Definitive Implementation Guide: A High-Performance KV Store

This document provides a complete roadmap for developing the high-performance transactional key-value store. It includes a full macOS development environment setup, a detailed project and directory structure, and a granular, phased implementation plan with specific tasks, deliverables, and verification steps.

1. Development Environment Setup (macOS)

This section details the tools and steps required to configure a C++ development environment on macOS.

1.1. Core Tools Installation

1. **Xcode Command Line Tools:** This package provides the Clang C++ compiler, Git, and other essential command-line utilities.
    - **Action:** Open the Terminal and run:
      Bash
      `xcode-select --install`

    - **Verification:** After installation, verify that the Clang compiler is available:
      Bash
      `clang --version`

2. **Homebrew:** A package manager for macOS used to install software not provided by Apple.
    - **Action:** Install Homebrew by running the command from its official website:
      Bash
      `/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"`

3. **CMake:** The build system generator for managing the project's compilation process.
    - **Action:** Use Homebrew to install CMake [1]:
      Bash
      brew install cmake

    - **Verification:** Check the installed version:
      Bash
      cmake --version

1.2. Editor and Extensions

- **Visual Studio Code:** A lightweight but powerful code editor.
    - **Action:** Download and install from the official website.
    - **C/C++ Extension Pack:** For debugging, code completion, and syntax highlighting.
    - **Action:** Inside VS Code, open the Extensions view (Cmd+Shift+X), search for C/C++ Extension Pack, and install it.[1]

2. Project Directory and File Structure

A well-organized directory structure is crucial for maintainability and scalability. The following structure separates concerns such as source code, public headers, tests, and dependencies.[2]

```
VrootKV/
├──.gitignore
├── CMakeLists.txt          # Root CMake file: project definition, enables testing, adds subdirectories
├── build/                  # Build artifacts (ignored by Git)
├── docs/                   # Project documentation (design docs, etc.)
├── include/
│   └── VrootKV/            # Public headers, namespaced by project name
```

```
│       ├── common/
│       │   ├── bloom_filter.h
│       │   └── status.h
│       ├── io/
│       │   ├── file_manager.h
│       │   └── sstable.h
│       ├── memtable/
│       │   └── memtable.h
│       ├── transaction/
│       │   └── transaction.h
│       └── wal/
│           └── wal.h
├── src/
│   ├── CMakeLists.txt        # Builds the main static library `libVrootKV.a`
│   ├── common/
│   │   ├── bloom_filter.cpp
│   │   └── status.cpp
│   ├── io/
│   │   ├── file_manager.cpp
│   │   ├── sstable_builder.cpp
│   │   ├── sstable_reader.cpp
│   │   └── sstable_format.h  # Private header for SSTable block structures
│   ├── index/
│   │   ├── art.cpp
│   │   └── art.h             # Private header for ART implementation
│   ├── memtable/
│   │   ├── memtable.cpp
│   │   └── skip_list.h       # Private header for lock-free Skip List
│   ├── transaction/
│   │   ├── transaction.cpp
│   │   └── mvcc.cpp
│   └── wal/
│       ├── wal_reader.cpp
│       ├── wal_writer.cpp
│       └── wal_format.h      # Private header for WAL record structure
└── tests/
    ├── CMakeLists.txt        # Defines test executables, links against GTest and our library
    ├── common/
    │   ├── test_bloom_filter.cpp
    │   └── test_status.cpp
    ├── io/
    │   └── test_sstable.cpp
    ├── memtable/
```

```
│   └── test_memtable.cpp
├── transaction/
│   └── test_mvcc.cpp
└── wal/
    └── test_wal.cpp
```

3. Granular Implementation Plan

This plan breaks down the development process into four distinct phases. Each task is designed to be a small, verifiable step.

Phase 1: Core Data Structures & I/O Primitives

Goal: Build the fundamental, self-contained building blocks. All components in this phase can be developed and unit-tested in isolation.

