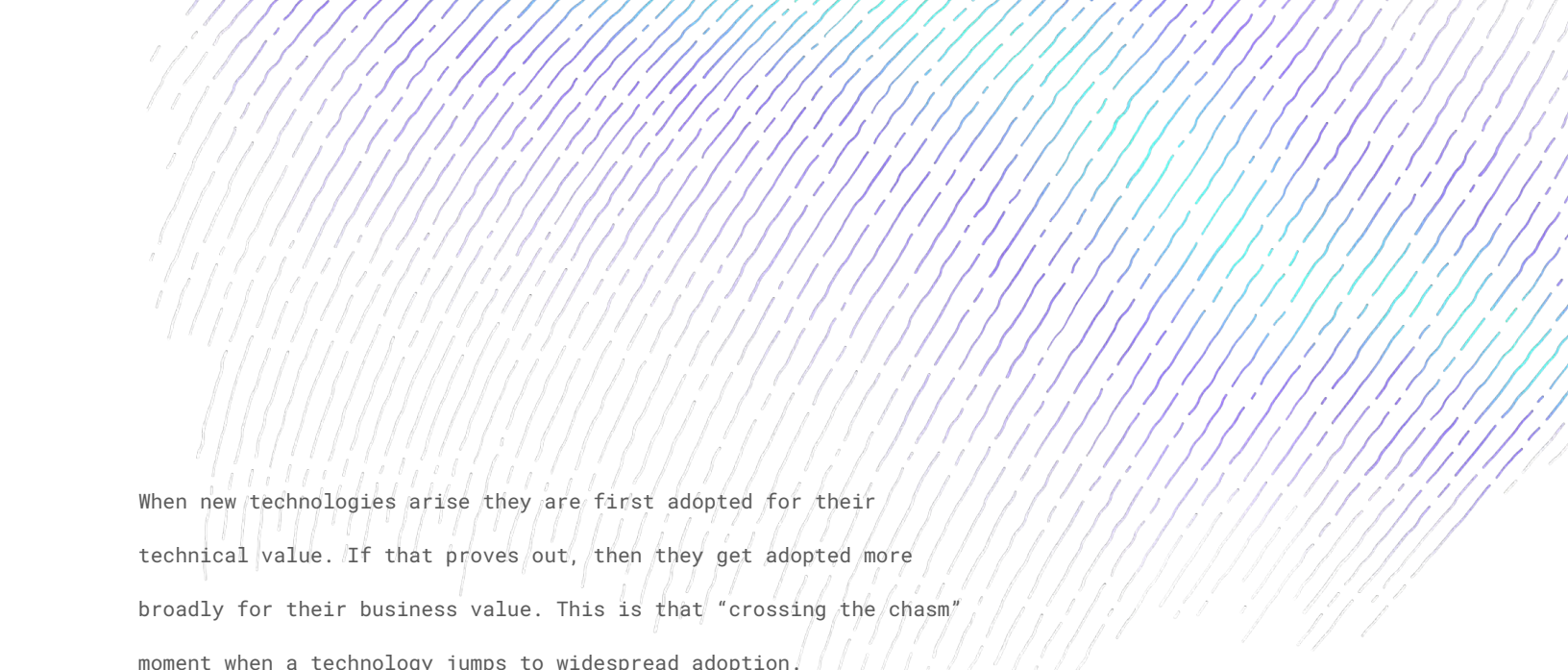


Architecture of a Serverless Database

```
/* What is a serverless database, how does it work,  
and when should you use it? */
```



Cockroach
Labs



When new technologies arise they are first adopted for their technical value. If that proves out, then they get adopted more broadly for their business value. This is that "crossing the chasm" moment when a technology jumps to widespread adoption.

Serverless architecture is being adopted right now for its technical value. Soon it will become mainstream. And then it's going to make one more leap from widespread adoption to existential imperative.

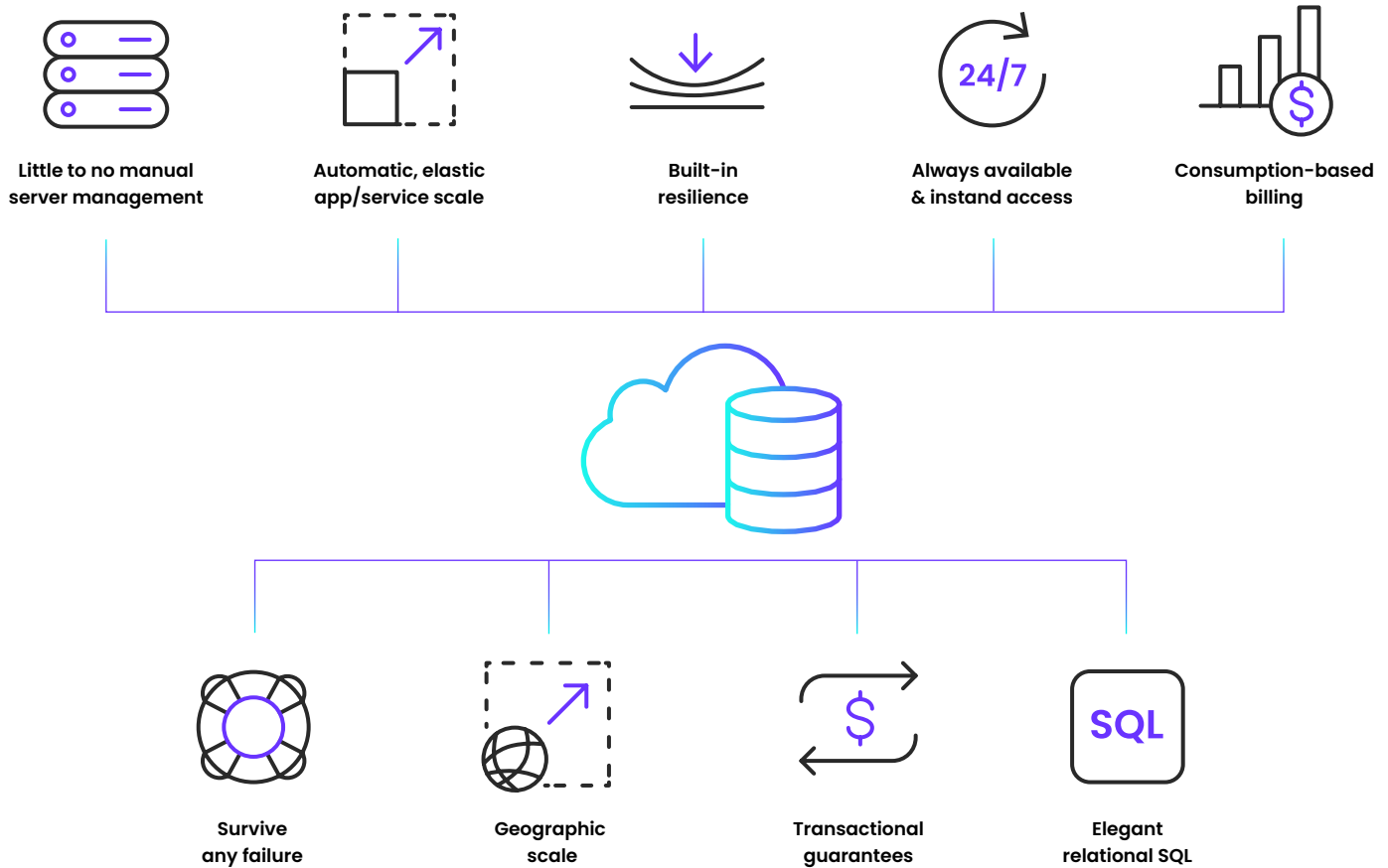
Serverless architecture will eventually become one of those paradigm shifting technologies that almost every business needs to adopt in order to survive. Relational databases did this in the 1970s. Graphic interfaces did this for personal computers. The rise of the internet gave us email, e-commerce, and mobile computing.

