# Episode Book

EPISODE #6521

# Multitenancy in Qdrant Vector Databases

| | |
|---|---|
| PUBLISHED | 10 November 2026 at 11:46 UTC |
| DURATION | 28:39 |
| LISTEN | myweirdprompts.com/episode/multitenancy-qdrant-vector-storage-58880/ |
| HOSTS | Corn, Herman, Hilbert |
| SCRIPT MODEL | z-ai/glm-5.3 |
| VOICES | chatterbox-regular (chatterbox-local) |
| PIPELINE | V5.26 |
| SOURCES | 8 cited by the research (listed at the end) |
| BOOK GENERATED | 11 November 2026 at 06:49 UTC |

## SYNOPSIS

Multitenancy in vector databases involves four distinct isolation trade-offs: storage, compute, enforcement, and operations. When partitioning in Qdrant, the dominant cost is rarely the vector data itself, but the HNSW index overhead required for each collection or shard. This deep dive maps out Qdrant's four isolation models—from separate collections and payload partitioning with the tenant index to custom sharding and tiered promotion—contrasting real-world memory benchmarks against latency trade-offs across different corpus sizes.

# TRANSCRIPT



### Corn

Somewhere in every architecture meeting there's a moment where someone says, we'll just put a tenant underscore ID field on it, and the room nods, and four separate decisions get made without anyone noticing. Daniel's prompt for today is a deep dive on Qdrant and exactly that moment. He wants the four levels it gives you for splitting tenants — collections, payload partitioning, custom sharding, the new tiered thing — and he wants them mapped onto three real situations: a multi-user app with private memories per user, a SaaS with a few hundred business customers, and one with millions of small users. Plus what actually goes wrong when you pick wrong.



### Herman

And what goes wrong is the interesting part, because almost none of it shows up on a dashboard.



### Corn

That's the hook, really. The reason this is harder than it sounds is that "isolation" isn't one thing — it's at least four.



### Herman

Right, so let me put the frame up, because it runs through the whole episode. When people say isolation they usually mean one thing, and Qdrant is actually giving you four dials. Storage isolation — do tenants share physical structures, which is mostly a cost question. Compute isolation — can one noisy tenant degrade everyone else's query latency. Enforcement isolation — can a client actually get at another tenant's data. And operational isolation — can you restore, erase, or bill one tenant without touching the rest. A team that says "we'll pool it" or "we'll silo it" has picked one word and inherited four decisions they never made.



### Corn

And the four levels of Qdrant are really four different positions on those dials.



### Herman

Level one, a separate collection per tenant. Level two, one shared collection with a tenant-ID payload field and the is_tenant index. Level three, custom sharding — a shard key per tenant, so each tenant's data lives on its own physical shard. And level four, tiered multitenancy, which shipped in 1.16 about a year ago — a shared fallback shard for the small tenants, plus a mechanism to promote a big tenant into its own dedicated shard without the application knowing.



### Corn

Which sounds like a menu until you find out what each item costs.



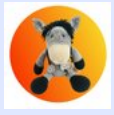
### Herman

And that's the second idea to plant before we walk the levels: the dominant cost is not the vectors. It's the ANN index. In Qdrant that's an HNSW graph, and every partition you create — every collection, every shard — carries its own graph, its own segments, its own ID and version trackers, its own optimizer working set. Split the corpus and the graph is rebuilt in full, per partition. There's a nice line from a field study I read: in a relational system the marginal cost of a partition is near zero — catalog entries, a connection pool. In a vector store, the data is the small part.



### Corn

HNSW being the thing where every vector keeps a little list of which nearby vectors it's friends with.



### Herman

That's honestly a fair description. Each node keeps up to m outbound links, roughly two m at the base layer, each link a four-byte identifier. So the graph alone costs on the order of one point five times m times n times four bytes, over and above the vectors. And you pay that per partition.



### Corn

So let's walk the levels, because the mechanics matter for everything after.



