CHECKLIST

Algorithms Implementation Roadmap Checklist

A practical checklist for turning an algorithm from concept to production. Use it to verify fit, implementation quality, validation evidence, deployment readiness, and operational ownership before release.

Programming - July 04, 2026

Purpose

The practical problem is not whether an algorithm works in theory, but whether it can be implemented safely, validated correctly, and operated reliably in your environment. A weak implementation roadmap often leads to hidden complexity, missed edge cases, poor performance under real data, or production failures that are expensive to roll back.

This checklist helps you turn an algorithms implementation roadmap into a verifiable plan. After using it, you should be able to decide whether the approach fits the problem, apply a structured implementation workflow, validate correctness and performance, and confirm what must be checked before production use.

How to use this checklist

Use this checklist during design review, implementation, pre-production validation, and release approval. Treat every checkbox as a verifiable action, not a general reminder. Capture evidence as you go: test results, review notes, benchmarks, assumptions, and ownership decisions.

If an item cannot be proven, mark it incomplete. For a practical companion to the first two phases, you can also compare this workflow with How to get started with algorithms and use Algorithms Checklist: Verify Correctness, Performance, and Safety Before Production for a focused production-readiness review.

Phase 1: Problem fit and implementation scope

Purpose: Confirm the algorithm is the right tool before any code is written. This phase prevents teams from optimizing the wrong thing or adopting an approach that cannot meet operational constraints.

Checklist items

- ? Confirm the problem statement, input domain, output contract, and success criteria in writing.
- ? Review whether the algorithm?s assumptions match the real data distribution, traffic pattern, or threat model.
- ? Validate that the expected accuracy, latency, memory, or determinism requirements are explicit and measurable.
- ? Document the failure modes that matter most, including incorrect output, timeouts, resource exhaustion, or unsafe behavior.
- ? Assign an owner for implementation, validation, and operational sign-off.
- ? Review whether a simpler or safer approach satisfies the same requirement with less operational risk.

Evidence to capture

- Problem statement and acceptance criteria.
- Input/output examples, including edge cases.
- Assumption list and known constraints.

- Initial complexity estimate and risk notes.
- Ownership record and review participants.

Acceptance criteria

- The algorithm solves a clearly stated problem.
- Constraints are explicit and testable.
- The approach is justified against at least one alternative.
- A responsible owner is named for each phase.

Common mistakes

- Selecting an algorithm because it is familiar rather than appropriate.
- Treating average-case behavior as sufficient when worst-case behavior matters.
- Ignoring data skew, adversarial input, or resource caps.
- Leaving success criteria vague, which makes validation non-verifiable.

Owner and review cadence

- Owner: Technical lead or algorithm implementer.
- Review cadence: At project start and whenever requirements, data shape, or constraints change.

Phase 2: Design and implementation plan

Purpose: Translate the algorithm into a concrete implementation plan that is testable, reviewable, and safe to build.

Checklist items

- ? Review the chosen data structures, control flow, and invariants before coding.
- ? Validate that the algorithm can be expressed within the language, runtime, and dependency constraints in use.
- ? Document the expected time and space complexity for best, average, and worst cases.
- ? Assign coding responsibilities for core logic, test harnesses, instrumentation, and integration points.
- ? Confirm how the implementation will handle invalid input, empty input, boundary values, and partial failures.
- ? Document any nondeterministic behavior and the method used to make tests repeatable.
- ? Review whether concurrency, parallelism, or distributed execution changes correctness requirements.

Evidence to capture

- Design notes or implementation sketch.
- Complexity analysis with assumptions.
- Interface definition and error-handling rules.
- Dependency list, including library or runtime version dependencies.
- Review comments on invariants and edge cases.

Acceptance criteria

- The design maps cleanly to code without unresolved ambiguities.
- Complexity is understood and within target limits.
- Error paths and boundary conditions are defined.
- Dependencies and version constraints are recorded.

Common mistakes

- Starting implementation before invariants are documented.
- Assuming libraries will preserve algorithmic guarantees without checking their behavior.
- Forgetting that concurrency can break ordering or state assumptions.
- Skipping explicit handling for malformed or adversarial input.

Owner and review cadence

- Owner: Implementer with review by a peer engineer.
- Review cadence: Before implementation begins and again when the design changes.

Phase 3: Code implementation and local verification

Purpose: Build the algorithm in a way that preserves the intended logic and exposes defects early through local checks.

