

Title: Testing the Interrupted Update: A Fault-Injection Method for Verifying Recovery in Automotive Over-the-Air Software Updates

Author name and affiliation:

Manvith Dharavath

Senior Design Release Engineer

Access, Authentication, Columns & Gateway Control

General Motors

GM Global Technical Center

Abstract

Over-the-air (OTA) software updates allow vehicle software to be changed without a service visit, but an interruption during download, verification, installation, activation, or recovery can leave a controller unavailable or in an uncertain software state. UN Regulation No. 156 addresses recovery after a failed or interrupted update, while ISO 24089:2023 establishes software-update engineering processes; neither defines the specific fault-injection matrix described in this paper. This paper presents a structured method for testing OTA recovery using a hardware-in-the-loop (HIL) bench and repeatable fault injection. The method organizes test cases across update phases, fault types, and vehicle states. Faults include power loss, network interruption, storage errors, timeouts, and controlled user interruption. Each test records the starting software version, injected condition, observed system response, recovery action, final software state, and status of affected dependent functions. Pass criteria address restoration of the previous valid version or entry into a defined safe state, confirmation that no unverified version is activated, completion of rollback or recovery, and availability or reporting of dependent functions. The method is intended to produce traceable evidence that can be compared across test environments and organizations. It does not replace cybersecurity assessment, functional-safety analysis, vehicle-level validation, or fleet monitoring, and bench results may not represent every production condition.

1. Introduction and Problem

An over-the-air (OTA) software update is not a single file transfer. It is a sequence of state changes that may include package download, authenticity and integrity checks, installation, activation, restart, and post-update verification. Each transition creates a different failure condition. A network interruption during download may allow a retry, while a power loss after installation has started may require the system to select between the previous software version, a partially installed image, or a defined recovery state. If that

decision is not handled correctly, a controller may become unavailable, a vehicle function may be disabled, or the update result may be unclear to the vehicle user.

This distinction matters because a test showing that an update succeeds from beginning to end does not demonstrate that the update can recover from an interruption. Recovery behavior must be tested at the points where the system changes state. UN Regulation No. 156 establishes recovery-related expectations for failed or interrupted OTA updates, and ISO 24089 provides a broader engineering framework for software-update activities. However, neither standard gives an engineering team a ready-made test matrix that combines update phase, fault type, and vehicle state. The result can be uneven coverage: teams may test the normal update path and a few manually selected failures without knowing which interruption points remain untested.

This paper presents a structured fault-injection method for addressing that gap. The method organizes recovery tests around the update lifecycle, introduces controlled interruptions through a hardware-in-the-loop bench, and defines the evidence needed to determine whether recovery succeeded. The goal is not to prescribe one implementation, but to provide a repeatable way for manufacturers and suppliers to evaluate recovery behavior using common test logic and traceable results.

2. Requirements and Scope

Software-update recovery is both an engineering concern and a compliance concern. UN Regulation No. 156 requires an OTA-capable vehicle to restore systems to the previous version after a failed or interrupted update, or to place the vehicle in a safe state. The update must also be executed under conditions that provide enough power for completion or recovery. These requirements describe the outcome that the vehicle must achieve, but they do not prescribe one test architecture or one fault-injection sequence.

ISO 24089:2023 provides a broader framework for software-update engineering. It addresses activities at the organizational and project levels, as well as software-update infrastructure, vehicle functions, package assembly, and update campaigns. This paper uses that framework as engineering context but does not treat ISO 24089 as a substitute for vehicle-level verification.

For this method, a **failed update** is an update that cannot compete successfully because a defined error condition occurs. An **interrupted update** is an update stopped by an external event, such as power loss, network loss, storage failure, timeout, or controlled user action. **Recovery complete** means that the system has reached a defined and verifiable outcome:

the previous valid software version is restored, the new version is fully validated, or the vehicle has entered a documented safe state.

The proposed method focuses on execution and recovery behavior after an update has begun. It does not attempt to replace software-design reviews, cybersecurity assessments, compatibility analysis, functional-safety analysis, or production-fleet monitoring. It also does not prescribe a specific bootloader, partitioning strategy, communication protocol, cloud architecture, or vehicle platform.

The method can be applied to a representative electronic control unit, a subsystem test environment, or a broader hardware-in-the-loop setup. Dependent functions, such as phone-based digital-key availability, should be included only when they are within the defined system boundary and can be evaluated without using confidential implementation details.

