Automated Phased Hybrid PQC-TLS Migration via DevSecOps Pipeline

Ha-Gyeong Kim[1 [0009−0008−5840−5517]], Hyeon-Woo Choi[1 [0009−0004−6826−4149]], Min-Seo Kim[1 [0009−0008−1979−5778]], Yu-lim Hyoung[1 [0009−0009−7530−6497]], and Hwa-Jeong Seo[1 [0000−0003−0069−9061]]

Hansung University, Seoul, Korea
kimhaha4420@gmail.com, apinball91@gmail.com, msqoo0212@gmail.com, yulim4hyoung@gmail.com, hwajeong84@gmail.com

Abstract. The "Harvest Now, Decrypt Later (HNDL)" attack—which intercepts and stores encrypted communications before quantum computers become practical and decrypts them afterward—poses a long-term threat to TLS communication channels. Following the completion of NIST PQC standardization, organizations must pursue a phased migration of their TLS layers; however, existing approaches are limited by late detection of configuration errors, absence of cryptographic asset tracking, and insufficient automated response. This paper proposes an 11-Step DevSecOps pipeline grounded in Policy as Code. The pipeline automatically validates and deploys a 3-Stage TLS migration—Classical ECC (Stage 1), Hybrid PQC (Stage 2), and Advanced Hybrid PQC (Stage 3)—through 11 Steps per execution, integrating OQS-Nginx custom builds, security scanning, CBOM Diff, and matrix CI regression detection. In our evaluation against 6 PQC toolchain pitfalls, standard SCA/SAST/SBOM pre-detected 0/6, whereas custom builds pre-prevented 5/6 and matrix CI detected 6/6 at runtime. In an evaluation of LLM-based code migration across 80 trials (55 responses received), even with static AST validation applied, 24 failures due to signature/argument hallucination (43.6% of response trials) were only caught at runtime, quantitatively confirming that semantic execution validation is an essential component of multi-layered defense.

Keywords: Post-Quantum Cryptography · DevSecOps · Hybrid PQC-TLS · TLS Migration · CI/CD Pipeline

1 Introduction

The advancement of quantum computers poses a substantial threat to the public-key cryptographic systems that currently underpin Internet security. Shor's algorithm can break widely-used public-key algorithms such as RSA and ECC in polynomial time, meaning that once a cryptographically relevant quantum computer emerges, all existing encrypted communications become fundamentally decryptable [32,33]. A more immediate concern is the "Harvest Now, Decrypt Later (HNDL)" attack. Adversaries collect large volumes of TLS-encrypted

network traffic today and decrypt it in the future using quantum computers [11,12]. The essence of this threat lies in the temporal asymmetry between the act of collection and the time of decryption; communications in domains requiring long-term confidentiality—such as healthcare, defense, and finance—are already exposed even before quantum computers become practical. TLS is the most widely deployed encryption protocol on the Internet today, and its ECDH/DH-based algorithms are vulnerable to Shor's algorithm. Consequently, TLS traffic transmitted today may already be subject to collection, and migration to Post-Quantum Cryptography (PQC) must be carried out proactively before quantum computers become practical.

In response to these threats, NIST officially finalized ML-KEM (FIPS 203), ML-DSA (FIPS 204), and SLH-DSA (FIPS 205) as PQC standards in 2024 [15,14,16]. While PQC standardization demands system migration, a wholesale cutover is impractical due to insufficient performance and compatibility validation. A phased migration model that ensures availability while incrementally strengthening security is therefore essential.

Phased PQC-TLS migration is not completed by algorithm replacement alone. In practice, the following six problems arise structurally in production environments.

1. Without automated verification of TLS configuration cryptographic policy compliance at the pre-deployment stage, configuration errors are only discovered at the time of production deployment.
2. Compatibility verification is required during the transitional period when PQC-capable and PQC-incapable clients coexist. While Hybrid PQC provides a classical fallback for non-PQC clients, there is no automated mechanism to verify that this behavior actually operates as intended.
3. Without a means to systematically record and compare cryptographic asset changes before and after migration, it is impossible to objectively prove which algorithms were removed or added, and regressions are difficult to detect promptly.
4. Without automated response when performance degrades or negotiation fails, operational applicability suffers. PQC migration can introduce computational and communication overhead beyond that of ECC-based configurations, and a mechanism for automatically rolling back to the previous Stage when performance degradation exceeds thresholds is needed.
5. Library heterogeneity and protocol parameter mismatches arising from PQC stack integration form new operational risks that exceed the detection coverage of existing security tools (SCA, SAST, etc.).
6. Hallucination errors arising during LLM-based automated migration are at high risk of bypassing static validation and directly causing runtime availability failures.

This study defines its protection scope as TLS communication channels directly exposed to HNDL attacks. We assume an adversary who can passively

observe and collect TLS-encrypted traffic on the network path, does not currently possess a quantum computer, but can leverage a sufficiently scaled quantum computer in the future to decrypt collected traffic. The target of attack is the confidentiality of session data protected by key exchange, and this study mitigates the threat through quantum-resistant migration of TLS key exchange and certificate signature algorithms.

The following are excluded from the scope of this study: database encryption, at-rest data encryption, and application-internal cryptographic migration. However, the rationale for generalizing this pipeline structure to adjacent domains and directions for extension are discussed separately in §5.3. Stage 3, given that standardization and implementation support for pure PQC-only TLS have yet to mature, is validated in an Advanced Hybrid PQC (NIST L5)[1] configuration, and pure PQC-only TLS is left for future research.

This study makes the following contributions to address these problems.

1. **Design and implementation of a Policy as Code-based 3-Stage TLS migration model and 11-Step DevSecOps pipeline.** We define a phased migration model comprising Classical ECC (Stage 1), Hybrid PQC (Stage 2), and Advanced Hybrid PQC (NIST L5, Stage 3), and build an 11-Step pipeline integrating TLS configuration changes, security scanning, dynamic validation, performance verification, and automatic rollback for each Stage. The pipeline (i) proactively blocks policy-violating configurations from reaching production via a Security Gate, (ii) generates automatic rollback PRs when performance thresholds are exceeded to eliminate dependence on manual recovery, and (iii) automates the validation and deployment cycle for Stage 1→2→3 transitions within a single pipeline, thereby catching at the PR stage the configuration errors and performance regressions that previously only surfaced at production deployment time under manual validation workflows.
2. **Cataloging 6 PQC toolchain pitfalls and proposing a custom build + matrix CI-based mitigation structure.** We structure 6 infrastructure-level pitfalls (F-1~F-6) arising during OQS stack integration in terms of trigger, symptom, cause, and countermeasure. We quantitatively demonstrate that standard SCA (Software Composition Analysis)/SAST (Static Application Security Testing)/SBOM (Software Bill of Materials) tools pre-detect 0/6, and show that applying our custom build (5/6 pre-prevented) + matrix CI (6/6 runtime-detected) mitigation structure blocks all 6 pitfalls via pre-prevention or runtime detection, achieving a pitfall coverage improvement of 0/6→6/6 over standard tools.
3. **Cryptographic asset migration tracking via CycloneDX-based CBOM Diff.** We build a structure that automatically classifies cryptographic as-

[1] In this paper, "NIST L5" refers to the security strength of Stage 3's priority key exchange group (`p521_mlkem1024`, ML-KEM-1024 based). ML-DSA-65 used for certificate signing corresponds to NIST Category 3; upgrading the signature algorithm to L5 (ML-DSA-87) is left as future work pending maturation of oqs-provider support.

set changes before and after migration into four states—BASELINE, IMPROVED, UNCHANGED, and REGRESSED—via CycloneDX 1.7-based CBOM generation and cross-run Diff. In conventional approaches that do not use CBOM Diff, tracking cryptographic asset changes depends on manual review or general component comparison in SBOM, making it difficult to promptly identify algorithm-level regressions (e.g., when a PQC group reverts to classical). Our automatic classification structure detects algorithm-level regressions at CI execution time and blocks them before production deployment.

4. **Quantifying LLM-based PQC migration failure modes and demonstrating the necessity of multi-layered validation gates.** We build an extension module that detects legacy cryptography with Semgrep, automatically converts it to ML-KEM/ML-DSA-based code via LLM (gpt-4o-mini), and creates PRs. The evaluation purpose of this module is not to propose LLM as a production automation tool, but to measure the effectiveness of multi-layered validation gates that safely block its output. Through an 80-trial quantitative evaluation (N=16 patterns orthogonally designed across library, use case, and complexity; k=5 repetitions), we confirm that all 55 trials that received responses failed semantic execution validation (L3=0/55, one-sided upper bound 5.5%), and separately measure that 24 signature hallucination failures (43.6% of 55 response trials) out of 7-type failure mode taxonomy cannot be blocked by static AST validation (L2) and can only be blocked by semantic execution validation (L3). By quantitatively demonstrating that the overlap between failure categories caught by L2 and L3 is 0, we show that the two validation layers are mutually complementary.

2 Related Work

2.1 PQC TLS Migration Research

The Open Quantum Safe (OQS) project is an open-source project supporting quantum-resistant cryptographic migration, providing liboqs as a PQC algorithm implementation and oqs-provider to enable PQC algorithms in OpenSSL 3 [38,19,21]. OQS provides an environment for experimenting with PQC key exchange and authentication in TLS 1.3 [28], X.509, and S/MIME based on the OpenSSL 3 provider, and supports TLS handshake-level experimental configurations via OQS-Nginx and OQS-curl demo images [20]. Groups such as `X25519MLKEM768` for supporting hybrid key exchange in TLS 1.3 have been proposed as IETF Internet-Drafts [8,37], and major web service providers have conducted experiments validating the feasibility of deploying hybrid post-quantum TLS in real-traffic environments [39,40,10,9]. NIST finalized ML-KEM (FIPS 203), ML-DSA (FIPS 204), and SLH-DSA (FIPS 205) as PQC standards in 2024 [15,14,16].

