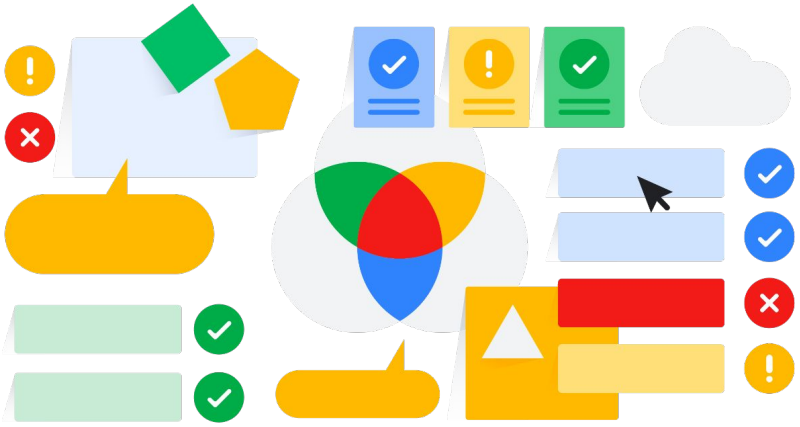
White paper
January 2023

Cloud SQL Architecture Patterns for Microservices



Google Cloud

Table of Contents

Google Cloud

Introduction

What should the database deployment architecture look like for an application that is implemented according to a microservices paradigm? Should it be a single centralized database? Are several databases more adequate?

There are many different alternatives and we provide a rationalization of a database deployment that fits best in a specific application and business context. There is no one-size-fits all database deployment option and therefore various criteria play into the decision process.

We try to help you the application developer to understand the complexity within the context of database deployments, for database architects to rationalize the decision process, and for database administrators to understand implications of the various database deployment possibilities.

There is no one-size-fits all database deployment option

We try to help you the application developer to understand the complexity within the context of database deployments

Chapter 1

Definition of Microservices

What is a microservice?

See a Google description of a microservice here.
There is no single accepted definition of the term "microservice." Different definitions exist and these often depend on the context in which microservices are implemented. However, in order to be able to discuss database deployments in the context of microservices, a working definition is derived in the following.

There is no single accepted definition of the term "microservice"

Martin Fowler, a pioneer in modern software engineering methodologies, avoids defining the term microservice, and instead outlines a microservice architectural style in his Microservices Guide:

Martin Fowler, a pioneer in modern software engineering methodologies, avoids defining the term microservice

"In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."

The key aspects are:

- **Decomposition:** A larger application is built out of multiple services where by each microservice is in general focused on a specific business capability.

- **Independence:** Each microservice is independently deployable.
- **Interface vs. implementation:** Microservices communicate to each other over lightweight mechanisms: typically HTTP endpoints.

We will describe a microservice as a set of operations that is deployed as a single unit. There are different deployment technologies within GCP and those are named below.

Microservice type and instance

The invoice microservice is a microservice type that refers to the implementation of the capability whereas the microservice type can be instantiated several times in execution technology, each being a microservice instance of the invoice type.

Aside from scalability there are additional reasons for service instances, for example, to implement a multi-tenant system, or redundancy in different zones to be ready for failover due to a possible zone outage. The number of instances can be dynamic, for example, a few during regular months, and many more during high volume periods in the year like shopping seasons.

Microservice execution technologies

A microservice is an implementation that exposes one or more operations for example as HTTP endpoints.

In principle, any technology that supports the deployment of code in this form is a possible technology available to run microservice instances.

Currently in many cases microservices are equated with images and containers deployed into Kubernetes or GKE. While this is one option, this is not the only option. Serverless technologies like Cloud Run, Cloud Functions and App Engine are technologies that can execute microservice instances as well. And, of course, microservices can be deployed using Compute Engine and Instance Groups. Some references in the context of various technologies are Microservices Architecture on Google App Engine or Migrating a monolithic application to microservices on Google Kubernetes Engine.

