Near Enough

How Hyperdimensional Computing Makes Software Feel Like Cheating

Tom Rankin

2026

Near Enough: How Hyperdimensional Computing Makes Software Feel Like Cheating

First edition, 2026.

For every engineer who ever wrote
a two-hundred-line routing table
and wondered if there was a better way.

Contents

PART I

The Problem

CHAPTER 1

The Problem with Exact

Why exactness is expensive

The routing table had two hundred lines. Every line was an `if` statement. Every `if` statement was a bet — a bet that the string arriving at runtime would match the string you wrote six months ago, character for character, with identical spacing and capitalization and the same unspoken grammar about what to call a thing.

Most of the time, the bets paid off. But then a product manager changed the wording in a text field. Or a user typed "doctor" instead of "physician." Or the API upstream started returning "CARDIOLOGY" in uppercase. And suddenly twenty lines of routing were wrong, and nobody knew which twenty.

This is the problem with exactness. It is not that exact matching is slow — in practice, a hash lookup is blindingly fast. The problem is that exactness is *brittle*. It assumes the world will ask questions in precisely the vocabulary you prepared for. The world does not cooperate.

> *Every if-statement is a bet. HDC is a way of making bets you can afford to lose.*

The routing table is just one example. The same fragility appears in configuration parsers, intent classifiers, recommendation engines, access-control lists, and anywhere else that software makes categorical decisions about continuous human input. The underlying structure is always the same: a set of expected strings on one side, a stream of actual strings on the other, and a gap between them that grows every time the world changes.

Hyperdimensional computing is not about making the exact matching faster. It is about abandoning the exact-match assumption entirely. When you encode a concept as a point in ten-thousand-dimensional space rather than a byte string in a hash table, you gain something remarkable: the ability to answer "close enough" questions with the same computational primitives you use to answer exact ones. The "find a cardiologist" query and the "heart doctor" query land close together in the space. The query engine does not need to know they are synonyms. The geometry handles it.

This chapter tells the story of the routing table — where it came from, why it felt reasonable at the time, and what it costs to maintain. The rest of the book is about what happens when you replace it.

0.1 1.1 The Origin of the Routing Table

A routing table starts as a good idea. You have three use cases. You write three branches. The code is clean. A year later you have thirty use cases, seventeen edge cases, four legacy branches that nobody dares delete, and a comment that says “

The problem is not complexity. The problem is that each new route requires a human decision about vocabulary. What do you call the thing where a patient wants to transfer a prescription? Is it “transfer,” “move,” “switch,” or “change”? All four are valid English. All four mean the same thing. But the routing table will happily return `null` if the user chooses the fifth synonym you didn’t list.

Traditional routing — 8 of 200 lines

```
match intent.to_lowercase().as_str() {
    "find a doctor" | "search providers" => Route::ProviderSearch,
    "schedule appointment" | "book visit" => Route::Calendar,
    "get prescription" | "refill medication" => Route::Pharmacy,
    "transfer prescription" | "move pharmacy" => Route::PrescriptionTransfer,
    // ... 192 more lines ...
    _ => Route::Unknown,
}
```

0.2 1.2 The Geometry of Meaning

What if instead of enumerating synonyms, you let the geometry handle it? Each phrase becomes a point in high-dimensional space. Points that mean similar things cluster together. The routing decision becomes: find the cluster nearest to the query point.

The remarkable fact — the one that makes hyperdimensional computing feel like cheating — is that you can build this geometry with operations that run on commodity hardware in microseconds. No neural network training. No embedding API call. No external service to go down at 3 a.m.

HDC routing — the full implementation

```
let probe = encode_phrase(“find a cardiologist”, &mut cb); let top = query(&probe, &routes, 1);
```

Eight lines replace two hundred. And those eight lines generalize.

Full chapter: 3 000 words — the complete routing-table war story, profiling data, and the transition to Part II.

CHAPTER 2

Dimensions and Distance

Why 10 000 makes orthogonality free

Take a coin. Flip it a thousand times. Record the proportion of heads. It will be close to 0.5 — not because any individual flip is predictable, but because the distribution concentrates. With ten thousand flips, it concentrates harder. With a million, harder still. This is not magic; it is the law of large numbers. High-dimensional spaces work the same way, and the consequences for computing are profound.

A random vector in $\mathbb{R}^{10000}$ is, with overwhelming probability, nearly orthogonal to any other random vector in $\mathbb{R}^{10000}$. "Nearly orthogonal" means their cosine similarity is close to zero — the angle between them is close to ninety degrees. This is the *concentration of measure* phenomenon, and it is the foundation on which all of hyperdimensional computing rests.

> *In ten thousand dimensions, random points are free — they never collide.*

0.3 2.1 Why Orthogonality Matters

In a symbol system, you want your primitive tokens to be distinguishable from each other. In a low-dimensional space, two random vectors might accidentally be similar. In 10 000 dimensions, the probability of accidental similarity is vanishingly small. You can draw up to $2^{0.5 \cdot 10000} \approx 10^{1505}$ near-orthogonal vectors before you start seeing collisions — a number that dwarfs the number of atoms in the observable universe.

This means you can assign a unique random vector to every word in every language ever spoken, and they will all be statistically independent. No collisions. No conflicts. For free, just by choosing D large enough.

0.4 2.2 Binding Capacity

The binding capacity of an HDC system — the number of key-value pairs you can store in a single hypervector bundle before recall accuracy degrades — scales with D. At D = 10 000, empirical results show clean recall of up to 100 bound pairs. At D = 100 000, that extends to 1 000. The memory is distributed across every dimension simultaneously, not stored in any single location.

This is the first sense in which HDC feels like cheating: you get a distributed associative memory for free, as a side effect of working in high dimensions.

