

# In-Memory Data Caching for Microservices Architectures

## Table of contents

Adding instances of microservices for performance and scalability . . . . .	4
Operational advantages of a shared caching layer	4
Adding instances for availability and resilience	4
Scaling to meet a high volume of concurrent requests for data	4
Event-based microservices architectures . . . . .	5
Isolation between microservices promotes autonomy	5
How to support concepts that span microservices	5
Sharing events across microservices	5
Event streams and the unified event log	6
Designing for high availability . . . . .	8
Recovering from server or availability zone failures	8
Disaster recovery from site/region-wide failures	9
Self-healing data redistribution after failures	9
Legacy systems are part of your cloud native evolution . . . . .	10
VMware Tanzu Gemfire for microservices . . . . .	11
Get fast access to data	11
Start quickly with a Tanzu Gemfire cluster	11
Maintain business continuity	12
Change capacity dynamically, as and when needed	12
Retain control over capacity allocation and data access	14
Conclusion . . . . .	15

## Introduction

Over the last few years, a microservices-based approach emerged as the ideal way to build web-scale applications, and more and more enterprises are considering this approach for their own custom software development. The benefits of a microservices-based approach are compelling, given the enormous architectural complexity of web-scale apps. In a microservices-based approach, monolithic applications are broken down into smaller, isolated services that different teams can work on autonomously, while minimizing their dependence on each other. Microservices, however, do not exist in total isolation because they need to communicate changes in data and changes in their state to other microservices. An event-driven architecture provides a non-intrusive approach to this communication that does not compromise isolation. This approach supports incremental, rapid release of software, characteristic of Agile methodology, and reduces the risk of failure inherent in the “all-or-nothing” monolithic approach.

Beyond team autonomy, isolation between microservices is at the crux of several other benefits. Individual microservices make it easier to scale applications by adding additional capacity for specific portions of a holistic application. This places demands on the underlying data layer, since multiple instances create a multiplier effect with respect to concurrent access to data. We will describe how these concurrent workload needs can be fulfilled by using a caching layer for access to data.

Isolation between microservices also forms the basis of a highly distributed architecture. It shouldn't matter where a microservice is deployed as long as it can communicate with other microservices over a network. For performance and availability reasons, it should be possible to run different microservices, and multiple instances of the same microservice across multiple availability zones and data centers. To do this, however, requires us to address the data layer implications of this geographical distribution. How does data get shared across geographically distributed microservices?

Microservices represent a generational shift in how applications are built. For this new approach to succeed, it must co-exist with previous generations of investments. It is unrealistic to expect a rip and replacement of legacy systems to accommodate this new approach. Fortunately, microservices support an evolutionary approach to adoption that preserves and extends the value of legacy applications while providing an on-ramp to the next generation of applications.

This paper covers key areas of consideration for data layer decisions in a microservices architecture, and how a caching layer, satisfies these requirements:

- Microservices allow easy scale out and in of application capacity, so both the service layer and the data layer need to support this elasticity.
- Microservices simplify and speed up the software delivery lifecycle by splitting up effort into smaller isolated pieces that autonomous teams can work on independently. Event-driven systems further promote autonomy.
- Microservices can be distributed across availability zones and data centers for addressing performance and availability requirements. Similarly, the data layer should support this distribution of workload.
- Microservices can be part of an evolution that includes your legacy applications. Similarly, the data layer must accommodate this graceful on-ramp to microservices.

## Adding instances of microservices for performance and scalability

A common approach to scaling applications for increasing demands is by adding additional microservice instances. Adding instances scales the business logic of the microservice. This is one of the benefits of microservices - you can scale different services independently. Large scale deployments are likely to require many instances of each microservice.

### Operational advantages of a shared caching layer

Instances of a microservice will all have the same data requirements, so it makes sense to share a caching layer across these instances. This practice is often not followed when, for instance, each instance has its own internal "cache" (in memory) for storing session state. This fragments data across instances that should be treated as a whole.

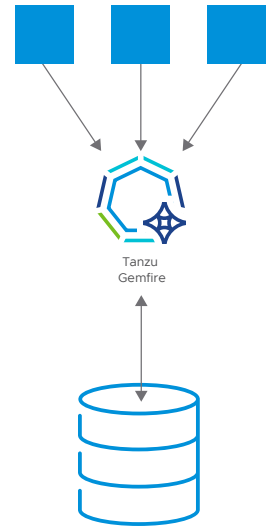


FIGURE 1: Adding instances of microservices for performance and scalability.

