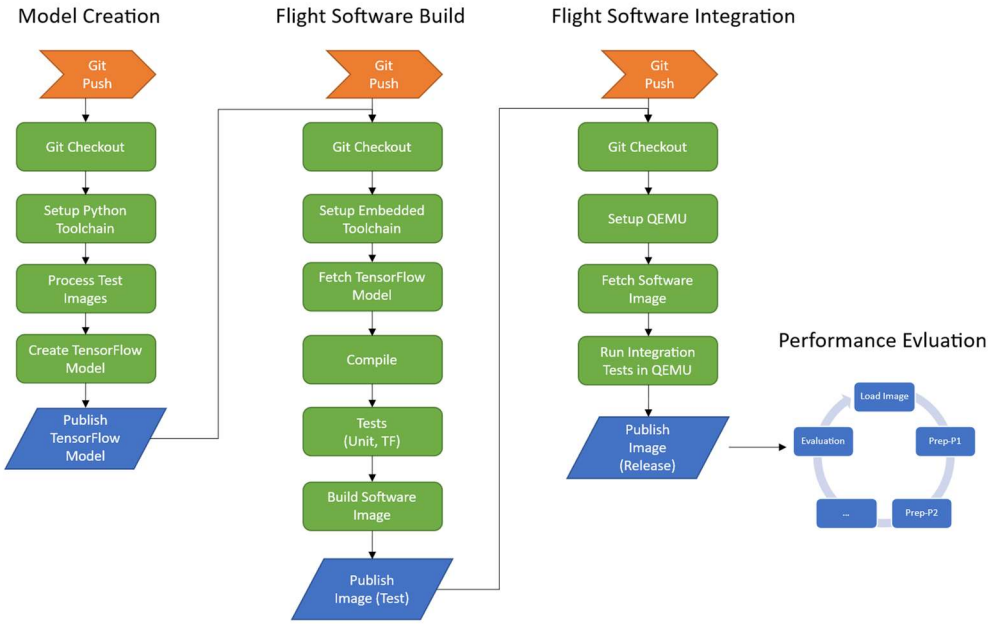

Figure 16: This illustration shows a preliminary CI/CD pipeline design, from Model Creation to Performance Evaluation.

8.4.3. Benefits

Though the focus of the study was much broader than building an automated CI/CD pipeline, using this approach of automated CI/CD and virtualization-driven provided significant benefits:

- **Rapid and Reliable Integration:** Continuous automated builds and tests enabled fast iteration cycles, quickly surfacing and resolving integration issues and running regular evaluations.
- **Improved Quality and Stability:** Regular functional and performance testing in realistic (virtualized) conditions significantly reduced runtime errors and integration faults.
- **Efficient Performance Evaluation:** Continuous performance evaluations allowed rapid testing and performance evaluation of new approaches, e.g., after changing the TF-based processing graph.
- **Enhanced and Parallel Collaboration:** Unified tooling and processes for software and ML engineers improved team transparency, alignment, and collaboration, allowing for parallel work to be executed.
- **Full Traceability and Reproducibility:** Version-controlled automation provided full traceability and effortless reproduction of builds and tests, significantly simplifying debugging and evaluation.

8.4.4. Lessons Learned

Key takeaways from this first study included:

- **TensorFlow as a Prototyping Tool:** Using TensorFlow allowed flexible experimentation with the image processing pipeline. It enabled us to evaluate and iterate individual processing steps quickly. However, performing computationally intensive tasks, particularly debayering and processing large images, on a CPU with limited memory proved suboptimal. Ideally, these image-processing steps should run on dedicated hardware like GPUs or FPGAs.
- **CI/CD and Virtualization Trade-offs:** Combining CI/CD with virtualization (QEMU) significantly improved our development workflow and enabled frequent automated testing without constant hardware access. Nevertheless, building the entire platform is resource-intensive and should be limited to periodic (e.g., nightly) builds rather than triggered continuously.
- **Realistic Performance Measurement Limitations:** While virtualization allowed accurate memory usage measurements, CPU performance measurements using QEMU on standard GitHub Actions runners were not realistic in absolute terms. However, relative CPU metrics remained useful for detecting performance regressions or improvements over time.
- **Flexible Platform Evaluation:** The pipeline structure allowed us to easily switch underlying platform components, for example, replacing Zephyr with Buildroot, and quickly evaluate changes using different QEMU CPU emulations. In our study, switching to Buildroot on a virtualized RISC-V CPU enabled the standard TensorFlow Lite runtime, which supports a broader instruction set. The Zephyr-based approach required TensorFlow Lite Micro, which supported fewer operations supporting our test scenarios.