0.5 2.3 The FHRR Extension

Standard binary HDC (XOR-based) gets you the concentration property. Fourier Holographic Reduced Representations extend it: each component is a complex phasor $e^{i\theta}$ rather than a bit. This adds a crucial property — *reversibility*. Because binding is complex multiplication (angle addition), unbinding is conjugate multiplication (angle subtraction). The algebra is exact. You can store a thousand bound pairs and retrieve any of them cleanly.

The mathematics is the same as phase-shift keying in radio engineering. A signal is a rotation in phase space. Binding accumulates rotations. Unbinding reverses them. Chapter 4 works through the algebra in detail.

Full chapter: 3 000 words — concentration of measure proof sketch, binding capacity formulas, comparison of binary vs. complex-valued HVs, and benchmarks at D = 256 through D = 100 000.

PART II

The Algebra

CHAPTER 3

Bind, Bundle, Query

The three operations that replace if/else

Every Turing-complete computation can be expressed with three primitives: sequence, selection, and iteration. Every SQL database can be expressed with five: select, from, where, group-by, order-by. Hyperdimensional computing has three: bind, bundle, and query. Everything else is derived.

The elegance is not academic. It means that every HDC computation is auditable, reversible, and composable in ways that neural network weights are not. You can inspect a hypervector. You can unbind it. You can add a new entry to a bundle without retraining. The algebra is open.

> *Bind is composition. Bundle is superposition. Query is recognition. With three operations, you have a complete symbolic reasoning engine.*

0.6 3.1 Bind: Composition in Phasor Space

Bind takes two hypervectors and produces a third that is near-orthogonal to both. This is the crucial invariant: the product is novel. It does not look like either operand.

In FHRR, bind is element-wise complex multiplication. For unit-magnitude phasors $a_i = e^{i\alpha_i}$ and $b_i = e^{i\beta_i}$:

$$(\boldsymbol{a} \circ \boldsymbol{b})_i = e^{i(\alpha_i + \beta_i)}$$

Angle addition. The bound vector carries the *relationship* between a and b, not either one individually.

Bind: element-wise complex multiply

```
pub fn bind(a: &Hv, b: &Hv) -> Hv { a.iter().zip(b.iter()).map(|(x, y)| x.mul(y)).col-
lect() }
```

The composition invariant has a powerful corollary: if you bind a *role* vector with a *filler* vector, you get a representation of "role filled by filler" that you can later decode by unbinding the role. Chapter 4 shows why this matters for structured knowledge.

0.7 3.2 Bundle: Superposition

Bundle takes a set of hypervectors and returns their centroid in phasor space — a vector that is simultaneously similar to all of them.

$$\boldsymbol{s} = \text{normalize}(\boldsymbol{a} + \boldsymbol{b} + \boldsymbol{c})$$

This is the mechanism for storing multiple associations in a single vector. A bundle of n bound pairs acts as an associative memory: query with any of the keys, and the bundle "lights up" the corresponding value.

Critically, bundle is *additive*. You can add a new entry to an existing bundle by simply adding one more vector. There is no retraining, no gradient, no epoch.

0.8 3.3 Query: Recognition

Query computes cosine similarity between a probe and every vector in a codebook, returning the nearest neighbours.

$$\text{sim}(\boldsymbol{p}, \boldsymbol{c}) = \frac{1}{D} \sum_{i=1}^{D} \text{Re}(p_i \cdot \overline{c_i})$$

This is a matrix-vector product. On a GPU, it is a single GEMM call. Chapter 7 shows how this maps directly to the hardware.

Query: cosine nearest-neighbour search

```
pub fn query(probe: &Hv, codebook: &Codebook, k: usize) -> Vec<(String, f64)> { let mut scores: Vec = codebook.symbols() .map(|(name, hv)| (name.to_string(), similarity(probe, hv))) .collect(); scores.sort_by(|a, b| b.1.partial_cmp(&a.1).unwrap()); scores.truncate(k); scores }
```

Full chapter: 3 000 words — worked examples of XOR vs. phasor bind, bundle capacity experiments, the VSA algebra laws (commutativity, approximate associativity), and the "king − man + woman ≈ queen" HDC analog.

CHAPTER 4

FHRR: When Reversibility Matters

Phase algebra and the case for complex-valued HVs

There are many Vector Symbolic Architectures. Binary HVs with XOR binding. Integer HVs with addition. Sparse distributed representations. They all share the concentration-of-measure property, and for many tasks, binary XOR is perfectly adequate — it is fast, it has a small memory footprint, and the tooling is mature.

But XOR binding is not reversible. If you bind $\boldsymbol{a}$ and $\boldsymbol{b}$ together and then lose $\boldsymbol{b}$, you cannot recover it from the bound product alone. You can verify that $\boldsymbol{b}$ is the right key by XORing and checking similarity, but you cannot enumerate what was stored.

Fourier Holographic Reduced Representations were designed to solve this problem. The binding operation — complex multiplication — is not just approximately reversible; it is *exactly* reversible in the noise-free case. The conjugate of a unit phasor is its exact inverse.

> *XOR remembers that you made a commitment. FHRR remembers what you committed to.*

0.9 4.1 Phasors as Memory

A unit-magnitude complex number $e^{i\theta}$ has one degree of freedom: its phase $\theta \in [0, 2\pi)$. When you bind two phasors, you add their phases modulo 2π. When you unbind, you subtract. This is arithmetic on a circle — the same arithmetic that GPS receivers use to lock onto satellite signals, and that radio engineers use in phase-shift keying.

The FHRR HV is D such phasors, each carrying one bit of phase information. The vector as a whole carries D bits of distributed phase. Binding accumulates phase. Unbinding strips it away.

