code basicstyle=, frame=single, rulecolor=, breaklines=true, showstringspaces=false, columns=fullflexible, keepspaces=true, framesep=6pt, aboveskip=1em, belowskip=1em, xleftmargin=0pt

out basicstyle=, frame=tb, rulecolor=, breaklines=true, showstringspaces=false, columns=fullflexible, keepspaces=true, framesep=4pt, aboveskip=1em, belowskip=1em

Arsitektur Berlapis Sistem Kontrol Versi dalam Rust Membangun libvctrl dari Nol

Muhammad Riduwan Khafidi
Program Studi Teknik Rekayasa Perangkat Lunak
Politeknik Negeri Batam
4342611034@students.polibatam.ac.id

November 5, 2026

Abstrak

Sistem kontrol versi modern seperti Git menopang kolaborasi perangkat lunak global, namun arsitektur internalnya jarang dibedah dalam literatur berbahasa Indonesia. Makalah ini membedah libvctrl, sebuah workspace Rust dengan enam crate publik yang membangun sistem kontrol versi dari nol menggunakan arsitektur berlapis yang ketat. Makalah menyajikan delapan visualisasi TikZ yang mencakup pemetaan lapisan, graf dependensi, model objek, pohon Merkle, siklus content addressing, perbandingan penyimpanan, pipeline kodek, dan timeline studi kasus. Setiap lapisan dibahas dengan analogi konkret, contoh berangka, dan penjelasan mengapa pemisahan kontrak dari implementasi menjadi fondasi yang dapat diuji secara independen. Melalui kerangka ini, pembaca memperoleh mental model untuk membangun sistem serupa atau membaca kode sumber VCS mana pun.

1 Pendahuluan

1.1 Analogi Pembuka: Pabrik Elektronik dengan Tujuh Divisi

Bayangkan sebuah pabrik elektronik yang memproduksi ponsel. Pabrik itu punya tujuh divisi yang bekerja berurutan, tetapi dengan batas yang tegas. Divisi pertama menetapkan standar: ukuran sekrup harus M2, bukan M3; ketebalan baterai harus 4 milimeter; antena harus punya impedansi 50 ohm. Divisi kedua merancang rangkaian sesuai standar itu. Divisi ketiga membuat komponen dasar: resistor, kapasitor, chip. Divisi keempat merakit papan sirkuit. Divisi kelima mengemas ponsel jadi unit siap jual. Divisi keenam melayani pelanggan: panduan, garansi, servis. Divisi ketujuh mengelola pabrik secara keseluruhan.

Yang penting: divisi pertama tidak peduli ponselnya warna apa. Divisi ketiga tidak peduli ponselnya dijual ke mana. Divisi keenam tidak peduli chip apa yang dipakai. Setiap divisi hanya berkomunikasi dengan divisi tetangga melalui kontrak tertulis. Jika chip diganti dari ARM ke RISC-V, divisi perakitan tetap sama selama pin dan tegangan sesuai kontrak. Jika layanan pelanggan ditambah fitur, divisi perakitan tidak perlu disentuh.

libvctrl mengikuti pola yang sama untuk sistem kontrol versi. Enam crate publik dan satu backend opsional membentuk tujuh divisi yang tidak saling mencampuri.

1.2 Mengapa Topik Ini Penting

Seorang pengembang yang mengerti cara Git disusun bisa mendiagnosis masalah dengan lebih cepat. Ketika `git fsck` melaporkan *dangling commit*, pengembang yang paham pohon objek bisa menelusuri ke mana commit itu seharusnya menempel. Ketika `git log` terasa lambat, pengembang yang paham `ObjectStore` bisa curiga pada konkatenasi file di `.git/objects`. Tanpa mental model arsitektur, semua diagnosis berhenti pada tebakan.

Selain itu, arsitektur berlapis adalah pola yang dapat dipakai di domain lain. Sistem basis data memisahkan parser dari planner dari executor dari storage. Sistem build memisahkan dependensi dari kompilasi dari linking. Sistem paket memisahkan manifest dari resolver dari fetcher dari installer. Setiap domain punya masalah yang sama: bagaimana memisahkan aturan dari implementasi agar satu bagian bisa diganti tanpa merusak yang lain.

Rust sebagai bahasa implementasi memberi jaminan keamanan memori yang tidak dimiliki C, bahasa utama Git. libvctrl memakai Rust untuk alasan yang sama seperti sistem lain memilih Rust pada 2024 ke atas: menghilangkan kelas bug memori tanpa mengorbankan performa. Kombinasi Rust dan arsitektur berlapis adalah kombinasi yang layak dipelajari.

1.3 Peta Pembahasan

Makalah ini dibagi menjadi sembilan bagian utama. Bagian kedua membahas tujuh lapisan arsitektur dan mengapa pemisahan kontrak dari implementasi menjadi fondasi. Bagian ketiga membahas model objek VCS: `Blob`, `Tree`, `Commit`, `Tag`, dan `Hash`. Bagian keempat membahas content addressing dengan SHA-512 dan mengapa fungsi hash kriptografis dipakai alih-alih CRC. Bagian kelima membahas kodek biner dan backend penyimpanan. Bagian keenam menyandingkan seluruh lapisan dalam tabel perbandingan. Bagian ketujuh memuat studi kasus delapan langkah membangun commit pertama. Bagian kedelapan dan kesembilan memberi daftar kesalahan umum dan lembar rujukan cepat. Bagian kesepuluh menutup dengan tiga prinsip.

2 Tujuh Lapisan Arsitektur

2.1 Mengapa Tujuh, Bukan Tiga

Sebuah sistem kontrol versi bisa saja ditulis sebagai satu binary monolitik. Itu pilihan yang lebih sederhana pada awalnya, tetapi membawa tiga biaya yang tidak terlihat sampai sistem membesar.

Pertama, pengujian menjadi lambat. Setiap kali ingin menguji `ObjectStore`, harus ada filesystem sungguhan. Setiap kali ingin menguji `Decoder`, harus ada encoder yang sudah jalan. Tidak ada cara menguji komponen secara terisolasi.

Kedua, penggantian backend menjadi mustahil. Jika ingin menyimpan objek di SQLite sebagai ganti filesystem, seluruh kode perintah di atas harus ditulis ulang karena tidak ada batas yang

jelas antara "cara menyimpan" dan "cara memakai".

Ketiga, dokumentasi menjadi kabur. Pembaca yang ingin memahami apa itu `ObjectStore` harus membaca kode penyimpanan, bukan membaca kontrak.

Tujuh lapisan libvctrl menjawab ketiga masalah itu dengan satu prinsip: setiap lapisan hanya bergantung ke lapisan di bawahnya, dan hanya melalui kontrak tertulis. Gambar **??** memetakan ketujuh lapisan itu.

