How to rewrite a legacy code

Context and Goals

Explain why you are rewriting, the scope, key risks, expected deliverables, and success criteria.

- Problem statement: <fill in>
- Target outcome: <fill in>
- Non-goals: <fill in>
- Success criteria: <fill in>

Test Strategy

Define what each test validates, how to run it locally and in CI, and the pass criteria.

Test types

- Golden test
    - Purpose: Guard against regressions by diffing current output against an approved canonical output.
    - Run: `make test_golden` or `pytest -m golden`
    - Pass criteria: No diffs or only approved diffs committed via an explicit approval step.
- Integration test
    - Purpose: Validate end-to-end behavior across boundaries and I/O.
    - Run: `make test_integration` or `pytest -m integration`
    - Pass criteria: All scenarios pass, with stable runtime and deterministic fixtures.
- Snapshot test

    - Purpose: Lock complex structured output or rendered text.
    - Run: `pytest -m snapshot`
    - Pass criteria: Snapshots match. Changes must be reviewed and re-snapshotted intentionally.
- Approval test
    - Purpose: Human review and approval of output changes that are semantically meaningful.
    - Run: `pytest -m approval && ./tools/approve.sh <artifact>`
    - Pass criteria: All changes explicitly approved.
- Unit test
    - Purpose: Verify small units with fast feedback. Currently emphasized for Python. For other languages, add equivalent unit tests.
    - Run: `pytest -q` or your language's unit runner.

Execution order (recommended)

1. Unit tests
2. Golden and Snapshot tests
3. Integration tests
4. Approval tests (for any changed artifacts)

Thresholds and gates

- Test pass rate: 100% required
- Coverage minimum: 80% lines overall, 70% per package or module
- Max test time: <5 min local, <10 min CI per job>
- Flakiness: Retries disabled by default. Any flaky test must be fixed or quarantined with a linked issue

---

OCaml → Python: test method (function by function)

Goal: port each OCaml function to Python while preserving observed behavior. Lock parity with tests before moving to the next function.

Steps per function

1. Map the OCaml function
    - Name, signature, input and output types, side effects, possible errors
    - Happy paths and edge cases identified in the code and existing OCaml tests
2. Extract or create reference test data
    - Reuse OCaml tests (expect or Alcotest) to generate canonical input–output pairs
    - Save pairs as JSON files under `tests/golden/<function>/cases.json`
3. Implement the Python equivalent
    - Preserve the same invariants and error behavior. Raise equivalent exceptions in Python
4. Write parity tests
    - Golden tests: compare Python output to canonical outputs produced by the OCaml executable
    - Deterministic unit tests for edge cases
    - Optional: property-based tests with Hypothesis to cover input ranges
5. Measure and validate
    - Coverage ≥ 80% overall and ≥ 70% per module
    - Test runtime short and stable
6. Lock it in
    - If parity holds, merge and move to the next function

Reference tooling

- Generate OCaml reference outputs (example):

```
# assumes a dune binary that evaluates a function on a stream of JSON inputs
jq -c '.[]' tests/golden/normalize/cases.json | \
  dune exec bin/ref_eval.exe -- --fn normalize --json > tests/golden/normalize/outputs.json
```

- Python harness to compare Python vs OCaml:

```
import json, subprocess
from mypkg import normalize

with open("tests/golden/normalize/cases.json") as f:
    cases = json.load(f)

ocaml_out = subprocess.check_output([
    "dune", "exec", "bin/ref_eval.exe", "--", "--fn", "normalize", "--json"
], input="\n".join(json.dumps(c) for c in cases).encode())
ref_outputs = [json.loads(l) for l in ocaml_out.decode().splitlines()]

for case, ref in zip(cases, ref_outputs):
    assert normalize(**case["input"]) == case.get("expected", ref)
```

Pytest skeleton per function

```
import json
import pytest
from mypkg import normalize

CASES = json.load(open("tests/golden/normalize/cases.json"))

@pytest.mark.golden
@pytest.mark.parametrize("inp, expected", [
    (c["input"], c["expected"]) for c in CASES
])
```

```
def test_normalize_golden(inp, expected):
    assert normalize(**inp) == expected

@pytest.mark.integration
def test_normalize_edge_cases():
    assert normalize(text="", mode="strict") == ""
    with pytest.raises(ValueError):
        normalize(text=None)
```

Pass criteria per function

- All golden tests pass
- Any intentional differences are documented and approved
- Known edge cases are covered by deterministic tests
- Local coverage ≥ 70% on the relevant module

CI integration (excerpt)

```
jobs:
  golden_ref:
    steps:
      - run: dune build
      - run: |
          jq -c '.[]' tests/golden/**/cases.json | \
          dune exec bin/ref_eval.exe -- --fn $FN --json > tests/golden/$FN/outpu
ts.json
  python_tests:
    steps:
      - run: pytest -m "golden or integration" --maxfail=1 -q
```

Branching and Naming Convention

Create feature branches from `dev` with consistent, discoverable names.

- Pattern examples

    - `<binary_name>_golden_master`
    - `<binary_name>_integration_test`
- Examples
    - `csv_importer_golden_master`
    - `csv_importer_integration_test`

Git flow

```
# from dev
git checkout dev
git pull

# create a focused branch
git checkout -b csv_importer_golden_master

# commit as you go
git add -A
git commit -m "csv_importer: add golden test harness"

# push and open a PR to dev
git push -u origin csv_importer_golden_master
```

CI/CD Required Before Merge

All checks must be green before merging back to `dev`.

- Required jobs
    - Lint and static checks
    - Unit tests
    - Golden, Snapshot, and Integration tests
    - Security scan (SAST)
- Required status

    - All jobs green
    - Minimum 2 code reviews approved
    - No TODO or FIXME introduced
- Artifacts and reports
    - Test report and coverage
    - Snapshot diffs as build artifacts
    - SBOM or dependency scan report

Example CI snippet (pseudocode)

```
jobs:
  test:
    steps:
      - run: pip install -r requirements.txt
      - run: pytest -m "not slow" --junitxml=report.xml --cov=.
  integration:
    steps:
      - run: pytest -m integration
  golden:
    steps:
      - run: pytest -m golden
  security:
    steps:
      - run: bandit -r src/ -f xml -o bandit.xml
  quality_gate:
    needs: [test, integration, golden, security]
    steps:
      - run: ./ci/check_coverage.sh --min 80
      - run: ./ci/check_reviews.sh --min 2
```

Documentation and Deliverables

Checklist to complete before merge.

- [ ] README updated with run instructions and flags
- [ ] Migration notes including data or interface changes
- [ ] Changelog entry with user-visible impact
- [ ] Diagrams updated if applicable
- [ ] Runbook for operations and on-call
- [ ] Linked tickets and tracking issues
- [ ] Technical owner listed

Quick Commands and References

Common commands to keep at hand.

```
# run tests
pytest -q
pytest -m golden
pytest -m integration

# update snapshots consciously
pytest -m snapshot --snapshot-update

# approve artifacts
./tools/approve.sh path/to/artifact
```