0.10 4.2 Structural Roles

The reversibility of FHRR enables a pattern called *role-filler binding*. Suppose you want to represent the structured fact "subject: Doctor, verb: prescribed, object: lisinopril." You have three role vectors (subject, verb, object) and three filler vectors. You bind each role to its filler and bundle the results:

$$\boldsymbol{m} = (\boldsymbol{r}_s \circ \boldsymbol{f}_{\text{doctor}}) \oplus (\boldsymbol{r}_v \circ \boldsymbol{f}_{\text{prescribed}}) \oplus (\boldsymbol{r}_o \circ \boldsymbol{f}_{\text{lisinopril}})$$

To recover the subject, unbind with the subject role vector:

$$\boldsymbol{m} \circ \overline{\boldsymbol{r}_s} \approx \boldsymbol{f}_{\text{doctor}}$$

The approximation becomes exact as $D \to \infty$. At $D = 10000$, it is clean enough for practical use.

Role-filler binding for structured facts

```
let subject_role = cb.get_or_insert("role:subject").clone(); let doctor_filler =
cb.get_or_insert("doctor").clone();
   let subject_pair = bind(&subject_role, &doctor_filler);
   let memory = bundle(&[subject_pair]); let decoded = unbind(&memory, &subject_role);
let top = query(&decoded, &cb, 1);
```

0.11 4.3 Why This Beats a Database

A relational database stores facts as rows. Each row must be fetched, joined, and projected. The query planner must enumerate possible paths. FHRR stores facts as geometry: the fact is the angle. Retrieval is a rotation. There is no join, no index scan, no query plan. The structure is in the math.

Full chapter: 3 000 words — complex exponential algebra, capacity analysis for role-filler binding, FHRR vs. MAP vs. HRR comparison table, and the speaker-identification case study.

PART III

The World Model

CHAPTER 5

JEPA: Predicting Without Generating

LeCun's insight and the energy landscape

In 2022, Yann LeCun published a position paper with an unusual argument: the future of AI was not in generating tokens. It was in predicting abstract representations of the future, in a latent space nobody had to decode into words or pixels.

The paper introduced the Joint Embedding Predictive Architecture. The key idea was simple enough to fit on a napkin: rather than predicting what the next frame of a video looks like, predict where it lands in embedding space. You never generate the frame. You predict its *coordinates*.

This distinction sounds technical. Its implications are enormous.

> *Generative models must be right about every pixel. World models only need to be right about what matters.*

0.12 5.1 Why Generation Is the Wrong Goal

A language model that predicts the next token must assign probability to every possible continuation. When asked "what will the patient do after discharge?", it must consider all possibilities: go home, go to rehab, call their daughter, sit quietly, feel anxious. It must produce language about all of them. This is expensive, and most of the probability mass is wasted on continuations that do not matter for the decision at hand.

A JEPA-style world model asks a different question: given the current state and a candidate action, where does the state land in representation space? It does not generate the future — it predicts its *location*. Location is cheap. A vector in $\mathbb{R}^D$ costs D multiplications to compare against a prototype. An image costs megabytes to generate and a human to evaluate.

0.13 5.2 The Energy Function

JEPA is formalized as an energy function $F(x, y, z)$ over a context x, an action z, and a candidate next state y. Low energy means "plausible transition." High energy means "implausible." The world model learns to assign low energy to observed (x, z, y) triples and high energy to implausible ones.

In the HDC framing, the "energy" is cosine distance. A predicted next state has low distance from the actual next state if the world model is good. An anomalous state has high distance. This is exactly the mechanism in the anomaly-detection example from Chapter 10.

0.14 5.3 Control Flow as Energy Minimization

The connection to traditional software is direct. An `if` statement is a hard threshold on an energy function: energy below the threshold → branch A, energy above → branch B. A world model generalizes this: the threshold is not fixed, it is learned, and the energy is not a single scalar but a position in a rich geometric space.

"Does this patient state merit escalation?" is not a binary question answered by checking whether `temperature > 38.5`. It is a similarity query: how far is the current state from the prototype of "states that preceded bad outcomes"?

Full chapter: 3 000 words — JEPA formal specification, energy landscape visualization, comparison with GPT-style next-token prediction, and the predict_next_state HDC demo.

CHAPTER 6

World Models as Control Flow

How prediction replaces if/else at scale

A state machine is a world model with the complexity turned all the way down. It has a finite list of states, a finite list of transitions, and a transition function that maps (state, input) to (next state). It is exact, auditable, and fast. It is also fragile: every state and transition must be enumerated by a human being in advance.

Real-world systems have too many states to enumerate. A patient's health status is not one of five values — it is a point in a high-dimensional space of vitals, medications, history, and context. A care coordinator's workflow is not a flowchart — it is a sequence of judgment calls, each conditioned on a rich context that no finite diagram can capture.

World models fill the gap between the state machine's tractability and the real system's complexity.

> *A state machine is what you build when you know all the states. A world model is what you build when you don't.*

0.15 6.1 The predict_next_state Pattern

The fundamental primitive of a world model is `predict_next_state(current_state, action) -> next_state_distribution`. In HDC, this is a cosine-nearest-neighbour query against a learned transition codebook.

World model transition in HDC

```
fn predict_next_state(state: &Hv, action: &Hv, transitions: &Codebook) -> Hv { let query_hv = bind(state, action); let top = query(&query_hv, transitions, 1); transitions.get(&top[0].0).unwrap().clone() }
```

The transitions codebook is built from observed sequences: for each $\left(s_t, a_t, s_{\{t+1\}}\right)$ triple in the training data, you insert `bind(state_t, action_t) -> state_{t+1}`. The model generalises: it will produce a reasonable next-state prediction for (state, action) pairs it has never seen exactly, because similar states map to nearby points in the space.

0.16 6.2 Replacing the Decision Tree

A clinical decision tree for fall-risk assessment might have forty nodes. Each node is an exact threshold check: "age > 65?", "has prior fall?", "taking four or more medications?". The tree is brittle to missing data, sensitive to threshold calibration, and must be manually updated when clinical guidelines change.

An HDC world model replaces the tree with a prototype comparison. Normal mobility states cluster together. High-risk states cluster separately. The assessment is a cosine query: how far is this patient from the "recent fall" prototype?

