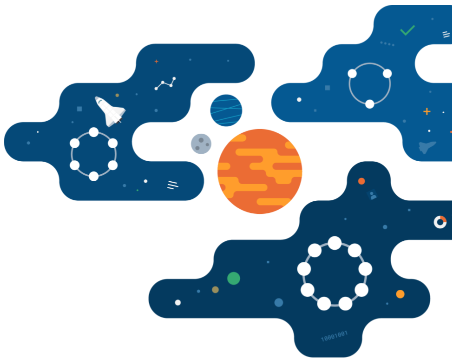
WHITEPAPER



DataStax Astra DB

Designing a Serverless Cloud-Native Database-as-a-Service Based on Apache Cassandra™

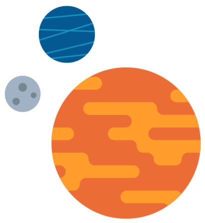
Astra DB is a globally distributed, serverless, multi-model database service built by DataStax to satisfy the needs of users on their cloud provider of choice. It is the first and only serverless and multi-region database service that is based on an open-source NoSQL database, namely Apache Cassandra.

In this paper, we share our experience, rationale and lessons learned of adapting Cassandra into a multi-tenant database to serve the serverless needs of Astra users. We present a novel microservices-based, cloud-native architecture that integrates natively with Kubernetes to bring true, safe, stateful workloads to the cloud-native age. This design enables fine-grained, elastic scalability of individual components to meet the capacity demands of modern application workloads. In this work, our main contributions are: (i) novel microservices-based, cloud-native architecture; (ii) multi-tenancy and fault isolation approach; and (iii) smart auto scaling design.

DataStax

01

Introduction



Apache Cassandra is a leading open-source NoSQL database that is known for its ability to scale to meet the demands of the Internet's most massive and demanding workloads while maintaining the highest degrees of availability and reliability. In the past decade, Cassandra has proven its ability to successfully manage data workloads for some of the most mission-critical, high-throughput, global-scale, always-on, real-time applications. More recently, the Cassandra community has been working on making Cassandra the top-choice database for modern cloud-native applications.

At the core of Cassandra's success is its shared-nothing, leaderless, distributed architecture. As shown in Figure 1.1, all Cassandra nodes are self-sufficient peers that can do every operation: request coordination, read and write paths, metadata, data and commit log management, compaction, and so forth. This makes Cassandra extremely fault-tolerant and straightforward to scale. Since a Cassandra cluster has no single point of failure, losing one or more Cassandra nodes is not a problem as long as a data replication factor is large enough. Since all Cassandra nodes in the cluster are equal, storing more data or supporting higher throughput is resolved by simply adding more Cassandra nodes to scale horizontally.

However, in the new era of cloud database services, serverless computing and microservices, Cassandra's monolithic architecture presents some of the biggest challenges. Each Cassandra node is a single process with subsystems that cannot be scaled independently; this also precludes scaling compute independently from storage. This scale-all-or-nothing approach means that scaling reads and writes independently from compaction is not an option, and scaling down presents a unique set of challenges to operators. The lack of elasticity, combined with the lack of multi-tenancy and auto scaling support, limits Cassandra's ability to efficiently utilize resources, which is crucial in modern cloud-native environments.

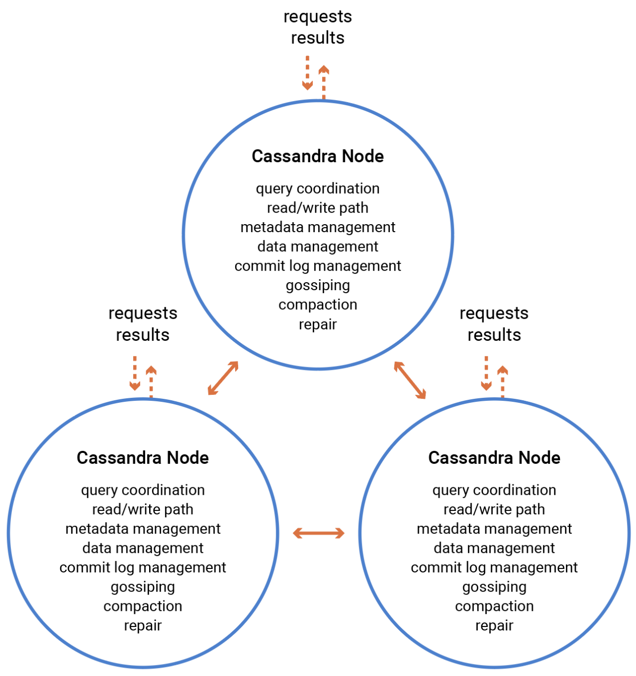


Figure 1.1 Cassandra Shared-Nothing, Monolithic Architecture

Astra DB overcomes the described challenges by breaking Cassandra's monolith into microservices that can be scaled independently from each other. Astra DB is designed to be the first and only serverless Cassandra-based database service that works in multiple clouds, namely AWS, GCP and Azure. In this paper, we share our serverless Cassandra design considerations and present the following contributions:

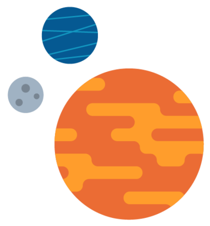
- Novel microservices-based, cloud-native architecture;
- Multi-tenancy and fault isolation approach;
- Smart auto scaling design.