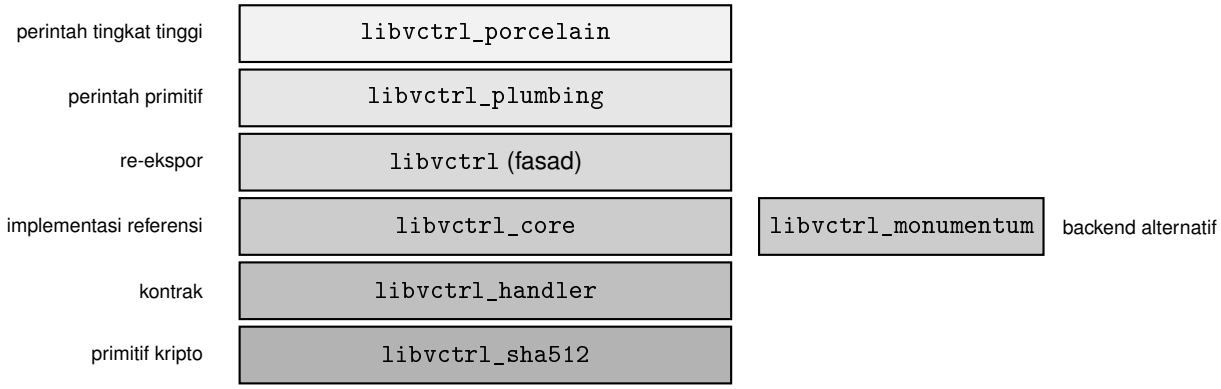


Figure 1. Tujuh lapisan arsitektur libvctrl. Enam lapisan membentuk tumpukan utama; `libvctrl_monumentum` adalah backend opsional yang duduk sejajar dengan `libvctrl_core`.

Cara membaca Gambar **??**: gambar ini menampilkan enam kotak bertumpuk di sisi kiri, dari `libvctrl_porcelain` di atas sampai `libvctrl_sha512` di bawah. Setiap kotak punya latar abu-abu yang sedikit lebih gelap dari kotak di atasnya. Di sebelah kanan tumpukan, sejajar dengan kotak `libvctrl_core`, ada kotak ketujuh berlabel `libvctrl_monumentum` yang lebih sempit. Di sisi kiri setiap kotak ada label peran singkat. Inti pesan gambar ini adalah tujuh lapisan dengan tanggung jawab terpisah, di mana backend alternatif dapat menggantikan lapisan implementasi tanpa mengubah lapisan lain.

2.2 Peran Setiap Lapisan

Lapisan `libvctrl_sha512` hanya berisi primitif kriptografi: SHA-512, HMAC-SHA512, HKDF-SHA512, dan SHA-384 opsional. Ia tidak tahu apa itu commit, tree, atau blob. Ia hanya tahu cara mengubah byte menjadi hash 64 byte.

Lapisan `libvctrl_handler` hanya berisi kontrak: trait, tipe data, konstanta, dan fungsi validasi. Ia tidak berisi satu pun implementasi konkret. Ia mendefinisikan apa itu `Blob`, apa itu `ObjectStore`, apa itu `Encoder`, tanpa menentukan bagaimana semuanya diwujudkan.

Lapisan `libvctrl_core` berisi implementasi referensi: kodek biner, builder, hasher adapter, dan penyimpanan in-memory. Ia mengimplementasikan setiap trait dari `libvctrl_handler`.

Lapisan `libvctrl` adalah fasad. Ia tidak berisi logika apa pun; setiap item publiknya adalah re-ekspor. Tujuannya hanya menyatukan tiga crate fondasi (`handler`, `core`, `sha512`) dalam satu namespace.

Lapisan `libvctrl_plumbing` berisi perintah primitif seperti `cat-file`. Ia memakai `libvctrl_core` sebagai backend, tetapi tidak menyembunyikan langkah apa pun.

Lapisan `libvctrl_porcelain` berisi perintah tingkat tinggi yang membungkus plumbing menjadi alur kerja yang mudah dipakai. Contohnya adalah `commit` yang melakukan encode, hash, dan store sekaligus.

Lapisan `libvctrl_monumentum` adalah backend alternatif: implementasi `ObjectStore` dan `RefStore` yang menyimpan data di mesin penyimpanan `monumentum`, bukan di `HashMap`. Ia duduk sejajar dengan `libvctrl_core` karena sama-sama mengimplementasikan kontrak dari `libvctrl_handler`.

2.3 Arah Dependensi yang Satu Arah

Arah dependensi bersifat satu arah dan wajib. Gambar **??** menunjukkan graf lengkapnya.

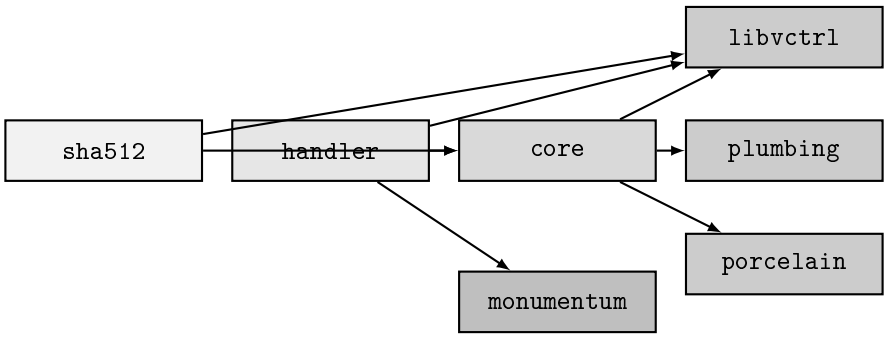


Figure 2. Graf dependensi antar crate libvctrl. Panah hanya mengalir satu arah dan tidak pernah membentuk siklus.

Cara membaca Gambar **??**: gambar ini menampilkan tujuh kotak bernama crate. `sha512` dan `handler` di paling kiri tidak punya panah masuk, artinya keduanya adalah fondasi yang tidak bergantung ke crate lain. `core` menerima panah dari keduanya. `libvctrl` menerima panah dari tiga crate fondasi sekaligus. `plumbing` dan `porcelain` menerima panah dari `core`. `monumentum` menerima panah dari `handler`. Tidak ada panah yang mundur. Inti pesan gambar ini adalah arah dependensi ketat satu arah dan tidak ada siklus.

Arah satu arah ini bukan sekadar konvensi. Jika `handler` sampai bergantung ke `core`, seluruh sistem akan menjadi satu kesatuan monolitik dan kehilangan seluruh manfaat pemisahan. Jika `core` sampai bergantung ke `plumbing`, maka perintah-perintah di atasnya tidak dapat diuji secara terpisah. Arah dependensi yang satu arah adalah fondasi dari seluruh klaim modularitas arsitektur ini.