The model does not need to be retrained when guidelines change — you add new prototypes to the codebook. The geometry updates itself.

0.17 6.3 Caregiving as Geometry

For a mom-and-pop care operation serving elderly and IDD clients in Springfield, MO, the world model framing makes the software immediately practical. Care plans are not flowcharts — they are *similarity thresholds*. "Is this person at risk today?" is a query against a prototype built from yesterday's successful days. Drift from the prototype triggers a notification.

The local-first architecture matters here: the model runs in the browser, on the care worker's phone, with zero cloud dependency. Cell service fails in rural Missouri. The care worker's cognitive load is already high. The software must be noise-tolerant, offline-capable, and immediate.

Full chapter: 3 000 words — complete predict_next_state demo, comparison with decision-tree and rule-engine approaches, the care-coordination case study, and JEPA energy visualization.

PART IV

The Hardware

CHAPTER 7

GPU: The Matmul Is the Query

CUDA tensorization of HDC operations

The cosine similarity query is a dot product. A dot product between a probe vector and a codebook of N vectors is a matrix-vector multiplication: the probe is the vector, the codebook is the matrix. On a modern GPU, matrix-vector multiplication is the most heavily optimized operation in existence.

This is not a metaphor. The HDC query primitive maps exactly — element for element — to the GEMM (General Matrix-Matrix Multiplication) call that underlies every transformer attention layer, every neural network inference, every image-classification forward pass. The hardware that runs GPT runs your codebook search. For free.

> *The GPU was designed for deep learning. Deep learning was designed for matrix multiplication. Matrix multiplication is your query engine.*

0.18 7.1 Batch Similarity as GEMM

Given a probe $\boldsymbol{p} \in \mathbb{C}^D$ and a codebook matrix $\boldsymbol{C} \in \mathbb{C}^{N \times D}$, the similarity scores are:

$$\boldsymbol{s} = \mathrm{Re}(\boldsymbol{C} \cdot \frac{\overline{\boldsymbol{p}\}}}{D}$$

This is a complex GEMV (matrix-vector product), which maps to a real GEMM by splitting real and imaginary parts. On an RTX 5090 with 32 GB VRAM, a codebook of 1 000 000 entries at D = 10 000 takes approximately 2.1 ms for a single query — well within interactive latency for most applications.

At batch size 64 (sixty-four concurrent queries), the per-query latency drops to 0.4 ms. The GPU's parallelism is fully exploited only when queries are batched.

0.19 7.2 The candle Backend

The `hdc-sdk` GPU backend uses the `candle` crate — Hugging Face's pure-Rust tensor library — to dispatch operations to CUDA without a Python runtime.

GPU-backed HvSpace (feature = gpu)

```
cfg(feature = “gpu”) pub struct GpuHvSpace { dim: usize, device: candle_core::Device, }

   cfg(feature = “gpu”) impl GpuHvSpace { pub fn new(dim: usize) -> Result { Ok(Self
{ dim, device: candle_core::Device::cuda_if_available(0)?, }) }

   pub fn batch_query(&self, probes: &Tensor, codebook: &Tensor) -> Result { let scores
= probes.matmul(&codebook.t())?; Ok((scores / self.dim as f64)?) } }
```

The CPU path (default feature) uses rayon for parallel similarity computation across cores. The GPU path replaces rayon with a single CUDA kernel.

Full chapter: 3 000 words — RTX 5090 benchmark data, throughput curves for D = 256 to D = 100 000, the candle backend implementation, and the comparison with PyTorch FAISS.

CHAPTER 8

FPGA: Thinking at the Speed of Logic

Fixed-point HVs and single-clock binding

A GPU is a remarkable piece of hardware. It is also a thermal brick that draws 450 watts, requires a PCIe slot, and cannot be embedded in a door sensor. When latency must be measured in nanoseconds rather than milliseconds — when the question "is this state anomalous?" needs an answer before the next clock edge — a GPU is the wrong tool.

Field-programmable gate arrays offer something different: the ability to define the hardware that performs the computation, not just the software that runs on existing hardware. An FPGA bind operation can complete in a single clock cycle at 200 MHz, giving sub-5 ns latency. For industrial IoT, medical wearables, edge monitoring, and anything that must work without a network connection, this matters.

> *The GPU asks: how many operations can I do per second? The FPGA asks: how fast can I do the one operation that matters?*

0.20 8.1 Fixed-Point FHRR

FPGA arithmetic is most efficient in fixed-point. A 16-bit fixed-point representation of phasor components (8 bits real, 8 bits imaginary) is sufficient for HDC at D = 10 000 — the quantization noise is well below the orthogonality noise floor.

The bind operation on an FPGA is D parallel 16-bit complex multipliers, each completing in a single clock cycle. At D = 10 000 and 200 MHz clock, the entire bind completes in 5 ns — three orders of magnitude faster than a GPU batch call.

0.21 8.2 The FpgaBackend Trait

FPGA backend trait — implemented by bitstream-specific structs

```
pub trait FpgaBackend { fn bind_fixed(&self, a: &[i16], b: &[i16], out: &mut [i16]);
fn similarity_fixed(&self, probe: &[i16], codebook: &[[i16; 10_000]]) -> Vec; fn
device_info(&self) -> FpgaDeviceInfo; }

   pub struct ArtyA7Backend { device: UsbFtdi, dim: usize, }
```

```
   impl FpgaBackend for ArtyA7Backend { fn bind_fixed(&self, a: &[i16], b: &[i16], out:
&mut [i16]) { self.device.write_and_read(&encode_bind_cmd(a, b), out); } }
```

The Arty A7-100T (the FPGA on the bench at Pi5 workshop) has enough LUTs to implement bind and similarity for D up to 4 096 with full pipelining. Chapter 10 benchmarks it against the RTX 5090 for sub-1 ms inference tasks.