Checklist items

- ? Confirm the code follows the documented invariants and data flow.
- ? Validate that helper functions have narrow responsibilities and testable outputs.
- ? Test the implementation with representative samples, boundary values, and intentionally malformed inputs.
- ? Review whether assertions, guards, or type checks are needed to protect critical assumptions.
- ? Document any deviations from the initial design and the reason for each change.
- ? Validate that logging or tracing does not expose sensitive data or distort performance materially.

- ? Assign a reviewer to check the code path for logic gaps, dead branches, and hidden mutation.

Evidence to capture

- Source code diff.
- Unit test results.
- Local run output for representative cases.
- Review notes on deviations and assumptions.
- Logging or tracing examples if instrumentation is added.

Acceptance criteria

- Core logic matches the design and passes targeted unit tests.
- Boundary cases are covered and produce expected outcomes.
- Deviations are explained and approved.
- Observability does not violate security or performance requirements.

Common mistakes

- Overfitting code to one sample dataset.
- Using mutable shared state without proving it is safe.
- Treating debug output as a substitute for tests.
- Leaving unhandled exceptions in expected error paths.

Owner and review cadence

- Owner: Implementer.

- Review cadence: On each significant code change and before merging to the main branch.

Phase 4: Correctness validation

Purpose: Prove the algorithm produces the right result for the right inputs, including edge cases and known failure patterns.

Checklist items

- ? Validate the implementation against a reference result, oracle, or hand-calculated examples where possible.
- ? Test all documented boundary cases, including zero values, minimum values, maximum values, and empty inputs.
- ? Review whether property-based tests, invariant checks, or randomized tests improve coverage.
- ? Confirm that error conditions fail safely and predictably.
- ? Validate that outputs remain stable across repeated runs when determinism is required.
- ? Document any cases where approximate results are acceptable and the tolerated error bounds.
- ? Assign a reviewer to confirm that the test set covers known edge cases and adversarial inputs.

Evidence to capture

- Unit, integration, and property-based test results.
- Reference outputs or manually verified cases.
- Failure-case reports.
- Determinism notes or reproducibility evidence.
- Coverage notes for critical branches and edge cases.

Acceptance criteria

- Correctness is demonstrated for representative and edge-case inputs.
- Any approximation is bounded and justified.
- Error handling is predictable and safe.
- Repeated runs produce acceptable, documented behavior.

Common mistakes

- Confusing code coverage with correctness.
- Using only ?happy path? tests.
- Failing to validate behavior on malformed or adversarial inputs.
- Accepting approximate outputs without documenting tolerances.

Owner and review cadence

- Owner: Test owner or validation engineer.
- Review cadence: Before pre-production sign-off and after any logic change.

Phase 5: Performance, scale, and resource validation

Purpose: Confirm the implementation meets operational performance goals under realistic load, not just in isolated tests.

Checklist items

- ? Validate runtime and memory behavior using realistic input sizes and growth patterns.

- ? Review whether asymptotic complexity aligns with production data volume and latency targets.
- ? Test for pathological inputs that trigger worst-case performance or excessive resource use.
- ? Confirm that timeouts, backpressure, or batching rules are defined where needed.
- ? Document CPU, memory, I/O, and network impacts if the algorithm participates in a larger pipeline.
- ? Assign responsibility for profiling, load testing, and interpreting performance regressions.
- ? Review whether performance remains acceptable when the algorithm is embedded in retry or failover logic.

Evidence to capture

- Benchmark results with input sizes and test conditions.
- Profiling output or resource measurements.
- Worst-case input examples.
- Latency and throughput observations.
- Capacity assumptions and limits.

Acceptance criteria

- Observed performance meets stated targets.
- Resource usage remains within approved limits.
- Worst-case behavior is known and acceptable.
- Operational controls exist for overload or timeout conditions.

Common mistakes

- Benchmarking only small or synthetic inputs.
- Ignoring memory growth while focusing only on runtime.
- Forgetting that retry logic can amplify load.
- Declaring success before testing pathological inputs.

Owner and review cadence

- Owner: Performance reviewer or systems engineer.
- Review cadence: After each meaningful optimization and before release.

Phase 6: Security, safety, and misuse resistance

Purpose: Ensure the algorithm cannot be abused, can fail safely, and does not introduce avoidable operational or security risk.