As a serverless database, Astra DB completely eliminates the need for a user to manage physical infrastructure or perform database maintenance operations. Moreover, the serverless design, microservices architecture, multi-tenancy, elasticity and auto scaling enable Astra DB to use and share resources very efficiently.

The rest of the paper is organized as follows. Section 2 describes our serverless design goals. Section 3 presents the new microservices architecture for the serverless Cassandra. Section 4 elaborates on our approach to multi-tenancy and fault isolation. Section 5 covers our smart auto scaling solution. Section 6 explains the improvements related to Cassandra's gossip, compaction, repair and bootstrapping. Finally, Section 7 concludes the paper.

02

Serverless Design Goals



Our foundational principles for Astra DB state that Astra DB must retain and enhance the best characteristics of Cassandra, including fast response time, linear scalability, and high availability. Astra DB also has to be 100% compatible with existing Cassandra workloads so that users could easily migrate existing workloads to the new platform. With these foundational principles in mind, we made a decision early on to not reinvent Cassandra, but rather adapt the original battle-tested codebase much like any other successful journey from a monolithic architecture to a microservices-based architecture. In the following, we set out to define the primary set of operational requirements that would guide our technical decision making through the design and development process.

Cloud-native and multi-cloud design

The system must be designed from day one for reliability and easy automation. Regardless of cloud provider, infrastructure and the database component services should be straightforward to provision, operate, upgrade, and elastically scale up or down. To work in multiple clouds, the system should rely on managed services, such as Kubernetes, that are supported by all major cloud providers.

Fine-grained scalability

The system must be able to scale compute to accommodate continuously changing throughput requirements, while storage must scale with the data size. Moreover, different component services must scale independently of one another to meet the immediate demands on the system and allow for scaling of reads and writes independent of each other to accommodate read-heavy, write-heavy or hybrid workloads.

Complete multi-tenancy

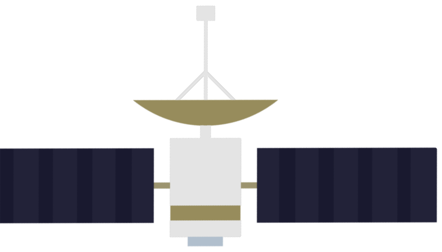
The system must be able to support thousands of tenant databases on each cluster and allow for movement of those databases between clusters to make the most efficient use of resources. It is essential that it does this while maintaining data, performance and fault isolation among tenants. It must guarantee data privacy and ensure that fluctuating workloads and high demands of one tenant do not have any substantial consequences for its neighboring tenants.

Elasticity and auto scaling

The system must be able to dynamically and automatically provision, de-provision and re-distribute resources in order to meet the changing demands of users in the most performant and efficient means possible. To achieve this, AIOps must be utilized to monitor, collect and analyze metrics for every tenant workload, and make intelligent decisions to automate the scaling of individual compute and storage resources. With AIOps, tasks such as capacity planning, over-provisioning and under-provisioning should become obsolete.

Cost effectiveness

The system must be cost-efficient when compared to Cassandra deployments and other Database-as-a-Service offerings. One of the downsides of dedicated, distributed database deployments is that they come with a significant cost to operate. Each of the previously described goals contributes to Astra DB's ability to be resource conscious and thus cost-effective to operate. In addition, the cost of operating an always-on database is spread across the thousands of tenants, driving that cost to as close to zero as possible.



03

Microservices Architecture

With respect to the goals of cloud-native, multi-cloud design and fine-grained scalability, we focus on the following architectural solutions:

- Transforming Cassandra's monolithic architecture into Astra DB's microservices architecture;
- Separating compute and storage in Astra DB;
- Converting Cassandra's subsystems into Astra DB's services;
- Running Astra DB on top of Kubernetes;
- Using S3-compatible object storage services in AWS, GCP and Azure for data storage.

