International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V5I4P125
Eureka Vision Publication | Volume 5, Issue 4, 221-226, 2024

Original Article

Risk-Adaptive Cloud-to-Edge Application and Data Update Architecture for Android-Based Embedded Systems Using Lightweight Edge AI Validation

Srikanth Puram
Independent Researcher, Mobile and Embedded Software Architecture Novi, Michigan, USA.

Abstract - Android-based embedded systems increasingly receive application packages, configuration data, policy files, machine-learning models, and security metadata from cloud services. These cloud-to-edge updates improve product maintainability, but they also expand the attack surface: a device can receive a stale manifest, a mismatched split package, a corrupted artifact, an unsafe policy revision, or a model update that is technically valid but operationally risky for the current device state. This paper proposes a risk-adaptive cloud-to-edge application and data update architecture for Android-based embedded systems using lightweight edge AI validation. The architecture combines signed update manifests, artifact-level integrity checks, application data-security controls, persisted update-state management, package compatibility verification, rollback-safe orchestration, and a quantized on-device risk classifier. The edge AI component does not replace deterministic security gates; instead, it prioritizes, defers, quarantines, or escalates updates based on contextual risk features such as signature status, manifest freshness, package version drift, retry history, rollback frequency, network trust, power state, storage pressure, and prior installation failures. The proposed architecture is designed for industry environments where Android applications run on embedded, automotive-style, kiosk, retail, industrial, or managed-device platforms and where cloud-delivered updates must be secure, observable, recoverable, and operationally safe. A reproducible evaluation framework is provided using metrics for false acceptance, false rejection, validation latency, memory overhead, rollback detection time, manifest replay resistance, and update completion reliability.

Keywords - Android Embedded Systems, Cloud-to-Edge Security, Secure Software Update, Edge AI, LiteRT, TensorFlow Lite, Application Data Security, Package Validation, Risk Scoring, Rollback Safety.

1. Introduction

Cloud-connected embedded systems have changed the way software is delivered and maintained. Modern Android-based embedded platforms may receive feature flags, application updates, configuration policies, security rules, model files, UI resources, and telemetry instructions from remote services. This pattern is useful because it reduces service cost and enables continuous improvement after deployment. However, it also turns the update workflow into a high-value security boundary.

Traditional software update clients often rely on deterministic checks such as transport security, package signature validation, version comparison, and hash verification. These checks remain essential, but they are not sufficient for all operational risks. A package may be correctly signed but unsuitable for the current device profile. A manifest may be valid but stale. A device may have enough storage at the start of the workflow but become constrained during staging. A history of repeated install failures may indicate that the next attempt should be deferred or escalated instead of retried automatically.

This paper argues that secure cloud-to-edge update architecture should combine deterministic security gates with contextual risk classification at the edge. A small on-device AI model can analyze device-local risk signals without uploading sensitive data. The model can help decide whether to install now, defer until safer conditions, quarantine an artifact, request cloud revalidation, or trigger rollback-safe cleanup. The approach is particularly relevant for Android-based embedded systems where connectivity, power state, storage, package compatibility, and lifecycle interruptions vary across device classes.

2. Contributions and Scope

The paper makes four practical contributions. First, it defines a cloud-to-edge update threat model for Android-based embedded systems that includes both application packages and update-associated data artifacts. Second, it proposes a layered architecture that combines cloud policy services, signed manifests, device-side validators, package orchestration, and an edge AI risk classifier. Third, it defines a recovery-aware update state machine and deterministic override rules so that AI does not weaken security. Fourth, it provides a reproducible evaluation framework with measurable metrics suitable for industry validation and peer review.

The scope is intentionally application-layer and platform-adjacent. The paper does not replace Android platform package verification, TLS, or secure boot. Instead, it addresses the orchestration layer that decides when and how application and data updates should be accepted, staged, installed, retried, deferred, or rolled back.

3. Threat Model

The proposed architecture assumes an attacker may observe network traffic, attempt replay of older manifests, tamper with non-protected artifacts, trigger repeated update retries, exploit storage pressure, or attempt to influence device telemetry used by a risk model. The attacker is not assumed to break modern cryptographic primitives or possess the private signing key for legitimate package artifacts. The architecture also assumes that operational faults such as network loss, reboot, suspend, and interrupted installation can occur without malicious intent.