8.5. Automated Multi-Platform Protocol Generation

Communication protocols are the lifeblood of any space mission, governing how data is exchanged between onboard and ground systems. In complex projects, there can be numerous interfaces – between onboard instruments, instruments, and the spacecraft platform, and between the spacecraft and various ground facilities. All of these interfaces must speak the same language (protocol) flawlessly. Even a minor inconsistency in a protocol implementation can lead to serious issues, from data misinterpretation to complete loss of communication with a subsystem. Defining and implementing these protocols traditionally involves writing detailed interface control documents and hand-coding the message structures and parsers in various software components (flight software, ground software, test tools, etc.). This manual approach is error-prone – ambiguities or mistakes in documentation can result in different teams implementing the protocol slightly differently. In the worst case, such issues might only be discovered during mission operations. Our team experienced this pain in past projects: late-stage bugs caused by a mismatch between the spacecraft sent telemetry and how the ground expected to decode it.

In this case study, we set out to apply Model-Based System Engineering (MBSE) principles to the problem of communication protocols. The idea was to have a single, authoritative definition of the protocol (a formal model) and then automatically generate all the software artifacts (code and documentation) from that model. By doing so, we aim to ensure complete consistency across all systems and reduce human error to near zero. The context included supporting standard space communication protocols

– specifically, those defined by CCSDS (Consultative Committee for Space Data Systems) and ESA's PUS (Packet Utilization Standard) – and any mission-specific custom protocols. The solution needed to produce code in multiple languages for use on different platforms (embedded C/C++ in flight software, maybe Ada for some systems, and even scripts or higher-level code for ground tools). The challenge and opportunity here was to streamline protocol handling software development and improve its quality dramatically.

8.5.1. Challenges

Some of the challenges inherent to communication protocol development that we aimed to overcome were:

- **Complex, Evolving Specifications:** Protocols for space systems (like CCSDS telemetry/telecommand or PUS services) are detailed and can evolve with new revisions. Manually aligning every piece of software with the latest spec update is tedious and error-prone. A small change in the packet definition (say, adding a field) must be propagated to many documents and codebases by hand, often leading to missing something.
- **Multiple Stakeholders and Interfaces:** In a typical mission, different teams might handle different segments (platform vs. payload, spacecraft vs. ground). Any team's misinterpretation of an interface document can introduce inconsistency. Moreover, each team might implement the protocol in a different programming language. Ensuring all those independent implementations behave identically is a major challenge.
- **Testing and Validation Overhead:** Verifying that every interface respects the protocol spec requires considerable effort. It usually involves writing separate test cases to send known data and check that it's interpreted correctly by the other side, essentially re-validating the protocol logic in each implementation. This is duplicate work if each side codes it independently.
- **High Stakes for Mistakes:** An outdated or misaligned protocol implementation in the space environment can cause severe operational disruptions. For example, if an instrument expects a telemetry packet in one format but the spacecraft sends it in another, the data could be lost or, worse, a command could be misinterpreted. These errors can be highly costly to diagnose and fix after launch. We wanted to eliminate the possibility of such mistakes by construction.

Given these challenges, the team turned to Abstract Syntax Notation One (ASN.1) as the formal modeling language for our protocols. ASN.1 is an internationally standardized language used to define data structures for communication protocols (widely used in telecom, cryptography, etc.). The idea was to have a single ASN.1 specification that describes all the messages (telemetry packets, telecommands, etc.) for our system and then generate everything. Figure 17 illustrates the concept: it shows the many places where protocol definitions come into play (onboard payload-to-platform comms, platform-to-ground, ground data systems, etc.). By modeling the protocol once, we could ensure that every actor in this diagram uses the exact same definitions.