2.4 Mengapa Kontrak Terpisah dari Implementasi

Lapisan `libvctrl_handler` hanya berisi trait, tipe, konstanta, dan fungsi validasi. Keputusan ini punya tiga konsekuensi.

Pertama, pengujian menjadi mungkin tanpa backend. Setiap kali ingin menguji `ObjectStore`, cukup menulis struct kosong yang mengimplementasikan trait dengan perilaku minimal. Pengujian berjalan dalam hitungan milidetik, bukan detik.

Kedua, penggantian backend menjadi murah. Ketika `libvctrl_monumentum` ditulis, ia hanya perlu mengimplementasikan dua trait: `ObjectStore` dan `RefStore`. Seluruh kode perintah di atasnya tidak tahu bahwa backend berubah.

Ketiga, dokumentasi menjadi jelas. Setiap trait punya docstring yang menjelaskan kontraknya tanpa perlu menjelaskan implementasinya.

2.5 Ringkasan Tujuh Crate

Table 1. Ringkasan tujuh crate pada workspace libvctrl.

Crate	Peran	Dependensi
`libvctrl_sha512`	Primitif kriptografi (SHA, HMAC, HKDF)	tidak ada
`libvctrl_handler`	Kontrak: trait, tipe, validasi	tidak ada
`libvctrl_core`	Implementasi referensi	handler, sha512
`libvctrl`	Fasad: re-ekspor tiga crate fondasi	handler, core, sha512
`libvctrl_plumbing`	Perintah primitif	core
`libvctrl_porcelain`	Perintah tingkat tinggi	plumbing
`libvctrl_monumentum`	Backend penyimpanan alternatif	handler, monumentum

3 Model Objek VCS

3.1 Mengapa Empat Tipe, Bukan Satu

Jika seluruh data disimpan sebagai byte stream tanpa tipe, sistem tidak bisa membedakan file dari direktori, atau direktori dari commit. Setiap operasi harus menebak tipe dari isinya, yang lambat dan rawan salah. Object store modern memecah data menjadi empat tipe dengan peran jelas: `Blob`, `Tree`, `Commit`, dan `Tag`. Gambar **??** menunjukkan keempatnya.

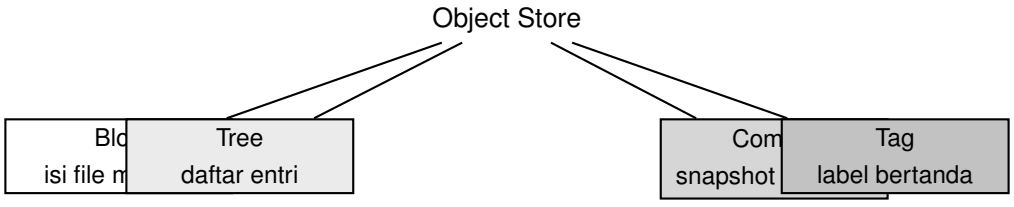


Figure 3. Empat tipe objek pada content-addressable object store.

Cara membaca Gambar **??**: gambar ini menampilkan satu kotak pusat berlabel Object Store, dengan empat kotak bercabang ke bawah. Cabang paling kiri berlabel Blob dengan keterangan isi file mentah. Cabang kedua berlabel Tree dengan keterangan daftar entri. Cabang ketiga berlabel Commit dengan keterangan snapshot dan parent. Cabang paling kanan berlabel Tag dengan keterangan label bertanda. Setiap cabang punya latar abu-abu yang berbeda, dari putih

ke abu-abu tua. Inti pesan gambar ini adalah object store dibagi menjadi empat tipe dengan tanggung jawab berbeda.

3.2 Blob: Isi File Tanpa Nama

`Blob` adalah singkatan dari *binary large object*. Isinya adalah byte mentah dari sebuah file, tanpa nama, tanpa path, tanpa metadata apa pun. Blob tidak tahu namanya sendiri; ia hanya tahu isinya.

Pemisahan nama dari isi adalah kunci deduplikasi. Jika dua file bernama `notes.txt` dan `backup.txt` berisi hal yang sama persis, keduanya merujuk ke blob yang sama. Sistem menyimpan satu blob, bukan dua. Karena blob tidak menyimpan nama, cara sistem menemukan blob adalah melalui `Tree`.

Konstruksi blob memakai konstruktor fallible. Jika isi melebihi batas `MAX_BLOB_SIZE`, konstruksi gagal dan mengembalikan `VctrlError`. Inilah contoh bagaimana kontrak dipaksakan di batas sistem.

[style=code] use libvctrl::Blob, VctrlError;

fn main() -> Result<(), VctrlError> let data = b"hello, vcs".to$_{v}ec();letblob = Blob :: new(data)?; println!("size = ", blob.size()); Ok(())

[style=out] size = 10

Jika isi melebihi `MAX_BLOB_SIZE` (100 MiB), `Blob::new` mengembalikan `ExceededMaxSize`. Pemanggil tidak bisa membuat blob raksasa secara tidak sengaja.

3.3 Tree: Daftar Entri Direktori

`Tree` adalah satu tingkat direktori. Isinya adalah daftar entri, di mana setiap entri memiliki tiga komponen: nama, `EntryKind` (file biasa, executable, symlink, subdirektori, submodule), dan hash dari objek target.

Tree adalah tempat nama dan isi dipertemukan kembali. Sebuah blob bisa dirujuk oleh banyak tree dengan nama berbeda, dan sebuah tree bisa merujuk banyak blob dengan nama berbeda. Pada konstruksi tree, entri diurutkan dengan aturan yang sama seperti Git: nama diurutkan berdasarkan byte, dan jika nama sama persis, tree diurutkan seolah namanya ditambahi `/`.

[style=code] use libvctrl::EntryKind, Hash, Tree, TreeEntry;

let hash = Hash::from$_{b}ytes([0u8; 64])?; lete = TreeEntry :: new("a.txt".to$_{s}tring(), EntryKind :: Blob, hash)?; lettree = Tree :: new(vec![e])?; println!("entries = ", tree.len());

[style=out] entries = 1

3.4 Commit: Snapshot dengan Sejarah

`Commit` adalah snapshot dari seluruh tree di root proyek pada satu titik waktu, ditambah satu atau lebih parent yang menunjuk ke commit sebelumnya. Sebuah commit menyimpan hash tree, daftar parent, author, committer, message, dan metadata opsional seperti timestamp dan encoding.

Perbedaan antara tree dan commit adalah perbedaan antara "bagaimana isi direktori saat ini" dan "mengapa isi itu seperti ini". Tree hanya menyimpan keadaan. Commit menyimpan keadaan plus alasannya.

