WHITE PAPER

commercetools' Resiliency

January 2026

What's inside

Summary

This paper outlines how commercetools ensures resiliency across availability, scalability and data recovery. Our platform has powered some of the world's busiest commerce events — Black Friday, Cyber Monday and major global launches — without disruption. Outages remain exceptionally rare.

commercetools is a cloud-native commerce platform designed with a multi-tenant architecture, versionless APIs, and an event-driven microservices foundation. It is engineered to withstand failures, adapt to demand in real time and scale seamlessly to billions of API calls per day while meeting strict SLAs.

Resiliency today goes beyond uptime. It means being secure by design, AI-ready, observable and compliant with global regulations. This document explains the architecture behind commercetools, our approach to redundancy, backup and recovery, customer support and how we continue to strengthen resiliency in the age of agentic commerce.

Basic architecture

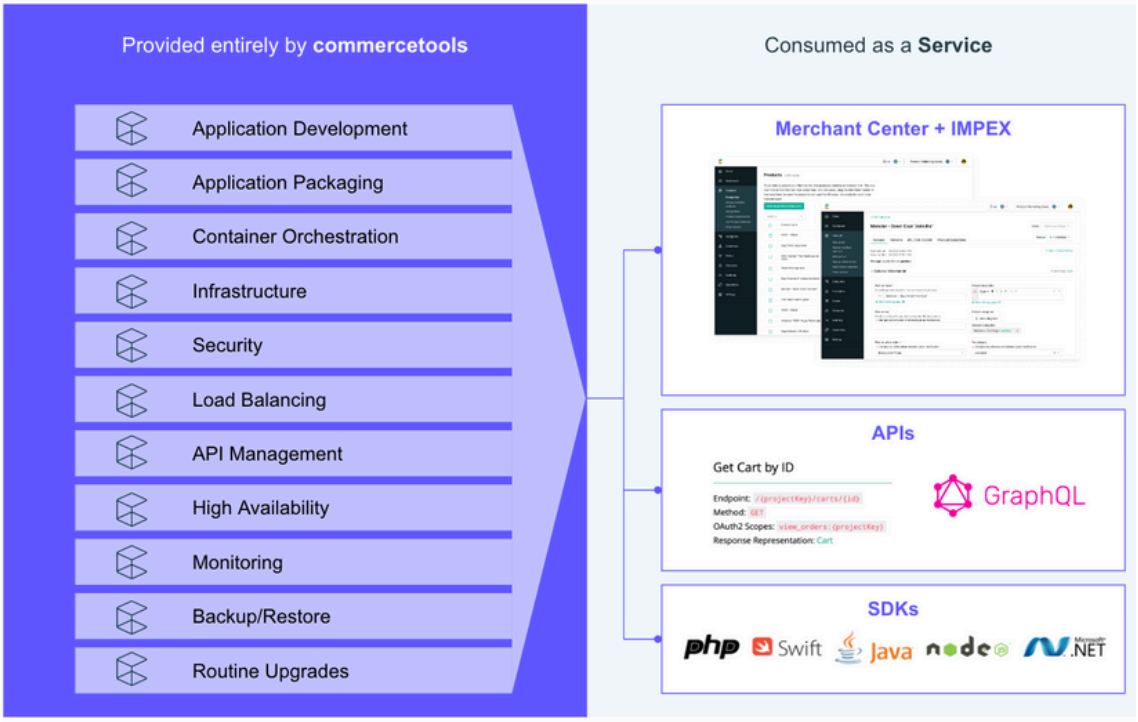
commercetools is built on a multi-tenant, cloud-native, microservices-based architecture designed to ensure high availability, elasticity and fault tolerance at a global scale. This architecture represents the evolution from monolithic and single-tenant designs to an event-driven, versionless API-first platform that's resilient by default.

By embracing cloud-native design principles — container orchestration, service mesh, distributed databases and immutable infrastructure — we deliver a platform that adapts dynamically to demand while isolating failures to minimize customer impact.

Cloud services

commercetools runs on leading hyperscale providers, **Google Cloud Platform (GCP)** and **Amazon Web Services (AWS)**, ensuring access to their most resilient primitives: Multi-zone deployments, distributed storage, global networking,and managed recovery services. This multi-cloud strategy prevents lock-in and allows us to meet regional compliance requirements while offering choice and redundancy.

