PQCImplement-Bench: Evaluating LLMs on Implementing NIST Post-Quantum Standards in Production Cryptographic Libraries

Anonymous submission

Abstract

A large-scale quantum computer would break the public-key cryptography that protects today's communications, and NIST has standardized ML-KEM, ML-DSA, and SLH-DSA (FIPS 203–205) as replacements that cryptographic libraries must implement. It remains unexplored whether coding agents can turn such a standard into a working and conforming patch for an existing library. This paper presents PQCImplement-Bench, a benchmark comprising 15 tasks derived from 14 pull requests across nine repositories and seven programming languages, each pairing a pre-change repository snapshot with a reviewed specification of requirements and interfaces. The benchmark grades patches with two independent test oracles that separate correctness within the repository from conformance to the standard. One oracle consists of relevant repository tests, and the other consists of NIST validation vectors for each standard. We evaluate three models, GPT-6.1-Sol, Kimi K3.1 Pro, and GLM-5.3-FlashX in OpenHands. Given only the problem statement, GPT-6.1-Sol reaches a mean pass rate of 29.0% on the validation vectors and 18.9% on repository tests. With the interface specification added, the pass rate for GPT-6.1-Sol rises to 100%, while Kimi K3.1 Pro reaches 55.8% and GLM-5.3-FlashX 6.7%. On the repository tests, the three models reach mean pass rates of 93.7%, 52.4%, and 0.0%. We release the dataset and evaluation framework for future research.

Code and data —
https://anonymous.4open.science/r/pqc_impl_bench/

Introduction

A cryptographically relevant quantum computer would break RSA and elliptic-curve cryptography through Shor's algorithm (Shor 1997), and encrypted traffic recorded today could be decrypted once such a computer exists (Moody et al. 2026). NIST has standardized ML-KEM for key encapsulation and ML-DSA and SLH-DSA for digital signatures in FIPS 203, 204, and 205 (National Institute of Standards and Technology 2024b,a,c). Adopting these post-quantum cryptography (PQC) standards requires libraries to implement the algorithms within their existing interfaces, key representations, and build systems (Dip et al. 2026).

LLM coding agents could help implement these standards, but their patches must satisfy both repository requirements and the algorithms' specified behavior. Repository-level benchmarks assess generated patches with executable tests. SWE-bench pairs repository snapshots and issue descriptions with tests from upstream pull requests (PRs) (Jimenez et al. 2024), and SWE-Bench Pro adds explicit requirements and, where needed, interface definitions to clarify the behavior expected by those tests (Deng et al. 2025). Passing repository tests alone provides incomplete evidence of conformance to a cryptographic standard. Implementations also need to reproduce the expected outputs of published validation vectors.

We present PQCImplement-Bench, a benchmark of 15 implementation tasks drawn from 14 PRs in nine repositories and seven programming languages. Each task provides a pre-change repository and a reviewed problem statement, requirements, and interfaces. Agents work in isolated containers with access to the relevant NIST standards, while the upstream implementation and PR test patch remain hidden.

In statically typed languages, tests for new features may not compile or run until the types or methods they reference are implemented (Fakorede et al. 2026). Once these tests can run, the reported results may still hide partial progress. A single test may iterate over many validation vectors while the runner reports only one aggregate verdict, obscuring which cases passed, failed, or were never reached. We run the selected PR tests before and after applying the upstream implementation, then freeze the individual cases that pass with the reference patch as grading targets.

The grader distinguishes failed cases from cases that never execute and retains both in the score denominator. To assess algorithm conformance beyond the selected PR tests, we use a separate evaluator that compares implementation outputs against published NIST Automated Cryptographic Validation Protocol (ACVP) vectors through drivers we wrote for each repository (National Institute of Standards and Technology 2026).

The correctness setting uses the PR task description and grades the frozen PR cases. The conformance setting adds the requirements and interfaces needed by the ACVP drivers and grades the assigned ACVP cases. We generate separate

This is an anonymized submission for review purposes only. Distribution, citation, or public sharing of this manuscript is strictly prohibited. Copyright and publication details will appear in the final version if accepted.

patches in the two settings and also run PR tests on conformance patches to compare the evaluators on the same code. For each setting, we report per-task pass rates and give every task equal weight in aggregate results.
We make the following contributions:

- **A repository-level PQC implementation benchmark.** We provide tasks for ML-KEM, ML-DSA, and SLH-DSA with reviewed specifications, validated reference implementations, and containerized, offline execution environments.
- **An evaluation framework for new cryptographic implementations.** We evaluate generated patches with fixed PR test cases and independent NIST ACVP vectors, reporting repository correctness and algorithm conformance separately.
- **An empirical study of coding agents and task specifications.** We evaluate three models using OpenHands and show that PR tests and ACVP vectors each detect failures missed by the other. An ablation that removes requirements and interfaces reduces GPT-6.1-Sol's mean ACVP pass rate from 100% to 29.0%, primarily because generated patches fail to build against the tests and drivers.

Background

Post-Quantum Cryptography Standards

FIPS 203, 204, and 205 standardize ML-KEM for key encapsulation and ML-DSA and SLH-DSA for digital signatures (National Institute of Standards and Technology 2024b,a,c). ML-KEM and ML-DSA build on module lattices and SLH-DSA on hash functions. Each standard fixes parameter sets and byte-exact encodings, so independent implementations can be compared on the same inputs.

The standards still leave decisions to each library, including seed or expanded key formats, context strings, pre-hash variants, hedged or deterministic signing, and implicit rejection of invalid ciphertexts. Libraries expose these choices through their own key types, interfaces, and build configuration.