Sharing a caching layer eliminates the operational complexity that results from this otherwise fragmented data tier, but it places requirements on the shared caching layer.

The application layer has a single view of user data and it is accessible via any instance. When using a shared cache, updates to data are available to all microservice instances. If the data layer is not shared then each microservice would have a myopic view of the data, and the architecture would have to be set up so that any given user is always routed to the same instance. Not only does this result in a more complex solution than using a shared caching layer, it also results in bad user experiences when an instance to which that user has affinity is unexpectedly lost.

A shared caching layer provides an isolation layer to the backing store(s). Changes to the backing store can be done in just one place, and these changes benefit all the microservice instances.

Adding instances gives cloud native applications an effective and efficient way of scaling the application logic and improving performance. For the overall system to benefit from this, the data layer must match this capability with a corresponding set of capabilities. The application layer's performance and scalability can be significantly eroded if the data layer is a bottleneck. The appropriate technology choice for a shared cache across instances can handle low-latency performance at scale, while maintaining high-availability, using mechanisms like data distribution and replication.

### Adding instances for availability and resilience

Adding instances also helps with availability and resilience. If a microservice instance goes down, another instance of the same microservice can simply replace it. Microservices instances can be added or removed at will, depending on capacity requirements. An added degree of availability can be assured by running each instance on a different server, or even a different site for the highest degree of fault tolerance. The shared data cache should provide a similar degree of fault tolerance from server failures or site outages.

### Scaling to meet a high volume of concurrent requests for data

Running multiple application instances on a shared caching layer will correspondingly increase the number of concurrent requests for access to data. As the instances increase, the number of times the business logic is executed increases, with commensurate increases in requests for data.

The shared caching layer will need to scale to handle the increased concurrent requests for data while maintaining low-latency for each request. Adding instances provides elasticity at the business logic layer. This has to be matched with elasticity at the shared cache layer for the entire stack to scale.

### Event-based microservices architectures

#### Isolation between microservices promotes autonomy

To fulfill the promise of faster delivery cycles, teams need autonomy. Dependence across teams is a recipe for slow progress. This is why monolithic architectures progress slowly. Isolation between microservices is how teams can retain autonomy - maintaining this isolation is critical.

Each team must be empowered to make independent decisions even about their data layer without impacting or becoming dependent on any other team. Even the choice of the type of data store should be independent - a concept known as polyglot persistence. How the data is modeled should also be an autonomous decision, local to each microservice. The team should have full control over making schema changes, i.e. adding or dropping tables and entities, or columns and attributes. What about modifying, adding, or deleting classes and objects? In autonomous teams, these need to be non-breaking changes for other teams, to protect each team's autonomy. The safest way to ensure that each team has independence to make their own data layer choices is to not share the data store across microservices.

#### How to support concepts that span microservices

Some communication between microservices is a requirement, even when they're isolated. Since applications consist of several microservices, the microservices will need to function together as an application in some way. Changes in the state of a given microservice may be of interest to other microservices. Data from one microservice may be needed by another microservice. There are many reasons for microservices to communicate. Good architectures manage communications by making the microservice API the only entry point for accessing its services.

Microservices APIs can be either synchronous or asynchronous. Synchronous patterns can be problematic because of network latencies and intermittent connectivity. Hence asynchronous, non-blocking messaging is on the rise because it lets microservices continue processing without waiting for each other. These messages form a basis of the loose coupling between microservices. Asynchronous messaging requires a small compromise in consistency - it is an eventually consistent model. The loose coupling and performance gained as a result makes this a good tradeoff.

An asynchronous, message-based, event-driven system honors the need for isolation between microservices by making the required communications between them non-intrusive. Microservices can produce events without needing to be aware of which microservices are consuming these events and how the events are being handled. For microservices development teams, an event-driven architecture allows each team to focus on their own problem domain.

#### Sharing events across microservices

The isolation between microservices sets boundaries around each microservice, while the event-driven mechanism addresses how microservices communicate. The role of the event-driven system is critical to the overall architecture operation of the architecture.

