CHECKLIST

Smart contract audit-readiness checklist

30 checks before your smart contract audit: scope and docs, tests, Foundry fuzz and invariant tests, Slither triage, access control, integrations and deployment.

30 checks Tick each item once it is in place. The online version has the same explanations, with links.

Online version: sigmacode.io/en/insights/smart-contract-audit-readiness-checklist

A smart contract audit is expensive and strictly time-boxed. The auditors get a fixed number of days, and every hour they spend working out what the code is supposed to do, chasing a moving commit or writing the tests you did not write is an hour not spent on the logic that can actually lose funds. The findings report then fills up with missing NatSpec, floating pragmas and untested revert paths instead of the issues you are paying to find.

A well-prepared codebase gets a deeper audit for the same budget. This checklist collects what we put in place before handing code to an external auditor: 30 concrete checks across documentation, code hygiene, testing, access control, integrations, deployment and logistics. It does not replace the audit; it makes sure the audit time goes where it matters.

1. Scope and documentation

Auditors can only verify behaviour against an intent they understand, so the intent has to be written down.

- [ ] 01 **Freeze a commit hash.** Give the auditors one tagged commit and do not change the code under review during the engagement. Fixes go on a separate branch and are reviewed in the fix-review round.
- [ ] 02 **Publish a scope file list with nSLOC.** List every contract in scope with its path and normalised source lines of code, and state explicitly what is out of scope (libraries, mocks, scripts, previously audited files). Auditors quote and plan based on nSLOC, so an accurate count avoids surprises on both sides.
- [ ] 03 **Write an architecture overview.** One or two pages describing the contracts, how they call each other, which contracts hold funds and the main user flows. A simple diagram of calls and token flows saves the auditors hours of reverse engineering.
- [ ] 04 **Specify behaviour and invariants in prose.** Describe what each external function must do and the properties that must always hold, for example "the sum of user balances equals `totalAssets` " or "only the timelock can change fees". These statements become the basis for both the auditors' review and your own invariant tests.
- [ ] 05 **Document known issues and accepted risks.** List limitations you already know about and design decisions you accept, such as centralisation trade-offs or unsupported token types. This keeps them out of the findings report and shows the auditors where you have already thought things through.

2. Code hygiene and static analysis

Noise in the codebase costs audit time and produces low-value findings, so remove it before the auditors see it.

- [ ] 06 **Pin the compiler and reach zero warnings.** Use an exact `pragma solidity 0.8.x;` instead of a floating `^0.8.0`, and make the version, optimizer runs, `via_ir` and `evm_version` in `foundry.toml` match what you will deploy. Every compiler warning should be fixed or consciously explained.
- [ ] 07 **Pin and list dependencies.** Lock OpenZeppelin Contracts and every other library to an exact release, for example a tagged v5.x submodule or an exact version in `package.json`. List all dependencies in the audit documentation and flag any files you have copied or modified.
- [ ] 08 **Remove dead code, TODOs and debug output.** Delete unused functions, commented-out code, `console.log` imports and leftover test helpers from the scoped contracts. Open TODOs signal unfinished logic and will be reported as such.
- [ ] 09 **Complete NatSpec on external and public functions.** Every external and public function should have `@notice`, `@param`, `@return` and, where relevant, a note on access control and reverts. Use custom errors and emit events for every state change that matters off-chain.
- [ ] 10 **Run Slither and triage every finding.** Run `slither` on the frozen commit and classify each finding as fixed, false positive or accepted, with a one-line justification. A second tool such as Aderyn often catches different patterns, and sharing the triaged output tells auditors which automated findings are already handled.

3. Tests

Tests show the auditors how the code is meant to be used and let them write proofs of concept quickly.

- [ ] 11 **Cover every external function and publish the report.** Each external and public function needs at least one test of the happy path. Generate a line and branch coverage report with `forge coverage` and explain any uncovered branches instead of hiding them.
- [ ] 12 **Test revert paths and edge cases.** Assert that unauthorised calls, invalid parameters, zero amounts and boundary values revert with the expected custom error via `vm.expectRevert`. Untested revert paths are where validation and access-control bugs hide.
- [ ] 13 **Run fork tests against real integrations.** If the protocol talks to external contracts such as DEXes, lending markets or oracles, test against a mainnet or L2 fork with pinned block numbers. Mocks only prove that your code works with your assumptions about the other side.

4. Fuzz and invariant testing

Fuzzing finds the input combinations nobody thought of, and invariant tests check that the system stays consistent across sequences of calls.

- [ ] 14 **Fuzz every function that takes numeric input.** Write Foundry fuzz tests for amounts, timestamps, share calculations and fee math, and constrain inputs with `bound()` rather than excessive `vm.assume`. Rounding direction and overflow at extreme values are the typical targets.