Published ACVP vectors check the specified functions on fixed inputs (National Institute of Standards and Technology 2026), but they do not detect timing leaks. KyberSlash recovered secret keys from several Kyber implementations, including the reference code, through secret-dependent division timings (Bernstein et al. 2024). Machine-checked proofs, such as those for Kyber in Jasmin, give stronger guarantees for individual implementations (Almeida et al. 2023). Studies of open-source software describe how projects adopt PQC and the challenges they face (Dip, Fakorede, and Farooq 2026; Dip et al. 2026; Näther et al. 2024).

LLM Agents for Repository-Level Code

Repository-level benchmarks such as SWE-bench (Jimenez et al. 2024), SWE-Bench Pro (Deng et al. 2025), MobileDev-Bench (Fakorede et al. 2026), and GSO (Shetty et al. 2025) apply agent patches to real codebases and grade them with tests derived from upstream commits. SWE-Bench Pro also gives each task requirements and an interface that names the functions the tests call, a format we adopt.

AICrypto evaluates cryptographic knowledge, exploitation, and proof reasoning through multiple-choice questions, capture-the-flag challenges, and proof problems (Wang et al. 2026). For generated code, Elsayed, Fulton, and Yang (2026) study compilation and cryptographic misuse in Rust samples for AES-GCM and ChaCha20-Poly1305 using static analysis. Our tasks require changes to existing PQC libraries and evaluate their behavior with executable tests.

LLMs have also been applied to cryptographic code, including FHE programs (Yu et al. 2026; Kumar et al. 2026), cryptographic API misuse detection (Xia et al. 2026; Masood and Vargas Martin 2024), and post-quantum migration (Strauss et al. 2025; Pallarés de Bonrostro, González-Tablas, and González Vasco 2026; Alquwayfili 2026). PQCMigrate-Bench, the closest repository-level benchmark to our setting, grades agent patches with held-out tests from commit ranges that added a PQC algorithm (Upadhyay and Farooq 2026). PQCImplement-Bench complements this repository-test-based evaluation with independent ACVP vectors and separate correctness and conformance scores. Kumar et al. (2026) find that baseline agents write FHE code that compiles and passes functional tests while violating security constraints, and Alquwayfili (2026) finds migration patches that pass local verification but fail interoperability checks.

In hardware, Park et al. (2026) evaluate an agent-assisted ML-KEM/ML-DSA FPGA port using ACVP vectors, mutation testing, and formal checks. Their study identifies verification gaps despite passing tests. Our focus is software repository patches, with PR tests and ACVP vectors measuring integration correctness and conformance separately; neither score establishes cryptographic security.

Repository tests cover a standard only in part. A patch can pass them and still disagree with the standard on untested inputs, or compute the standard correctly through an interface the tests do not expect. Reporting the two evaluations separately makes these differences visible.

Methodology

Our pipeline has three stages (Figure 1). First, we construct repository tasks, each pairing a base revision with a model-visible specification: a problem statement, requirements, and interfaces. Second, reference runs on the base revision with the PR's tests, with and without its gold patch, establish the grading targets, which we freeze before any model is evaluated. Third, a model receives the base revision and specification and returns an implementation patch, which we grade against the frozen targets. In the correctness setting, we grade the patch on every PR grading target. The conformance setting adds the requirements of the NIST ACVP vector sets assigned to each task and grades the patch on every assigned ACVP case.

Candidate Discovery and Selection

The dataset contains 15 tasks from 14 PRs in nine repositories, covering ML-KEM, ML-DSA, and SLH-DSA, standardized in FIPS 203, 204, and 205, respectively (National Institute of Standards and Technology 2024b,a,c). Table 1 summarizes the collection and final dataset.

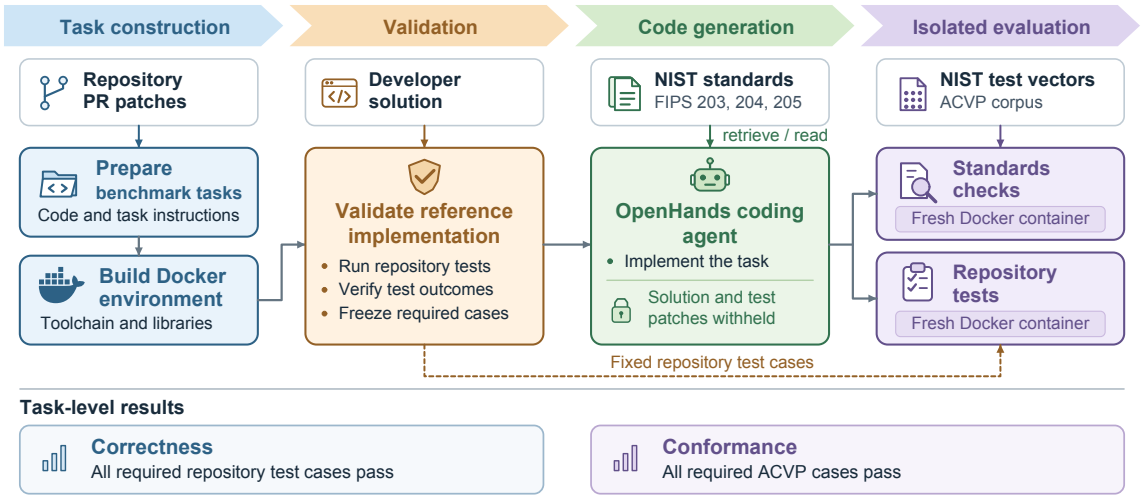


Figure 1: An overview of PQCImplement-Bench showing task construction, validation, code generation, and isolated evaluation of implementation correctness and ACVP conformance.

Measure	Count
Unique closed PRs retrieved	111,996
GitHub-recorded merges	58,646
Selected from recorded merges	11
Additional verified integrations	3
Selected PRs	14
Final tasks*	15
Repositories	9
Primary implementation languages	7
NIST algorithm families	3

Table 1: Dataset collection and composition. *OpenSSL #26715 adds ML-DSA and then ML-KEM; we split it into one task per algorithm.