However, existing work focuses on providing PQC TLS experimental environments (OQS demos) [38,20] or measuring hybrid deployment feasibility in real-traffic environments (Cloudflare, Google) [40,10], while operational structures in which phased migration policy is specified as code and a CI/CD pipeline

automatically enforces validation, deployment, rollback, and regression detection in a single workflow remain an area requiring additional user-side design (see Table 1). This study fills this gap by designing and implementing an operational workflow that automatically enforces phased migration of Classical ECC, Hybrid PQC, and Advanced Hybrid PQC at the pipeline level on top of the OQS stack, and additionally provides a blocking and detection structure for 6 infrastructure integration pitfalls that standard SCA/SAST/SBOM tools cannot pre-detect.

2.2 DevSecOps and CI/CD Security Integration

DevSecOps is a methodology that integrates development, security, and operations to embed security as part of the CI/CD pipeline [27]. Tools such as Trivy, Gitleaks, and Semgrep are used to automate vulnerability scanning, secret detection, and static analysis in CI/CD pipelines [1,29,30]. Furthermore, the Policy as Code (PaC) approach, exemplified by Open Policy Agent (OPA), provides a structure for specifying and automatically enforcing infrastructure and security policies as code [18].

However, existing DevSecOps approaches focus on general vulnerability detection and deployment blocking, and the structure of defining TLS cryptographic policy incrementally and having a pipeline enforce PQC migration progress is relatively underaddressed. This study is distinguished from prior work by enforcing TLS configuration cryptographic algorithm compliance via Policy as Code and integrating a Security Gate into the pipeline that automatically blocks deployment on policy violations.

2.3 CBOM and Cryptographic Asset Management

A Cryptographic Bill of Materials (CBOM) is an object model for describing cryptographic assets in a system—such as algorithms, keys, and certificates—and their dependencies; CycloneDX supports CBOM representation [23,22,7]. Using CBOM, one can identify the cryptographic algorithms and certificate configurations in use within a system and compile an inventory of quantum-vulnerable cryptographic assets. PQC migration guidelines generally present identification and inventory of target cryptographic assets as a prerequisite step [17].

However, existing approaches focus on CBOM generation itself, and the structure for automatically comparing CBOMs before and after migration to classify progress as BASELINE, IMPROVED, UNCHANGED, or REGRESSED is not sufficiently addressed. This study implements CycloneDX 1.7-based CBOM generation and automated cross-run Diff, presenting a structure for objectively tracking migration progress and promptly detecting regressions.

2.4 LLM-based Security Code Migration

Research leveraging large language models (LLMs) for automatic code transformation spans diverse areas including general code migration and automated

repair of vulnerable code [4,2]. LLM-based security code generation and analysis studies have reported that LLMs can misuse cryptography-related APIs and generate code containing security vulnerabilities [25,13]. Attempts to automate code repair by combining static analysis with LLMs also exist [26].

However, existing research primarily focuses on general code generation, vulnerability repair, and cryptographic API misuse detection; studies that separately measure static validation and semantic execution validation targeting PQC migration code from RSA to ML-KEM/ML-DSA are limited. This study designs N=16 synthetic patterns dedicated to PQC migration and separately measures the roles of static AST validation and semantic execution validation through an 80-trial quantitative evaluation with 4-layer validation (L1–L4).

2.5 Comparison with Prior Work

To clarify the positioning of this study, Table 1 summarizes the functional scope of key prior work and systems against this study. Each entry is evaluated on: (i) Hybrid PQC TLS support, (ii) phased migration model, (iii) Policy as Code-based policy enforcement, (iv) automated CBOM Diff tracking, (v) LLM-based code migration evaluation, (vi) automatic rollback on performance degradation, and (vii) matrix CI-based regression detection.

Table 1. Functional Scope Comparison of Prior Work and This Study

Study/System	Hybrid TLS	Phased Mgr.	PaC	CBOM Diff	LLM Eval.	Auto Roll-back	Matrix Regr.
OQS Demo[38,20]	✓	–	–	–	–	–	–
Cloudflare X25519Kyber768[40,5]	✓	✓	–	–	–	–	–
Google CECPQ2[10,9]	✓	✓	–	–	–	–	–
AWS s2n-tls PQ[6]	✓	✓	–	–	–	–	–
CycloneDX CBOM[23,22]	–	–	–	△	–	–	–
NIST IR 8547[17]	Rec.	Rec.	–	–	–	–	–
This Study	✓	✓	✓	✓	✓	✓	✓

✓: Supported, △: Specification-level only, –: Not supported / out of scope

The comparison above shows that this study is complementary to existing hybrid TLS deployment cases (Cloudflare, Google, AWS), the CBOM specification (CycloneDX), and the migration guide (NIST IR 8547). While prior work addresses empirically measured hybrid TLS deployment feasibility or asset identification and migration recommendations, this study integrates automated validation, enforcement, rollback, and regression detection for phased migration at the operational pipeline level on top of these foundations.

3 Design and Implementation

This section first describes the phased TLS migration model and the DevSecOps pipeline structure that enforces it from a design perspective, then details the implementation of each component. The overall structure of the pipeline implemented in this study, as shown in Fig. 1, consists of three workflows: AI migration, matrix CI, and the main pipeline. Sections 3.1–3.2 cover the model and overall structure; 3.3–3.4 cover the PQC TLS server and security gate; 3.5 covers the 5-layer CBOM evidence system; and 3.6–3.8 cover performance verification, automatic rollback, LLM migration, and matrix CI regression detection.

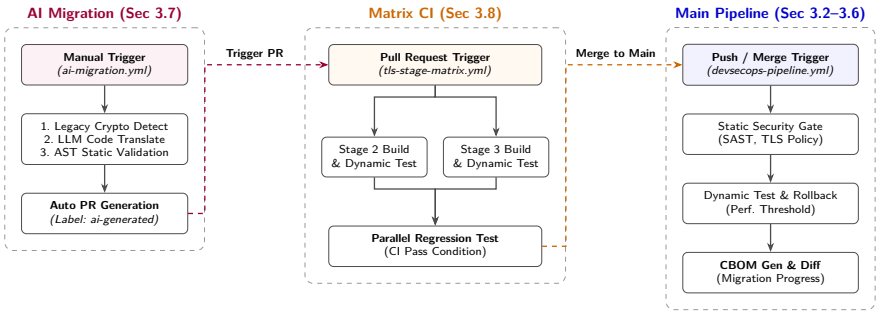


Fig. 1. Integrated Architecture of the PQC Migration DevSecOps Pipeline

3.1 3-Stage TLS Migration Model [2]

This study defines TLS migration into three Stages (Table 2). Each Stage specifies permitted algorithms via the `ssl_ecdh_curve` and `ssl_protocols` directives in the nginx configuration file. Policy compliance is automatically determined by the pipeline's Security Gate.

Stage 1 is the baseline phase that maintains existing ECC-based TLS while recording current cryptographic assets in CBOM. Stage 2 prioritizes `X25519MLKEM768` and additionally permits the classical group so that clients without PQC support negotiate via X25519, ensuring service continuity. Stage 3 permits only `p521_mlkem1024:p384_mlkem768` hybrid groups, blocks classical

[2] The coexistence of notations such as `X25519MLKEM768` (consecutive uppercase) and `p521_mlkem1024` (lowercase + underscore) in this table and throughout the paper reflects that each name is defined in a different source. The former follows the TLS group codepoint naming convention of the IETF Internet-Draft (`draft-kwiatkowski-tls-ecdhe-mlkem`), while the latter follows the group naming rules of the OpenSSL/oqs-provider implementation. To maintain consistency with the original specifications and configuration files, each notation is used as-is in this paper.

Table 2. 3-Stage TLS Migration Model

Stage	Key Exchange Group	Cert. Signature	Characteristics
Stage 1 (Classical ECC)	X25519, P-256, P-384	RSA-2048	TLS 1.2/1.3, no PQC
Stage 2 (Hybrid PQC)	`X25519MLKEM768` `:X25519`	RSA-2048	Hybrid, classical group permitted
Stage 3 (Advanced Hybrid PQC)	`p521_mlkem1024` `:p384_mlkem768`	ML-DSA-65	NIST L5, classical fallback blocked

fallback, and uses an ML-DSA-65 (Dilithium3)-based certificate for signing. Stage 3 is positioned as an experimental extension phase validated in an Advanced Hybrid PQC (NIST L5) configuration, given that standardization and implementation support for pure PQC-only TLS have yet to mature.

3.2 11-Step DevSecOps Pipeline Structure

This study implements an 11-Step pipeline based on GitHub Actions. The pipeline runs automatically on code push or PR events and has a sequential structure in which each Step's output becomes the input for the next (Fig. 2).