### Herman

Level one, then. Separate collection per tenant. Each one gets its own HNSW graph, its own segments, its own everything. Qdrant's own docs are blunt about it — creating a separate collection per tenant is "rarely the most efficient approach," each collection carries resource overhead, and they say to use it only for a limited number of tenants that need strict isolation. And on Qdrant Cloud there's a hard default cap of one thousand collections per cluster. So that's a ceiling you hit fast if your tenant count is your user count.



### Corn

What's a tenant actually get out of it, though? Because if the answer is "a lot," the ceiling is an argument, not a verdict.



### Herman

It depends on the dial. What they get is maximum everything — separate config, separate index, separate replication, collection-level permissions. What they don't automatically get is enforcement, which we'll come to, because that's the one everyone assumes comes along.



### Corn

Level two.



### Herman

Payload partitioning. Everything goes in one collection. Every point carries a group ID in its payload — a keyword field saying which tenant it belongs to. Then you create a keyword payload index on that field with is_tenant set to true — that's been available since version 1.11 — and the flag does something physical: it tells Qdrant to co-locate vectors of the same tenant on disk, so a tenant's data can be read as one sequential pass instead of many scattered random seeks. That's the whole trick. The catch is procedural: every query, every update, every delete must carry the tenant filter. Miss one and you're querying everyone.



### Corn

Miss one and you're querying everyone is doing a lot of load-bearing work in that sentence.



### Herman

It is, and it's the security section in disguise. Hold that thought. Level three — custom sharding. You create the collection with sharding method custom, then create named shards and send a shard key with your reads and writes. The shard key routes the operation to that shard. So a tenant can own a shard, and the index on that shard is physically theirs — real compute isolation, real noisy-neighbor protection. Qdrant positions this for a smaller number of larger tenants, which makes sense, because each shard carries the same fixed overhead story we just talked about.



### Corn

One distinction I want you to make sharp, because I think it's where people get tangled — shard key versus filter.



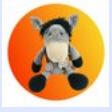
### Herman

Good, because they're orthogonal. The shard key selects where an operation runs. The payload filter selects which points match, within wherever the operation ran. A single shard can contain several tenants separated by payload filters — that's a documented combination. So you can have a shard for a big customer, and inside the fallback-style shards, filters doing the fine-grained work.



### Corn

So the shard key is the postcode, the filter is the apartment number.



### Herman

Sure, if you like — the postcode is routing, the apartment number is matching. Level four is where it gets clever. Tiered multitenancy, from 1.16. Three components. User-defined sharding, so named shards, same as level three. Fallback shards — routing that sends a request to the tenant's dedicated shard if one exists, otherwise to a shared fallback shard. And tenant promotion — moving a tenant from the fallback shard to its own shard using Qdrant's internal shard transfer, transparent to the application, with reads and writes supported during the move.



**Corn**

The application not needing to know which tenants are dedicated — that's the interesting claim.



**Herman**

It's the whole design. Every request carries a shard key selector with two keys — target, the tenant's dedicated shard name, which may not exist yet, and fallback, the shared shard. If target exists and is active, route there; if not, route to fallback. The app sends the same request shape regardless, and it's a nice piece of API honesty that the target key is allowed to name a shard that isn't there.



**Corn**

We'll get to the tens in a moment, but first, Hilbert has his segment.



**Hilbert**

Medieval Pottage Facts. Garlic and onions were eaten raw by ordinary people as a cheap seasoning. It kept the meal interesting and the neighbours at a distance.



**Corn**

Practical on both counts.



**Herman**

So, promotion mechanics. The procedure itself is concrete. You create the new shard in initial state Partial, one shard, replication factor one. Then you post a replication job to the cluster endpoint — replicate points, with a filter on group ID equals the tenant, from shard key default, to shard key the tenant's name. When the transfer finishes, the target shard goes Active and all requests route to it automatically. Then — and this ordering matters — you delete the tenant's data from the fallback to free space, and only then do you create replicas.



### Corn

Why does the ordering matter? Other than the obvious.



### Herman

Because during the move, the tenant's data exists in both places, and the fallback copy is what keeps reads and writes alive if anything goes slowly. You delete it last for the same reason you don't pack the parachute while jumping. And there are constraints on setup — shard number must be one at collection creation, you still need the is_tenant index because the fallback shard holds many tenants, and the fallback shard has to fit on a single node. Plan capacity for that, because it's the one shard that can't be split.