Table 1. Threat Model and Security Controls for Cloud-To-Edge Update Workflows

Threat ID	Threat	Risk to update workflow	Primary control
T1	Manifest replay	Device accepts an older but valid-looking update contract.	Manifest timestamp, monotonic version, nonce or server-issued epoch, policy freshness check.
T2	Artifact tampering	Package, split, model, or policy data is altered in transit or storage.	Hash validation, package signature verification, TLS, immutable artifact identity.
T3	Split/package mismatch	Base package and split artifacts do not form a compatible set.	Package-name, version-code, signing, ABI, density, and feature-module validation.
T4	Unsafe operational timing	Update begins during poor power, network, or storage state.	Constraint evaluation and edge risk scoring before staging.
T5	Repeated retry storm	Device repeatedly downloads or installs failing artifacts.	Retry classification, backoff, risk escalation, quarantine.
T6	Model update abuse	Edge AI model or thresholds are replaced with unsafe versions.	Model signing, model manifest, version pinning, validation before activation.
T7	Telemetry leakage	Update diagnostics expose sensitive device or user data.	Local feature extraction, minimization, hashing, redaction, aggregated reporting.
T8	Rollback ambiguity	Device cannot determine whether prior version remains safe.	Post-install verification, known-good version tracking, cleanup state machine.

4. System Architecture

The architecture is organized into cloud, edge, and Android client layers. The cloud layer publishes signed manifests, policy constraints, expected artifact hashes, rollout targeting rules, and approved risk-threshold profiles. The edge layer, which may run on the device or on a local gateway, performs lightweight contextual inference using a quantized risk model. The Android client layer performs deterministic validation, package staging, state persistence, installation orchestration, rollback-safe cleanup, and diagnostic reporting. A key design decision is that AI is advisory and bounded. Deterministic security checks have absolute priority. If a signature, package identity, artifact hash, or manifest freshness check fails, the update is rejected regardless of the AI score. The AI classifier is used only after hard security checks pass, and its output affects operational timing and escalation rather than cryptographic trust.

Table 2. Layered Cloud-To-Edge Update Architecture

Layer	Component	Responsibility
Cloud	Manifest and policy service	Produces signed update contract, version constraints, rollout group, expiry, risk threshold, and artifact metadata.
Cloud	Artifact repository	Stores application packages, configuration data, AI model files, and validation hashes.
Edge/device	Risk feature extractor	Creates a feature vector from local runtime, update history, package metadata, and diagnostic state.
Edge/device	Lightweight AI validator	Runs quantized classification or regression model to estimate operational update risk.
Android client	Deterministic validator	Checks signature, package identity, version, hash, base/split compatibility, and manifest freshness.
Android client	Stateful orchestrator	Stages artifacts, persists checkpoints, manages install session, verifies result, and triggers cleanup or rollback.
Android client	Observability module	Reports non-sensitive status, failure class, risk score bucket, and recovery action.

5. Edge AI Validation Model

The edge AI validator is designed for low overhead rather than deep semantic reasoning. Suitable model families include quantized gradient-boosted decision trees, compact multilayer perceptrons, or logistic regression models converted into an on-device runtime. For Android embedded devices, LiteRT or TensorFlow Lite style deployment is appropriate because the runtime supports optimized on-device inference, quantization, and accelerator-aware execution while keeping inference local.

The feature vector intentionally avoids sensitive content. It uses operational and security-state features such as manifest age, artifact hash result, package version drift, retry count, rollback count, storage headroom, power-state bucket, network trust class, past install error class, and package-set complexity. This allows risk scoring without uploading raw user data or proprietary payload contents.

The model output is a bounded risk score R in the interval [0,1], a confidence bucket C, and a reason vector indicating dominant factors. The policy engine maps this output to one of five actions: install, defer, retry later, quarantine, or require cloud revalidation.

Table 3. Edge AI Feature Groups for Update-Risk Classification

Feature group	Example features	Rationale
Artifact security	hash_match, signature_valid, manifest_epoch_valid	Hard gates and model features help identify non-installable states.
Package compatibility	base_version_gap, split_count, abi_match, density_match	Complex package sets have higher mismatch risk.
Operational condition	storage_headroom_mb, power_bucket, network_class, device_idle_state	Install safety depends on current runtime constraints.
History	retry_count_24h, rollback_count_30d, last_error_class	Repeated failures indicate degraded device or artifact compatibility.
Freshness	manifest_age_hours, policy_age_hours, model_age_days	Old contracts should be revalidated before activation.
Observability	last_checkpoint_state, session_reconcile_status	Uncertain state should bias toward validation or cleanup.

6. Risk Decision Model

The risk score is used after deterministic checks pass. A simple decision rule is shown below, where H represents hard validation results, R represents model risk score, C represents model confidence, and P represents policy constraints.

Decision = Reject if H fails; otherwise Quarantine if R >= tau_q; Defer if tau_d <= R < tau_q; Install if R < tau_d and P is satisfied; otherwise request revalidation. This form intentionally keeps the AI model bounded by deterministic policy. The model cannot override package signature failure, manifest replay detection, or an unsupported device profile.