1. **Code Checkout**: Fetches repository code and initializes the execution environment.
2. **Stage Detection and TLS Configuration Preparation**: Parses the `ssl_ecdh_curve` directive in `nginx.conf` to automatically detect the current Stage, and copies the designated Stage's configuration file on `workflow_dispatch` execution.
3. **Automatic nginx Configuration Update**: Detects a Classical TLS configuration lacking PQC key exchange groups and automatically replaces it with a Hybrid PQC configuration; commits and pushes the change so that subsequent Steps run against the updated configuration.
4. **Classical Cryptography Detection**: Uses Semgrep's `crypto-classical.yaml` custom rule to detect legacy cryptographic usage (RSA, ECDSA, MD5, SHA-1, DES) across the entire codebase.
5. **Static Security Analysis**: Runs Trivy (CVE + Misconfiguration), Gitleaks (secrets), and Semgrep (SAST) sequentially and outputs results to `scan-summary.json`. Blocks the pipeline on Gitleaks or classical cryptography detection.
6. **TLS Cryptographic Policy Validation (Security Gate)**: Re-validates the nginx configuration directory with Semgrep's `tls-policy.yaml` rule, excluding the Stage 1 configuration file (`nginx-ecc.conf`) from scanning; checks for use of TLSv1.0/1.1, RC4, and DES, and blocks deployment on violation.

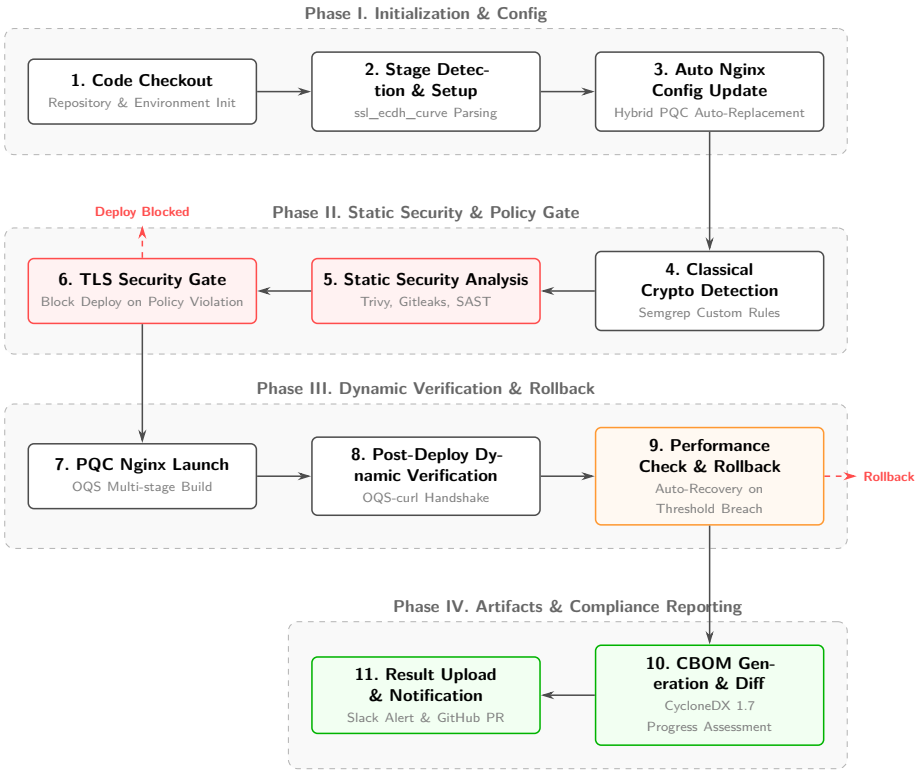


Fig. 2. Step Structure of the 11-Step DevSecOps Pipeline

7. **PQC Nginx Docker Build and Startup**: Builds and starts an OQS-Nginx container using the custom Dockerfile. Proceeds to the next Step after passing the healthcheck.
8. **Post-Deployment Dynamic Validation**: `tls_check.sh` specifies Stage-allowed groups via OQS-curl's `-curves` option to verify handshake success/failure and determines policy compliance for the current Stage.
9. **Performance Verification and Automatic Rollback**: `perf_check.sh` measures TLS handshake time, and `rollback.sh` automatically reverts to the previous Stage when thresholds are exceeded.
10. **CBOM Generation and Reporting**: `cbom_gen.py` generates a CycloneDX 1.7-based CBOM, and `cbom_diff.py` compares it with the previous-run CBOM to determine migration progress.
11. **Result Upload and Post-processing**: Uploads scan and CBOM artifacts, and automatically publishes GitHub Actions Job Summary and PR comments. On main branch failure, automatically creates a GitHub Issue and sends a Slack alert.

3.3 PQC TLS Server Based on OQS-Nginx Custom Build

The publicly available OQS-Nginx image (`openquantumsafe/nginx:0.11.0`) is based on a static OpenSSL 1.1.1q build, and PQC handshakes structurally fail because its group/sigalg code points differ from the OpenSSL 3 + oqs-provider line used by OQS-curl (`openquantumsafe/curl:0.11.0`). To resolve this, a multi-stage Dockerfile was created to source-build OpenSSL 3.4.0, liboqs 0.13.0, oqs-provider 0.9.0, and nginx 1.27.2, unifying them on the same library line as OQS-curl.

The key design decisions during the build process are as follows. When building liboqs, the `OQS_USE_OPENSSL=OFF` and `OQS_BUILD_ONLY_LIB=ON` flags were used to avoid OpenSSL KAT conflicts. For oqs-provider, `openssl_conf = openssl_init` was specified in `openssl.cnf` and the module path was set as an absolute path to guarantee automatic loading. The issue in Stage 3 where oqs-provider 0.9.0 reports PQC key security strength as 0, causing OpenSSL SECLEVEL validation failure, was temporarily mitigated with `ssl_ciphers DEFAULT:@SECLEVEL=0`, while `ssl_protocols TLSv1.3` and a group whitelist were used to block side effects.

The TLS configuration for each Stage is applied by having `entrypoint.sh` read the `TLS_STAGE` environment variable and replace the nginx configuration file with that Stage's file. This eliminates the need to rebuild the container for Stage transitions.

3.4 Vulnerability Analysis and TLS Policy Validation Module

The Security Gate consists of three scanners with clearly separated roles and a TLS policy validation script.

Trivy simultaneously scans for CVEs and configuration errors via the `-scanners vuln,misconfig` option and determines Pass/Fail based on HIGH and CRITICAL severity. Secret detection is handled exclusively by Gitleaks, so Trivy's secret scanner is not used. Items unavoidably triggered in the OQS image environment (root user, latest tag) are suppressed via `.trivyignore`. The specific reason for excluding Trivy's secret scanner lies in the difference in allowlist scope. The allowlist rules in `.gitleaks.toml` apply only to Gitleaks and not to Trivy's secret scanner. Therefore, enabling Trivy's secret scanner could cause PQC algorithm names appearing in configuration files (e.g., `X25519MLKEM768`) to be reported as false positives with no means of suppressing them via allowlist.

Gitleaks detects secrets in the current file state using `gitleaks detect -source`. False positives from PQC algorithm names (e.g., `X25519MLKEM768`) are eliminated via the `.gitleaks.toml` allowlist; exit code 0 (pass), 1 (fail), and 2+ (execution error) are distinguished.

Semgrep uses two types of custom rules operating in generic mode. `tls-policy.yaml` is configured to detect TLSv1.0/1.1 tokens and weak cipher suites such as RC4 and DES. `crypto-classical.yaml` detects legacy cryptographic API usage across the entire codebase including Python and Java. Since nginx configuration files are not natively supported by Semgrep, false positives

may arise from generic mode use; core TLS policy determination is handled separately by `tls_check.sh`. Semgrep's exit code is handled by distinguishing exit 1 (findings present) from exit 2 and above (execution error). Exit 1 means the rule operated normally and findings exist; treating it the same as an execution error would prevent accurate determination of whether findings are present. `semgrep_scan.sh` first blocks exit 2 and above as execution errors, then separately determines the presence of ERROR severity findings based on exit 0/1.

`tls_check.sh` specifies Stage-allowed groups via OQS-curl's `-curves` option to verify handshake success/failure and determines policy compliance per Stage. The design rationale for choosing this approach lies in the characteristics of `curl -v` output format. The `curl -v` output of `openquantumsafe/curl` displays the negotiated protocol and cipher but does not include the negotiated key exchange group name in a parseable form. Therefore, instead of parsing the negotiated group name, the approach of specifying only the Stage-specific `-curves` group and determining policy from the handshake success/failure result was adopted. The cross-validation method differs per Stage. For Stage 2, both PQC handshake success and classical-only client connection success are verified simultaneously to validate hybrid behavior. For Stage 3, both PQC handshake success and classical-only client rejection are verified simultaneously to validate classical fallback blocking. Pass conditions per Stage are shown in Table 3.

Table 3. TLS Policy Evaluation Criteria per Stage

Stage	Evaluation Items	Pass Conditions
Stage 1	`ssl_protocols`, `ssl_ecdh_curve`	classical handshake success, PQC group request rejected
Stage 2	`ssl_ecdh_curve`	`X25519MLKEM768` handshake success, classical-only client connection success
Stage 3	`ssl_ecdh_curve`	`p521_mlkem1024:p384_mlkem768` handshake success, classical-only client rejected

TLS policy validation consists of two layers: static and dynamic. Step 6's Semgrep `tls-policy.yaml` inspects text patterns in nginx configuration files to block weak protocol and cipher suite configurations before deployment. Step 8's `tls_check.sh` performs a handshake after actually starting the container to verify that the configuration is applied as intended at runtime. Semgrep pre-blocks configuration errors before deployment but cannot verify actual TLS negotiation results, while `tls_check.sh` validates runtime behavior but cannot detect pre-deployment configuration errors. The two layers are mutually complementary, ensuring policy compliance at both the configuration level and the runtime level.