Focusing on the relationship between microservice instances and database deployment architectures, all possible scenarios of microservice execution technologies should be considered.

microservices are equated with images and containers deployed into Kubernetes or GKE

All possible scenarios of microservice execution technologies should be considered

Google Cloud

Chapter 2

Database Properties

Instance is the deployment unit

Functional and non-functional aspects of databases

Using a database for querying is only one aspect. Additional aspects affect deployment architecture as well:

- Data consistency
- Data sharding
- Database size
- Backup/restore
- Distributed transactions
- Fleet management
- Disaster recovery
- Vertical and horizontal scaling

An instance might contain one or more databases

Instance vs. database

Instance is the deployment unit, while database is a data container and does not correspond to a unit of deployment. In this sense the headers of this section are all wrong and probably should not contain “database.” The instance-database discussion is a point of confusion (still after 50+ years of databases) and we need to point that out. For example, you cannot run a database as a container in K8s, but you can in an instance.

A database instance is the computational part of the database as a whole. It includes memory allocation, background processes, communication with networks, and I/O with background processes. An instance might contain one or more databases. This distinction is important because in many cases the term “database” refers to “instance” in reality.

Data consistency and microservices

Maintaining data consistency is a key requirement for enterprise applications. Your microservices applications need to guarantee data consistency or manage it in a predictable manner. In other words, microservices will need the support of various consistency options depending on their use cases.

Microservices database architecture by nature is distributed. In monolithic applications transactions are written to a central shared database. In microservices architecture the databases are deployed in a distributed manner with each database responsible for its own transaction management and consistency. In some cases the databases may be non-relational (NoSQL) databases.

Maintaining data consistency is a key requirement for enterprise applications

Chapter 3

Deployment Architectures

A single schema approach lends itself to a monolithic type of architecture deployment

List of Database Architectures

The different architectures and methods for deploying microservices in Cloud SQL databases include the following:

- **Instance** - A single instance per microservice. For example, an order management system with services for order tracking, inventory management, shipping, and payment processing would have a separate Cloud SQL instance for each of these services.
- **Database** - A single database per microservice. In this case, microservices share an instance. An internal enterprise system may be suited for this type of architecture as all the services may share similar SLA and requirements that can be provided by a single instance.
- **Schema** - A single schema per microservice. Microservices share a database and instance. An example use case for this architecture would be small departmental systems implemented as microservices that would undergo only small changes over time.

Chapter 3

Deployment Architectures

Single instance and database per microservice architectures are easy to deploy and provide isolation

List of Database Architectures

- **Tables** - Technically, the tables for every service architecture could be deployed within a single schema. However, this single schema approach lends itself to a monolithic type of architecture deployment from day one and does not align with the goals of adopting microservices implementations. It is vulnerable to the same decisions that lead to the complexities of monolithic architectures over time. For example, implementing procedural language or complex database views within the database that would be shared by multiple services.

  The Table approach could be considered for systems consisting of only a small number of microservices such as a booking system or reservations system with strict controls at the application or services level to keep tables separate eliminate potential complexities over time.

- **Hybrid** - A combination of Instance, database, or schema level separation with microservices grouped based on function and criteria (listed below). For example, in an online order or retail system the logging service could be in a separate instance due to resource needs, the order service in a separate database for performance, and the customer and address services in the same database but separate schemas for security.

List of criteria

Applications of different types and use cases have differing database requirements and criteria for deployment. You should analyze the list of possible criteria and top priority for your application to determine which database architecture will best suit the needs of your application. The following is a list of criteria that should be considered and prioritized in each case to determine the architecture needs. For example, for a healthcare application, isolation, security, and availability might be the most critical and should be considered as top priority to evaluate the architecture needs of such an application. In such a case deploying each service in a separate Cloud SQL instance may best meet the needs for isolation.

- **Isolation** - The extent to which a microservice needs to operate without affecting the data and operations of other microservices or in turn be affected by similar operations on other microservices. For example, upgrading the database platform to affect a new feature without affecting the data or databases of other microservices.
- **Agility** - Ease of provisioning, deprovisioning or scaling the database.
- **Configuration** - The flexibility to update the configuration of the database, instance, or schema without being restricted by or impacting other services.
- **Operations and Maintenance** - Can the database for a service be backed up, restored, patched, upgraded independently.
- **Scalability** - Ability and ease of scaling the database for each service.
- **Performance** - Can performance be managed independently, without impacting other services, and with ease.
- **Security** - Applying Security consistently according to the principle of least privileges or as needed by each service using the tools and services provided.
- **Availability** - Can Availability be deployed independently for each service and can it be managed in isolation.

