commercetools
Next generation commerce

commercetools' Resiliency

commercetools.com

Content

commercetools' Resiliency

Summary 3
Basic Architecture 4
Cloud Services 4
Regions and Availability Zones 5
Independent APIs Backed by Microservices 7
Database Sharding 8
Command Query Responsibility Segregation (CQRS) 9
Multi-tenancy 10
Cloud Service Security 11
Microservices vs. Monoliths: Resiliency Within 11
- Measures Against Single Points of Failure 12
- Traffic Overload 12
- Software Bug or Infrastructure Misconfiguration 13
  - Software Bug 13
  - Infrastructure Misconfiguration 14
- Stateless Component Failure 14
- Database Failing 14
- Other Services Failing 15

Backup Concept 15
- Recovery Concept 16

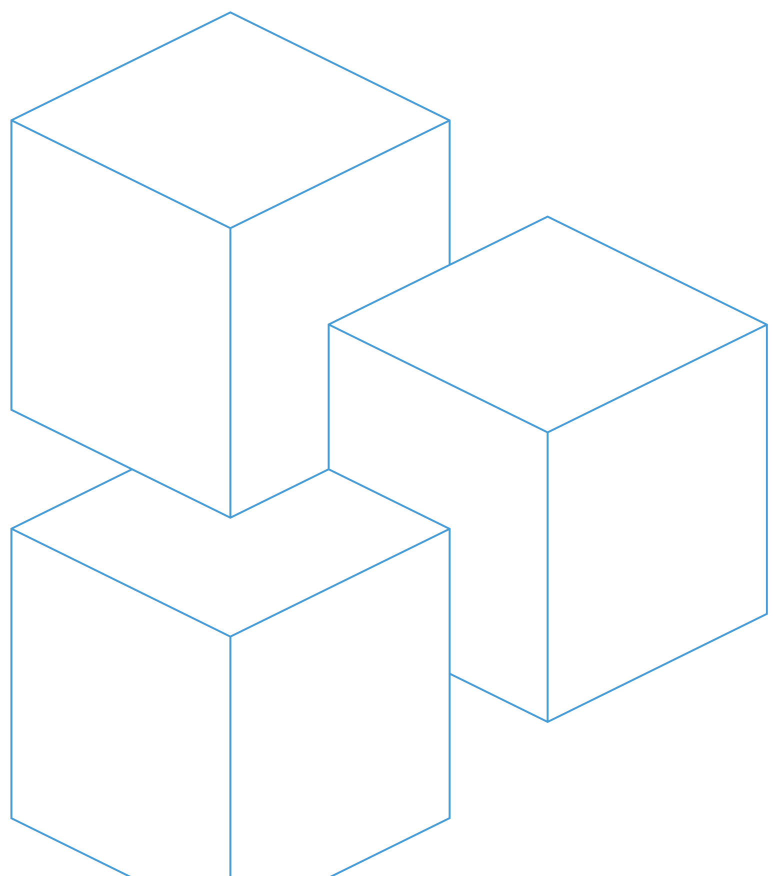
Support 19
- Support Services 19

Customer's Role 20
Continued Resiliency 21
Appendix A: Glossary 22
Appendix B: Technical Stack Example 23
About commercetools 24
Contact Us 24

Summary

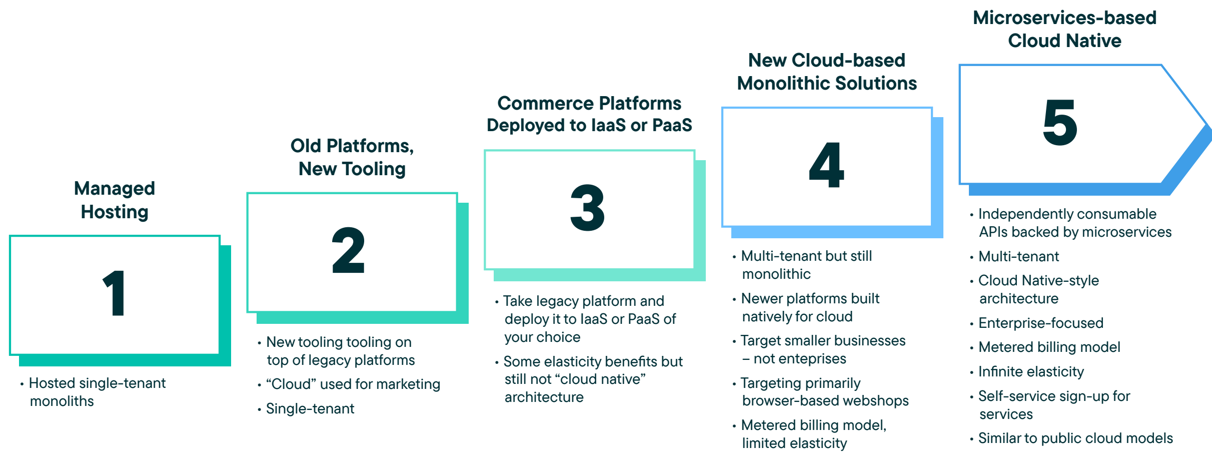
In this document, we demonstrate the resiliency of commercetools in both service availability and data recovery. Our products have supported companies through multiple Black Fridays, Cyber Mondays, dedicated product launches, and all the days in between. Through all of this, product outages are extremely rare.

commercetools' portfolio is divided into several SaaS products. Each product is built and delivered natively in the cloud as a multi-tenant service. This approach toward software design and development is fundamental to how we build and deliver products. Each of these products comes with an SLA and is built via a combination of microservices, and the functionality is accessible via API and business user tooling. commercetools has dedicated years to building a scalable, robust, and secure set of products that are available in a diverse set of geographical regions and are resilient against issues and outages. In the following document, we explore the basic architecture of the commercetools solution, identify key aspects in the microservices approach we follow, explore our backup and recovery concepts, and describe our outstanding customer support services. At commercetools we are confident in our products, and by the end of this document, we hope you feel confident in them, too.



