Below is a single copy-paste prompt you can drop into Claude Code to scaffold and build the entire **VectorVault** repo end-to-end.

Claude Prompt — “VectorVault Build Plan + Full Repo Scaffolding”

```
You are a senior C++ infra engineer + perf-obsessed reviewer.
Create a COMPLETE, production-quality repository called **vectorvault**:
a blazing-fast ANN vector search engine (HNSW + SIMD) with a tiny REST API,
benchmarks, tests, CI, and polish.

# 0) Project Goals (must-have)
- C++20 library implementing HNSW over float32 (L2 + cosine) with SIMD (AVX2)
kernels.
- REST microservice for add/query/save/load using cpp-httplib (or crow if
cleaner).
- Thread-safe queries with a lightweight thread pool.
- mmap-backed snapshot files; deterministic save/load.
- Benchmarks on synthetic datasets (100k x {384,768}) with recall@k vs brute
baseline,
  and latency percentiles (P50/P95) + QPS under different efSearch.
- Clean, modern CMake, strict warnings, sanitizers, clang-tidy, unit/integration
tests.
- Dockerfile + GitHub Actions (Ubuntu + Windows) building, testing, linting.
- README with quickstart, architecture diagram (ASCII), and charts (generated
from CSVs).
- MIT license, release-ready repo.

# 1) Stack / Tooling
- C++20, CMake (>=3.22)
- Deps via FetchContent:
  - eigen or xsimd (choose one; expose abstraction so kernels can be swapped)
  - cpp-httplib (HTTP), nlohmann_json
  - GoogleTest
  - spdlog
  - (stretch) pybind11 + scikit-build-core for Python bindings
- Build types: Debug (ASan/UBSan, -O0), Release (-O3 -march=native -DNDEBUG)
- Code style: clang-format file + .clang-tidy with modernize/readability/
performance checks
- Warnings: -Wall -Wextra -Wpedantic -Wconversion -Wshadow -Werror (toggle via
option)
- Static analysis target (clang-tidy), and CTest integration.
```

```
# 2) Repo Layout (create all files with minimal working content)
vectorvault/
├─ CMakeLists.txt
├─ cmake/
│  ├─ FetchContent.cmake
│  └─ warnings.cmake
├─ include/vectorvault/
│  ├─ hnsw.hpp
│  ├─ distance.hpp
│  ├─ thread_pool.hpp
│  ├─ index.hpp            # unified interface
│  ├─ mmap_io.hpp
│  └─ version.hpp
├─ src/
│  ├─ hnsw.cpp
│  ├─ distance_avx2.cpp    # with scalar fallback
│  ├─ distance_naive.cpp
│  ├─ thread_pool.cpp
│  ├─ mmap_io.cpp
│  └─ index_factory.cpp
├─ api/
│  ├─ main.cpp             # REST server
│  └─ server.hpp
├─ bench/
│  ├─ bench_main.cpp
│  ├─ gen_data.cpp         # generates .fvecs/.npy or raw .bin
│  └─ brute_force.cpp
├─ tests/
│  ├─ test_distance.cpp
│  ├─ test_hnsw_small.cpp
│  ├─ test_persistence.cpp
│  └─ test_api_integration.cpp
├─ scripts/
│  ├─ run_bench.sh
│  ├─ plot_bench.py        # matplotlib; read CSV and emit PNG
│  └─ dataset_specs.md
├─ docker/
│  └─ Dockerfile
├─ .github/workflows/
│  └─ ci.yml
├─ .clang-format
├─ .clang-tidy
├─ README.md
├─ LICENSE
└─ VERSION
```

```
# 3) CMake Requirements
- Targets:
  - vectorvault_lib (static)
  - vectorvault_api (REST binary)
  - vectorvault_bench (benchmark binary)
  - vectorvault_tests (gtest)
- Options:
  - VECTORVAULT_ENABLE_AVX2 (default ON if available)
  - VECTORVAULT_WERROR (default ON)
  - VECTORVAULT_BUILD_PYBIND (default OFF)
- Export compile_commands.json and install rules (headers + lib + api binary).
- Add `add_custom_target(format …)` and `tidy` targets.

# 4) Core APIs (C++)
## distance.hpp
Provide:
```