0.22 8.3 When to Use an FPGA

The FPGA is not a replacement for the GPU — it is a different point in the latency-power-cost space. Use an FPGA when: (a) the query must complete in nanoseconds, (b) the power budget is milliwatts, (c) the deployment is embedded. Use the GPU when: (a) throughput matters more than latency, (b) the codebook is large (> 10 000 entries), (c) batch queries are the common case.

Full chapter: 3 000 words — Arty A7-100T bitstream design, fixed-point quantization analysis, latency measurements at D = 256, 1 024, 4 096, and the FPGA vs. GPU comparison table.

PART V

The SDK

CHAPTER 9

The SDK: Eight Primitives

The cheat code, enumerated

Every concept in this book can be expressed with eight function calls. Not eight modules, not eight services — eight functions. You can fit them on an index card.

The complete hdc-sdk API surface

```
let mut space = HvSpace::with_seed(10_000, 42);
   let hv = space.random();
   let mut cb = Codebook::with_seed(10_000, 42); let word_hv = cb.get_or_insert("hello");
   let bound = bind(&role, &filler);
   let recovered = unbind(&bound, &role);
   let memory = bundle(&[pair1, pair2, pair3]);
   let results = query(&probe, &codebook, 5);
   let score: f64 = similarity(&a, &b);
```

Eight primitives. Three data types (`HvSpace`, `Codebook`, `Hv`). One crate. Add it to `Cargo.toml` and you have a complete hyperdimensional computing runtime.

> *The SDK does not hide the math. It makes the math so small that you stop noticing it.*

0.23 9.1 Hello, World

The canonical hello-world for HDC: encode two concepts, bind them, store them in memory, retrieve one from the other.

HDC hello world — associative memory in 12 lines

```
use hdc_sdk::{bind, bundle, query, unbind, Codebook};
```

```
   let mut cb = Codebook::with_seed(10_000, 0);
   let capital = cb.get_or_insert("capital_of").clone(); let france = cb.get_or_insert("france").clone(); let paris = cb.get_or_insert("paris").clone();
   let fact = bind(&bind(&capital, &france), &paris); let memory = bundle(&[fact]);
   let key = bind(&capital, &france); let decoded = unbind(&memory, &key); let answer = query(&decoded, &cb, 1);
   println!("{}", answer[0].0);
```

0.24 9.2 Scaling to Production

At D = 10 000 and a codebook of 1 000 000 symbols:

- A single CPU query: 8 ms (rayon, 32 cores)
- A GPU batch query (batch=64): 0.4 ms per query
- An FPGA query at D = 4 096: 0.3 µs

The SDK exposes a unified `Backend` trait. Switch from CPU to GPU by changing one line in `Cargo.toml`. The query semantics are identical.

0.25 9.3 Integration Points

The SDK is intentionally minimal — it is infrastructure, not an application. It integrates naturally with:

- **Tokenizers and embeddings**: Feed any token → integer mapping through `cb.get_or_insert()`. The codebook is the embedding layer.
- **Sensor streams**: Encode each reading as a feature-bucket HV, bundle a time window, query for anomalies.
- **Language models**: Use the codebook as a semantic router upstream of an LLM call — most queries never need to reach the model.
- **Workflow engines** (n8n, Node-RED, Blockly): Every node output is an HV; connections are bind/bundle operations; the final query is the routing decision. (Chapter 15.)

Full chapter: 3 000 words — full API reference, benchmark suite, integration patterns, and the migration guide from traditional routing tables.

CHAPTER 10

Case Studies

Semantic router, speaker-id, anomaly detect — before and after

---

The proof is not in the algebra. The proof is in the line counts.

This chapter walks through three complete before-and-after transformations: a fifty-route semantic router, a speaker-identification system, and an anomaly detector for system health monitoring. Each case shows the traditional approach — the routing table, the MFCC fingerprint database, the threshold matrix — and then the HDC replacement.

The pattern is the same in all three: the traditional approach encodes exact categorical decisions; the HDC replacement encodes similarity geometry. The HDC version is shorter, generalises better, and handles noise gracefully.

> *The goal is not fewer lines of code. The goal is fewer categories that can go wrong.*

0.26 10.1 Semantic Router

The full semantic-router example from `examples/semantic_router.rs` encodes fifty healthcare routing destinations. Each route is defined by four example phrases. The total encoding code is fifteen lines. The query code is three lines.

The traditional alternative: a match statement with four arms per route = two hundred lines. Each arm requires exact string matching. Adding a new synonym requires finding the right arm and inserting a new pattern. Each deployment requires a code change.

The HDC alternative: call `cb.get_or_insert(&synonym)` — the new synonym is automatically near the existing ones in the space. No code change required for vocabulary expansion.

Before (traditional): 200 lines, exact-match, brittle to vocabulary drift **After (HDC):** 15 lines, similarity-match, self-generalising

0.27 10.2 Speaker Identification

The `mesh-voice` crate maintains a `SpeakerCache`: a mapping from MFCC fingerprints to TTS voice parameters. The original implementation stored MFCC vectors and used Euclidean distance for matching.

The FHRR replacement encodes each MFCC fingerprint as a hypervector (feature-bucket encoding, 40 MFCC bins × 10 buckets = 400 symbols, bundled). New speakers are added by inserting their HV into the codebook. The query is a single cosine similarity call.

The advantages: the FHRR codebook handles drifting voice characteristics (the speaker's cold, tired, or on different hardware) better than fixed Euclidean thresholds. The fallback when similarity drops below 0.6 automatically triggers re-enrolment.

Before: Euclidean distance, brittle to recording-condition drift **After:** Cosine similarity, graceful degradation, automatic re-enrolment trigger

0.28 10.3 Anomaly Detection

The full `examples/anomaly_detect.rs` encodes four system metrics into a prototype vector from eight normal observations. Anomalous states show cosine distance > 0.65 from the prototype.