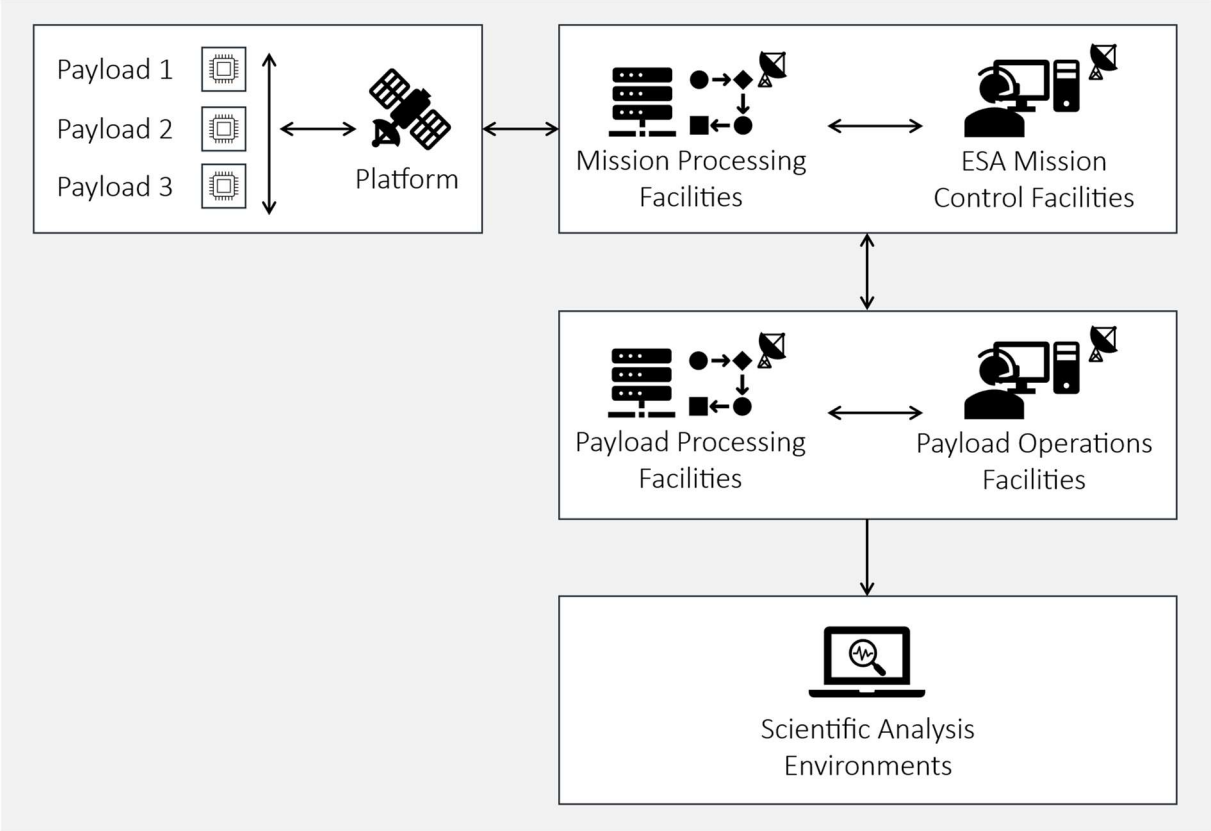

Figure 17: This illustration shows the communication paths between various actors in a space system. Protocol packets are generated, modified, integrated, and processed at each stage.

8.5.2. Approach

Our approach leverages the existing ASN1SCC compiler and the ASN.1-based PUS-C libraries, transforming them into a comprehensive, turn-key automation toolchain. This toolchain takes ASN.1 protocol definitions and automatically generates consistent code and documentation across multiple platforms and programming languages. We did not build ASN1SCC itself or the existing ASN.1 PUS-C libraries. However, we developed the ASN1SCC Scala backend, enabling the generation of formally verified Scala encoders and decoders. For other languages and targets, we can leverage existing ASN1SCC compiler backends.

Key elements of our implemented solution included:

- **Single Source of Truth (ASN.1 Specifications):** All protocol message definitions, such as spacecraft telemetry packets, telecommand formats, and similar data structures, are captured in ASN.1 syntax. There is a comprehensive ASN.1 PUS-C library available [RD5]. Each message is formally defined, specifying fields, data types, constraints, and complex structures (like nested sequences, choices, or enumerations). Thus, ASN. 1 is an authoritative specification source, replacing traditional textual documents and eliminating ambiguity.
- **Automated Code Generation:** Starting from these ASN.1 specifications, we automatically generated source code for various subsystems. We primarily targeted:

- **C** for flight software and embedded ground systems
- **Scala** for certain ground applications, particularly JVM-based mission control systems, benefiting from formally verified encoder and decoder generation
- **Ada** to maintain compatibility with legacy or safety-critical systems
- **Python** (under development) to create data processing pipelines and analysis software.