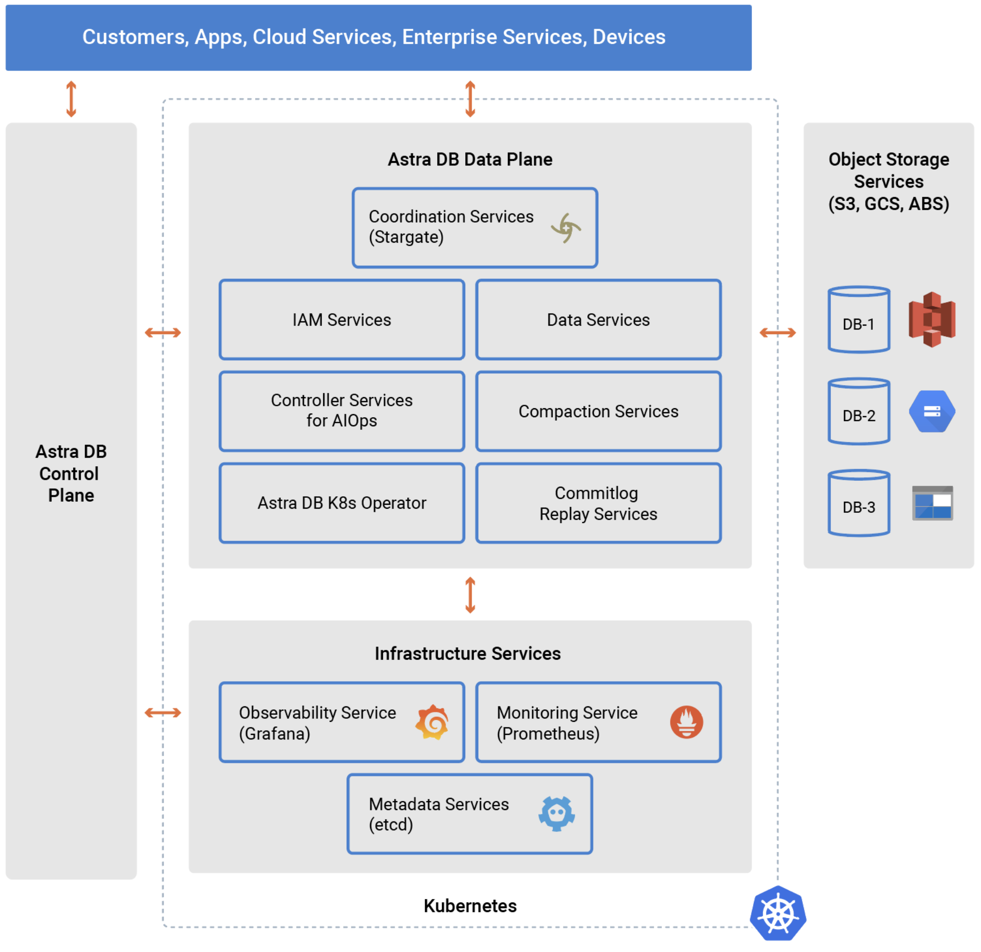

Customers, Apps, Cloud Services, Enterprise Services, Devices
Astra DB Control Plane
Astra DB Data Plane
Coordination Services (Stargate)
IAM Services
Data Services
Controller Services for AIOps
Compaction Services
Astra DB K8s Operator
Commitlog Replay Services
Object Storage Services (S3, GCS, ABS)
DB-1
DB-2
DB-3
Infrastructure Services
Observability Service (Grafana)
Monitoring Service (Prometheus)
Metadata Services (etcd)
Kubernetes

Figure 3.1 Astra DB Microservices Architecture

The high-level architecture of Astra DB is shown in Figure 3.1. Astra DB is composed of many smaller independent services that belong to the control plane, data plane, infrastructure services, and object storage services. ***Kubernetes*** is used to scale and orchestrate all services in the data plane, as well as infrastructure services.

The control plane is responsible for maintaining and configuring the data plane based on customer-specified settings and operational status reported by the data plane. The control plane services are beyond the scope of this paper.

Data plane services are responsible for executing user and application data requests, as well as performing required database maintenance operations in the background. The data plane includes these types of services:

- **Coordination Service** is used for query coordination and data APIs. This service represents the open-source ***Stargate*** data gateway and supports REST, GraphQL, Document and CQL APIs to interact with the database.
- **Data Service** is responsible for reading, writing and managing a subset of data as defined by its token range assignment based on the Cassandra ring. This service manages Cassandra's in-memory data structures like Bloom filters and MemTables. It also relies upon fast, local NVMe volumes for caching data, commit log and partition index files. A local commit log is always synchronized with a commit log managed by the object storage services. Similarly, MemTables are flushed to become SSTables managed by the object storage services.
- **Compaction Service** is utilized for compacting SSTables managed by the object storage services. Similarly to Cassandra's compaction process, this service merges multiple SSTables into a smaller SSTable to optimize reads. Unlike Cassandra's compaction process, this service is shared across tenants. It uses a unified compaction strategy and can also repair any inconsistent copies of data that may have been written by different data services.
- **Commitlog Replayer Service** is used for replaying a commit log managed by the object storage services in case when a data service fails. This service replays a commit log and writes a new SSTable managed by the object storage services.
- **IAM Service** is responsible for identity and access management in Astra DB. This service controls authentication and authorization based on predefined roles and permissions defined in the control plane.
- **Controller Service for AIOps** is designed to use analytics and machine learning to make intelligent decisions about operating Astra DB. This service is capable of analyzing both real-time and historical operational data to react to immediate issues and predict what may happen in the future. This is where decisions to scale a service up or down are made.

- **Astra DB K8s Operator** is created to help run and operate various Astra DB services using the Kubernetes operator paradigm. This deterministic service can start and stop other services, scale them up or down to maintain a proper state of the system.

The infrastructure services include additional services that are based on existing open-source systems:

- **Monitoring Service** and **Observability Service** are responsible for collecting and visualizing metrics about Astra DB operations. These services are based on Prometheus and Grafana, respectively.
- **Metadata Service** is used for consistent management of database schema and cluster topology information. This service replaces Cassandra's gossiping for peer discovery and state information propagation. It is based on the ***etcd*** key-value store.

Last but not least, **Object Storage Services** rely on fully managed services to store data that Cassandra normally persists on disk, including commit logs, SSTables, indexes and other files. Depending on a cloud selection for deploying a database, this service can be represented by Amazon Simple Storage Service (S3), Google Cloud Storage (GCS), Azure Blob Storage (ABS), or any other cloud storage with an S3-compatible API.