We search GitHub for closed PRs whose titles, bodies, or comments contain any of nine NIST-standard terms: `ML-KEM`, `MLKEM`, `ML-DSA`, `MLDSA`, `SLH-DSA`, `SLHDSA`, `FIPS 203`, `FIPS 204`, and `FIPS 205`, with closure dates from January 1, 2008 through November 12, 2026. After deduplicating results by repository and PR number, the search yields 111,996 unique closed PR records, of which 58,646 have a GitHub-recorded merge (a non-null `mergedAt` field).

Task selection requires recoverable base revisions, separable implementation and test patches, executable evaluation environments, and passing reference targets. We select 11 PRs from the recorded merges. Three further PRs have no GitHub merge record but were integrated upstream: OpenJDK #21364 (ML-DSA) and #21478 (ML-KEM), which OpenJDK's integration bot pushed as commits `8b98f958` and `13987b42`, and OpenSSL #26715 (both families), which landed as a commit series whose commits, such as `0575755e`, identify the PR. This series adds ML-DSA first and then ML-KEM, so we split it at the upstream commit between the two halves (`1c2ac86`) into an ML-DSA task and an ML-KEM task. Each half reproduces its upstream commit exactly, and the ML-KEM task starts from a tree that already contains ML-DSA. Both OpenSSL builds first run OpenSSL's own `make errors` step, which regenerates error definitions from the checked-out source. We prepare all 15 tasks using the same criteria and validate their reference outcomes in the three execution phases described below.

Patch Split and Task Specification

Patch split. We separate the code a model must implement from the tests that assess it. Each task has a base commit, an implementation patch (the *gold patch*), and a test patch. We split each PR with a rule-based classifier. A changed file goes to the test patch if its path marks it as a test (for example, a `tests/` directory or a `_test.go` suffix) or its added lines contain an inline test marker; all other files go to the implementation patch. Modified Rust source files are split by hunk, so `#[cfg(test)]` blocks move to the test patch. We then review and correct the split by hand where needed: for the three Rust tasks, whose unit tests and test-support code share source files with the implementation, and for V and Pascal, whose test layouts the classifier does not recognize. For every task, we verify that applying the test patch and then the gold patch to the base revision reproduces the PR's source tree exactly.

Task specification. PR descriptions and linked issues are often brief or incomplete, so they are not sufficient as task descriptions. Following SWE-Bench Pro (Deng et al. 2025), we therefore write a model-visible specification for each task with three parts. The problem statement, written in the style of an issue, explains what the repository lacks, why the change is needed, and what to add. The requirements describe the expected behavior. The interfaces give the names, kinds, repository-relative locations, and callable contracts of the code to be written. No part of the specification contains implementation code or expected test outputs. A second author reviews each specification. This gives two task sets: PR

			Gold patch			Graded cases	
Algorithm	Repository / PR	Language	Files	Added	Removed	PR cases	ACVP cases
ML-KEM	CIRCL #470	Go	18	1,484	27	180	435
	OpenJDK #21478	Java	6	1,887	1	180	435
	OpenSSL #26715 (b)	C	86	8,002	331	7	435
	Orion #431	Rust	13	2,808	1	180	435
	Botan #3893	C++	41	1,333	308	225	435
ML-DSA	CIRCL #480	Go	82	2,454	2,171	135	1,335
	liboqs #1919	C	225	3,513	1,799	180	1,335
	OpenJDK #21364	Java	5	1,496	11	180	1,335
	OpenSSL #26715 (a)	C	67	7,854	33	47	1,335
	Orion #654	Rust	28	4,876	45	336[†]	1,335
	RustCrypto #877	Rust	16	2,828	40	180	1,335
	V #26711	V	10	1,992	0	64	1,335
	CryptoLib4Pascal #115	Pascal	83	12,995	1,318	4,158	1,335
SLH-DSA	CIRCL #512	Go	14	1,780	5	31	1,248
	liboqs #2175	C	211	36,706	314	4,908	1,248
Total			905	92,008	6,404	10,991	15,351

Table 2: Benchmark tasks. Gold patch counts the files and lines the reference implementation changes, excluding the PR's tests. Graded cases count executions of the PR's developer tests and of the assigned NIST ACVP cases. OpenSSL #26715 is split at upstream commit `1c2ac86` into (a) ML-DSA and (b) ML-KEM. [†]The gold run skips 279 more cases, which are not graded.

tasks, which describe the PR's change and are used in the correctness setting, and PR-and-ACVP tasks, which add the requirements and interfaces needed by the task's assigned ACVP vector sets and are used in the conformance setting.

Execution Environments and Reference Phases

Evaluation environment. We build a container image for each task at the base revision, with the required toolchains and dependencies. Every evaluation starts in fresh containers and records the image identity, commit, patch fingerprints, commands, and logs. Builds and tests run offline; where Cargo dependencies are needed, we fetch them first and then disconnect the network. As a check, we build all 15 images and run each repository's existing tests at the base revision, offline and without the PR patches: every build succeeds and every test passes. These existing tests are not graded.

Reference phases. Let B_i, T_i, and G_i denote task i's base revision, test patch, and gold patch. We establish reference outcomes in three phases, where $+$ denotes patch application:

Phase	Repository state
R1	Base revision B_i
R2	Base with test patch $B_i + T_i$
R3	Base with test and gold patches $B_i + T_i + G_i$

In every phase, a reporting overlay (described in the next subsection) makes the selected tests report individual cases when their source is present; it does not add newly introduced tests to R1. A model patch M_i is evaluated as $B_i + T_i + M_i$, with the same overlay and case catalog as the reference and without the gold patch.