3.5 Tag: Label yang Bisa Ditandatangani

`Tag` adalah label yang menunjuk ke objek lain, biasanya commit. Ada dua jenis: lightweight dan annotated. Lightweight hanyalah alias, sedangkan annotated adalah objek tersendiri yang menyimpan nama tag, hash target, tagger, pesan, dan tanda tangan kriptografis opsional.

Annotated tag adalah satu-satunya objek yang dirancang untuk ditandatangani. Tanda tangannya membuktikan bahwa tag dibuat oleh pemegang kunci privat, bukan oleh siapa pun yang bisa menulis ke repositori.

3.6 Hash sebagai Alamat

`Hash` adalah tipe fixed-size 64 byte. Ia mewakili alamat konten dari sebuah objek, dihitung dengan SHA-512 atas byte hasil encode objek tersebut. Karena hash adalah tipe `Copy`, ia dapat dipakai sebagai kunci pada `HashMap` tanpa cloning.

Hash menyimpan array `[u8; HASH_LENGTH]`, yang membuat setiap nilai hash punya ukuran yang sama persis. Ukuran tetap inilah yang membuat `HashMap` dapat menyimpan jutaan objek tanpa fragmentasi.

3.7 Pohon Merkle: Mengapa Satu Byte Mengubah Semuanya

Setiap objek pada object store memiliki hash. Yang membuat struktur ini kuat adalah kenyataan bahwa tree menyimpan hash dari anak-anaknya, dan commit menyimpan hash dari root tree. Artinya, hash sebuah commit secara transitif mencakup seluruh isi repositori pada saat commit dibuat. Ubah satu byte pada satu file, dan hash dari blob, tree, commit, dan semua commit turunan berubah.

Struktur ini disebut pohon Merkle. Gambar **??** menunjukkan bagaimana perubahan pada satu blob merambat naik sampai ke commit teratas.

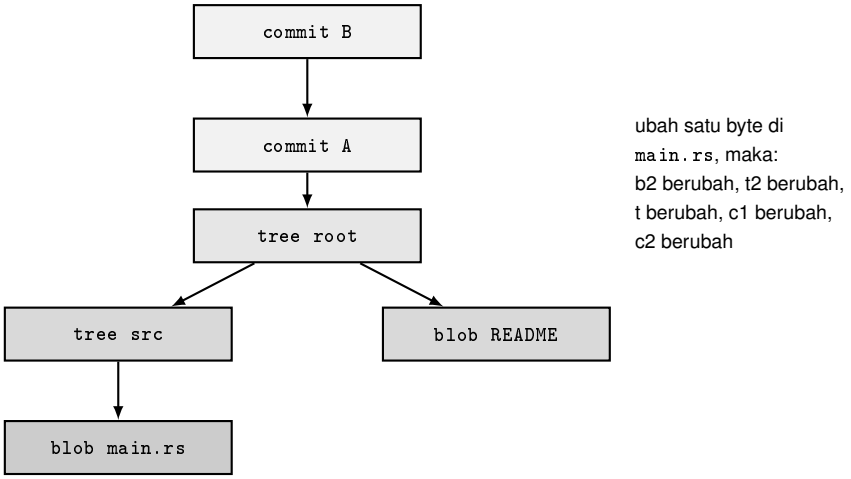


Figure 4. Pohon Merkle pada object store. Perubahan pada satu blob merambat sampai ke commit teratas.

Cara membaca Gambar **??**: gambar ini menampilkan enam kotak bernama objek dengan panah dari anak ke induk. Blob main.rs di bawah, tree src di atasnya, lalu tree root, commit A, dan commit B di paling atas. Kotak blob README ada di sejajar tree src, juga bercabang ke tree root. Di sebelah kanan ada keterangan bahwa perubahan satu byte di main.rs merambat naik lewat setiap level. Inti pesan gambar ini adalah hash di level atas bergantung transitif pada hash di level bawah.

3.8 Ringkasan Tipe Objek

Table 2. Ringkasan empat tipe objek dan hash.

Tipe	Isi utama	Peran
`Blob`	byte mentah satu file	menyimpan isi tanpa nama
`Tree`	daftar nama, mode, hash	memetakan nama ke isi
`Commit`	tree, parent, metadata	merekam snapshot dan sejarah
`Tag`	target, tagger, pesan	label permanen yang bisa ditandatangani
`Hash`	64 byte hasil SHA-512	alamat konten seluruh objek

4 Content Addressing dengan SHA-512

4.1 Mengapa Content Addressing, Bukan Address-Based

Sistem berkas konvensional memberi alamat pada file berdasarkan nomor inode, path, atau offset. Alamat itu stabil selama file tidak dipindahkan, tetapi tidak memberi informasi apa pun tentang isi. Sistem kontrol versi membutuhkan tiga hal yang tidak bisa diberikan oleh alamat konvensional.

Pertama, deduplikasi. Jika dua commit menyimpan file identik, sistem tidak perlu menyimpannya dua kali. Dengan alamat berbasis isi, keduanya memiliki alamat yang sama, dan penyimpanan kedua tidak melakukan apa-apa.

Kedua, verifikasi integritas. Jika file di disk rusak, sistem bisa menghitung ulang hash dan membandingkannya dengan alamat. Ketidakcocokan menandakan korupsi.

Ketiga, immutability. Objek yang alamatnya ditentukan oleh isi tidak bisa diubah tanpa mengubah alamatnya. Sistem bisa mengasumsikan bahwa objek dengan alamat tertentu selalu memiliki isi yang sama, di mana pun dan kapan pun.

4.2 Siklus Hidup Satu Objek

Proses penyimpanan satu objek pada content-addressable storage mengikuti lima langkah tetap. Gambar **??** menunjukkan siklusnya.

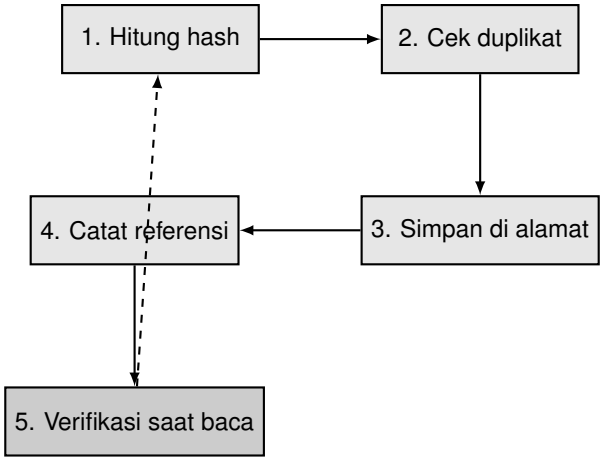


Figure 5. Siklus lima tahap penyimpanan objek pada content-addressable storage.