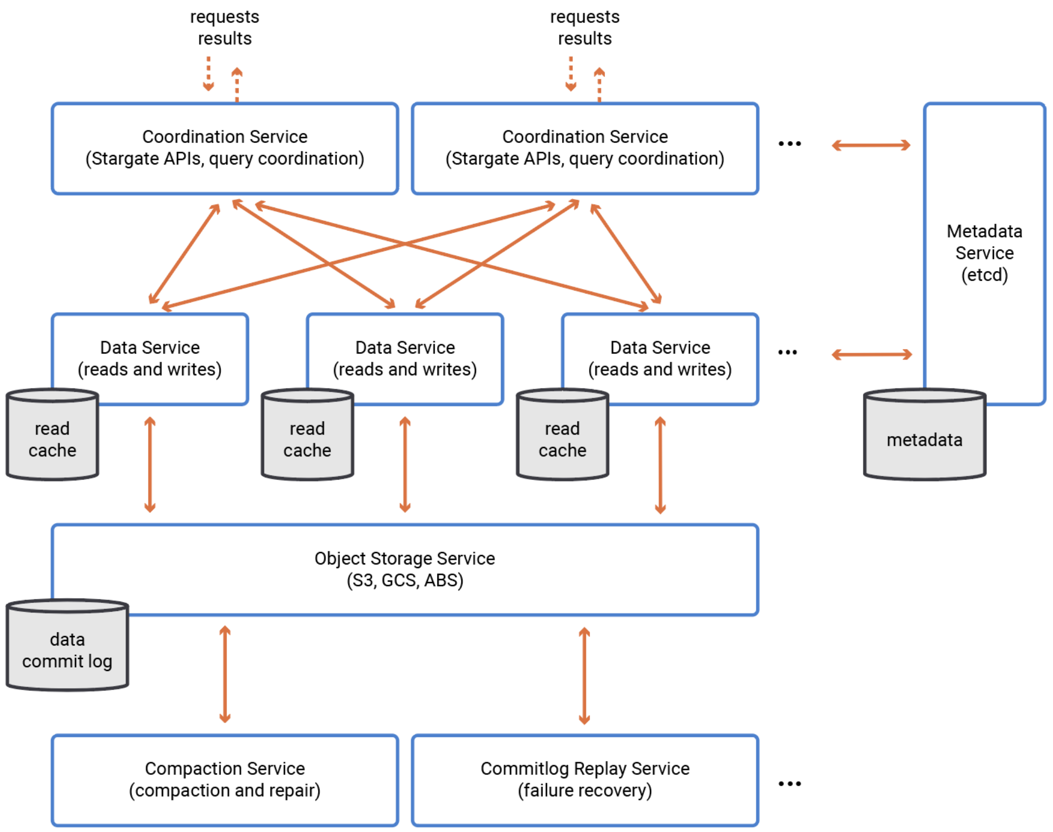


Figure 3.2 Services Interaction in Astra DB

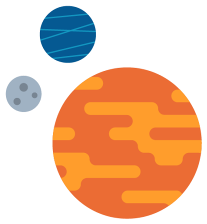
To better understand how Astra DB services interact with each other and scale horizontally, consider the diagram in Figure 3.2. First, a read or write request goes to a coordination service that forwards it to one or more data services. Specific data services are selected based on data partition keys that map to token ranges assigned to these data services. In addition, the number of data services that get a request depends on the data replication factor and request consistency level. For example, with the datacenter replication factor of 3 and read request consistency level of LOCAL_QUORUM, a read request has to be forwarded to two data services that have copies of the data. Second, in case of a read request, a data service can either process the request locally from the data cache or retrieve data from an object storage service like S3. In case of a write request, each participating data service stores data both locally in a commit log and MemTables and remotely in a commit log and SSTables via the object storage service. In particular, the active local commit log segment is synched to the object storage every 10 seconds, where it can be used by the commitlog replay service for recovery purposes. During the flush operation, a MemTable is written to the local disk as an SSTable, a copy of the SSTable is sent to the object storage service, and the new SSTable range and location are reported to the metadata service. Third, both coordination and data services interact with a metadata service to report and retrieve up-to-date information about any schema and topology changes. Fourth, while running in background independently of requests, a compaction service compacts and simultaneously reconciles any inconsistencies between data replicas stored by an object storage service. Finally, the commitlog replay service is triggered when a data service fails to replay its commit log stored in an object storage service and replace it with a new SSTable.

All the services in Figure 3.2 can scale horizontally. For example, when there is a temporary spike in the number of requests, Astra DB can provision additional coordination and data services to increase the database service throughput. Moreover, since compute and storage are decoupled, the object store services can independently increase or decrease storage capacity to store more or less data on demand. Once the number of requests decreases, Astra DB can drain and deprovision some service instances to utilize less resources. In the background, compaction and commitlog replay services may not be affected at all by this temporary spike. This kind of fine-grained scalability is possible due to Astra DB's new microservices architecture. This is a major improvement over the traditional distributed database architecture, where scaling is achieved by adding and removing physical nodes and storage.

Astra DB's new microservices architecture is also designed to be cloud-native and multi-cloud. Running on top of Kubernetes and supporting data storage via fully managed object storage services in AWS, GCP and Azure enable Astra DB to operate in any of the three public clouds.