For instance, generated C code included appropriate data structures and encoding/decoding functions with built-in boundary checks. This ensured flight software did not require manually written packet-handling code, significantly reducing human error risks.

- **Automated Documentation Generation:** Besides executable code, the toolchain automatically generates human-readable documentation (HTML and PDF). This documentation serves as an interface control document (ICD), directly derived from ASN.1 definitions. It described field structures, data types, valid ranges, and constraints. Because documentation was generated from the same ASN.1 source as the code, documentation, and implementation remained consistently synchronized. New telemetry packet definitions immediately appeared in generated code and corresponding documentation, significantly reducing maintenance effort.
- **Integration with CI/CD:** Our toolchain can be integrated directly into continuous integration pipelines. Although ASN.1 specifications typically change infrequently, the toolchain automatically produces updated code and documentation when updates occur, ensuring consistent and rapid deployment.
- **Support for Standards (CCSDS/PUS):** Our solution leverages existing standardized ASN.1 PUS-C libraries to generate encoders/decoders compliant with standard PUS-C services. When missions require standard services (such as PUS Service 1 for Telecommand Verification or Service 6 for Memory Management), we directly use or extend these existing ASN.1 libraries. Thus, our approach inherently guarantees compliance with CCSDS/PUS standards.

Figure 18 illustrates this automated ASN.1-to-code workflow. In practical use, engineers update the protocol specification (e.g., adding a new telemetry message definition to support a new sensor or subsystem) in ASN.1. Once the specification has been updated, the automation toolchain runs, either manually or automatically as part of a CI pipeline, and immediately produces updated, consistent outputs: the C code used by flight software for telemetry packet encoding, the formally verified Scala classes used by the ground software for decoding, and the corresponding documentation snippets. These generated artifacts were then reviewed and integrated into their respective codebases. The key benefit is guaranteeing consistency across all outputs, eliminating the risks associated with manual, error-prone duplication of implementation efforts.

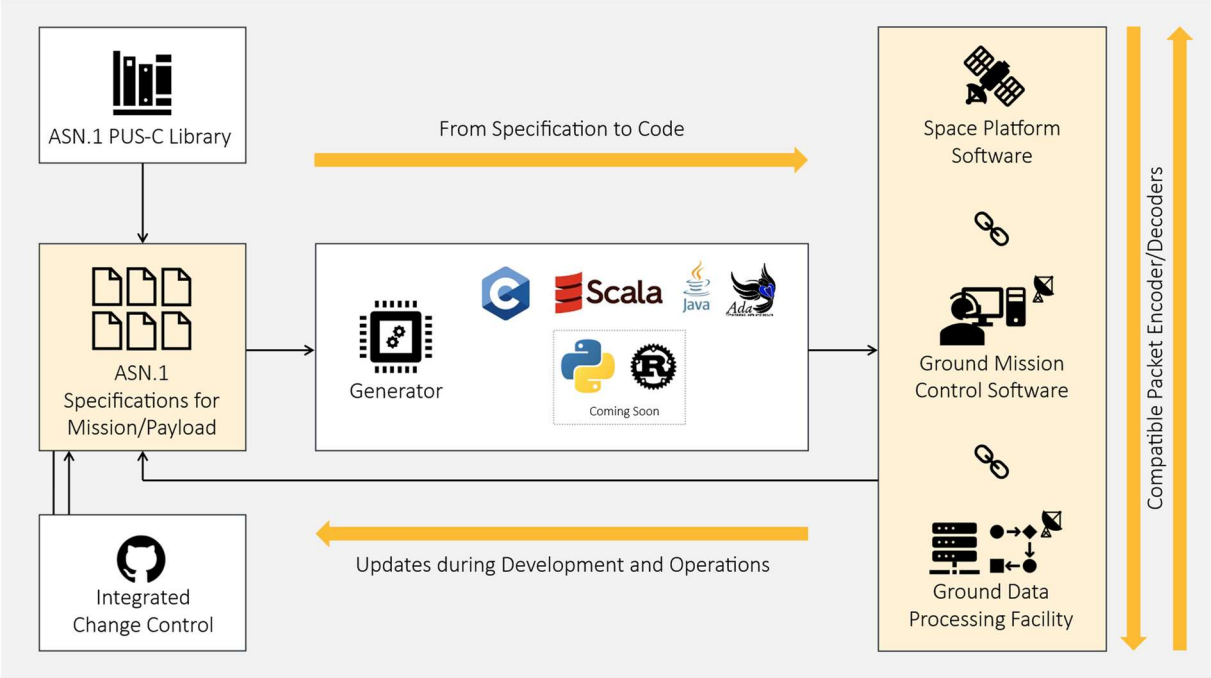

Figure 18: The automated ASN.1-to-Code process allows updates to the protocol specifications to be easily integrated into all software components and version-controlled.