3.5 CBOM Generation, Diff, and TLS Cross-Validation

The system's CBOM module consists of two scripts—`cbom_gen.py` (generation) and `cbom_diff.py` (history comparison and cross-validation)—and combines static and dynamic analysis to generate a CycloneDX 1.7 CBOM, then performs comparison with the previous-run CBOM and cross-validation against actual TLS negotiation results, all within a single pipeline Step (Step 10).

CBOM Generation via Combined Static and Dynamic Analysis. `cbom_gen.py` combines 5 data sources to identify cryptographic assets within the TLS termination scope (Table 4). On the static side, both the repository's nginx configuration file and the container's `nginx -T` output are parsed together, recording discrepancies between local and effective values caused by CI auto-replacement as `notes`. On the dynamic side, `docker exec` is used to call `tls_check.sh` in the tls-tester container to collect handshake results (protocol, cipher, key group, verdict), and `openssl x509 -text` is run in the pqc-proxy container to retrieve the server certificate and extract subject, issuer, validity period, signature algorithm, and public key information. Semgrep's `crypto-findings.json`, an external pipeline result, is post-injected as `securecapstone:classical_crypto_findings` and `securecapstone:migration_summary` properties immediately after CBOM generation.

Table 4. Data Sources and Mappings Combined for CBOM Generation

Data Source	Analysis Method	CBOM Mapping
`nginx-*.conf` (3 variants, one per Stage)	Static (file parsing)	configured protocol, cipher, key exchange, certificate path
Container `nginx -T`	Static (applied config)	effective config, local/applied value mismatch `notes`
`tls_check.sh` (tls-tester)	Dynamic (handshake)	negotiated protocol, cipher, key group, Stage verdict
`openssl x509` (pqc-proxy)	Dynamic (certificate lookup)	certificate, publicKey components, signature algorithm
Semgrep `crypto-findings.json`	External (code SAST)	`classical_crypto_findings` property

Component Structure and Dependency Graph. The generated CBOM uses CycloneDX 1.7's `cryptographic-asset` type to represent four asset types—algorithm, certificate, related-crypto-material (public-key, private-key), and protocol—as components, and specifies inter-asset reference relationships via the `dependencies` graph (Fig. 3).

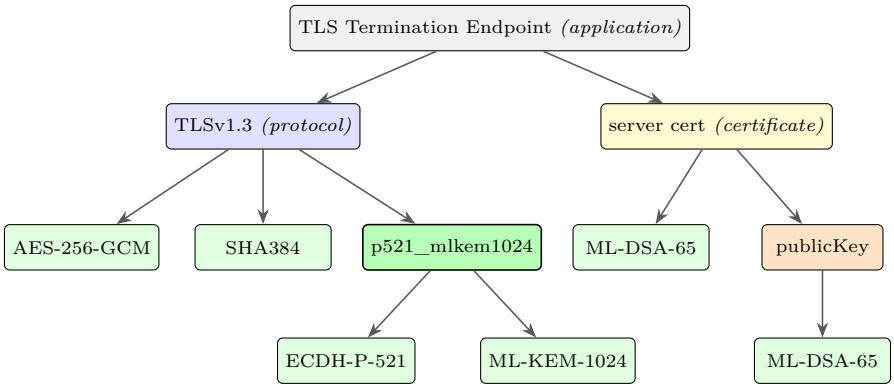


Fig. 3. Example CBOM component dependency graph for Stage 3. The hybrid key exchange group `p521_mlkem1024` is decomposed by `decompose_key_exchange_name` into `ECDH-P-521` and `ML-KEM-1024`, registered as separate algorithm components, and bound to the parent (composite) via `dependsOn`. Stage 3 registers two hybrid groups, `p521_mlkem1024:p384_mlkem768`, in `ssl_ecdh_curve`; this figure visualizes only the priority group. Node color distinguishes asset type (green: algorithm, blue: protocol, yellow: certificate, orange: related-crypto-material). In Stage 2, the key exchange group is replaced by `X25519MLKEM768` and the certificate signature by RSA-2048; in Stage 1, the key exchange group is reduced to a single `X25519`.

The key implementation decisions are as follows. First, hybrid key exchange groups such as `X25519MLKEM768` are decomposed into `X25519` and `ML-KEM-768` by the `decompose_key_exchange_name` function, registered as separate algorithm components, and the parent (composite) component is linked to both children via `dependsOn`. This enables individual tracking and comparison of classical and PQC assets within a hybrid group. Second, following nginx recommendations, since OpenSSL auto-negotiates when `ssl_ciphers` is not specified in TLS 1.3, this module automatically injects the 5 standard cipher suites from RFC 8446 §B.4 as assets and marks them with the `securecapstone:source=rfc8446-default` property to ensure completeness of the CBOM asset list. Third, algorithm families such as ML-KEM, ML-DSA, AES, SHA-2/3, and ChaCha20 are automatically inferred in `infer_algorithm_properties` and recorded simultaneously in CycloneDX 1.7's native `algorithmFamily` field and the `securecapstone:algorithmFamily` custom property.

Stage Classification and Triple Consistency Verification. The `classify_stage` function matches the negotiated key group string against regular expressions for hybrid PQC (`x25519_mlkem*`) and strong hybrid/pure PQC (`p\d+_mlkem*`, `mlkem\d+`) to automatically classify it as one of STAGE_1_CLASSICAL, STAGE_2_HYBRID_PQC, or STAGE_3_POST_QUANTUM. For accurate determination, the `cross_validate` function compares three values—-(i) the user-requested

Stage (`-stage` argument), (ii) this module's negotiation result-based classification, and (iii) `tls_check.sh`'s verdict—and records warnings in the CBOM along with `securecapstone:validation.consistent=false` if any of the three disagree. This triple verification goes beyond simple asset inventory to provide a self-checking mechanism embedded in the CBOM for the question "was the requested policy actually applied?"

Diff-based Migration Progress Classification. The `compare` function in `cbom_diff.py` compares the current-run CBOM with the previous-run CBOM and classifies progress into four states: BASELINE, IMPROVED, UNCHANGED, and REGRESSED. The comparison key is the set of `(file, line, rule)` tuples from Semgrep findings, and the metric is the delta of the `securecapstone:migration_summary.manual_action_required` count. If no previous CBOM exists, it is labeled BASELINE; a positive delta yields IMPROVED (legacy assets reduced); negative yields REGRESSED; and zero yields UNCHANGED. The set differences also produce `resolved` (present previously but not now) and `new_findings` (new in current run) to preserve the details of changes. The previous-run CBOM is automatically downloaded as the latest artifact matching the `cbom-stage{N}-run*` format via the GitHub Actions API. This metric defines code-level PQC migration progress (residual legacy cryptographic API usage) as the primary indicator, while Stage transitions of TLS configuration assets are tracked as separate layers by the triple consistency verification above and the TLS cross-validation described below. That is, Diff and consistency verification cover different asset domains (application code vs. TLS termination policy).

Cross-Validation with the TLS Stack. While CycloneDX schema validation ensures *format* conformance of the CBOM, the `verify_tls_against_cbom` function in `cbom_diff.py` verifies the *content* accuracy of the CBOM. Specifically, `openssl s_client -brief -connect proxy-server:443` is executed via `docker exec tls-tester` to collect the actually negotiated protocol, cipher, and key group; the algorithm keyword set extracted from the negotiation result is then compared with the CBOM's algorithm component name set after normalization for exact match. Since the CI runner host cannot resolve docker-compose internal DNS and lacks the OQS provider, a workaround of running `openssl` inside the tls-tester container is used.

To verify that Stage policy is actually applied, this module explicitly forces each Stage's intended key exchange group via the `-groups` option per Stage. For Stage 2, `-groups X25519MLKEM768` is specified; for Stage 3, `-groups p521_mlkem1024:p384_mlkem768` is specified to directly verify that each Stage negotiates the policy-intended hybrid group. The reason for explicitly forcing the group is that `openssl s_client` called without the `-groups` option prioritizes classical groups even when the OQS provider is active, so this ensures negotiation of the intended PQC group. The verdict is recorded in the `securecapstone:tls_verification` property as one of PASS, MISMATCH, or

SKIPPED: PASS if the negotiation result matches the CBOM, MISMATCH if it does not, and SKIPPED if the handshake itself failed and no negotiation result was obtained. The `-fail-on-mismatch` option blocks the pipeline when MISMATCH occurs.

Schema Validation and External Compatibility. The final CBOM is mandatorily validated for 1.7 schema conformance by the CycloneDX CLI (`cyclonedx validate -input-version v1_7 -fail-on-errors`), then accumulated as a permanent artifact in `cbom-stage{N}-run{run_number}` format to serve as the baseline for the next run. Since CycloneDX is a format directly consumable by standard SBOM ecosystem tools such as OWASP Dependency-Track, `cdxgen`, and `grype`, this CBOM functions not as a closed artifact but as standards-compatible evidence directly integrable into external asset management systems. As a result, CBOM provides an evidence system consisting of five layers—static/dynamic asset identification, triple consistency verification, automated progress classification, real TLS stack cross-validation, and schema conformance validation—enabling objective proof of cryptographic asset changes and regressions before and after Stage transitions at every pipeline execution.