04

Multi-Tenancy and Fault Isolation



Designing a multi-tenant system requires finding a fine balance between cost effectiveness and maintaining performance guarantees. On one end of this spectrum is running a separate Cassandra cluster for every customer. This approach will perform well but comes at a high cost. The other extreme is running one giant cluster serving all customers. This approach is cost-effective but can have many negative performance implications. An ideal solution lies somewhere in between.

Another important consideration for a multi-tenant system is performance and fault isolation. A tenant that experiences a sudden increase in traffic, software bug or denial-of-service attack should not have any substantial effect on other tenants that share the same underlying infrastructure. Even if all resources assigned to a tenant are taken down by a malicious attack, all the other tenants should continue to operate normally.

With respect to the multi-tenancy goal, we make the following design decisions:

- Supporting thousands of tenants per Kubernetes cluster;
- Enabling safe and secure sharing of different services by multiple tenants;
- Using shuffle sharding for performance and fault isolation;
- Storing each tenant data in its own private bucket in S3, GCS, ABS, or other cloud storage with an S3-compatible API.

In Astra DB, there can be multiple Kubernetes clusters, each serving thousands of tenants. Each tenant is assigned to a single, shared-tenancy Kubernetes cluster, a number of services depending on workload requirements, and one S3, GCS or ABS bucket. This is illustrated in Figure 4.1. In this example, we have four tenants denoted as A, B, C, and D assigned to the same Kubernetes cluster. For simplicity, we assign two coordination services and three data services per tenant. These numbers can change dynamically as each tenant scales up or down. While tenants A, B and C share various services among each other, tenant D takes advantage of dedicated services. This could be because tenant D runs a higher-throughput workload than other tenants. And this can also change over time: tenant D may start sharing services with other tenants. A multi-tenant coordination or data service is responsible for one or more Cassandra keyspaces and sets of tokens per tenant. A Cassandra token ring for each tenant database is completely independent from any other tenant token ring, which enables scaling tenants and keyspaces independently by assigning more or fewer services to a specific tenant. Next, our tenants share the same metadata service with each tenant metadata being stored separately. Finally, each tenant has a separate bucket in the object storage service, where all tenant keyspaces, tables and data reside.

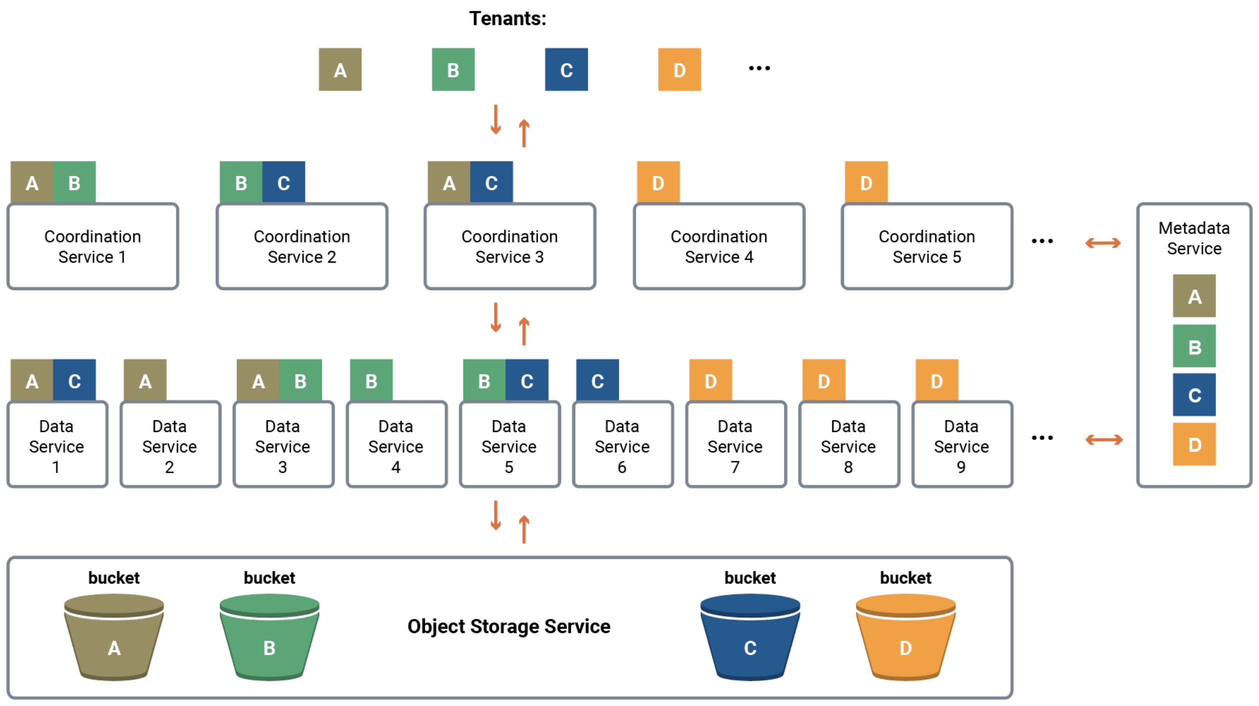


Figure 4.1 Multi-Tenancy in Astra DB

To assign tenants to different services, Astra DB uses a ***shuffle sharding*** algorithm that in practice utilizes the following pattern. Consider the architecture shown in Figure 4.2. We have four tenants and nine data services, and each tenant gets assigned to three distinct data services that are picked from the pool of available services. Such an assignment forms a virtual shuffle shard for each tenant. With nine services, we can have 84 different three-service virtual shuffle shards.