Serverless is the next paradigm shift. Now is a good time to start understanding it.

What is a Serverless Database?	5
Key Capabilities of a Serverless Database	6–8
The Architecture of a Serverless Database	9–14
Serverless Architecture	15–17
Serverless Database Scaling	18–21
When to Use a Serverless Database	22–25
When NOT to Use a Serverless Database	26
Future of Serverless Architecture	27

What is a Serverless Database?

A serverless database is any database that embodies the core principles of the serverless computing paradigm. The exact flavor of the application doesn't matter; whether a serverless database, a cloud data warehouse or even a custom backend to a CRM app – anything calling itself serverless should be built with the following principles in mind.

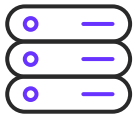


These are some additional, explicit principles that help define a serverless database. Data should be accurate and of high integrity, but – and of equal importance – data must also be available everywhere, and with very low latency. The very nature of serverless is inherently multi-regional: never tied to a single region and able to deliver all of this value, anywhere and everywhere.

When all of these elements come together we get a look at the next generation of what our databases will become: A familiar database that's delivered as a service, eliminates ops and reduces costs down to a count of the transactions and required storage used by your application.

What Are the Key Capabilities of a Serverless Database?

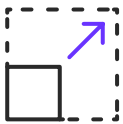
Now that we've defined the core principle of serverless applications, let's examine what each of these mean and how they are specifically realized in a serverless database.



Little to No Manual Server Management

Serverless is quite the misnomer; it is actually a bunch of servers that are abstracted away and automated so you don't have to manage them. The manual tasks of provisioning, capacity planning, scaling, maintenance, updates, etc, all still happen but are behind the scenes. Using them requires little to no manual intervention and very limited thought.

A serverless database eliminates the operational overhead of deployment, capacity planning, upgrading and management. It does all this without downtime and allows developers to focus on what matters – coding business logic.



Automatic, Elastic App/Service Scale

Within the serverless paradigm, elastic scale allows your service or app to consume the right amount of resources necessary for whatever your workload demands at any time. This elastic scale is automated and requires no changes to your app and will help optimize compute costs.

A serverless database allows for elastic scale for both storage and read/write transactional volumes. It will expand and contract based on workload all without interaction from a developer or ops personnel. It will scale up for a sudden spike in traffic and scale back down post-event to make sure you only pay for the compute you actually need/use.



Built-in Resilience & Inherently Fault-Tolerant

A serverless application will survive the failure of any backend compute instance, or other network or physical issue. This resilience allows for your service to be always available even as you upgrade.

A serverless database will survive backend failures and guarantee data correctness even when these issues happen. It can even allow you to implement online schema changes and roll software updates across multiple instances of the database, all without issue and while always remaining available. It eliminates both planned and unplanned downtime.



Always Available & Instant Access

A key challenge with any serverless compute platform is not just ensuring a service is available but also that you can “wake it up” quickly whenever it has been dormant but then is all of sudden needed. Putting a non-used service to sleep allows you to save money on compute, but it should be there instantly whenever you need it again.

Your apps and services will rely on your serverless database to be always on and always available. More importantly, it should minimize any “waking up” time so that all customer requests are serviced in a timely manner.



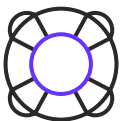
Consumption-based Rating of Billing Mechanism

A few of these core serverless capabilities help conserve compute. However, they are only as good as their built-in mechanism that tracks activity. A serverless app needs to be able to capture how much it consumes.

For a database, the two vectors of consumption are storage and transactional volume. A serverless database should capture both so that the app owner only has to pay for what they consume. Large apps may cost more than small but, no matter what, each can grow volume and the relative spend accordingly. It’s serverless, so you only pay for the resources you use, when you use them.

The next three requirements are unique to a serverless database as the concepts of ubiquity and availability have a very profound effect on how to think about data and transactions – especially in serverless, and especially at global scale. If you agree that the serverless concept should divorce us from worrying about servers, then shouldn’t this also extend to worrying about where we run serverless?

For a serverless database the “where” presents significant problems as the speed of light encumbers the speed at which we can perform transactions, especially if we want any guarantee that data is correct. We can’t simply set up asynchronous replication to accomplish this as the serverless system is too fluid and too complex.



Survive Any Failure Domain, Including an Entire Region

Behind the scenes, a serverless platform will most likely be available across multiple regions. When you do not need to persist anything, this is relatively simple. However, if you want any sort of guarantees for data correctness and availability, it presents a significant challenge for the application. In order for a serverless database to survive the collapse of any failure domain (instance, rack, AZ, region, cloud provider, etc) it should persist multiple copies of data and then intelligently control where the data resides to avoid these failures. The placement of data should be able to be controlled, but by default automated to maximize availability.



Geographic Scale

Within the context of a database, scale is typically thought of as the total storage volume needed, but this also extends to the necessary transactional scale. For a serverless database, scale can also be extended to geography as data is needed everywhere.

The database should automate geo-partitioning, moving data dynamically around the globe to minimize data access latency concerns. And, in more mature serverless databases, it will be able to automate the placement of data cross regions in order to help solve regional data domiciling requirements (such as GDPR).



Transactional Guarantees

If we want transactions within a serverless database, we need to place a lot of attention on their performance across board geographies. After all, if you have a user hitting your service in Asia and another in Europe and they are both trying to access the same record, who wins? Transactional guarantees and data integrity are less complex in a single region, but we aren't talking about a single region when we speak of the ultimate definition of serverless. Serverless quite literally knows no bounds.



