The Automotive Software Update Process Workflow: Development, Validation, Delivery, and Execution

Executive Summary

This report provides a comprehensive examination of the automotive software update process workflow, detailing its critical phases from development and release through validation, packaging, delivery, and in-vehicle execution. The automotive industry has undergone a significant transformation, evolving from a hardware-centric model to one profoundly influenced by software. This shift, driven by advanced driver-assistance systems (ADAS), autonomous capabilities, and connected car technologies, necessitates robust Over-The-Air (OTA) update mechanisms. The report highlights how OTA updates are not merely tools for bug fixes but essential enablers for continuous product improvement, enhanced safety, and the introduction of new features throughout a vehicle's lifecycle. Paramount importance is placed on integrating security by design, implementing rigorous validation procedures, and adhering to evolving regulatory frameworks such as UNECE WP.29 R156. The discussion underscores that a meticulously managed and secure software update pipeline is fundamental for ensuring vehicle safety, maintaining customer trust, and achieving competitive advantage in the era of the software-defined vehicle (SDV).

1. Introduction: The Imperative of Software Updates in Automotive

Evolution of Software in Vehicles

The automotive industry, traditionally characterized by its focus on mechanical components and manufacturing processes, has experienced a profound transformation. The advent of advanced driver-assistance systems (ADAS), autonomous vehicles, and sophisticated connected car technologies has elevated software to an increasingly critical component of modern vehicles.[1] This evolution has led to a dramatic increase in the volume of software embedded within vehicles. Current estimates suggest that an average vehicle contains over 100 million lines of code, with projections indicating this number could escalate to 300 million by 2030.[2] This extensive software infrastructure powers a wide array of essential features, including adaptive cruise control, lane departure warning, automatic emergency braking, infotainment systems, navigation, smartphone integration, and Vehicle-to-Everything (V2X) communications.[2]

A significant architectural shift accompanies this growth in software, moving towards the concept of the Software-Defined Vehicle (SDV). Unlike traditional distributed architectures where each vehicle function typically relied on its own Electronic Control Unit (ECU), the SDV paradigm utilizes a reduced number of more powerful, centralized ECUs or domain controllers.[3] These domain controllers, often built upon Systems-on-Chip (SoCs), consolidate functions across specific domains such as driving assistance, infotainment, and connectivity. This centralization not only simplifies the vehicle's electrical and electronic architecture and reduces weight but, crucially, enhances the connectivity capabilities essential for seamless software updates.[3] The increasing dominance of software and the architectural shift towards SDVs underscore the critical need for advanced software management capabilities, particularly in the realm of updates. Managing such vast and interconnected software through traditional physical updates, such as dealership visits, would be logistically and economically unsustainable for addressing bugs, deploying security patches, or introducing new features across a global fleet. This compels the automotive industry to adopt sophisticated Over-The-Air (OTA) update mechanisms.

Importance of Software Updates for Safety, Security, Performance, and New Features

Software updates have become indispensable for ensuring that vehicles remain safe, secure, and performant throughout their operational lifespan.[1] These updates enable

manufacturers to address a wide range of critical objectives. They facilitate the rapid resolution of bugs, thereby improving overall vehicle reliability.[5] Furthermore, software updates can significantly enhance ADAS and autonomous driving capabilities, leading to improved accuracy and reliability of these systems and, consequently, a reduced risk of accidents.[1] They are also vital for rectifying safety-related issues, including those necessitating recalls or addressing critical software bugs that could potentially lead to accidents.[1] Beyond safety, updates improve vehicle diagnostics and troubleshooting capabilities, allowing technicians to diagnose and repair issues more efficiently.[1]

The utility of software updates extends beyond corrective measures to proactive value addition. They enable the introduction of new features and functionalities, such as enhanced infotainment systems or novel driver assistance features, which significantly elevate the customer experience and foster brand loyalty.[1] Updates can also optimize vehicle performance and efficiency, contributing to reduced fuel consumption and emissions.[1] Critically, software updates serve as a primary mechanism for bolstering vehicle security by patching vulnerabilities and defending against evolving cyber threats.[1] Finally, they ensure compliance with evolving regulatory requirements, which are becoming increasingly stringent in the automotive sector.[5] This expansion of software functionality and the ability to update it remotely transforms the vehicle from a static product at the point of sale into a dynamic, evolving platform, redefining the OEM-customer relationship from a transactional one to a continuous service model. This paradigm shift has profound implications for customer loyalty, brand differentiation, and the potential for new revenue streams, such as subscription-based features enabled by ongoing OTA updates.

Overview of Over-The-Air (OTA) Updates

Over-The-Air (OTA) updates represent the process of remotely updating the firmware or software of embedded devices without requiring physical access.[6] This technology facilitates the replacement of current software on a device with new versions, which are downloaded wirelessly from a central server to the client device, typically a microcontroller within the vehicle.[8]

The general flow of an OTA update involves a sequence of critical steps designed to ensure secure and reliable delivery. Initially, the device receives a notification of a pending OTA update, which prompts it to decide whether to accept or ignore the

update.[7] Upon acceptance, the device begins downloading the update package wirelessly from the server, a step that often requires explicit user consent.[7] Due to the potentially large size of software packages, the new software is typically transferred as a sequence of binary bytes, often broken down into smaller packets.[8] Following the download, the device performs crucial verification checks to confirm the authenticity and integrity of the downloaded update package. This robust verification process is designed to prevent malicious actors from installing unauthorized firmware and to protect the device from corruption.[6] Once verified, the device proceeds to install the update package, a process that may involve reorganizing internal memory, initiating reboots, or other specific actions.[6] User consent may again be required at this installation stage.[9] Finally, after successful installation, the device validates the proper application of the update, often by performing post-install checks for functionality.[6] The device then reports the successful update status back to the update provider.[7]

OTA updates offer substantial benefits, including significant reductions in maintenance costs by eliminating the need for manual updates and physical visits to dealerships.[1] They also enhance security through the rapid deployment of security patches, thereby reducing the risk of vulnerabilities and cyberattacks.[1] The seamless delivery of new features and functionalities through OTA updates improves the overall user experience.[1] Furthermore, OTA updates provide manufacturers with increased flexibility, enabling them to respond swiftly to evolving market conditions, customer needs, and regulatory mandates.[5] However, the implementation of OTA updates is not without its challenges, which include managing security and compatibility issues, mitigating the risk of update failures, and addressing the variability across diverse device configurations.[1]

2. Software Update Process Workflow: A Foundational Overview

General OTA Update Process Flow

The high-level process for an Over-The-Air (OTA) update in embedded systems, particularly in automotive applications, involves a precise sequence of critical steps designed to ensure secure and reliable software delivery and installation. This

workflow is fundamentally structured to build resilience and trust in remote operations. The inherent risks associated with remote, wireless updates, such as malicious injection, data corruption, or network interruptions, necessitate the inclusion of robust verification and feedback loops. Without these safeguards, confidence in the update mechanism would erode, potentially leading to user reluctance and significant safety hazards.

The process typically commences with **Update Creation**, where the manufacturer develops and packages the new firmware or software update.[6] This involves compiling source code, linking it into an executable, and converting it into a portable binary format.[8] Once prepared, the next step is

Notification, where the target device is informed of a pending OTA update. This notification initiates a decision-making process on the device side, allowing it to choose whether to accept or ignore the update.[7] This step is crucial for user agency, which can significantly influence update acceptance.

Following notification, the **Download** phase begins, where the device wirelessly retrieves the update package from a server. This often requires explicit user consent before proceeding.[7] Given the potential size of binary files, the software is transferred as a sequence of bytes, frequently packetized to manage transfer efficiency.[8] The subsequent

Verification step is paramount. The device performs rigorous checks to confirm the authenticity and integrity of the downloaded update package.[6] This robust verification is a critical defense against malicious actors attempting to install unauthorized firmware and serves to protect the device from corruption. This process reflects a mature understanding that the "air" is an untrusted medium and that the device itself must be the final arbiter of update legitimacy and success, moving beyond a passive receiver of data.