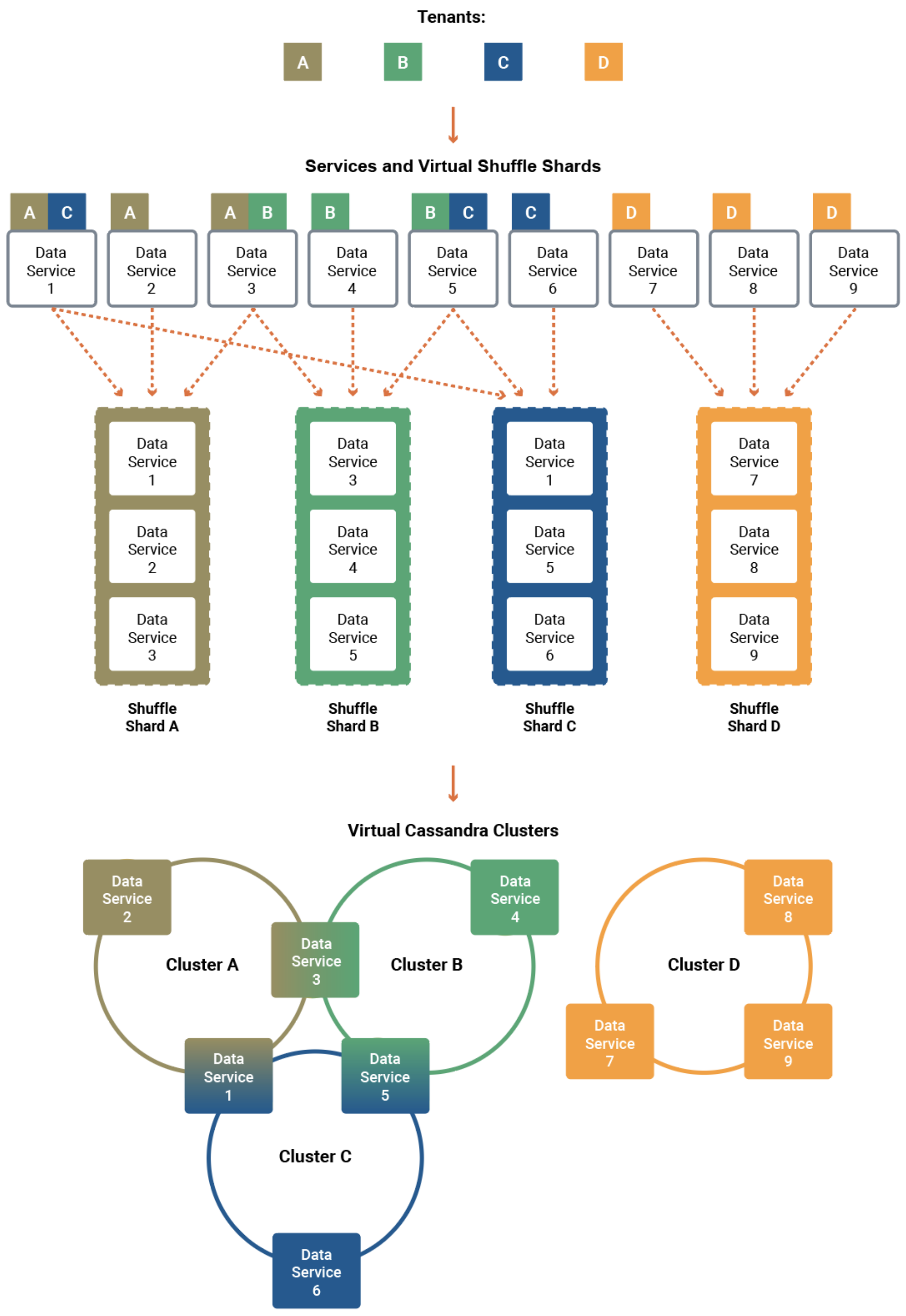


Figure 4.2 Shuffle Sharding for Fault Isolation

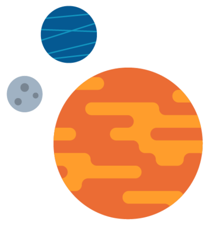
Since shuffle shards may overlap, such as one or more services may belong to two different shuffle shards, we must introduce additional guarantees about overlapping. In particular, for any two shuffle shards, we want the overlap to be at most one data service. In our example, we have shuffle shards A, B, C, and D. There are one-service overlaps between A and B, A and C, and B and C. Shuffle shard D does not overlap with any other shard. Since we have data services in this example, it could be convenient to think of them as Cassandra nodes, even though this would be technically inaccurate. Such Cassandra “nodes” form virtual Cassandra “clusters” as shown in the bottom part of Figure 4.2.