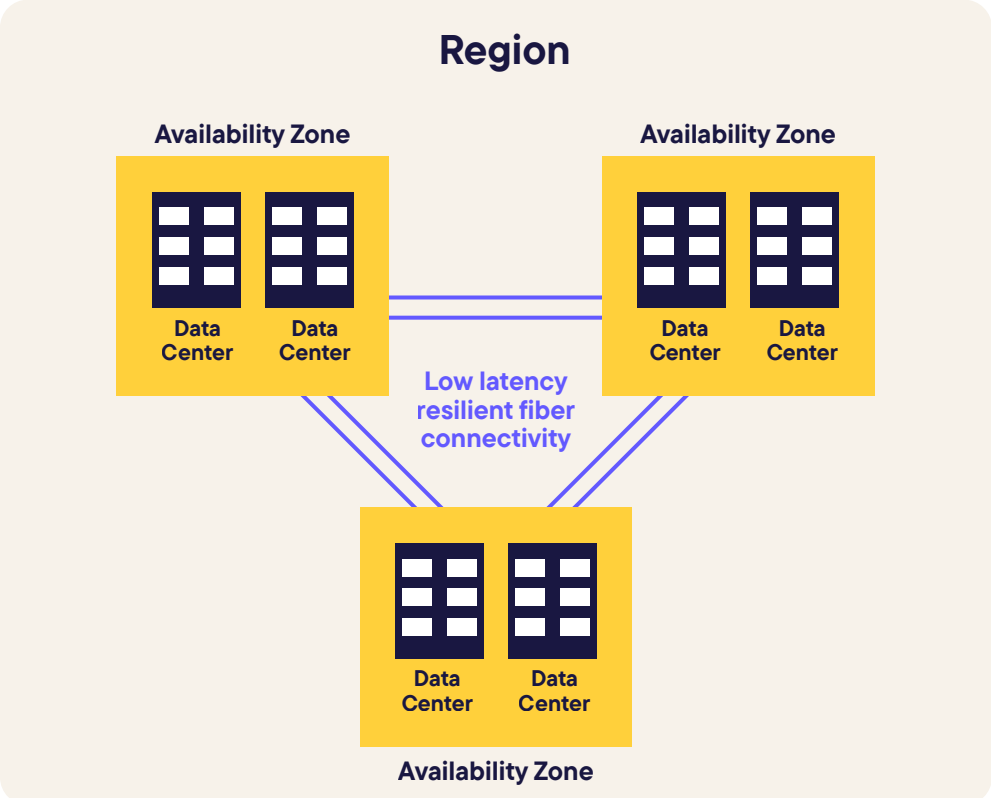
We extend these provider capabilities with our **own service mesh, zero-trust security model** and **observability layer.** The result is a unified commerce platform that takes advantage of the reliability and scale of hyperscalers while incorporating platform-level controls for resiliency, security and performance.



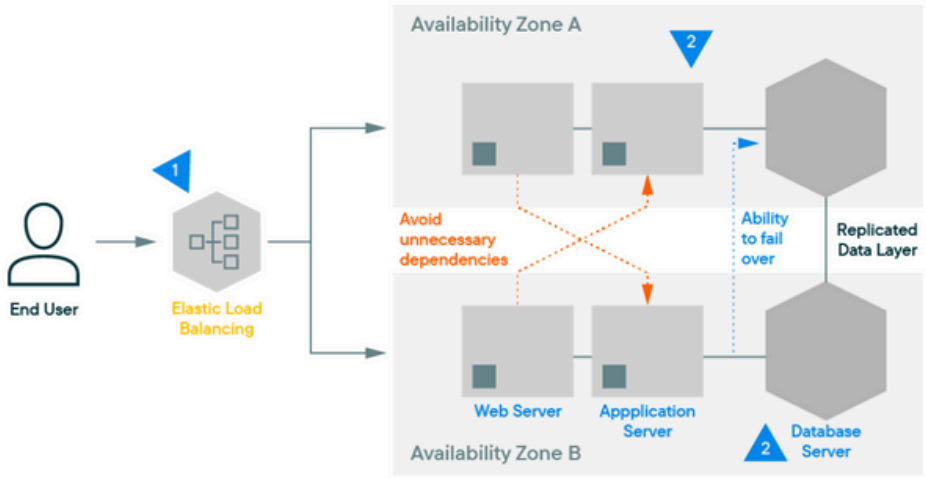
Regions and availability zones

Cloud service architecture enables the distribution of hardware infrastructure across the globe via multiple regions. Our services were designed to be cloud-native, taking advantage of the benefits of cloud service architecture. Each service in our software stack leverages a minimum of three **Availability Zones** within each **Region**. This cloud deployment approach is inherently resilient in protecting against application node failures. Leveraging this ability, hardware failure is not an issue, as it's automatically maintained and redundancies are kept by the cloud provider.

Regions are independent geographic areas that consist of **Availability Zones**. Each Availability Zone is a deployment area for cloud resources and should be considered a single failure domain. These zones have distinct locations with independent network connections and power supplies; however, zones within a region are strategically located to ensure round-trip network latencies of under 1ms in the 95th percentile. Through Availability Zones, cloud deployments achieve fully active-active availability within a region. Below is a visual example of how Availability Zones are utilized within a Region.



Being in the cloud doesn't leverage Availability Zones by default. Customers can choose to deploy an on-premise application into a single zone in the cloud. If the application is not architected/designed to be cloud-native, customers are sometimes locked into this type of deployment model. Deploying an application this way would forgo some of the built-in advantages the cloud can provide. The image below shows how Availability Zones function for redundancy and availability.