Once verified, the **Installation** phase commences, during which the device installs the new software. This may involve organizing the new application into volatile or nonvolatile memory, initiating system reboots, or other specific actions required for the update to take effect.[6] User consent may again be required at this stage.[9] Finally, after installation, the

Validation step ensures the successful application of the update. This often includes post-installation checks to confirm full functionality.[6] The device then reports the successful update status back to the update provider, completing the cycle.[7] This

distributed trust model, where the device actively participates in self-preservation and validation, has implications for the processing power and memory requirements of embedded devices, as they must be capable of performing cryptographic checks and managing complex state transitions.[8] It also underscores the importance of a robust backend infrastructure that can track update status and respond to failures.

Benefits and Challenges of Automotive Software Updates

Automotive software updates offer a multitude of advantages that are transforming the industry, yet they also present significant challenges that require careful management.

Benefits:

- **Reduced Maintenance Costs:** OTA updates eliminate the need for manual updates and physical visits to dealerships or service centers, leading to substantial reductions in the time and cost associated with maintaining large fleets of vehicles.[1]
- **Improved Security:** The capability to rapidly deploy security patches through OTA updates significantly reduces the risk of vulnerabilities and cyberattacks, thereby maintaining the integrity and safety of the device and vehicle.[1]
- **Enhanced Customer Experience and Loyalty:** Seamless and efficient delivery of new features and functionalities, such as improved infotainment systems or new driver assistance features, directly improves the overall user experience and fosters greater customer loyalty.[1]
- **Increased Flexibility and Responsiveness:** OTA updates empower manufacturers to respond quickly to changing market conditions, evolving customer needs, and new regulatory requirements, allowing for agile product evolution.[5]
- **Improved Vehicle Performance and Efficiency:** Software updates can optimize vehicle performance and efficiency, potentially leading to reduced fuel consumption and emissions.[1] They can also enhance ADAS and autonomous driving capabilities, improving accuracy and reliability.[1]

Challenges:

- **Security and Compatibility Issues:** Ensuring that updates are secure from unauthorized tampering and remain compatible across the diverse range of

hardware and software configurations present in a vehicle fleet is a complex undertaking.[1]

- **Update Failure and Bricking Risk:** Issues such as network problems, device limitations (e.g., insufficient memory, processing power), or errors within the update package itself can lead to partial updates or, in severe cases, render Electronic Control Units (ECUs) inoperable, a condition known as "bricking".[5]
- **Device Variability and Fragmentation:** The wide array of device configurations and the inherent fragmentation across a large vehicle fleet make it challenging to ensure universal compatibility and consistent update success.[6]
- **Complexity of Infrastructure:** Implementing and managing OTA updates requires a sophisticated and complex infrastructure, encompassing robust communication protocols and backend systems to ensure secure and reliable delivery.[5]
- **Resilience to Power and Network Interruptions:** Devices must be designed to be highly resilient to power and network losses during the update process to prevent abrupt shutdowns and catastrophic failures, which can affect thousands or even hundreds of thousands of devices.[11]

To mitigate these challenges, best practices include thorough analysis of update logs, rigorous verification of update package integrity and authenticity, and meticulous checking of device configurations for compatibility.[6]

The following table provides a high-level overview of the key phases involved in the automotive software update workflow, illustrating the comprehensive scope of the process.

Table 1: Key Phases of Automotive Software Update Workflow

Phase Name	Brief Description	Key Activities/Goals
Update Development & Release	Creation and preparation of new software versions.	Requirements gathering, software design, coding, code review, version control, secure development practices.
Validation & Approval	Rigorous testing and regulatory compliance checks to ensure quality, safety, and security.	Unit, integration, system, and acceptance testing; simulation-based, HIL/SIL testing; OTA-specific validation; compliance with UNECE WP.29 R156 and

		ISO/SAE 21434.
Packaging & Delivery	Preparing the update for efficient and secure transmission to vehicles.	Generating full or incremental/delta update packages; applying cryptographic signatures and encryption; managing metadata; secure distribution over networks.
Vehicle Integration & Update Execution	The in-vehicle process of receiving, verifying, and installing the update on ECUs.	Telematics unit reception, OTA Manager orchestration, A/B partitioning or in-place updates, robust rollback mechanisms, post-update verification.

3. Update Development and Release

3.1. Automotive Software Development Lifecycle (SDLC)

The escalating complexity of software embedded within modern vehicles necessitates a highly structured and robust Software Development Lifecycle (SDLC). This formalized approach is critical for managing the vast and interconnected software components that define contemporary automotive systems. The sheer volume of code in modern vehicles, projected to reach 300 million lines by 2030 [2], and the inherent criticality of these systems, demand an industrial-grade software development approach. Ad-hoc methods are simply untenable in this environment. This necessitates the adoption of mature SDLC models, sophisticated version control systems, and automated processes to manage changes, ensure quality, and accelerate delivery at scale.

The SDLC typically begins with **Requirements Gathering and Analysis**. This foundational stage involves identifying the diverse needs and requirements of all

stakeholders, including end-customers, regulatory bodies, and internal development teams.[2] Key activities encompass stakeholder analysis, requirements elicitation through techniques such as interviews and surveys, and rigorous analysis to ensure that all requirements are clear, concise, and technically feasible.[2]

Following the definition of requirements, the process moves into **Design and Implementation**. During this phase, the software is meticulously designed and subsequently developed. This involves several critical activities: **Software Design**, where a detailed design for the software is created, specifying its architecture and individual components.[2] This is followed by

Implementation, which entails writing the actual code for the software, commonly utilizing programming languages such as C++ and Python.[2] A crucial activity within this stage is

Code Review, where the written code is systematically reviewed to ensure it meets all specified requirements and adheres to high-quality standards.[2]

The final stage of the core SDLC is **Testing and Validation**. This phase is dedicated to rigorously testing and validating the software to ensure it fully meets requirements, is safe, and operates reliably. This includes multiple levels of testing: **Unit Testing** for individual software components, **Integration Testing** to verify interactions between multiple components, and comprehensive **System Testing** to evaluate the software as a whole, ensuring it performs as expected in real-world scenarios.[2] The automotive industry is increasingly adopting agile and DevOps methodologies to achieve faster time-to-market and improved collaboration, which are critical for managing the rapid growth and complexity of vehicle software.[2] This implies that automotive OEMs must fundamentally transform their engineering capabilities and organizational culture to become proficient software development houses, with investment in software talent, advanced tooling, and continuous process improvement becoming as critical as traditional mechanical engineering.

3.2. Version Control Strategies for Embedded Systems

Version control is an indispensable aspect of software development, and its importance is amplified in the context of complex embedded systems found in automotive applications. It provides a clear history of changes made to a codebase,

enabling collaboration among developers and allowing for the easy reversion of changes if necessary.[13] Git is currently the most widely adopted version control system in the industry.[13]

The selection of a version control system is a critical initial step, with factors such as scalability for large repositories and numerous users being key considerations.[13] Once chosen,

Repository Initialization and Management involves creating a new repository, adding the codebase, and committing changes. Commits are intended to represent meaningful modifications to the source code, facilitating a clear historical record and the ability to revert specific changes.[13]

Access Controls are vital for maintaining the security and integrity of the codebase. Best practices include the use of SSH keys for authenticating users, implementing role-based access control to restrict access to certain users or groups, and applying branch permissions to limit access to specific branches.[13]

Branching Workflows are employed to manage different lines of development, releases, and fixes concurrently. Common types of branches include:

- **Feature Branches:** Dedicated to the development of new features or enhancements.
- **Release Branches:** Used to prepare software for a specific release or deployment.
- **Hotfix Branches:** Created to address critical issues or bugs requiring immediate fixes.[13]

  Popular branching workflows include Git Flow, which extensively utilizes feature, release, and hotfix branches, and Trunk-based development, which relies primarily on a single main branch supplemented by feature branches.13 General best practices for branching include using descriptive branch names to clearly indicate their purpose, enforcing branch permissions for security, and merging regularly to ensure changes are properly integrated and to minimize complex merge conflicts.13

Integration with Continuous Integration/Continuous Deployment (CI/CD) Pipelines is essential for automating the testing, validation, and deployment processes. Version control systems can trigger these automated pipelines through webhooks or specific CI/CD pipeline triggers whenever code is pushed to a repository.[13] This automation ensures that changes are continuously tested and

validated, improving software quality and accelerating delivery. Furthermore,

Integration with Issue Tracking Systems, such as Jira, is crucial for linking code changes directly to specific issues or bugs. This can be achieved through issue tracking system APIs or by referencing issues within commit messages.[13] For managing large binary files, which are common in embedded development, strategies like Git Large File Storage (Git LFS) or dedicated binary storage solutions such as Artifactory or Nexus are employed.[13] The scale of automotive software demands industrialized SDLC and version control, requiring OEMs to transform their engineering capabilities and organizational culture to become proficient software development houses.