3.6 Performance Verification and Automatic Rollback

`perf_check.sh` measures TLS handshake completion time 30 times using curl's `-write-out "%time_appconnect"` option. The median is used as the representative value, and the 90th percentile and failure rate are also recorded to mitigate the effect of transient delays in the pipeline environment. The performance gate applies all three independent thresholds, each set based on the following rationale. The relative threshold is 7× the Stage 1 baseline median. Prior work [35] reports normal PQC TLS handshake overhead compared to ECC at approximately 5.8–6.4× in compute-bound measurements on the same host virtualization. This study adopts these measurements as a reference for normal overhead, and defines 7×—exceeding the upper end—as the threshold for abnormal delay to avoid misclassifying measurement environment differences and transient pipeline fluctuations as regressions. The absolute threshold is a median exceeding 3,000 ms, set at a level that clearly departs from normal PQC handshake latency and suggests server errors or configuration defects. The failure rate threshold is 10% or more, a relaxed value accounting for network noise and container initialization delays on GitHub Actions runners [31,3]. If any one of the three thresholds is violated, the result is determined as `result=fail`. Stage 1 is excluded from rollback determination as it is the baseline collection phase.

`rollback.sh` reads `perf_check.sh`'s `perf-check-summary.json` and executes rollback only when `result=fail`. The target configuration file for rollback is determined as `nginx-ecc.conf` (Stage 1) for Stage 2, and `nginx-hybrid.conf` (Stage 2) for Stage 3, and copied to `nginx/nginx.conf`. To comply with protected branch rules, instead of pushing directly to `develop`, a dedicated branch in the format `rollback/stage-N-to-M-timestamp` is created and a PR is automatically generated via `gh pr create`. The rollback commit message includes a

`[skip ci]` tag to prevent the rollback commit from re-triggering the pipeline. The thresholds are provided as policy examples adjustable according to the operating environment.

3.7 AI-based Legacy Code Migration

This module is an extension of the PQC migration automation pipeline, consisting of a 4-step flow that automatically converts legacy cryptographic usage in the codebase via LLM.

1. **Detection**: Identifies files and locations with legacy cryptographic usage via Semgrep `crypto-classical.yaml` and outputs to `crypto-findings.json`.
2. **LLM Conversion**: Automatically converts RSA code in detected files to ML-KEM/ML-DSA-based oqs-python code via GitHub Models API (`gpt-4o-mini`). The prompt consists of an instruction specifying API signatures, migration rules, and output format.
3. **Validation**: Verifies oqs API usage and detects structural errors such as direct class calls via AST-based static validation.
4. **PR Creation**: Commits validated code to a new branch and automatically creates a PR with the `ai-generated` label to prompt human final review.

In the automatic PR creation step of this implementation, AST-based static validation is applied first, and the necessity of semantic execution validation is quantitatively analyzed via a separate evaluation harness in Section 4. This module is manually triggered via the `ai-migration.yml` workflow and is separated from the main pipeline to isolate cost and noise.

3.8 Matrix CI-based Regression Detection

The main pipeline (`devsecops-pipeline.yml`) operates with the specified Stage on `workflow_dispatch` execution, but on default push/PR triggers it operates with the Stage detected from the configuration file. This structure creates a gap where regressions triggered by nginx-related file changes in Stage 2/3 are not automatically detected at PR time.

`tls-stage-matrix.yml` is a matrix workflow that triggers only on changes to TLS stack-related files—such as `nginx/**`, `tester/**`, `scanner/tls_check.sh`, `tls-policy.yaml`, and `docker-compose.yml`—to fill this gap by validating Stage 2 and Stage 3 in parallel. A path filter is applied to minimize CI cost by not running on unrelated commits. `fail-fast: false` is applied to the matrix strategy so that even if validation of one Stage fails, validation of the remaining Stage runs to completion. Setting `fail-fast: true` (the default) would cancel Stage 3 execution on Stage 2 failure, making it impossible to identify regressions in both Stages within a single pipeline run. For each Stage, OQS-Nginx build, startup, healthcheck, and `tls_check.sh` execution are performed; on healthcheck failure, `docker inspect`, `docker logs`, and `nginx -T` output are automatically

dumped to enable immediate diagnosis of container startup failure causes from the pipeline log. If any Stage fails, the CI status is marked as failed, enabling regression verification before PR merge. This enables Stage 3 regressions to be detected immediately at PR time, reducing post-merge hotfix costs.

4 Evaluation

4.1 Evaluation Design and Research Questions

This evaluation validates the proposed system from three independent perspectives. Evaluation A catalogs infrastructure-level pitfalls arising during PQC toolchain integration and analyzes whether standard security tools can pre-detect them. Evaluation B quantitatively measures the accuracy of LLM-based legacy code migration and separately analyzes the role of each validation layer. Evaluation C measures the blocking and detection effectiveness of the proposed pipeline against the pitfalls identified in Evaluations A and B.

The evaluation questions (Q1∼Q4) were derived in an ablation-style structure that compresses the study's problem statement (Introduction, six structural problems) into two axes and pairs "limitations of existing approaches" with "effectiveness of the proposed structure" for each axis. The two axes are (i) infrastructure-level toolchain integration pitfalls (corresponding to problems 1, 2, 4, 5 in the Introduction) and (ii) LLM-based code migration hallucinations (corresponding to problem 6); for axis (i), the limitations of standard SCA/SAST/SBOM (Q1) are paired with the blocking/detection effectiveness of the proposed pipeline (Q3), and for axis (ii), the limitations of a single validation layer (Q2) are paired with the effectiveness of multi-layered validation (Q4). Problem 3 from the Introduction (tracking and comparing cryptographic assets before and after migration) is a separate evaluation target orthogonal to the above two axes, addressed in Evaluation D through CBOM Diff module verification.

This 2-axis × 2-phase (gap → mitigation) evaluation structure reflects the domain-specific characteristic that no prior framework simultaneously addresses both PQC toolchain integration evaluation and LLM-based code migration multi-layer validation evaluation. This study synthesizes evaluation practices from adjacent fields (PQC performance and compatibility measurement [24,34,36], LLM code generation accuracy evaluation frameworks [4,2], DevSecOps static analysis tool effectiveness measurement [27], etc.) to independently design the evaluation structure.

The questions answered by each evaluation are as follows.

- **Q1** (Evaluation A): What types of pitfalls arise during PQC toolchain integration, and can standard SCA/SAST/SBOM tools pre-detect them via static or SBOM-based analysis?
- **Q2** (Evaluation B): At which validation layer does LLM-based RSA→ML-KEM/ML-DSA code migration fail, and what failure types do static AST validation and semantic execution validation each block?

- **Q3** (Evaluation C): How many of the 6 pitfalls that standard tools cannot pre-detect does the proposed pipeline block or detect?
- **Q4** (Evaluation C): How much does the risk of failed code reaching production decrease when multi-layered validation is applied compared to LLM alone?

4.2 Experimental Environment

The **pipeline execution environment** uses GitHub Actions ubuntu-latest runners. The OQS-Nginx server is based on source builds of OpenSSL 3.4.0, liboqs 0.13.0, oqs-provider 0.9.0, and nginx 1.27.2; the TLS validation client uses OQS-curl (`openquantumsafe/curl:0.11.0`).

The **LLM migration evaluation environment** was carried out on Windows 10 Pro, Docker Desktop, and `python:3.11-slim`-based containers. liboqs 0.14.0 and liboqs-python 0.14.1 are used; the LLM endpoint is GitHub Models API (`gpt-4o-mini`, temperature=0). Measurements were conducted on 2026-05-08, with a total of 80 trials (16 patterns × k=5). Tool and library versions used in the experiments are shown in Table 5.

Table 5. Experimental Environment and Tool Versions

Component	Version
OpenSSL	3.4.0 (source build)
liboqs (server)	0.13.0
liboqs (eval. harness)	0.14.0
oqs-provider	0.9.0
nginx	1.27.2
Trivy	0.69.3
Semgrep	1.156.0
Gitleaks	8.21.2
LLM Model	gpt-4o-mini (temperature=0)

4.3 Evaluation A: PQC Toolchain Pitfall Catalog

Six infrastructure-level pitfalls encountered during this study's OQS stack integration were structured by trigger, symptom, cause, and countermeasure (Table 6).

Analysis of whether standard security tools can pre-detect these pitfalls found that none of Trivy (SCA), Semgrep (SAST), or CycloneDX CBOM could pre-detect any of the 6 pitfalls via static or SBOM-based analysis (0/6). This indicates that standard tools focus on known CVEs, configuration errors, and cryptographic asset inventorying, and that integration-level pitfalls such as OQS stack library line asymmetry or code point mismatches are outside their pre-detection scope.

Table 6. Catalog of 6 PQC Toolchain Pitfalls

ID	Pitfall	Symptom	Countermeasure
F-1	oqs-provider version mismatch	TLS 1.3 handshake `alert 552`	Unify library line via custom build
F-2	OpenSSL 1.1.1 ↔ 3.x OID mismatch	`SSL_CTX_use_certificate failed`	Unify cert-builder to same line as nginx
F-3	ML-KEM IETF code point conflict	nginx parse pass, handshake group match 0	Unify to same library line
F-4	PQ key strength 0 reported	`ee key too small (SECLEVEL)`	`@SECLEVEL=0` + TLSv1.3 + group whitelist
F-5	Docker image tag / build reproducibility	Previously CI-passing image combination fails on later reconstruction	Custom build removes external image tag dependency
F-6	Standalone PQ group non-negotiation	nginx startup normal, actual handshake fails	Revert to hybrid group + matrix CI detection

4.4 Evaluation B: LLM-based Code Migration Accuracy

This evaluation is not intended to argue for the production applicability of the LLM-based PQC migration module; rather, its purpose is to (i) quantify the semantic accuracy level of LLM output alone, and (ii) separately measure which failure modes each multi-layered validation layer (L1∼L4) blocks. Therefore, the results of this section should be interpreted not as the LLM module's success rate, but as evidence of the safety guarantee that validation gates do not pass dangerous code to production.