Cara membaca Gambar **??**: gambar ini menampilkan lima kotak yang tersusun dalam pola empat di atas dan satu di bawah. Kotak pertama di kiri atas berlabel Hitung hash, kotak kedua di kanan atas berlabel Cek duplikat, kotak ketiga di kanan bawah berlabel Simpan di alamat, kotak keempat di kiri bawah berlabel Catat referensi, dan kotak kelima di bawah lagi berlabel Verifikasi saat baca. Panah mengalir dari kiri atas ke kanan atas, lalu turun, lalu ke kiri, lalu turun lagi. Ada panah putus-putus dari kotak kelima kembali ke kotak pertama. Inti pesan gambar ini adalah siklus penyimpanan berulang setiap kali objek dibaca kembali.

4.3 Implementasi dengan SHA-512

libvctrl memakai SHA-512, bukan SHA-256, karena SHA-512 pada perangkat 64-bit justru lebih cepat daripada SHA-256. Ukuran outputnya 64 byte, yang pas dengan `HASH_LENGTH`. Tipe `Hash` di `libvctrl_handler` adalah `[u8; 64]`.

[style=code] use libvctrl::Hash;

let raw = [0xAB$_u$8; 64]; $let hash = Hash :: from_bytes(raw)?; println!("", hash);$

[style=out] abababababababababababababababababababab abababababababababababababababababababa

Representasi hex untuk hash 64 byte selalu 128 karakter. Representasi ini dipakai pada output terminal dan pada nama file objek di disk.

4.4 Pipeline Encode, Hash, Store, Decode

Satu putaran penuh pada object store mengikuti pola tetap. Gambar **??** menunjukkan pipeline-nya.

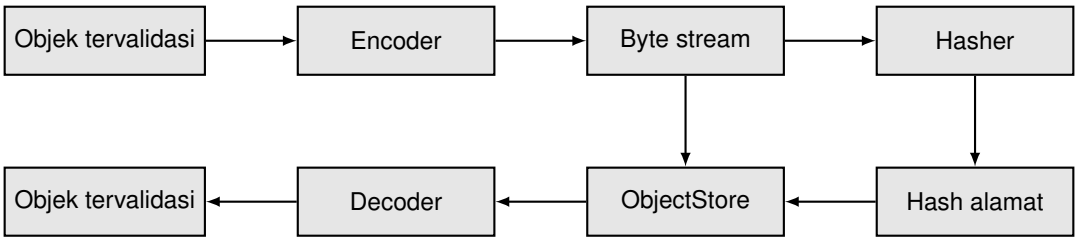


Figure 6. Pipeline encode, hash, store, dan decode. Encoder dan Decoder adalah dua sisi dari kodek biner yang sama.

Cara membaca Gambar **??**: gambar ini menampilkan empat kotak di baris atas dan empat kotak di baris bawah. Baris atas dari kiri ke kanan berisi Objek tervalidasi, Encoder, Byte stream, dan Hasher. Baris bawah dari kanan ke kiri berisi Hash alamat, ObjectStore, Decoder, dan Objek tervalidasi. Panah mengalir dari kiri atas ke kanan atas, lalu turun ke hash alamat, lalu ke object store, lalu ke decoder, lalu ke objek tervalidasi di kiri bawah. Inti pesan gambar ini adalah satu siklus penuh mengubah objek menjadi byte, byte menjadi hash, byte disimpan, lalu byte dibaca dan diubah kembali menjadi objek.

4.5 Mengapa Hash Kriptografis, Bukan CRC

CRC32 dan MD5 sengaja tidak dipakai. Alasannya ada tiga.

Pertama, tabrakan. CRC32 punya 32 bit, MD5 punya 128 bit tetapi sudah dianggap rusak untuk keperluan kriptografis. Untuk repositori dengan miliaran objek, tabrakan MD5 menjadi masalah nyata. SHA-512 punya 512 bit, cukup untuk seluruh repositori dalam beberapa abad.

Kedua, ketahanan terhadap upaya membalik. Untuk keperluan alamat, kita ingin fungsi hash yang tidak bisa dibalik tanpa memegang input aslinya. SHA-512 memenuhi syarat ini; CRC32 tidak.

Ketiga, ketahanan terhadap perubahan kecil. Perubahan satu bit pada input harus mengubah sekitar separuh output. SHA-512 memenuhi efek avalanche ini dengan baik; CRC32 juga memenuhinya, tetapi tidak cukup besar untuk mencegah tabrakan.

4.6 Mengapa Batas Resource Wajib Ada

Decoder membaca dari stream yang mungkin berasal dari input berbahaya. Tanpa batas, satu byte header bisa menyatakan bahwa blob berukuran 100 GB, dan proses akan mencoba

mengalokasikan 100 GB sebelum menyadari bahwa stream sebenarnya hanya berisi 10 byte. Ini adalah serangan denial-of-service klasik.

libvctrl menambahkan batas keras pada seluruh tipe objek. Tabel **??** merangkumnya.

Table 3. Konstanta batas resource pada `libvctrl_handler`.

Konstanta	Nilai	Tujuan
`HASH_LENGTH`	64 byte	ukuran hash SHA-512
`MAX_NAME_LENGTH`	255 byte	batas nama file dan referensi
`MAX_BLOB_SIZE`	100 MiB	DoS breaker pada blob
`MAX_TREE_ENTRIES`	100 ribu	batas jumlah entri per tree
`MAX_MESSAGE_LENGTH`	1 MiB	batas pesan commit dan tag
`MAX_PARENT_COUNT`	65535	batas parent (u16)

Setiap batas dipaksakan di konstruktor. Blob yang terlalu besar tidak bisa dibuat; tree yang terlalu panjang tidak bisa dirakit; commit dengan terlalu banyak parent ditolak dengan `ExceededMaxSize`. Pemaksaan di konstruktor berarti objek yang melewati batas tidak pernah ada di sistem.

4.7 Immutability sebagai Konsekuensi

Immutability bukan fitur yang ditambahkan; ia adalah konsekuensi langsung dari alamat berbasis hash. Jika objek dengan alamat H bisa diubah isinya, maka hash dari isi yang baru tidak akan sama dengan H, dan alamat itu tidak lagi merujuk objek tersebut. Dengan kata lain, "mengubah objek" berarti "membuat objek baru dengan alamat baru"; objek lama tetap ada dengan alamat lamanya. Gambar **??** menyandingkan sifat mutable dan immutable pada sistem penyimpanan.