Module 1.1: Low-Level I/O Abstraction

- **Task 1.1.1: Define File System Interfaces**
  - **Description:** Create abstract base classes IWritableFile, IReadableFile, and IFileManager to define a clean interface for file operations.
  - **Deliverable:** include/VrootKV/io/file_manager.h.
- **Task 1.1.2: Implement POSIX File System Wrapper**
  - **Description:** Implement concrete classes that use standard C++ file streams or POSIX system calls (open, read, write, fsync).[5]
  - **Deliverable:** src/io/file_manager.cpp.
- **Task 1.1.3: Create Unit Tests with Mocking**
  - **Description:** Write unit tests for the I/O layer. Use a mocking framework (like Google Mock, part of Google Test) to simulate disk behavior.
  - **Deliverable:** tests/io/test_file_manager.cpp.
  - **Verification:** Confirm all interface methods behave as expected under normal and error conditions.

Module 1.2: On-Disk Data Formats (Serialization)

- **Task 1.2.1: Implement WAL Record Serialization**
  - **Description:** Define the WALRecord struct in a private header. Implement serialize and deserialize methods. The format must be ``.[5]
  - **Deliverable:** src/wal/wal_format.h.
  - **Verification:** Unit test in tests/wal/test_wal.cpp that a serialized record can be deserialized perfectly and that corruption is detected.
- **Task 1.2.2: Implement SSTable Block Formats**
  - **Description:** In a private header, define structs for BlockHandle (offset, size), and SSTableFooter (magic_number, index_block_handle, filter_block_handle).
  - **Deliverable:** src/io/sstable_format.h.
- **Task 1.2.3: Implement Data Block Builder & Reader**
  - **Description:** Create a DataBlockBuilder to add key-value pairs and generate a serialized data block with prefix compression. Create a DataBlockReader to parse it.
  - **Deliverable:** Add classes to src/io/sstable_builder.cpp and src/io/sstable_reader.cpp.
  - **Verification:** Unit test in tests/io/test_sstable.cpp by building a block and verifying the reader can retrieve all original key-value pairs.
- **Task 1.2.4: Implement Index Block Builder & Reader**
  - **Description:** Create an IndexBlockBuilder that takes divider keys and BlockHandles, and an IndexBlockReader that can perform a binary search.
  - **Deliverable:** Add classes to src/io/sstable_builder.cpp and src/io/sstable_reader.cpp.
  - **Verification:** Unit test in tests/io/test_sstable.cpp by building an index block and verifying lookups.

Module 1.3: Standalone Data Structures (Single-Threaded)

- **Task 1.3.1: Implement Bloom Filter**
  - **Description:** Create a BloomFilter class with add(key) and might_contain(key) methods. It must be serializable to a byte buffer.
  - **Deliverable:** include/VrootKV/common/bloom_filter.h and src/common/bloom_filter.cpp.
  - **Verification:** Unit test in tests/common/test_bloom_filter.cpp. Verify no false negatives and that the false positive rate is within configured bounds.
- **Task 1.3.2: Implement Skip List**
  - **Description:** Implement a standard, single-threaded Skip List for storing sorted key-value pairs.

    - **Deliverable:** A SkipList class in the private header src/memtable/skip_list.h.
    - **Verification:** Unit test in tests/memtable/test_memtable.cpp. Verify all operations and ordered iteration.
- **Task 1.3.3: Implement Adaptive Radix Tree (ART)**
    - **Description:** Implement a single-threaded ART with adaptive node types (Node4, Node16, Node48, Node256) and path compression.
    - **Deliverable:** An ART class in the private header src/index/art.h and implementation in src/index/art.cpp.
    - **Verification:** Write extensive unit tests in tests/index/test_art.cpp covering all operations and node type transitions.

Phase 2: Building the Write & Read Path

Goal: Assemble the Phase 1 components into functional, single-threaded data paths.