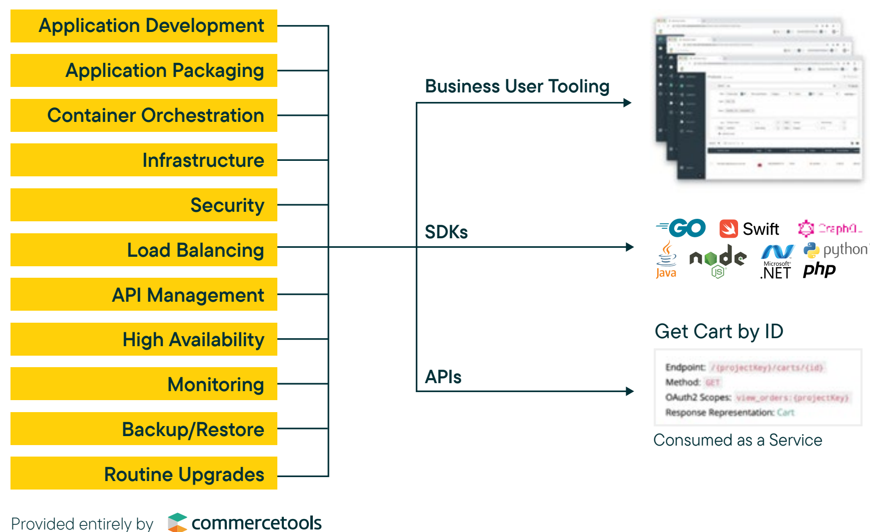
Basic Architecture

This section details the multi-tenant, cloud-native, microservice architecture deployed by commercetools. This architecture follows the current industry best practices and allows us to be fast and flexible in our design and resilient against failures. The image below demonstrates the forward evolution of software architectures, including commercetools' microservices-based, cloud-native architecture.



Cloud Services

Cloud service providers supply a built-in toolkit of highly-available and scalable components that can be leveraged to create complex products. commercetools offerings are available on Google Cloud and Amazon Web Services (AWS). The image below provides a brief overview of how the commercetools' architecture works together in unison.

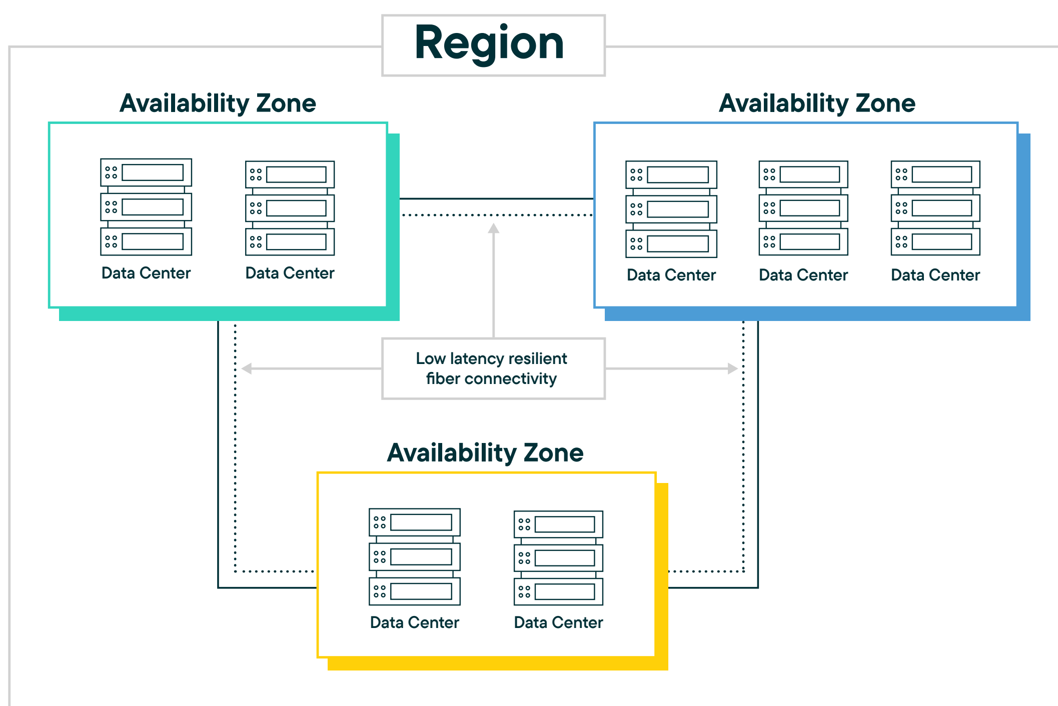


As a customer, you can deploy services that integrate with our offerings, like a front-end, in the cloud vendor of your choosing. These cloud-native services are designed and maintained by the providers and other expert software companies. The cloud provides more resilient performance and allows for significant cost and time savings.