The central testing question is therefore:

When an OTA update is interrupted at a defined phase, does the system produce a known, valid, and traceable outcome?

Each test should map that question to a defined requirement, an injected fault, an expected recovery behavior, and objective evidence.

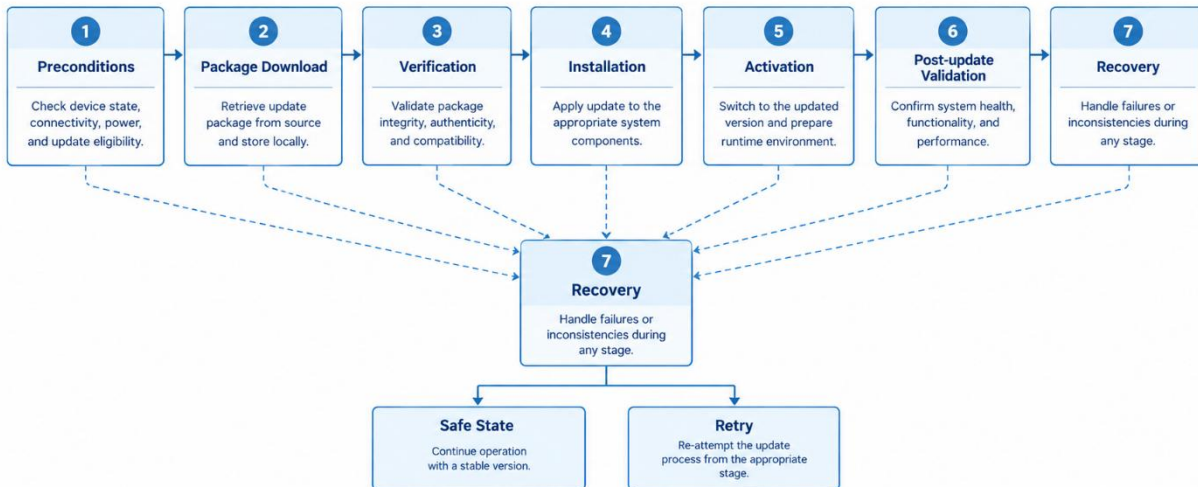
3. OTA Update Lifecycle and Test Phases

The test method begins by dividing the OTA process into logical phases. These phases may be combined or named differently across implementations, but the test objective remains the same: identify where an interruption occurs and verify the resulting system state.

Figure 1. OTA Update Lifecycle:

Logical lifecycle of an OTA software update, showing preconditions, package download, verification, installation, activation, post-update validation, and recovery. Recovery may be triggered by an interruption during any execution phase.

OTA Update Lifecycle



3.1 Preconditions

Before the update begins, the test establishes the initial conditions. These may include the target software version, vehicle configuration, available storage, communication status, power condition, and vehicle operating state. The test record should identify which conditions are required for the update to proceed and which conditions should prevent execution.

A precondition test may intentionally begin with insufficient storage, inadequate power, an incompatible configuration, or an unavailable communication path. These cases are different from interruptions because the update has not yet entered the execution phase.

3.2 Download

During download, the update package is transferred to the vehicle or its supporting system. Relevant interruptions include network loss, packet delay, incomplete transfer, and loss of vehicle power.

The test should determine whether the system detects the interruption, preserves the existing valid software, records the failure, and supports a safe retry or cancellation. A partially received package should not be treated as an installable update.

3.3 Verification

Verification may include package identity, integrity, authenticity, compatibility, and available resources. Faults at this phase can include a corrupted package, an invalid signature, an unexpected package version, or a mismatch between the package and vehicle configuration.

The expected result is that the system rejects the package or prevents activation without disturbing the currently valid software version.

3.4 Installation

Installation changes software stored on the target controller. This is a high-risk interruption point because power loss, storage failure, or an unexpected restart may occur after the original software has been partly modified.

The test should record whether the controller can still boot, whether a recovery image or previous version remains available, and whether the installation status is reported accurately.

3.5 Activation and restart

Activation determines which software image the controller will use after restart. Tests should include interruption during restart, activation failure, repeated restart, and an image that fails its initial health check.

The expected outcome should be defined before testing. Depending on the design, the controller may continue with the previous valid image, retry activation, or enter a controlled recovery state.

