TEMPLATE

Algorithms Production Readiness Checklist

A practical template for reviewing algorithms before production release. Verify problem fit, correctness, complexity, edge cases, safety, observability, and operational readiness.

Algorithms Checklist: Verify Correctness, Performance, and Safety Before Production

Resource type: Template Use case: Production review for algorithm implementations Applies to: Standalone algorithms, backend services, batch jobs, data pipelines, and embedded algorithmic components

---

How to use this template

Review the checklist in order, from problem definition through operational readiness. For each phase:

- Confirm the checks listed below.
- Capture the evidence requested.
- Compare the result against the acceptance criteria.
- Stop and fix any failed item before moving on.

Recommended review roles

- Algorithm owner: Responsible for algorithm design and problem fit
- Implementation owner: Responsible for code quality and validation

- Review/validation owner: Responsible for test evidence, performance checks, and release readiness

If this algorithm is part of a larger release, align this review with broader operational readiness practices.

Review summary

Field	Value
Algorithm or feature name	
System or service	
Owner	
Reviewer(s)	
Review date	
Release target	
Production environment	
Risk level	
Decision	? Pass ? Conditional pass ? Fail

Notes

- Problem scope:
- Input/output contract:
- Known constraints:
- Dependencies:

- Open risks:
- Required follow-ups:

Phase 1: Problem definition and scope

Goal: Confirm the algorithm solves the right problem, under the right constraints, with the right inputs and outputs.

Checklist

- [ ] Confirm the problem statement is specific, measurable, and bounded.
- [ ] Review the input domain, including data types, ranges, ordering, nullability, and maximum size.
- [ ] Validate the expected output format, determinism requirements, and tie-breaking rules.
- [ ] Document performance constraints such as latency, throughput, memory usage, and batch size.
- [ ] Assign ownership for algorithm logic, data assumptions, and runtime validation.
- [ ] Review whether the algorithm must be exact, approximate, stable, or idempotent.

Evidence to capture

- Written problem statement and acceptance criteria
- Input/output contract or interface specification
- Known constraints from product, data, or platform owners
- Estimate of workload volume and peak behavior

Acceptance criteria

- The problem statement is unambiguous and testable.
- All required input and output constraints are documented.
- The algorithm?s required behavior is clear under normal and abnormal inputs.
- The implementation scope is narrow enough to validate independently.

Common mistakes

- Treating a vague business request as a complete algorithm specification
- Failing to define how duplicates, missing values, or unordered inputs are handled
- Ignoring memory limits or expected peak input size

Review notes

- Findings:
- Required fixes:
- Approval status: ? Approved ? Needs changes

Phase 2: Algorithm choice and correctness model

Goal: Validate that the chosen approach is suitable and that its correctness can be reasoned about, not merely observed on a few examples.

Checklist

- [ ] Confirm the selected algorithm matches the problem class and constraints.
- [ ] Review whether the approach depends on sorted input, graph structure, recursion depth, or probabilistic assumptions.

- [ ] Validate that invariants are documented and preserved at each step.
- [ ] Document the termination condition and prove or explain why the algorithm cannot loop indefinitely.
- [ ] Review whether the algorithm produces the same result for the same input when determinism is required.
- [ ] Assign a reviewer to challenge the algorithm with counterexamples and boundary cases.

Evidence to capture

- Design notes, proof sketch, or correctness argument
- Invariants and termination conditions
- Counterexamples considered during review
- Dependency assumptions, including ordering and data structure requirements

Acceptance criteria

- The correctness argument is understandable and internally consistent.
- Required assumptions are explicit and verified elsewhere in the system.
- No hidden dependency undermines the algorithm?s correctness.
- Failure modes are identified where correctness cannot be guaranteed.

Common mistakes

- Choosing a familiar algorithm without checking whether its assumptions hold
- Relying on empirical success instead of a correctness argument
- Ignoring the difference between functional correctness and acceptable approximation

Review notes

- Findings:
- Required fixes:
- Approval status: ? Approved ? Needs changes

Phase 3: Complexity and resource analysis

Goal: Confirm that the algorithm is efficient enough for the expected workload and that resource use is bounded.

Checklist

- [ ] Review the time complexity for best, average, and worst cases.
- [ ] Validate the space complexity, including auxiliary structures, recursion stack, and buffering.
- [ ] Confirm that complexity claims match the actual implementation path, not just the intended design.
- [ ] Test whether pathological inputs trigger unacceptable behavior.
- [ ] Document any trade-offs between CPU, memory, and I/O.
- [ ] Assign a performance owner to verify the algorithm against production-like data.