Module 2.1: WAL Manager

- **Task 2.1.1: Implement WALWriter**
    - **Dependencies:** I/O Abstraction (1.1), WAL Record Format (1.2.1).
    - **Description:** Create a class to manage writing to a log file. Implement group commit logic to buffer records and sync them in a single batch.[7]
    - **Deliverable:** WALWriter class in src/wal/wal_writer.cpp.
- **Task 2.1.2: Implement WALReader and Recovery**
    - **Dependencies:** I/O Abstraction (1.1), WAL Record Format (1.2.1).
    - **Description:** Create a WALReader to read records sequentially, validating checksums. Implement a recover() function that replays log entries.
    - **Deliverable:** WALReader class in src/wal/wal_reader.cpp.
- **Task 2.1.3: Test WAL Durability**
    - **Dependencies:** WALWriter, WALReader.
    - **Description:** In tests/wal/test_wal.cpp, write a test that writes records, simulates a crash (by not closing the file), and verifies recover() reconstructs the state.

Module 2.2: SSTable Manager

- **Task 2.2.1: Implement SSTableBuilder**
  - **Dependencies:** I/O Abstraction (1.1), All SSTable Formats (1.2), Bloom Filter (1.3.1).
  - **Description:** Create a class that orchestrates writing a complete SSTable file.
  - **Deliverable:** SSTableBuilder class in src/io/sstable_builder.cpp.
- **Task 2.2.2: Implement SSTableReader**
  - **Dependencies:** I/O Abstraction (1.1), All SSTable Formats (1.2), Bloom Filter (1.3.1).
  - **Description:** Create a class to open an SSTable, load its index and filter, and perform key lookups.
  - **Deliverable:** SSTableReader class in src/io/sstable_reader.cpp.
- **Task 2.2.3: Test SSTable Read/Write Path**
  - **Dependencies:** SSTableBuilder, SSTableReader.
  - **Description:** In tests/io/test_sstable.cpp, write an integration test that builds an SSTable, then uses the reader to perform lookups.
  - **Verification:** Confirm correct values are returned and the Bloom filter prevents disk reads for non-existent keys.

Module 2.3: Memtable and Flush Logic

- **Task 2.3.1: Create Memtable Class**
  - **Dependencies:** Skip List (1.3.2).
  - **Description:** Create a Memtable class that wraps the SkipList and tracks its memory usage.
  - **Deliverable:** include/VrootKV/memtable/memtable.h and src/memtable/memtable.cpp.
- **Task 2.3.2: Implement Flush Process**
  - **Dependencies:** Memtable (2.3.1), SSTableBuilder (2.2.1).
  - **Description:** Implement a method in the Memtable class that takes its sorted iterator and feeds the key-value pairs into an SSTableBuilder.
  - **Deliverable:** flush_to_sstable method in src/memtable/memtable.cpp.
- **Task 2.3.3: Test Memtable Flush**
  - **Dependencies:** Memtable (2.3.1), SSTableReader (2.2.2).
  - **Description:** In tests/memtable/test_memtable.cpp, populate a Memtable, flush it, and use an SSTableReader to verify the contents.

Phase 3: Concurrency and Transactions

Goal: Introduce the MVCC framework and make the data paths thread-safe and transactional.

Module 3.1: Lock-Free Data Structures

- **Task 3.1.1: Implement ConcurrentSkipList**
  - **Dependencies:** Single-threaded Skip List (1.3.2).
  - **Description:** Re-implement the SkipList in src/memtable/skip_list.h using std::atomic pointers and compare-and-swap (CAS) loops for thread-safe operations.[8]
  - **Deliverable:** A thread-safe ConcurrentSkipList class.
  - **Verification:** Write multi-threaded stress tests in tests/memtable/test_memtable.cpp. Use tools like ThreadSanitizer to detect data races.
- **Task 3.1.2: Implement Persistent ART with Path Copying**
  - **Dependencies:** Single-threaded ART (1.3.3).
  - **Description:** Modify the ART's update methods to be non-destructive, using path copying to create a new tree version and return a new root pointer.
  - **Deliverable:** A persistent (functional) ART class in src/index/art.h and src/index/art.cpp.
  - **Verification:** Unit test that an update creates a new root, and that traversals from old and new roots see the correct data.