Case-Level PR Test Grading

Case splitting. Native test runners often report a whole loop of test vectors as a single test. We instrument these loops with a reporting overlay so that each vector, scenario, or trial receives its own verdict. Before any run, we enumerate every case from the frozen test sources and fixtures into a case catalog, which gives each case a stable identifier and links it to its parent test, source and input fingerprints, and operation. Cases from ACVP-formatted fixtures keep their group and case identifiers and vector-set context; other tests keep their native scenario and trial names. Parent-test results are not counted again as cases, and assertions within one scenario are not split into separate cases by default.

Case outcomes. The overlay does not change what the tests check: it keeps the original implementation calls and assertions and only reports their results case by case. Every case ends with one of five outcomes: pass, fail, skip, error, or not run. An error means the case started but stopped without a verdict, for example after a crash or timeout. Not run means the case never started. This happens when a patch fails to build, or when a test stops at its first failure and never reaches its remaining cases. A case that is not run counts as not passed. We also discard any run whose results do not match the case catalog, are malformed, or contradict each other.

Randomized tests. Some PR tests in Pascal, liboqs-2175, and OpenSSL draw random inputs. Gold and candidate runs of these tests use the same versioned entropy fixtures, generated outside the evaluated implementation from a public seed and operation-specific domains. This fixes the test inputs without replacing the cryptographic implementation or adding another source of expected outputs. Each repeated trial counts as a separate case.

Grading Targets and Scores

Grading targets. We choose the cases to grade by comparing each case's status in R2 (tests added, no implementation) with its status in R3 (tests and gold implementation). Let $s_{i,r}(c)$ be the status of case c of task i in phase r. A case is a target only if it passes in R3, and we group targets by their R2 status:

$$\begin{aligned} F_i &= \{c : s_{i,2}(c) = \text{fail},\ s_{i,3}(c) = \text{pass}\}, \\ N_i &= \{c : s_{i,2}(c) = \text{not run},\ s_{i,3}(c) = \text{pass}\}, \\ P_i &= \{c : s_{i,2}(c) = \text{pass},\ s_{i,3}(c) = \text{pass}\}. \end{aligned}$$

Cases in F_i and N_i test new behavior, and we write $A_i = F_i \cup N_i$ for them; cases in P_i test behavior the patch must keep. All three groups are graded and reported separately. R1 is recorded for context only and removes no targets. We include N_i because a new feature's tests often cannot run before the feature exists: in compiled repositories, the missing implementation stops the test suite from building, so these tests never produce the observed failure that fail-to-pass selection as in SWE-bench (Jimenez et al. 2024) requires.

Freezing. We compute these sets once, from the verified reference runs, and keep them fixed for every candidate. Each reference record stores fingerprints of the case catalog, reporting overlay, patches, image, and archived phase logs, and the grader refuses to score if any of them differ. The manifest keeps the SWE-bench field names, storing A_i as `FAIL_TO_PASS` and P_i as `PASS_TO_PASS`.

Current targets. The catalogs contain 11,270 cases. The gold patch passes 10,991 of them in R3 and skips 279 in Orion-654; skipped cases appear in reports but are not targets. All 10,991 targets are in N_i, so F_i and P_i are currently empty. These counts include repeated trials, so they count executions rather than distinct NIST vectors. Table 2 gives the counts per task.

Scores. In the correctness setting, the score for task i is its PR-case pass rate

$$p_i = \frac{|\{c \in A_i \cup P_i : s(c) = \text{pass}\}|}{|A_i \cup P_i|}, \tag{1}$$

where $s(c)$ is the candidate's status. A target stays in the denominator whether it fails, is skipped, errors, or does not run, so a patch that does not apply or build scores zero. Before grading, changes to existing test files are removed and flagged; new test files are kept, so that a build referencing them still works, but they are not graded. The PR evaluator rejects any remaining change to frozen tests or fixtures. In the conformance setting, the score is the ACVP pass rate a_i: the fraction of the task's assigned ACVP cases whose outputs match the expected values, as described in the next subsection. The ACVP drivers call the candidate's own code, so a conformance patch must still implement the whole task. We also run its PR tests, but only as a diagnostic of whether adding the ACVP interfaces changed PR behavior; they do not affect a_i.

We report p_i and a_i for every task, with breakdowns by operation and group, and summarize each setting by the mean over the 15 tasks, $\bar{p} = \frac{1}{15}\sum_i p_i$ and $\bar{a} = \frac{1}{15}\sum_i a_i$. Each task counts equally, even though Pascal and liboqs-2175 have many more trials. If grading fails because of an infrastructure error, the attempt is regraded before results are reported.

Independent ACVP Evaluation

Evaluator. Our second evaluator uses NIST Automated Cryptographic Validation Protocol (ACVP) test vectors, pinned to a fixed revision of the public corpus (National Institute of Standards and Technology 2026). It does not use the PR's tests or grading targets. For each task, a repository-specific driver runs in a fresh container and passes the prompt vectors to the implementation; the expected answers stay on the host. Drivers may translate input formats, control randomness, and prepare the encodings declared by the task's interfaces, but the implementation under test performs every cryptographic operation. The grader matches each response to its vector set, group, and case, and compares the expected output fields exactly.

Drivers. ACVP vectors are test files, not programs, so we wrote drivers for each of the 9 repositories that run them, independently of any ACVP tests the projects ship. A driver is written in the repository's language, copied into the checkout, and built with the repository's own toolchain inside the evaluation container; it reads the prompt file, calls the implementation, and writes its outputs in the ACVP response format. A configuration for each of the 45 task–vector-set pairs names the driver, its build and run commands, and the test groups it covers; these group lists record which cases each original reference can run. Drivers call only the interfaces declared in the task description, so the same driver evaluates the reference and any candidate that follows those interfaces.