Let’s examine how shuffle sharding helps with tenant performance and fault isolation. For example, in Figure 4.2, tenant B gets a “poisonous” request that takes down data service 3. At this point, tenants A and B are minimally affected because their respective shuffle shards contain data service 3. With additional “poisonous” requests to tenant B, the impact can cascade and take down data services 4 and 5. All data services in shuffle shard B become unresponsive. At this point, while tenant B is no longer able to use the system, the other tenants are well isolated from the attack. Tenants A and C should continue to operate normally, even though they may be minimally affected by unresponsive data services 3 and 5 that they share with tenant B.

It is worth noting that Astra DB runs multiple Kubernetes clusters with thousands of tenants in each. Each cluster can have many thousands of services that can support billions or even trillions of possible virtual shuffle shards, taking performance and fault isolation to a whole new level. Different clusters can have distinct multi-tenancy policies to accommodate customers with different requirements. As customers go from development to production to high-throughput workload, they may be automatically moved between clusters that can better serve their needs.

05

Smart Auto Scaling



With the microservices architecture and multi-tenancy in place, the final missing piece of the serverless puzzle is auto scaling. Auto scaling is a hard optimization problem that looks to minimize the total cost of computational resources while meeting the continuously changing demand of every tenant. This is definitely about scaling up the services when demand is high. But scaling back down is as important for cost effectiveness when demand decreases. Auto scaling enables database service elasticity, which boils down to how many operations per second a tenant currently needs.

There are numerous considerations and dimensions that Astra DB has to take into account for auto scaling purposes:

- Different component services have unique characteristics and requirements for healthy operation.
- Horizontal scaling is always the primary scaling mechanism. In addition, different services can have different default operations-per-second rate limits that can be dynamically tuned for each tenant.
- Some workloads may seem fluctuating and unpredictable. Others may exhibit well-defined periodic patterns that can help predict the future demand.
- On one hand, there is scaling of services for individual tenants. On the other hand, the system has to scale its total operational capacity across all tenants.
- Different scaling measures are required under abnormal circumstances, such as huge demand spikes or large-scale cloud outages.
- Both decisions to scale and scaling actions should be very fast to support real-time applications.

With respect to our serverless design goal of elasticity and auto scaling, we develop a suite of auto scaling policies, strategies, heuristics, and predictive models. Since our solutions evolve frequently, we illustrate auto scaling in Astra DB using a concrete example.

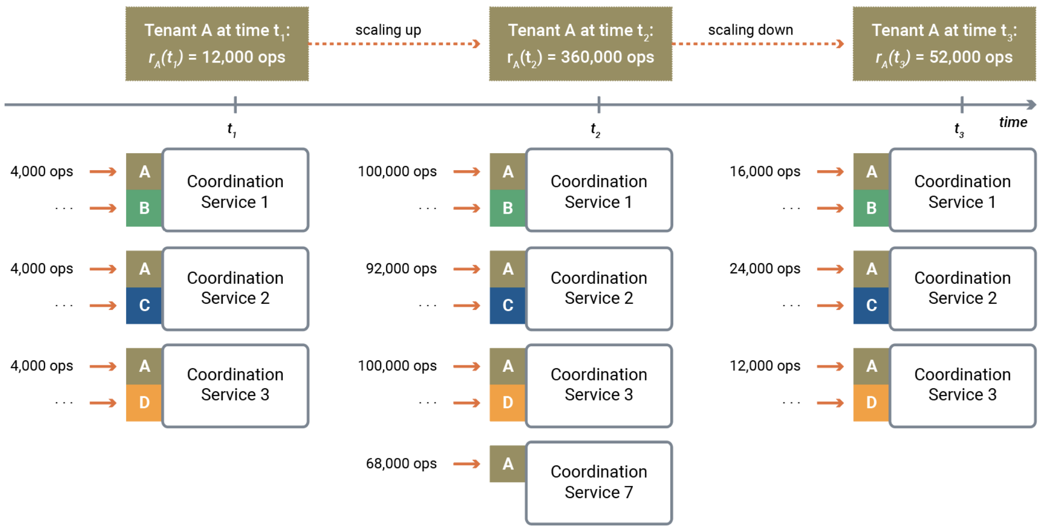


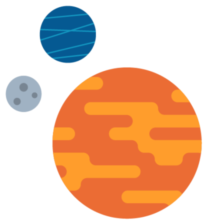
Figure 5.1 Auto Scaling in Astra DB

Figure 5.1 shows how Astra DB scales up and down coordination services for tenant A based on a varying demand. At time t_1, tenant A is assigned three coordination services with the default rate limit of 4,000 operations per second (ops) per coordination service. In other words, across the three coordination services, tenant A can perform up to 12,000 ops, which is shown as $r_A(t_1)$ in the figure. As the demand gradually increases past 12,000 ops and close to $r_A(t_2)$ = 360,000 ops at time t_2, two things happen automatically. First, we gradually increase rate limits to up to 100,000 ops for coordination services that are already assigned to the tenant. Second, since three coordination services cannot handle more than 300,000 ops, we assign an additional coordination service to meet the demand of this tenant. Finally, once the demand decreases to under $r_A(t_3)$ = 52,000 ops at time t_3, we decrease both the rate limits and number of coordination services to save resources.