A key component of the solution is an event store. The event store system is immutable, sequential, and serves as the destination for the event streams from each microservice. Consumers of these events can then subscribe to and read the events of interest. The event store essentially serves as an event source for each consumer. Consumers maintain their own logic related to the filters that will be applied to determine whether an event is of interest. Each consumer also maintains their own pointer/offset into the event store to serially process the events.

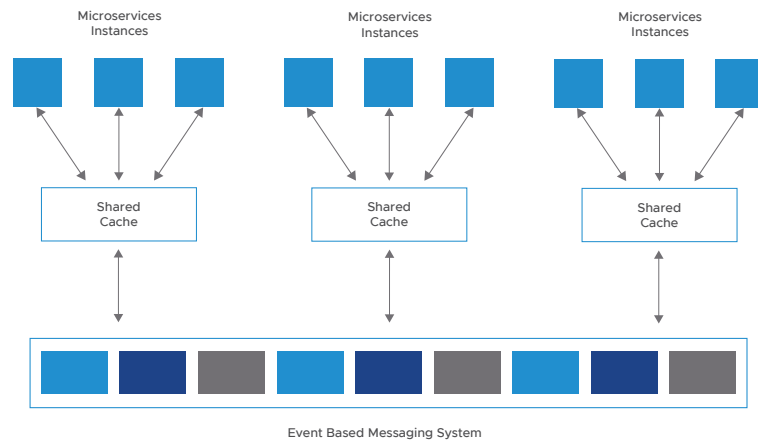


FIGURE 2: Sharing events across microservices.

Common product choices include Tanzu Gemfire for the caching layer, and either Apache Kafka or RabbitMQ for the messaging layer. This pattern can be augmented to include a pipeline orchestration layer, like Spring Cloud Data Flow (SCDF). SCDF provides a framework for composing data pipelines, which includes connectivity to various sources, the ability to specify transformations and branching logic, and the ability choose from various data stores as targets. SCDF allows developers to create, orchestrate and refactor data pipelines with a single programming model for common use cases like data ingestion, real time analytics, and data import/export. SCDF has spawned and subsumes an ecosystem of projects that embrace the microservices approach with loosely coupled services. SCDF can bind to Apache Kafka or RabbitMQ as the underlying messaging system.

Events can be generated either from the application layer, or directly from the data layer. Generating events from the application layer provides visibility into and control over the flow of events, but this comes at the cost of having to manage and maintain the flow of events across all the producers and consumers. Having the events emanate from the data layer frees the application layer, and developers, from having to essentially build major pieces of an event-driven system within the application.

### Event streams and the unified event log

A unified event log is the collection point for all events (state changes) that occurred in any participating microservice. Each participating microservice can also opt to retain a local log of its own state changes. Local logging is optional, especially since the same information is available from the unified event log.

Events collection across microservices opens up an entire category of use cases - those having to do with cross-domain information, i.e. information that spans microservices. The complete view of all events presented in the unified event log can be used to play back selected events and create a projection of the information in any way desired. Various types of custom analytics are a common category of use cases because they often involve some data consolidation. For example, determining whether there is any relationship between customer satisfaction and your suppliers, requires a unified view of data, i.e. the customer satisfaction information and the supplier information are likely handled by different microservices. A unified log can be used to create a projection that includes both customer satisfaction information and supplier information to make this analysis possible. Other common analytic use case examples include the detection and prevention of security infractions and fraud and creating consolidated information snapshots to allow for point-in-time queries. Moreover, snapshots can be pieced together to build trend lines of changes to data. Searching across microservices, and monitoring system performance are other use cases that require cross-microservice data.

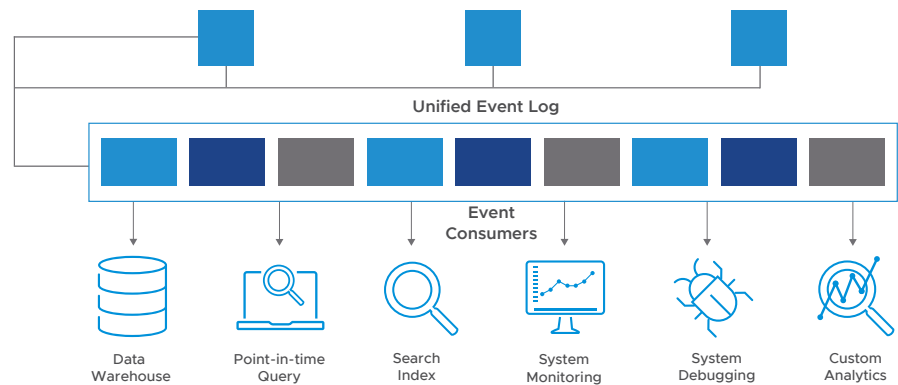


FIGURE 3: Microservices produce event streams.