float l2_naive(const float *a, const float* b, int d) noexcept; float cosine_naive(const float *a, const float* b, int d) noexcept; float l2_simd(const float *a, const float* b, int d) noexcept; // AVX2 if available float cosine_simd(const float *a, const float* b, int d) noexcept;

```
- Choose simd vs naive with a function pointer resolved at init based on CPU
flags.
- Unit tests: random vectors d ∈ {16, 32, 64, 128, 384, 768, 1024}, assert |simd
- naive| < 1e-4.

## hnsw.hpp
Class `HNSWIndex`:
- ctor: `HNSWIndex(int dim, int M=16, int efConstruction=200, uint64_t seed=42)`
- `void add(int id, std::span<const float> vec)`
- `std::vector<std::pair<int,float>> search(std::span<const float> q, int k, int
efSearch=50) const`
- `void reserve(size_t n)`
- `bool save(const std::string& path) const`  // metadata + layers + neighbors +
vectors blob
- `bool load(const std::string& path)`
- Thread-safe `search` (const) using per-thread visited pools; `add` is single-
writer (guarded).
- Heuristic neighbor selection, multi-layer enterpoint, maxM per layer, level
generator.

## mmap_io.hpp
- Simple portable abstraction: memory-map a file for read/write snapshots.
- Ensure endian + version safety (use VERSION file + version.hpp constants).
- CRC32 or xxhash on vector payload blocks.

## thread_pool.hpp
```

```
- Fixed-size pool with work-stealing or simple queue; futures-based API.
- Used by REST for parallel queries and by brute-force baseline in bench.

## index.hpp / index_factory.cpp
- `std::unique_ptr<Index> make_hnsw(int dim, const HnswParams&);`
- Optional: allow future IVF/PQ implementations behind same interface.

# 5) REST API (cpp-httplib)
Start server on `:8080`. JSON via nlohmann_json.
Routes:
- `POST /add` body: `{ "id": int, "vec": [float...] }`  → 200/400
- `POST /query?k=10&ef=50` body: `{ "vec": [float...] }` → `{ "results":
[{"id":..., "dist":...}, ...], "lat_ms": ... }`
- `POST /save` body: `{ "path": "data/index.vv" }`
- `POST /load` body: `{ "path": "data/index.vv" }`
- `GET  /stats` → `{ "size": N, "dim": D, "levels": L, "params": {...} }`
Add integration test that boots server on ephemeral port, adds ~1k vectors,
queries, validates top-1 recall vs brute.

# 6) Benchmarks
Executables:
- `vectorvault_bench --mode=build --N=100000 --d=768 --M=16 --efC=200`
- `vectorvault_bench --mode=query --Q=1000 --k=10 --ef=10,50,100 --threads=8`
- `vectorvault_bench --mode=brute --baseline`  (exact search for recall
reference)
Artifacts:
- Write CSVs: columns `[mode, N, d, ef, k, threads, p50_ms, p95_ms, qps,
recall_at_k, build_s, index_mb]`
Scripts:
- `scripts/run_bench.sh` runs a matrix of configs, stores CSV to `bench/out/`.
- `scripts/plot_bench.py` reads CSV and emits PNGs:
  - efSearch vs Recall@10
  - efSearch vs QPS
  - Build time & Index size
Document machine specs printed by the bench binary: CPU model, cores, RAM,
compiler, flags.

# 7) Tests (GoogleTest)
- `test_distance.cpp`: simd ~ naive (tolerance 1e-4), randomized seeds.
- `test_hnsw_small.cpp`: small synthetic set (N=200, d=32), recall@5 ≥ 0.95 vs
brute.
- `test_persistence.cpp`: build index → save → load → identical search results
for fixed queries.
- `test_api_integration.cpp`: start server, add/query, validate JSON schema.
Add CTest with reasonable timeouts.

# 8) CI (GitHub Actions)
```

```
- Matrix: ubuntu-latest, windows-latest
- Steps: checkout → cmake configure+build (Debug+Release) → run unit tests → clang-tidy on src/include → cache ccache.
- Artifact: upload Release binaries (api + bench) per platform.

# 9) Docker
- Base: ubuntu:24.04
- Install deps + build Release
- Entrypoint serves API on 8080
- Example: `docker build -t vectorvault .` / `docker run -p 8080:8080 vectorvault`

# 10) README.md (generate content)
Include:
- One-paragraph pitch (“FAISS-adjacent HNSW in modern C++20”).
- Quickstart (3 commands).
- REST examples with curl:
  - `curl -X POST localhost:8080/add -d '{"id":1,"vec":[0.1, ...]}' -H 'Content-Type: application/json'`
  - `curl -X POST 'localhost:8080/query?k=5&ef=50' -d '{"vec":[...]}'`
- Architecture ASCII diagram (components: distance kernels, HNSW layers, mmap snapshot, REST).
- Bench results table + plots (insert PNGs produced by script).
- Config knobs (M, efConstruction, efSearch, thread count).
- Limitations & roadmap (PQ, IVF-Flat, PQ compression, Python bindings).

# 11) Python bindings (stretch; behind option)
- If VECTORVAULT_BUILD_PYBIND=ON:
  - Create `python/pyvectorvault/` with pybind11 module exposing add/search/save/load.
  - scikit-build-core pyproject.toml
  - Example notebook code snippet calling into the module.

# 12) Acceptance criteria (block PR until all pass)
- `cmake -S . -B build -DCMAKE_BUILD_TYPE=Release && cmake --build build -j` succeeds on Ubuntu.
- `ctest --output-on-failure` passes.
- `vectorvault_api` runs and returns valid JSON for /stats and /query.
- `scripts/run_bench.sh` produces CSVs; `scripts/plot_bench.py` outputs PNGs.
- README contains GIF or asciinema of querying + plots.
- CI green on Ubuntu + Windows.
- `docker run -p 8080:8080 vectorvault` serves.

# 13) Nice-to-haves (if time)
- spdlog logger w/ env var log level.
- Config file (TOML) for API server (thread count, ef defaults, snapshot path).
- Graceful shutdown saving latest index.
```

```
# 14) Deliverables
Return the FULL repo as code blocks, file by file, ready to copy,
with minimal but compiling implementations for every listed file,
and TODO markers where deeper logic continues (but HNSW + distance
SIMD kernels must be real, correct, and benchmarkable).
Also include a short section instructing how to run unit tests,
benchmarks, and the API.

Begin now.
```

Usage tip: paste the whole block into Claude Code, then immediately ask it to “run tests, build Release, and print exact commands for Ubuntu + Windows.”