3.6 Post-update validation

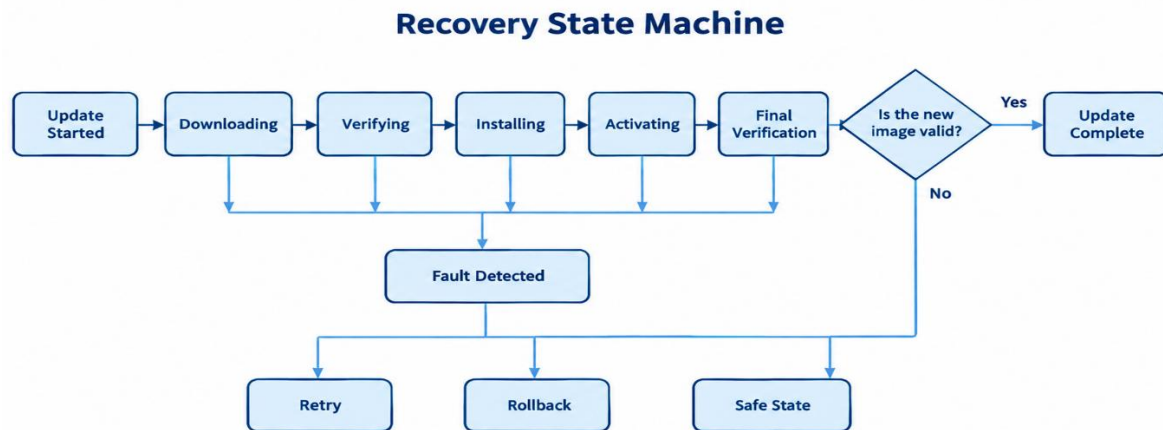
After activation, the test verifies the software version, update result, diagnostic status, and functions within scope. A successful reboot alone is not enough. The system must confirm that the intended version is active and that required dependent functions are available or clearly reported as unavailable.

3.7 Recovery and rollback

Recovery is treated as an exception path rather than a separate normal update phase. It may begin after a failure in any earlier phase. The test records whether recovery restores the previous valid version, completes installation of the new version, or places the system in a defined safe state.

The recovery result must be observable, repeatable, and traceable to the original interruption.

Figure 4. Recovery State Machine: Representative recovery state machine showing the transition from normal update execution to fault detection, retry, rollback, safe-state entry, or successful completion.



4. Interruption Matrix

The interruption matrix is the core of the proposed method. It combines three dimensions:

1. **Update phase** — where the interruption occurs.
2. **Fault type** — what causes the interruption.
3. **Vehicle state** — the operating condition when the fault occurs.

Testing only one dimension can leave important gaps. For example, a power-loss test during download does not represent a power-loss test after installation has begun. The same fault may produce different results depending on the phase and vehicle state.

Figure 2. Interruption Matrix:

Test matrix combining OTA update phases, fault types, and vehicle states. Each selected combination represents a controlled test case with defined expected behavior and evidence requirements.

Interruption Matrix

OTA Phase	Fault Type				
	Network interruption	Power interruption	Invalid package	Communication timeout	Reset or restart
Download	    	    	    	    	    
Verification	    	    	    	    	    
Installation	    	    	    	    	    
Activation	    	    	    	    	    
Validation	    	    	    	    	    

Vehicle State Legend
 Parked
  Ignition on
  Ignition off
  Charging
  Service mode

4.1 Update phases

The phase dimension includes preconditions, download, verification, installation, activation, post-update validation, and recovery. Recovery is normally observed as the result of an interruption rather than treated as an independent injection point.

Each phase should have a defined entry condition and exit condition. For example, an installation test should begin only after the package has passed the required verification checks. This allows the test result to be attributed to the intended phase rather than to an earlier setup error.

4.2 Fault types

The initial matrix should include one controlled fault at a time:

- **Power loss:** remove or reduce available power at a defined point.
- **Network loss:** interrupt communication between the update service and vehicle.
- **Storage error:** simulate an unavailable, full, or failed storage location.
- **Timeout:** delay a required response beyond the defined limit.
- **User action:** cancel, postpone, restart, or otherwise interrupt the operation where permitted.

- **Unexpected restart:** restart the relevant controller or subsystem without completing the current step.

Single-fault testing makes failure attribution easier. Multiple simultaneous faults can be added later when the single-fault behavior is understood, and the test objective requires the combination.