The Beauty & Elegance of a Relational Database (SQL)

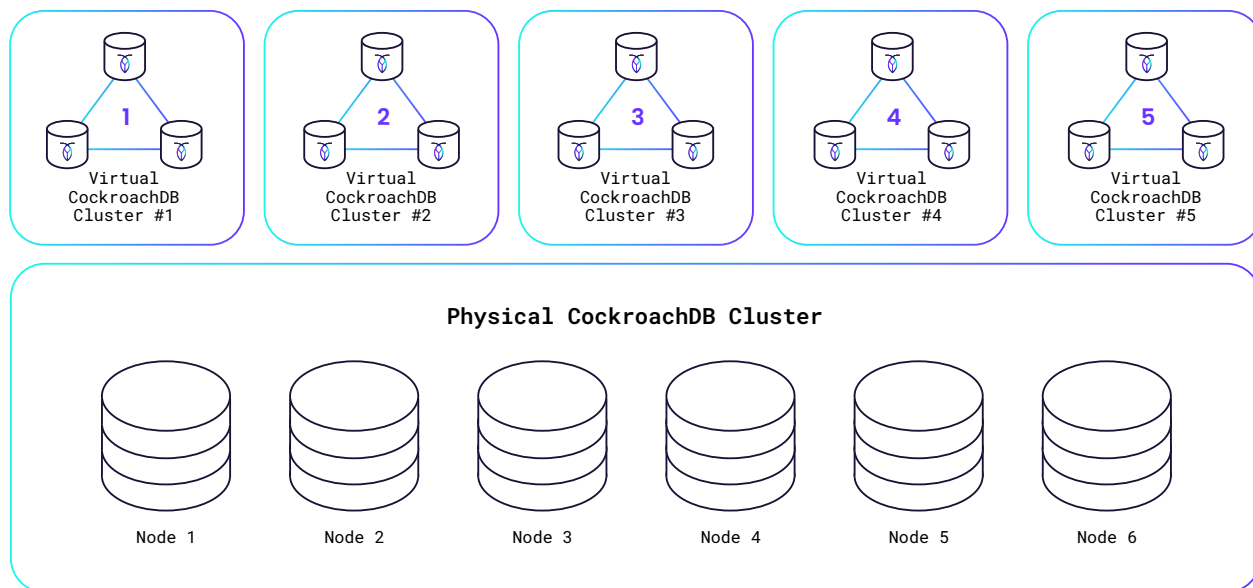
Finally, we have been talking about a database throughout all of these requirements thus far. There are many types, but the most complex to deliver is a relational database that delivers referential integrity, joins, secondary indexes, etc.

Delivering this the elegance of SQL may not be a requirement of every serverless database, but it should be considered an adjunct and type as it is important to most of our operational workloads.

The Architecture of a Serverless Database

If you've created a database before, you probably had to estimate how many servers to use based on the expected traffic. If you guessed too low, your database would fall over under load and cause an outage. If you guessed too high or if your traffic came in bursts, you'd waste money on servers that are just sitting idle. A truly serverless database alleviates those concerns.

At Cockroach Labs we've created an innovative serverless architecture that allows us to securely host thousands of virtualized CockroachDB database clusters on a single underlying physical CockroachDB database cluster. This means that a tiny database with a few kilobytes of storage and a handful of requests costs us almost nothing to run, because it's running on just a small slice of the physical hardware. All the details will be explained below, but here's a diagram to get you thinking.



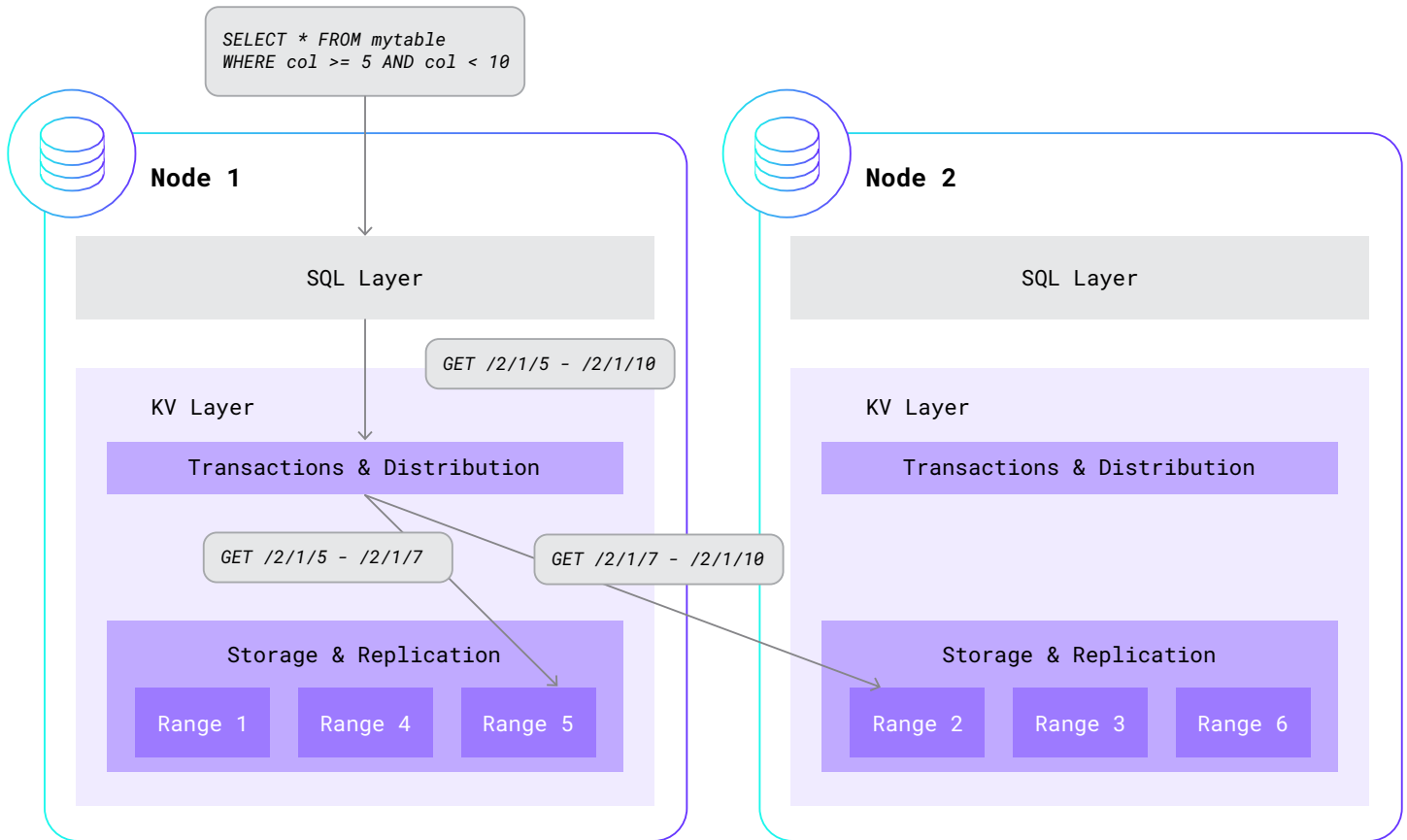
Serverless Architecture

Single-Tenant Architecture

Before the release of CockroachDB Serverless, a single physical CockroachDB cluster was intended for dedicated use by a single user or organization. That is called single-tenancy. Over the past several CockroachDB releases, we've quietly been adding multi-tenancy support, which enables the physical CockroachDB cluster to be shared by multiple users or organizations (called "tenants"). Each tenant gets its own virtualized CockroachDB cluster that is hosted on the physical CockroachDB cluster and yet is secure and isolated from other tenants' clusters. You're probably familiar with how virtual machines (VMs) work, right? It's kind of like that, only for database clusters.