- [ ] 15 **Write stateful invariant tests with handlers.** Use Foundry invariant testing with handler contracts that call the system in realistic sequences from several actors. Check solvency, conservation of value and access-control properties after every call sequence.
- [ ] 16 **Keep the written invariants and the test suite in sync.** Every invariant from your specification should map to an invariant test, and every invariant test should trace back to the specification. Run the suite with a meaningful number of runs and depth in CI, not only locally with defaults.

5. Access control and upgradeability

Privileged functions and upgrades can move or lock every asset in the system, so the trust model has to be explicit.

- [ ] 17 **List every privileged role and function.** Document each role, which functions it can call, who holds it and what the worst case is if its key is compromised. This is the trust assumptions section auditors will ask for first.
- [ ] 18 **Put admin power behind a multisig and a timelock.** Critical parameters and upgrades should be controlled by a multisig such as Safe, with a `TimelockController` or equivalent delay for changes that affect user funds. Document the signer threshold and the delay.
- [ ] 19 **Check storage layout for upgradeable contracts.** For proxies, use OpenZeppelin v5 upgradeable contracts with ERC-7201 namespaced storage and validate upgrades with the OpenZeppelin upgrades tooling. Include the storage layout diff between versions if an upgrade is in scope.
- [ ] 20 **Protect initializers and upgrade functions.** Call `_disableInitializers()` in the implementation constructor, use the `initializer` and `reinitializer` modifiers correctly and restrict `_authorizeUpgrade` for UUPS proxies (ERC-1822). An unprotected implementation contract is a finding auditors should never have to report.

6. External integrations

Every external call is an assumption about someone else's code, and each assumption should be written down and tested.

- [ ] 21 **Handle oracle staleness and failure.** Check `updatedAt` against the feed's heartbeat, reject zero or negative prices and, on L2s, check the sequencer uptime feed. Decide what the protocol does when the oracle is unavailable and document it.
- [ ] 22 **Account for token quirks.** State which token types are supported and handle or explicitly exclude fee-on-transfer, rebasing, non-18-decimal and non-standard ERC-20 tokens. Use `SafeERC20` for transfers and measure balance differences where the received amount matters.
- [ ] 23 **Map reentrancy surfaces.** List every external call and token transfer, follow checks-effects-interactions and use `ReentrancyGuard` (or `ReentrancyGuardTransient` with EIP-1153) where state is shared. Consider cross-function and read-only reentrancy, and callbacks from ERC-721, ERC-1155 and ERC-777 tokens.
- [ ] 24 **Consider MEV and front-running.** Add slippage limits and deadlines to swaps and deposits, protect ERC-4626 vaults against the first-depositor inflation attack and review any logic that depends on transaction ordering. Document which ordering risks you accept.

7. Deployment and operations

The audited code is only as safe as the way it is deployed and operated.

- [ ] 25 **Make deploy scripts reproducible and parameters documented.** Deploy with Foundry scripts from the frozen commit, keep every constructor and initializer parameter in version-controlled config and include the scripts in the audit scope or at least in the handoff. A correct contract deployed with wrong parameters is still broken.
- [ ] 26 **Rehearse on a testnet with verified sources.** Run the exact deployment script on a testnet and on a mainnet fork, verify the sources on the block explorer and check that roles ended up with the intended addresses. Give the auditors the testnet addresses so they can interact with a real deployment.
- [ ] 27 **Prepare pause, runbook and monitoring.** Decide who can pause what, write a short incident runbook and set up alerts for role changes, upgrades, large withdrawals and paused states. Auditors will review the emergency paths, so they need to exist before the audit.

8. Audit logistics

Good logistics keep the audit moving and make sure the fix round is actually used.

- [ ] 28 **Name a technical point of contact and a channel.** Assign one developer who knows the codebase and can answer questions within hours, and agree on a shared channel for the duration of the audit. Every unanswered question stalls the review.
- [ ] 29 **Reserve time for fixes and a fix-review round.** Plan developer capacity right after the report arrives and agree up front that the auditors review the fixes. Unreviewed fixes can introduce new bugs after the audit has signed off.
- [ ] 30 **Disclose previous audits and reviews.** Share earlier audit reports, their fix status and any internal reviews. Auditors can then focus on what changed and check that earlier findings did not regress.

How we can help

We offer a fixed-scope Audit-Readiness Sprint of one to two weeks: a threat model, a test-suite gap analysis, Foundry fuzz and invariant tests, Slither triage, a prioritised list of fixes and an audit handoff pack with scope, documentation and known issues. We are not an audit firm and the sprint does not replace an external audit; it prepares your code so that the auditors can spend their time on the logic. Details are on our pricing page and in our Blockchain & Web3 services.

If you would like a free estimate for your codebase, tell us about your project. We are happy to sign a mutual NDA before you share any code.