QuantoniumOS Developer Manual (Architecture & Operations)

As shipped in the latest push

This manual gives a complete, implementer-level view of QuantoniumOS as shipped in the latest push. It explains how the system is organized, how to build and run every major component, how to extend it safely, and how to validate results end-to-end.

0) Design Tenets

- **Determinism first:** math and engines must yield bit-for-bit reproducible round-trips where promised.
- **Isolation of concerns:** math (RFT), crypto, quantum simulation, and frontends are separable packages with thin adapters.
- **Evidence over assertion:** every claim must be paired with an executable test, a JSON result artifact, and a provenance note.
- **Pure Python path + accelerated path:** Python reference code exists for clarity; C++/pybind11 provides speed.
- **One-command repro:** scripts at repo root and `01_START_HERE/` / `07_TESTS_BENCHMARKS/` reproduce validation without editing source.

1) Repository Topography (what lives where)

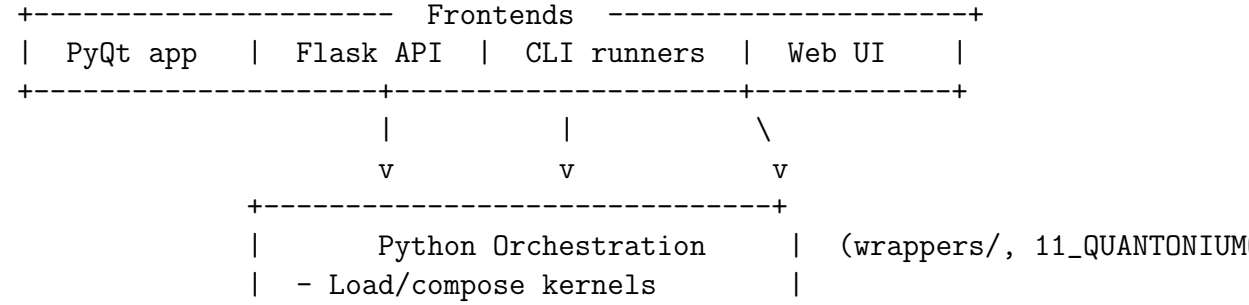
Top-level directories/files you'll touch most:

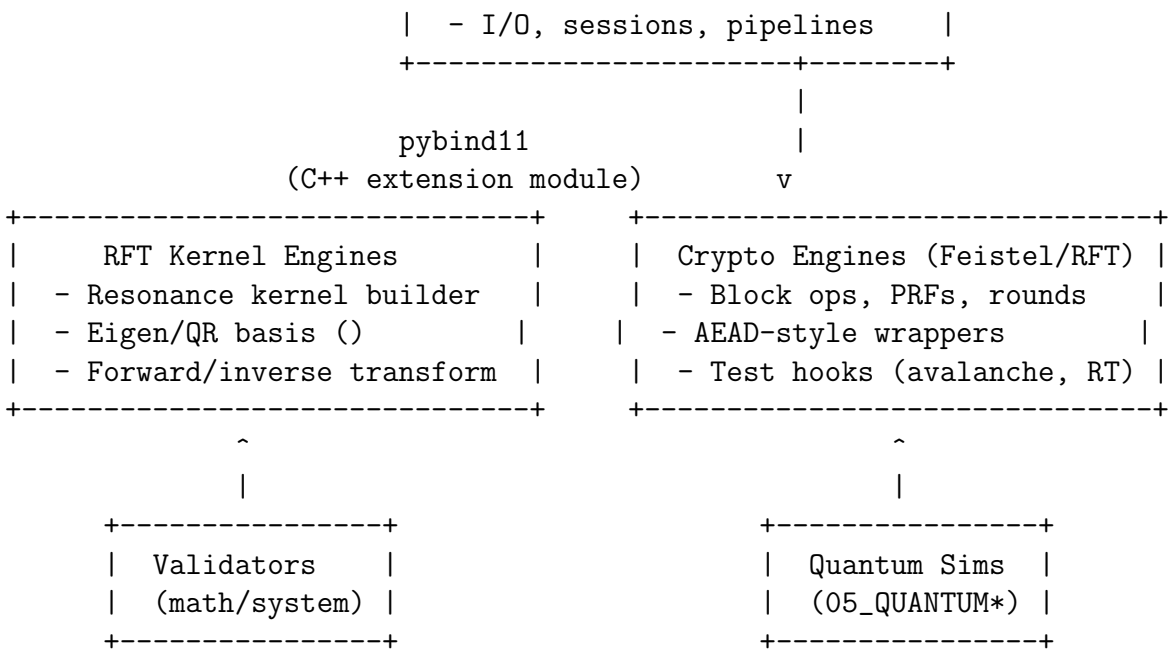
- `01_START_HERE/` — quick entries, minimal runners, and sanity smoke tests.
- `02_CORE_VALIDATORS/` — mathematical and systems validators (unitarity, reversibility, distinctness, property tests).
- `03_RUNNING_SYSTEMS/` — orchestration/launchers for UIs and services (CLI/Flask/PyQt flows).
- `04_RFT_ALGORITHMS/` — canonical RFT constructions, kernels, eigenbasis builders, transform drivers.
- `05_QUANTUM_ENGINES/` — qubit/vertex simulators and supporting numerics.