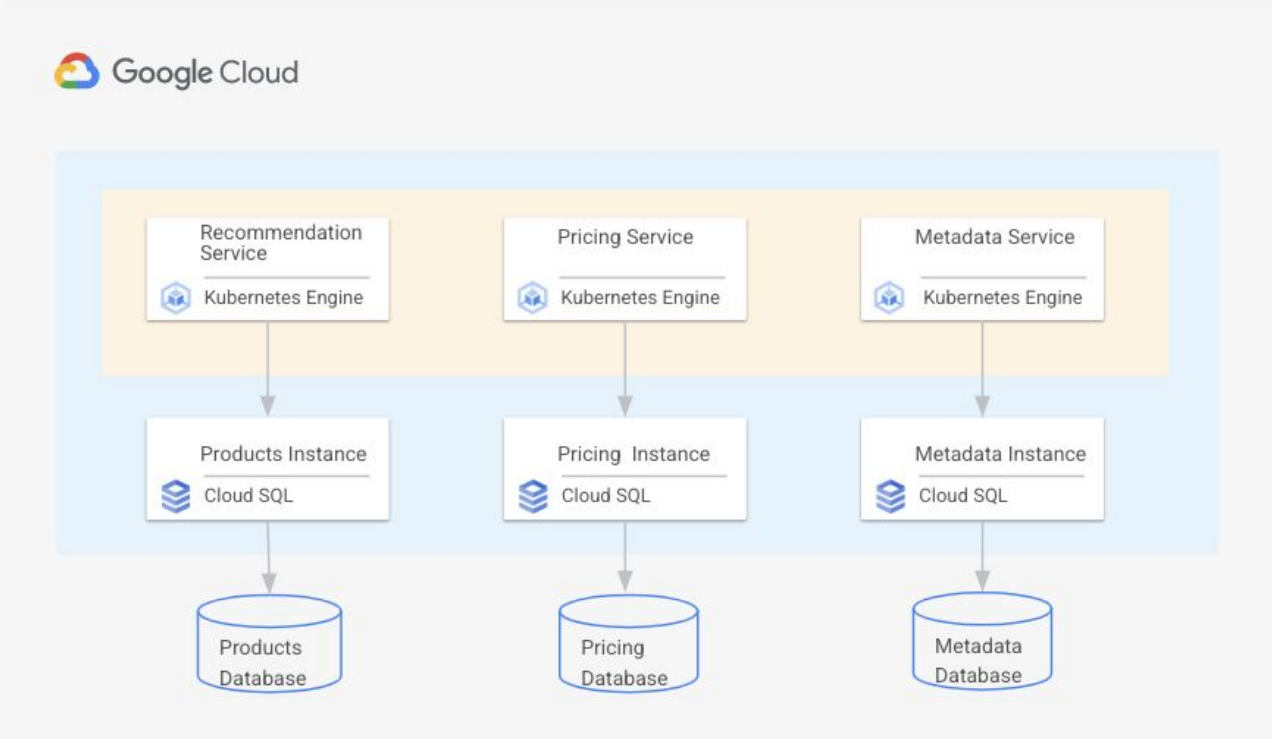
List of criteria

- **Recovery Time Objective (RTO) / Recovery Point Objective (RPO)** - Architecting just the right solution that meets the objective for each service without over or under allocating resources.
- **Cost** - Cost implications and efficiency. Cloud SQL is billed for each instance, storage, CPU and memory. Increasing density can reduce costs.
- **Regulatory and Compliance requirements** - Regulatory and Compliance requirements may dictate that services are deployed in certain regions or zones, or that they are completely isolated and secured in a certain manner.

The different database architectures supporting a microservices type solution are described below:

Instance - Each microservice is deployed in a separate Cloud SQL instance

In this architecture each microservice persists data in its own Cloud SQL instance. Cloud SQL (PostgreSQL or MySQL) can have multiple databases per instance. However a single database within an instance is deployed for each microservice.



Microservices architecture

Isolation:
Provides the greatest level of isolation. Databases and Instances do not share resources and can be managed, maintained, and deployed separately.