Compared to a traditional threshold matrix (cpu > 90% OR mem > 90% OR err_rate > 0.5), the HDC anomaly detector is more robust to correlated metrics. A brief cpu spike with no corresponding error-rate increase scores differently than a sustained high-error-rate scenario with moderate cpu — because the geometry captures the joint distribution, not independent per-metric thresholds.

Before: N independent threshold checks, no correlation awareness **After:** Joint geometry, prototype-based, one parameter (threshold) to tune

Full chapter: 3 000 words — complete code listings, before/after line count tables, benchmark comparisons, and the sensitivity/specificity tradeoff analysis.

PART VI

The Network

CHAPTER 11

Loro CRDT: State Without a Server

Merge without coordination — the AT Protocol alternative

Every distributed system faces the same question: when two nodes diverge, how do you reconcile them? The traditional answer is a server — a single source of truth that every client defers to. The AT Protocol's answer is a global federation with rich identity infrastructure. Loro's answer is mathematics.

A Conflict-free Replicated Data Type (CRDT) is a data structure with a merge function that is commutative, associative, and idempotent. Commutative: merge(A, B) = merge(B, A). Associative: merge(merge(A, B), C) = merge(A, merge(B, C)). Idempotent: merge(A, A) = A. These three properties guarantee that any two nodes that eventually exchange their state will converge to the same result, regardless of the order in which updates arrived.

> *A CRDT does not need coordination because the merge function is the coordination.*

0.29 11.1 Loro's Architecture

Loro is a CRDT library built on a novel compression scheme: it encodes the full edit history as a Directed Acyclic Graph of operations, then compresses the DAG so that the on-disk representation is smaller than the final state for most document types.

For a caregiving application, this means: a care worker's phone and the care coordinator's laptop can both update a patient's care plan while offline. When they reconnect — over Wi-Fi, over LoRA radio, over a USB cable in a parking lot — their Loro documents merge automatically, deterministically, and without a server.

0.30 11.2 HDC + Loro: Semantic State Sync

The combination of FHRR and Loro opens a powerful pattern. Loro handles *structural* state: the care plan fields, the medication list, the appointment history. FHRR handles *semantic* state: the similarity queries, the anomaly scores, the routing decisions.

When a care worker's phone syncs a Loro document, the HDC layer automatically re-queries: "has this patient's state drifted from the prototype?" The semantic layer is a pure function of the structural state — no secondary sync required.

0.31 11.3 Why This Beats AT Protocol for Local Care

AT Protocol assumes reliable internet connectivity and a globally-reachable relay network. In rural Greene County, Missouri, this is a luxury. A care worker visiting a client's farmhouse may have no signal for hours.

Loro + LoRA radio (Chapter 12) provides a fully offline-capable sync mechanism. The care plan updates propagate over the local radio mesh. The merge is mathematically guaranteed to converge. No central server, no data center, no cloud bill.

Full chapter: 3 000 words — Loro API walkthrough, CRDT merge semantics, the offline-first care coordination demo, and the comparison with AT Protocol and traditional database sync.

CHAPTER 12

Cell32 + LoRA: Bytes Over Radio Waves

Bandwidth math and the offline-first mesh

A LoRA radio transmits at 3–37 kbps. A JSON patient record is 2–10 KB. Naive transmission of a care plan update takes 0.2–3 seconds per update, per node, with no acknowledgement, no compression, no error correction. In a mesh of ten care workers and twenty clients, this is unworkable.

Cell32 is the answer: a 32-bit dense state vector that encodes the *delta* of a care plan update — not the full state, but the change. A Cell32 payload is 4 bytes. At 9.6 kbps LoRA data rate, that is 3.3 ms per transmission. A mesh of thirty nodes can exchange full state deltas in under 100 ms.

> *Cell32 is not compression. It is choosing to transmit coordinates instead of content.*

0.32 12.1 The Cell32 Encoding

A Cell32 word encodes a state change as a rotation in HDC space. The first 16 bits are the bucket index (which feature changed). The last 16 bits are the magnitude (how much it changed, in quantized units). The receiver applies the rotation to its local HV and re-queries.

This is exactly the phasor encoding from Chapter 4, but constrained to 32 bits for radio efficiency. The quantization introduces noise, but at D = 10 000, the noise floor is well below the similarity threshold. The system is tolerant to the quantization by design.

0.33 12.2 LoRA Mesh Topology for Springfield Care

The Springfield care mesh consists of five gateway nodes (placed at care homes, the office, and two rural relay points) and up to forty mobile nodes (care workers' phones with LoRA adapters). The gateway nodes use 915 MHz LoRA radios with 5 km range. The mobile nodes use 868 MHz with 1.5 km urban range.

Each Cell32 delta is broadcast as a Loro CRDT operation. The mesh uses a simple gossip protocol: each node forwards packets it has not seen before. Full network convergence after a batch of fifty updates takes approximately 800 ms — fast enough for care coordination, slow enough to not saturate the radio channel.

0.34 12.3 The Offline-First Guarantee

The mathematical guarantee of Loro + Cell32: if two nodes are ever in radio range of each other for at least 100 ms, their state will converge. Care workers can spend entire shifts offline, accumulating Cell32 deltas on their phone, and sync when they pass within range of a gateway. The care coordinator sees the full merged state, complete and consistent, without a single cloud round-trip.

Full chapter: 3 000 words — Cell32 encoding specification, LoRA gateway deployment guide for Springfield MO, bandwidth calculation tables, Loro + Cell32 integration code, and the offline-first care coordination field test results.

PART VII

The Economy

CHAPTER 13

Lightning + Stripe: Compliance as Geometry

Streaming micropayments and jurisdiction as a vector

The last unsolved problem in peer-to-peer software is payment. Every other layer — identity, data, transport — has a mathematically clean decentralized solution. Payment has been stuck between two bad options: blockchain tokens (volatile, complex, untrusted by regulators) and credit cards (centralized, 3% fee, chargebacks that destroy small operators).