Case counts. Each of the 15 tasks implements one algorithm profile and is assigned ACVP vector sets for that algorithm, which supply 435 ML-KEM, 1,335 ML-DSA, or 1,248 SLH-DSA cases. Across all tasks, this gives 15,351 case evaluations. For each task, we report the assigned cases, the verdict for each case, and the reason for every failure or non-execution. These counts are separate from the 10,991 PR grading targets. Because PR fixtures and the ACVP corpus may share vectors, we do not add the two totals as distinct test coverage.

Original reference coverage. The original upstream implementations do not expose every interface the drivers need; for example, most ML-DSA references lack the internal signing interface that some ACVP tests call. The original references can run 10,230 of the 15,351 cases and pass all of them. The other 5,121 need interfaces, key representations, or implementation changes that the original PRs do not provide; the conformance task adds them. Table 3 gives the breakdown by algorithm.

Conformance setting. The conformance setting closes this gap. Its expanded task description states the additional requirements, and matching drivers evaluate all 15,351 assigned cases; cases that fail or cannot run stay in the denom-

Algorithm	Tasks	Assigned	Interfaces	
			Original	Added
ML-KEM	5	2,175	1,845	330
ML-DSA	8	10,680	6,225	4,455
SLH-DSA	2	2,496	2,160	336
Total	15	15,351	10,230	5,121

Table 3: ACVP cases by algorithm. The conformance setting grades all assigned cases. *Original* counts cases that the upstream PR implementation can run through the interfaces it already exposes; all of them pass. *Added* counts cases that become runnable only through the interfaces, key formats, or adapters the conformance task adds. With the additional reference patches, all 15,351 cases execute and pass.

inator. For 3,846 of the 5,121 added cases, the driver only prepares inputs for operations the reference already has, for example by deriving an ML-DSA key from its seed with the library's own key generation. The other 1,275 need reference patches for nine tasks. Six are based on changes the maintainers later made upstream, and we wrote the other three because upstream has no equivalent. We withhold these patches and adapters from candidates. With them, the references execute and pass all 15,351 assigned cases. Each candidate supplies its own implementation of the whole task, and the evaluator applies no reference patch. We keep the results for the original references, the expanded references, and the candidates separate.

Evaluator Validation

To check that the evaluator detects failures as well as successes, we use deliberately failing controls and run the gold patch through the candidate evaluation path. We replay stored reference evidence and verify source and input fingerprints to check consistency with the frozen catalogs. These runs validate the evaluator. Per-case outcomes and raw logs allow each aggregate result to be traced to its underlying executions.

Experimental Setup

Agent. We run each model as an OpenHands agent (Wang et al. 2025) (SDK 1.62.0). The agent has a shell terminal, a file viewer and editor, a task tracker for its plan, and a finish action; with specification access, it also has a `spec_search` tool. The system prompt names the target standards (FIPS 203, 204, and 205), warns that they differ from the round submissions and initial drafts, and asks the agent to build often and run the project's own tests. It also states that the machine is offline, that the work is graded with hidden project tests (and, in the conformance setting, with NIST ACVP vectors run through the task's interface), and that edits to existing tests are discarded. It does not say which requirements are graded. The task prompt is the task description for the setting. If the agent stops without calling finish, it receives up to three prompts asking it to continue on its own.

Environment. Each attempt runs in its own container, built from a scrubbed copy of the task image. The gold patch, test patch, and later history are removed, and the repository is replaced by a fresh git repository with a single commit of the base files; two probes check that no hidden material remains. The agent has no internet access. It can reach only a per-attempt gateway, which forwards model calls to OpenRouter and specification searches to the retrieval service, holds the API key, enforces the limits below, and logs every request. Each attempt records its configuration, limits, trajectory, and resulting patch, which is then graded as described in Section .

Specification access. The `spec_search(query, k)` tool searches the final FIPS documents. A retrieval service matches the query against page images using ColQwen2 (Faysse et al. 2025) and returns up to three pages, each as a page image with its extracted text, document name, and page number. Each task can search only the standards it implements, and each attempt is limited to 25 searches. The documents are hash-pinned, and a run does not start unless the search index matches them.

Models and limits. We evaluate three models through OpenRouter: GPT-6.1-Sol, Kimi K3.1 Pro, and GLM-5.3-FlashX. Each model is pinned to its developer's own endpoint (OpenAI, Moonshot AI, and Z.AI), and keeps the provider's default temperature. GPT-6.1-Sol and Kimi K3.1 Pro use medium reasoning effort. GLM-5.3-FlashX offers only low, high, and max effort, so we use high, the setting closest to medium.Every attempt has the same limits: 150 agent steps, 3 hours of wall time, 20 minutes per shell command, and $4 of model cost. OpenHands' summarizing condenser compresses the history once it exceeds 75 steps or 400k tokens. We chose the reasoning effort and condenser thresholds in a pilot on liboqs-2175: at high effort, most of the cost went to reasoning tokens, and condensing more often made the agent re-read files it had already seen. We run one attempt per model, task, and setting. Infrastructure failures, such as a container or gateway fault, are retried up to three times and are never scored.

Experiments. The main experiment runs all three models in both settings on all 15 tasks, with specification access. A *statement-only* ablation uses GPT-6.1-Sol in the conformance setting and is compared with its main conformance results. It keeps `spec_search` but gives only the problem statement, without the requirements and interface sections; the prompts then ask the agent to follow the project's public API instead of a declared interface. The hidden tests and ACVP drivers still call the declared interface, so this ablation measures how much the requirements and interface sections contribute.