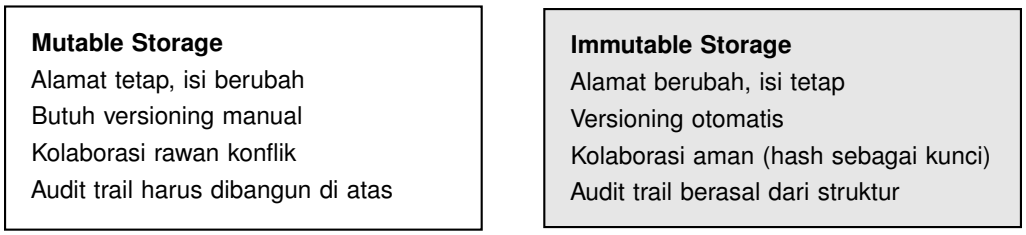


Figure 7. Perbandingan mutable dan immutable storage.

Cara membaca Gambar **??**: gambar ini menampilkan dua kotak sejajar. Kotak kiri berlabel Mutable Storage dengan empat poin: alamat tetap isi berubah, butuh versioning manual, kolaborasi rawan konflik, audit trail harus dibangun di atas. Kotak kanan berlabel Immutable Storage dengan empat poin berlawanan: alamat berubah isi tetap, versioning otomatis, kolaborasi aman dengan hash sebagai kunci, audit trail berasal dari struktur. Kotak kiri berlatar putih dan kotak kanan berlatar abu-abu muda. Inti pesan gambar ini adalah dua sifat penyimpanan yang berlawanan dan konsekuensinya pada versioning, kolaborasi, dan audit trail.

5 Kodek Biner dan Backend Penyimpanan

5.1 Mengapa Kodek Biner, Bukan JSON

Menulis objek VCS sebagai JSON terlihat menggoda karena mudah dibaca manusia dan mudah diparse. Tetapi ada tiga alasan mengapa libvctrl memakai kodek biner.

Pertama, determinisme. JSON tidak mendefinisikan urutan key, whitespace, dan representasi float. Dua serializer JSON dapat menghasilkan byte yang berbeda untuk objek yang sama, dan hash dari dua hasil itu akan berbeda. Kodek biner libvctrl menetapkan setiap field dengan urutan tetap, integer little-endian, dan string dengan prefix panjang. Byte hasil encode deterministik.

Kedua, ukuran. JSON membawa banyak karakter tambahan: kurung kurawal, tanda kutip, koma, dan nama field yang diulang setiap objek. Kodek biner tidak. Untuk repositori dengan jutaan objek, selisih ini menjadi ratusan megabyte.

Ketiga, tipe. JSON tidak membedakan integer dengan string, dan tidak membedakan byte biner dengan string UTF-8. Kodek biner libvctrl menetapkan tipe setiap field, sehingga decoder dapat menolak input yang salah tipe sebelum sampai ke konstruktor.

5.2 Header Versioning

Setiap objek yang dikodekan dimulai dengan satu byte version. Byte ini wajib bernilai 3 pada implementasi saat ini. Decoder menolak input yang byte pertamanya tidak bernilai 3, sehingga format bisa berubah di masa depan tanpa harus mempertahankan kompatibilitas mundur pada decoder yang sama.

[style=code] use libvctrl::Blob, Encoder, BinaryEncoder;

let blob = Blob::new(b"abc".to$_{v}ec()$)?; $letmutbuf = Vec ::$ $new()$; $BinaryEncoder.encode_{b}lob(blob, mutbuf)$?; $println!("bytes = :?", buf)$;

[style=out] bytes = [3, 3, 0, 0, 0, 0, 0, 0, 0, 97, 98, 99]

Byte pertama adalah version (3). Delapan byte berikutnya adalah panjang data dalam little-endian (3). Tiga byte terakhir adalah isi data itu sendiri (a, b, c). Total dua belas byte. Format ini berbeda dari Git yang memakai format teks header bertipe ("blob 300abc"), dan perbedaan ini disengaja agar proses decode tidak perlu memindai untuk menemukan batas header.

5.3 Mengimplementasikan ObjectStore Sendiri

Salah satu klaim utama arsitektur libvctrl adalah backend bisa diganti. Untuk membuktikan klaim itu, cukup mengimplementasikan empat method dari trait `ObjectStore`.

[style=code] use std::io::self, Read; use libvctrl::Hash, ObjectStore, VctrlError;

struct NullStore;

impl ObjectStore for NullStore fn put(mut self, $_h : Hash, _d : [u8]) -> Result<(), VctrlError> Ok(()) fnget(self, _h : Hash) -> Result<Box<dynRead + Send + '_>, VctrlError> Ok(Box :: new(io :: empty())) fndelete(mutself, _h : Hash) -> Result<(), VctrlError> Ok(()) fnexists(self, _h : Hash) -> Result<bool, VctrlError> Ok(false)$

Implementasi ini tidak menyimpan apa pun, tetapi ia memenuhi kontrak dan dapat diserahkan ke `Repository::new` tanpa perubahan kode lain. Pengujian bisa memakai `NullStore` untuk memverifikasi alur tanpa menyentuh filesystem.

5.4 Backend Monumentum

`libvctrl_monumentum` adalah backend alternatif yang menyimpan objek di mesin `monumentum`. Ia mengimplementasikan dua trait (`ObjectStore` dan `RefStore`) dengan memetakan hash dan data ke tabel-tabel `monumentum`. Pemakai `Repository` tidak perlu tahu perbedaan antara `MemoryStore`, `MonumentumStore`, dan backend kustom; semuanya diserahkan sebagai `Box<dyn ObjectStore>`.

Pola ini adalah alasan utama arsitektur berlapis: seseorang yang ingin menyimpan objek di PostgreSQL, SQLite, atau S3 tidak perlu menulis ulang `Repository`. Ia cukup mengimplementasikan trait.

5.5 Ringkasan Format Kodek

Table 4. Ringkasan format kodek biner versi 3.

Field	Format
Version	1 byte, wajib bernilai 3
Hash	64 byte (SHA-512)
Integer	little-endian
Panjang	u8, u16, u32, atau u64 sesuai kebutuhan
String	prefix panjang (u8 atau u32) + UTF-8
Tree entry	panjang nama + nama + kind + hash
Commit parent	u16 jumlah + hash per parent
Optional field	1 byte penanda + isi jika ada

6 Perbandingan dan Sintesis

Setelah memahami tujuh lapisan, model objek, content addressing, dan kodek, seluruh keputusan arsitektur dapat diringkas dalam satu tabel. Tabel **??** menyandingkan tujuh aspek.

Table 5. Sintesis keputusan arsitektur libvctrl.