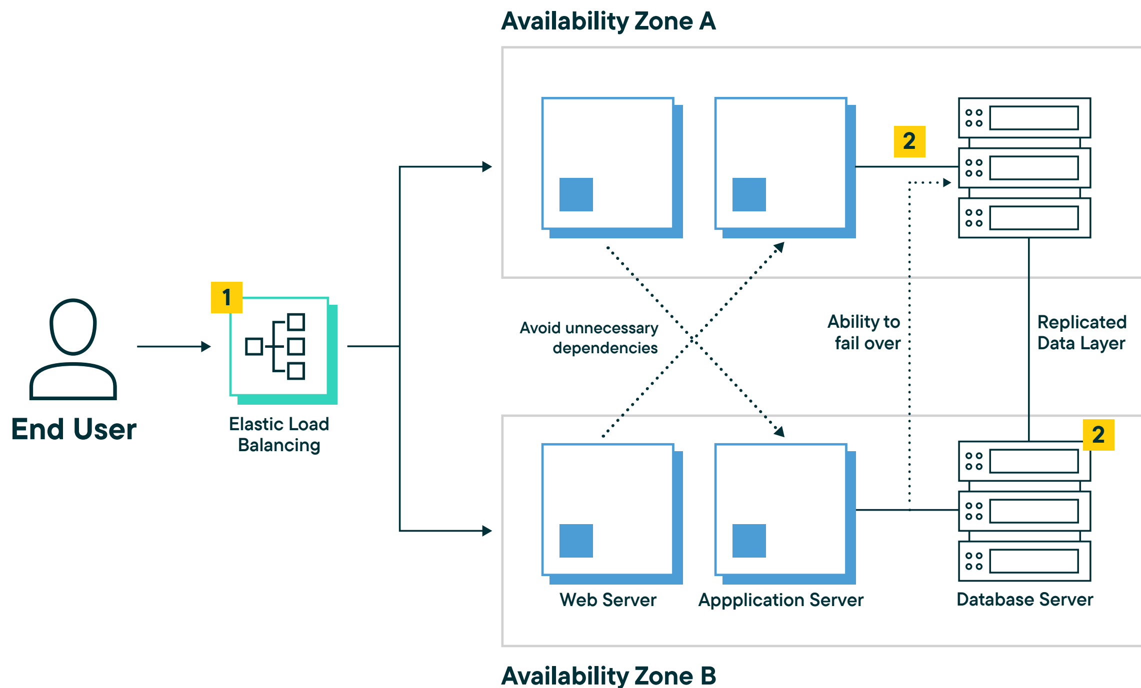
Regions and Availability Zones

Cloud service architecture enables the ability for hardware infrastructure to be distributed across the globe via regions. Our services were designed as cloud-native to take 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 is 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 have a fully active/active availability within a region. Below is a visual example of how Availability Zones are utilized within a Region.



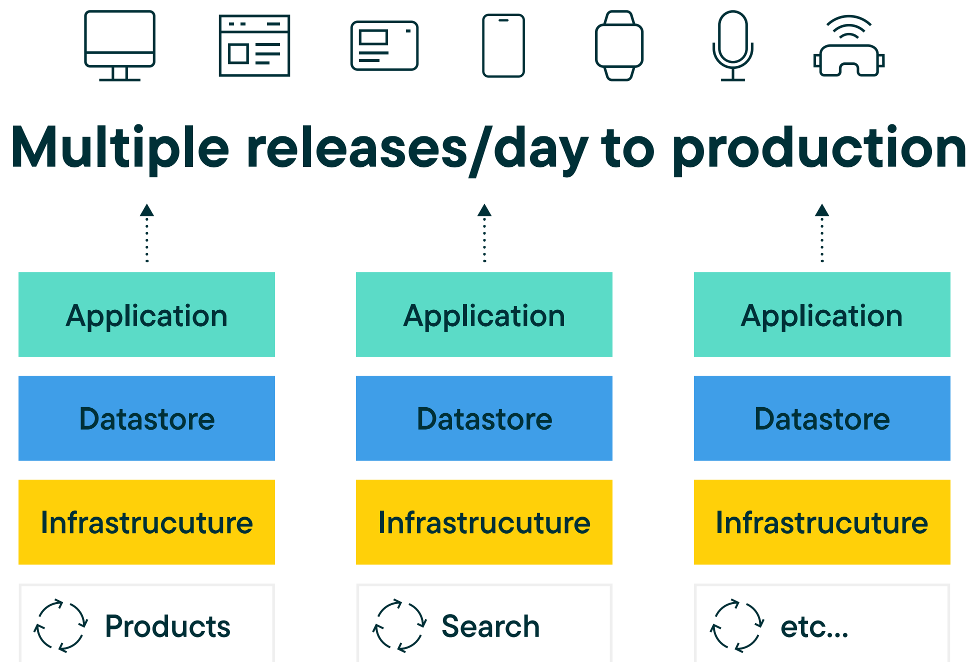
Being in the cloud does not leverage Availability Zones by default. Customers can choose to deploy an on-premise application into a single zone in the cloud. If the application is not architected/designed to be cloud-native, customers are sometimes locked into this type of deployment model. Deploying 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 3x redundancy exists, ensuring that individual service (application, database, etc) is never interrupted regardless of hardware failures, system issues, or zone outages.

Independent APIs Backed by Microservices

Our product features are delivered via independent APIs backed by microservices. Each microservice is deployed into the cloud region using Kubernetes and maintains its own application layer, datastore, and infrastructure. This architecture enables expert teams to deploy updates through independent development and release cycles resiliently and quickly, as demonstrated in the image below.