Within each specified region, a minimum of three times redundancy exists, ensuring that individual services (applications, databases, etc.) are never interrupted, regardless of hardware failures, system issues, or zone outages.

Each service application, database, or supporting infrastructure is deployed redundantly across zones. For example:

- Databases are replicated across AZs with automated failover.
- Stateless services scale horizontally across Kubernetes clusters running in multiple zones.
- Load balancers distribute traffic across regions and zones, ensuring that there is no single point of failure.

Cloud-native by Design

Being "in the cloud" does not automatically mean resiliency. Legacy applications migrated into a single cloud zone remain vulnerable to outages. commercetools avoids this pitfall by being architected cloud-native from inception:

- All core services are stateless where possible, relying on distributed storage and event streams.
- Stateful components like databases and queues are replicated across AZs with built-in durability guarantees.
- Infrastructure is codified with Infrastructure as Code (IaC) using Terraform, ensuring reproducibility and rapid redeployment.

This design ensures that hardware failures, network disruptions or zone outages never result in service disruptions for customers.

APIs backed by microservices

Every commercetools capability is exposed through an API, powered by its own microservice. These services are deployed across regions using Kubernetes-based orchestration, with dedicated compute, datastore and runtime environments. This design ensures:
Independent scalability: Each service scales horizontally based on its own workload profile.
Rapid delivery cycles: Teams deploy features and fixes on independent cadences without disrupting other services.

Because commercetools APIs are versionless, upgrades and enhancements flow seamlessly into production without breaking changes. Customers benefit from continuous innovation without downtime or costly migrations.

Continuous delivery and observability

Resiliency also depends on how code reaches production. commercetools uses Continuous Integration and Continuous Deployment (CI/CD) pipelines with automation, testing and incremental rollouts. Key practices include:

- Canary releases: Rolling out new functionality to a subset of traffic before global deployment.
- Automated rollback: Failing changes can be reverted in minutes.
- Observability everywhere: Services emit metrics, logs and traces via OpenTelemetry, feeding into real-time dashboards and alerting systems.
- Chaos and resilience testing: Fault injection and recovery drills validate that services degrade gracefully.

This automation ensures that changes enhance the platform without introducing instability.

Database sharding and resilient persistence

commercetools employs database sharding to achieve horizontal scalability. Instead of storing all data in a single system, we partition (“shard”) the data into keyed partitions, distributed across multiple nodes and disks. This approach enables the platform to process commerce data at a global scale, handling workloads that exceed the capacity of any single machine.
Our primary data persistence is delivered via distributed, cloud-native databases with built-in support for:

- Elastic sharding for horizontal scaling as demand grows.
- Replication across at least three Availability Zones per region.

Automated failover if a shard or node becomes unavailable.

Resiliency by design in data layer

By combining sharding and replication, commercetools ensures that:

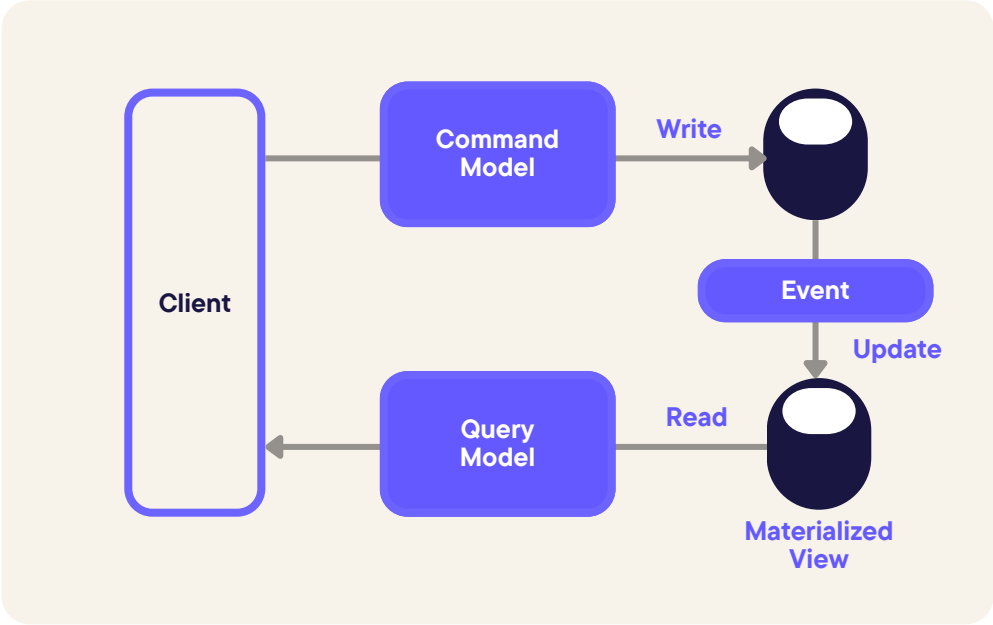
- Write-intensive workloads (e.g., orders, carts) scale independently from read-heavy workloads (e.g., product catalogs).
- Strong consistency is guaranteed for critical transactions, while eventual consistency powers asynchronous operations at scale.
- Immutable backups and snapshots provide rapid recovery options in te event of data corruption or catastrophic failure.