Agility:
Cloud SQL Instances are easy and fast to provision and deprovision. Compared to the other architectures this requires the most effort and time. The number of steps involved and the configuration required is commonly automated for efficiency and consistency. Instance provisioning and de-provisioning can be automated using Terraform.

Configuration:
Each microservice database can be configured independently at the database or instance level. For example, database flags can be changed at the instance level specifically to meet the requirements of the service. The database can be tuned without undue consideration for the impact on other databases or services.

Operations & Maintenance:
Any operation can be performed for a database or service independent of and without affecting another service. For example, a database for one service can be patched or upgraded without affecting other services with regards to downtime, testing or functionality.

Scalability:
A microservice database can consume the entire instance. As a result there are no restrictions within the instance itself and the database service scales to the full capabilities of a Cloud SQL instance.

Performance:
There is no resource sharing within the instance and therefore there is no possibility of resource contention between the databases or noisy neighbor effect. The database for the service can perform to the full capability and scale of a single instance.

Google Cloud

Security:
Data for each microservice can be secured separately and different levels of access provided for each function or user. Data access is not shared by the services. Additionally, data can be easily secured with access provided only through the services themselves via service APIs.

Instances can be configured separately for each service and deployed in different Data Centers or Regions depending on regulation and compliance requirements.

Availability:
Instances can be configured for different levels of availability depending on the requirements for the service. Cloud SQL provides HA and non-HA configurations for each instance. Services that do not require 99.95% availability can be configured without HA.

Recovery Time Objective (RTO) / Recovery Point Objective (RPO):
Services with different RTO and RPO requirements can be configured differently at the database instance level. For example, a micro service that requires near zero RPO can be configured with Point-In-Time-Recovery, and a service that requires a longer RTO across zones can be configured without HA at the instance level.

Cost:
Allocating a single instance for each microservice would incur a higher cost relative to other architecture options. Cloud SQL instances are vertically scalable. However, sufficient vCPUs and memory must be initially assigned to account for small increases in demand and sudden spikes in load. Factor in possible over provisioning and idle instances.

Cloud SQL reduces your database instance costs and allows you to optimize by providing automatic discounts with Commited Use Discounts (CUDs), allowing you to provision custom instance sizes, providing per second billing, not charging for extra IOPs and leveraging the Active Assist feature . This is on top of normal cloud benefits such as the ability to shutdown idle instances , scale up or down on demand to reduce overprovisioning, and pay for only what you use.

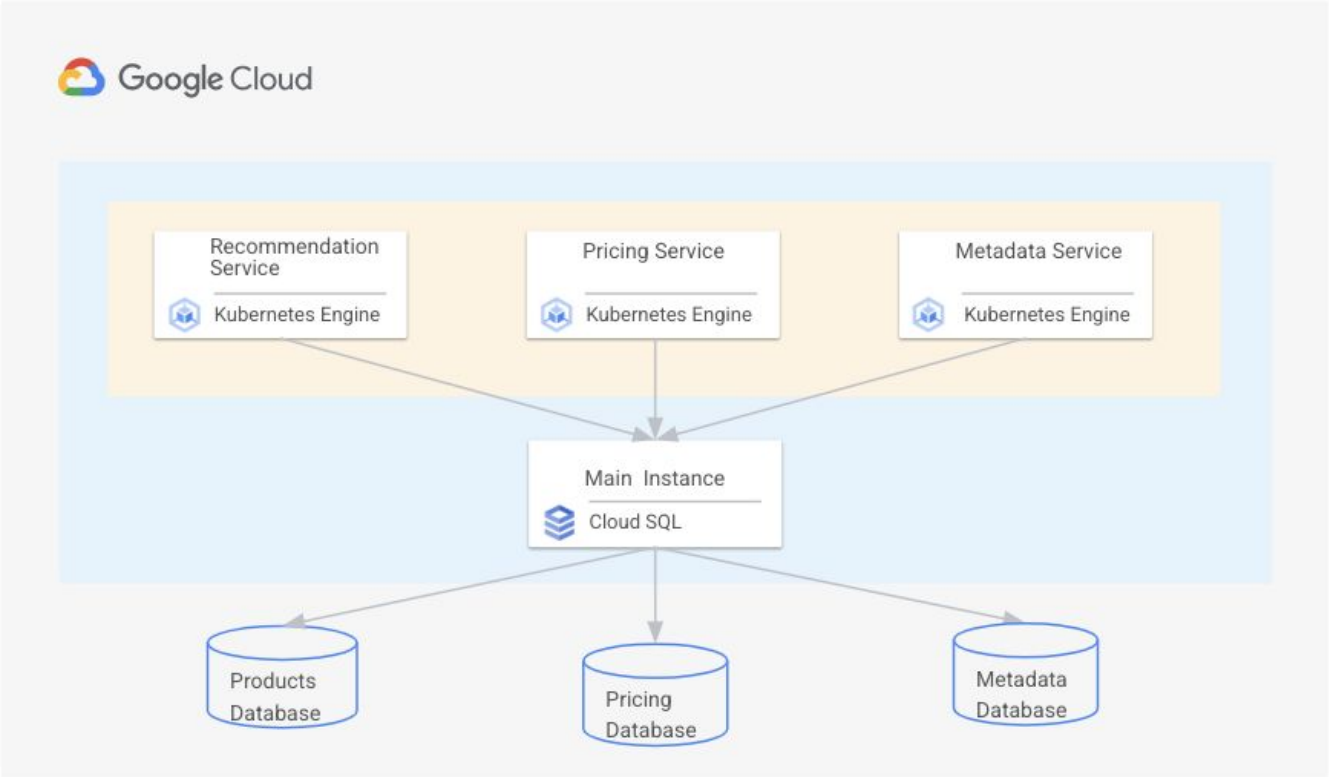
Cloud SQL reduces your database instance costs