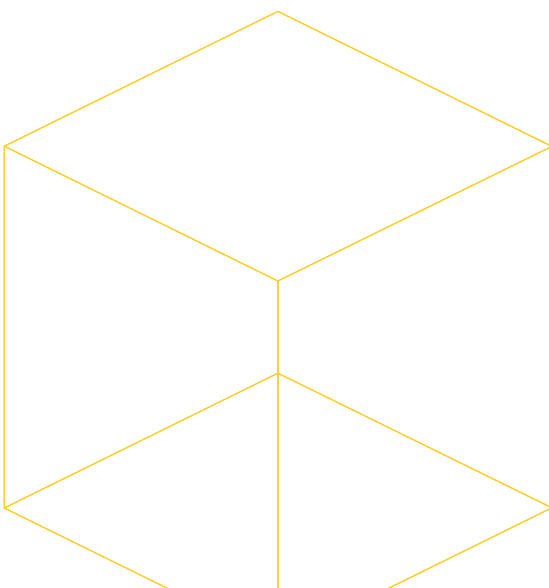


We have also implemented Continuous Integration and Continuous Deployment (CD/CD) pipelines for our services. This approach enables us to leverage automation and lifecycle monitoring to simplify the process of delivering code changes to production and allow for safe deployment to production within minutes.

Database Sharding

Database sharding enables on-demand, horizontally scalable data infrastructure. Distributed systems process and store data in capacities that far exceed the capabilities of any commodity virtual machine. Distributed system architectures commonly partition data, or **shard**, data into “keyed partitions” distributed over a vast number of virtual machine instances and disks.

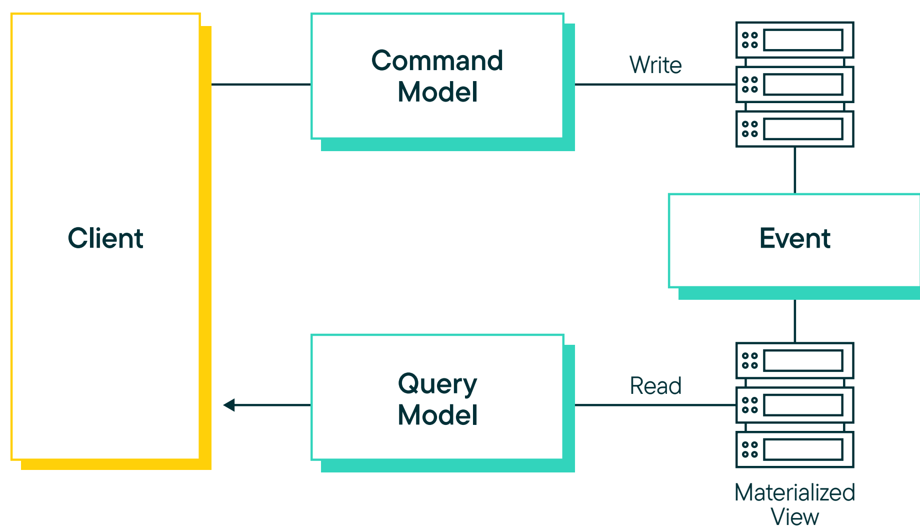
We leverage MongoDB Atlas for our commerce database needs. MongoDB Atlas is a fully-managed, cloud-native, document database-as-a-service with elastic scalability that inherently supports **sharding** and replication, as demonstrated in the image below. We distribute load across multiple shards, and if load increases, we can increase our ability to handle database interactions by adding additional database servers. MongoDB Atlas is also a multi-could and multi-region offering that leverages at least three Availability Zones in a region, ensuring continued resiliency.



Command Query Responsibility Segregation (CQRS)

We leverage the power of Command Query Responsibility Segregation (CQRS) design patterns, which operate under the principle that data is not written the same way it is stored and read. With this pattern, any incoming update requests are processed and turned into multiple events that are received by independent microservices for asynchronous handling. This allows for minimal load and maximum resiliency.

CQRS dramatically enhances our availability by allowing each datastore to scale according to its own use case-specific needs. Datastores that handle writes are configured very differently from datastores that handle reads. By separating reads from writes, we maintain optimal availability and scalability. This process is demonstrated in the image below.



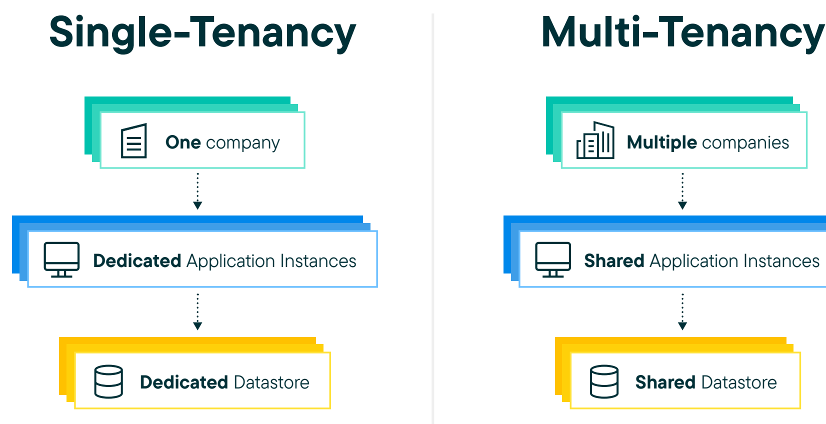
Our architecture also takes advantage of two types of data consistency: strong consistency and eventual consistency. With **strong consistency**, also known as read-after-write consistency, changes are immediately reflected after the API returns a response code. So an immediate read request will reflect the change.