This approach allows commercetools to handle billions of API interactions daily without compromising performance, reliability, or customer experience.

Command Query Responsibility Segregation (CQRS) and data consistency

To achieve both scale and resiliency, commercetools leverages the Command Query Responsibility Segregation (CQRS) pattern, a proven architectural principle in distributed systems. CQRS separates the handling of commands (writes) from queries (reads), ensuring that each workload is optimized independently.

When an update request is made, for example, adding an item to a cart, the command is not written directly to a single database. Instead, the request is transformed into one or more domain events. These events are placed onto an asynchronous event stream, where independent microservices consume and process them in parallel. This approach reduces system load, decouples services and ensures that a localized failure doesn't cascade across the platform.



Resiliency benefits of CQRS

- **Failure isolation:** Write operations (e.g., checkout, orders) and read operations (e.g., product catalog search) are decoupled. Issues in one workload don't degrade the other.
- **Independent scaling:** Write-heavy stores can be tuned for durability, while read-heavy stores can be optimized for performance, caching and low latency.
- **Event-driven recovery:** Because the state is derived from an event log, failed processes can be replayed from the stream, allowing the system to self-heal and catch up without data loss.
- **Asynchronous durability:** Even under sudden spikes, such as flash sales or autonomous agent-driven shopping, commands are queued safely and processed reliably, ensuring no request is dropped.

Consistency models

commercetools supports both strong consistency and eventual consistency, depending on the business use case:

- **Strong consistency (Read-After-Write.)** Critical operations, such as payment confirmation or inventory reservation, require that once a transaction is acknowledged, it's immediately reflected in the datastore. A read after a confirmed write always reflects the latest state.
- **Eventual Consistency.** For operations that can tolerate slight delays, such as product search indexing or asynchronous catalog updates, changes are processed as events on a queue. The API acknowledges the request, but downstream systems may take a short time to reflect the update. Within seconds, the change propagates to all consumers.

This dual approach balances safety (for financial and transactional correctness) with scalability (for high-volume, non-critical updates).

Why this matters for resiliency

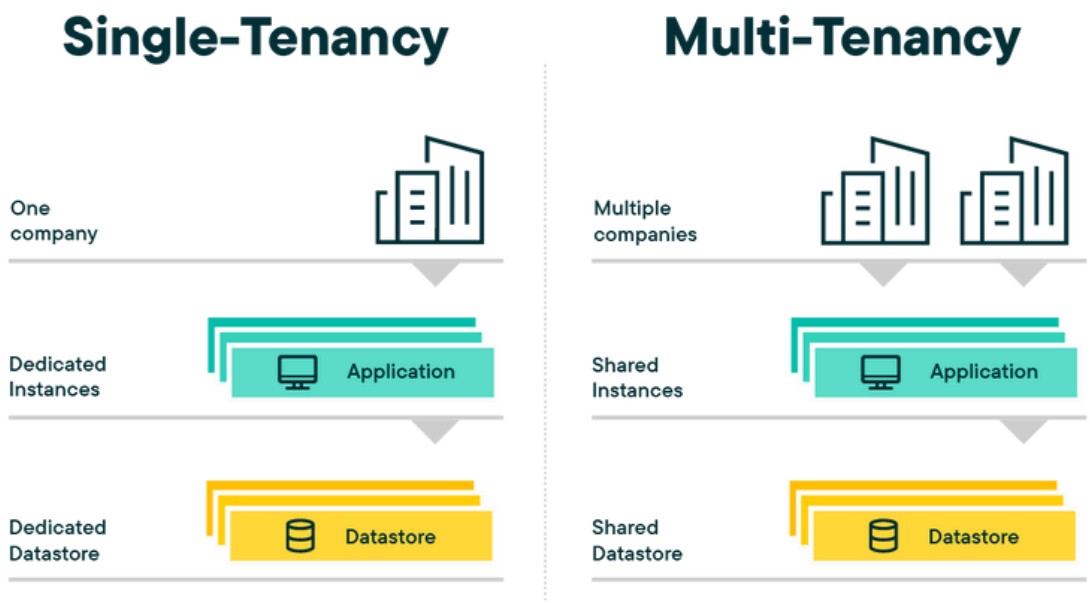
By combining CQRS with multiple consistency models, commercetools can:

- Handle millions of API requests per minute without bottlenecks.
- Deliver sub-second response times for reads even under peak traffic.
- Guarantee correctness where it matters most, while optimizing throughput for non-critical operations.
- Rapidly recover from partial failures by replaying event streams into fresh datastores.

Multi-tenancy and tenant isolation

commercetools is multi-tenant by design, meaning that multiple customers share the same infrastructure and services while maintaining complete logical and security isolation. This model allows us to deliver resiliency at scale, ensuring that every customer benefits from the same performance, updates and security controls.