In order to effectively explain how multi-tenancy works, we need to review the [single-tenant architecture](#). To start with, a single-tenant CockroachDB cluster consists of an arbitrary number of nodes. Each node is used for both data storage and computation, and is typically hosted on its own machine. Within a single node, CockroachDB has a layered architecture. At the highest level is the SQL layer, which parses, optimizes, and executes SQL statements. It does this by a [clever translation of higher-level SQL statements to simple read and write requests](#) that are sent to the underlying key-value (KV) layer.

The KV layer maintains a transactional, distributed, replicated key-value store. That's a mouthful, so let's break it down. Each key is a unique string that maps to an arbitrary value, like in a dictionary. KV stores these key-value pairs in sorted order for fast lookup. Key-value pairs are also grouped into ranges. Each range contains a contiguous, non-overlapping portion of the total key-value pairs, sorted by key. Ranges are distributed across the available nodes and are also replicated at least three times, for high-availability. Key-value pairs can be added, removed, and updated in all-or-nothing transactions.



Simplified example of how a higher-level SQL statement gets translated a simple KV GET call.

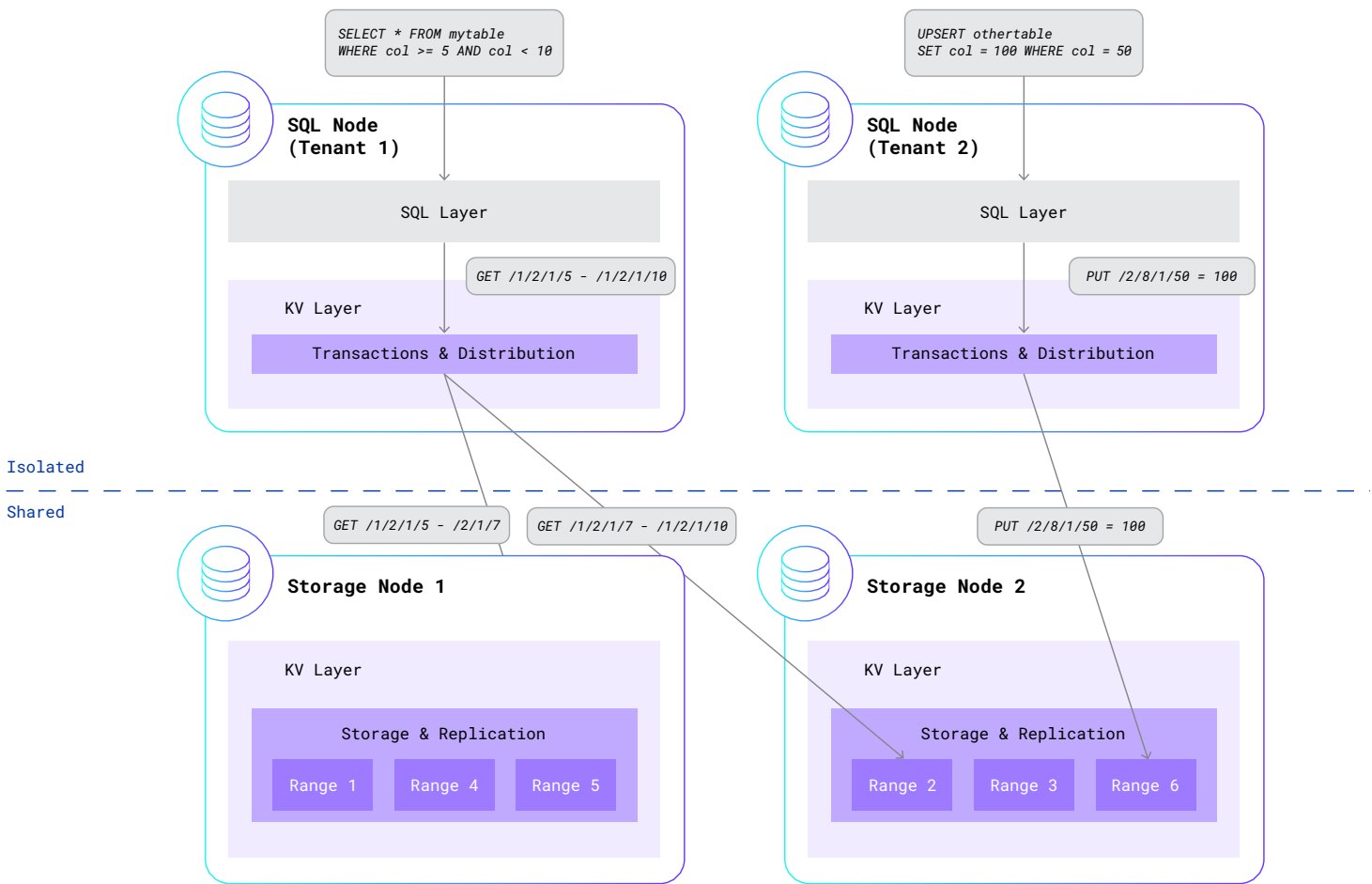
In single-tenant CockroachDB, the SQL layer is co-located with the KV layer on each node and in the same process. While the SQL layer always calls into the KV instance that runs on the same node, KV will often “fan-out” additional calls to other instances of KV running on other nodes. This is because the data needed by SQL is often located in ranges that are scattered across nodes in the cluster.

Multi-Tenant Architecture

How does CockroachDB extend that single-tenant architecture to support multiple tenants? Each tenant should feel like they have their own dedicated CockroachDB cluster, and should be isolated from other tenants in terms of performance and security. But that's very difficult to achieve if we attempt to share the SQL layer across tenants. One tenant's runaway SQL query could easily disrupt the performance of other tenants in the same process. In addition, sharing the same process would introduce many cross-tenant security threats that are difficult to reliably mitigate.

One possible solution to these problems would be to give each tenant its own set of isolated processes that run both the SQL and KV layers. However, that creates a different problem: CockroachDB would be unable to share the key-value store across tenants. That eliminates one of the major benefits of a multi-tenant architecture: the ability to efficiently pack together the data of many tiny tenants in a shared storage layer.

After mulling over this problem, it became clear that the dilemma can be elegantly solved by isolating some components and sharing other components. Given that the SQL layer is so difficult to share, it made sense to isolate that in per-tenant processes, along with the [transactional](#) and [distribution](#) components from the KV layer. Meanwhile, the KV replication and storage components continue to run on storage nodes that are shared across all tenants. By making this separation, CockroachDB gets “the best of both worlds” – the security and isolation of per-tenant SQL processes and the efficiency of shared storage nodes.



Two isolated, per-tenant SQL nodes interacting with a shared storage layer.