Results

Main results. Table 4 summarizes the main experiment, and Table 5 gives every task. GPT-6.1-Sol reaches $\bar{a} = 100\%$: its conformance patches answer all 15,351 assigned ACVP cases correctly. In the correctness setting it reaches $\bar{p} = 93.7\%$, with 12 of 15 tasks at 100%. Its three misses

	Correctness		Conformance			Agent		
Model	$\bar{p}$ (%)	p_i=1	$\bar{a}$ (%)	a_i=1	PR (%)	$/attempt	Steps	At limit
GPT-6.1-Sol	93.7	12	100.0	15	99.4	1.54	49	2/30
Kimi K3.1 Pro	52.4	6	55.8	6	51.7	2.32	74	8/30
GLM-5.3-FlashX	0.0	0	6.7	0	0.1	1.04	147	30/30

Table 4: Main results, one attempt per model, task, and setting. $\bar{p}$ and $\bar{a}$ are mean PR-case and ACVP pass rates, and p_i=1 and a_i=1 count fully passing tasks. Conformance PR is the diagnostic PR rate of the conformance patches. Agent columns give mean cost per attempt, median steps, and attempts stopped by the 150-step or $4 limit.

are OpenSSL-26715-b (1 of 7 PR cases), OpenSSL-26715-a (46 of 47), and Orion-654 (315 of 336). Kimi K3.1 Pro fully passes six tasks in each setting, with $\bar{p} = 52.4\%$ and $\bar{a} = 55.8\%$; it is strongest on ML-KEM ($\bar{a} = 87.6\%$ over five tasks) and weakest on ML-DSA (37.5% over eight). GLM-5.3-FlashX passes no PR case in the correctness setting ($\bar{p} = 0.0\%$) and fully passes no task. Its only ACVP credit comes from partial results on OpenJDK-21364 (67.4%), CIRCL-470 (11.5%), liboqs-1919 (10.8%), and V-26711 (10.8%), for $\bar{a} = 6.7\%$.

How the patches fail. We label a result below 100% a *build failure* when the PR tests or the ACVP driver do not compile against the patch, and a *test failure* when they compile but cases fail. GPT-6.1-Sol has no build failures. Kimi K3.1 Pro's misses are mostly test failures (6 of 9 PR results in the correctness setting and 8 of 9 ACVP results in conformance), so its code usually builds against the declared interfaces but computes wrong results. GLM-5.3-FlashX's misses are mostly build failures: 9 of 15 PR results in the correctness setting and 10 of 15 ACVP results. Its OpenSSL-26715-a attempts left no changes in either setting.

PR tests and ACVP vectors disagree. The two evaluators catch different failures, even on the same patch. Kimi K3.1 Pro's liboqs-2175 conformance patch passes all 4,908 PR cases but only 668 of 1,248 ACVP cases (53.5%). In the other direction, its OpenSSL-26715-b conformance patch passes every ACVP case but none of the 7 PR cases, and GPT-6.1-Sol's OpenSSL-26715-a and Orion-654 conformance patches pass every ACVP case while still missing 1 and 21 PR cases. The PR tests check repository behavior that ACVP does not exercise, and ACVP vectors expose computation errors that the selected PR tests miss. Reporting only one evaluator would overstate both kinds of patch.

Ablation. Table 6 compares GPT-6.1-Sol's conformance results with its statement-only ablation. With the problem statement alone, $\bar{a}$ falls from 100% to 29.0% and the diagnostic PR rate from 99.4% to 18.9%, and only four tasks keep full ACVP credit. Twenty of the 24 results below 100% are build failures: without the interface section, the agent names types, functions, and files differently from what the tests and drivers call. Omitting the requirements and interfaces therefore mainly causes compatibility failures. Because those patches never reach the graded cases, the ablation does not show whether their underlying implementations were correct.

Agent behavior and cost. GPT-6.1-Sol finishes 28 of its 30 attempts on its own, with a median of 49 steps and a mean cost of $1.54 per attempt (median $1.22); the other two stop at the cost limit. Kimi K3.1 Pro takes a median of 74 steps at a mean cost of $2.32 per attempt (median $2.00), and 8 of its 30 attempts stop at the step or cost limit. GLM-5.3-FlashX never calls finish: all 30 attempts run until the 150-step limit (median 147 steps), at a mean cost of $1.04 per attempt (median $0.98). The main experiment cost $146.92 in total. One attempt needed a grading decision: the ACVP grader marked Kimi K3.1 Pro's Pascal-115 conformance attempt as an infrastructure error because the driver's output was not valid JSON. The cause was two debug print statements that the patch left in its ML-DSA signing and verification code, so we count all 1,335 of its ACVP cases as failed rather than regrading it.

Discussion

Correctness and conformance need separate evidence. Passing repository tests does not establish conformance to the standard, and passing ACVP cases does not establish compatibility with the repository. Kimi K3.1 Pro's liboqs-2175 conformance patch illustrates the first gap: it passes every PR case but only 53.5% of its ACVP cases. Reporting both evaluations makes these differences visible and helps identify what remains to be fixed. Similar gaps between local checks and broader requirements appear in FHE code generation and PQC migration (Kumar et al. 2026; Alquwayfili 2026). Neither evaluation establishes security properties such as constant-time execution.

Feature tasks need not-run-to-pass targets. Every frozen PR target in this dataset is not run before the implementation is added and passes with the gold patch. Requiring an observed test failure would exclude these targets because missing implementation code prevents the tests from building. Feature benchmarks therefore need to distinguish non-execution from an executed failure while keeping both eligible for grading when the reference passes. Freezing the targets before model evaluation prevents a candidate from reducing its denominator by leaving tests unbuildable. Individual-case reporting then reveals partial progress that a binary task-resolution score would hide.

