Code Review Report -- simple-calculator

Date: 2026-02-19 **Reviewers:** Sonnet (security) -- Haiku (performance) -- Sonnet (coverage) -- Opus (lead, consolidation)

Executive Summary

The simple-calculator project is a well-structured, beginner-friendly CLI application that meets its stated architectural contract. The codebase is free of injection vectors, hardcoded secrets, and third-party runtime dependencies, giving it a strong baseline security posture. However, the code assumes it will only ever be run interactively by a cooperative human user; this single assumption is the root cause of every medium-and-above finding across both the security and coverage reports. Performance is a non-issue -- the application is I/O-bound by design and all identified micro-inefficiencies are immeasurable or intentional. Test coverage provides a solid happy-path foundation but has critical gaps around error signals (`EOFError`, `KeyboardInterrupt`), pathological-but-valid numeric input (`nan`, `inf`), and test-harness fidelity (the `iter()`/`next()` pattern masking real EOF behavior). Addressing the findings below will harden the application without sacrificing its simplicity.

Verdict at a glance:

Dimension	Status
Security	Solid baseline; needs defensive input handling for non-interactive and edge-case scenarios
Performance	Clean -- no changes warranted
Test Coverage	Happy-path adequate; critical gaps in error-path and boundary-value testing

Cross-Cutting Themes

The following findings were independently raised by more than one specialist reviewer, confirming their significance.

Theme	Reports	Unified Description	Single Fix
`EOFError` unhandled	Security, Coverage	`input()` raises `EOFError` when stdin is closed or a pipe is exhausted. Neither `get_number()` (line 11) nor `main()` (line 31) catches it, producing a raw traceback that leaks file paths and Python internals. Coverage confirms zero test exercises this path.	Catch `EOFError` in both `get_number()` and `main()`, print a friendly message, and exit cleanly. Add corresponding tests using `side_effect=EOFError`.

Theme	Reports	Unified Description	Single Fix
`nan`/`inf` silently accepted	Security, Coverage	`float()` parses `"nan"`, `"inf"`, `"-inf"`, `"infinity"`, and case variants without raising `ValueError`. These values propagate into arithmetic producing nonsensical output (`nan + nan = nan`). No test covers this path.	After `float()` succeeds, add a `math.isfinite()` guard that rejects non-finite values with the existing "Invalid number" message. Add tests for `"nan"`, `"inf"`, `"infinity"`, and uppercase variants.
`StopIteration` masking in tests	Security, Coverage	All `iter()`/`next()`-based monkeypatches raise `StopIteration` (not `EOFError`) when the input sequence is exhausted. This masks infinite-loop bugs and produces misleading test failures rather than clear assertions.	Replace bare `next(inputs)` with a helper function that raises `AssertionError("Unexpected extra input() call")` on exhaustion. Apply to every `iter()`-based monkeypatch in the test file.
`KeyboardInterrupt` unhandled	Security, Coverage	`Ctrl+C` at any prompt produces a raw traceback. No test exercises this path.	Wrap the `main()` loop in `try/except KeyboardInterrupt` that prints `"Goodbye!"` and breaks. Add a test.

Findings

Security

All findings from the security review, merged and de-duplicated against coverage where overlap exists.

#	Sev	Location	Finding	Recommendation
S1	**MED**	`calculator.py:13`	`float()` silently accepts `"nan"`, `"inf"`, `"-inf"`, `"infinity"` and all case variants. These propagate into arithmetic producing nonsensical output.	Add `math.isfinite()` check after parsing; reject non-finite values with the "Invalid number" message.

#	Sev	Location	Finding	Recommendation
S2	**MED**	`calculator.py:9-15`	`get_number()` unbounded `while True` loop has no exit path when receiving only invalid data from non-interactive piped input -- denial-of-service condition.	Add a maximum retry counter (approximately 10 attempts) after which the function raises `RuntimeError` or returns `None`, handled gracefully in `main()`.
S3	**MED**	`calculator.py:9-15`	`EOFError` not caught in `get_number()`; stdin close or pipe exhaustion produces a raw traceback exposing file paths and Python version.	Catch `EOFError` inside the `while True` loop; raise `SystemExit` or print a user-friendly exit message.
S4	**MED**	`calculator.py:31`	`KeyboardInterrupt` (`Ctrl+C`) not caught in `main()`; raw traceback leaks deployment path and Python internals.	Wrap `main()` loop body in `try/except KeyboardInterrupt` that prints `"Goodbye!"` and exits cleanly.
S5	**MED**	`tests/test_calculator.py:59`	`iter()`-backed monkeypatch raises `StopIteration` (not `EOFError`) on exhaustion, masking infinite-loop bugs as misleading iterator errors.	Replace `next(inputs)` with a helper that raises `AssertionError` with a clear message when inputs are exhausted.
S6	**MED**	`tests/test_calculator.py:107-108`	Same `StopIteration` masking pattern in all `iter()`-based `main()` tests.	Apply the exhaustion-guard pattern to all `iter()`-based monkeypatches.
S7	LOW	`calculator.py:13`	`float("-0")` returns `-0.0`, which displays as `-0.0` in output -- confusing for a beginner calculator.	Normalize negative zero after parsing: if `value == 0` and `math.copysign(1, value) == -1`, set to `0.0`.