The storage nodes no longer run tenant SQL queries, but they still leverage the sophisticated infrastructure that powers single-tenant CockroachDB. Node failures are detected and repaired without impacting data availability. [Leaseholders](#), which serve reads and coordinate writes for each range, move according to activity. Busy ranges are automatically split; quiet ranges are merged. Ranges are rebalanced across nodes based on load. The storage layer caches hot ranges in memory and pushes cold ones to disk. Three-way replication across availability zones ensures that your data is safely stored and highly available.

After seeing this architecture, you might be wondering about the security of the shared storage nodes. Significant time was spent designing and implementing strong security measures to protect tenant data. Each tenant receives an isolated, protected portion of the KV keyspace. This is accomplished by prefixing [every key generated by the SQL layer](#) with the tenant's unique identifier.

Rather than generating a key like:

```
1 /<table-id>  
2 /<index-id>  
3 /<key>  
4  
5
```

SQL will generate a key like:

```
1 /<tenant-id>  
2 /<table-id>  
3 /<index-id>  
4 /<key>  
5
```

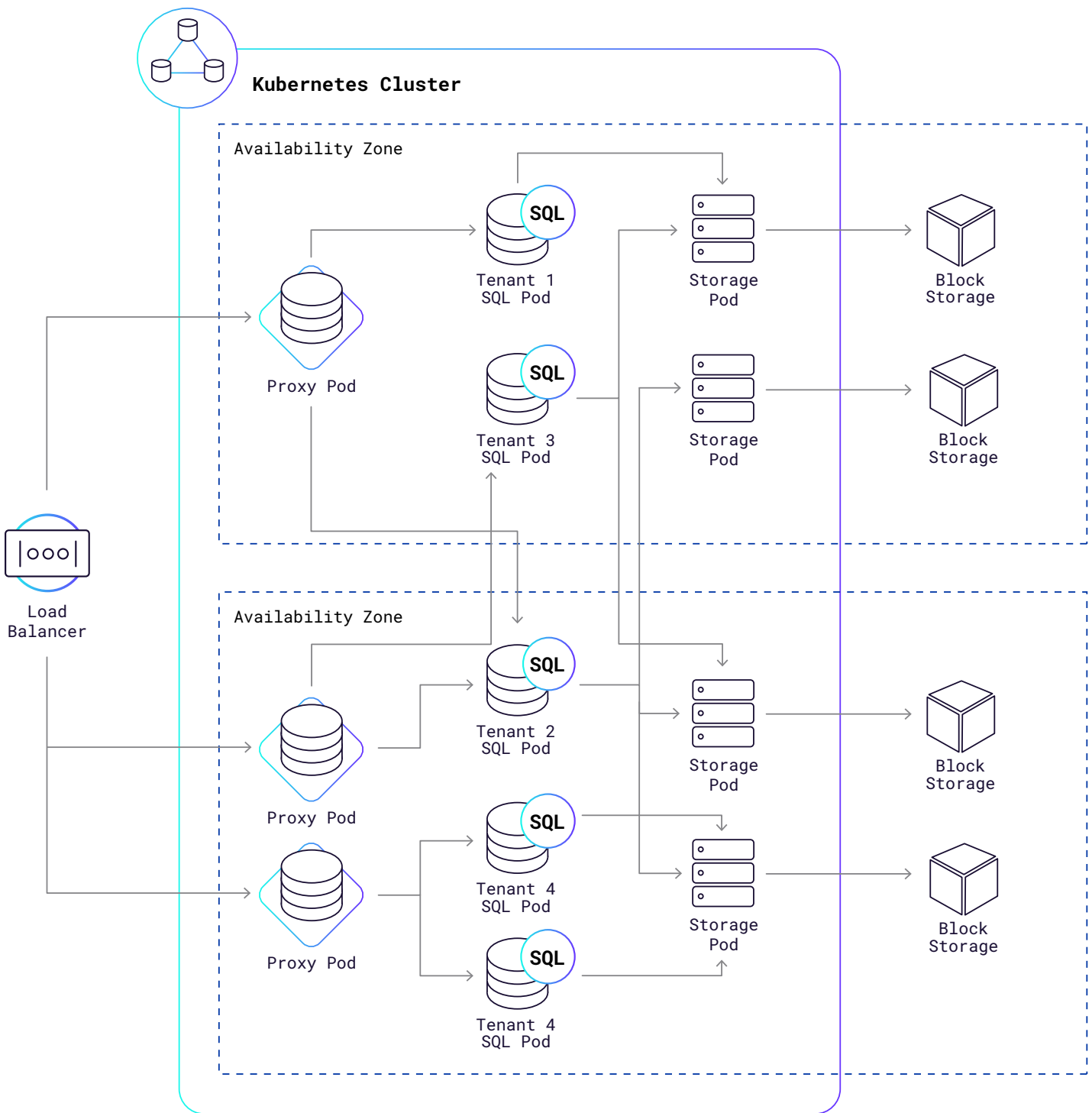
This means that key-value pairs generated by different tenants are isolated in their own ranges. Furthermore, the storage nodes authenticate all communication from the SQL nodes and ensure that each tenant can only touch keys that are prefixed by its own tenant identifier.

Besides security, we were also concerned about ensuring basic quality of service across tenants. What happens when multiple tenants are making KV calls to the same storage node? In order to ensure that a single tenant cannot monopolize resources on a storage node, we measure the number and size of read and write requests coming from each tenant and throttle its activity if it exceeds some threshold. Unlike SQL statements, KV calls are relatively simple operations like GET, PUT, and DELETE on key-value pairs that can be effectively regulated in a shared process.

Serverless Architecture

Wait...wasn't the last section about the serverless architecture? Well, yes and no. As discussed, CockroachDB made significant upgrades to the core database architecture to support multi-tenancy. But that's only half of the story. In order to make serverless possible, big enhancements were made to how multi-tenant CockroachDB clusters are deployed and operated.

The managed version of CockroachDB (CockroachDB Dedicated) uses Kubernetes (K8s) to operate serverless clusters, including both shared storage nodes and per-tenant SQL nodes. Each node runs in its own K8s pod, which is not much more than a Docker container with a virtualized network and a bounded CPU and memory capacity. Dig down deeper, and you'll discover a Linux cgroup that can reliably limit the CPU and memory consumption for the processes. This allows us to easily meter and limit SQL resource consumption on a per-tenant basis. It also minimizes interference between pods that are scheduled on the same machine, giving each tenant a high-quality experience even when other tenants are running heavy workloads.



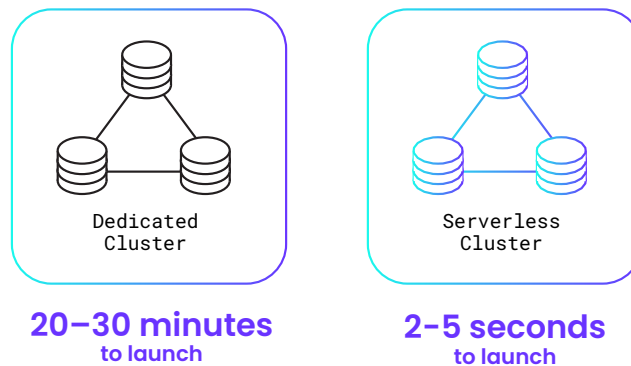
High-level (simplified) representation of what a typical setup looks like.