A unified log can have demanding requirements for performance and scalability given the large number of microservices that can potentially source event streams. Apache Kafka's design for speed, scale, durability, and massive concurrency, together with its model allowing only immutable records to be written to it, makes it an increasingly popular choice as a unified log. Kafka maintains message feeds in distributed and replicated partitions. Kafka can support a large number of consumers and retain large amounts of data with very little overhead. Similar to the multi-instance discussion in the previous section, SCDF can be used as an optional orchestration layer for creating, orchestrating and defining data pipelines using Kafka as an event store. SCDF can bind to Apache Kafka as a messaging system using a Kafka starter.

Based on use case, event consumers can generate data projections by replaying the appropriate events in the log. As data flows through the system, it can be validated and enriched.

Projections from the unified event log can be stored in the most appropriate data management technology for the anticipated workload. For example, with analytics use cases, creating projections in Tanzu Greenplum is a good option. For high volume concurrent lookups with low-latency responses, VMware's Tanzu Gemfire's in-memory cache would be an appropriate target. Graph databases should be used if queries will be run based on graph-oriented relationships between the entities in the domain model. Special purpose databases abound, and the choice of a data store can be based on anticipated workload. Both the underlying data management technology and the data model can be specialized for the use case.

## Designing for high availability

Horizontally scalable systems grow by adding servers (or nodes) to a cluster. In large clusters, often consisting of hundreds or thousands of servers, it is not surprising to see occasional server failures. Designing systems that explicitly make this assumption can ensure continued processing despite these failures. The overall goal is to avoid single points of failure by keeping the application running no matter what fails.

If a server running a microservice fails, the system automatically re-routes work to an alternate instance of the microservice, spins up a new instance to restore capacity, and provides access to the same data from this new instance. This recovery scenario has several implications to how the data layer is set up for accommodating various types of failures.

### Recovering from server or availability zone failures

Cloud-native platforms, like Tanzu Application Service, allow deployment options that support availability zones. In this scenario, microservice instances are distributed across geographically nearby sites, i.e. availability zones. Availability zones often map to physical racks of servers. Data is replicated across these availability zones to provide an alternate path for accessing data in the event of a failure that impacts an entire availability zone. Public IaaS vendors, like AWS, also provide options for choosing availability zones for deploying application instances to locations that are fault-tolerant from each other, with high bandwidth connections between them.

Availability zones can be impacted by far reaching failures that bring down an entire zone, like the loss of a rack's power supply or a network partition that makes an entire zone unavailable, etc.. If this happens, the workload should be automatically routed to one of the surviving availability zones. The probability of multiple failures across availability zones is small, making this an effective way of protecting against zone failures. Placing racks or servers in multiple zones such that a failure of one zone doesn't disrupt the service in other zones should allow the system to continue operating normally with the exception of some short-term performance degradation because of the increased workload being sent to the surviving availability zones.

Availability zones are not just extra infrastructure components that are used only when a site failure occurs. Each application instance should be active, and performing useful work even under normal operation. The workload can be load balanced across different application instances in different availability zones within a region. Data can be sharded across availability zones and the partition key can be used as a mechanism for routing requests to the instance that has the data related to the request. This active-active configuration provides availability assurances related to site outages, without the expense of a redundant infrastructure that is not being utilized under normal operation.

Making copies of data across availability zones raises questions around the consistency model for updating copies of data. The system should allow for the desired level of consistency, i.e. copies of data can be replicated asynchronously, whereby a small compromise in consistency results in much higher performance because the application that updated data does not have to wait for all the copies to be updated before continuing its processing. Synchronous replication, on the other hand, blocks applications from accessing data until all copies have been updated. This provides the highest level of data consistency so that all applications will only see the same latest version of data, and no application will ever see stale data. Since availability zones in the same region are often connected via high-speed, low-latency networks, synchronous updates are a practical option when a high degree of data consistency is desired. Application will not see much of a performance degradation when switching from its primary availability zone to another availability zone that houses redundant data.