### Corn

Now the money. You mentioned a field study.



### Herman

Aditya Mhaske's writeup, and it's one of the better ones because he did the migration and then did the math. Production system — industrial fault-log search, a hundred plus tenant environments, each a physically separate plant, originally one Qdrant collection per environment. He moved them all into a single pooled collection with is_tenant true, and resident memory dropped by roughly eighty percent.



### Corn

Eighty. Because the fixed overhead exceeded the payload.



### Herman

Because the tenants were small — a few thousand vectors each — and the fixed per-collection cost is constant in tenant size but linear in tenant count. His crossover estimate, in the seven-hundred-sixty-eight-dimension regime, is around fifty thousand vectors per tenant. Below that, per-partition fixed cost dominates, and per-tenant indexes are the single largest line item in your memory bill. His framing was that eighty percent of the memory budget was going to fixed overhead for partitions holding a few thousand vectors.



**Corn**

So a hundred plants, a hundred HNSW graphs, each mostly paying rent on an empty apartment.



**Herman**

Mostly paying for the privilege of having walls at all. And then — the part I love — there's counter-evidence, and it's good counter-evidence. Roland Szabó ran a benchmark in January of last year, up to two thousand users. Separate collections: three point eight milliseconds per search. Single collection with a keyword index: thirty-one point six. Without the index, thirty-five point eight. Roughly eight times faster with separate collections.



**Corn**

Eight times. The exact opposite recommendation.



**Herman**

His explanation is plausible — no filtering overhead, smaller search space per query, better cache utilization because the hot tenants' graphs stay in memory. And to his credit he flags the whole thing himself. His words: "of course this isn't a very comprehensive benchmark." It also predates the tenant index entirely. But the disagreement with Qdrant's own docs is real and out in the open — Qdrant's 1.16 blog says payload multitenancy "causes practically no overhead. Quite the opposite: a query with a tenant payload filter can be faster than a full search." Szabó measured it eight times slower.



**Corn**

Both of them looking at the same product and publishing opposite numbers, and neither obviously lying.



**Herman**

I think they're both right, and here's the reconciliation. Szabó tested small tenant counts, presumably with small per-tenant corpora, where a dedicated graph is tiny and sits in cache — of course it's fast. Qdrant's claim is about large shared collections, where the tenant index turns scattered reads into a sequential pass. Different regimes. And the honest version is: if you have a handful of small tenants, separate collections are faster and the memory cost is trivial. If you have thousands of them, the memory cost eats you



### Herman (continued)

before latency ever becomes the constraint.



### Corn

Szabó measured latency, Mhaske measured memory. The question for anyone listening is which regime they're actually in — and I suspect most people building the three things Daniel asked about don't know yet.



### Herman

Which is exactly what tiered multitenancy is for, so let's do the three workloads. First — the multi-user app with private memories per user. Think a personal AI assistant where every user's conversation history and notes get embedded and searched.



### Corn

That's the shape Daniel's own projects run in, so it's worth being concrete.



### Herman

Then this is the millions-of-small-users shape in miniature, and the answer is payload partitioning with the is_tenant index, almost unconditionally. Per-user collections blow the thousand-collection ceiling almost immediately — you'd hit it at a thousand users, which is a rounding error. And the crossover, fifty thousand vectors per user, is far above where a real user sits. Even a heavy personal-memory user, a few years of embedded notes — you're in the low tens of thousands, maybe.



### Corn

I'd say the typical user is a few thousand vectors and never thinks about it again.



### Herman

Right, and below the crossover, per-tenant indexes are pure overhead. So the level is clear. The risk in this workload isn't storage at all — it's enforcement. Every user is an untrusted party, the app is the only thing standing between user A's memories and user



### Herman (continued)

B's query, and the whole security story rests on the app layer applying the filter. Which leads us to the thing that has to be said plainly.



### Corn

Say it.