What are those “proxy pods” doing in the K8s cluster? It turns out they’re pretty useful:

They allow many tenants to share the same IP address. When a new connection arrives, the proxy “sniffs” the incoming Postgres connection packets in order to find the tenant identifier in a PG connection option. Now it knows which SQL pods it should route that connection to.

- 1 They balance load across a tenant’s available SQL pods. New connections are routed to the pod with the least CPU load.
- 2 They detect and respond to suspected abuse of the service. This is one of the security measures we take for the protection of your data.
- 3 They automatically resume tenant clusters that have been paused due to inactivity. We’ll get into more detail on that in the Scaling section below.

After the cloud load balancer routes a new connection to one of the proxy pods, the proxy pod will in turn forward that connection to a SQL pod owned by the connecting tenant. Each SQL pod is dedicated to just one tenant, and multiple SQL pods can be owned by the same tenant. Network security rules prevent SQL pods from talking to one another, unless they are owned by the same tenant. Finally, the SQL pods communicate via the KV layer to access data managed by the shared storage pods, each of which stores that data in a cloud provider block storage system like AWS EBS or GCP PD.



One of the best things about serverless clusters is how fast they can be created. A regular Dedicated cluster takes 20-30 minutes to launch, since it has to create a cloud provider project, spin up new VMs, attach block storage devices, allocate IP and DNS addresses, and more. By contrast, a serverless cluster takes just a few seconds to create, since we only need to instruct K8s to create a new SQL pod on an existing VM that it is already managing.

Besides speed of creation, serverless SQL pods also have a big cost advantage. They can be packed together on a VM, sharing the same OS as well as available CPU and memory. This substantially reduces the cost of running “long-tail” tenants that have minuscule workloads, since they can each use just a small slice of the hardware. Contrast this with a dedicated VM, which generally requires at least 1 vCPU and 1GB of memory to be reserved for it.

Serverless Database Scaling

As the amount of data owned by a tenant grows, and the frequency with which that data is accessed grows, the tenant's data will be split into a growing number of KV ranges which will be spread across more shared storage pods. Data scaling of this kind is already well supported by CockroachDB, and operates in about the same way in multi-tenant clusters as it always has in single-tenant clusters.

Similarly, as the number of SQL queries and transactions run against a tenant's data increases, the compute resources allocated to that tenant must grow proportionally. One tenant's workload may need dozens or even hundreds of vCPUs to execute, while another tenant's workload may just need a part-time fraction of a vCPU. In fact, we expect most tenants to not need any CPU at all. This is because a large proportion of developers who try CockroachDB Serverless are just "kicking the tires". They'll create a cluster, maybe run a few queries against it, and then abandon it, possibly for good. Even keeping a fraction of a vCPU idling for their cluster would be a tremendous waste of resources when multiplied by all inactive clusters. And even for tenants who regularly use their cluster, SQL traffic load is not constant; it may greatly fluctuate from day to day and hour to hour, or even second to second.

How does CockroachDB Serverless handle such a wide range of shifting resource needs? By dynamically allocating the right number of SQL pods to each tenant, based on its second-to-second traffic load. New capacity can be assigned instantly in the best case and within seconds in the worst case. This allows even extreme spikes in tenant traffic to be handled smoothly and with low latency. Similarly, as traffic falls, SQL processing capacity can be reassigned elsewhere, so that there is a minimum of unused capacity. If traffic falls to zero, then all SQL pods owned by an inactive tenant are terminated, and yet a new SQL pod can be spun back up within a few hundred milliseconds as soon as new traffic arrives. This allows a seldom-used CockroachDB Serverless cluster to still offer production-grade latencies for almost no cost to Cockroach Labs, and no cost at all to the user.

Such responsive scaling is only possible because multi-tenant CockroachDB splits the SQL layer from the KV storage layer. Because SQL pods are stateless, they can be created and destroyed at will, without impacting the consistency or durability of tenant data. There is no need for complex coordination between pods, or for careful commissioning and decommissioning of pods, as we must do with the stateful storage pods to ensure that all data stays consistent and available. Unlike storage pods, which typically remain running for extended periods of time, SQL pods are ephemeral and may be shut down within minutes of starting up.