With **eventual consistency**, when an API call performs a modify action on an entity, following the CQRS design pattern, the request is processed, queued, and eventually sent out as an event for handling. The API returns a response code before the changes are processed. An immediate read request may or may not reflect the change, though in time the change will be reflected.

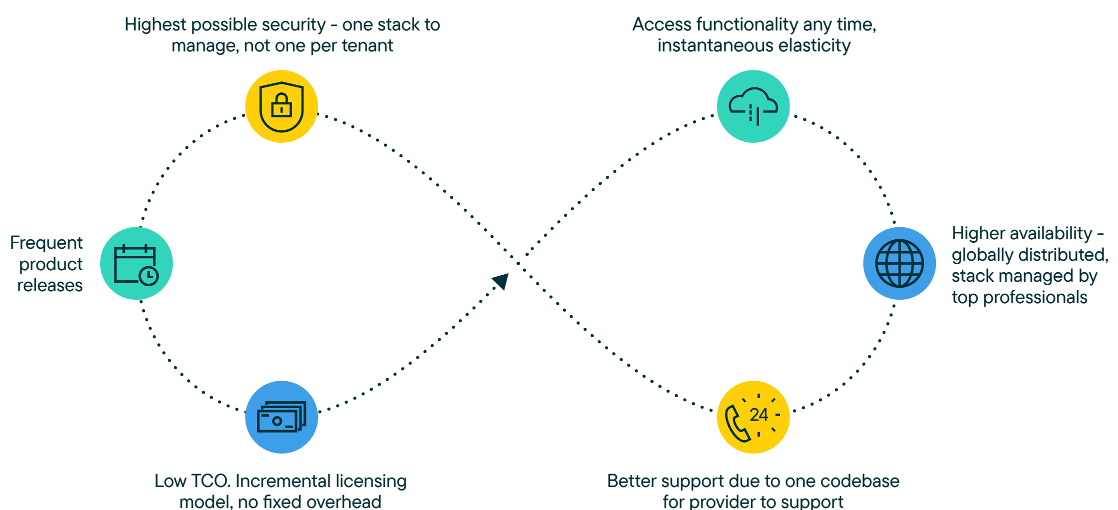
Leveraging two types of data consistency allows us to minimize load and handle many millions of API requests resiliently.

Multi-tenancy

Our products are multi-tenant by design, meaning that multiple customers share the same infrastructure and application resources but do not have access to other customers' data. To achieve this, each tenant project is provisioned with a unique logical database in our shared MongoDB instances upon creation. The logical databases share the same infrastructure resources but are otherwise separate. We then manage access to these logically separated projects via our API credentials. The image below offers a brief overview of single-tenancy vs. multi-tenancy design.



Our multi-tenant system has much higher limits for scalability than a single-tenant system. That said, there is always a limit, and a misbehaving tenant can hit these limits. To protect other tenants, such cases are automatically identified, and we limit the number of resources we dedicate to such a tenant (e.g. by shedding load). Since we have a resilient, robust, and secure multi-tenant solution, we do not do one-off deployments. This approach enables us to ensure all of our customers in a region are getting exactly the same experience and all customers immediately benefit from performance improvements and fixes. The image below demonstrates the ways in which our multi-tenant solution is most beneficial for customers.



Cloud Service Security

Cloud service deployments provide optimal environments and streamlined processes to update services with significantly fewer assets to secure. This is also accomplished without the considerable risk of on-premises infrastructure, facilities, and networks to protect, manage, and fund. Our cloud-native services facilitate the rapid deployment of important code revisions to a minimal scope of systems along with drastically reduced time from start to finish for time-sensitive changes.

Our native cloud services significantly reduce the inventory of infrastructure software to manage and protect with a small footprint of systems to secure, considerably less than on-premises environments. We also avoid the risk of securing servers as hosts by utilizing Kubernetes in the cloud. With our cloud-native services, customers no longer need to oversee the vast scale and complexity of infrastructure, hardware, and scope of vulnerabilities with on-premises services.

We remove the burden and risk of enterprise operations and security inherent with on-premises systems, facilities, and software. Our customers no longer require the extensive controls, effort, operations, monitoring, and management required to secure on-premises products, upholding our standards of resiliency.

Microservices vs. Monoliths: Resiliency Within

This section further details the resiliency of the microservice architecture deployed by commercetools. In a microservice architecture, each microservice is isolated. This means that a failure on one service may or may not impact the entire application. For example, at commercetools, our Import API is a completely separate service from our Audit Log. Each service maintains dedicated databases and runtime applications. They are connected in that when a change is made with Import API, Audit Log records it. If the Import API has a critical failure, it will cease to work because they are independent microservices. This failure is painful for those leveraging the Import API functionality, but Audit Log will continue to function tracking changes made by other services. Similarly, a shop will be unaffected by an Import API or Audit Log service failure.

This is a significant advantage over monolithic designs, in which if any small piece of the system has a critical failure, all components within the system are prevented from running. The impact of failure situations can be minimized through microservice architecture by breaking functionality into individual pieces. This also simplifies the development cycle, as each piece is only meant to perform a specific task.