Evidence to capture

- Complexity analysis with assumptions
- Memory profile or sizing estimate
- Performance test results on representative and worst-case inputs
- Notes on pathological or adversarial input patterns

Acceptance criteria

- Complexity is compatible with expected scale and peak load.
- Resource consumption stays within platform limits with margin.
- Worst-case behavior is known and acceptable.
- Any expensive operation is justified and bounded.

Common mistakes

- Assuming average-case complexity is sufficient without checking worst-case behavior
- Underestimating the cost of recursion, copying, or repeated scanning
- Ignoring hidden costs from serialization, allocation, or cache misses

Review notes

- Findings:
- Required fixes:
- Approval status: ? Approved ? Needs changes

Phase 4: Edge cases and failure handling

Goal: Verify that the algorithm behaves predictably when inputs are incomplete, malformed, empty, extreme, or adversarial.

Checklist

- [ ] Validate behavior for empty, single-item, duplicate, and maximum-size inputs.
- [ ] Review handling for null, missing, NaN, overflow, underflow, and out-of-range values where applicable.
- [ ] Test boundary transitions such as off-by-one limits and exact threshold values.
- [ ] Confirm the algorithm fails safely or returns a defined result for invalid input.
- [ ] Document fallback behavior, retries, or error propagation rules.
- [ ] Assign negative test cases that intentionally break assumptions.

Evidence to capture

- Edge-case test matrix
- Validation and sanitization rules
- Error handling examples or logs
- Overflow, underflow, and boundary test results

Acceptance criteria

- Every documented edge case has an expected result.
- Invalid input produces a safe and explicit failure mode.
- No edge case produces undefined behavior or silent corruption.
- Input validation occurs before dangerous operations whenever possible.

Common mistakes

- Testing only representative "happy path" data
- Allowing silent truncation, overflow, or implicit coercion
- Assuming upstream systems will always sanitize input correctly

Review notes

- Findings:
- Required fixes:
- Approval status: ? Approved ? Needs changes

Phase 5: Implementation quality and code review

Goal: Ensure the code matches the design, is readable enough to maintain, and does not introduce avoidable defects.

Checklist

- [ ] Confirm the implementation matches the documented algorithm step for step.
- [ ] Review naming, structure, and comments for maintainability.
- [ ] Validate that helper functions, recursion, and loops are bounded and purposeful.
- [ ] Test whether refactoring changed behavior or complexity.
- [ ] Document any language-specific pitfalls such as integer overflow, iterator invalidation, or reference aliasing.
- [ ] Assign a peer reviewer who did not author the original code.

Evidence to capture

- Code review record and approval status
- Static analysis output where available
- Comparison between design notes and implementation

- Notes on language-specific hazards

Acceptance criteria

- The code follows the reviewed design without unapproved deviations.
- Readability is sufficient for future maintenance and incident response.
- Static analysis, linting, or compile-time checks pass where applicable.
- Known language-specific pitfalls are mitigated or explicitly accepted.

Common mistakes

- Letting implementation drift from the approved design
- Using clever shortcuts that make future debugging harder
- Missing language-specific edge cases in numeric or memory handling

Review notes

- Findings:
- Required fixes:
- Approval status: ? Approved ? Needs changes

Phase 6: Testing and validation

Goal: Confirm the algorithm performs correctly under realistic conditions and with meaningful coverage.

Checklist

- [ ] Unit test the core logic and all decision branches.
- [ ] Include boundary, regression, and negative tests.
- [ ] Add property-based or randomized tests where suitable.
- [ ] Validate integration points and data transformations.
- [ ] Run tests with representative production-like data.
- [ ] Verify reproducibility when deterministic behavior is required.

Evidence to capture

- Test plan and test results
- Coverage summary for critical paths
- Regression cases and bug reproductions
- Determinism or repeatability evidence

Acceptance criteria

- Core behavior is covered by tests that fail when assumptions are broken.
- Edge cases and regression scenarios are explicitly validated.
- Integration tests confirm the algorithm works in its real execution context.
- Test outcomes support the documented correctness and performance claims.

