DOWNLOADABLE RESOURCE

AI Self-Checking Code Prompt Pack

Prompts for code that validates inputs, tests behavior, detects incomplete work, and exposes failure.

Edition: July 2026 **Use:** Personal and business reference

The strongest form of AI self-checking is not asking the model to say the code looks correct. It is requiring executable tests, independent expected results, invariants, integrity checks, visible failure, and a record of what actually ran.

Define what correct means

1 Write the contract before the code

```
Before writing code, restate the inputs, outputs, invariants, error conditions,
performance limits, and side effects. Ask questions about any requirement that cannot
be tested objectively.
```

2 Tests before implementation

```
Write the test cases before the implementation. Include normal cases, boundary values,
empty input, malformed input, duplicate input, and failure cases. Then write the
smallest implementation that passes them.
```

3 Known-answer tests

```
Create at least five input cases with manually verifiable expected outputs. Explain how
each expected result was derived independently of the implementation.
```

4 Preconditions and postconditions

```
Add explicit precondition checks before the main operation and postcondition checks
afterward. Fail with a clear message rather than continuing with an invalid state.
```

5 Identify invariants

```
List the conditions that must remain true throughout execution. Add assertions or
validation code for those invariants at the points where they could be violated.
```

Make failure visible

6 No silent error handling

Do not use empty exception handlers or generic catch-all behavior that hides failure. Log the operation, the input context safe to record, the exception type, and the recovery decision.

7 Dry-run mode

Add a dry-run mode that reports intended file, database, or system changes without applying them. Ensure the dry-run uses the same selection logic as the real operation.

8 Transactional or rollback behavior

Where possible, stage changes and commit only after validation succeeds. Describe rollback behavior for partial failure and add a test that forces failure midway through the operation.

9 Integrity verification

For copy, export, backup, or migration code, verify item counts, byte counts where meaningful, and cryptographic hashes for files that must be identical. Report mismatches rather than silently retrying forever.

10 Idempotency test

Determine whether running the program twice should produce the same final state. Add a test for the second run and prevent duplicate records, repeated edits, or destructive overwrites.

Use independent test strategies

11 Differential testing

Create a simple reference implementation or use a trusted library to compute the same result for small inputs. Compare outputs across a generated test set and report every disagreement.

12 Property-based testing

Identify properties that should hold for many inputs, then generate randomized cases to test them. Include the random seed and preserve any failing example for reproduction.

13 Mutation challenge

```
List several realistic defects, such as off-by-one errors, reversed conditions, missing
rows, and wrong path selection. Explain which test would catch each defect. Add a test
where coverage is missing.
```

14 Static checks

```
Provide the appropriate formatter, linter, type checker, and security scanner commands
for this language and project. Distinguish warnings from failures and do not claim
runtime correctness from static checks alone.
```

15 Coverage gap review

```
After tests run, identify untested branches and error paths. Explain whether each gap
is acceptable, difficult to trigger, or important enough to require another test.
```

Require evidence from the run

16 Return test output

```
Run the tests when execution is available. Return the exact command used, test count,
pass and failure count, and relevant output. If execution is unavailable, state that
clearly and do not describe the code as tested.
```

17 Self-audit the implementation

```
Compare the completed code against the original contract line by line. Identify any
requirement handled only partially, any new dependency, and any assumption introduced
during implementation.
```

18 Adversarial input review

```
Try inputs intended to break the program: very large values, Unicode, duplicate names,
missing files, permission failures, network interruptions, and interrupted writes. Add
safe handling for relevant cases.
```

19 Observability review

```
Add concise logs or metrics that show how many items were received, processed, skipped,
failed, and produced. Reconcile the totals at the end of the run.
```

20

Final release gate

```
Before presenting the code as complete, produce a release gate with requirements met,
tests run, static checks run, known limitations, untested environments, rollback plan,
and the exact conditions that still require human review.
```

Generated tests can repeat the same misunderstanding as generated code. Keep at least some expected results independent, run the program in a controlled environment, and review destructive operations manually.