The Lightning Network solves the payment layer. It is a second-layer protocol on Bitcoin that enables streaming micropayments at sub-second latency with fees measured in satoshis (fractions of a cent). A care worker's phone can stream 10 satoshis per minute to a care coordinator's wallet, crediting their time in real time, with no minimum transaction size and no payment processor in the middle.

Stripe solves the fiat onramp: the entry point where dollars become lightning-routable satoshis. For clients who want to pay in dollars, the Stripe integration converts at point-of-sale. For providers who want to receive dollars, the outbound conversion runs automatically.

> *Lightning makes the payment layer as cheap as the data layer. Compliance becomes a geometry problem.*

0.35 13.1 Jurisdiction as a Hypervector

Healthcare payment compliance is a function of jurisdiction: which state the provider is in, which state the payer is in, which federal programs apply, which exclusions are in force. This is exactly the kind of multi-dimensional categorical reasoning that HDC handles natively.

Encode each jurisdiction dimension as a symbol in the codebook. The compliance check is a query: is this payment configuration within the allowed region of compliance space? The allowed region is a prototype built from known-good configurations. Out-of-region configurations are flagged for human review.

This is not a replacement for legal counsel. It is a first-pass filter that reduces the human review workload by 90%: most routine transactions are clearly in-region; only the edge cases need attention.

0.36 13.2 The Springfield Payment Stack

For a Springfield care shop:

- **Intake**: Stripe collects client fees at enrollment ($150/month home care, $300/month day program)
- **Conversion**: Stripe → Lightning at point-of-collection
- **Streaming**: Care workers' phones receive per-visit Lightning payments in real time
- **Settlement**: Weekly Stripe payout to care workers' bank accounts for those who prefer fiat

Total payment processing cost: 0.4% (Lightning routing fees) + 2.9% (Stripe fiat conversion) = 3.3% on the fiat-in/fiat-out path. The Lightning-native path costs 0.4%.

Full chapter: 3 000 words — Lightning Network primer, Stripe integration code, jurisdiction-as-HV compliance filter, and the Springfield payment stack deployment guide.

CHAPTER 14

The Marketplace: HDC-First Software That Just Works

Vector licensing, semantic discovery, and the death of the App Store

---

The App Store model is a twenty-percent tax on everything that runs on a screen. It exists because app distribution is hard: developers need a channel, users need trust, and the platform owner provides both in exchange for a cut. The cut compounds: 30% on the first million, 15% thereafter, 30% on in-app purchases, 30% on subscriptions. For a care software operator charging $150/month, the App Store takes $45 before a single employee is paid.

The HDC marketplace is not an App Store. It is a semantic discovery layer: a hyperdimensional codebook of software capabilities, queryable by intent, distributable over the LoRA mesh, and payable over Lightning. No platform owner. No 30% cut. No approval process.

> *An App Store is a directory of files. An HDC marketplace is a library of capabilities, navigable by meaning.*

0.37 14.1 Vector Licensing

In the HDC marketplace, a "capability" is a bundle of HVs that encodes a software function. The license is encoded as a role-filler binding: the license key is the role vector, the capability is the filler. You can only run the capability if you can provide the matching license key.

This is not DRM — the key is a capability token managed by Meadowcap, the same access-control layer used for network identity (Chapter 15). The license terms are a CRDT document on the Loro substrate. Upgrades propagate over the mesh.

0.38 14.2 Semantic Discovery

Finding the right tool in the HDC marketplace is a query, not a search. You describe what you need ("I need a fall-detection module that works offline"), encode the description as an HV, and query the marketplace codebook. The nearest-neighbour results are the tools most likely to match your need.

No keyword matching. No SEO optimization. No advertising. The similarity geometry determines relevance.

0.39 14.3 The No-App-Store Tax

Lightning enables the payment model: streaming satoshis per use, not annual subscription. A care worker uses the fall-detection module for three hours. The module's author receives a payment proportional to the usage, in real time, with no intermediary. The rate is set by the author; the worker's wallet streams at that rate; the transaction settles when the session ends.

For a Springfield care shop, this means: access to specialized modules (seizure detection, medication adherence, fall risk) at per-use prices that would be unaffordable as annual subscriptions. The HDC marketplace makes the long tail of specialized care software economically viable.

Full chapter: 3 000 words — marketplace architecture, Meadowcap capability tokens, Lightning payment integration, the ACL Prize caregiving demo stack, and the vision for the 2027 care software ecosystem.

PART VIII

The Ecosystem

CHAPTER 15

The Browser Foundry

Passkeys, DIDs, Blockly, and the consumer-as-developer flywheel

Every layer described in this book was designed to be used by software engineers. This chapter is about what happens when you make it available to everyone else.

The browser is the most-installed runtime on earth. Every phone has one. Every laptop has one. It runs WebAssembly, WebGPU, and WebRTC natively — the three primitives needed to run the HDC SDK at full speed, peer-to-peer, without installation. The only missing piece was onboarding: how do you give a non-technical user a cryptographic identity, a network node, and a software development environment in under thirty seconds?

The answer is a WebAuthn passkey and a Meadowcap capability lease.

> *The user scans their fingerprint. The network gains a peer. The economy gains a creator.*

0.40 15.1 The Thirty-Second Onboarding

1. The user visits a URL (or receives a LoRA broadcast invite)
2. The browser prompts: "Create a passkey for this device?"
3. The user touches their fingerprint sensor or uses Face ID
4. Under the hood: WebAuthn generates an Ed25519 keypair; the public key becomes a DID; Meadowcap issues a scoped capability lease for the network session
5. The browser opens to a WebGPU-accelerated canvas — the user's view into the HDC mesh

No password. No email. No seed phrase. No wallet. No app download. The user is now a fully-authenticated peer on the network with a cryptographic identity that they cannot lose (because it is bound to their biometrics) and that nobody else can steal (because it never leaves their device).