#	Sev	Location	Finding	Recommendation
S8	LOW	`calculator.py:13`	`1e309` silently evaluates to `float("inf")`, bypassing validation. Users can enter astronomically large values without warning.	Add `abs(value) > 1e15` upper-bound guard after the `math.isfinite()` check.
S9	LOW	`tests/test_calculator.py:44-47`	No test exercises `nan`/`inf`/`infinity` input paths through `get_number()`.	Add explicit test cases (covered in New Tests Checklist below).

Overall security posture: No injection vectors, no `eval`/`exec`, no hardcoded secrets, no third-party runtime dependencies. All findings stem from a single root cause: the code assumes interactive execution by a cooperative human user.

Performance

The performance review (Haiku) identified seven observations, all rated LOW severity with a unanimous verdict of "acceptable for scope -- no change needed." The application is I/O-bound by design: it blocks on `input()` waiting for the user, and all computation is trivial single-operation arithmetic. Specific items examined include multiple `print()` calls in `show_menu()`, f-string construction for result output, `.strip()` on every input attempt, menu re-rendering per loop iteration, and the one-frame overhead of the `add()`/`subtract()` wrapper functions (required by the architecture contract). None of these present measurable overhead, and no changes are warranted.

Test Coverage

All findings from the coverage review, merged and de-duplicated. Items that overlap with security findings reference the corresponding S-number. **HIGH** items are in bold.

#	Sev	Function	Gap	Suggested Test
C1	**HIGH**	`get_number()`	**`"nan"`/`"inf"`/`"-inf"`/`"infinity"` silently accepted -- no test. (See S1)**	**Test that `get_number()` rejects `"inf"`, `"nan"`, `"infinity"`, and uppercase variants, printing "Invalid number".**
C2	**HIGH**	`get_number()`	**`EOFError` from `input()` completely unhandled -- no test. (See S3)**	**Monkeypatch `input` with `side_effect=EOFError`; assert clean exit.**
C3	**HIGH**	`main()`	**`EOFError` at menu prompt unhandled -- no test. (See S3)**	**Monkeypatch `input` with `side_effect=EOFError`; assert clean exit message.**

#	Sev	Function	Gap	Suggested Test
C4	**HIGH**	`main()`	**Invalid number input during arithmetic not tested end-to-end.**	**Test `iter(["+","abc","3","4","q"])` shows both "Invalid number" and `"3.0 + 4.0 = 7.0"`.**
C5	MED	`add()`/`subtract()`	`float("inf")`, `float("-inf")`, `float("nan")` never passed directly to arithmetic functions.	Test `add(float("inf"), float("inf"))` and `add(float("nan"), 1)` to document IEEE 754 propagation behavior.
C6	MED	`get_number()`	Only one invalid input before valid is tested; multiple consecutive invalids not covered.	Test `iter(["abc","xyz","!!!","9"])` returns `9.0` with 3 error messages.
C7	MED	`get_number()`	`KeyboardInterrupt` not tested. (See S4)	Monkeypatch `input` with `side_effect=KeyboardInterrupt`; assert clean exit.
C8	MED	`main()`	`KeyboardInterrupt` at menu prompt not tested. (See S4)	Monkeypatch `input` with `side_effect=KeyboardInterrupt`; assert graceful handling.
C9	MED	`main()`	Multiple operations in one session not tested.	Test `iter(["+","1","2","-","10","3","q"])` yields both results.
C10	MED	`main()`	Multiple consecutive invalid menu options not tested.	Test `iter(["x","!","2","q"])` shows "Invalid option" twice.
C11	MED	`main()`	`StopIteration` masking pattern across all `iter()`-based tests. (See S5, S6)	Ensure all input sequences end with `"q"`; add defensive `"Goodbye"` assertion; use exhaustion-guard helper.
C12	LOW	`add()`	Zero operands (`0+0`, `n+0`, `0+n`) not tested.	Add `add(0,0)`, `add(5,0)`, `add(0,5)`.
C13	LOW	`subtract()`	Subtracting zero (`n-0`, `0-n`, `n-n==0`) not tested.	Add `subtract(5,0)`, `subtract(0,5)`, `subtract(7,7)`.
C14	LOW	`add()`	Negative zero (`-0.0`) not tested.	Test `add(-0.0, 0.0)` and `subtract(0.0, -0.0)`.
C15	LOW	`add()`	Mixed int/float operands not tested.	Test `add(2, 3.5) == 5.5`.