Each tenant project is provisioned with a unique logical database within our distributed persistence layer. While physical infrastructure (compute, storage, networking) is shared for efficiency, all access is strictly isolated using tenant-specific API credentials, authentication layers and authorization policies. This guarantees that no customer has visibility into another customer's data or workloads.



Resiliency advantages of multi-tenancy

1. **Scalability without fragmentation:** Unlike single-tenant deployments, where each environment must be scaled separately, multi-tenancy allows us to pool resources and automatically distribute them across customers. This enables elastic scaling up to billions of requests per day while maintaining predictable performance.

2. **Continuous improvements for all:** Because all tenants run on a shared platform, every fix, performance optimization and resiliency enhancement benefits all customers immediately. There are no one-off deployments or lagging upgrade cycles.

3. **Noisy neighbor protection:** Misbehaving tenants, such as those generating excessive load, are automatically identified through **real-time monitoring and QoS controls**. Rate limiting and **load shedding mechanisms** reduce one tenant from degrading the experience of others.

4. **Resilient by default:** Multi-tenancy amplifies resiliency by removing brittle, custom deployments. Customers share a hardened, tested, and continuously validated platform instead of isolated, bespoke instances.

Why commercetools avoids one-off deployments

Single-tenant or custom deployments create **operational fragility**: Performance issues, delayed updates and inconsistent resiliency guarantees. By operating a unified, multi-tenant architecture, commercetools ensures:

- Every customer benefits equally from resiliency engineering, chaos testing and incident response drills.
- Global uptime guarantees are enforceable and transparent.
- Platform improvements compound over time, strengthening resiliency for the entire customer base.

Security as a foundation of resiliency

At commercetools, resiliency and security are inseparable. A platform cannot be resilient if it's vulnerable, and it can't be secure if it's fragile. That's why commercetools is engineered with security-by-design principles that are deeply embedded in our cloud-native architecture.

Zero-trust service mesh

All microservices within commercetools communicate through a zero-trust service mesh. Every request between services is authenticated, authorized and encrypted, ensuring that no traffic inside the platform is ever assumed to be trusted by default.

Encryption everywhere

- In transit: All API and service communications are secured with TLS 1.3.
- At rest: Data is encrypted with AES-256 across databases, storage and backups.
- Keys: Managed through cloud-native KMS (Key Management Service) with rotation policies.

Automated compliance and governance

The platform continuously meets and maintains global compliance standards, including SOC 2 Type II, ISO 27001, GDPR and CCPA. Built-in monitoring and reporting systems provide:

- Continuous compliance checks for misconfiguration drift.
- Audit trails for every action through immutable logs.
- Automated enforcement of security controls across regions.

Supply chain and runtime security

- Immutable containers: Services run in signed, verified Docker images.
- Dependency scanning: Every build undergoes automated scanning for CVEs and vulnerabilities.
- Runtime monitoring: Anomalous behaviors are flagged in real time using observability tools and AI-based anomaly detection.

Shared responsibility, minimized risk

Unlike traditional on-premises systems, where customers must secure the full infrastructure stack, commercetools minimizes the customer's burden:

- Customers focus only on securing their integrations and frontend applications.
- commercetools manages infrastructure hardening, service patching and incident response.
- Regular penetration testing and chaos-security drills validate both defenses and recovery procedures.

Resiliency through security

By embedding security into every layer — network, data, application and operations — commercetools reduces the risk of disruptions caused by misconfigurations, breaches or vulnerabilities. Security isn't bolted on; it's the foundation of our resiliency strategy.

Microservices vs. monoliths: Resiliency in practice

Resiliency at commercetools begins with architecture. Unlike monolithic systems, where a single failure can bring down the entire application, our platform is composed of independent microservices. Each service owns its own runtime, datastore and scaling logic, and communicates with others via secure APIs and event streams.
This isolation ensures that when one service fails, others continue operating without disruption. For example, if the Import API experiences a fault, it may temporarily affect data imports, but Orders, Checkout and Search continue to function normally. The impact radius is minimized, protecting the broader commerce experience.

Single Points of Failure (SPOFs)

Even in microservice architectures, certain services are critical for the platform as a whole. Authentication is one such example: All API requests must be validated to protect customers and data. If authentication were unavailable, other services would still technically run, but all requests would fail authorization.

Traffic Overload

Traffic overload is one of the primary resiliency challenges for any multi-tenant SaaS platform. In commerce, demand can surge unpredictably — whether during Black Friday flash sales, viral product drops or increasingly, traffic generated by autonomous shopping agents. A resilient architecture must not only withstand these spikes but do so without degrading the experience for other tenants.

At commercetools, services are intentionally operated well below maximum utilization during normal load, maintaining headroom that acts as the first line of defense against unexpected bursts. Our stateless Kubernetes clusters scale elastically, bringing new nodes online in under two minutes and new pods in seconds. Databases and persistence layers are also configured for horizontal scaling, with built-in replication and auto-scaling to absorb sustained growth.