3.3. Secure Software Development Principles

Given the inherently safety-critical nature of automotive software, security must be an intrinsic part of the entire development lifecycle, rather than an afterthought. The introduction of frameworks like Uptane highlights the extreme security risks and the need for robust defenses.

The **Uptane Framework** is a pioneering software update security system specifically designed for the automotive industry, engineered to withstand even nation-state level attacks.[15] Its design is based on a hierarchical security model, which ensures that the security of software updates does not degrade all at once. Instead, it mandates that multiple levels of access to vehicles or the automaker's infrastructure must be compromised before irreparable damage can be inflicted.[15] This multi-layered approach aims to prevent various types of harm even if an attacker gains access to servers, bribes operators, or infiltrates vehicular networks.

Key design principles underpinning Uptane include:

- **Resilience:** Acknowledging that all systems will eventually be vulnerable, Uptane is designed to minimize the damage caused by a compromise and to assure rapid recovery.[15]
- **Adaptability:** It employs a modular approach to problem-solving, allowing legacy systems to upgrade their security without requiring extensive retrofitting.[15]
- **Responsiveness:** The Uptane Standard is a dynamic document that quickly incorporates community input, ensuring it remains relevant against evolving

threats.[15]

Uptane's **Threat Model** addresses a wide range of vulnerabilities inherent in Electronic Control Units (ECUs), which have become increasingly complex and interconnected.[15] The threats it aims to prevent or deflect include:

- **Read updates:** This category focuses on intellectual property theft, where attackers attempt to eavesdrop on unencrypted updates to gain access to sensitive software designs.[15]
- **Deny updates:** Attacks in this category aim to prevent vehicles from receiving crucial software fixes. This includes tactics like drop-request attacks (blocking network traffic), slow retrieval attacks (delaying updates to exploit known vulnerabilities), freeze attacks (continuously sending outdated updates), and partial bundle installation attacks (allowing only a portion of an update to install).[15]
- **Deny functionality:** These attacks seek to cause vehicles to malfunction. Examples include rollback attacks (tricking an ECU into installing outdated, vulnerable software), endless data attacks (overwhelming an ECU with data until it crashes), mixed-bundles attacks (forcing incompatible software updates), and mix-and-match attacks (releasing arbitrary combinations of new image versions if repository keys are compromised).[15]
- **Control:** This represents the most severe threat, where an ECU is forced to install attacker-chosen software, leading to the attacker gaining full control of the unit and allowing arbitrary modification of vehicle performance.[15]

Challenges in software delivery, such as opaque processes, insufficient release planning, and poorly defined quality gates [16], are precisely the weaknesses that robust security frameworks and regulations aim to eliminate. The critical safety implications of vehicle software, combined with the escalating sophistication of cyber threats, compel a "security by design" philosophy throughout the entire software development and release process. Security cannot be an afterthought; it must be embedded from requirements gathering through coding, testing, and deployment. This implies that compliance with emerging cybersecurity regulations, such as UNECE WP.29 R155/R156, and industry standards like ISO/SAE 21434, is becoming a mandatory prerequisite for market entry. OEMs must invest significantly in secure coding practices, threat modeling, penetration testing, and the adoption of advanced security frameworks, making cybersecurity a core differentiator and a major area of investment.

4. Validation and Approval

4.1. Quality Assurance (QA) in Embedded Systems

Quality Assurance (QA) is a systematic process designed to ensure that products or services meet specific requirements and standards. In the context of embedded systems, particularly in the automotive sector, QA encompasses a range of activities, including comprehensive testing, verification, and validation, all aimed at detecting and rectifying defects early in the development cycle.[12] The fundamental goal of QA is to instill confidence that the product will perform as expected, satisfy customer needs, and comply with all relevant industry standards and regulations.[12]

The importance of quality in embedded systems cannot be overstated, especially given that these systems often operate in critical environments such as medical devices, automotive control systems, or aerospace applications, where any failure can have severe and potentially catastrophic consequences.[12] Therefore, ensuring the reliability, safety, and performance of these systems is paramount. Compliance with industry standards, such as IEC 61508 for functional safety and ISO 26262 for automotive functional safety, is a non-negotiable aspect of embedded systems development.[12]

Various QA processes and methodologies are applicable to embedded systems. Agile methodologies, with their iterative and incremental development approaches, emphasize continuous testing and feedback. The V-Model, a structured development process, focuses on testing and validation at each stage of the development cycle. DevOps principles aim to improve collaboration between development and operations teams, streamlining the entire pipeline.[12]

Testing is a critical component of QA, employing diverse strategies:

- **Functional Testing:** Verifies that vehicle systems and components operate precisely as intended, meeting all specified requirements.[17]
- **Performance Testing:** Evaluates a vehicle's performance characteristics, such as acceleration, braking, and handling capabilities.[17]
- **Safety Testing:** Assesses a vehicle's ability to protect occupants and other road users in various crash scenarios, ensuring compliance with safety regulations.[17]

- **Durability Testing:** Evaluates the long-term robustness and resilience of vehicle systems and components.[17]
- **Unit Testing:** Focuses on testing individual components of the software in isolation.[2]
- **Integration Testing:** Verifies the interactions and seamless functionality between multiple components or subsystems.[2]
- **System Testing:** Tests the entire system as a cohesive unit to ensure it meets specified requirements and functions correctly in real-world scenarios.[2]
- **Acceptance Testing:** The final stage of testing, ensuring the system meets customer requirements and is ready for deployment.[12]

Advanced testing technologies are increasingly adopted to overcome the limitations of traditional methods. **Simulation-Based Testing** utilizes computational models and simulations to evaluate vehicle performance, safety, and durability. This approach enables virtual testing of scenarios that are difficult or impossible to replicate physically and employs mathematical models to predict vehicle behavior under different conditions, reducing the need for physical prototypes.[17]

Hardware-in-the-Loop (HIL) Testing combines physical components with virtual models to test complex system interactions and functionality.[17]

Software-in-the-Loop (SIL) Testing involves testing software components in a virtual environment, ensuring they function correctly before integration with physical hardware.[17]

Effective **Test Planning and Execution** involves clearly defining test objectives, developing detailed and prioritized test cases, executing these cases according to the plan, meticulously reporting any defects encountered, and tracking their resolution until completion.[12] Various

Tools and Techniques support embedded systems QA, including simulators, emulators, and debuggers. Automated testing frameworks like JUnit, PyUnit, Unity, and Ceedling are used for writing unit and integration tests, often integrated into CI/CD pipelines for continuous testing and feedback.[12] Code coverage tools measure the extent of testing. Key

Metrics used to assess QA effectiveness include defect density, test coverage, Mean Time To Detect (MTTD), and Mean Time To Resolve (MTTR).[12] Furthermore, hardware-specific QA activities encompass verification of signals on PCBs, circuit protection testing, power consumption testing, antenna match testing, functional

testing with test firmware, heating and temperature range testing, and EMI/EMC evaluation.[18]

4.2. OTA Update Validation Process

The Over-The-Air (OTA) testing process is a crucial pre-launch activity, serving as the final gate before firmware is deployed to production devices.[10] Its primary objective is to validate that firmware updates can be remotely delivered, installed, and executed on embedded or IoT devices without causing system failure.[10] This comprehensive validation process is essential to prevent issues such as bricking due to partial or interrupted updates, security flaws, or hardware variant incompatibilities.[10] The inherent high risks associated with remote software updates in safety-critical automotive environments, such as the potential for bricking, the introduction of new vulnerabilities, or power loss during an update, mandate the implementation of highly resilient testing strategies and fail-safe mechanisms. The ability to automatically revert to a previous, known-good state is not merely a feature but a fundamental requirement for maintaining device functionality, ensuring vehicle safety, and building customer trust.