Checklist items

- ? Review whether user-controlled input can cause denial of service, resource exhaustion, or unsafe branching.
- ? Validate that sensitive data is not leaked through logs, exceptions, metrics, or traces.
- ? Confirm that authorization, authentication, or policy checks occur before algorithmic work that should not be exposed.
- ? Test that malformed, oversized, or adversarial inputs are rejected or bounded safely.
- ? Document any cryptographic, privacy, or compliance constraints that affect implementation choices.
- ? Assign a security reviewer for threat modeling or abuse-case review where risk is material.
- ? Review whether fallback modes preserve safety and do not silently degrade control boundaries.

Evidence to capture

- Threat or abuse-case notes.
- Sanitized log samples.
- Rejection behavior for invalid input.
- Security review comments.
- Safety constraints and approved fallbacks.

Acceptance criteria

- Known abuse paths are addressed or explicitly accepted.
- Sensitive data is not exposed in operational telemetry.
- Unsafe inputs are bounded or rejected.
- Fallback behavior is safe and documented.

Common mistakes

- Assuming algorithmic code is automatically secure because it is deterministic.
- Logging raw input that contains secrets or identifiers.
- Allowing unbounded recursion, loops, or allocations.
- Treating safety review as optional for ?internal? systems.

Owner and review cadence

- Owner: Security reviewer with implementation support from the engineering owner.
- Review cadence: Before any production exposure and after changes to input handling or telemetry.

Phase 7: Deployment readiness and operational handoff

Purpose: Make the implementation operable. Production readiness depends on monitoring, rollback, ownership, and documented response actions as much as code quality.

Checklist items

- ? Confirm that deployment is gated by passing tests, reviews, and required approvals.
- ? Validate that monitoring covers success rate, error rate, latency, resource usage, and abnormal output patterns.
- ? Document rollback, disablement, or fallback procedures and confirm they are executable.
- ? Assign an on-call or operational owner who knows the expected behavior and failure signals.
- ? Review whether config, feature flags, or runtime parameters are versioned and auditable.
- ? Test the release process in a staging or pre-production environment that resembles production.
- ? Document the support threshold for investigating regressions and the path for escalation.

Evidence to capture

- Release checklist or approval record.
- Monitoring dashboard definitions or metric list.
- Rollback procedure and rehearsal notes.
- Staging validation results.
- Operational runbook or handoff notes.

Acceptance criteria

- The algorithm can be observed and controlled after release.
- Rollback or disablement is practical and tested.
- Operational ownership is clear.
- The deployment path is repeatable.

Common mistakes

- Shipping without a rollback method.
- Monitoring only infrastructure while ignoring algorithm-specific correctness signals.
- Leaving runtime parameters undocumented.
- Assuming the implementer will remain available for operations.

Owner and review cadence

- Owner: Release manager or service owner.
- Review cadence: At every release, after rollback rehearsal, and whenever runtime configuration changes.

Readiness and maturity scoring

Use a simple score to decide whether the roadmap is ready to move forward or needs more work. Score each phase from 0 to 2:

- 0 = Not done or unverified
- 1 = Partially done, but evidence is incomplete
- 2 = Complete, reviewed, and evidenced

Maximum score: 14 points.

Interpretation

- 0?6 points: Not ready. Stop and complete the missing evidence.
- 7?10 points: Conditionally ready. Proceed only with explicit risk acceptance and a named owner for the gaps.
- 11?14 points: Ready for production approval, provided no critical risk item remains open.

Scoring evidence rules

- Score only what is evidenced, not what is believed to be true.
- Downgrade a phase if its acceptance criteria are not met.
- Treat unresolved security, correctness, or rollback gaps as blocking issues regardless of total score.

Follow-up actions when items fail

If any phase fails, use the failure type to choose the response. Keep the action narrow and evidence-driven.

- Problem fit failure: Revisit the problem statement, compare alternatives, or simplify the approach before coding further.
- Design failure: Update invariants, interfaces, or error-handling rules; then rerun design review.
- Correctness failure: Fix the logic, expand the test set, and require retesting with reference outputs.
- Performance failure: Profile the slow path, reduce complexity if possible, or add bounded execution controls.
- Security or safety failure: Block release until the input handling, telemetry, or fallback behavior is corrected and reviewed.
- Deployment failure: Rehearse rollback, patch monitoring gaps, and confirm that operational ownership is complete.

Final check

Before approving the algorithms implementation roadmap, confirm that the problem is well-defined, the implementation is testable, the outputs are correct, the resource profile is acceptable, and the release can be monitored and reversed. If any of those are still uncertain, the roadmap is not complete enough for production use.