Google Cloud

Regulatory and Compliance requirements:
Cloud SQL allows instances to be configured for different regions. They are also configured separately to the security and availability requirements to meet various Regulatory and Compliance needs. With a per instance configuration these requirements are automatically deployed for each specific service.

Database - Each microservice is deployed in a separate database within the same instance

Microservices in this architecture can share a Cloud SQL instance but are deployed within a single isolated database. Cloud SQL (PostgreSQL, MySQL, SQL Server) can have multiple databases per instance. This potentially maximizes the density and the utilization of resources allocated to each instance. This architecture leverages the feature of Cloud SQL database engines that allows multiple databases per instance as shown in the diagram below.



Microservices architecture

Google Cloud

Isolation:
Complete isolation at the database level. However databases share the same resources in a Cloud SQL instance (memory & CPU). Configurations such as flag settings potentially affect all databases, including operational activities.

Agility:
Cloud SQL databases can be easily provisioned and deleted from an instance. This process would generally be faster compared to when an instance must be created first. Database provisioning and deletion can be automated and included in the CI/CD pipelines.

Configuration:
Databases for each service can be configured at the database level. For example, connections. Configurations done at the instance level affect all databases. Most configurations for MySQL, PostgreSQL are done at the instance level.

Operations & Maintenance:
Operations can be performed at the instance level for multiple database services. This will provide ease of operation and scale in cases when the same operation must be performed for all databases. For example shutting down the databases or patching the instance. This does not provide isolation in some cases. However, operations such as backup and recovery can be done at the database level when only some databases must be affected.

Scalability:
Databases share the same resources within a Cloud SQL instance. Each database will be limited to the available resources available up to the maximum Cloud SQL resource allocations (e.g. CPU, memory, and storage).

Performance:
There is resource sharing within the instance and the possibility of resource contention between the databases. For example, databases needing to access CPU resources at the same time or the Write Ahead Log (WAL) processes simultaneously. Overprovisioning of capacity or over allocation of processes is required to mitigate conflicts.

Google Cloud

Security:
It is easy to securely configure all databases at the instance level. Conversely a lapse in the configuration of the security or a delay in patching can affect all databases. Database level and object level security can be configured separately to isolate each database.

Availability:
Database and the microservice that they support share the same level of Availability. For example if the Cloud SQL instance is configured for HA then all databases share the HA configuration. Whenever the instance fails over to another zone all databases will be protected.