Key validation steps within the OTA testing process include:

- **Firmware Image Validation:** This involves verifying that the OTA update package is correctly signed, hashed, and appropriately sized for the intended hardware and software version constraints.[10] This step is critical for ensuring the authenticity and integrity of the update.
- **Recovery and Rollback Testing:** It is imperative to ensure that devices possess robust mechanisms to recover in the event of a bad or corrupted update. This includes thoroughly testing the device's ability to revert to a previous, known-good firmware version if the update fails or introduces critical errors.[10]
- **Failure Scenario Simulation:** To validate the system's resilience, issues such as incomplete downloads, corrupt firmware files, or power loss during installation are intentionally introduced and tested.[10] This proactive testing of failure modes ensures predictable behavior and automatic recovery where possible.[10]
- **Field Network Condition Testing:** This assesses the OTA performance under unstable real-world conditions, including testing with weak signal strength, intermittent connectivity, and scenarios involving long installation durations.[10]
- **Post-Update Device Monitoring:** After updates are deployed, it is critical to

continuously track various metrics such as boot success rates, memory usage, error rates, and telemetry data. This monitoring helps to catch any regressions or instability early in the rollout process.[10]

- **Secure Boot and Authentication Checks:** This step confirms that the updated firmware is properly authenticated and that the device enforces secure boot policies at runtime, preventing unauthorized code execution.[10]

It is recommended to conduct at least 100 successful OTA test updates across representative hardware variants before promoting any firmware to production. This extensive testing helps to uncover regression bugs, memory issues, or compatibility gaps that might not be apparent in laboratory tests alone.[10] This comprehensive validation and rollback strategy reflects an industry-wide recognition that OTA updates, while beneficial, also represent a significant attack surface and point of failure. The emphasis is on building resilient protection that acknowledges the inevitability of system compromises and focuses on minimizing damage and assuring rapid recovery.

4.3. Regulatory Compliance: UNECE WP.29 R156

Regulatory scrutiny has elevated quality assurance from a best practice to a legal mandate within the automotive industry. As of July 2022, compliance with UNECE WP.29 UN R156 became mandatory for new whole vehicle types seeking EU type approval.[19] This regulation specifically requires the establishment and operation of a Software Update Management System (SUMS).[19] This legal enforceability directly compels automotive OEMs to implement not just quality assurance, but demonstrably traceable and auditable QA and validation processes for software updates. It transforms QA into a critical compliance gate, requiring OEMs to prove, through meticulous documentation and process adherence, that every step of their update lifecycle meets stringent safety and security criteria.

Manufacturers are explicitly required to have a documented process for Over-The-Air (OTA) updates, which must be made available to approval authorities upon request. This means that every aspect of the OTA software update process needs to be meticulously recorded and executed in a manner that satisfies the regulation's stringent standards to receive certification.[20]

The detailed requirements under Chapter 7.1.1. Processes to be verified at initial

assessment of R.156 include:

- **7.1.1.1. Processes at the Manufacturer:** The manufacturer must possess a documented process for certification, and all these processes must be readily available to certifying authorities upon request.[20]
- **7.1.1.2. Unique Identifiers:** All software versions and updates must be distinguishable by unique identifiers. This is crucial for ensuring the validity of updates within the vehicle and for preventing the deployment of unverified or unapproved updates.[20]
- **7.1.1.3. Vehicle Software Database:** A comprehensive record of each vehicle's software history is required. This includes detailed documentation of update failures as well as updates pending installation.[20]
- **7.1.1.4. Vehicle Manifest:** OEMs must maintain a complete vehicle manifest that enables the traceability of all software and hardware changes to every vehicle in the fleet, accounting for varying combinations of software and hardware at any given time.[20]
- **7.1.1.5. Vehicle Integration:** The OTA software update processes must incorporate a mechanism for integration verification. This enables the OEM to identify interdependencies between the updated system and existing systems within the vehicle.[20]
- **7.1.1.6. Vehicle Grouping:** The software system must possess the capability to group vehicles by specific features or identifiers, such as vehicle age, location, color, drivetrain, or trim level. This facilitates targeted updates, allowing, for example, an update applicable only to automatic transmissions to be distributed exclusively to those vehicles.[20]
- **7.1.1.7. Compatibility Validation:** A robust process must be in place to validate the specific hardware and software combinations within a particular vehicle grouping. This is essential for future update rollouts to accurately assess the compatibility of new updates with existing components.[20]
- **7.1.1.8. Change Impact Analysis:** Before any update is disseminated, it must undergo a thorough analysis to identify its potential impact on the entire system. This includes assessing whether the software update will create problems when integrated and if it will perform its intended function once installed.[20]
- **7.1.1.9. Feature Update Analysis:** This level of analysis specifically examines potential changes to customer-facing features and vehicle performance, scrutinizing whether an update will modify any existing features beyond acceptable parameters.[20]
- **7.1.1.10. Safety Impact Analysis:** Measures must be implemented to test whether an update will integrate safely or introduce new hazards. For instance, an update

designed to repair a headlight fault that inadvertently creates a braking issue would fail this crucial analysis.[20]

- **7.1.1.11. User Information:** A clear process must be established to notify vehicle owners of new software installations and changes. The owner must be provided with the option to accept or postpone the change based on when the vehicle will not be in use. Notifications regarding failed updates and installation issues are also required.[20]
- **7.1.1.12. Update Documentation:** Every single aspect listed above must be thoroughly documented and made available for review upon request.[20]

All these requirements must be satisfied by the OTA software update processes to obtain WP.29 compliance certification. This regulatory pressure will lead to significant investments in automated testing frameworks, sophisticated simulation environments, and dedicated compliance and audit teams. It also necessitates greater transparency and auditability throughout the entire software development and deployment pipeline, potentially requiring new internal organizational structures and external certifications. Non-compliance risks severe market access restrictions.

Table 4: UNECE WP.29 R156 Key Requirements Summary

Requirement ID	Requirement Name	Description/Purpose
7.1.1.1	Processes at the Manufacturer	Manufacturer must have documented processes for certification, available to authorities.
7.1.1.2	Unique Identifiers	All software versions must have unique identifiers to ensure validity and prevent unapproved updates.
7.1.1.3	Vehicle Software Database	Comprehensive record of each vehicle's software history, including update failures and pending updates.
7.1.1.4	Vehicle Manifest	Full manifest for traceability of all software and hardware changes to every vehicle in the fleet.

7.1.1.5	Vehicle Integration	Mechanisms to identify interdependencies between updated and existing vehicle systems.
7.1.1.6	Vehicle Grouping	Ability to group vehicles by features (age, location, drivetrain) for targeted updates.
7.1.1.7	Compatibility Validation	Process to validate hardware/software combinations within vehicle groupings for future updates.
7.1.1.8	Change Impact Analysis	Thorough analysis of potential system-wide impact before any update.
7.1.1.9	Feature Update Analysis	Analysis of potential changes to customer-facing features and vehicle performance.
7.1.1.10	Safety Impact Analysis	Measures to test if an update integrates safely or introduces new hazards.
7.1.1.11	User Information	Process to notify owners of updates, provide options to accept/postpone, and report failures.
7.1.1.12	Update Documentation	All aspects of the update process must be thoroughly documented and available for review.