### Herman

The tenant index does not authorize anything. Qdrant's docs state it twice. "The tenant index helps storage and search. It does not authorize requests. A client with unrestricted credentials can omit the filter and access other tenants' points." And the companion line: "Derive the tenant identity from the authenticated user, rather than accepting an unrestricted tenant ID from the request." If your API takes a tenant ID from the request body and passes it to Qdrant, a client can change it. Full stop.



### Corn

That second line should be on a poster. Derive it, don't accept it.



### Herman

Mhaske's framing is the one I'd keep taped above the desk: "Pooled storage does not require pooled enforcement. Storage isolation is an efficiency decision; enforcement isolation is a security decision. Conflating them is how pooled architectures get their bad reputation." And then the line that really lands: "Pooling the first while siloing the second is not a compromise between them — it is the configuration that gets you the cost profile of a pool and the failure profile of a silo."



### Corn

The off-diagonal configuration. Pool economics with silo blast radius.



### Herman

That's it exactly. And notice what that means for level one — separate collections don't get you out of this either, if the app holds unrestricted credentials. Collection-per-tenant with an admin key is the same security posture as pooling, just with more overhead.



### Corn

So the private-memories app: pooled storage, siloed enforcement, and the app layer is the security boundary. Second workload — the few hundred business customers.



### Herman

This is the tiered sweet spot. A few hundred is comfortably under both the thousand-collection ceiling and the roughly thousand-dedicated-shard recommendation, so all four levels are physically available — which means you're choosing on tenant shape, not capacity. And tenant shape in B2B is a power law. A handful of big customers, a long tail of small ones, and — this is the line from the docs — "you might not know in advance which tenants will be large and which will stay small."



### Corn

That's the actual problem with splitting at signup. You're asking sales to predict the future.



### Herman

And application-level splitting has its own cost — extra logic, plus an additional source of truth for tenant placement that can drift from reality. Tiered says: start everyone on the fallback shard, promote on evidence. The documented threshold is around twenty thousand points — when a tenant's data grows past that, promote it to its own shard. You react to what the customer actually did instead of guessing what they might do.



### Corn

The promotion being transparent to the app means you're not doing a coordinated migration dance with your customers.



### Herman

Reads and writes keep working during the move. The shard transfer is Qdrant's internal mechanism, the target goes Active when it completes, routing flips on its own. The app doesn't know.



### Corn

What's the trap?



### Herman

The trap is that the decision to be ready for tiered is irreversible. You cannot change an existing collection's sharding method in place. If you want the tiered growth path, you must choose custom sharding at collection-creation time — day one, before you have any customers, before you know anything. The promotion itself is reversible and graceful; the door to it isn't.



### Corn

So the one decision you have to make blind is the door.



### Herman

And it's cheap insurance — creating the collection as custom-sharded with a fallback shard costs you nothing if every tenant stays small forever. You just never promote anyone. The other planning numbers: the fallback shard must fit on one node, so your biggest-shared-shard sizing is a real capacity question, and on Qdrant Cloud you disable automatic shard rebalancing before you start manually controlling placement.



### Corn

Third workload. Millions of small users.



### Herman

Here I'll flag that this is inference, not a documented recommendation — Qdrant's docs top out at "a large number of small, similarly-sized tenants." But the ceilings do the arguing: a thousand collections per cluster, roughly a thousand dedicated shards recommended. At millions of users, levels one, three, and four are arithmetically out. Payload partitioning is the only level left standing.



### Corn

Which makes the index-build problem acute, because you're building one giant index over everyone.



### Herman

And that's where the interesting config trick lives. The docs warn that indexing speed can bottleneck when many tenants share a collection, since their vectors are all indexed together. The fix: disable the global vector index and build it only per group. In HNSW config, set payload_m to a non-zero value like sixteen, and set m to zero.



### Corn

Wait — set m to zero. That disables the index, and then you build it... per payload group?



### Herman

The payload-aware variant, yes. The trap is you must set both. If you leave payload_m unset, Qdrant uses the value of m for the payload links too — so m zero on its own would kill the payload-aware links as well, and you'd have no index anywhere. Both fields, or nothing.



### Corn

And the fine print?