L4 is an auxiliary metric that measures only structural similarity (API surface matching) between reference code and candidate, and is unrelated to semantic correctness. Since all trials that passed L4 in this measurement are runtime-failing code under L3=0%, L4 passing is not used as grounds for production safety determination. L4 is reported separately for two reasons: first, to check threshold sensitivity of multi-layer metrics; second, to quantify the proportion of hallucination code that is structurally similar but behaviorally different; only L1, L2, and L3 are used for the production gate.

To verify normal operation of the evaluation tooling itself, two extreme values were measured: (i) when correct code is submitted as candidate, L2/L3/L4 all pass 16/16; (ii) when original RSA code is submitted as-is, L2/L3/L4 all pass 0/16. Zero false negatives and zero false positives were confirmed at both extreme values.

The input patterns consist of an orthogonal design N=16 across library (cryptography.hazmat, PyCryptodome) × use case (keygen, encrypt, sign, kex) × complexity (simple, complex). For each pattern, conversion from RSA to ML-KEM-768/ML-DSA-65 (oqs-python)-based code was requested from gpt-4o-mini, and results were measured using the 4-layer validation framework.

The 4-layer validation framework (L1~L4) is defined as follows. L1 is whether an LLM API response is received; L2 is AST-based static validation (oqs API usage, instance creation patterns, prohibition of direct class calls); L3 is per-pattern round-trip semantic execution validation; and L4 is weighted-average AST feature equivalence verification between reference and candidate (threshold 0.8).

Multi-layer Ablation Results The pass rate per trial for 4-layer validation L1–L4 is shown in Table 7. All 55 trials that received responses failed somewhere, and zero cases produced semantically correct code (L3=0/55, Wilson 95% one-sided upper bound 5.5%). The fact that L4 pass rate is higher than L3 (L4=72.7%, L3=0.0%, response basis) suggests that structural similarity at the AST feature level does not guarantee actual executability, and that superficially similar code is the predominant pattern of LLM hallucination.

Table 7. Pass Rate by 4-Layer Validation (80 trials total; N=55 responses received)

Layer	Pass / 80	Wilson 95% CI	Response Basis	Wilson 95% CI
L1 (LLM Response Received)	55 (68.8%)	[57.9, 77.8]%	55/55	—
L2 (AST Validation)	24 (30.0%)	[21.1, 40.8]%	24/55 (43.6%)	[31.4, 56.7]%
L3 (Semantic Execution)	**0 (0.0%)**	**[0.0, 3.8]%**[a]	**0/55 (0.0%)**	**[0.0, 5.5]%**[a]
L4 (Struct. Similarity Ref.)[b]	— 40 (50.0%)	[39.3, 60.7]%	40/55 (72.7%)	[59.8, 82.7]%

[a]One-sided 95% upper bound for 0/n by Rule of 3. [b]L4 is a structural similarity metric not used for production decisions (see L4 scope in this section).
LLM API errors 25/80 (estimated rate limit), concentrated in patterns 13–16.

Failure Mode Taxonomy and Blocking Layer Assignment All 55 trials that received responses failed somewhere and are classified into 7 categories (Table 8). This table separately specifies, along with frequency, which of L2 (AST) and L3 (semantic execution) each category is blocked at.

The overlap between failure categories blocked by L2 and L3 is 0. `wrong_arg_count`, `wrong_kwarg`, `method_hallucination`, and `algorithm_name_hallucination` are 100% blocked only at L3 (signature and argument hallucinations, 24 total / 43.6%), while `missing_instance_construction`, `no_migration`, and `class_direct_call` are 100% blocked at L2 (structural omissions, 31 total / 56.4%). Therefore, static AST validation alone cannot block the 24 signature hallucination cases (43.6%), and semantic execution validation must be added to block them.

Table 8. 7 LLM Failure Modes and Blocking Layer Assignment (55 response trials)

Failure Mode	Freq. (%)	L2 Blocked	L3 Blocked
`missing_instance_construction`	20 (36.4%)	20	0
`wrong_arg_count`	16 (29.1%)	0	16
`no_migration`	6 (10.9%)	6	0
`class_direct_call`	5 (9.1%)	5	0
`wrong_kwarg`	5 (9.1%)	0	5
`method_hallucination`	2 (3.6%)	0	2
`algorithm_name_hallucination`	1 (1.8%)	0	1
Total	**55 (100%)**	**31 (56.4%)**	**24 (43.6%)**

4.5 Evaluation C: Pipeline Pitfall Blocking and Detection Effectiveness

Pipeline Effectiveness on Infrastructure Pitfalls Table 9 summarizes how the three configurations—standard tools, custom build, and matrix CI—each handle the 6 infrastructure pitfalls.

Table 9. Pipeline Layer Effectiveness on 6 Pitfalls

ID	Pitfall	Std. Tools (Pre-detect)	Custom Build (Pre-prevent)	Matrix CI (Runtime-detect)
F-1	oqs-provider version mismatch	×	✓ prevented	✓ detected
F-2	OID mismatch	×	✓ prevented	✓ detected
F-3	group code point	×	✓ prevented	✓ detected
F-4	SECLEVEL rejection	×	✓ prevented	✓ detected
F-5	image tag reproducibility	×	✓ prevented	✓ detected
F-6	standalone non-negotiation	×	—	✓ detected
Total		**0/6**	**5/6**	**6/6**

The custom build pre-prevents 5 of the 6 pitfalls, and matrix CI detects all 6 at runtime. Standard SCA/SAST/SBOM cannot pre-detect any of the 6 via static or SBOM-based analysis (0/6). F-6 (standalone PQ group non-negotiation) cannot be prevented even by the custom build and is addressed by reverting to a hybrid group and matrix CI detection.

LLM Multi-layered Defense Effectiveness Table 10 summarizes the risk of failed code being passed to production by validation layer application level.

When both AST and semantic execution validation are applied, zero code is passed to the production stage in this evaluation, blocking delivery of failed code. This does not mean LLM migration was successfully completed; it means that

Table 10. Failed Code Delivery Risk by Validation Layer Scenario

Scenario	Passing Trials	Runtime Failure Risk
① No validation (LLM output applied directly)	55	100%
② AST validation only (L2)	24	100%
③ AST + Semantic validation (L2 + L3)	0	N/A

the current model output failed semantic validation and thus dangerous code was not passed to the deployment stage.

Performance Verification Results TLS handshake time measured 30 times in the Stage 2 environment yielded a median of 44.575ms and a failure rate of 0%, passing the absolute threshold (3,000 ms). Relative threshold determination is designed to be performed against the Stage 1 baseline artifact; in this experiment, the path where `rollback.sh` creates a rollback-dedicated branch and PR was verified via a forced failure condition (`PERF_FORCE_FAIL=1`).

4.6 Evaluation D: CBOM Diff Module Behavior Verification

Evaluation Design Among the 5-layer evidence system of the CBOM module presented in §3.5, this section quantitatively verifies at a fixture-based smoke test level whether Diff-based progress classification (layer 3) and triple consistency verification (layer 2) operate as intended. Six Stage transition scenarios and five consistency verification scenarios are artificially constructed, and whether the label output by the module in each scenario matches the ground truth is confirmed. The label determination logic is based on the delta of the `migration_summary.manual_action_required` value.

Diff-based Progress Classification Results Diff labeling results for 6 Stage transition scenarios are summarized in Table 11.

This module confirmed with 6/6 matches that BASELINE, IMPROVED, REGRESSED, and UNCHANGED 4-label classification operates deterministically at the fixture level. The S4 (regression) scenario reported 2 new findings along with the REGRESSED label when `manual_action_required` increased 0→2, demonstrating that PQC migration progress regression is automatically detectable at the PR stage.

Triple Consistency Verification Results The `cross_validate` function compares three signals—(i) user-requested Stage, (ii) cbom_gen's own verdict, and (iii) `tls_check.sh`'s verdict—generating `consistent=true` on agreement and `consistent=false` with a warning message on disagreement. Direct call measurement results for 5 scenarios are summarized in Table 12.

Table 11. CBOM Diff Labeling Accuracy (6 fixture scenarios)

ID	Scenario	Prev. manual	Curr. manual	Delta	Expected Label	Actual Label	Match
S1	First run (no CBOM)	—	5	—	BASELINE	BASELINE	✓
S2	Stage 1 → Stage 2 transition	5	2	3	IMPROVED	IMPROVED	✓
S3	Stage 2 → Stage 3 transition	2	0	2	IMPROVED	IMPROVED	✓
S4	Stage 3 → Stage 2 forced regression	0	2	-2	REGRESSED	REGRESSED	✓
S5	Same Stage re-execution	2	2	0	UNCHANGED	UNCHANGED	✓
S6	Code addition only (no asset change)	2	2	0	UNCHANGED	UNCHANGED	✓
						Total	**6/6**

Table 12. Triple Consistency Verification Results (5 scenarios)

ID	Scenario	Expected consistent	Actual consistent	Warnings	Match
C1	All three signals Stage 2 consistent	True	True	0	✓
C2	Requested Stage 2, actual Stage 1 negotiated	False	False	1	✓
C3	cbom_gen Stage 2 vs script Stage 3	False	False	1	✓
C4	All three signals Stage 3 consistent	True	True	0	✓
C5	All three signals inconsistent	False	False	2	✓
				Total	**5/5**

In the C5 case (all three signals different), 2 warnings being reported separately means that which signal pair is inconsistent is exposed in a form that allows operators to debug.