Caching is used strategically to minimize pressure on core services. Frequently accessed data is cached both internally and at the edge, close to customers, reducing round-trip latency and offloading repeated queries. Write operations, which cannot be cached, are intelligently distributed across shards and zones to prevent hotspots and maintain throughput under heavy load.

For non-critical or latency-tolerant operations, commercetools relies on asynchronous background processing. Tasks are queued rather than executed in-line, allowing core APIs to respond quickly while downstream systems process workloads in batches. This queuing mechanism not only smooths load but also enables automatic retries if a process fails, ensuring durability without blocking the user experience.

Software bug or infrastructure misconfiguration

Even the most resilient systems can be threatened by errors introduced through new code or infrastructure changes. At commercetools, we apply a multi-layered approach to ensure that neither software bugs nor configuration missteps disrupt production environments.

Software bug

Every code change is subjected to rigorous peer review and automated testing before it can progress. Reviews require multiple approvals to ensure independent validation, while test pipelines encompass unit, integration, regression and contract testing across microservices. Security checks are integrated early in the process, utilizing automated scans to identify vulnerabilities and dependency risks. By combining peer oversight with automated assurance, we significantly reduce the likelihood of flawed code reaching production.
Deployment practices are designed to further minimize risk. commercetools follows a continuous integration and continuous deployment (CI/CD) model where small, incremental changes are promoted multiple times per day. Each release passes through staging environments for validation, and higher-risk changes are introduced through progressive delivery techniques, such as canary deployments. In this model, only a small percentage of production traffic is initially exposed to a new change, and the rollout expands only after real-time metrics confirm stability. Should an issue be detected, our pipelines allow us to instantly roll back to a previously validated version, often in a matter of minutes.

Infrastructure misconfiguration

Infrastructure changes are treated with the same discipline as application code. All configurations are defined in version-controlled Infrastructure as Code (IaC) using Terraform. This ensures that environments are reproducible and consistent across regions. If a misconfiguration occurs, we can revert to a previous version with precision. In critical situations, controlled manual interventions are possible, but they are always executed by at least two engineers to prevent human error. Automated drift detection monitors for unintended configuration changes, alerting teams and restoring compliance when deviations occur.

Stateless component failure

A cornerstone of commercetools' resiliency strategy is the principle of statelessness. Services that handle API calls are designed without dependency on local state, meaning they can be terminated, restarted or rescheduled without losing data or disrupting ongoing processes. This stateless approach ensures that failures are transient events, not systemic outages.

All stateless services run on multi-zone Kubernetes clusters, Amazon EKS for AWS regions and Google GKE for GCP regions. If a container fails due to a hardware fault or software issue, Kubernetes automatically removes it from the load balancer, spins up a healthy replacement and resumes traffic handling within seconds. Because no local state is tied to the failing instance, customers experience no data loss and only the in-flight requests handled by the failed node are impacted.

Resiliency extends beyond individual pods or nodes. If an entire availability zone experiences an outage, the global load balancer instantly redirects traffic to healthy zones. commercetools maintains buffer capacity in every zone so that this redirection can occur without overloading the remaining resources. Kubernetes then automatically scales out additional capacity to restore the original resource balance and recover redundancy. This combination of zonal redundancy, elastic scaling and automated healing ensures that even zone-level failures have minimal impact on customer workloads.

The use of a service mesh across microservices adds another layer of resiliency. Health checks, retries and circuit breakers are applied consistently, ensuring that failures are detected and routed around quickly. Combined with observability pipelines, these mechanisms allow the platform to detect and mitigate issues in near real-time.

Database failing

Data availability is fundamental to resiliency, and commercetools is designed to withstand database failures without impacting customer operations. Our persistence layer is built on cloud-native distributed databases that are deployed in multi-zone clusters across every region. Each cluster spans at least three independent availability zones, ensuring that a failure in one zone can be absorbed seamlessly by the others.

When a node or zone experiences disruption, traffic is automatically redirected to healthy replicas. This process is fully automated and happens within seconds, maintaining continuity without manual intervention. Because data is synchronously replicated across nodes, reads and writes continue with minimal latency, even during failover events. Customers experience uninterrupted service, with only a small fraction of in-flight requests needing retries.
Our architecture also leverages event-driven design, allowing the state to be rebuilt from event streams in cases where rehydration is faster or more precise than snapshot restoration. This capability is particularly valuable in scenarios where downstream indices or derived data stores, such as search, need to be reconstructed quickly.

Other Services Failure

Beyond databases and application services, commercetools relies on a range of supporting components load balancers, messaging queues, object storage and content delivery networks, that are critical to maintaining overall resiliency. Rather than building and maintaining these complex systems in-house, we leverage cloud-native, state-of-the-art managed services from hyperscale providers. This approach allows us to take advantage of their proven durability, multi-zone distribution, and continuous investment in resiliency engineering.