4.4. Relation to ISO/SAE 21434

The regulatory framework established by UNECE WP.29 UN R155 and R156 sets a

common international framework for vehicle cybersecurity, with these regulations officially coming into force in January 2021.[19] UN R155 specifically mandates the operation of a certified Cybersecurity Management System (CSMS), while UN R156 requires a Software Update Management System (SUMS) as a future condition for type approval.[19] Together, these UNECE regulations define four key disciplines: managing cyber risks to vehicles, securing vehicles "by design" to mitigate supply chain risks, detecting and responding to security incidents across vehicle fleets, and safely and securely updating vehicle software, which includes providing a legal basis for over-the-air updates.[19] This comprehensive approach extends beyond mere vulnerability testing to encompass a broader cybersecurity strategy.

ISO/SAE 21434, titled "Road vehicles — Cybersecurity engineering," is a standard co-developed by the International Organization for Standardization (ISO) and the Society of Automotive Engineers (SAE). This standard specifically focuses on cybersecurity risks throughout the design and development phases of car electronics.[19] The WP.29 cybersecurity regulations (UN R155/R156) and the ISO/SAE 21434 standard were developed in parallel, ensuring that they are complementary and non-contradictory. This synchronized development ensures that the regulatory requirements can be met by applying the existing standards.[19] While the UNECE regulations are mandatory in certain countries and outline specific requirements, the ISO/SAE 21434 standard represents the industry's consensus on the essential cybersecurity practices that should be followed to achieve a relative level of security across the automotive market.[19] This close relationship signifies that adherence to ISO/SAE 21434 is a practical pathway to demonstrating compliance with the UNECE regulations.

5. Packaging and Delivery

5.1. OTA Update Packaging Formats

Over-The-Air (OTA) update packages are typically constructed using specialized tools, such as Android's ota_from_target_files utility, which processes a target-files.zip input.[21] The choice of packaging format is a core technical and strategic decision in

this phase, influencing bandwidth, storage, and update efficiency.

Two primary types of update packages are commonly employed:

- **Full Updates:** These packages contain the entire final state of the device, encompassing system, boot, and recovery partitions.[21] Their advantage lies in their robustness; they can install a build regardless of the device's current state, provided the device is capable of receiving and applying the package.[21] However, the comprehensive nature of full updates results in larger package sizes, requiring more network bandwidth and storage.
- **Incremental/Delta Updates:** These packages are designed to transmit only binary patches, or "deltas," representing the changes between the current and the new software versions.[21] This approach leads to significantly smaller package sizes, which in turn reduces network bandwidth consumption and storage requirements.[5] The primary limitation of incremental updates is their dependency: they can only be installed on devices that possess the exact source build used to construct the package.[21] To ensure devices can catch up with updates, it is recommended to offer a full update after every 3-4 incremental updates.[21]

The large size of full software images for complex automotive systems and the high costs associated with data transfer over cellular networks for OTA updates compel the adoption of delta update technologies. Without this optimization, frequent, large-scale OTA deployments across a vehicle fleet would be prohibitively expensive, consume excessive bandwidth, and lead to longer update times, negatively impacting user experience and operational efficiency. This necessitates significant investment by OEMs in sophisticated build and packaging tools capable of efficiently generating and managing delta updates. It also influences network infrastructure planning, data consumption models, and ultimately the total cost of ownership for connected vehicle services, making delta updates a fundamental enabler for widespread, cost-effective OTA.

For devices running Android 11 and higher, a single OTA package can be built to support multiple device Stock Keeping Units (SKUs). This functionality requires configuring target devices to utilize dynamic fingerprints and updating the OTA metadata to include the device name and fingerprint in the pre- and post-condition entries.[21] SKUs represent variations of combined build parameter values, allowing OEMs to share a single image across devices with minor customizations but different product names.[21]

Every OTA package includes a metadata file, typically located at META-INF/com/android/metadata, which describes the package. This metadata

specifies the required pre-condition (the source build state) and the post-condition (the target build state) for successful installation.[21] These values are derived from corresponding build properties, such as

ro.product.device, ro.build.version.incremental, and ro.build.fingerprint.[21]

A/B updates have also introduced the concept of **Streaming Updates**, where the device can apply the update as it reads it directly from the network, thereby eliminating the need for dedicated download space.[9] In contrast,

Non-Streaming Updates require the entire zip file to be downloaded before the device updates can be applied.[9]

Table 2: Comparison of Full vs. Incremental/Delta Updates

Feature	Full Updates	Incremental/Delta Updates
Mechanism	Contains entire final state of device (system, boot, recovery partitions).	Contains only binary patches (deltas) representing changes.
Package Size	Larger.	Significantly smaller.
Network Bandwidth Impact	Higher.	Lower, minimizes impact on network bandwidth.
Storage Requirement	Requires more storage space on device for download.	Requires less storage space for download.
Installation Time	Can be longer due to larger data transfer.	Generally faster due to smaller data transfer.
Dependency on Previous Version	No reliance on previous firmware; can install regardless of current state.	Highly dependent on the exact source build; only installs if previous version matches.
Use Cases	Initial deployments, major version upgrades, recovery from corrupted states, catching up after multiple incremental updates.	Frequent bug fixes, security patches, minor feature updates, incremental software improvements.

Advantages	Highly robust, can fix corrupted systems, simpler distribution logic.	Reduced data transfer costs, improved efficiency, faster updates, less user disruption.
Disadvantages	High bandwidth usage, longer download times, more storage needed.	Requires specific previous version, more complex server-side generation, potential for "stale" devices if not regularly updated.

5.2. Delta Updates in Embedded Systems

Delta updates represent a critical implementation strategy for delivering software updates to embedded systems, particularly in industries such as automotive, IoT, and consumer electronics, where bandwidth and storage resources are often limited.[22] This approach involves transmitting only the changes, or 'deltas,' between the current and new versions of the software, rather than the entire software package.[22]

The core advantages of utilizing delta updates are significant:

- **Reduced Data Transfer:** By transmitting only the modified portions, delta updates substantially minimize the amount of data transferred.[5] This is especially crucial in automotive systems for updating Electronic Control Units (ECUs), where minimal data transfer leads to reduced costs, particularly over cellular networks, and enhances overall efficiency.[22]
- **Bandwidth and Storage Efficiency:** This method directly addresses the constraints prevalent in environments with limited bandwidth and storage capacity, making it a practical solution for widespread device deployment.[22]
- **Improved Efficiency:** The reduced update size inherently improves the efficiency of the update process, decreasing the time required for completion.[5]
- **Seamless Updates:** Delta updates facilitate a smooth application process without disrupting the device's functionality. This is vital for critical applications like medical devices, industrial equipment, and automotive systems, where continuous operation is essential.[22]
- **Maintenance of Functionality and Security:** This approach is fundamental for maintaining system functionality and ensuring that devices remain current with the latest features and crucial security patches.[22]

Common use cases for delta updates include incremental updates and the deployment of patches for bug fixes or security vulnerabilities.[5] The large size of full software images for complex automotive systems and the high costs associated with data transfer over cellular networks for OTA updates compel the adoption of delta update technologies. Without this optimization, frequent, large-scale OTA deployments across a vehicle fleet would be prohibitively expensive, consume excessive bandwidth, and lead to longer update times, negatively impacting user experience and operational efficiency.

5.3. Secure Software Delivery Mechanisms

Securing the Over-The-Air (OTA) update process is of paramount importance to prevent unauthorized access, tampering, and the injection of malicious code into vehicles.[24] This requires a multi-layered defense strategy that protects the software at every stage: in transit, at rest within the vehicle, and against manipulation by compromised in-vehicle components.