- 06_CRYPTOGRAPHY/ — RFT-driven crypto engines (C++ core + Python bindings) and wrappers.
- 07_TESTS_BENCHMARKS/ — property tests, avalanche/round-trip metrics, timing benches; writes JSON artifacts.
- 08_RESEARCH_ANALYSIS/ — analysis notebooks/scripts; meta-tests summarizing distinctness and corpus stats.
- 09_LEGACY_BACKUPS/ — archived prototypes/migrations retained for provenance only.
- 10_UTILITIES/ — maintenance scripts (auto-fixers, unicode/docstring repair, corpus hygiene).
- 11_QUANTONIUMOS/ — package entry points and integration glue shared by UIs.
- apps/ — runnable UI apps (Flask, PyQt, experimental)
- core/ — C++ kernels and support headers used by accelerated paths.
- frontend/, web/, wave_ui/ — web/UI assets.
- wrappers/ — Python bindings and adaptors around C++ engines.
- third_party/ — vendored deps (if any) or pin manifests.
- quantum_vault_data/ — persisted test outputs (JSON artifacts) and example corpora.
- Root scripts: canonical_true_rft*.py, build_true_rft_engine.py, build_engines.py, definitive_quantum_validation.py, full_patent_test*.py, final_paper_compliance_test.p benchmark_controller.py, platform launchers launch_*.ps1/.py/.bat.
- Build/packaging: CMakeLists.txt, MANIFEST.in, mypy.ini, quantoniumos.egg-info/.
- Docs: QUANTONIUM_DEVELOPER_GUIDE.md, QUICKSTART.md, SECURITY.md, CONTRIBUTING.md, CODE_OF_CONDUCT.md, research reports & verification summaries.

Pro tip: Keep new math under 04_RFT_ALGORITHMS/ and its tests under 07_TESTS_BENCHMARKS/ with matching names.

2) System Architecture at a Glance





Data contracts:

- Numeric arrays: `numpy.ndarray`, complex128 (state vectors, bases) and float64 (metrics). C++ side uses double/complex.
- Transform outputs: vectors or matrices with shape semantics documented per function.
- Results: JSON artifacts with versioned schema (*.json under root / `quantum_vault_data/`).

3) Build & Toolchain

3.1 Prereqs

- Python 3.10+ (recommend 3.11/3.12)
- Pip/venv and NumPy/SciPy stack
- A C++17 compiler (MSVC 2019+/Clang/GCC), CMake $\geq$ 3.16
- (Windows) PowerShell for `launch_*.ps1`; (Linux/macOS) Bash for `make_repro.sh`

3.2 Environment

```
# Clone
git clone https://github.com/mandcony/quantoniumos.git
cd quantoniumos

# Virtual env
python -m venv .venv && source .venv/bin/activate   # Windows: .venv\
   Scripts\activate
python -m pip install -U pip wheel setuptools

```

```
# (Optional) type checking
python -m pip install mypy
```

Listing 1: Setup Environment

3.3 Build the C++ engines (accelerated path)

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build --config Release -j
```

Listing 2: Generic CMake Build

This compiles the RFT/crypto engines described in `CMakeLists.txt`, producing a Python-loadable extension (platform-specific name) and/or binaries in `build/`. Helper scripts:

```
python build_engines.py            # one-shot dispatcher (if provided)
python build_true_rft_engine.py    # builds only the RFT core/bindings
```

Windows tip: If the module doesn't import, add `build/Release` (or `build/<cfg>`) to `PYTHONPATH` or copy the built `.pyd/.dll` next to the calling script.

3.4 Python-only reference path

Run the canonical Python transform without C++ acceleration:

```
python canonical_true_rft.py      # demo/validate transform
```

4) Running the system

4.1 Quick smoke