Algorithm 1. Risk-adaptive update validation and action selection

```
Input: manifest M, artifact set A, device state D, history H, policy P, model f_theta
Output: action in {INSTALL, DEFER, RETRY, QUARANTINE, REVALIDATE}
1: if not verifyManifestSignature(M) then return QUARANTINE
2: if isManifestExpiredOrReplayed(M, D) then return REVALIDATE
3: if not verifyArtifactHashes(A, M) then return QUARANTINE
4: if not verifyPackageCompatibility(A, D) then return QUARANTINE
5: if not deterministicPolicyAllows(D, P) then return DEFER
6: x <- buildRiskFeatureVector(M, A, D, H)
7: (risk, confidence, reasons) <- f_theta(x)
8: if confidence = LOW then return REVALIDATE
9: if risk >= tau_quarantine then return QUARANTINE
10: if risk >= tau_defer then return DEFER
11: return INSTALL
```

7. Stateful Update Orchestration

Update reliability depends on persisted workflow state. The orchestrator should not rely on memory-only assumptions because embedded devices may reboot, suspend, terminate the application process, or lose network connectivity mid-workflow. Each stage must have an explicit checkpoint and a revalidation rule. The proposed state machine is: Candidate -> ManifestVerified -> ArtifactsDownloaded -> ArtifactsValidated -> RiskScored -> PolicyAccepted -> Staged -> Committed -> PostInstallVerified -> Completed. Error paths lead to Deferred, Quarantined, FailedRecoverable, FailedNonRecoverable, CleanupRequired, or RollbackInitiated states.

Table 4. Recovery Checkpoints for Update Orchestration

State	Persisted checkpoint	Resume action
ManifestVerified	manifest ID, version, expiry, signature result	Check freshness and replay status again.
ArtifactsDownloaded	artifact IDs, byte sizes, local paths	Recompute hashes before reuse.
ArtifactsValidated	hash result, package metadata, split map	Revalidate compatibility if device profile changed.
RiskScored	risk score bucket, confidence, reason codes	Re-score if runtime condition has materially changed.
Staged	install session ID and package set	Reconcile session and recreate if abandoned.
Committed	commit timestamp and expected version	Verify installed package state before reporting success.
CleanupRequired	temporary artifacts and session identifiers	Remove stale artifacts before next attempt.

8. Application and Data Security Controls

Cloud-to-edge update workflows often deliver more than executable packages. They may deliver configuration files, feature flags, resource bundles, policy data, and small ML models. Each artifact class should be versioned, integrity-protected, and bound to a manifest. The system should avoid accepting free-floating files that are not referenced by a signed update contract.

Data security controls include transport encryption, manifest signing, artifact hashing, package signature verification, anti-rollback policy, local encryption for sensitive cached data, and telemetry minimization. Model files used by the edge AI validator must also be signed and versioned. A risk model update should follow the same validation rules as other application data artifacts.

Privacy is improved by performing risk scoring locally. The cloud does not need raw device logs, user data, file contents, or application payloads. It only receives bounded event categories such as success, deferred, quarantine, rollback initiated, and non-sensitive error class.

9. Evaluation Framework and Metrics

A publishable evaluation should use fault injection, replay tests, package-mismatch tests, network-transition tests, storage-pressure tests, and model-threshold tests. The purpose is to measure whether the architecture reduces unsafe acceptance while preserving practical update completion.

The framework should be evaluated across at least three device classes: a low-memory Android embedded device, a mid-range Android device, and an Android Automotive or automotive-style embedded target. The same update corpus should include valid packages, corrupted artifacts, expired manifests, replayed manifests, mismatched splits, unsupported ABI packages, and transient failures.

Table 5. Evaluation metrics for secure risk-adaptive cloud-to-edge updates

Metric	Definition	Why it matters
False Acceptance Rate (FAR)	Invalid or unsafe update accepted / total invalid update cases	Primary safety metric; lower is better.
False Rejection Rate (FRR)	Valid update rejected or quarantined / total valid update cases	Measures excessive conservatism.
p95 validation latency	95th percentile time for manifest, artifact, and risk validation	Controls update overhead.
Peak RAM overhead	Maximum additional memory during validation and model inference	Critical for embedded devices.
Rollback detection time	Time from failed post-install verification to rollback or cleanup action	Measures recovery responsiveness.
Replay rejection rate	Replayed manifests rejected / replay test cases	Measures anti-replay effectiveness.
Update completion reliability	Successful valid updates / attempted valid updates under interruptions	Measures practical reliability.
Telemetry minimization ratio	Raw diagnostic fields retained after minimization / raw collected fields	Indicates privacy discipline.

10. Benchmark Protocol

The benchmark should include three phases. Phase 1 validates deterministic controls using corrupted, mismatched, and replayed artifacts. Phase 2 evaluates operational risk classification using device-state variations such as low storage, poor network, repeated failures, and unsafe power conditions. Phase 3 evaluates recovery by injecting interruption during download, validation, staging, commit, and post-install verification.