The establishment of **Secure Protocols and Encryption** is foundational. The use of secure protocols such as HTTPS and Transport Layer Security (TLS) is essential for creating secure connections between OEM servers and vehicles, thereby protecting data during transmission over cellular networks.[5] Encryption, often using algorithms like AES, prevents unauthorized parties from reading plaintext firmware, which safeguards intellectual property and sensitive data from reverse engineering or theft.[5]

Authenticity and Integrity Protection are critical to ensure that firmware originates from a trusted source and has not been modified in transit. This is achieved through methods such as HMAC, CMAC authentication, or digital signatures.[24] These mechanisms also protect the communication channel against "man-in-the-middle" attacks, where an attacker intercepts and potentially alters the update package.[24] Proper

Secure Key Management practices are also crucial for managing the cryptographic keys utilized for software updates.[5]

The **Trusted Location of the OTA Manager** within the vehicle is a strategic security consideration. The in-vehicle OTA Manager, which orchestrates the update process, must be robustly protected against manipulation. The central gateway ECU is often

considered an ideal and trusted location for this component. Its role as a firewall between external and internal networks, coupled with its typical execution of only OEM-trusted software, provides essential isolation from potential external attack vectors like cellular, Wi-Fi, or USB interfaces.[24]

Lifecycle Management features are necessary to securely log the current firmware versions and to prevent unauthorized rollbacks to older, potentially vulnerable software versions.[24] Frameworks like Uptane are specifically designed for resilient protection, aiming to minimize damage and assure rapid recovery even if system compromises occur.[15] This comprehensive security posture is absolutely essential to prevent catastrophic outcomes such as vehicle hijacking, intellectual property theft, or widespread operational failures. It demands deep expertise across various security domains (cryptography, network security, embedded system hardening) and continuous security audits and penetration testing across the entire delivery chain, from development servers to the vehicle's ECUs. This makes robust security a non-negotiable aspect of OTA implementation and a significant area of ongoing investment for OEMs.

6. Vehicle Integration and Update Execution

6.1. Automotive ECU Update Mechanism

The in-vehicle update process is intricately orchestrated by key components to ensure the secure and efficient delivery of new software to individual Electronic Control Units (ECUs).[24] This process begins with a secure connection established over a cellular network between the OEM's servers and the vehicle's Telematics Unit. The updated firmware is then securely transmitted to the Telematics Unit, which subsequently forwards it to the OTA Manager.[24]

The **OTA Manager** serves as the central nervous system for in-vehicle software lifecycle management. It is a crucial component that orchestrates the entire update process for all ECUs within the vehicle. Its responsibilities include controlling the distribution of firmware updates, instructing individual ECUs when to perform updates

(which is particularly important for simultaneous multi-ECU updates), and sending confirmation back to the OEM upon successful completion.[24] This component is far more than a simple download client; it acts as a gatekeeper, a coordinator, a security enforcer, and a recovery agent for the entire vehicle's software ecosystem. Its capabilities directly influence the efficiency, security, and reliability of complex, multi-ECU updates characteristic of modern Software-Defined Vehicles (SDVs).

For **Storage and Backup**, the OTA Manager can utilize external NAND flash memory. This memory stores firmware updates until they are needed and holds backup copies of firmware for other ECUs. These backups are secured through encryption and authentication to prevent tampering, providing a critical safety net in case of update failures.[24] The OTA Manager also plays a vital role in

Verification and Authentication. It maintains a comprehensive table of all ECUs within the vehicle, including their serial numbers and current firmware versions. This information allows the OTA Manager to verify incoming updates and ensure they are authorized for the specific vehicle. If a target ECU lacks inherent security functionality, the OTA Manager assumes responsibility for decrypting and authenticating the update on its behalf.[24] Furthermore, it prevents unauthorized rollbacks by checking firmware version numbers against a secure record.[24]

The **Secure Location** of the OTA Manager is paramount for vehicle security. The central gateway ECU is often considered an ideal and trusted location for the OTA Manager. This is because it acts as a firewall between external and internal networks and typically runs only OEM-trusted software, providing isolation from potential attack vectors that hackers might exploit through external interfaces.[24]

While software mechanisms can facilitate updates, specific **Required Microcontroller Features** significantly enhance update speed, security, and reduce vehicle downtime. These include:

- **Read-While-Write flash:** Allows the application to continue execution while another flash block is being erased or programmed.
- **Flash remapping/MMU:** Enables instant switching between new and old firmware images stored in different physical locations.
- **Cryptographic Security:** Facilitates secure storage of private keys for decryption and authentication of incoming updates.
- **Lifecycle Management:** Securely logs the current firmware version and prevents unauthorized rollbacks or software installations.
- **Small code flash block sizes:** Improves update speed and efficiency, especially for in-place differential updates.

- **Lockable flash regions:** Provides protection for critical code like the bootloader during updates.
- **Brownout Detection:** Detects unexpected resets during updates, allowing the bootloader to initiate a recovery attempt.
- **Multi-core architectures:** Can dedicate one core to performing the update while another runs the existing application, minimizing performance impact.
- **System Integrity Checking:** Detects and prevents system anomalies during an update.[24]

The development, securing, and continuous evolution of the OTA Manager become a core strategic competency for OEMs. Its design decisions directly impact the vehicle's overall electrical/electronic architecture, internal network communication protocols, and cybersecurity posture. Its ability to manage interdependencies between various ECUs, as required by UNECE WP.29 R156 for "Vehicle Integration," is crucial for enabling complex new features and ensuring the holistic integrity of the software-defined vehicle.[20]

6.2. Update Execution Methods: A/B Partitioning vs. In-Place

The choice of software update execution method within the Electronic Control Unit (ECU) involves a critical trade-off between hardware cost, vehicle downtime, and the inherent risk of update failure.[24] The paramount importance of vehicle safety and the high customer expectation for continuous, uninterrupted operation drives the automotive industry's preference for certain methods, despite potential higher costs.

A/B Partitioning (Seamless Updates):
This method is becoming the gold standard for automotive OTA execution due to its inherent safety and user experience advantages.

- **Mechanism:** A/B partitioning involves maintaining two complete copies of updateable software components (typically labeled Part A and Part B) on the device's flash memory.[9] The vehicle operates from one partition (the active partition) while the new software version is securely installed on the inactive partition in the background.[11]
- **Benefits:**
    - **Seamless Operation:** A key advantage is that the vehicle can continue normal operation during the update process, resulting in no vehicle downtime for the user.[24]

    - **Robustness and Rollback:** If the update process is interrupted (e.g., due to power loss or network issues) or if the new software fails validation checks, the active partition remains unchanged, ensuring continued functionality. A subsequent reboot can seamlessly direct execution back to the previous, known-good partition, guaranteeing a reliable rollback mechanism.[9] This ability to guarantee a working system and recover gracefully from failed updates is a non-negotiable requirement in this safety-critical domain.
    - **Streaming Updates:** A/B devices can apply updates as they are read directly from the network, reducing the need for large dedicated download space on the device.[9]
- **Disadvantages:** The primary drawback is the requirement for double the flash memory on each ECU, which leads to increased hardware costs.[24]
- **Android Implementation:** Modern Android devices (Android 11 and later) utilize Virtual A/B with compression, which maintains two physical slots for boot-critical partitions and uses compressed snapshots for dynamic partitions to reduce space requirements. Legacy A/B (Android 10 and earlier) kept two copies of *every* single partition.[25] The update_engine is a system service specifically designed to support A/B updates on Android devices.[9]

In-Place Updates:

- **Mechanism:** In contrast, the in-place update method involves only one version of the firmware existing on the device. During the update, individual blocks of this single firmware are erased and reprogrammed.[24]
- **Disadvantages:** The most significant drawback is that the update cannot run while the ECU is in normal operation. This means the vehicle will be inoperable for the duration of the update process, which can range from tens of seconds for large ECUs to longer periods.[24] Furthermore, an error or unexpected reset during an in-place update can be difficult to recover from, potentially rendering the module (and consequently, the entire car) non-functional until it receives professional servicing.[24]

The choice between A/B partitioning and in-place updates involves a critical trade-off between the added hardware cost of the A/B method and the customer inconvenience and higher risk associated with the in-place method. This implies that future automotive ECU hardware designs must explicitly account for the increased memory footprint required by A/B partitioning. Furthermore, it necessitates the development of sophisticated bootloader and system management software to handle partition switching, self-test procedures, and complex rollback logic effectively. This

choice reflects a mature understanding of operational risk and customer experience in the connected vehicle ecosystem.