Discussion This evaluation is a fixture-based behavioral verification in a single project environment; cumulative statistics-based quantitative evaluation in diverse OSS and operating environments is left as future work in Section 5. However, the results of 6/6 labeling accuracy and 5/5 triple consistency verification confirm at the smoke test level that progress classification (layer 3) and consistency

verification (layer 2) among the 5-layer evidence system designed in §3.5 operate as intended.

4.7 Evaluation E: Automatic Rollback Mechanism Static Verification

Evaluation Scope The automatic rollback module in §3.6 reverts the nginx configuration to the previous Stage and automatically creates a PR when the performance thresholds described in §3.6 (absolute 3,000 ms, 7× Stage 1 baseline, 10% failure rate) are exceeded. This section verifies via static analysis of `rollback.sh` code the mechanisms preventing a *rollback loop*—the most dangerous operational incident that automatic rollback can cause (the commit made by rollback re-triggers the next pipeline, leading to infinite repetition). Measurement of trigger behavior in actual performance regression scenarios (R1–R3) and end-to-end MTTR quantification are left as future work in Section 5.

Rollback Loop Prevention Mechanism `rollback.sh` includes three independent safeguards (Table 13).

Table 13. Static Verification of 3 Rollback Loop Prevention Mechanisms

ID	Mechanism	Present	Principle
M1	`[skip ci]` marker in commit message	✓	GitHub Actions skips the workflow trigger for that commit
M2	`git diff -cached -quiet` guard	✓	Skips commit entirely if nginx.conf already matches the rollback target
M3	Stage 1 short-circuit	✓	Stage 1 has no rollback target so exits early
	Total	**3/3**	

All three mechanisms were confirmed to exist in the `rollback.sh` source via static code verification. M2 (empty-diff guard) ensures that duplicate rollback PRs are not created for the same nginx.conf, and M3 blocks rollback attempts below Stage 1 to prevent an incorrect STAGE argument from causing infinite recursion. While this static verification does not measure actual behavior in R1–R3 threshold scenarios, it demonstrates that safeguards against the most dangerous failure mode (loop) of the automatic rollback mechanism are secured at the code level.

Discussion This evaluation statically confirmed only the existence of rollback loop prevention mechanisms; actual measurement of threshold trigger accuracy and end-to-end MTTR (trigger → PR merge-ready state) requires building a separate load injection environment and is left as future work in Section 5.

4.8 Discussion

Evaluations A and B show that pitfalls in the PQC migration process occur at both the infrastructure integration level and the code migration level, and that standard tools alone are insufficient to address either layer. Evaluation C quantitatively demonstrates that the combination of custom build (5/6 pre-prevented) and matrix CI (6/6 runtime-detected) fills the blind spots of standard tools for infrastructure pitfalls, and that the multi-layered structure of AST + semantic validation blocks delivery of failed code to production.

The scope and interpretive caveats of this evaluation are as follows. The infrastructure pitfall catalog is an empirical case from an N=1 project and thus has limitations for statistical generalization. The LLM evaluation targets only a single model (gpt-4o-mini) and N=16 synthetic patterns; results may differ with larger models or real OSS code. Performance measurements were conducted in GitHub Actions runner and Docker environments and should be interpreted as relative comparisons within the same environment rather than absolute performance values. Detailed limitations and future research directions are discussed in Section 5.

5 Conclusion

5.1 Summary

This paper designs and implements an 11-Step DevSecOps pipeline that automates phased migration from Classical ECC through Hybrid PQC to Advanced Hybrid PQC, targeting TLS communication channels directly exposed to HNDL attacks. The proposed system integrates TLS configuration changes, security scanning, dynamic validation, CBOM generation, performance verification, and automatic rollback into a single pipeline, and automatically enforces policy compliance for each migration Stage based on Policy as Code.

Three key findings were confirmed through evaluation.

First, against 6 infrastructure pitfalls arising during PQC toolchain integration, standard SCA/SAST/SBOM tools pre-detected 0/6, whereas custom builds pre-prevented 5/6 and matrix CI detected 6/6 at runtime.

Second, a structure was designed and implemented that automatically classifies cryptographic asset changes before and after Stage transitions as BASELINE, IMPROVED, UNCHANGED, or REGRESSED via CycloneDX 1.7-based CBOM Diff, enabling objective tracking of migration progress.

Third, the 80-trial quantitative evaluation of LLM-based code migration is not intended to argue for the production applicability of the LLM module, but to separately measure the effectiveness of multi-layered validation gates. Evaluation results confirmed that all 55 trials that received responses failed semantic execution validation (L3=0/55, one-sided upper bound 5.5%), and that 24 signature/argument hallucination failures (43.6% of 55 response trials) are not blocked by static AST validation but only by semantic execution validation. By quantitatively demonstrating that the overlap between failure categories caught

by L2 and L3 is 0, the study showed that the mutually complementary structure of the two validation layers is essential in the LLM migration context.

5.2 Limitations

This study has the following limitations.

First, the infrastructure pitfall catalog (F-1~F-6) is an empirical case from an N=1 project; there is no statistical guarantee that the same pitfalls occur in other OQS stack configurations or organizational environments.

Second, Stage 3 was validated in an Advanced Hybrid PQC (NIST L5) configuration due to the constraint that standardization and implementation support for pure PQC-only TLS have yet to mature. Pure PQC-only TLS is left for future research after the relevant standards and implementation support have stabilized.

Third, the LLM evaluation targets only a single model (gpt-4o-mini) and N=16 synthetic patterns; results with larger models such as GPT-4o, Claude, and Gemini, or with real OSS code, may differ.

Fourth, behavioral verification of this CBOM module's 5 layers (asset identification, triple consistency verification, progress classification, TLS cross-validation, schema conformance) was performed at a fixture-based smoke test level in Evaluation D of §4.6 (labeling 6/6, consistency 5/5); cumulative statistics-based quantitative evaluation in diverse OSS and operating environments is outside the scope of this study. Additionally, the automatic rollback module in Evaluation E of §4.7 statically confirmed only the existence of 3 rollback loop prevention mechanisms; measuring R1∼R3 threshold trigger accuracy and end-to-end MTTR via actual performance regression injection requires building a separate load injection environment. Both limitations are left as future work.

Fifth, performance measurements were conducted in GitHub Actions runner and local Docker environments, so generalization of absolute performance values has limitations. They should be interpreted as relative comparisons within the same environment, and long-term stability at real production traffic scale has not been validated.

Sixth, a publicly trusted CA-based PQC certificate infrastructure was not applied. The ML-DSA-65 certificate used in Stage 3 was configured as an internal experimental certificate rather than a publicly trusted CA system; applicability in real PKI environments requires separate review.

5.3 Future Work

Future research directions to address the limitations of this study are as follows.

First, completing Stage 3 on a standalone ML-KEM basis when pure PQC-only TLS support matures. After IETF standardization completion and normalization of oqs-provider's security strength reporting, transitioning from the current Advanced Hybrid configuration to PQC-only will be feasible.

Second, real PKI integration for ML-DSA-based certificate signing. Extending from the current internal experimental certificate approach to an internal CA-based ML-DSA certificate issuance system will increase applicability in real operating environments.

Third, enhancing the automatic cipher replacement capability upon classical cryptography detection. While the current pipeline focuses on automatic replacement of nginx TLS configuration, applying larger models and integrating runtime semantic tests are needed to improve the accuracy of the LLM migration module that automatically converts legacy cryptographic APIs in the codebase.

Fourth, expanding the infrastructure pitfall catalog. Since F-1~F-6 are based on a single project's experience, the catalog can be expanded through comparative research to determine whether similar patterns occur in other PQC stacks such as BoringSSL and Java Bouncy Castle.

Fifth, cumulative statistics-based quantitative evaluation of the CBOM module. The effectiveness of the 5-layer verification structure proposed in this study should be measured with data from multiple pipeline executions, statistically analyzing the accuracy of BASELINE, IMPROVED, UNCHANGED, and REGRESSED 4-state classification across diverse Stage transition scenarios, the PASS/MISMATCH/SKIPPED distribution of TLS cross-validation, and the false positive/negative rates of triple consistency verification.

Sixth, load injection experiments to validate the effectiveness of the automatic rollback module. Follow-up research will measure trigger accuracy in three scenarios—R1 (absolute threshold 5,000 ms injection), R2 (relative threshold 8× baseline latency), R3 (failure rate 15% induced)—and quantify the end-to-end MTTR from performance regression occurrence to rollback PR becoming merge-ready via N≥10 repetitions as mean and standard deviation. Measurement of the false-positive rollback rate that misidentifies legitimate performance variations as regressions under real production traffic can also be performed.

Seventh, expanding the protection scope. This study limited the protection scope to TLS communication channels as the primary HNDL exposure surface; however, quantum threats apply equally to all channels requiring long-term confidentiality, including database encryption (at-rest), stored data, inter-service mTLS, and JWT/JWS. The pipeline structure of this study is separated into TLS domain-specific logic (handshake validation in `tls_check.sh`, `TLS_DIRECTIVE_PATTERNS` and `TLS_GROUP_ALIASES` mappings in `cbom_gen.py`, Stage 2/3 nginx startup in matrix CI, etc.) and domain-agnostic logic (Policy as Code-based policy enforcement, CBOM Diff 4-state classification, LLM migration multi-layered validation gate, automatic rollback threshold and loop prevention mechanism); the latter can be directly generalized to adjacent domains by replacing only the domain validation logic. Specifically, follow-up research is possible extending the same 11-Step pipeline pattern to: ① KMS key algorithm policy enforcement for at-rest encryption (e.g., PostgreSQL TDE, S3 SSE-KMS), ② PQC migration of inter-service mTLS, and ③ application-level signature algorithm migration such as JWT/JWS; in such cases, the CBOM scope also expands from TLS termination to a full cryptographic asset inventory.