#	Sev	Function	Gap	Suggested Test
C16	LOW	`subtract()`	Very large floats (overflow to `inf`) not tested.	Test `add(sys.float_info.max, sys.float_info.max) == float("inf")`.
C17	LOW	`get_number()`	Whitespace-only input (`" "`) not tested.	Test `iter([" ", "4"])` returns `4.0` with error message.
C18	LOW	`get_number()`	Padded-valid input (`" 3.5 "`) not tested.	Test returns `3.5` without error.
C19	LOW	`show_menu()`	Only substring checks; exact output format not verified.	Assert full exact strings: `"--- Simple Calculator ---"`, `" + : Addition"`, separator line.
C20	LOW	`main()`	Welcome banner never asserted.	Add `"Welcome"` assertion in `test_main_quit`.

Prioritised Action Plan

Implementation changes first, then test additions, then final validation.

Implementation Changes

Step	File	Lines	Change	Resolves
1	`calculator.py`	1	Add `import math` at top of file.	Prerequisite for S1, S7.
2	`calculator.py`	9-15	In `get_number()`: after `float()` succeeds, add `if not math.isfinite(value): raise ValueError`. This rejects `nan`, `inf`, and variants using the existing "Invalid number" error path.	S1, C1.
3	`calculator.py`	9-15	In `get_number()`: catch `EOFError` alongside the `while True` loop. Print `"\nInput closed. Exiting."` and raise `SystemExit(0)`.	S3, C2.
4	`calculator.py`	9-15	In `get_number()`: add a retry counter (max ~10). After exhaustion, print an error and raise `SystemExit(1)`.	S2.
5	`calculator.py`	26-48	In `main()`: wrap the `while True` body in `try/except KeyboardInterrupt` that prints `"\nGoodbye!"` and breaks.	S4, C8.
6	`calculator.py`	26-48	In `main()`: catch `EOFError` on the menu `input()` call. Print `"\nInput closed. Exiting."` and break.	S3, C3.

Step	File	Lines	Change	Resolves
7 (optional)	`calculator.py`	13	After `math.isfinite()` check, normalize negative zero: `if value == 0.0: value = 0.0`.	S7, C14.
8 (optional)	`calculator.py`	13	Add magnitude guard: `if abs(value) > 1e15: raise ValueError`.	S8.

Test Improvements

Step	File	Change	Resolves
9	`tests/test_calculator.py`	Create a reusable `make_input_fn(sequence)` helper that raises `AssertionError("Unexpected extra input() call")` on exhaustion. Refactor all existing `iter()`/`next()` monkeypatches to use it.	S5, S6, C11.
10	`tests/test_calculator.py`	Add `get_number()` tests for `"nan"`, `"inf"`, `"infinity"`, `"-inf"`, `"Inf"` -- assert rejection with "Invalid number".	S1, S9, C1.
11	`tests/test_calculator.py`	Add `get_number()` test with `side_effect=EOFError` -- assert `SystemExit`.	S3, C2.
12	`tests/test_calculator.py`	Add `main()` test with `side_effect=EOFError` -- assert clean exit.	S3, C3.
13	`tests/test_calculator.py`	Add `main()` test with `side_effect=KeyboardInterrupt` -- assert `"Goodbye"`.	S4, C8.
14	`tests/test_calculator.py`	Add `main()` end-to-end test: `["+","abc","3","4","q"]` -- assert both "Invalid number" and result.	C4.
15	`tests/test_calculator.py`	Add `main()` multi-operation test: `["+","1","2","-","10","3","q"]` -- assert both results.	C9.
16	`tests/test_calculator.py`	Add `main()` multiple-invalid-option test: `["x","!","2","q"]` -- assert "Invalid option" appears twice.	C10.
17	`tests/test_calculator.py`	Add `get_number()` multiple-consecutive-invalid test: `["abc","xyz","!!!","9"]` -- assert 3 error messages.	C6.
18	`tests/test_calculator.py`	Add `get_number()` tests for whitespace-only input and padded-valid input.	C17, C18.