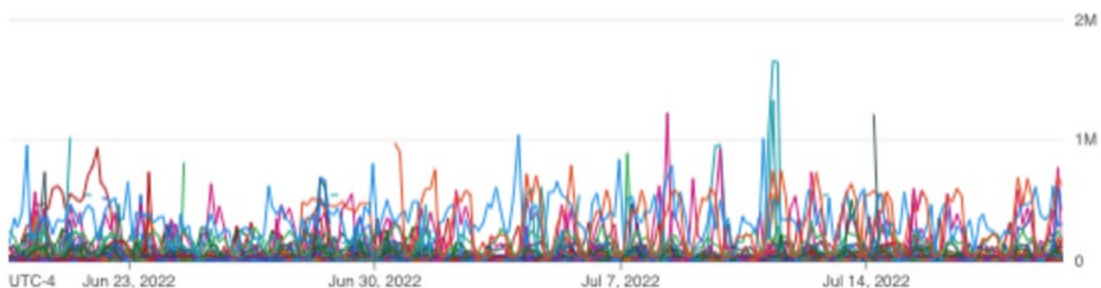
It is important to note that even in microservice architecture, there are some application critical services that may not directly prevent another service from running but stop it from functioning properly. These microservices are Single Points of Failure. An example of this would be our Authentication service. All of our API calls must be authenticated. Not doing so would obviously be a huge security risk. So if the Authentication service were to fail, all other services would still run but could not authenticate. All API calls would respond but with failure codes. In our case, this service is a Single Point of Failure. We highlight these differences between monolithic and microservice architectures to demonstrate the resiliency of microservices in the commercetools solution.

Measures Against Single Points of Failure

Given that microservice architecture can have Single Points of Failure services, we guard against various scenarios via carefully crafted, resilient methodologies as outlined in the following sections.

Traffic Overload

Traffic overload is always a concern for a multi-tenant SaaS product. Caching data is a great way to lessen the burden of heavy traffic. We ask our customers to cache as much information as possible, and we also do internal caching for reads. Of course, writes cannot be cached, but they are distributed based on individual customer demands.



Example graph of write requests color coded per project

Even with safeguards, unexpected traffic spikes do happen. We have several preventative measures in place to guard against traffic overload. To mitigate potential issues, we always run our services with a buffer of resources. We try to keep resource usage to around 50% of the

available amount. This provides the first layer of defense against a traffic spike. Being cloud-native allows us to auto-scale to quickly adapt to growing traffic. Our stateless Kubernetes nodes can up-scale quickly (about 2 minutes for a new node and less for individual applications). While our MongoDB Atlas databases take a bit longer to adapt, they can also auto-scale when needed.

Additionally, we use background processing whenever possible. While in some cases it's critical that an API call synchronously processes the response, other work can be performed asynchronously. This work can be put into a queue and handled via background processes.

The queued changes are not updated all in parallel. This approach would unnecessarily tax the system. Instead, our background processes can pull updates off the queue in small batches to lessen application load. In the case of a failure, the queuing mechanism enables retries without impacting the other updates.

Software Bug or Infrastructure Misconfiguration

In any software application, a failure scenario can be introduced by a software bug or infrastructure misconfiguration of the software. We have a multi-tiered methodology to guard against these types of issues from affecting production environments.

Software Bug

First, we do code reviews on every change that is delivered. At a minimum, we need two approvals on every code review. This ensures that at least two developers other than the author have seen and verified the changes. Then we run extensive test runs before the code is deployed to staging environments. These automated test suites validate new code behavioral correctness and guard against behavioral regression.

Leveraging CI/CD methodology, we deploy multiple times a day with small change sets. First, we deploy to a staging environment and validate. The validation of a change takes as long as required. We will not rush a change into production.

After validation, we deploy to production. We have the ability to deploy the change to only a subset of our overall traffic set. For example, we can deploy a change where only 1% of our overall production traffic will encounter that behavioral change. Once the behavior is validated, we can roll it out to a larger segment of the traffic base. Not all changes are deployed incrementally, but this is a good way to ensure higher-risk changes have minimal negative impact should an issue arise.

If a bug makes it into a production environment, we can quickly identify the change causing the failure and roll it back to a working version. The process of rolling back to a working version can be done in a matter of minutes.

Infrastructure Misconfiguration

Our infrastructure and configuration is versioned and applied during a deployment. Should a misconfiguration be introduced, we can easily roll back to a previous version. For time-critical situations, we can apply manual changes. These changes are always done with at least two engineers managing the change.

Stateless Component Failure

We have also architected against a component or service within our product fail. Our services (e.g. those handling API calls) are always stateless and deployed on a multi-zone Kubernetes (EKS for AWS and GKE for Google Cloud) cluster. This means that a single instance of the service failing, due to a hardware failure or software bug, will only affect the API calls it is currently handling. The failed instance will be quickly taken out of the load balancer and replaced by Kubernetes.

If an entire availability zone fails, the load balancer will redirect the traffic to another availability zone. Because we always keep extra resources on hand, this availability zone will be able to handle the increased traffic load while the Kubernetes cluster auto-scales for increased load and repairs the damaged zone. This effectively minimizes the impact of these types of failures.

Database Failing

Another scenario we prepare against is inaccessible data due to database failure. Most of our data is stored in cloud-native databases, specifically MongoDB Atlas. MongoDB Atlas is a state-of-the-art cloud-native managed service that provides automatic recovery for most issues. The database is deployed in a cluster of three nodes each in an independent zone (separate location and hardware resources) within a cloud region. If any one zone fails, traffic is automatically redirected to one of the other availability zones. Additionally, MongoDB has a full support team at the ready should any issue not automatically recover. In a worst-case scenario that requires a data rebuild, we can restore from a saved data backup.