```
python 01_START_HERE/quick_sanity.py                # if present
python definitive_quantum_validation.py             # end-to-end math
    validation
python full_patent_test.py                          # full stack checks
python final_paper_compliance_test.py               # publication bundle
    check
```

Artifacts are written as JSON under the repo root or `quantum_vault_data/`.

4.2 Frontends

- **Flask API:** `python app.py` (or `launch_flask*.ps1`) to start an HTTP API exposing transform/crypto endpoints.
- **PyQt UI:** `python launch_pyqt5.py` (or `launch_pyqt5.ps1`) for a desktop controller/visualizer.
- **CLI:** invoke runners in `03_RUNNING_SYSTEMS/` or the root scripts for batch jobs.

4.3 Benchmarks & scaling

```
python benchmark_controller.py
python analyze_50_qubit_scaling.py
```

Use these to record timing/scale curves and store metrics.

5) Core Concepts & APIs

5.1 Resonance Fourier Transform (RFT)

- Kernel form: $R = \sum_i w_i D_{\phi i} C_{\sigma i} D_{\phi i}^\dagger$, with eigenvectors forming basis Ψ.
- Forward/Inverse: $X = \Psi^\dagger x$, $x = \Psi X$.
- Reference implementation: within `04_RFT_ALGORITHMS/` and `canonical_true_rft*.py`.
- Accelerated: C++ engine compiled via CMake and bound into Python (module in `wrappers/`), mirroring the reference API.

Typical usage pattern (Python):

```python
import numpy as np
from wrappers import rft  # or the built extension module name

x = np.random.randn(N).astype(np.complex128)
X = rft.forward(x)        # compute       x
x2 = rft.inverse(X)       # compute    X
assert np.allclose(x, x2, atol=1e-12)
```

Listing 3: RFT Python Usage

5.2 Crypto Engines (research)

- Exposed as a block cipher / AEAD-style wrapper driven by RFT components.
- C++ core provides the round function and permutation; Python wrappers add salt/KDF/HMAC options and test harness.
- Validation hooks: avalanche measurement, key sensitivity, round-trip correctness.

Example (Python):

```python
from wrappers import rft_crypto as rc
ct = rc.encrypt(key, nonce, plaintext, aad=b"")
pt = rc.decrypt(key, nonce, ct, aad=b"")
```

Listing 4: Crypto Engine Usage

5.3 Quantum Vertex/Simulator

- Vertex/qubit simulation utilities under `05_QUANTUM_ENGINES/` with validation scripts such as `enhanced_quantum_vertex_validation.py`.
- Focus: superposition/entanglement preservation across RFT-like ops; property checks live in `02_CORE_VALIDATORS/`.

6) Validation & Reproducibility

6.1 One-click validations

- `definitive_quantum_validation.py` — math invariants (unitarity, reconstruction, distinctness vs classical transforms).
- `full_patent_test*.py` — aggregates math, crypto, geometric, and simulation checks into a single verdict.
- `final_paper_compliance_test.py` — collects figures/tables/artifacts for publication bundles.

6.2 Artifact schema

Each run emits JSON with:

```
{
  "tool": "<script_name>",
  "version": "<semver/commit>",
  "datetime": "ISO-8601",
  "params": { ... },
  "metrics": { "reconstruction_error": 1.1e-15, ... },
  "verdict": "PASS|FAIL",
  "provenance": { "git_commit": "...", "platform": "..." }
}
```

Store under `quantum_vault_data/` or root `*_results.json`.

6.3 Adding a new validator

1. Create `02_CORE_VALIDATORS/validate_<claim>.py`.
2. Write a pure function returning a dict with metrics/verdict.
3. Add a CLI `if __name__ == "__main__":` block to run and dump JSON.
4. Register in `full_patent_test.py` (or its table) so it's included in the rollup.

7) Extensibility Patterns

7.1 Add a new RFT construction

- Place math in `04_RFT_ALGORITHMS/<name>/` or `04_RFT_ALGORITHMS/<name>.py`.
- Expose a stable interface: `build_kernel(params)` -¿ `ndarray`, `eigenbasis(kernel)` -¿ (`Psi`, `evals`), `forward(x)`, `inverse(X)`.
- Write `07_TESTS_BENCHMARKS/test_<name>_properties.py` validating invariants.

7.2 Add a crypto mode

- Implement a C++ round/permutation in `core/` and bind in `wrappers/`.
- Provide a Python façade with `encrypt/decrypt` and deterministic tests.
- Extend avalanche/round-trip tests and emit metrics JSON.

7.3 Add a quantum module

- Add operators/sim primitives in `05_QUANTUM_ENGINES/` with docstrings and small examples.
- Extend `enhanced_quantum_vertex_validation.py` to cover new ops.

8) Frontends & Services

8.1 Flask service

- `app.py` hosts endpoints for transform/crypto operations.
- Configure via env vars (`PORT`, `DEBUG`, optional `MODEL_PATH`).
- Run: `python app.py` (or `launch_flask*.ps1`).

8.2 PyQt desktop

- `launch_pyqt5.py` boots a controller to run transforms/benchmarks locally.
- Packager notes: freeze with pyinstaller once C++ extension is built.

8.3 Web assets

- `frontend/`, `web/`, `wave_ui/` contain JS/HTML/CSS assets if you're wiring a browser UI to the Flask backend.

9) Coding Standards & Tooling

- **Types:** opt into type hints; see `mypy.ini`.
- **Docs:** doctrings are required for public functions; run `fix_docstrings.py` for hygiene.
- **Formatting:** use black/ruff (pin versions in your environment); clang-format for C++.
- **Commits:** Conventional Commits style recommended (feat/fix/docs/test/chore).
- **Reviews:** attach JSON artifacts and commands to reproduce for any PR touching math/crypto.

10) Performance Guidance

- Prefer `complex128` end-to-end to avoid promotion churn.
- Vectorize: batch transforms over axis-0; avoid Python loops in hot paths.
- Enable compiler optimizations: `-O3` and hardware flags where appropriate.
- Measure, don't guess: use `benchmark_controller.py` + pinned seeds/data.

11) Security Posture (research only)

- Treat all crypto code as research grade until externally audited.
- Keep private keys and test data out of the repo.
- Use deterministic test vectors for reproducibility, not for real deployments.
- Follow the guidance in `SECURITY.md` and ensure side-channel considerations are documented in tests.

12) Troubleshooting

- **ImportError on C++ module:** ensure the built extension directory is on `PYTHONPATH`; check compiler/ABI.
- **CMake can't find Python:** pass `-DPython3_EXECUTABLE=`$(which python) to CMake. NumPy ABI
- **Windows build errors:** open a "x64 Native Tools Command Prompt" then re-run CMake.
- **Validation fails intermittently:** fix the RNG seeding; pin BLAS (MKL/OpenBLAS) to reduce nondeterminism.

13) Governance & CI

- Use GitHub Actions (workflows under `.github/`) to run validators on push.
- Block merges unless JSON artifacts are attached for math/crypto changes.
- Generate a signed release that includes the compiled extension and a frozen set of validation artifacts.

14) Roadmap Notes (Phase 3 → Phase 4)

- Phase 3 consolidates math proofs, bit-reversible demos, and CI reproducibility.
- Phase 4 productizes: persistent services, hardened API schemas, and vetted crypto profiles; document integration in `PHASE3_PHASE4_INTEGRATION_GUIDE.md`.

15) Glossary

- **RFT** — Resonance Fourier Transform, eigen-basis of a structured kernel.
- **Ψ (Psi)** — unitary eigenbasis used for forward/inverse transforms.
- **Round-trip** — encrypt→decrypt or forward→inverse returning original data within tolerance.
- **Avalanche** — bit-flip diffusion metric used in crypto evaluation.
- **Validator** — script producing a JSON verdict for a specific claim or invariant.

16) Change Safely Checklist (pre-PR)

1. Unit tests + property tests pass locally.
2. Benchmarks recorded against previous commit; no >5% regressions unless justified.
3. New/changed claims backed by validators and JSON artifacts.
4. Public APIs documented with examples.
5. Build the C++ extension and run both Python and accelerated paths.

17) Appendix — Useful Commands

```
# Build (generic)
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release && cmake --build build -
    j

# Python reference transform
python canonical_true_rft.py

# All-up validation
python definitive_quantum_validation.py
python full_patent_test.py
python final_paper_compliance_test.py


# Benchmarks
python benchmark_controller.py
python analyze_50_qubit_scaling.py

# Launch frontends
python app.py              # Flask
python launch_pyqt5.py     # PyQt
```

Listing 5: Useful Commands

18) Core RFT Equation

$$\mathrm{R} = \sum_i w_i D_{\phi i} C_{\sigma i} D_{\phi i}^{\dagger} \quad \text{(Resonance Kernel)}$$

$$\mathrm{X} = \Psi^{\dagger} x \quad \text{(Transform via eigenvectors)}$$

Mathematical Components:

- w_i = Golden ratio weights: ϕ^{-k} normalized
- $D_{\phi i}$ = Phase modulation matrices: $\exp(i\phi m)$
- $C_{\sigma i}$ = Gaussian correlation kernels with circular distance
- Ψ = Orthonormal basis from eigendecomposition of R