Common mistakes

- Relying on one or two sample inputs
- Omitting regression cases for previously observed failures
- Treating coverage numbers as a substitute for meaningful validation

Review notes

- Findings:
- Required fixes:
- Approval status: ? Approved ? Needs changes

Phase 7: Security and abuse resistance

Goal: Make sure the algorithm does not create unnecessary security or reliability risk.

Checklist

- [ ] Validate all external inputs before use.
- [ ] Review for denial-of-service risks from large, malformed, or adversarial inputs.
- [ ] Confirm the algorithm does not leak sensitive data through logs, errors, or side channels.
- [ ] Review time, memory, and recursion risks that could be weaponized.
- [ ] Confirm any randomness is appropriate, controlled, and documented.
- [ ] Verify access control or authorization assumptions if the algorithm depends on protected data.

Evidence to capture

- Input validation and threat assumptions
- Abuse-case review or security notes
- Logging and error-handling review
- Security test results, if applicable

Acceptance criteria

- Inputs cannot trigger unsafe behavior without detection.
- Error paths do not expose sensitive information.
- Resource exhaustion risks are understood and mitigated.
- Security assumptions are explicit and reviewed.

Review notes

- Findings:
- Required fixes:
- Approval status: ? Approved ? Needs changes

Phase 8: Observability and operations

Goal: Ensure the algorithm can be monitored, supported, and debugged after release.

Checklist

- [ ] Define metrics for success, latency, failure rate, and resource usage.
- [ ] Add logs that are useful without being noisy or sensitive.
- [ ] Confirm alerts exist for abnormal error rates, performance regressions, and saturation.
- [ ] Define rollback, fallback, or feature-flag behavior if the algorithm misbehaves.
- [ ] Document runbooks or troubleshooting steps for operators.
- [ ] Verify that operational ownership is assigned after release.

Evidence to capture

- Metrics and alert definitions
- Logging and tracing plan
- Rollback or fallback procedure
- Runbook or support documentation

Acceptance criteria

- Operators can detect failure or degradation quickly.
- Logs and metrics support root-cause analysis.
- There is a clear rollback or mitigation path.
- Operational responsibilities are assigned and understood.

Review notes

- Findings:
- Required fixes:
- Approval status: ? Approved ? Needs changes

Final release decision

Go-live checklist

- [ ] All required phases are approved.
- [ ] Open risks are documented with owners and due dates.
- [ ] Required mitigations are complete or explicitly accepted.
- [ ] Monitoring, rollback, and support readiness are in place.

- [ ] The release decision is recorded.

Decision

Item	Status
Problem definition and scope	? Pass ? Fail
Algorithm choice and correctness model	? Pass ? Fail
Complexity and resource analysis	? Pass ? Fail
Edge cases and failure handling	? Pass ? Fail
Implementation quality and code review	? Pass ? Fail
Testing and validation	? Pass ? Fail
Security and abuse resistance	? Pass ? Fail
Observability and operations	? Pass ? Fail

Final decision: ? Approve for production ? Approve with conditions ? Do not approve

Conditions or follow-ups:

- - -

Sign-off

Role	Name	Date	Signature/approval
Algorithm owner			
Implementation owner			
Reviewer/approver			

Quick reference: minimum evidence set

Before production approval, collect at least:

- Problem statement and acceptance criteria
- Input/output contract
- Correctness argument or proof sketch
- Complexity analysis
- Edge-case test matrix
- Performance test results
- Security and abuse review notes
- Observability plan
- Final sign-off record

Optional appendices

A. Counterexample review prompts

Use these prompts to challenge the design:

- What happens if the input is empty, duplicated, unordered, or larger than expected?
- What assumption must be true for correctness?
- What breaks if the input arrives in a different order?
- What if the algorithm is called repeatedly at peak load?
- What if an attacker sends adversarial input?
- What is the worst-case runtime and memory usage?

B. Example risk ratings

Risk	Description
Low	Well-bounded algorithm with strong tests and low operational impact
Medium	Moderate complexity or dependency risk, mitigated by tests and monitoring
High	Sensitive correctness, performance, or safety concerns; requires close review

C. Release gating rule

A production release should not proceed if any of the following are true:

- The problem statement is unclear.
- Correctness depends on unverified assumptions.
- Worst-case resource use exceeds limits.
- Invalid input can cause undefined behavior.
- Required monitoring or rollback is missing.

Notes

This template is intentionally vendor-neutral and suitable for general engineering review. Adapt the evidence fields and approvals to your team?s release process, but keep the core checks intact so the review remains repeatable.