### Herman

Two pieces. Global requests — anything without the group filter — get slower, because they scan all groups. So this configuration is only appropriate when every single search carries a tenant filter, which, in a millions-of-users memory app, is hopefully an invariant you can enforce. And Qdrant notes that m zero isn't the same as exact search — with a positive effective payload_m you're still getting payload-aware links built, not brute force.



### Corn

Now the failure modes, because Daniel asked what goes wrong, and this is the part I'd want a listener to lose sleep over.



### Herman

The silent one is recall. Mhaske again: "Filtering is not free isolation. A tenant predicate over a shared proximity graph degrades recall in a way that scales with how selective the predicate is — the failure is silent and does not show up in latency graphs."



### Corn

Fast, cheap, and wrong.



### Herman

Potentially very wrong. Qdrant's own benchmark for this — the ACORN work in 1.16 — five million vectors, dimension ninety-six, two payload fields with five values each, two filters applied so about four percent of vectors pass. At ef sixty-four: fifty-three percent accuracy. You're missing nearly half the true neighbors. Raise ef to five hundred twelve — you only get to seventy-one percent, and you've multiplied your search effort along the way. The fix is ACORN itself, enabled per query with the acorn parameter, no index-time change: ef sixty-four plus ACORN gets you ninety-seven percent accuracy, at thirteen point nine milliseconds instead of one point two five.



### Corn

So eleven times the latency to go from half-right to right.



### Herman

Which is exactly why Qdrant's guidance is not to enable it on every query — only for multiple filters, low selectivity. It's a per-query decision matrix, which means someone has to know the selectivity at query time, which is its own operational burden. But the deeper point for the multitenancy question is what a team can ship without noticing: a pooled system, fast, cheap, dashboard green, quietly returning the wrong neighbors to small tenants.



### Corn

Because nobody builds a recall dashboard per tenant.



### Herman

Nobody builds a recall dashboard at all, usually. The other failure axis is operational. Mhaske: "Operational isolation is the axis that gets discovered last and hurts most. Per-tenant restore, per-tenant erasure, and per-tenant cost attribution are trivial in a silo and are genuine engineering projects in a pool."



### Corn

Right to erasure, in a pool, is a fun afternoon.



### Herman

A GDPR delete request in a pooled vector collection means finding and removing one tenant's vectors from a shared HNSW graph without degrading everyone else's index. Doable — Qdrant supports filtered deletion — but it's an engineering project, not a checkbox. And per-tenant cost attribution, if your billing is usage-based, means tracking per-tenant usage inside a structure that was designed to make tenants indistinguishable at the storage layer.



### Corn

There's one more dial you haven't mentioned — the scoring one.



### Herman

Per-tenant IDF. This one's new, version 1.19. If you're doing BM25 full-text or sparse-vector search with the IDF modifier, the statistics — document counts, term rarity, average document length — are computed across the entire shard being queried. Under payload multitenancy that blends every tenant's data into one set of statistics, so a term's IDF no longer reflects its rarity within a specific tenant's data. A term that's distinctive inside one tenant's corpus gets diluted by being common across everyone's. The idf search parameter narrows the population to a single tenant.



### Corn

Which matters the moment relevance is per-tenant, which in a SaaS it always is.



### Herman

It's the fourth dial resurfacing in a place nobody thought to look. Even your scoring statistics are a shared resource.



### Hilbert

I ran a recipe search thing for a while, my own embeddings, my own notes. Eleven cuisines. One collection per cuisine. Corn will tell me that's the same as tenant splitting, it is not. Italian and Thai have nothing to say to each other. A tenant ID is an accounting fact. A cuisine is a semantic fact. Herman, you said the overhead is constant per partition — fine, but mine are meaningful partitions.



### Corn

How many vectors per cuisine, Hilbert?



### Hilbert

Around four hundred. Eleven collections, four hundred vectors each, plus a notes collection that is off by itself because it is notes, not recipes. When I change the embedding model I have a shell script that drops all twelve and rebuilds them from the source text. People complain about re-embedding. The script takes eleven minutes and it makes you think about whether you really want to. I always want to.



**Herman**

You've never once decided not to re-embed.