Other Services Failing

Similar to a managed database, we will sometimes leverage other state-of-the-art solutions from cloud providers (e.g Load Balancers). This ensures that we always have the best solution to solve the problem and don't over-burden our engineers with creating an in-house solution when one already exists. We always ensure that the services deployed deliver high availability, leveraging multiple cloud availability zones in a region.

Backup Concept

On 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, the backup of data is the core element of commercetools' backup and recovery plan.

A large segment of our data is persisted within MongoDB databases. Any other data-handling technology of the platform is dependent on that data. commercetools performs full database backups with incremental backups in-between. In addition, disk snapshots and hot incremental backups are performed.

Our practice is to maintain three copies of your data stored at two separate locations. The disk snapshot process is fully automated and occurs every two hours. We make a full encrypted copy of all persisted data and replicate it across multiple Google Cloud or Amazon Web Services (AWS) regions within the same compliance geography.

For full details on how we manage backups, please visit this site:
https://docs.commercetools.com/offering/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 the services are appropriately restored. Whatever the incident, we are held to our SLAs for RPO and RTO to ensure timely restoration of functionality.

Our full SLAs are documented here: https://docs.commercetools.com/offering/sla

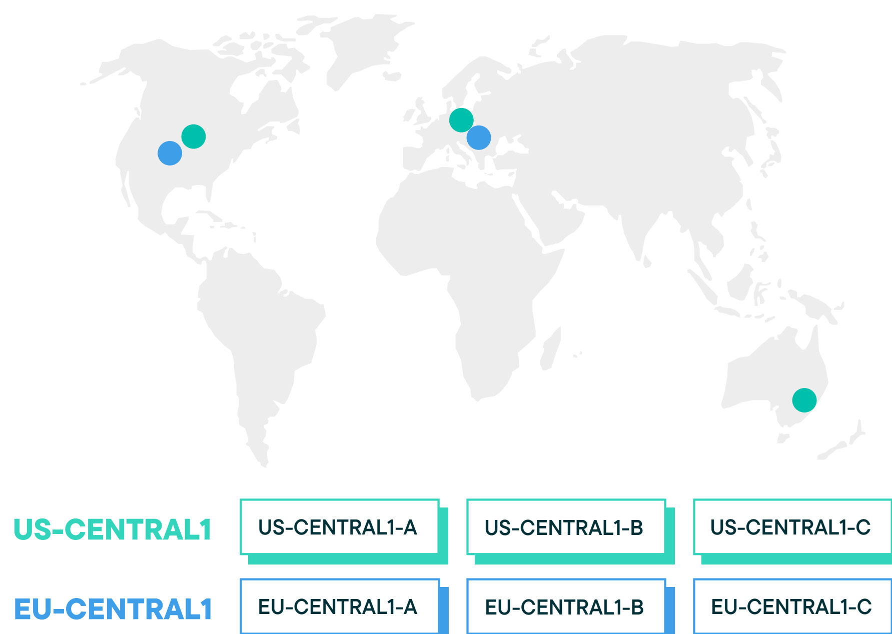
In-Region Recovery

As mentioned earlier, we work hard to guard against an issue making it into production. But in the rare cases that an issue does make it into production, we are able to quickly identify it and roll back to a previous working version within minutes, often mitigating the problem before customers are aware or impacted.

If a version roll-back is not 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.

Multi-regional Recovery

At commercetools we can recover to a separate region should the situation demand it. This recovery would be a multi-phase process. While not all of our products follow the same exact procedure, below is an example of areas where we can perform a complex recovery in a separate cloud region.



It is worth noting that regional outages affect all SaaS applications running within that region, regardless of the vendor deploying the application. Recovery of our services may not be enough to restore the functionality of your solution.

- **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 that are needed to run this particular platform are deployed directly to the new region.

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

Deploying the 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 Search. Various ElasticSearch setups are used to drive internal services but also, even more importantly, 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 re-use 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 of those clients may be needed to reconnect.

Support

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

Our full support offering is described here: https://docs.commercetools.com/offering/support

Support Services

We are dedicated to providing our customers with all the support they need to accomplish their eCommerce goals. 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 the user to ensure the right escalation is immediately triggered.

When problems are detected in individual usage patterns of a tenant 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 happen consistently to cover process and technology learnings, which leads to a reduction of future incidents and further resiliency.