Module 3.2: MVCC Core

- **Task 3.2.1: Implement TransactionManager**
  - **Description:** Create a manager that issues unique TxnIDs using std::atomic<uint64_t> and tracks active transactions thread-safely.
  - **Deliverable:** TransactionManager class in src/transaction/mvcc.cpp.
- **Task 3.2.2: Implement Versioned Data Records**
  - **Description:** Modify all data-holding structures to store versioned records: (key, creation_TxnID, deletion_TxnID, value). A delete operation inserts a record with a "tombstone" marker.
  - **Deliverable:** Updated data record definitions in private headers.
- **Task 3.2.3: Implement Transaction Snapshots and Visibility**
  - **Description:** Create a Transaction class that gets a snapshot from the TransactionManager. Implement the core isVisible(record, snapshot) function according to the rules of Snapshot Isolation.
  - **Deliverable:** include/VrootKV/transaction/transaction.h, src/transaction/transaction.cpp, and the isVisible function in

src/transaction/mvcc.cpp.
    - **Verification:** Write extensive unit tests for the isVisible function in tests/transaction/test_mvcc.cpp.

Module 3.3: Persistent & Versioned ART Integration

- **Task 3.3.1: Implement ART Persistence and Lazy Loading**
    - **Dependencies:** Persistent ART (3.1.2).
    - **Description:** Implement the on-disk format using a post-order traversal and the pointer swizzling mechanism using the most significant bit of a 64-bit pointer as a flag.
    - **Deliverable:** Serialization and lazy-loading logic for the ART in src/index/art.cpp.
    - **Verification:** Create a large ART, serialize it, restart, and perform lookups. Verify that only necessary nodes are read from disk.

Phase 4: Background Processes & Final Integration

Goal: Implement maintenance tasks and assemble the final, user-facing database engine.

Module 4.1: Compaction Engine

- **Task 4.1.1: Implement MANIFEST Logic**
    - **Description:** Implement classes to represent the LSM-tree state (Version, VersionEdit). Implement logic to write a new MANIFEST file and atomically update the database state using a rename on a CURRENT file.
    - **Deliverable:** Version management and MANIFEST I/O classes in src/io/version.cpp (and private header).
- **Task 4.1.2: Implement K-Way Merge Iterator**
    - **Description:** Create a merging iterator that takes a list of SSTable iterators and yields key-value pairs in sorted order, correctly handling multiple versions of the same key.
    - **Deliverable:** A generic merging iterator in src/io/merge_iterator.h.
- **Task 4.1.3: Implement Leveled Compaction Strategy**
    - **Dependencies:** MANIFEST (4.1.1), Merge Iterator (4.1.2), SSTable Manager (2.2).
    - **Description:** Implement the logic for Leveled Compaction: decide when and what to compact.

    - **Deliverable:** Compaction logic in src/compaction/compactor.cpp.
- **Task 4.1.4: Implement Background Thread Pool**
    - **Description:** Create a dedicated thread pool for running flush and compaction jobs.
    - **Deliverable:** A configurable background thread pool in src/common/thread_pool.cpp.

Module 4.2: Garbage Collection

- **Task 4.2.1: Integrate Data GC with Compaction**
    - **Dependencies:** Compaction Engine (4.1), TransactionManager (3.2.1).
    - **Description:** The TransactionManager must expose a getLowWatermark() method. The compactor must use this to discard any version whose deletion_TxnID is committed and older than the low-water mark.
    - **Deliverable:** GC logic integrated into src/compaction/compactor.cpp.
- **Task 4.2.2: Implement ART Node GC**
    - **Dependencies:** Persistent ART (3.1.2).
    - **Description:** Implement a mark-and-sweep garbage collector for ART nodes. Roots are the latest committed ART root and all roots held by active transaction snapshots.
    - **Deliverable:** A garbage collector for persistent ART nodes in src/index/art_gc.cpp.
- **Task 4.2.3: Test Garbage Collection**
    - **Description:** Write a test that runs a workload with updates/deletes, holds a long-running transaction to pin the low-water mark, runs compaction, and verifies old versions are kept. Then, commit the transaction, advance the mark, run compaction again, and verify the old versions are now purged.