**Hilbert**

No. But the thinking is the point. The scripts are numbered, one to eleven, and I have never needed to renumber them. I index the prompt cache too, which is on the same machine, and that was a nuisance to set up with claws on a laptop keyboard, but it is done now. The cuisines do not mix and the notes do not mix with the cuisines. That is the whole architecture and it has never failed me.



**Corn**

Semantic partitions. Four hundred vectors each.



**Hilbert**

They are not tenants.



**Herman**

No, they're cuisines.



**Hilbert**

Thank you.



**Corn**

The thing that's unresolved here is that nobody has published a benchmark of tiered multitenancy itself. No measurement of promotion cost, no query-latency comparison of a promoted tenant versus one still on the fallback shard, no overhead figure for a cluster running five hundred dedicated shards. The twenty-thousand-point threshold and the thousand-shard ceiling are stated as recommendations, without a published derivation behind either.



### Herman

And the recall question is even more open. The ACORN numbers come from two low-selectivity payload filters, not a tenant filter. The claim that a tenant predicate degrades recall silently is Mhaske's field observation — a good one, from a real migration — but there's no published curve for the exact case most listeners care about. So the silent-failure risk is real and, right now, unquantified.



### Corn

Before we close, the misconception — the one I'd bet most teams walk in with.



### Herman

That the tenant index is a security feature. It isn't. It's a storage and search optimization — a very good one — and a client with unrestricted credentials can simply omit the filter and read another tenant's points. Isolation of enforcement lives in your application layer and your credentials, and it lives there no matter which of the four levels you pick. The other half of it is that pooled versus siloed isn't one decision. It's four dials, and the configuration you probably want — pool the storage, silo the enforcement, know which shards are whose for the operational stuff — is only available if you notice they're separate.



### Corn

Next time someone says we'll just pool it, ask which of the four dials. Then enjoy the silence.



### Herman

That question has ended architecture meetings, I promise you.



### Corn

Thanks as always to Hilbert Flumingtop at the desk — the man keeps twelve collections in alphabetical order and the show in broadcast order. If this was your kind of episode, go back for episode eleven, How Does Fine Tuning Work Anyway; episode twenty, Architectural AI; and episode five, Fine-Tuning ASR For Maximal Usability. This has been My Weird Prompts. You can find everything at my weird prompts dot com.



**Herman**

See you tomorrow, everyone.



**Corn**

See you tomorrow.

# SOURCES

What the research for this episode read before the script was written. Primary sources first.


1. **Qdrant, Configure Multitenancy, retrieved 2026-11-10 [primary]**
   https://qdrant.tech/documentation/manage-data/multitenancy/

2. **Qdrant, Choose a Multitenancy Design, retrieved 2026-11-10 [primary]**
   https://qdrant.tech/documentation/deploy-and-scale/multitenant-search/

3. **Qdrant, agent-readable mirror of the multitenancy docs (includes Cloud 1,000-collection limit, `is_tenant` v1.11.0 note, per-tenant IDF v1.19.0), retrieved 2026-11-10 [primary]**
   https://skills.qdrant.tech/md/documentation/manage-data/multitenancy/

4. **Qdrant, Tiered Multitenancy section with full promotion procedure and limitations, retrieved 2026-11-10 [primary]**
   https://skills.qdrant.tech/md/documentation/manage-data/multitenancy/?s=tiered-multitenancy

5. **Abdon Pijpelink, Qdrant 1.16 [primary]**
   Tiered Multitenancy & Disk-Efficient Vector Search, published 2025-11-19, retrieved 2026-11-10
   https://qdrant.tech/blog/qdrant-1.16.x/

6. **Qdrant, older multitenancy guide URL (now redirecting/superseded by source 1), retrieved 2026-11-10 [primary]**
   https://qdrant.tech/documentation/guides/multiple-partitions/

7. **Roland Szabó, Optimizing performance in Qdrant, published 2025-01-10, retrieved 2026-11-10**
   https://rolisz.com/optimizing-performance-in-qdrant/

8. **Aditya Mhaske, Isolation Is Four Dials · Multi-Tenant Retrieval Systems, retrieved 2026-11-10**
   https://adityamhaske.com/articles/multi-tenancy