### Disaster recovery from site/region-wide failures

For the highest level of fault tolerance, it should be possible to set up the distribution of microservice instances across a wide geographical separation. This makes the system insusceptible to disasters that may impact an entire region (bad weather conditions, natural disasters, security breaches, etc.). It should be possible to set up this configuration using a wide area network topology. In these cases, network latencies make it impractical to adopt synchronous replication, making asynchronous replication the only practical choice.

With the exception of the impracticality of synchronous replication, the operational aspects of systems that are distributed across a wide geographical separation should carry the same advantages of distribution across availability zones, covered in the previous section. If a site or region goes down, the workload should be automatically routed to one of the surviving sites or regions. Each site or region can have a primary data set against which it actively completes useful work, while at the same time providing an alternate resource for accessing data that is the primary data set of another site or region. So, under normal operation each site shares some responsibility for the multi-site fault tolerance of the entire infrastructure, but also does its own useful work.

With the exception of the impracticality of synchronous replication, the operational aspects of systems that are distributed across a wide geographical separation should carry the same advantages of distribution across availability zones, covered in the previous section. If a site or region goes down, the workload should be automatically routed to one of the surviving sites or regions. Each site or region can have a primary data set against which it actively completes useful work, while at the same time providing an alternate resource for accessing data that is the primary data set of another site or region. So, under normal operation each site shares some responsibility for the multi-site fault tolerance of the entire infrastructure, but also does its own useful work.

### Self-healing data redistribution after failures

Recovery from failures will require a dynamic rebalancing of the data to effectively remove the failed server and evenly spread the data across the surviving servers. The server containing the secondary copy of data will now be designated as the primary server for that data. The system will need to return to a high level of system-wide fault tolerance. A new redundant copy of the primary data will have to be created. The failed server most likely also contained secondary copies of data, which will also need to be recreated in the surviving nodes. All this dynamic redistribution of data should be done while the system is still running and responding to application requests.

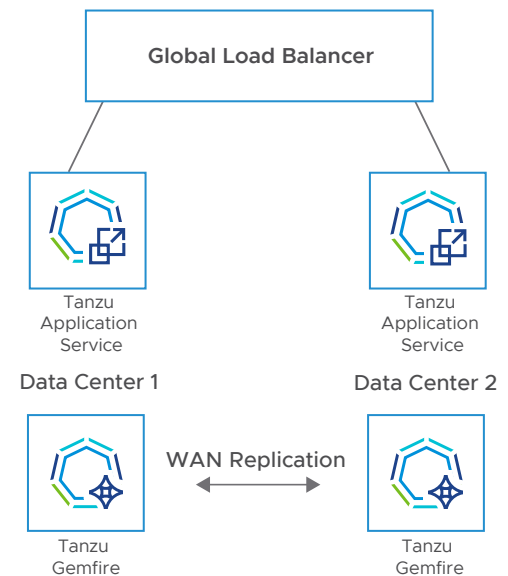


FIGURE 4: Disaster recovery from site/region-wide failures.

## Legacy systems are part of your cloud native evolution

Legacy monolithic applications are notorious for undermining team autonomy. With so many layers of functionality, these applications cause teams to be heavily dependent on each other. On the other hand, these legacy systems are well entrenched, and adequately satisfy many unchanging business requirements. Hence it is not reasonable to undertake a wholesale replacement.

Instead, an evolutionary approach to microservices architecture adoption is both possible and desirable. A new microservices-based system can be created around the edges of the legacy system. This is known as the strangler pattern because it lets the new system grow slowly until the legacy system is strangled and no longer needed.

Building caching layers sourced from the legacy data store(s) provides the starting point for a new generation of microservices. These caching layers mediate between the legacy data store and the microservices that rely on each of the caches as their data layer.

This evolutionary approach embraces legacy systems by protecting and extending investments in them. Existing workloads that are unchanging and well supported by the legacy system can continue to operate against the legacy system until a major change or upgrade is needed, at which point a microservices-based replacement can be considered. Meanwhile, new microservices will benefit from the low-latency, highly concurrent performance, and the event-driven mechanism delivered via the caching layers.