Task contracts shape what is measured. The statement-only failures support making test-required contracts explicit, so that evaluation can reach the implementation's behavior. The ablation removes requirements and interfaces together,

		Cases		GPT-6.1-Sol		Kimi K3.1 Pro		GLM-5.3-FlashX	
Algorithm	Task	PR	ACVP	PR	ACVP	PR	ACVP	PR	ACVP
ML-KEM	CIRCL-470	180	435	100	100	100	100	0	11.5
	OpenJDK-21478	180	435	100	100	100	37.9	0	0
	OpenSSL-26715-b	7	435	14.3	100	0	100	0	0
	Orion-431	180	435	100	100	100	100	0	0
	Botan-3893	225	435	100	100	33.3	100	0	0
ML-DSA	CIRCL-480	135	1,335	100	100	100	100	0	0
	RustCrypto-877	180	1,335	100	100	58.3	0	0	0
	Pascal-115	4,158	1,335	100	100	0	0	0	0
	liboqs-1919	180	1,335	100	100	100	100	0	10.8
	OpenJDK-21364	180	1,335	100	100	0	10.8	0	67.4
	OpenSSL-26715-a	47	1,335	97.9	100	0	5.6	0	0
	Orion-654	336	1,335	93.8	100	93.8	16.4	0	0
	V-26711	64	1,335	100	100	100	67.4	0	10.8
SLH-DSA	CIRCL-512	31	1,248	100	100	0	45.4	0	0
	liboqs-2175	4,908	1,248	100	100	0	53.5	0	0
Mean pass rate (%)				93.7	100.0	52.4	55.8	0.0	6.7
Tasks fully passed				12	15	6	6	0	0

Table 5: Per-task results of the main experiment. Cases gives each task's number of PR grading targets and assigned ACVP cases. For each model, PR is the PR-case pass rate (%) of its correctness patch, and ACVP is the ACVP pass rate (%) of its separate conformance patch. Means weight each task equally.

	ACVP		Diagnostic PR		Failures	
Variant	$\bar{a}$	$a_i{=}1$	Mean	Full	Build	Test
Main	100.0	15	99.4	13	0	2
Statement only	29.0	4	18.9	2	20	4

Table 6: GPT-6.1-Sol conformance ablation (rates in %). Statement only omits the requirements and interfaces from the task text. Build and Test count failed evaluations over both evaluators.

however, so it does not isolate their individual contributions. The full-task setting therefore measures implementation against a declared contract, including some repository-internal interfaces, and leaves less freedom to design the API.

Performance is conditional on the setup. Model comparisons reflect the chosen agent, reasoning settings, and resource limits. GLM-5.3-FlashX reaches the step limit on every attempt, so its results establish performance within that budget, not what additional time would achieve. Reporting build failures, executed failures, and stopping reasons alongside pass rates helps separate these constraints from algorithmic errors.

Limitations and Threats to Validity

The 15 tasks are selected for executable evaluation across three PQC algorithm families. Results apply to this dataset and its tested requirements. Some PR tests require internal interfaces that constrain implementation choices beyond the NIST standards. Individual-case totals include repeated trials and overlapping PR and ACVP inputs, so they do not represent independent or unique vectors; task-level scoring gives each task equal weight.

The evaluation measures functional behavior on finite inputs. It does not assess constant-time execution or side-channel resistance, and passing does not constitute a security proof or certification. Upstream patches and vectors are public, so prior model exposure remains a potential confound.

Conclusion

Implementing NIST's post-quantum standards in existing cryptographic libraries requires both correct algorithm behavior and compatibility with repository interfaces. PQCImplement-Bench evaluates these requirements through 15 tasks across nine repositories, covering ML-KEM, ML-DSA, and SLH-DSA. We combine validated reference runs with separate PR-test and NIST ACVP evaluation. In our study of three models using OpenHands, we find patches that pass every selected PR test while failing ACVP cases, and others that pass ACVP checks while failing repository tests. With explicit requirements and interfaces, GPT-6.1-Sol passes every assigned ACVP case. Removing those sections reduces its mean pass rate to 29.0%, mainly because the resulting patches fail to build against the tests and drivers. Evaluating generated cryptographic code therefore requires distinguishing failures to satisfy an interface from failures in the algorithm's behavior. The released tasks, validated environments, and evaluators let researchers compare coding agents under consistent requirements, examine partial progress, and detect implementation errors that repository tests alone would miss.

References

Almeida, J. B.; Barbosa, M.; Barthe, G.; Grégoire, B.; Laporte, V.; Léchenet, J.-C.; Oliveira, T.; Pacheco, H.; Quaresma, M.; Schwabe, P.; Séré, A.; and Strub, P.-Y. 2023. Formally verifying Kyber Episode IV: Implementation Correctness. Cryptology ePrint Archive, Paper 2023/215.

Alquwayfili, A. 2026. Can Coding Agents Migrate to Post-Quantum Cryptography? arXiv:2512.12989.

Bernstein, D. J.; Bhargavan, K.; Bhasin, S.; Chattopadhyay, A.; Chia, T. K.; Kannwischer, M. J.; Kiefer, F.; Paiva, T.; Ravi, P.; and Tamvada, G. 2024. KyberSlash: Exploiting secret-dependent division timings in Kyber implementations. Cryptology ePrint Archive, Paper 2024/1049.

Deng, X.; Da, J.; Pan, E.; He, Y. Y.; Ide, C.; Garg, K.; Lauffer, N.; Park, A.; Pasari, N.; Rane, C.; Sampath, K.; Krishnan, M.; Kundurthy, S.; Hendryx, S.; Wang, Z.; Bharadwaj, V.; Holm, J.; Aluri, R.; Zhang, C. B. C.; Jacobson, N.; Liu, B.; and Kenstler, B. 2025. SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks? arXiv:2509.16941.

Dip, S. S.; Fakorede, M. A.; and Farooq, U. 2026. PQC-Corpus: A Dataset of Linked Development Histories for Post-Quantum Cryptography. arXiv:2611.10966.