Step	File	Change	Resolves
19	`tests/test_calculator.py`	Add `add()`/`subtract()` boundary tests: zero operands, mixed int/float, negative zero, large floats, IEEE 754 specials.	C5, C12, C13, C14, C15, C16.
20	`tests/test_calculator.py`	Strengthen `show_menu()` assertions to check exact output lines. Add welcome-banner assertion to `test_main_quit`.	C19, C20.

Final Validation

Step	Command	Gate
21	`python -m black .`	Formatting clean
22	`python -m ruff check .`	Lint clean
23	`python -m pytest -q`	All tests pass
24	`python -m pytest --cov=. --cov-report=term-missing --cov-fail-under=85`	Coverage >= 85%
25	`python -m bandit -r . -q`	No high-severity issues
26	`python -m pip_audit`	No known vulnerabilities (or remediation documented)
27	`python -m detect_secrets scan --all-files`	No secrets detected

New Tests Checklist

`get_number()`

- □ Reject `"nan"` with "Invalid number" message
- □ Reject `"inf"` with "Invalid number" message
- □ Reject `"-inf"` with "Invalid number" message
- □ Reject `"infinity"` with "Invalid number" message
- □ Reject `"Inf"` (uppercase variant) with "Invalid number" message
- □ `EOFError` from `input()` exits cleanly (use `side_effect=EOFError`)
- □ `KeyboardInterrupt` from `input()` exits cleanly (use `side_effect=KeyboardInterrupt`)
- □ Multiple consecutive invalid inputs (`"abc"`, `"xyz"`, `"!!!"`, then `"9"`) returns `9.0` with 3 error messages
- □ Whitespace-only input (`"  "`) triggers "Invalid number", then accepts valid follow-up
- □ Padded-valid input (`"  3.5  "`) returns `3.5` without error

`show_menu()`

- □ Assert exact output lines: `"--- Simple Calculator ---"`, `" + : Addition"`, `" - : Subtraction"`, `" q : Quit"`, `"-------------------------"`

`main()`

- □ `EOFError` at menu prompt exits cleanly with friendly message
- □ `KeyboardInterrupt` at menu prompt prints `"Goodbye!"` and exits
- □ Multiple operations in one session: `["+","1","2","-","10","3","q"]` yields both results
- □ Multiple consecutive invalid menu options: `["x","!","2","q"]` shows "Invalid option" twice
- □ Invalid number during arithmetic end-to-end: `["+","abc","3","4","q"]` shows "Invalid number" and result
- □ Welcome banner asserted in quit test
- □ All `iter()`-based tests refactored to use exhaustion-guard helper

`add()` / `subtract()`

- □ `add(0, 0) == 0`
- □ `add(5, 0) == 5` and `add(0, 5) == 5`
- □ `subtract(5, 0) == 5`, `subtract(0, 5) == -5`, `subtract(7, 7) == 0`
- □ `add(-0.0, 0.0)` -- document or normalize result
- □ `subtract(0.0, -0.0)` -- document or normalize result
- □ `add(2, 3.5) == 5.5` (mixed int/float)
- □ `add(float("inf"), float("inf"))` -- document IEEE 754 behavior
- □ `add(float("nan"), 1)` -- document IEEE 754 behavior
- □ `add(sys.float_info.max, sys.float_info.max)` -- document overflow to `inf`

What Is Already Good

- **No injection surface.** No use of `eval()`, `exec()`, `os.system()`, or any dynamic code execution.
- **No hardcoded secrets.** The codebase is clean of credentials, tokens, and API keys.
- **Zero third-party runtime dependencies.** The only external packages are dev/test tooling, minimising supply-chain risk.
- **Architecture compliance.** The code faithfully follows the CLAUDE.md contract: five required functions, single-file layout, no global variables, no unnecessary abstractions.
- **Readable, beginner-friendly code.** Function names are self-documenting, control flow is linear, and there are no advanced Python idioms that would confuse a newcomer.
- **Solid happy-path test coverage.** All primary user flows (add, subtract, invalid option, quit) are exercised with clear, well-commented tests.
- **Input validation present.** `get_number()` correctly loops on `ValueError`, and `main()` handles invalid menu choices -- the core validation contract is in place.
- **Performance posture is appropriate.** The application is correctly I/O-bound with no unnecessary computation, data copying, or resource consumption.
- **Clean static analysis baseline.** Bandit, pip-audit, and detect-secrets all pass with no findings on the current codebase.
- **Pragma coverage exclusion** on `if __name__ == "__main__"` shows thoughtful test configuration.