4.3 Vehicle states

The vehicle-state dimension ensures that the same fault is not tested under only one operating condition. Candidate states include parked and inactive, vehicle wake-up, charging, update preparation, controller restart, and post-recovery operation. A phone-based digital-key state may be included when that function is within the defined system boundary.

The state should be established before the update begins and recorded in the test case. If a state cannot be reproduced reliably, it should not be used as a controlled test condition until the setup is improved.

4.4 Test identification

Each test case should receive an identifier that shows its principal dimensions. For example:

INST-PWR-PARKED-001

This identifier represents an interruption during installation caused by power loss while the vehicle is parked. The test record should also include the software version, hardware configuration, injected fault timing, expected response, observed response, recovery outcome, and supporting log references.

The matrix does not require every possible combination to be tested immediately. Teams can prioritize combinations based on safety relevance, update architecture, historical failures, and the likelihood that a fault will occur in service. However, every omitted combination should have a documented rationale rather than being left unexamined.

A matrix entry is complete only when the interruption was introduced at the intended point, the system response was recorded, recovery reached a defined outcome, and the evidence supports the pass or fail decision.

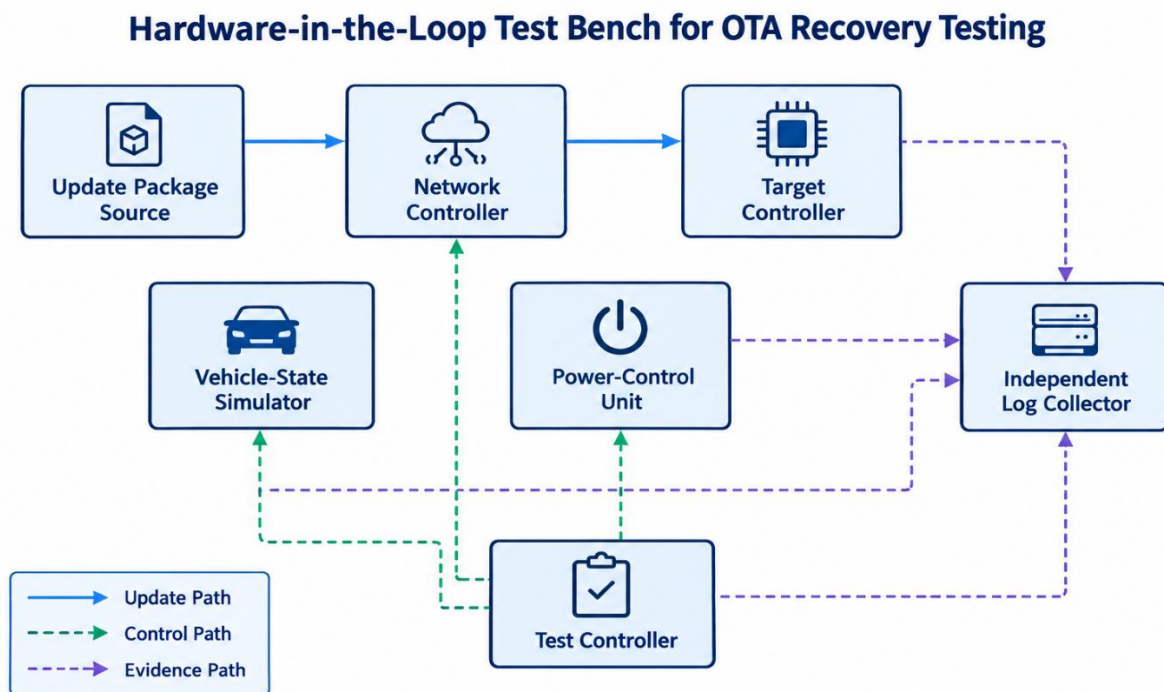
5. Hardware-in-the-Loop Test Bench

The proposed method uses a hardware-in-the-loop (HIL) bench to reproduce update conditions and introduce controlled interruptions. The bench does not need to replicate

every component of a complete vehicle. It needs to reproduce the update states, communication paths, power conditions, and recovery behavior relevant to the test objective.

Figure 3. Hardware-in-the-Loop Test Bench:

Generic hardware-in-the-loop arrangement for OTA recovery testing. The bench controls the update source, communication path, power state, and vehicle-state simulation while independently collecting evidence from the target controller and supporting systems.