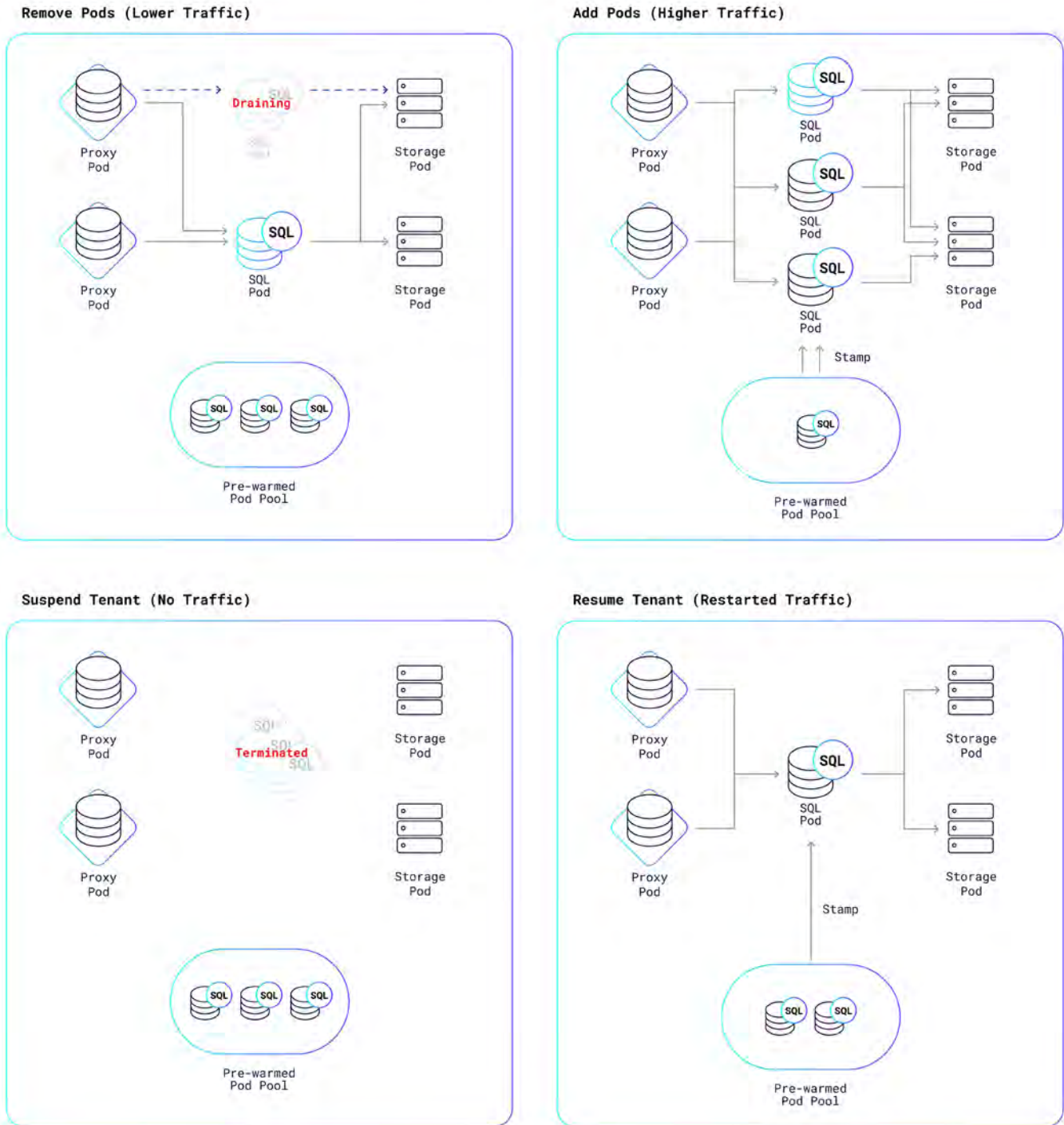
The Autoscaler

Let's dig a little deeper into the mechanics of scaling. Within every serverless cluster, there is an autoscaler component that is responsible for determining the ideal number of SQL pods that should be assigned to each tenant, whether that be one, many, or zero. The autoscaler monitors the CPU load on every SQL pod in the cluster, and calculates the number of SQL pods based on two metrics:



Average CPU usage determines the “baseline” number of SQL pods that will be assigned to the tenant. The baseline deliberately over-provisions SQL pods so that there is spare CPU available in each pod for instant bursting. However, if peak CPU usage recently exceeded even the higher over-provisioned threshold, then the autoscaler accounts for that by increasing the number of SQL pods past the baseline. This algorithm combines the stability of a moving average with the responsiveness of an instantaneous maximum. The autoscaler avoids too-frequent scaling, but can still quickly detect and react to large spikes in load.

Once the autoscaler has derived the ideal number of SQL pods, it triggers a K8s reconciliation process that adds or removes pods in order to reach the ideal number.



This diagram shows the possible outcomes.

As the previous diagram shows, CockroachDB maintains a pool of “prewarmed” pods that are ready to go at a moment’s notice; they just need to be “stamped” with the tenant’s identifier and security certificates. This takes a fraction of a second to do, versus the 20–30 seconds it takes for K8s to create a pod from scratch. If instead, pods need to be removed, they are not abruptly terminated, because that would also result in the rude termination of all SQL connections to that pod. Rather, the pods are put into a draining state, which gives them a chance to shed their SQL connections more gracefully. A draining pod is terminated once all connections are gone or once 10 minutes have passed, whichever comes first.

If application load falls to zero, then the autoscaler will eventually decide to suspend the tenant, which means that all of its SQL pods are removed. Once the tenant no longer owns any SQL pods, it does not consume any CPU, I/O, or bandwidth. The only cost is for storage of its data, which is relatively cheap compared to other resources. This is one of the reasons that CockroachDB can offer free database clusters to all of you.

However, there is one problem left to solve. How can a tenant connect to its cluster if there are no SQL pods assigned to it? To answer that question, remember that a set of proxy pods runs in every Serverless cluster. Each SQL connection initiated by an external client is intercepted by a proxy pod and then forwarded to a SQL pod assigned to the tenant. However, if the proxy finds that there are currently no SQL pods assigned to the tenant, then it triggers the same K8s reconciliation process that the autoscaler uses for scaling. A new pod is pulled from the prewarmed pool of SQL pods and stamped, and is now available for connections. The entire resumption process takes a fraction of a second (and it’s getting faster).

CockroachDB allows you to scale easily by simply adding additional nodes to a cluster, and the database will take care of creating ranges (similar to shards) that are then balanced across the nodes according to policy set at the table level. There is no need to perform costly, painful, manual sharding.

With CockroachDB, every node is a consistent gateway to the entirety of the database, which means every node can service reads and writes and execute queries across the distributed cluster. This eliminates the write bottleneck concern and ensures scale for not only size of database but also for volume of transactions.

When To Use a Serverless Database

Now that you understand the architecture of a serverless database there are two logical followup questions. What are serverless databases actually good for? When is a serverless database not the best choice?

When you need automated elastic scaling

Any database can “scale,” in the sense that it’s always possible to throw more engineers and machines at a problem. But that approach is expensive, inefficient, and it doesn’t really work for spiky workloads or “surprise” scaling incidents like having your application suddenly go viral.

Serverless databases solve this problem because they can scale up and down automatically, reacting almost instantly to changes in demand. When traffic to your application is light, your serverless database will scale down so that it consumes fewer compute resources (and thus costs you less). When traffic spikes, it scales back up to meet the new demand without impacting application performance.

Automated scaling is beneficial for any business, but it’s particularly critical for businesses with spiky or unpredictable workloads. Instead of having to try to predict the future when you’re doing capacity planning, you just spin up a database cluster and let it take care of the scaling for you.



When you need to minimize costs

Traditionally, database servers were consumed machine by machine. In other words, you paid for as many servers as your workloads required. If you were not using all of the storage and compute capacity of those machines you were paying for, too bad. No refunds.

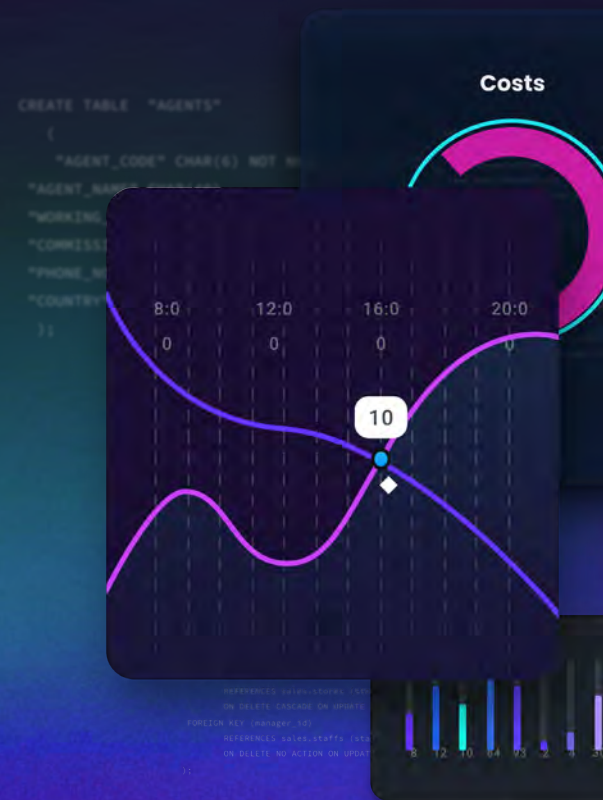
The reverse was also true — if you were paying for a set number of machines and your workload spiked beyond their capacity, your application's performance would suffer.

