CHAPTER 3

Cloud Native Architecture Principles

Cloud native principles define the underlying general rules and guidelines for the use and architecture of your system. They reflect a level of consensus for the various elements of your system and enterprise and offer a basis for making future decisions. The principles are typically created at the same time as the architecture is defined.

Without architecture principles, your enterprises has no compass to guide its journey from its current state to its future cloud native state and no standard way to measure its progress.

In this chapter, we will cover the following principles; some have existed for three to four decades, but they are still very much relevant in modern cloud native architecture and design.

- Orthogonal architecture principles such as coupling and cohesion
- Principles such as KISS, DRY, isolate, encapsulate, group-related function, use layering
- The SOLID design principles such as single responsibility, open-close principle, Liskov substitution, interface segregation, etc.
- Modern architecture principles such as automated deployment, no single point of failure, polylithic and polyglot, API first, event-driven, choreography, etc.
- Cloud native architecture principles such as infrastructure independent, location independent, resilient to latency, etc.
- Development principles such as shift-left testing, shift-left security, containerization, infrastructure as code, agile, etc.

S. R. Goniwada, *Cloud Native Architecture and Design*, https://doi.org/10.1007/978-1-4842-7226-8_3

What Are Architecture Principles?

A principle is a law or rule that is usually followed when making key architecture decisions. It's important to note that principles are not commandments; exceptions are acceptable when necessary.

Architecture and design principles play a critical role in guiding the software architecture work that includes defining an enterprise's future direction and the transitions it needs to reach the future state of architecture. The principles are usually created at the beginning of the architecture definition and are reviewed and ratified by the architecture board. While defining principles, you need to align with the existing enterprise's principles.

Architecture and design principles define the fundamental assumptions of the IT organization when creating and maintaining the IT capabilities. Without principles, IT projects have no compass to guide their journey. Without a common set of principles, the executives in an IT organization will be left on their own to determine which projects will be funded, which assets will be leveraged, which cloud model will be used, etc.

It is useful to understand the definition of various architecture and design principles. In addition, you need to understand the associated rationale and implications of these principles. The most important step is to promote these principles across all the stakeholders and development teams so that the adoption of the principles will achieve the desired result.

Architecture and design principles are usually developed by architects and designers, in conjunction with various key stakeholders, and all the defined principles must be clearly traceable and clearly articulated to guide the decision-making.

According to TOGAF,

> *"A good set of principles will be founded in the beliefs and values of the organization and expressed in language that the business understands and uses. Principles should be few in number, future-oriented, and endorsed and championed by senior management. They provide a firm foundation for making architecture and planning decisions, framing policies, procedures, and standards, and supporting the resolution of contradictory situations. A poor set of principles will quickly become discussed, and the resultant architectures, policies, and standards will appear arbitrary or self-serving, and thus lack credibility. Essentially, principles of driver behavior."*

These are six criteria to distinguish a good set of principles:

- *Understandable*: The principles should be written in plain language that is easy to understand.
- *Robust*: The principles should enable good-quality decisions about the architecture and plans.
- *Complete*: The statements must be accurate and complete.
- *Consistent*: All the principles must be consistent and work together.
- *Stable*: The principles should be enduring and accommodate change when required.
- *Resilience*: Failure is unavoidable in systems; these principles enable companies to self-heal quickly from difficulties.

Cloud Native Design Principles

The following sections cover cloud native design principles.

API First Principle

Using an application programming interface (API) is not a new approach in IT, as APIs have been used in IT for more than 20 years. But APIs were limited to specific internal applications.

The *API first principle* is the de facto principle of modern architecture. Every application is designed and developed with the API first principle. This principle allows all implementation details to be exposed through APIs to the consumers and encourages the application design and development teams to have resources accessible through REST HTTP interfaces.

The API first approach means designing an API so that it has consistency, as well as adaptability, regardless of the type of projects. The API first principle is as follows:

- The API is the first user interface of an application.
- The API comes first and then the implementation.
- The API is described.

- The API is contracted between the provider and the consumer.

For example, let's say client 1 and client 2 are two client-facing applications, as shown in Figure 3-1, and interact with various users by consuming its implementation in services A, B, and C through APIs via API management. The APIs are contracts between clients 1 and 2 with services A, B, and C.