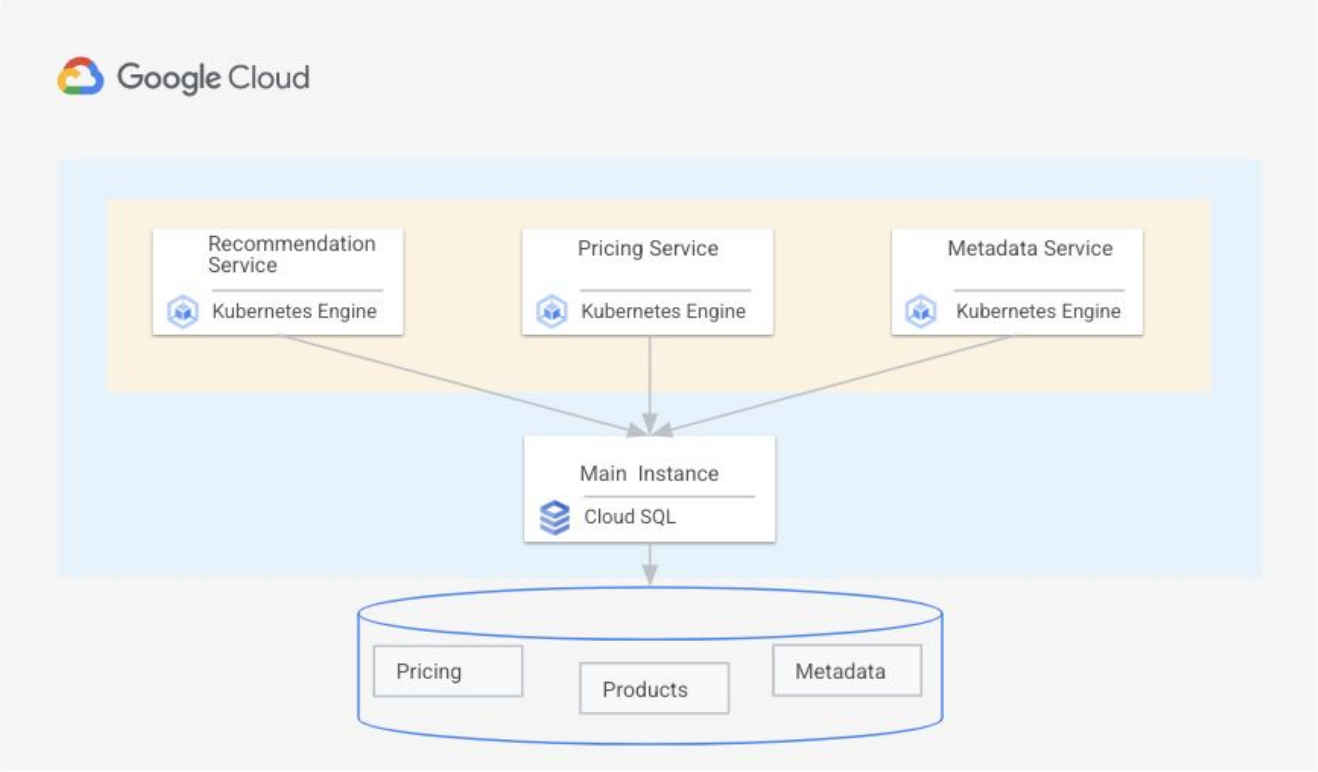
Recovery Time Objective (RTO) / Recovery Point Objective (RPO):
RTO and RPO are mostly configured at the instance level. Services will share the same level of RTO and RPO. However, databases can be backed up or replicated independently for some special scenarios.

Cost:
Allocating a single database for each microservice increased the density of each Cloud SQL instance. Resources allocated are shared and used more efficiently. This can reduce the overall cost of Cloud SQL across all microservices.

Regulatory and Compliance requirements:
Cloud SQL allows instances to be configured for different regions. Microservices for databases within a single instance will share the same Regulatory and Compliance configuration. Efficient to configure all databases and microservices to meet the same Regulatory and Compliance requirements.

Schema - Each microservice is deployed within a separate schema within the same database

In the Schema per service architecture the microservice is deployed and constrained to only a single schema within a database. All Cloud SQL platforms allow for the creation of multiple schemas within a database. This architecture leverages this feature and ensures that the objects (tables, views, indexes) for the microservice are constrained to a single schema. Monitoring and controlling technical depth in this architecture is particularly important since it is easy to create SQL statements that cross schemas belonging to multiple microservices. Over time this could lead to the deployment developing into a monolithic architecture where services are tightly coupled and expensive to update and deploy independently.