References

1. Aqua Security: Trivy: Comprehensive security scanner for containers and other artifacts. https://github.com/aquasecurity/trivy (2025), accessed: 2026-05-16
2. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C.J., Terry, M., Le, Q.V., Sutton, C.: Program synthesis with large language models. arXiv preprint arXiv:2108.07732 (2021)
3. Beyer, B., Jones, C., Petoff, J., Murphy, N.R. (eds.): Site Reliability Engineering: How Google Runs Production Systems. O'Reilly Media (2016), https://sre.google/sre-book/table-of-contents/
4. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)
5. Cloudflare: X25519MLKEM768: Cloudflare's default post-quantum hybrid key agreement for TLS. Cloudflare SSL/TLS Developer Documentation (2024), https://developers.cloudflare.com/ssl/post-quantum-cryptography/
6. Crockett, E., Paquin, C., Stebila, D.: Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH. NIST 2nd Post-Quantum Cryptography Standardization Conference (2019), https://eprint.iacr.org/2019/858
7. IBM Research: CBOM: Cryptography bill of materials. https://github.com/IBM/CBOM (2024), upstreamed into CycloneDX 1.6 specification
8. Kwiatkowski, K., Kampanakis, P., Westerbaan, B., Stebila, D.: Post-quantum hybrid ECDHE-ML-KEM key agreement for TLSv1.3. Internet-Draft draft-kwiatkowski-tls-ecdhe-mlkem-03 (2025), https://datatracker.ietf.org/doc/draft-kwiatkowski-tls-ecdhe-mlkem/
9. Kwiatkowski, K., Valenta, L.: The TLS post-quantum experiment. Cloudflare Blog (2016), https://blog.cloudflare.com/the-tls-post-quantum-experiment/
10. Langley, A.: Real-world measurements of structured-lattices and supersingular isogenies in TLS. Imperial Violet Blog (2019), https://www.imperialviolet.org/2019/10/30/pqsivssl.html
11. Mosca, M.: Cybersecurity in an era with quantum computers: Will we be ready? IEEE Security & Privacy **16**(5), 38–41 (2018). https://doi.org/10.1109/MSP.2018.3761723
12. Mosca, M., Piani, M.: Quantum threat timeline report 2024. Tech. rep., Global Risk Institute (2024), https://globalriskinstitute.org/publication/quantum-threat-timeline-report-2024/
13. Mousavi, Z., Islam, C., Moore, K., Abuadbba, A., Babar, M.A.: An investigation into misuse of Java security APIs by large language models. In: Proc. 19th ACM Asia Conference on Computer and Communications Security (ASIA CCS). ACM (2024). https://doi.org/10.1145/3634737.3661134
14. National Institute of Standards and Technology: Module-Lattice-Based Digital Signature Standard. Federal Information Processing Standards Publication FIPS PUB 204, U.S. Department of Commerce (Aug 2024). https://doi.org/10.6028/NIST.FIPS.204
15. National Institute of Standards and Technology: Module-Lattice-Based Key-Encapsulation Mechanism Standard. Federal Information Processing Standards Publication FIPS PUB 203, U.S. Department of Commerce (Aug 2024). https://doi.org/10.6028/NIST.FIPS.203
16. National Institute of Standards and Technology: Stateless Hash-Based Digital Signature Standard. Federal Information Processing Standards Publication FIPS

PUB 205, U.S. Department of Commerce (Aug 2024). https://doi.org/10.6028/NIST.FIPS.205
17. National Institute of Standards and Technology: Transition to Post-Quantum Cryptography Standards. NIST Internal Report (Initial Public Draft) NIST IR 8547 ipd, U.S. Department of Commerce (Nov 2024). https://doi.org/10.6028/NIST.IR.8547.ipd, https://csrc.nist.gov/pubs/ir/8547/ipd
18. Open Policy Agent Project: Open Policy Agent (OPA): Policy-based control for cloud native environments. https://www.openpolicyagent.org (2025), accessed: 2026-05-16
19. Open Quantum Safe Project: liboqs: C library for quantum-safe cryptographic algorithms. https://github.com/open-quantum-safe/liboqs (2025), accessed: 2026-05-16
20. Open Quantum Safe Project: oqs-demos: Demo integrations of OQS into popular open-source applications. https://github.com/open-quantum-safe/oqs-demos (2025), accessed: 2026-05-16
21. Open Quantum Safe Project: oqs-provider: OpenSSL 3 provider for quantum-safe algorithms. https://github.com/open-quantum-safe/oqs-provider (2025), accessed: 2026-05-16
22. OWASP Foundation: CycloneDX authoritative guide to Cryptography Bill of Materials (CBOM). OWASP CycloneDX Working Group (2024), https://cyclonedx.org/capabilities/cbom/
23. OWASP Foundation: CycloneDX specification v1.7. OWASP Foundation. Ratified as ECMA-424, 2nd Edition (Dec 2025), https://cyclonedx.org/specification/overview/
24. Paquin, C., Stebila, D., Tamvada, G.: Benchmarking post-quantum cryptography in TLS. In: Ding, J., Tillich, J.P. (eds.) Proc. 11th International Conference on Post-Quantum Cryptography (PQCrypto) 2020. Lecture Notes in Computer Science, vol. 12100, pp. 72–91. Springer (2020). https://doi.org/10.1007/978-3-030-44223-1_5
25. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R.: Asleep at the keyboard? Assessing the security of GitHub copilot's code contributions. In: Proc. IEEE Symposium on Security and Privacy (S&P). pp. 754–768. IEEE (2022). https://doi.org/10.1109/SP46214.2022.9833571
26. Pearce, H., Tan, B., Ahmad, B., Karri, R., Dolan-Gavitt, B.: Examining zero-shot vulnerability repair with large language models. In: Proc. IEEE Symposium on Security and Privacy (S&P). pp. 2339–2356. IEEE (2023). https://doi.org/10.1109/SP46215.2023.10179324
27. Rahman, A.A.U., Williams, L.: Software security in DevOps: Synthesizing practitioners' perceptions and practices. In: Proc. International Workshop on Continuous Software Evolution and Delivery (CSED). pp. 70–76. IEEE (2016). https://doi.org/10.1109/CSED.2016.025
28. Rescorla, E.: The transport layer security (TLS) protocol version 1.3. Tech. Rep. RFC 8446, Internet Engineering Task Force (Aug 2018). https://doi.org/10.17487/RFC8446
29. Rice, Z.: Gitleaks: Detect hardcoded secrets in code and Git history. https://github.com/gitleaks/gitleaks (2025), accessed: 2026-05-16
30. Semgrep, Inc.: Semgrep: Static analysis at ludicrous speed. https://semgrep.dev (2025), accessed: 2026-05-16
31. Sheffer, Y., Saint-Andre, P., Fossati, T.: Recommendations for secure use of transport layer security (TLS) and datagram transport layer security (DTLS).

Tech. Rep. RFC 9325 (BCP 195), Internet Engineering Task Force (Nov 2022). https://doi.org/10.17487/RFC9325
32. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: Proc. 35th Annual Symposium on Foundations of Computer Science (FOCS). pp. 124–134. IEEE Computer Society, Los Alamitos, CA (1994). https://doi.org/10.1109/SFCS.1994.365700
33. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal on Computing **26**(5), 1484–1509 (1997). https://doi.org/10.1137/S0097539795293172
34. Sikeridis, D., Kampanakis, P., Devetsikiotis, M.: Post-quantum authentication in TLS 1.3: A performance study. In: Proc. Network and Distributed System Security Symposium (NDSS). Internet Society (2020). https://doi.org/10.14722/ndss.2020.24203
35. Sim, M., Jo, S., Song, H., Kim, E.: Zero-downtime post-quantum TLS 1.3 migration: A bridge-server-based approach. In: Seo, H. (ed.) Information Security and Cryptology – ICISC 2025. Lecture Notes in Computer Science, vol. 16487, pp. 127–149. Springer, Singapore (2026). https://doi.org/10.1007/978-981-95-8034-7_7
36. Sosnowski, M., Wiedner, F., Hauser, E., Steger, L., Schoinianakis, D., Gallenmüller, S., Carle, G.: The performance of post-quantum TLS 1.3. In: Companion of the 19th International Conference on emerging Networking EXperiments and Technologies (CoNEXT). ACM (2023). https://doi.org/10.1145/3624354.3630585
37. Stebila, D., Fluhrer, S., Gueron, S.: Hybrid key exchange in TLS 1.3. Internet-Draft draft-ietf-tls-hybrid-design-16 (Sep 2025), https://datatracker.ietf.org/doc/draft-ietf-tls-hybrid-design/
38. Stebila, D., Mosca, M.: Post-quantum key exchange for the Internet and the Open Quantum Safe project. In: Avanzi, R., Heys, H. (eds.) Proc. 23rd International Conference on Selected Areas in Cryptography (SAC) 2016. Lecture Notes in Computer Science, vol. 10532, pp. 1–24. Springer (2017). https://doi.org/10.1007/978-3-319-69453-5_2
39. Westerbaan, B., Wood, C.A.: Defending against future threats: Cloudflare goes post-quantum. Cloudflare Blog (Oct 2022), https://blog.cloudflare.com/post-quantum-for-all/
40. Westerbaan, B., Wood, C.A.: Post-quantum cryptography goes GA. Cloudflare Blog (Sep 2023), https://blog.cloudflare.com/post-quantum-cryptography-ga/