Aspek	Keputusan	Alasan
Lapisan	tujuh crate terpisah	pengujian dan penggantian backend
Arah dependensi	satu arah, tanpa siklus	isolasi perubahan
Kontrak	trait + tipe immutable	jaminan konstruktor
Validasi	konstruktor fallible	invalid state tidak ada
Alamat	SHA-512 atas byte encode	deduplikasi + integritas
Format	biner little-endian deterministik	hash stabil, ukuran kecil
Batas resource	konstanta MAX di konstruktor	DoS dicegah sebelum alokasi
Backend	trait ObjectStore + RefStore	penggantian tanpa ubah pengguna

Dua paragraf penutup dari perbandingan ini. Pertama, setiap keputusan adalah respons terhadap masalah konkret, bukan preferensi estetika. Batas resource ada karena serangan DoS pernah terjadi. Format biner ada karena hash stabil butuh determinisme. Kontrak trait ada karena backend SQLite perlu ditambahkan tanpa menyentuh kode perintah.

Kedua, keputusan-keputusan ini saling memperkuat. Format biner deterministik hanya berguna jika tipe data immutable; tipe immutable hanya berguna jika kontrak dipaksakan di konstruktor; kontrak dipaksakan hanya berguna jika tidak ada jalan pintas lewat trait yang sama. Rantai ini adalah alasan arsitektur berlapis bukan sekadar kosmetik.

7 Studi Kasus: Menyimpan Commit Pertama

Bagian ini mengikuti delapan langkah menyimpan satu commit di libvctrl, dimulai dari byte mentah sampai referensi yang bisa dibaca kembali.

7.1 Langkah 1: Menyiapkan Isi File

Isi file disiapkan sebagai byte stream. Nama file disimpan terpisah; blob hanya berisi isi.

[style=code] let content: Vec<u8> = b"hello, vcs".to$_{v}ec()$; $println!(”contentlen = ”, content.len())$;

[style=out] content len = 11

7.2 Langkah 2: Membangun Blob Tervalidasi

Blob dibangun lewat konstruktor fallible. Jika isi melebihi 100 MiB, konstruksi gagal sebelum alokasi besar terjadi.

[style=code] use libvctrl::Blob;

let blob = Blob::new(content)?; println!("blob size = ", blob.size());

[style=out] blob size = 11

7.3 Langkah 3: Mengencode Blob ke Byte Stream

Blob diubah menjadi byte stream deterministik dengan header version. Byte stream inilah yang akan di-hash dan disimpan.

[style=code] use libvctrl::BinaryEncoder, Encoder;

let mut $blob_bytes = Vec :: new(); BinaryEncoder.encode_blob(blob, mut blob_bytes)?; println!("encoded len = ", blob_bytes.len());$

[style=out] encoded len = 20

7.4 Langkah 4: Menghitung Hash dan Menyimpan Blob

Hash dihitung atas byte stream, bukan atas isi file. Byte stream inilah yang disimpan di object store dengan kunci hash.

[style=code] use libvctrl::Hasher, MemoryStore, ObjectStore, Sha512Hasher;

let $blob_hash = Sha512Hasher.hash(blob_bytes.as_slice())?; let mut store = MemoryStore :: new(); store.put(blob_hash, blob_bytes)?; println!("stored blob", blob_hash);$

[style=out] stored blob 4a8b1c...be7f

7.5 Langkah 5: Membangun Tree dengan Satu Entri

Tree menyimpan nama file, mode, dan hash blob. Nama file tinggal di tree, bukan di blob.

[style=code] use libvctrl::EntryKind, Tree, TreeEntry;

let entry = TreeEntry::new("hello.txt".to_string(), $EntryKind :: Blob, blob_hash,)?; let tree = Tree :: new(vec![entry])?; println!("treeentries = ", tree.len());$

[style=out] tree entries = 1

7.6 Langkah 6: Mengencode dan Menghash Tree

Tree diencode, di-hash, dan disimpan dengan pola yang sama seperti blob.

[style=code] let mut $tree_bytes = Vec :: new(); BinaryEncoder.encode_tree(tree, mut tree_bytes)?; let tree_hash = Sha512Hasher.hash(tree_bytes.as_slice())?; store.put(tree_hash, tree_bytes)?; println!("tree", tree_hash);$

[style=out] tree 8f3a5d...c0e2

7.7 Langkah 7: Membangun Commit

Commit menyimpan hash tree, daftar parent (kosong untuk commit pertama), author, committer, dan pesan. Metadata timestamp dan timezone disimpan dalam `CommitMeta`.

[style=code] use libvctrl::Commit, CommitMeta, UserID;

let user = UserID::new("Alice".to$_s tring(), "alice@example.com".to$_s tring(),)?; let meta = CommitMeta :: new(1$_7$00$_0$00$_0$00, 0, None)?; let commit = Commit :: with$_m$eta(tree$_h$ash, vec![], user.clone(), user, "first commit".to$_s$tring(), meta,)?;

7.8 Langkah 8: Mengencode Commit dan Memperbarui Referensi

Commit diencode, di-hash, disimpan, lalu referensi `refs/heads/main` diperbarui agar menunjuk ke hash commit. Setelah langkah ini, seluruh isi repositori dapat dibaca kembali dari referensi.

[style=code] use libvctrl::MemoryRefStore;

let mut commit$_b$ytes = Vec :: new(); BinaryEncoder.encode$_c$ommit(commit, mut commit$_b$ytes)?; let commit$_h$ash = Sha512Hasher.hash(commit$_b$ytes.as$_s$lice())?; store.put(commit$_h$ash, commit$_b$ytes)?;

let mut refs = MemoryRefStore::new(); refs.set$_r$ef("refs/heads/main", commit$_h$ash)?; println!("HEAD− > ", refs.get$_r$ef("refs/heads/main")?);

[style=out] HEAD -> 7d2e9a...41bc

7.9 Timeline Studi Kasus

Gambar **??** merangkum delapan langkah dalam satu garis waktu.

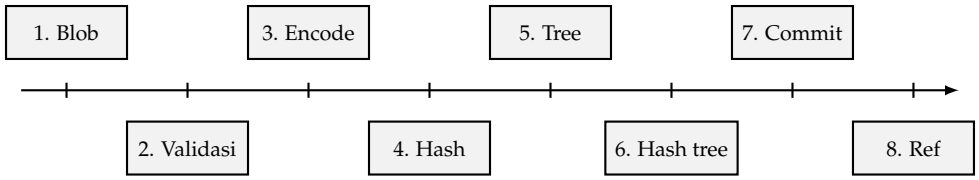


Figure 8. Timeline delapan langkah menyimpan commit pertama.

Cara membaca Gambar **??**: gambar ini menampilkan garis horizontal dengan panah ke kanan. Di sepanjang garis ada delapan tanda dan kotak kecil dengan label bernomor. Kotak label bergantian posisi: satu di atas garis, satu di bawah garis. Nomor dan label: satu Blob, dua Validasi, tiga Encode, empat Hash, lima Tree, enam Hash tree, tujuh Commit, delapan Ref. Inti pesan gambar ini adalah alur linear dari byte mentah sampai referensi.