Table 3: A/B Partitioning vs. In-Place Update Methods

Feature	A/B Partitioning	In-Place Update
Mechanism	Two copies of software (A & B); update written to inactive, then switch.	Single copy of software; individual blocks erased/programmed in place.
Vehicle Downtime	None (update in background).	Significant downtime (vehicle inoperable during update).
Risk of Bricking	Very low (always a working partition to roll back to).	Higher (error/reset during update can render ECU non-functional).
Memory Requirement	Requires double the flash memory.	Requires less flash memory (single copy).
Rollback Capability	Robust, instant rollback to previous working version.	Difficult to recover; potential for permanent failure without external intervention.
Suitability for Safety-Critical Systems	Highly suitable; preferred due to resilience and minimal risk.	Less suitable; higher risk profile for critical systems.
Advantages	Seamless user experience, high reliability, robust failure recovery, enables streaming updates.	Lower hardware cost.
Disadvantages	Higher hardware cost (more memory).	Vehicle inoperability, higher risk of update failure, complex recovery from errors.

6.3. Robust Rollback Procedures

The primary purpose of implementing robust rollback procedures in OTA updates is to ensure that the device remains functional even if a new software version introduces critical errors or fails during installation.[26] This fail-safe mechanism is essential for maintaining vehicle safety and operational continuity.

When the rollback process is enabled, and an OTA update provides a new version of the application, a specific sequence of states and actions occurs:

1. The new application is successfully downloaded, and the esp_ota_set_boot_partition() function marks this partition as bootable and sets its state to ESP_OTA_IMG_NEW. This state indicates that the application is new and requires monitoring during its first boot.[26]
2. Upon reboot (esp_restart()), the bootloader checks for the ESP_OTA_IMG_PENDING_VERIFY state. If this state was set (meaning a previous attempt to boot the new app failed confirmation), it will be overwritten to ESP_OTA_IMG_ABORTED.[26]
3. The bootloader then selects a new application to boot, ensuring its state is not ESP_OTA_IMG_INVALID or ESP_OTA_IMG_ABORTED. If the selected application is in the ESP_OTA_IMG_NEW state, the bootloader immediately changes this to ESP_OTA_IMG_PENDING_VERIFY. This state signifies that the application requires confirmation of its operability; if this confirmation does not occur and a subsequent reboot happens, the state will revert to ESP_OTA_IMG_ABORTED, preventing future boots of this application and triggering a rollback to the previous working application.[26]
4. Once the new application has started, it is expected to perform a self-test.[26]
5. If the self-test completes successfully, the application must call esp_ota_mark_app_valid_cancel_rollback() to confirm its operability. This marks the running application with the ESP_OTA_IMG_VALID state, removing any restrictions on its future booting.[26]
6. Conversely, if the self-test fails, the application should call esp_ota_mark_app_invalid_rollback_and_reboot(). This marks the running application with the ESP_OTA_IMG_INVALID state and triggers a reset, causing a rollback to the previous working application.[26]
7. If the application fails to confirm its operability (e.g., due to power loss or an unexpected crash during its first boot), its state remains ESP_OTA_IMG_PENDING_VERIFY. On the next boot, this state will change to ESP_OTA_IMG_ABORTED, which prevents the re-boot of this application and forces a rollback to the previous working application.[26] It is recommended to perform the self-test procedure as quickly as possible to mitigate the risk of rollback due to power loss during the critical initial boot phase.[26] Only OTA

partitions are typically subject to rollback; factory partitions are generally not rolled back.[26]

Various **OTA Application States** control the boot app selection process: ESP_OTA_IMG_VALID and ESP_OTA_IMG_UNDEFINED allow selection without restriction. ESP_OTA_IMG_INVALID and ESP_OTA_IMG_ABORTED prevent selection. ESP_OTA_IMG_NEW is selected once and transitions to ESP_OTA_IMG_PENDING_VERIFY, which requires confirmation of operability to avoid becoming ESP_OTA_IMG_ABORTED.[26]

Furthermore, **Anti-Rollback Schemes** are often implemented. New firmware versions may include increased security version numbers to prevent downgrading to older versions that might contain known vulnerabilities.[26] If a new application's security version is lower than the version stored in the chip, the new application will be erased, and the update will not be possible.[26] This robust approach to rollback and version management is a cornerstone of reliable and secure automotive software updates.

6.4. Software-Defined Vehicle (SDV) Context

The Software-Defined Vehicle (SDV) paradigm fundamentally leverages OTA updates to revolutionize the driving experience. This approach enables the continuous introduction of new functions and updates throughout the vehicle's lifecycle, ensuring it remains "fresh and new" with an evolving digital experience.[3]

The SDV concept has a profound **Impact on Vehicle Architecture**. It signifies a transition from a distributed architecture, where each function had its own ECU, to one utilizing a reduced number of powerful, centralized ECUs, known as domain controllers.[3] These domain controllers group together all functions for a specific domain, such as driving assistance systems, infotainment, connectivity, or body and chassis. Built on Systems-on-Chip (SoCs), these controllers offer high computing power and memory capacity, allowing them to host numerous functionalities and simplifying the vehicle's electrical architecture while also reducing its weight.[3]

Middleware plays a crucial role in the SDV architecture. It serves as the fundamental link that enables seamless and efficient communication between applications and the underlying SoC, ensuring the secure operation of the system as a whole.[3] This abstraction layer is vital for managing the complexity of diverse software components

interacting with powerful hardware.

The SDV model unlocks significant potential for **New Features and Personalization**. It allows for the activation of features, such as increased electric motor power for towing, or the addition of new functionalities like advanced driver assistance or infotainment features, even after the vehicle has been commissioned.[3] Artificial intelligence (AI) integration further enhances SDVs, enabling the development of more effective driver assistance systems capable of anticipating hazards, improving image recognition and computer vision for detecting pedestrians and other vehicles, and allowing vehicles to adapt their behavior to traffic conditions through machine learning. AI also facilitates predictive maintenance by analyzing large quantities of data to detect potential anomalies and failures proactively, thereby reducing breakdown risks and extending vehicle life. Intelligent voice assistants and recommendation systems further personalize the in-cabin experience, adapting temperature, lighting, and music to suit the driver's mood.[3]

Connectivity and Vehicle-to-Everything (V2X) communication are central to the SDV concept. Connectivity is essential for enabling OTA updates and V2X communication, which enhances vehicle safety by leveraging high-bandwidth, low-latency wireless connections like 5G to exchange data with other vehicles, infrastructure, and pedestrians.[4] This also facilitates driver services, such as automatic payments in restricted urban areas.[4]

Despite these advancements, **Cybersecurity Challenges** remain a key concern for SDVs. These vehicles require regular, secure software updates, robust intrusion detection systems, and encrypted communication protocols to protect against evolving threats.[3] Manufacturers must adopt more agile, modular development practices and work towards decoupling software from hardware, increasingly employing virtualization to maintain separation between functions and underlying hardware.[4] This will have the added benefit of improved performance, as manufacturers can focus on the best hardware possible without worrying about compatibility.[4]

7. Conclusion and Recommendations

The automotive industry is undergoing a profound transformation, moving from a hardware-centric model to a software-first domain. This shift is driven by the

increasing complexity of vehicle systems, the proliferation of advanced driver-assistance systems (ADAS), autonomous driving capabilities, and sophisticated connected car technologies. Over-The-Air (OTA) updates are not merely a technical capability but a strategic imperative, essential for ensuring vehicle safety, maintaining security, enhancing performance, and delivering new features throughout a vehicle's lifespan. The ability to continuously update and evolve vehicle software via OTA is fundamental to the concept of the Software-Defined Vehicle (SDV), transforming the vehicle from a static product into a dynamic, evolving platform that offers continuous value and personalized experiences to customers.