Dip, S. S.; Fakorede, M. A.; Upadhyay, K.; and Farooq, U. 2026. Understanding Post-Quantum Cryptography Adoption: Practices, Challenges, and Evolution in Open-Source Software. arXiv:2610.38054.

Elsayed, M.; Fulton, K.; and Yang, J. 2026. An Empirical Security Evaluation of LLM-Generated Cryptographic Rust Code. arXiv:2604.27001.

Fakorede, M. A.; Upadhyay, K.; Siddique, A. B.; and Farooq, U. 2026. MobileDev-Bench: A Benchmark for Issue Resolution in Mobile Application Development. arXiv:2603.24946.

Faysse, M.; Sibille, H.; Wu, T.; Omrani, B.; Viaud, G.; Hudelot, C.; and Colombo, P. 2025. ColPali: Efficient Document Retrieval with Vision Language Models. In *International Conference on Learning Representations (ICLR)*.

Jimenez, C. E.; Yang, J.; Wettig, A.; Yao, S.; Pei, K.; Press, O.; and Narasimhan, K. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:2310.06770.

Kumar, M.; Xue, J.; Zheng, M.; and Lou, Q. 2026. FHE-Coder: Benchmarking Secure Agentic Code Generation for Fully Homomorphic Encryption. In *The Fourteenth International Conference on Learning Representations*.

Masood, Z.; and Vargas Martin, M. 2024. Beyond Static Tools: Evaluating Large Language Models for Cryptographic Misuse Detection. arXiv:2411.09772.

Moody, D.; Perlner, R.; Regenscheid, A.; Robinson, A.; and Cooper, D. 2026. Transition to Post-Quantum Cryptography Standards. NIST Internal Report 8547, National Institute of Standards and Technology, Gaithersburg, MD.

National Institute of Standards and Technology. 2024a. Module-Lattice-Based Digital Signature Standard. Technical Report FIPS 204, National Institute of Standards and Technology.

National Institute of Standards and Technology. 2024b. Module-Lattice-Based Key-Encapsulation Mechanism Standard. Technical Report FIPS 203, National Institute of Standards and Technology.

National Institute of Standards and Technology. 2024c. Stateless Hash-Based Digital Signature Standard. Technical Report FIPS 205, National Institute of Standards and Technology.

National Institute of Standards and Technology. 2026. ACVP-Server: Published Algorithm Validation Vectors. Source repository. Pinned revision 975de31eb83d87039ec88934fdc47d8c312b892d, August 12, 2026.

Näther, C.; Herzinger, D.; Gazdag, S.-L.; Steghöfer, J.-P.; Daum, S.; and Loebenberger, D. 2024. Migrating Software Systems Toward Post-Quantum Cryptography-A Systematic Literature Review. *IEEE Access*, 12: 132107–132126.

Pallarés de Bonrostro, J.; González-Tablas, A. I.; and González Vasco, M. I. 2026. Empirical Evaluation of Large Language Models for Migration of Code Fragments to Post-Quantum Cryptography. arXiv:2606.07341.

Park, J.; Kim, E.; Kim, W.; Cho, S.; and Cha, B. 2026. From FIPS to FPGA with LLM Agents: Measuring What Tests and Reviews Miss in Post-Quantum Cryptographic Hardware. arXiv:2611.25339.

Shetty, M.; Jain, N.; Liu, J.; Kethanaboyina, V.; Sen, K.; and Stoica, I. 2025. GSO: Challenging Software Optimization Tasks for Evaluating SWE-Agents. arXiv:2505.23671.

Shor, P. W. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing*, 26(5): 1484–1509.

Strauss, J.; Upadhyay, K.; Siddique, A. B.; Baggili, I.; and Farooq, U. 2025. Assessing and Enhancing Quantum Readiness in Mobile Apps. arXiv:2506.00790.

Upadhyay, K.; and Farooq, U. 2026. Poster: Evaluating LLM Agents for Post-Quantum Cryptographic Migrations. In *Proceedings of the 2026 ACM Conference on Post-Quantum and Quantum-Based Security*, PQQS '26, 329–330. New York, NY, USA: Association for Computing Machinery. ISBN 9798400729201.

Wang, X.; Li, B.; Song, Y.; Xu, F. F.; Tang, X.; Zhuge, M.; Pan, J.; Song, Y.; Li, B.; Singh, J.; Tran, H. H.; Li, F.; Ma, R.; Zheng, M.; Qian, B.; Shao, Y.; Muennighoff, N.; Zhang, Y.; Hui, B.; Lin, J.; Brennan, R.; Peng, H.; Ji, H.; and Neubig, G. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *International Conference on Learning Representations (ICLR)*.

Wang, Y.; Liu, Y.; Ji, L.; Luo, H.; Li, W.; Zhou, X.; Feng, C.; Wang, P.; Cao, Y.; Zhang, G.; Li, X.; Xu, R.; Chen, Y.; and He, T. 2026. AICrypto: Evaluating Cryptography Capabilities of Large Language Models. arXiv:2507.09580v6.

Xia, Y.; Xie, Z.; Liu, P.; Lu, K.; Liu, Y.; Wang, W.; and Ji, S. 2026. Beyond Static Pattern Matching? Rethinking Automatic Cryptographic API Misuse Detection in the Era of LLMs. arXiv:2407.16576.

Yu, Q.; Li, G.; Zhang, S.; Xu, R.; Cai, Q.; Xue, J.; Feng, X.; and Cui, H. 2026. FHE-CodeEval: Benchmarking Large Language Models for Fully Homomorphic Encryption Code Generation. In Lee, H.-Y.; Dev, S.; Li, J.; and Elliott, D., eds., *Findings of the Association for Computational Linguistics: EMNLP 2026*, 12657–12673. Budapest, Hungary: Association for Computational Linguistics. ISBN 979-8-89176-462-0.