For each test case, the report should include the selected action, true validity label, deterministic validation result, AI risk bucket, device state, final workflow outcome, latency, memory overhead, and recovery action. This creates a reproducible evidence trail for reviewers and future implementers.

Table 6. Fault-Injection Benchmark Design

Test group	Fault injected	Expected safe outcome
Manifest security	expired, replayed, unsigned, wrong epoch	Reject or revalidate before download.
Artifact security	hash mismatch, altered file, unsigned package	Quarantine and clean up.
Package compatibility	wrong ABI, missing base, mismatched split	Reject before install session.
Runtime condition	low storage, low power, untrusted network	Defer or wait for policy-compliant conditions.
Recovery	process death, reboot, network loss during staging	Resume only from validated checkpoint.
Model safety	low-confidence risk output or model version mismatch	Use deterministic fallback and request cloud revalidation.

11. Discussion

The proposed architecture is not intended to make AI the trust anchor for software updates. That would be unsafe. Instead, it uses AI to rank operational risk after cryptographic and compatibility checks have passed. This makes the approach suitable for industry systems where deterministic rules are necessary but not always sufficient to make timing, retry, and escalation decisions.

A lightweight model can also improve privacy by keeping raw operational signals on the device. However, model governance is important. The risk model must be versioned, signed, audited, and evaluated for false acceptance and false rejection. The safest implementation is one where deterministic policy gates remain authoritative and AI is used for prioritization, deferral, and risk visibility. The architecture is especially useful when update workflows span application packages, data files, configuration policies, resource bundles, and edge AI models. As embedded Android systems become more connected and software-defined, the line between application update, data update, and model update becomes increasingly operationally important.

12. Limitations and Future Work

This paper provides a technical architecture and reproducible evaluation framework. A production deployment should further validate the model using measured device data, adversarial test cases, and long-running field telemetry. Future work may compare tree-based and neural risk classifiers, evaluate on Android Automotive reference hardware, publish an open-source benchmark harness, and study safe online threshold adaptation without exposing sensitive telemetry.

13. Conclusion

This paper proposed a risk-adaptive cloud-to-edge application and data update architecture for Android-based embedded systems. The architecture integrates deterministic security validation, package compatibility checks, secure data artifact handling, stateful update orchestration, rollback-safe recovery, and lightweight edge AI validation. By keeping hard security gates deterministic and using AI for contextual risk scoring, the design improves operational safety without weakening cryptographic trust. The proposed benchmark framework provides measurable criteria for validation latency, memory overhead, false acceptance, false rejection, rollback detection, replay resistance, and update reliability.

References

[1] M. Souppaya, K. Scarfone, and D. Dodson, “Secure Software Development Framework (SSDF) Version 1.1,” NIST Special Publication 800-218, Feb. 2022.
[2] National Institute of Standards and Technology, “The NIST Cybersecurity Framework (CSF) 2.0,” NIST CSWP 29, Feb. 2024.
[3] National Institute of Standards and Technology, “Artificial Intelligence Risk Management Framework (AI RMF 1.0),” NIST AI 100-1, Jan. 2023.
[4] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, “Zero Trust Architecture,” NIST Special Publication 800-207, Aug. 2020.
[5] OWASP Foundation, “Mobile Application Security Verification Standard,” 2024.
[6] J. Samuel et al., “The Update Framework: A Framework for Securing Software Update Systems,” USENIX Security, 2010.
[7] T. K. Kuppusamy, L. A. DeLong, and J. Cappos, “Uptane: Securing Software Updates for Automobiles,” ESCAR USA, 2016.
[8] T. K. Kuppusamy, L. A. DeLong, and J. Cappos, “Securing Software Updates for Automotives Using Uptane,” USENIX ;login:, vol. 42, no. 2, 2017.

[9] Google, “TensorFlow Lite is now LiteRT,” Google Developers Blog, Sep. 2024.
[10] Google, “LiteRT: High-Performance On-Device Machine Learning,” Google AI Edge Documentation, 2024.
[11] R. David et al., “TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems,” arXiv:2010.08678, 2020.
[12] C. Banbury et al., “MLPerf Tiny Benchmark,” arXiv:2106.07597, 2021.
[13] Android Developers, “PackageInstaller,” Android Developer Documentation, 2024.
[14] Android Open Source Project, “A/B System Updates,” AOSP Documentation, 2024.
[15] Android Open Source Project, “Dynamic Partitions,” AOSP Documentation, 2024.
[16] E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3,” RFC 8446, Aug. 2018.
[17] ISO, “ISO 26262: Road Vehicles - Functional Safety,” International Organization for Standardization, 2018.
[18] P. Koopman, Better Embedded System Software. Drumnadrochit Education, 2010.