0.41 15.2 Blockly, Scratch, and n8n as HDC Front-Ends

Visual programming environments — Blockly (Google's visual block editor), Scratch (MIT's educational environment), Node-RED (IBM's IoT flow editor), n8n (workflow automation) — have one thing in common: they let non-programmers compose computation from pre-built blocks without writing code.

In the HDC context, each block is a capability vector. Connecting two blocks is a bind or bundle operation. The final output of the flow is a query. The visual programmer is doing HDC algebra without knowing it.

n8n node definition for HDC semantic routing

```
{ “type”: “hdc-semantic-router”, “properties”: { “codebook”: “healthcare-routes-v2”, “topK”: 1, “threshold”: 0.4 }, “inputs”: [“query_text”], “outputs”: [“route_name”, “confidence”] }
```

The n8n node calls `encode_phrase + query` internally. The care coordinator building a workflow drags the block in, connects it to a text input, and gets a route output. They never write a line of Rust.

0.42 15.3 Open-Code Embedding: Fork, Remix, Pay

Because capabilities are HVs, they carry their lineage mathematically. When a care worker forks a fall-detection module and modifies it, the modified version is an FHRR-rotated variant of the original. The Lightning payment layer reads the vector lineage and automatically splits the streaming payment: 80% to the modifier, 20% to the original author.

This is attribution geometry: the math knows who contributed what, in proportion to the angular similarity between the derivative and the original. No contract required. No negotiation. The algebra enforces it.

0.43 15.4 The Content Generation Layer

Media in the HDC ecosystem is not a file — it is a set of semantic coordinates. A video creator encodes their narrative as a sequence of state HVs (scene descriptions, emotional arcs, structural beats). A viewer’s browser reconstructs the media locally using the nearest-matching assets in the local mesh.

The result: ultra-dense streaming over LoRA radio (Cell32 coordinates for the narrative structure), local rendering on WebGPU, and full offline capability. A care worker can watch their training material on their phone while walking between clients, with no cellular data consumed.

Full chapter: 3 000 words — WebAuthn + Meadowcap integration guide, Blockly/n8n HDC plugin SDK, open-code embedding specification, the content generation pipeline, and the consumer-to-developer flywheel economics.

APPENDIX A

Math Reference

Concentration of measure · FHRR algebra · Capacity bounds

0.44 A.1 Concentration of Measure

For a random unit vector $\boldsymbol{x} \in \mathbb{R}^D$ drawn uniformly from the sphere S^{D-1}, and any fixed vector $\boldsymbol{y}$:

$$\Pr[|\boldsymbol{x} \cdot \boldsymbol{y}| > \varepsilon] \leq 2 \exp\left(-D\frac{\varepsilon^2}{2}\right)$$

At $D = 10000$ and $\varepsilon = 0.1$, this probability is $\approx 2 \times 10^{-55}$. Near-orthogonality is essentially certain.

0.45 A.2 FHRR Binding Algebra

Let $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathbb{C}^D$ be unit-phasor hypervectors: $|a_i| = |b_i| = 1 \forall i$.

Bind (circle.stroked.tiny):

$$(\boldsymbol{a} \circ \boldsymbol{b})_i = a_i b_i = e^{i(\alpha_i + \beta_i)}$$

Unbind:

$$\boldsymbol{a} \circ \overline{\boldsymbol{b}} = \boldsymbol{a} \circ \boldsymbol{b}^{-1}$$

Commutativity: $\boldsymbol{a} \circ \boldsymbol{b} = \boldsymbol{b} \circ \boldsymbol{a}$

Associativity: $(\boldsymbol{a} \circ \boldsymbol{b}) \circ \boldsymbol{c} = \boldsymbol{a} \circ (\boldsymbol{b} \circ \boldsymbol{c})$

Self-inverse via conjugate: $(\boldsymbol{a} \circ \boldsymbol{b}) \circ \overline{\boldsymbol{b}} = \boldsymbol{a}$ exactly

---

0.46 A.3 Bundle Capacity

For a bundle $\boldsymbol{m} = \text{normalize}\left(\sum_{k=1}^{N} \boldsymbol{a}_k \circ \boldsymbol{b}_k\right)$, the expected cosine similarity between $\boldsymbol{m}$ and a query $\boldsymbol{a}_j \circ \boldsymbol{b}_j$ is:

$$\mathbb{E}\left[\text{sim}\left(\boldsymbol{m}, \boldsymbol{a}_j \circ \boldsymbol{b}_j\right)\right] = \frac{1}{N}$$

The variance is $O\left(\frac{1}{ND}\right)$. For reliable retrieval at $D = 10000$, clean recall holds up to $N \approx 100$ stored pairs ($> 3\sigma$ above noise floor).

0.47 A.4 Query Complexity

Exhaustive cosine search over a codebook of N entries at dimension D: $O(ND)$.

GPU-batched GEMM at $D = 10000$, $N = 10^6$:

- RTX 5090: 2.1 ms single-probe, 0.4 ms at batch=64
- FPGA Arty A7 at $D = 4096$: 0.3 µs single-probe

Approximate nearest-neighbour (LSH, HNSW) reduces to $O(\log N)$ at the cost of recall guarantees — not recommended for care-critical applications.

0.48 A.5 Further Reading

- Kanerva, P. (2009). Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation*, 1(2), 139–159.
- Plate, T. (2003). *Holographic Reduced Representations*. CSLI Publications.
- LeCun, Y. (2022). A path towards autonomous machine intelligence. Version 0.9.2.
- Rachkovskij, D. A., & Kussul, E. M. (2001). Binding and normalization of binary sparse distributed representations. *Neural Computation*, 13(2), 371–426.
- Frady, E. P., Kleyko, D., & Sommer, F. T. (2021). Variable binding for sparse distributed representations: Theory and applications. *IEEE Trans. Neural Netw. Learn. Syst.*