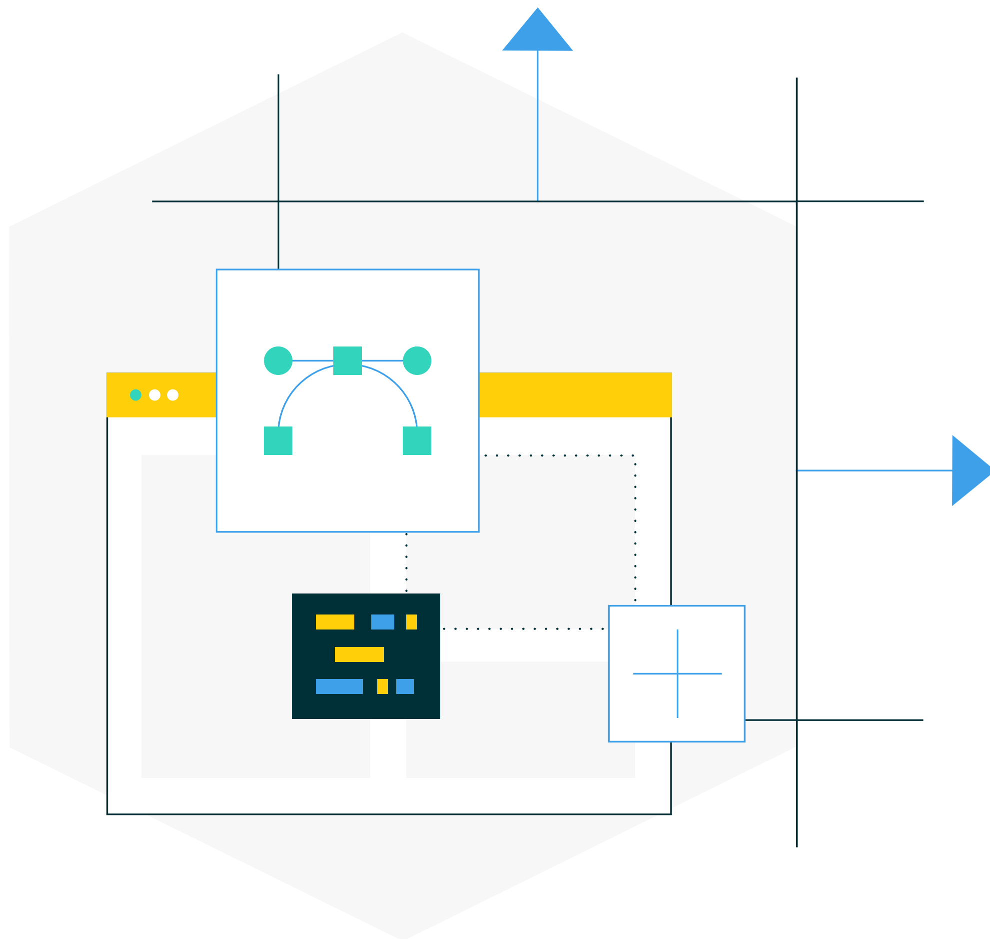
Customer's Role

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. Eg. Google Cloud, AWS, Azure, 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. Know the built-in backup frequency of 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 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 quarterly at a minimum.
    a. Set up automatic reminders

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 allow us to deliver updates to all of our customers quickly. Our product-centric culture strives to ensure all of our products are reliable, highly available, and resilient. **We look forward to building a better eCommerce portfolio for you.**



Appendix A: Glossary

Availability
APIs and application data are responsive and functioning optimally, typically described in terms of uptime. CT measures Availability by sending test requests at regular intervals for the targeted product. CT makes the results of such measurement available at https://status.commercetools.com.

Active/Active
Two instances of a software product running at the same time. 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. Data is current to the last synchronization period. If that sync was 10 minutes prior to the outage, then 10 minutes will be lost.

Customer Data Backup
The ongoing, frequent, or real-time backup of the Customer's data which 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 that the service is not available.

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 a 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 CT on a periodic basis that 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 (ex. 99.9%).

Appendix B: Technical Stack Example

While our services are independently developed, we do have common tech stacks. Here is 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

The inventor of headless commerce, commercetools is an innovative technology disruptor that has established itself as an industry-leading eCommerce software provider. Today, some of the world's most iconic brands and growth-focused businesses trust commercetools' powerful, flexible, scalable solutions to support their ever-evolving digital commerce needs. As the visionaries leading the modern MACH (Microservices-based, API-first, Cloud-native and Headless) architecture movement, commercetools provides customers with the agility to innovate and iterate on the fly, merge on and off-line channels, drive higher revenue, and future proof their eCommerce business.

Based in Munich, Germany, with offices in Europe, Asia, and the United States, commercetools is singularly focused on leading a future of limitless commerce possibilities.

More information at commercetools.com.

Contact Us

Europe - HQ
commercetools GmbH
Adams-Lehmann-Str. 44
80797 Munich, Germany
Tel. +49 (89) 99 82 996-0
info@commercetools.com

Americas
commercetools, Inc.
324 Blackwell, Suite 120
Durham, NC 27701
Tel. +1 212-220-3809
mail@commercetools.com

Munich - Berlin - Jena - Cologne - Amsterdam - Zurich - London - Valencia - Durham NC - Melbourne - Singapore - Shanghai

©2023 commercetools GmbH - All rights reserved

Appendix C: Service Tiers

(To be left out of initial document release)

Prioritizing critical services is the optimal strategy to ensure the highest level of availability and performance. We understand that while it is important for all services to be available, it is equally important to understand and identify which services are critical to primary product use cases because some use cases do not need optimal availability or performance. For example, a 5-year-old order does not require the same lightning-fast response times as a live cart discount. This also enables us to optimize the recovery process in the case of an incident. In a cloud-based ecosystem, prioritizing the essential services ensures that in the unlikely event of an issue, the business will keep moving forward. We have done a great deal of study in creating a strategy to optimize for all use cases and have segmented our services into tiers to help describe this prioritization.

Tier 1 – Services that enable product functionality and customer usage of our products.
Tier 2 – Services that enhance our product offering but do not affect primary product usage.

Product	Tier 1	Tier 2
Composable Commerce	Authentication Commerce APIs (orders, carts, products, etc) Search Subscriptions/Messages	Audit log Import / Export Merchant Center
Frontend	API Hub Delivery	Studio

In a scenario where more than one service is affected and needs restoration, we will focus on Tier 1 services first and follow with Tier 2 services once those are recovered.