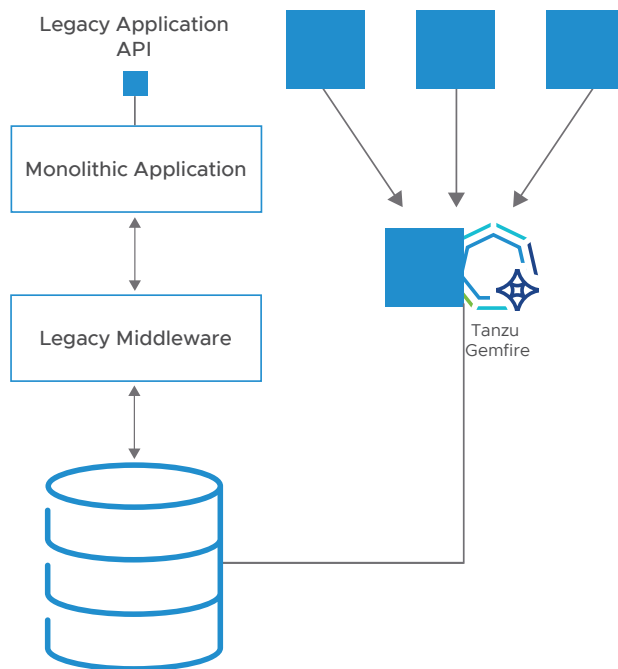


FIGURE 5: An evolutionary approach to embracing legacy systems.

## VMware Tanzu Gemfire for microservices

Tanzu Gemfire is an integrated caching service for the Tanzu Application Service platform. Tanzu Gemfire is available through the Tanzu Application Service Marketplace and can be installed as a service on OpsManager.

Tanzu Gemfire offers an in-memory, key-value store that makes it possible for applications and users to access data at high speed. Tanzu Gemfire was designed for responding to a large volume of concurrent requests while maintaining low latency and high throughput.

Tanzu Gemfire has been designed for easy provisioning of dedicated, on-demand clusters. Developers or application administrators can get started quickly, and adjust ongoing cache capacity based on the changing requirements of their applications and users.

Tanzu Gemfire is delivered as service plans by caching pattern, i.e. as separate, purpose built service plans for the look-aside and inline caching patterns. Each service plan (caching pattern) can be used independently as needed. With these service plans, Tanzu Gemfire speeds up microservices architectures and supports modern DevOps practices.

### Get fast access to data

For accessing data, a common point of congestion is the database. Tanzu Gemfire can increase the available capacity of the system as a whole and remove this source of congestion. Application performance can be improved multi-fold by using Tanzu Gemfire's in-memory cache for the most frequently accessed data. Also, caches can be co-located with the same application instance as the consuming application or user of data to reduce network latencies.

Requests for data are processed with low latency, high concurrency, and high throughput. Tanzu Gemfire can process a high volume of concurrent requests or lookups for specific data elements across a large pool of memory.

Tanzu Gemfire adopts a scale-out architecture. Data is partitioned across servers for parallel processing the data in-memory, and the server clusters can be scaled out as needed. The partitioning of data is automatically re-adjusted when servers are added.

Tanzu Gemfire replicates data to deliver high-availability. Outages related to servers or availability zones can be handled by re-routing requests to replicate data.

### Start quickly with a Tanzu Gemfire cluster

#### Choose the right cache capacity

Tanzu Gemfire is available in several plans to fit specific requirements. Plans can be configured to meet the sizing requirements of the application(s). The VM sizes and disk space have been pre-configured into a simple set of choices. These standard service plan offerings can be further customized to more closely meet requirements.

#### Configure cache clusters fast, on-demand

With Tanzu Gemfire, a developer can create an on-demand cluster in a few steps and just a few minutes, provisioning the VMs for the cluster on-demand. The number of configuration steps and settings are limited because Tanzu Gemfire is pre-configured for each caching pattern, using best practice settings that Tanzu recommends and supports. Developers can freely create clusters, subject only to the limits set by internal operators who allocate capacity and set quotas.

Configuration changes can be done quickly using Tanzu Application Service's standard and familiar command line interface. Similarly, migration between service plans can be quickly achieved via the command line interface. In addition, developers have access to Tanzu Gemfire's own command line interface (gfsh).