The bench should include an update-package source or simulator, a communication path that can be interrupted at controlled times, a representative target controller, and a programmable power or reset control. It should also provide a way to simulate the vehicle state required by the update, such as parked, waking, charging, or recovering after a restart.

The test controller should be able to start the update, identify the intended injection point, introduce one fault, and record the resulting behavior. Fault injection should be controlled independently from the system under test. This separation helps prevent the test mechanism from changing the result or hiding the failure being measured.

Evidence collection should include both system-generated records and independent observations. System records may include update status, diagnostic events, software-version identifiers, rollback status, and user notifications. Independent observations may include power state, network state, reset timing, controller availability, and the time at which recovery begins and ends.

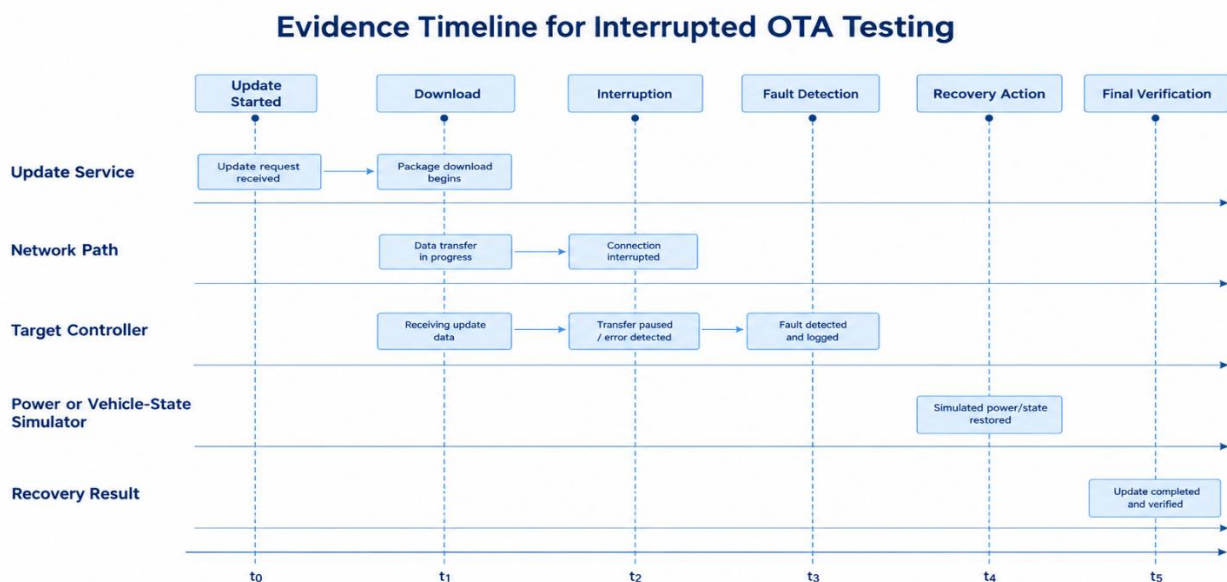
Time synchronization is important because an interruption may appear in several logs. The test record should allow events from the update service, communication controller, target controller, and bench equipment to be placed on one timeline. If exact synchronization is not possible, the timing relationship and known measurement limits should be documented.

The bench should support repeatable reset and restoration procedures. After each test, the target should be returned to a defined baseline, and the test operator should confirm that the baseline software version and configuration are correct. A test should not be repeated from an unknown state because the previous failure may influence the next result.

The method is intentionally vendor-neutral. It describes the functions needed to control, observe, and repeat the test without requiring a particular update protocol, bootloader design, cloud platform, or diagnostic tool.

Figure 5. Evidence Timeline

Example evidence timeline aligning update-service events, communication interruptions, controller responses, power changes, recovery actions, and final software-state verification.



6. Test Execution Procedure

Each test begins with a written test objective and an expected recovery outcome. The objective should identify the update phase, fault type, vehicle state, and function being evaluated. The expected outcome should be defined before the test begins so that the result is not judged after the fact.

First, the test operator establishes the baseline. The target controller must contain the approved starting software version, and the hardware and configuration must match the test record. The operator verifies that the controller can boot, communicate, and perform the functions required for the test. Any existing diagnostic events or abnormal conditions should be recorded before the update starts.

Next, the operator establishes the vehicle state. This may include a parked state, a wake-up sequence, a charging condition, or a post-restart recovery condition. The state should be observable through the bench or test equipment and should remain stable until the intended fault is introduced.