Module 4.3: Top-Level API Integration & Build System

- **Task 4.3.1: Implement Database Class**
    - **Dependencies:** All other components.
    - **Description:** Create the final top-level class that encapsulates all components and exposes the public API: Open, Close, BeginTransaction.
    - **Deliverable:** The main Database class in src/db.cpp.
- **Task 4.3.2: Configure Root CMakeLists.txt**
    - **Description:** Set up the main project file.
    - **Deliverable:** CMakeLists.txt with:
      ```CMake
      cmake_minimum_required(VERSION 3.14)
      project(VrootKV CXX)
      ```

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Enable testing
enable_testing()

# Add subdirectories
add_subdirectory(src)
add_subdirectory(tests)
```

- **Task 4.3.3: Configure src/CMakeLists.txt**
  - **Description:** Define the main library target.
  - **Deliverable:** src/CMakeLists.txt with:

```
CMake
# Gather all source files
file(GLOB_RECURSE LIB_SOURCES "*.cpp")

# Create the static library
add_library(VrootKV STATIC ${LIB_SOURCES})

# Expose the public include directory
target_include_directories(VrootKV PUBLIC../include)
```

- **Task 4.3.4: Configure tests/CMakeLists.txt**
  - **Description:** Set up Google Test and the test executable.[12]
  - **Deliverable:** tests/CMakeLists.txt with:

```
CMake
# Download and configure Google Test
include(FetchContent)
FetchContent_Declare(googletest URL
https://github.com/google/googletest/archive/03597a01ee50ed33e9dfd640b249b4be
3799d395.zip)
FetchContent_MakeAvailable(googletest)

# Gather all test source files
file(GLOB_RECURSE TEST_SOURCES "*.cpp")

# Create the test executable
add_executable(run_tests ${TEST_SOURCES})

# Link against our library and GTest
target_link_libraries(run_tests PRIVATE VrootKV GTest::gtest_main)

# Discover tests with CTest
```

```
include(GoogleTest)
gtest_discover_tests(run_tests)
```

- **Task 4.3.5: Write End-to-End Integration and Stress Tests**
  - **Description:** Create comprehensive tests that simulate real-world application workloads, including concurrent transactions, high write volumes, and crash-recovery scenarios.
  - **Verification:** The database must remain consistent and correct under high concurrent load and after simulated crashes.

Works cited

1. Setup Mac OS for C++, CMake development using Visual Studio Code - Medium, accessed September 4, 2025, https://medium.com/@divyendu.narayan/visual-studio-code-setup-in-mac-os-to-build-and-debug-c-cmake-projects-45a78b29e49
2. How to Structure C Projects: These Best Practices Worked for Me | Blog - Luca Cavallin, accessed September 4, 2025, https://www.lucavall.in/blog/how-to-structure-c-projects-my-experience-best-practices
3. Best Practices for Structuring C++ Projects (Industry Standards) | by Govind Yadav - Medium, accessed September 4, 2025, https://medium.com/@gtech.govind2000/best-practices-for-structuring-c-projects-industry-standards-71b82f6b145c
4. Structuring a C++ Project Effectively - slaptijack, accessed September 4, 2025, https://slaptijack.com/programming/project-structure-for-cpp.html
5. Filesystem Home - Boost, accessed September 4, 2025, https://www.boost.org/libs/filesystem/
6. Filesystem library (since C++17) - cppreference.com - C++ Reference, accessed September 4, 2025, https://en.cppreference.com/w/cpp/filesystem.html
7. Modern C++ Project Structuring | Gregory Kelleher, accessed September 4, 2025, https://gregorykelleher.com/interview_practice_questions
8. C++ Atomic Tutorial - Complete Guide - GameDev Academy, accessed September 4, 2025, https://gamedevacademy.org/c-atomic-tutorial-complete-guide/
9. Atomics and Concurrency in C++ - freeCodeCamp, accessed September 4, 2025, https://www.freecodecamp.org/news/atomics-and-concurrency-in-cpp/
10. C++ 11 -
11. std::atomic - cppreference.com, accessed September 4, 2025, https://en.cppreference.com/w/cpp/atomic/atomic.html
12. Quickstart: Building with CMake | GoogleTest, accessed September 4, 2025, http://google.github.io/googletest/quickstart-cmake.html
13. Testing with Google Test - CMake-Episode 022 - YouTube, accessed September 4, 2025, https://www.youtube.com/watch?v=W5jxjTsHXuA