### **Access a cache easily, directly from your application code**

The Spring Framework has emerged as an increasingly popular choice for developing applications, particularly applications built using event-driven microservice architectures. In fact, Gartner has recently declared that “Java EE is not an appropriate framework for building cloud-native applications.”<sup>1</sup> In contrast, Spring’s modularity allows developers to pick and choose from an expansive ecosystem of projects for accelerating various aspects of application development. When combined, these projects provide benefits that go beyond the sum of their parts.

Spring Boot is an indispensable project in the Spring ecosystem that lets developers get started quickly by pre-generating the necessary boilerplate code. It also defaults to creating “fat Jars” that contain all of the dependencies for a particular microservice, eliminating the troublesome need for managing disjointed microservice business logic from the Java context within which it runs.

The Spring Cache Abstraction provides a native implementation for common client-side caching patterns using the @Cacheable annotation. Results of calls to legacy systems can be cached with the introduction of these annotations and a simple configuration class. The following guide demonstrates this: <https://spring.io/guides/gs/caching/>

Spring’s other core areas of support for building modern application architectures include Spring Cloud Services, Spring Data Repositories backed by Tanzu Gemfire, and Spring Cloud Data Flow. These and other Spring projects should not be overlooked for the problems they solve, and are a least common denominator rather than a complete list.

The Spring Framework allows developers to focus on business logic by taking care of boilerplate code and other overhead tasks. Developers can build highly scalable microservices using Tanzu Gemfire as a distributed data management platform.

### **Maintain business continuity**

#### **Adopt application state caches as a best practice**

A key enabler of dynamic capacity provisioning and graceful recovery from application failures is the practice of storing and sharing an application’s state external to the application logic. This is a well documented best practice, with support from sources like the twelve-factor app, a set of principles for building cloud native applications. Computed or derived state can also be cached.

As services recover from failure, an application state cache ensures that these services will be able to recover without loss of application state information. A new instance of the service can replace the failed service by reading its state information and immediately responding to requests.

Similarly, capacity increases are easier because new instances of services can access state information and quickly start contributing toward addressing the shared workload across all instances of the service.

Tanzu Gemfire’s plan for the look-aside caching pattern is ideally suited as a high performance application state cache.

#### **Change capacity dynamically, as and when needed**

In a microservices architecture, when microservice instances are added for performance and availability, the underlying data store that services the data requests from all the microservice instances will need to support a high degree of concurrency (simultaneous requests). Tanzu Gemfire can process requests with low-latency, and high concurrency, resulting in a very high throughput system.

Tanzu Gemfire provides logging and metrics that relate to system performance, data that can then be used to recognize when additional caching capacity is needed. When added capacity is required, Tanzu Gemfire can be scaled dynamically without any impact to client applications, and without any downtime.

Tanzu Gemfire has a scale-out architecture. There is no need to pre-provision servers until they are actually needed. Data is partitioned across servers, and the cluster of servers can be scaled up as microservice instances are added. Data is replicated across partitions to facilitate recovery from failures, and for scaling data read operations by leveraging multiple copies of data for reading. Capacity is increased by adding servers, at which point the system automatically rebalances the data across the new configuration.

Partitioning of data across servers is how Tanzu Gemfire can rapidly process large volumes of in-memory data. For example, simple lookups are a common data access pattern for microservices. These lookups support the business logic and inner workings of the microservice, and they involve fast retrieval of a small number of specific data elements (maybe even a single value). Lookups by primary key are sent to the appropriate nodes that serve relevant partitions of data. In many cases, through the highly optimized memory utilization techniques designed into Tanzu Gemfire, partitions may be eliminated allowing the use of a single partition. But when multiple partitions are needed, each partition operates in parallel and the results are then merged and sent back to the client application.

The number of lookups a microservice has to perform is typically very high. For example, consider an insurance application that uses a microservice to answer the question “What is my copay for this procedure if I go to the doctor tomorrow?” The responding microservice would first lookup the patient’s policy by sending a request to the patient microservice. It would then have to lookup the details of the policy by sending a request to the policy microservice. Other required lookups include the coverage, provider, and procedure. All of this would have to be composed to answer the copay query. Retrieving specific data via lookups is at the root of driving your application’s behavior based on data. Optimizing lookup speeds pays dividends because of the sheer frequency with which it occurs.