The test controller then starts the update and monitors the defined phase transition. Fault injection should occur at a repeatable point rather than at an approximate time. For example, a power interruption during installation should be triggered after the installation phase is confirmed and before activation begins. If the exact transition cannot be observed, the test should document the measurement method and its timing uncertainty.

The test introduces one primary fault and records the immediate response. The records should show whether the update stopped, reported an error, retried, continued, or entered a recovery path. The operator should not manually intervene unless the test case specifically includes a user action.

After the fault is introduced, the system is allowed to complete its defined recovery behavior. The operator then verifies the final software version, boot status, update status, diagnostic state, and any dependent functions included in the test boundary. Recovery should not be judged only by whether the controller becomes responsive. The test must also confirm which software version is active and whether the update system reports the result correctly.

The final step is restoration of the baseline. The target controller and bench are returned to the defined starting condition, and the operator confirms that the next test will not be influenced by the previous failure. If the baseline cannot be restored, the test is marked incomplete and the reason is recorded.

A complete test record therefore contains the intended injection point, the actual injection point, the observed response, the recovery path, the final state, and the evidence supporting the result.

7. Pass/Fail Criteria and Evidence

Pass/fail criteria should be defined before testing begins. The criteria should evaluate both the technical recovery result and the evidence needed to support that result.

A test passes only when the fault occurs in the intended phase and the system reaches the expected outcome. Depending on the test case, the acceptable outcome may be successful completion of the new software version, restoration of the previous valid version, or entry into a defined safe state.

The following conditions should be evaluated:

- The injected fault is confirmed and recorded.
- The update system detects or handles interruption.
- No unverified or incomplete software image becomes active.
- The controller restores the previous valid version or reaches the defined safe state.
- Rollback or recovery completes without an undefined intermediate condition.
- The update result is reported accurately.
- Dependent functions within the test boundary are available or clearly identified as unavailable.
- The logs and test records are sufficient to reproduce the pass/fail decision.

A test should fail if the controller cannot boot, activates an unverified image, remains in an undefined state, loses required recovery information, or reports success when the intended software version is not active. A test may also be marked incomplete when recovery appears successful but the evidence cannot prove which version was running or when the interruption occurred.

The evidence package should include the test identifier, hardware and software configuration, starting software version, vehicle state, injected fault, expected result, actual result, event timestamps, diagnostic records, final software version, recovery status, and pass/fail rationale. Logs from different systems should be placed on a common timeline whenever possible.

The method separates a technical failure from an evidence failure. For example, a controller may eventually return to service, but the test should not receive a passing result if the team cannot determine whether the previous version or the new version is active. Recovery must be both successful and demonstrable.

This evidence structure also allows results to be compared across test environments. Different organizations may use different tools or update architectures, but they can still report the same core facts: where the interruption occurred, what fault was introduced, what state the system reached, and how that conclusion was verified.

8. Generic Worked Example: Power Loss During Installation

Consider a representative telematics controller with a valid software version, identified here as **Version A**. A new package, **Version B**, has been downloaded and verified. The vehicle is parked, the required preconditions are satisfied, and the controller begins installation.

At a defined point during installation, the test bench removes power from the controller. The interruption is recorded with the test identifier, update phase, and timestamp. The purpose of the test is not to determine whether the controller can continue writing after power is removed. The purpose is to determine whether the controller can recover without activating an incomplete software image.

The expected behavior is defined before the test begins. The controller should detect the interrupted installation, preserve or access its recovery mechanism, and avoid treating Version B as valid unless installation and validation are complete. Depending on the implementation, the controller may restore Version A, retry the installation, or enter a defined safe state.

When power is restored, the test verifies the controller's boot behavior and active software version. The test also checks whether the update system records the interruption and communicates the correct result. A success message should not be reported if Version B was not completely installed and validated. If the controller supports dependent functions, those functions are checked only after the primary software state has been confirmed.

The evidence package includes the initial software version, package status before interruption, power-interruption timestamp, controller response, diagnostic records, recovery action, final software version, and user or system notification. The test record also identifies whether the controller returned to Version A, completed Version B, or entered the defined safe state.

The test passes if the controller reaches one of the approved outcomes and the evidence proves which outcome occurred. It fails if the controller cannot boot, activates incomplete software, remains in an undefined state, or reports successful installation without supporting evidence.