The comprehensive software update process workflow, encompassing development, validation, packaging, delivery, and in-vehicle execution, is characterized by intricate interdependencies and a critical need for robustness and security. The scale of modern automotive software demands industrialized Software Development Lifecycle (SDLC) practices, sophisticated version control, and automated Continuous Integration/Continuous Deployment (CI/CD) pipelines. Regulatory frameworks, particularly UNECE WP.29 R156, have elevated quality assurance and secure update management from best practices to legal mandates, compelling manufacturers to implement demonstrably traceable and auditable processes. The adoption of delta updates is a technical necessity for scalable and cost-effective delivery, while end-to-end security, involving multi-layered defenses and secure in-vehicle orchestration by the OTA Manager, is non-negotiable for preventing catastrophic failures and maintaining trust. Furthermore, the preference for A/B partitioning in update execution highlights the industry's commitment to seamless updates, robust rollback capabilities, and minimizing vehicle downtime, prioritizing safety and user experience above all else.

Based on this comprehensive analysis, the following recommendations are put forth for automotive manufacturers:

- **Invest in Software Engineering Maturity:** Manufacturers must prioritize significant investment in software talent acquisition, advanced tooling, and the adoption of modern software development processes, including Agile and DevOps methodologies. This transformation of organizational culture and engineering capabilities is crucial for effectively managing the immense complexity and rapid evolution of vehicle software.
- **Embrace Security by Design:** Cybersecurity must be integrated from the earliest stages of software development, not as an afterthought. This involves leveraging advanced security frameworks like Uptane, adhering to industry standards such as ISO/SAE 21434, and implementing robust secure coding

practices, threat modeling, and continuous penetration testing to ensure end-to-end protection across the entire software delivery chain.

- **Prioritize Robust Validation and Rollback Mechanisms:** Implement comprehensive testing strategies, including simulation-based testing, Hardware-in-the-Loop (HIL), and Software-in-the-Loop (SIL) testing, along with intentional failure injection. The adoption of A/B partitioning as a standard for in-vehicle update execution is critical to minimize risk, ensure seamless recovery from failed updates, and guarantee continuous vehicle functionality.
- **Ensure Continuous Regulatory Compliance:** Establish meticulous documentation and process adherence for regulations like UNECE WP.29 R156. Compliance should be treated as an ongoing, auditable effort, with dedicated teams and systems in place to ensure all requirements, from unique identifiers to comprehensive impact analyses, are consistently met.
- **Optimize Packaging and Delivery Strategies:** Leverage delta updates as the primary method for software delivery to efficiently manage network bandwidth and reduce data transfer costs. Simultaneously, ensure that all distribution channels are highly secure and authenticated to prevent unauthorized access and tampering.
- **Develop a Sophisticated OTA Manager:** Recognize the in-vehicle OTA Manager as a critical orchestrator of the entire software lifecycle. Invest in its secure, intelligent, and resilient design, enabling it to manage complex multi-ECU updates, perform on-device verification, and facilitate robust rollback procedures. Its strategic placement within the vehicle's network architecture is also paramount.
- **Foster a Service-Oriented Mindset:** Beyond technical implementation, manufacturers should embrace a strategic vision where OTA updates are leveraged to deliver continuous value, introduce new features, and enable personalized experiences. This transforms the vehicle into an evolving platform, fostering long-term customer loyalty and unlocking new revenue streams through subscription-based services.

Works cited

1. Automotive Software Updates: A Comprehensive Guide, accessed July 22, 2025, https://www.numberanalytics.com/blog/automotive-software-updates-research-methodologies
2. Automotive Software Development Lifecycle - Number Analytics, accessed July 22, 2025, https://www.numberanalytics.com/blog/automotive-software-development-lifecycle

3. Everything you need to know about the Software Defined Vehicle (SDV) - Valeo, accessed July 22, 2025, https://www.valeo.com/en/everything-you-need-to-know-about-the-software-defined-vehicle-sdv/
4. Software-Defined Vehicles: The Ultimate Guide - QNX, accessed July 22, 2025, https://blackberry.qnx.com/en/ultimate-guides/software-defined-vehicle
5. Automotive Software Updates: A Comprehensive Guide, accessed July 22, 2025, https://www.numberanalytics.com/blog/automotive-software-updates-guide
6. Mastering OTA Updates in Embedded Systems - Number Analytics, accessed July 22, 2025, https://www.numberanalytics.com/blog/mastering-ota-updates-in-embedded-systems
7. Over the Air Updates (OTA) - FreeRTOS™, accessed July 22, 2025, https://www.freertos.org/Documentation/03-Libraries/07-Modular-over-the-air-updates/01-Over-the-air-updates
8. Over-the-Air (OTA) Updates in Embedded Microcontroller Applications: Design Trade-Offs and Lessons Learned | Analog Devices, accessed July 22, 2025, https://www.analog.com/en/resources/analog-dialogue/articles/over-the-air-ota-updates-in-embedded-microcontroller-applications.html
9. What Are Over-The-Air (OTA) Updates? - Esper.io, accessed July 22, 2025, https://www.esper.io/blog/the-evolution-of-android-ota-updates
10. How to Test OTA Updates Without Bricking Devices - Memfault, accessed July 22, 2025, https://memfault.com/blog/ota-testing-101-the-ultimate-guide/
11. OTA update: definition & best practices (Technical Guide), accessed July 22, 2025, https://theembeddedkit.io/blog/ota-update-definition-best-practices/
12. The Ultimate QA Guide for Embedded Systems Development, accessed July 22, 2025, https://www.numberanalytics.com/blog/embedded-systems-qa-ultimate-guide
13. Version Control Strategies for Embedded Systems - Number Analytics, accessed July 22, 2025, https://www.numberanalytics.com/blog/version-control-strategies-embedded-systems
14. Introduction to Embedded Systems - Using Git as Version Control - Google Sites, accessed July 22, 2025, https://sites.google.com/vt.edu/introduction-to-embeddedsystem/tutorials/github-integration/using-git-as-version-control
15. Uptane | Uptane, accessed July 22, 2025, https://uptane.org/
16. Winning the automotive software development race - McKinsey, accessed July 22, 2025, https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/winning-the-automotive-software-development-race
17. Automotive Testing Techniques Guide - Number Analytics, accessed July 22, 2025, https://www.numberanalytics.com/blog/automotive-testing-techniques-guide
18. Embedded Quality Assurance(QA) and testing - Droid-technologies, accessed

July 22, 2025, https://droid-technologies.com/services/qa
19. UNECE WP.29 R155/R156: new cybersecurity regulations for vehicles, accessed July 22, 2025, https://www.appluslaboratories.com/global/en/news/publications/new-cybersecurity-regulations-vehicles-unece-wp29
20. WP.29 R156 Compliant OTA Processes Explained with Examples ..., accessed July 22, 2025, https://www.sibros.tech/post/ota-processes-compliant-with-wp-29-r156-explained-with-real-world-examples
21. Build OTA packages | Android Open Source Project, accessed July 22, 2025, https://source.android.com/docs/core/ota/tools
22. Free Download Embedded OTA Delta Update ... - Meegle, accessed July 22, 2025, https://www.meegle.com/en_us/advanced-templates/firmware_development/embedded_ota_delta_update_implementation
23. Delta Update - Embedded Artistry, accessed July 22, 2025, https://embeddedartistry.com/fieldmanual-terms/delta-update/
24. Whitepaper Author: Daniel Mckenna BU Automotive NXP ..., accessed July 22, 2025, https://www.nxp.com/docs/en/white-paper/Making-Full-Vehicle-OTA-Updates-Reality-WP.pdf
25. OTA updates - Android Open Source Project, accessed July 22, 2025, https://source.android.com/docs/core/ota
26. Over The Air Updates (OTA) - ESP32 - — ESP-IDF Programming ..., accessed July 22, 2025, https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/ota.html