Microservices architecture

Google Cloud

Isolation:
Logical isolation for database objects (tables, indexes, views) is provided within the database at the schema level. Each schema shares a single database and instance. Operations and changes at the database and instance level affect all schemas and hence the services supported by those schemas. Additionally, resource consumption or changes within each schema can affect other schemas or services.

Agility:
Schemas can be quickly created and deleted within a database and instance that was previously provisioned. Database provisioning and deletion can be automated and included in the CI/CD pipelines using tools such as Liquibase.

Configuration:
Configuration at the database and Instance level affects all schemas. For example, connections or database flags. Most configurations for MySQL, PostgreSQL are done at the instance level. Services requiring specific database configurations must be deployed in separate databases or instance.

Operations & Maintenance:
Most operations in Cloud SQL are performed at the database or instance level. This will be efficient and scalable for Operations that need to be performed for schemas supporting all microservices. For example, exporting data. The schema per service architecture does not provide isolation in some cases. If the number of connections to the database must be restricted, this will affect all the schema and therefore every service deployed in the database.

Scalability:
Schemas share the same resources within a database and a Cloud SQL instance. Each schema will be limited by the amount of resources available and remaining within the database.

Performance:
There is resource sharing within the instance and the possibility of resource contention between the processes accessing each schema on behalf of individual services. For example, connections from different services needing to access CPU resources at the same time or the WAL processes needing to replicate transactions from multiple services simultaneously. Overprovisioning of capacity or over allocation of processes is required to mitigate conflicts.

Security:
It is easy to secure every service at the database or instance level. Conversely a lapse in the configuration of the security or a delay in patching can affect every service. Security can be configured at the schema and object level separately to isolate each service data.

Availability:
Each schema and the microservice that it supports share the same level of availability. For example, if the Cloud SQL instance is configured for HA then all databases share that HA configuration and SLA. Whenever the instance fails over to another zone every service will failover and be protected.

Recovery Time Objective (RTO) / Recovery Point Objective (RPO):
RTO and RPO are mostly configured at the instance level. Services will share the same level of RTO and RPO. However, databases can be backed up or replicated independently for some special scenarios.

Cost:
Allocating a single database for each microservice increased the density of each Cloud SQL instance. Resources allocated are shared and used more efficiently. This can reduce the overall cost of Cloud SQL across all microservices.

Regulatory and Compliance requirements:
Cloud SQL allows instances to be configured for different regions. Microservices for databases within a single instance will share the same Regulatory and Compliance configuration.

This choice is efficient for configuring all databases and microservices to meet the same Regulatory and Compliance requirements.

Consistency as exclusive or most important requirement

If data consistency is the only, or the most important requirement, what would that mean? Data consistency means that all data that needs to be consistent can always be modified in a single database transaction. There is an algorithm to determine the number of databases if consistency is paramount: adding a database and storing some data into it would cause consistency issues as it would not be located in the same database anymore that has related data that needs to be consistent with it.

Role and impact of global data

When global data is present it has a major impact. As soon as global data is involved that is part of every transaction (e.g. writing a history table, or counting transactions, or implementing a change data capture in a table) then this requires a focused evaluation. If consistency is the only requirement, then global data forces a single database. If consistency is optional from a database perspective (aka, application has to implement it) then global data can be located in any database.

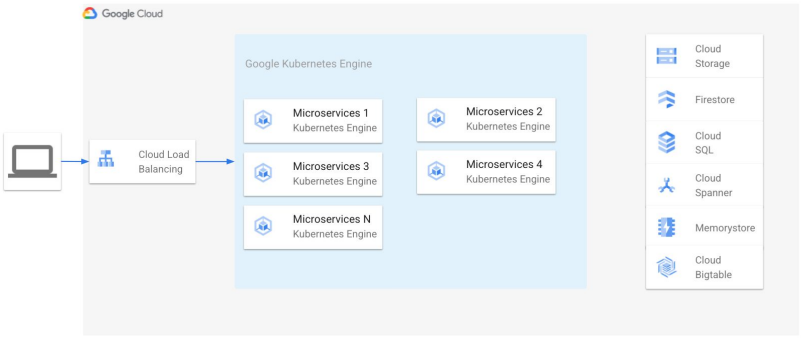
Chapter 4

Architecture Selection

How do you select a database architecture for your microservices?