Tanzu Gemfire can be scaled incrementally and dynamically as and when capacity is needed. Servers and data partitions can be added as data volumes increase. With Tanzu Gemfire, there is no need to overprovision capacity in anticipation of future needs.

#### **Achieve high availability throughout the stack**

Tanzu Gemfire is designed for high availability. The Tanzu Application Service platform and Tanzu Gemfire work hand-in-hand to ensure that high availability is offered at every layer of the stack. The BOSH layer monitors cluster nodes, removes unresponsive instances and restores capacity by spinning up new instances.

Similarly Tanzu Gemfire handles server failures because of how it stores data in partitions, with each partition on a different server, which ensures continued processing and access to data even when a server fails. Data is replicated across servers to facilitate recovery from failures, and for scaling data read operations by leveraging multiple copies of data for reading. As servers are added, the data is automatically rebalanced to adjust to the change.

Both, the Tanzu Application Service platform and Tanzu Gemfire service support the notion of multiple availability zones, which allows smooth recovery from server or availability zone failures. In this scenario, microservice instances are distributed across infrastructure that has been configured to isolate failure boundaries, i.e. availability zones. Tanzu Gemfire is also automatically deployed across availability zones and maps them to its own notion of redundancy zones to ensure that primary and secondary copies of the data are housed in different zones. In Tanzu Gemfire, multiple zones are set up in an active-active configuration so that all resources are being utilized for doing useful work (as opposed to a hot or cold standby configuration). If an entire zone goes down, the system will continue to operate normally and will then compensate by restarting new instances to keep capacity as constant as possible.

## Retain control over capacity allocation and data access

### Create customized plans and quotas

Providing self-serve access to caching services for application development and operations teams via Tanzu Gemfire is a key capability and enabler for highly productive teams, yet left unchecked, resource consumption can spin out of control. Tanzu Gemfire allows operators to configure a range of service plans to fit the needs of their application developers. Capacity consumption can be controlled by setting instance quotas to control the allocation of capacity and costs associated with consumption of capacity. These quotas limit the number of dedicated instances (servers) an individual or a group can create.

Quotas are set at two levels: the maximum number of instances across all the organizations for a plan; maximum number of servers that can be deployed in a service instance for a plan. The scope of these quotas applies to the space in which the service instance is created.

Operators can also control upgrades to the deployed instances.

### Control access to data using role based access control

Access to data can be controlled via authorized and authenticated role based access to clusters. Tanzu Gemfire is pre-configured with two roles and two users out-of-the-box - a developer role and an operator role. The Tanzu Gemfire operator role defines the data model (Tanzu Gemfire regions), while the developer role can read or write data to the Tanzu Gemfire data structures.

## Conclusion

Overlooking the implications for data in a microservices architecture can prevent organizations from reaping the full flexibility and scalability benefits of microservices. Similarly, dealing with the data needs of each microservice in an ad hoc, cottage industry fashion will lead to operational and developer challenges. Companies need to employ design patterns for their data stores to reap the full benefits of a microservices architecture.

The highly distributed nature of microservices-based Architectures introduces questions about how the data layer should be handled. In a microservices architecture, maintaining isolation and autonomy is key, but there are various design strategies to overcome scaling, aggregating and isolation challenges. The performance requirements will dictate the areas where these design strategies should apply.

- An in-memory caching layer brings fast response time for both read and write access to data.
- Asynchronous updates and event-driven architecture protect the autonomy between teams, and allow for high velocity software development.
- The elasticity and scalability of a microservices architecture is inextricably tied to the elasticity and scalability of the data layer.
- Legacy systems can be modernized and carried forward into the world of microservices with the help of a caching isolation layer.

With its reliable event notification features and scalable in-memory performance, Tanzu Gemfire's flexibility can accommodate a range of design options that specifically solve data challenges that can emerge with a microservices architecture. Its integrations with Spring, the de facto framework for microservices, make it ideal for developers.

Visit [tanzu.vmware.com/gemfire](https://tanzu.vmware.com/gemfire) to get started today with a Tanzu Gemfire evaluation license. Let Tanzu help you transform how you deliver microservices!