While the above example is specific to coordination services, it illustrates the basic idea of auto scaling in Astra DB. The exact numbers may be different for other types of services and particular scenarios, but the idea remains the same. Astra DB initially assigns some default rate limits per service per tenant. It then dynamically adjusts those limits up to some predefined maximum allowed values and automatically adds or removes services according to the demand. The demand is constantly collected and measured by the monitoring service and the controller service for AIOps. Auto scaling decisions are made and applied by the controller service for AIOps and the Astra DB K8s operator service.

06

Individual Service Improvements



The new microservices-based, cloud-native architecture creates more opportunities to experiment and innovate. In this section, we describe our Astra DB improvements related to Cassandra's gossip, compaction, repair and bootstrapping.

Metadata services replace gossiping and take over schema management

In Astra DB, metadata services use the strongly consistent, distributed key-value store present in all Kubernetes clusters called ***etcd*** to manage all necessary information about services topology and database schemas for each tenant. As a result, coordination, data and other services are immediately and consistently made aware of changes to the underlying metadata via pushes from the metadata service. This is an important improvement over Cassandra's eventually consistent gossiping mechanism for metadata dissemination. Eventually consistent schema updates in Cassandra become strongly consistent schema updates in Astra DB. Furthermore, the separation of metadata management from compute and data management contributes to Astra DB's scalability and cost effectiveness goals.

Compaction services use a unified compaction strategy

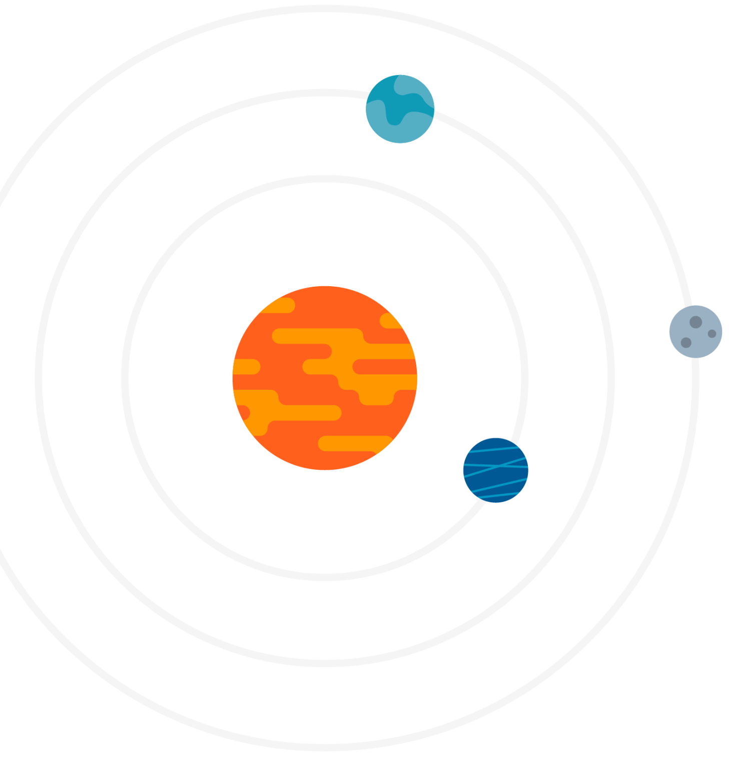
In Astra DB, compaction services use the unified compaction strategy (UCS)—a hybrid of Cassandra's size tiered compaction strategy (STCS), levelled compaction strategy (LCS) and time window compaction strategy (TWCS). In a nutshell, the new strategy partitions SSTables into multiple compaction shards to increase parallelism and decrease individual compaction times. Sharding is automatically configurable for different scenarios, including using TWCS to shard time series workloads. When defined as the property of a level in an LSM tree, the new strategy uses STCS on the lower levels and LCS on the higher levels.

Compaction services take on repair

In Astra DB, compaction services compact data in SSTables and simultaneously repair any inconsistencies among replicas. This becomes possible because a compaction service has access to every-replica data files via an object storage service. It makes sense to do a repair during compaction because the cost of added repair is very low in this scenario. In other words, repairing while compacting is significantly faster than running compaction and repair separately. It is also worth noting that a compaction service can scale up and down vertically to accommodate larger and smaller jobs efficiently.

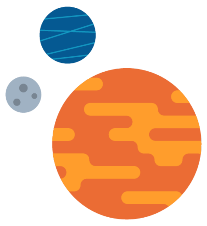
Object storage services simplify bootstrapping

In Astra DB, object storage services are responsible for data storage, which is separate from compute. When a new tenant is assigned to a data service, there is no need to stream replica data for the tokens that the service may now be responsible for. Instead, data still resides in the object storage and does not need to be moved. This is analogous to creating a new pointer to an existing data set rather than copying the whole data set to create a new one. The whole process is very efficient, which makes scaling up and down to be fast and inexpensive. It is good to know that data services do have read caches that get gradually hydrated to speed up read requests.



07

Conclusions



Astra DB launched Apache Cassandra™ into the space of cloud database services, serverless computing and microservices. To transform Cassandra into Astra DB, we designed a new microservices-based, cloud-native architecture, introduced a multi-tenancy and fault isolation approach, and implemented a smart auto scaling solution. As a result, DataStax Astra DB became the first serverless and multi-cloud database built on Cassandra. Astra DB runs on AWS, GCP and Azure, safely supports thousands of tenants, and scales automatically with capacity demands. Ultimately, Astra DB is an easier and more cost-efficient way to use Cassandra.