Resiliency is not assumed; it's continuously validated. commercetools integrates these managed services into our observability pipeline, where health checks, error rates and latencies are monitored in real-time. Any degradation in a provider service is detected quickly and triggers automated routing, retries or failover to alternate zones or components. In some cases, we maintain multi-provider strategies, ensuring that even systemic cloud-level issues can be mitigated by switching to equivalent services in other regions or providers.

Backup concept

At a high level, a commercetools solution consists of data, infrastructure, and the application itself. A guiding principle behind our approach is to build solutions that can function as stateless as possible, as this is a key driver for scalability, reliability, and resiliency. Consequently, data backup is the core element of commercetools' backup and recovery plan.

A significant portion of our data is stored in MongoDB databases. Any other data-handling technology of the platform is dependent on that data. commercetools performs full database backups. In addition, disk snapshots and hot incremental backups are performed.

The disk snapshot process is fully automated and occurs every two hours for the last two days and daily for 30 days. We make a fully encrypted copy of all persisted data and replicate it within the same compliance geography.

Check full details on how we manage backups.

Recovery concept

While our backups are performed automatically, the data recovery process is manual. This is because each individual incident is unique and requires different actions to ensure that the services are restored appropriately. Whatever the incident, we are held to our SLAs for RPO and RTO to ensure the timely restoration of functionality.

Check our full SLAs

In-region recovery

As mentioned earlier, we work diligently to prevent issues from entering production. However, in the rare cases where an issue does make it into production mitigating the problem before customers are aware of or impacted by it.

If a version rollback isn't an available solution, we may need to create and deploy a fix for the issue. While the time required to create a fix varies, deployment of the fix can be done within minutes.

In rare circumstances, a critical issue could require the redeployment of a service within the region where the issue exists. The complexity of this depends upon the service and will follow a similar path as multi-regional recovery, but isolated to the affected service.

Data recovery

We leverage the disk snapshots to expedite the recovery process of customer data. These snapshots will then be directly converted into a new virtual disk within any region and zone of the same cloud provider on the same continent and directly attached to a new MongoDB cluster. Leveraging native cloud services eliminates the need to install or sync any data during recovery, expediting the overall process.

Any other data, such as platform configuration data, is stored within a Git repository. This information is replicated live across multiple data centers and is pulled directly into the newly set up persistence.

Infrastructure recovery

All infrastructure details are managed through Terraform and tracked through Git repositories. These Terraform files allow for bootstrapping all MongoDB virtual machines, Kubernetes clusters, ElasticSearch services and all other resources through the execution of a Terraform script against the new cloud region.

As part of the regional setup, Terraform specifications are stored in a Git repository that is replicated live across multiple data centers and tracked. Any deviations from the default configurations required to run this particular platform are deployed directly to the new region.

Applications recovery

Once all infrastructure is provisioned and the snapshots are converted to virtual disks, the process of installing commercetools microservices begins.

Deploying microservices follows an automated process similar to the standard CI/CD deployment for all changes. During the standard CI/CD process, the deployment retrieves a set of Helm charts from the commercetools Git repositories. These Helm charts contain all details on how to deploy a specific microservice along with instance-specific scaling parameters. Executing these Helm charts includes the following:

1. Building Docker containers.
2. Sending containers to docker registry.
3. Deploying docker containers to Kubernetes.
4. Configuring Kubernetes.

The docker registry backs up all containers to Google Cloud or AWS storage, as applicable and replicates them across all regions. In a disaster recovery scenario, instead of leveraging these backups, the process bypasses steps one and two. This deployment pattern is significantly faster when deploying all microservices.

Search recovery

A special case within the commercetools application is the Search feature. Various Elasticsearch setups are used to drive internal services, but more importantly, they also power product search and Merchant Center services. ElasticSearch is being used as a key technology for that.

The commercetools ElasticSearch indices are derived from the MongoDB product databases as outlined above. Once the core application, including ElasticSearch, is up and running, a sync job between MongoDB and ElasticSearch is triggered to rebuild all search indices.

Cutover

The new region will reuse all of the failed regions' projects, configurations and settings. The final step will be to route all DNS records to the new instance. This routing takes place within the cloud provider and does not rely on DNS propagation. The traffic will immediately be routed to the new installation.

All URLs, Keys and Secrets will remain. Calling applications will not require modification and should begin functioning as expected after the cutover. Depending on the implementation of individual clients, a reboot may be necessary to reconnect them.

Support

commercetools' support organization is globally staffed with highly skilled full-time engineers who provide 24/7 incident communication, providing our customers with a single point of contact that covers all of our products.

Using best-of-breed tooling in an integrated setup of support, SRE and engineering teams, automated alerting processes and employee schedules are skillfully managed. Internal process definition and process automation tooling are continuously improved with learnings from every incident.

Check our full support offering.

Support Services

We are committed to providing our customers with all the support they need to achieve their eCommerce objectives. In the case of a service interruption, customers can monitor the status of our products via our status page. This page also allows users to register for proactive notification of incidents or degradations in their respective hosting Regions. New issues can be reported via our support channels, which guide users and ensure the right escalation is immediately triggered.