Serverless databases solve both of these problems by billing you for your actual storage and compute usage, not by the machine. This is made possible by using a serverless architecture ([more details here](#)) in tandem with the seamless elastic scaling we just talked about.

The resources your database is allocated scale up and down automatically with demand, and so do your costs. CockroachDB Serverless also enables you to set monthly spend limits to ensure that you never exceed your budget.

The bottom line is that for many users, serverless databases offer more reliable database performance at lower cost. Serverless databases aren't always going to be the optimal choice for every application. But because of this only-pay-for-what-you-use architecture, they are an effective way to minimize database costs for a wide range of workloads.

It's also important to remember that the cost savings often go beyond just what you see on your bill...



When you need to minimize ops

Traditionally, businesses with transactional database workloads (i.e., most businesses) had to choose between two rather unappealing options:

Easy scale using a NoSQL database, but at the cost of losing transactional consistency and having to learn a new proprietary query language, or Maintain transactional consistency and familiarity of SQL, but spend considerable operational effort to scale using manual sharding.

Serverless databases such as CockroachDB Serverless give developers a way out of this choice. With automated scale and resilience, it can provide the ops-free scaling experience of NoSQL without losing ACID transactions and the other advantages of SQL databases.

In practical terms, what this means is that you and your team can spend less time worrying about the database and more time building your application. This makes serverless databases an especially good choice for any development team that wants to focus on development and remove the burden of database ops.



When you're testing, experimenting, or evaluating

Serverless databases are great for lightweight applications, prototypes, test and dev environments, side projects, etc. because they're self-serve, fast, and — at least in the case of CockroachDB Serverless — free.

Seriously, you can sign up for an account, spin up a cluster, and get it connected to a simple app in about five minutes, with no credit card required:

The entire process is also completely self-serve. This makes it great for experiments and side projects because it allows you to “kick the tires” and work out any bugs before committing a dime. Or, you can make it part of your regular prototyping process, or use it for testing in your CI pipeline.



```
last_name VARCHAR (50) NOT NULL,  
email VARCHAR (255) NOT NULL UNIQUE,  
phone VARCHAR (20),  
active BOOLEAN NOT NULL,  
store_id INT NOT NULL,  
manager INT,  
FOREIGN KEY (store_id)  
REFERENCES sales.stores (store_id)
```



When you expect on/off usage patterns

CockroachDB Serverless specifically can be a great choice for any application that's likely to generate on/off usage patterns — periods of dormancy interspersed with periods of higher activity levels.

There is an important caveat here, though: this isn't true of all serverless databases. When a dormant database has to be spun back up, latency caused by that “cold start” can lead to a poor user experience.

However, CockroachDB Serverless specifically avoids the cold start problem with a [unique architecture](#) that allows a dormant database to consume zero compute resources but still be available instantly when a request comes in.

This makes it an optimal choice for any application that's likely to see on/off usage patterns, since it ensures you're not consuming (or paying for) any compute resources when they're not needed.

When you want to future-proof your application

Small companies and startups may not need features like automated scale and resiliency in the near-term. Many companies can and do get by with “old school” setups like single-instance Postgres or MySQL databases. But just because you can doesn’t necessarily mean you should.

If you hope to scale in the long term, there’s really no reason to put off the process of migrating to a cloud-native, elastic scaling serverless database. As your application grows in complexity, the process of migrating – or trying to retrofit cloud and scaling features into a database that wasn’t really built for either of those things – only becomes more painful. It’s much easier to start from the beginning with a database that can scale as your company grows, even if you don’t anticipate much need to scale in the first few years.

It’s also worth mentioning that choosing a next-generation cloud database such as CockroachDB Serverless can be helpful in attracting talented engineers. The best engineers want to work with new, exciting technologies. Giving devs a chance to work with a serverless, distributed SQL database such as CockroachDB can set your job postings apart from the crowd of MySQL, Postgres, and Oracle options.



When NOT To Use a Serverless Database

Serverless databases are great, but they aren't always the right solution. Here are some circumstances where choosing a serverless database might be a mistake.

When you need control over the hardware

A serverless database is a managed solution, meaning that you don't have to worry about details like precisely how your data is stored, or how the automated scaling works. For many developers, that simplicity is a selling point, as it allows them to focus on building their applications.

But it does mean that you don't have complete control over some of the details, and since it's cloud-based, you don't have control over the hardware. This can rule out serverless databases as an option for some companies, who may need an on-prem solution where they can control the hardware (for security or regulatory reasons, for example).

When you need a deep featureset

Serverless databases are amazing, but they're also relatively new. That means that serverless databases right now tend to come with fewer features than their non-serverless counterparts. In the current beta, for example, CockroachDB Serverless doesn't currently support some of the multi-region features found elsewhere in CockroachDB.

Of course, this is only a temporary disadvantage. Pretty soon, serverless databases will reach the level of maturity and feature depth that's available in non-serverless offerings. But if you need a database with multi-region capabilities right now, you may be better served by a non-serverless option such as CockroachDB Dedicated.

When security concerns rule out multi-tenancy

CockroachDB Serverless and other multi-tenant serverless databases certainly include security measures that isolate tenants' activities and their data from one another. But at the end of the day, tenants are still sharing the same machine, which means that a serverless database usually isn't the best choice for workloads that require a high level of security.

For those high-security workloads, a dedicated solution such as CockroachDB Dedicated is a better choice, providing many of the same advantages available in a serverless database.

When another option offers better performance or costs less

While serverless databases are the best choice for many use cases, there's no "perfect" database solution that's optimal for every possible use case/workload. There are cases where a solution like CockroachDB Dedicated or CockroachDB Self-Hosted is going to be preferable to CockroachDB Serverless.

The Future of Serverless Architecture

Serverless is a crucially better way to consume anything. It allows you to shift complex operational responsibilities like server or cluster provisioning, patching, system maintenance, and capacity management to your public cloud provider. (Or cloud providers, plural, because serverless can also remove the complexity from multi-cloud and hybrid deployments).

But, up until now, businesses have mainly focused on the execution side of serverless. Things like AWS Lambda or Google Cloud Run or Fargate, all products allowing you to just put your application logic in the cloud and let your cloud provider run it for you and scale it for you. Everyone understands that cloud infrastructure is almost universally the superior option as it offers seemingly endless scale for applications without the headache of managing enterprise infrastructure.

Serverless databases are the next step in the shift towards serverless. Solutions including [Fauna](#), [Mongo](#) and [CockroachDB Serverless](#) unlock limitless data and the infrastructure to finally use it right. As a result, we are on the cusp of a serverless database takeover — of crossing the chasm — as more and more enterprise companies realize that, though the rest of their tech stack may be cloud native, their database has been holding them back.



Cockroach
Labs

cockroachlabs.com