Multiple data formats and service requirements

Deploying a microservice architecture can require different types of database services depending on the service that is being deployed. The types of database services could include relational databases for OLTP, NoSQL databases for key-value or semi-structured data, document stores for storing JSON formatted data, low latency open source databases or highly scalable multi region distributed databases. Google Cloud offers a portfolio of database services to meet your database requirements. As an example, a microservice supporting a mobile application may store data in a document format requiring the scale and availability of Firestore. Another service may need to store transactions at large scale with ACID compliance requiring the scalability of Cloud Spanner. The diagram below illustrates what such an architecture may look like with each database service possibly leveraging the available database architecture types (instance, database, schema, table, document) for multiple microservices within that database service.



Deploying a microservice architecture can require different types of database services

Google Cloud offers a portfolio of database services to meet your database requirements

Google Cloud

Spectrum and deployment patterns

Throughout our work with customers around the world, we have surfaced a few architectural patterns utilized by customers to support their application modernization and microservice strategies.

- **Isolated, single database per microservice** where there is one physical instance per service.
    - Example: One PostgreSQL DB deployed per microservice.

- **Queue backed microservices** where a service reads and writes to a queue.
    - Example: A pub/sub queue is backed by a database and each service maintains its own persistence tier in the application.

- **Shared service, single database per microservice** where there is one logical database per service within an instance.
    - Example: PostgreSQL database created within a PostgreSQL server for each microservice.

- **Shared service, single database for all microservices** where there is one physical instance and database serving all the microservices.
    - Example: Spanner in support of the microservices.

Google Cloud

Summary Guidance

Cloud SQL provides seamless integration to easily deploy microservices on GCP. Built-In Security, High Availability, Easy Provisioning and integration with services such as GKE allows customers to easily deploy new microservices and migrate from complex monolithic services.

Leveraging various architecture types available with Cloud SQL provides a choice of options based on priorities. For example, if the main goal and top priority is isolation then that will lead us to choosing the Microservices per Instance architecture pattern as it provides the most isolation. However, if cost and computing resource efficiency is a top priority then choosing the Schema per service architecture pattern may be the best choice provided the trade off of lower agility is palatable.

The process of choosing the right pattern for your application or organization may leverage a matrix such as this comparing the top priorities:

Priority		Instance	Database	Schema
1	Regulatory and Compliance requirements	✓		
2	Isolation	✓		
3	Security	✓		
4	Performance	✓		
5	Agility	✓		
6	Operations and Maintenance			✓

Deploying multiple databases within a single Cloud SQL instance can provide the most density and resource utilization. This architecture type will also provide upfront cost optimization without introducing operational complexity and technical debt. Deploying each microservice in a separate Cloud SQL instance will provide the most flexibility, agility, scalability, and best performance in the long term.

Cloud SQL provides seamless integration to easily deploy microservices on GCP

Deploying each microservice in a separate Cloud SQL instance will provide the most flexibility, agility, scalability, and best performance in long term

Google Cloud

Customer References and Examples

Auto Trader (UK) : Migrated from Oracle to PostgreSQL, and using microservices.
"Our architecture consists of around 500 logical services," says Karl Stoney, Technical Architect at Autotrader. "Around 140 of those are already in production on GCP, and we plan to migrate the rest over the next year so everything is built, managed, and monitored in the same way."

H-E-B: Using Cloud SQL and microservices as part of mainframe modernization. H-E-B, like many enterprises, is moving away from legacy mainframes for microservices and public cloud infrastructure. With hundreds of applications powering their 100+ year-old business, H-E-B needs to be confident that the platform they are building will provide them the agility and security to continue to innovate for their customers. View the video to learn how the H-E-B engineering team started breaking down their Curbside and Home Delivery monoliths into microservices, why they chose to make Kubernetes a first-class citizen, and why they're leveraging Anthos as a hybrid cloud platform.

Google Cloud

References

1) Designing, building and deploying microservices architecture
2) Scalable commerce workloads with microservices
3) Modernization path for .NET applications on Google Cloud

Google Cloud