We successfully applied this approach to new protocol developments and adopting existing standards. For example, to handle CCSDS standards, we utilized standardized ASN.1 modules. For PUS-C (the current Packet Utilization Standard revision), we use existing ASN.1 modules defining standard services. Our automation toolchain thus directly supports generating packets such as the "Successful Acceptance Verification Report" (PUS Service 1, Subservice 1, typically denoted TM(1,1)).

Figure 19 provides an example of such a generated output, illustrating how a TM(1,1) telemetry packet defined in ASN.1 translates directly into generated code across different languages and platforms.

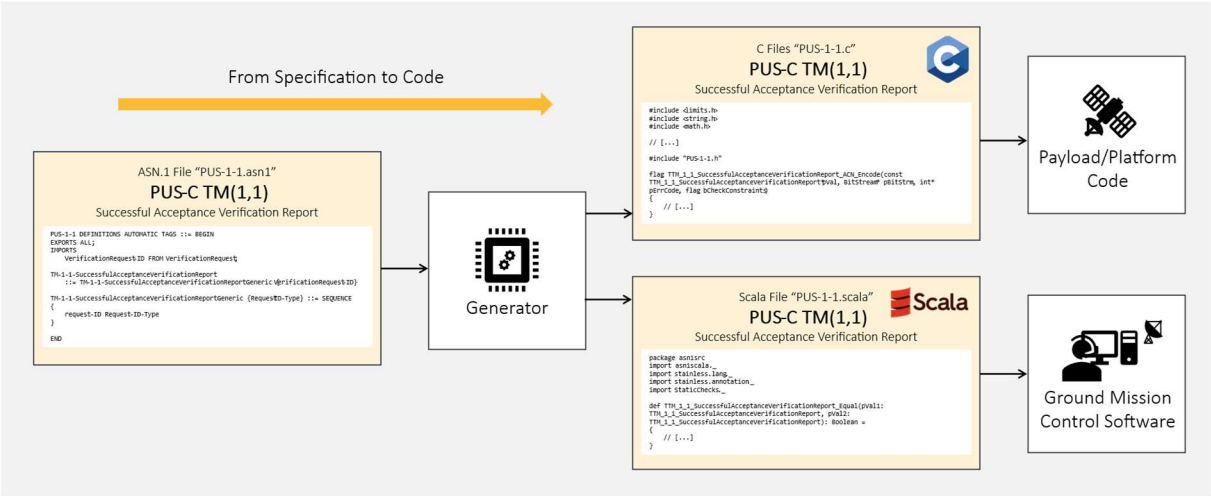

Figure 19: All communication packets are specified in ASN.1. Shown here is TM(1,1) "Successful acceptance verification report." The specification generates C, Scala, Ada, and other code that can be integrated into existing software.

8.5.3. Benefits

The automated protocol generation offers significant benefits:

- **Accelerated Development and Updates:** Defining protocol messages once in ASN.1 and generating code automatically across multiple languages and platforms significantly reduces development timelines. Updates and protocol changes are implemented rapidly by modifying a single specification, enabling end-to-end testing within hours rather than days or weeks. This makes the workflow agile and responsive to evolving system designs.
- **Enhanced Reliability and Quality:** The approach eliminates manual parsing and formatting code, removing a major source of errors. Automatically generated code from formal specifications is consistent and free from typical human mistakes, greatly reducing integration discrepancies. The ASN.1 specification is a single authoritative source of truth, ensuring reliable and consistent implementations across all subsystems.
- **Interoperability and Standards Compliance:** Because generated code strictly follows standardized ASN.1 definitions (e.g., CCSDS and PUS-C), interoperability with external tools and systems adhering to these standards is ensured by design. This simplifies integration with external partners or ground stations, minimizing the risk of incompatibilities and misunderstandings.
- **Scalability and Adaptability:** Multi-language and multi-platform support ensures the toolchain is flexible and future-proof. Adding new languages or platforms (such as Python or Rust) is straightforward. As projects scale in complexity and scope, the automated solution handles increasing complexity gracefully, allowing protocol maintenance effort to scale sub-linearly with system growth.
- **Consistency and Traceability:** Automatic generation ensures every generated artifact (code and documentation) directly maps back to a single, version-controlled ASN.1 specification. Naming and documentation inconsistencies are eliminated, simplifying verification, validation, and stakeholder communication. Changes are traceable through specification diffs, improving transparency and clarity.
- **Reduced Testing Burden and Increased Test Effectiveness:** By ensuring correct encoding and decoding through automated generation, testing efforts shift away from debugging low-level implementation errors toward more valuable, higher-level testing scenarios (e.g., robustness in packet loss, corruption, buffer overflows, or out-of-order delivery). As encoding correctness is guaranteed by the generation process, testing resources are effectively allocated to system-level integration and operational robustness, resulting in a more reliable and resilient communication subsystem with reduced effort.

8.5.4. Lessons Learned

This case study highlighted the power of model-based approaches and taught us valuable lessons:

- **Single Source of Truth is Invaluable:** Having one formal specification from which everything is generated proved to be a game-changer. We learned that eliminating duplicate definitions (in code, documentation, and across different systems) reduces errors and streamlines the workflow. The effort invested in maintaining this single source of truth is far lower than the cumulative effort of maintaining multiple parallel implementations. This reinforced our belief that Model-Based Systems Engineering (MBSE), when applicable, can significantly enhance consistency and efficiency. Having had a single source of truth would have helped us immensely with STIX (see Section 8.1), where the packet encoders and decoders had to be rewritten for multiple platforms and changed several times due to protocol adjustments.
- **Up-front Investment, Pay-off Later:** Developing and adopting an automated generation approach required an upfront investment in tooling and training. Initially, manually writing code for a few messages might seem faster. However, as the project scales and evolves, automation scales even better. Our lesson was that the early investment in automation pays off exponentially as the system grows. By the end of the project, we could not imagine having managed the protocols manually, given their complexity and evolution. This experience will encourage us to seek similar automation opportunities in other aspects of future projects beyond just protocol handling.
- **Standards Compliance by Design:** We experienced first-hand how embedding standards directly into our automated workflows removes ambiguity. Encoding standards into our tools and code generation processes ensure compliance "by design." This method proved far more reliable than relying on individual engineers to interpret a lengthy standard document correctly. This approach can also be extended to other domains, such as automatically generating tests from requirement specifications to ensure compliance.
- **Maintaining Flexibility:** While automation ensures consistency, we also learned the importance of maintaining flexibility. There were a few cases where generated defaults were not optimal, and we required manual interventions (for example, for performance tuning or exceptional cases). We designed our system to allow custom hooks or annotations in the ASN.1 specifications to handle these exceptions (such as instructing the generator to use specific encoding optimizations). The key lesson is that automation should not be a black box; having ways to intervene selectively is beneficial. Balancing fully automated generation with occasional tailored adjustments provides the best of both worlds: consistency, efficiency, and the ability to optimize when necessary.
- **Cross-Discipline Communication:** Implementing our solution required close coordination between systems engineers (defining protocols), software engineers (integrating generated code), and quality engineers (performing verification). It fostered a more collaborative approach to interface design. Instead of simply handing over an Interface Control Document (ICD), teams sat together to formalize protocols in ASN.1. This cultural shift was highly positive, fostering deeper engagement in interface design. People thought more carefully about every field (since they effectively were "coding by spec"), and we caught logical issues earlier at the specification stage. The lesson is that formal methods can significantly improve

communication, providing a concrete artifact (the model) that everyone can reference and discuss clearly.

- **Future-Proofing and Tool Evolution**: Finally, we recognized that establishing and maintaining a fully automated pipeline is not a one-time task; it requires ongoing care and updates. Although we leveraged an existing ASN.1 PUS-C library, we developed the ASN1SCC Scala backend to generate formally verified Scala encoders and decoders. By distinguishing between reusing mature libraries (such as the existing ASN.1 definitions for PUS-C) and developing customized generation backends (such as our Scala backend), we could better manage expectations and maintain a sustainable development process.