This example is illustrative. It does not represent a production defect or demonstrate the performance of a particular vehicle or supplier system. Its purpose is to show how one interruption can be translated into a repeatable test case.

9. Limitations and Open Questions

The proposed method has several limitations. A hardware-in-the-loop bench can reproduce selected power, communication, storage, and timing conditions, but it cannot represent every condition that may occur in a production vehicle. Temperature, electromagnetic effects, battery behavior, physical wiring faults, network congestion, and interactions among multiple controllers may require additional testing.

The method also depends on how the update architecture defines a safe state. A safe state for one controller or vehicle function may not be appropriate for another. Each implementation must therefore define its acceptable recovery outcomes before testing begins.

The matrix can grow quickly as additional software versions, vehicle configurations, fault types, and operating states are added. Full combination testing may not be practical. Teams must prioritize cases using safety relevance, system architecture, historical failures, and expected field exposure. Any excluded combination should have a documented rationale.

Cloud services, backend campaign logic, mobile applications, and phone-based digital keys may require separate validation. They should not be treated as verified merely because the vehicle controller recovered successfully.

Finally, this method does not replace cybersecurity testing, functional-safety analysis, compatibility assessment, or fleet monitoring. It provides a structured way to generate recovery evidence under controlled conditions. Further work could examine how recovery results from benches can be connected to vehicle-level validation and post-deployment monitoring.

10. Conclusion and Takeaways

OTA recovery should be tested as deliberately as the successful update path. An update can fail at several different phases, and the same fault may produce different results depending on when it occurs and what the vehicle is doing.

The method presented here organizes testing around three dimensions: update phase, fault type, and vehicle state. It uses controlled fault injection, a repeatable test bench, defined recovery outcomes, and evidence that connects the interruption to the final software state.

The method is intended to help manufacturers and suppliers discuss recovery using common test logic without requiring a specific platform or implementation. Its value depends on clear system boundaries, realistic pass criteria, and disciplined evidence collection.

Key takeaways:

- Test interruptions at defined update-phase boundaries.
- Cross fault types with relevant vehicle states.
- Define recovery outcomes before executing the test.
- Verify the active software version, not only controller availability.
- Treat incomplete evidence as a test issue, even when the system appears to recover.

12. Author Bio

Manvith Dharavath is a Senior Software Design and Release Engineer specializing in over-the-air software update testing and connected-vehicle diagnostics. His work focuses on recovery validation and release methods for connected vehicles. He applies system-level testing and evidence-based analysis to improve software update reliability and delivery across vehicle systems. The views expressed are his own.

Reference Links

1. UNECE. UN Regulation No. 156: Software Updates and Software Update Management System.
[Official UNECE source](#)
2. ISO 24089:2023. Road Vehicles—Software Update Engineering.
[Official ISO source](#)

3. NHTSA. Cybersecurity Best Practices for the Safety of Modern Vehicles, 2022.
[Official NHTSA PDF](#)
4. Uptane Alliance. Uptane Standard for Design and Implementation, Version 2.1.0.
[Official Uptane Standard](#)
5. ISO 26262-4:2018. Road Vehicles—Functional Safety—Product Development at the System Level.
[Official ISO source](#)
6. ISO 26262-6:2018. Road Vehicles—Functional Safety—Product Development at the Software Level.
[Official ISO source](#)
7. ASAM XIL 3.0.0. Generic Simulator Interface.
[Official ASAM source](#)
8. Himmeler, A., Lamberg, K., and Beine, M. “Hardware-in-the-Loop Testing in the Context of ISO 26262.” SAE Technical Paper 2012-01-0035.
[SAE source](#)
9. Halder, S., Ghosal, A., and Conti, M. “Secure Over-the-Air Software Updates in Connected Vehicles: A Survey.”
[Open-access paper](#)
10. Bauersachs et al. “Hardware-in-the-Loop-Based Real-Time Fault Injection Framework for Dynamic Behavior Analysis of Automotive Software Systems.” Sensors, 2022.
[Open-access paper](#)

AI Use Disclosure: Generative AI tools were used for drafting support, language editing, organizational assistance, and initial visualization drafts. The author independently reviewed, rewrote, fact-checked, and verified all text, references, technical claims, data, and figures. No AI-generated data or unverified sources were used. The author is solely responsible for the final manuscript.