Table 6. Ringkasan delapan langkah studi kasus.

Langkah	Operasi	Hasil
1	siapkan isi file	11 byte mentah
2	bangun `Blob`	blob tervalidasi
3	encode `Blob`	20 byte dengan header version
4	hash dan simpan	blob tersimpan dengan kunci hash
5	bangun `Tree`	tree dengan satu entri
6	encode dan hash tree	tree tersimpan dengan kunci hash
7	bangun `Commit`	commit dengan satu tree, tanpa parent
8	encode, hash, set ref	referensi menunjuk hash commit

8 Kesalahan Umum dan Cara Menghindarinya

Table 7. Kesalahan umum saat memakai atau membangun di atas libvctrl.

Kesalahan	Akibat	Perbaikan
hash data file mentah, bukan byte encode	hash tidak cocok dengan hash objek	hash selalu atas byte encode
menyimpan blob sebelum menghitung hash	objek tidak bisa ditemukan kembali	hitung hash dulu, lalu simpan
membuat tree tanpa mengurutkan entri	hash berbeda untuk isi sama	pakai `Tree::new` yang sudah mengurutkan
memakai `HashMap<String>` untuk hash	alokasi berlebih dan lambat	pakai `Hash` yang `Copy`
membaca stream tanpa batas	DoS dari input berbahaya	andalkan `read_bounded` di decoder
menyimpan objek kosong tanpa cek batas	alokasi kecil tak terbatas	`Blob::new` sudah menolak melebihi batas
melewati `VctrlError`	informasi hilang di lapisan atas	teruskan error apa adanya
memakai `unwrap` di luar pengujian	panic pada input tidak valid	pakai `?` untuk propagasi
menulis backend tanpa `Send`	backend tidak bisa dipakai lintas thread	tambahkan bound `Send + Sync`
mengganti `Hash` jadi `String`	hash 128 byte per objek, bukan 64	pakai tipe `Hash`

Dua kesalahan yang paling sering muncul adalah menghitung hash atas isi file mentah dan memakai `String` sebagai kunci hash. Yang pertama membuat deduplikasi gagal; file yang sama tetapi dengan metadata berbeda menghasilkan hash berbeda, sehingga dua salinan disimpan. Yang kedua membuat setiap lookup dan perbandingan memakan dua kali ruang; untuk repositori dengan dua puluh juta objek, selisih ini menjadi lebih dari satu gigabyte.

9 Lembar Rujukan Cepat

Table 8. Lembar rujukan cepat untuk bekerja dengan libvctrl.

Kebutuhan	Perintah atau tipe
tambah dependensi fasad	`libvctrl = "2.2"`
tambah dependensi kontrak	`libvctrl_handler = "5.2"`
tambah dependensi kripto	`libvctrl_sha512 = "3.2"`
bangun blob	`Blob::new(bytes)?`
bangun tree	`Tree::new(entries)?`
bangun commit	`Commit::with_meta(...)?`
bangun tag	`Tag::with_meta(...)?`
encode blob	`BinaryEncoder.encode_blob`
encode tree	`BinaryEncoder.encode_tree`
encode commit	`BinaryEncoder.encode_commit`
decode blob	`BinaryDecoder.decode_blob`
hash byte stream	`Sha512Hasher.hash(reader)`
simpan objek in-memory	`MemoryStore.put`
baca objek in-memory	`MemoryStore.get`
simpan referensi	`MemoryRefStore.set_ref`
implementasi backend sendiri	`impl ObjectStore for MyStore`
implementasi ref sendiri	`impl RefStore for MyStore`
cek hash hex	`format!("", hash)`
baca hash dari hex	`hash.parse::<Hash>()`
kompilasi seluruh workspace	`cargo test -workspace`
lint seluruh workspace	`cargo clippy - -D warnings`

10 Kesimpulan

Arsitektur berlapis adalah pola yang memisahkan aturan dari implementasi, dan libvctrl mewujudkannya dalam tujuh crate yang masing-masing hanya bergantung ke lapisan di bawahnya. Lapisan kontrak (`libvctrl_handler`) mendefinisikan apa itu blob, tree, commit, tag, dan hash tanpa menyebut cara menyimpannya. Lapisan implementasi referensi (`libvctrl_core`) mengisi kontrak dengan kodek biner dan penyimpanan in-memory. Lapisan kripto (`libvctrl_sha512`) memberi alamat berbasis isi yang membuat deduplikasi dan verifikasi integritas menjadi bawaan. Lapisan fasad menyatukan ketiganya; lapisan plumbing dan porcelain memberi perintah; lapisan backend alternatif membuktikan bahwa penggantian backend tidak perlu menyentuh kode pengguna.

Tiga prinsip penutup:

1. Kontrak adalah fondasi, bukan dokumentasi. Setiap trait di `libvctrl_handler` harus bisa diimplementasikan oleh crate mana pun tanpa menyentuh crate lain. Jika sebuah trait memaksa pemakai untuk menyentuh `libvctrl_core`, berarti pemisahan kontrak dari implementasi belum tercapai.

2. Alamat adalah isi. Jika dua objek harus dianggap sama, mereka harus memiliki alamat yang sama. Jika mereka harus berbeda, alamatnya harus berbeda. Aturan ini menghi-

langkan hampir semua kerumitan yang biasanya muncul pada sistem deduplikasi manual.

3. Uji di batas, bukan di tengah. Pengujian yang paling berharga adalah pengujian pada konstruktor dan pada decoder, karena di situlah input tidak tepercaya masuk. Pengujian di tengah (misalnya pada perintah porcelain) hanya menguji orkestrasi, bukan kebenaran.

Dengan kerangka ini, pembaca siap membaca kode sumber libvctrl, membangun VCS kecil sendiri di atas model yang sama, atau menambahkan backend baru tanpa menyentuh lapisan yang sudah stabil.

Referensi

References

[1] Klabnik, S., & Nichols, C. (2019). *The Rust Programming Language*. No Starch Press. https://doc.rust-lang.org/book/

[2] Chacon, S., & Straub, B. (2014). *Pro Git* (2nd ed.). Apress. https://git-scm.com/book/

[3] Merkle, R. C. (1988). A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87* (pp. 369–378). Springer.

[4] NIST. (2015). *Secure Hash Standard (SHS)* (FIPS PUB 180-4). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.FIPS.180-4

[5] The Rust Project Developers. (2026). *Rust API Guidelines*. https://rust-lang.github.io/api-guidelines/

[6] Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.

[7] Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson.

[8] Khafidi, M. R. (2026). *libvctrl: A Precision Toolkit for Building Custom Version Control Systems*. GitHub. https://github.com/mroczect/libvctrl