When problems are detected in an individual tenant's usage patterns through commercetools' automated status monitoring processes and on-call rotations, the support team proactively reaches out to the escalation contact provided by the customer. We strongly recommend that customers regularly update their emergency contact information through their Customer Success Manager (CSM).

As part of our culture of continuous improvement, internal incident post-mortem meetings are held consistently to cover process and technology learnings, which leads to a reduction in future incidents and further resiliency.

Customer's role

As with any provided software, there are things that the customer can do to leverage and improve commercetools' resiliency:

1. Create and maintain an updated list of services and programs that interact with commercetools and/or online commerce.
2. Create and maintain a list of dependencies.
3. Connect with a Customer Service Manager (CSM) regularly.
4. Document the HA/DR features and specs of all services in the solution.
    a. E.g. Google Cloud, AWS and MongoDB have built-in HA/DR systems
    b. If an external search is used, determine the provider's fail-safes and plan accordingly.
    c. If an external PIM is used.
    d. If external promotions are used.
    e. If external discounts are used.
    f. If there is integration with a CRM, etc.
    g. If tertiary services from the frontend are used.
5. Work with CSM and Professional Services to determine which areas might create issues.
6. Create an internal HA/DR plan of action.
    a. Be aware of the built-in backup frequency for each component in the solution.
        i. Provided backups may not have been designed to be utilized for internally triggered systemic issues that might cause service interruptions, such as accidentally pushing code with errors.
    b. The best practice is to augment these backups with an internal backup protocol based on individual needs.
7. Implement regular training on best practices.
    a. If a new person is not trained on what to do in the event of an issue, it can:
        i. Result in errors that exacerbate the situation and possibly cause an outage.
        ii. Significant delays in the recovery of service.
8. Create a list of all providers to contact.
9. Inform all providers, including commercetools, as soon as possible when a technical issue arises.
10. Update contact information for all vendors at least quarterly and set up automatic reminders to ensure the data is always up-to-date.

Continued resiliency

commercetools is continually improving the scalability, availability and resiliency of our products. Leveraging modern deployment technologies along with our agile mindset and processes allows us to deliver updates to all of our customers quickly. Our product-centric culture is committed to ensuring that all our products are reliable, highly available, and resilient.

We look forward to building a better commerce portfolio for you.

Appendix A: Glossary

Availability. APIs and application data are responsive and functioning optimally, typically described in terms of uptime. commercetools measures Availability by sending test requests at regular intervals for the targeted product. The results of such measurement are available at Status Updates.

Active/Active. Two instances of the software product running simultaneously. In the event that one goes down, traffic is automatically routed to the other instance. Least possible losses.

Active/Passive. Two instances of a software product, one running and the other passively updating at determined increments. In the event of an outage, the passive instance starts up and traffic is routed to it. the last synchronization period. If that sync was 10 minutes prior to the outage, then 10 min will be lost.

Customer data backup. The ongoing, frequent or real-time backup of the customer's data is solely the responsibility of the customer.

Disaster recovery. An organization's ability to respond to and recover from an event that negatively affects business operations.

Downtime. The percentage of the overall time during which the service is unavailable.

High availability. The ability of a system to operate continuously without failure for a designated period of time.

Resiliency. The ability of an application to react to problems in one of its components and still provide the best possible service.

RPO. Recovery point objective. The maximum amount of data, as measured by time, that can be lost after a recovery from a disaster.

RTO. Recovery time objective. The duration of time and the service level within which a business process must be restored after a disaster in order to avoid unacceptable consequences associated with a break in continuity.

System backup. Backups performed by commercetools on a periodic basis are point-in-time snapshots of the platform and the customer's data.

Uptime. Percentage of the overall time that a service is running and available (e.g., 99.9%).

Appendix B: Technical stack example

While our services are independently developed, we have common tech stacks. Here's an example tech stack from one of our services:

Cloud environments	Google Cloud / AWS
Container-orchestrator	Kubernetes (EKS for AWS and GKE for Google Cloud)
Programming languages	Scala
Runtime environment	JVM
Database	MongoDB Atlas
Message service	AWS SQS
Testing framework	ScalaTest / Cornichon
CI/CD	TeamCity / CircleCI
Monitoring	Kibana, Prometheus, Grafana, Pingdom, Pagerduty
Extra dependencies	Akka streams, Cats effect, Unfiltered, Scalafmt, Scala Steward

About commercetools

commercetools is the leading composable commerce platform, allowing companies to dynamically tailor and scale shopping experiences across markets. We equip some of the world's largest businesses with tools to future-proof digital offerings, reduce risks and costs, and build outstanding experiences that drive revenue growth.

Headquartered in Munich, commercetools has led a global renaissance in digital commerce by combining cloud-native, technology-agnostic, independent components into a unique system that addresses specific business needs. We empower brands — including Audi, Danone, Eurorail, NBCUniversal, Sephora and Volkswagen Group — to stay ahead of changing consumer and buyer behavior.

More information at commercetools.com.