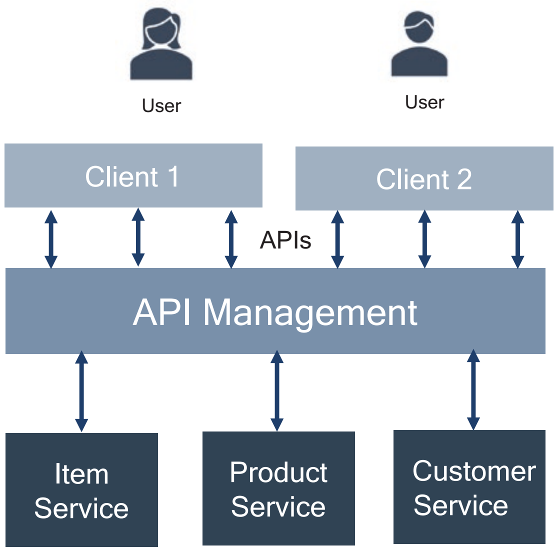


Figure 3-1. *API management*

These are the benefits of the API first principle:

- *Development teams can work in parallel*: API first involves establishing the contract. Creating a contract between services that are followed by the team across enterprises allows those teams to work on multiple APIs at the same time.
- *Reduces the cost of developing an application*: The reusability of the API first approach allows code to be recycled from project to project so that development teams always have a baseline architecture with which they can work.
- *Increases speed to market*: Automated discoverable APIs have the ability to be discovered quickly and automate development with readily available tools like Swagger.

- *Improved developer experience*: The consumers of APIs are most often the development team. API first ensures the developers have a positive experience using APIs.
- *Reduce the risk of failure*: The possibility of error is greatly reduced due to the inherent reliability and consistency of the design and implementation.

Monolithic Architecture Principle

The *monolithic architecture principle* (MAP) is building the architecture as a single unit with a single codebase. Most applications in an enterprise are based on this principle because enterprises have been using this approach for ages. Sometimes these applications are called multitiered applications and use the Model-View-Controller (MVC) pattern. The monolithic architecture can expose APIs to the client applications and also focus on desktop/laptop devices with a web browser as a client, as shown in Figure 3-2.

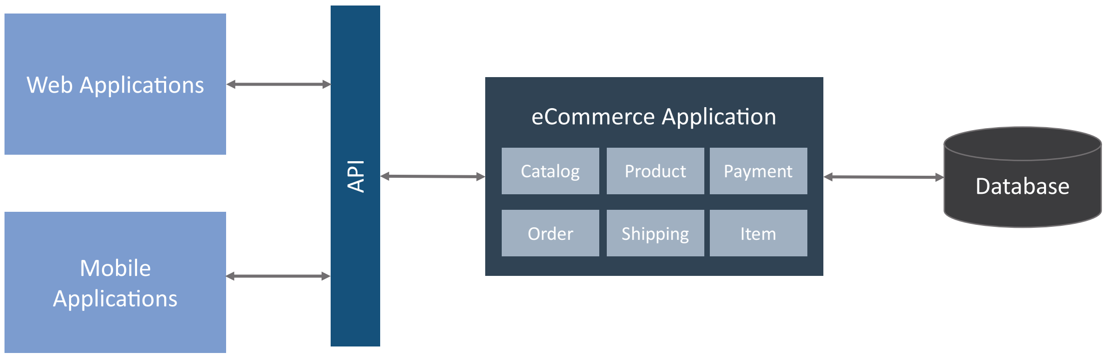


Figure 3-2. *Monolithic application*

The following are the drawbacks of monolithic applications:

- Scaling a monolithic application is a challenge.
- It is difficult to embrace agility.
- Monolithic applications require more infrastructure due to scaling the entire application irrespective of load.
- Monolithic applications are not business friendly, do not support business disruption, and are slow to market.

Polylithic Architecture Principle

The polylithic architecture principle (PAP) provides a different variant of microservices. Each microservice provides domain functionality. These separated modules are consolidated through several programming techniques. This principle refers to a technology-agnostic approach of building systems as a composition of multiple mini/microarchitectures for the granular subsystem.

The PAP simplifies your back-end services and tools by enabling you to construct them as modular monoliths using composable components.

In the polylithic principle, you create a domain-based service by using a domain-driven design methodology. Most communication within the polylithic system is done using industry-standard communication protocols.

Applying the Polylithic Principle in Architecture

An e-commerce platform, as shown in Figure 3-3, will deal with many types of business functionality instead of trying to implement all these business use cases in one programming language. For example, for the parallel processing use case, implementing parallel techniques in functional programming is better than object-oriented programming.

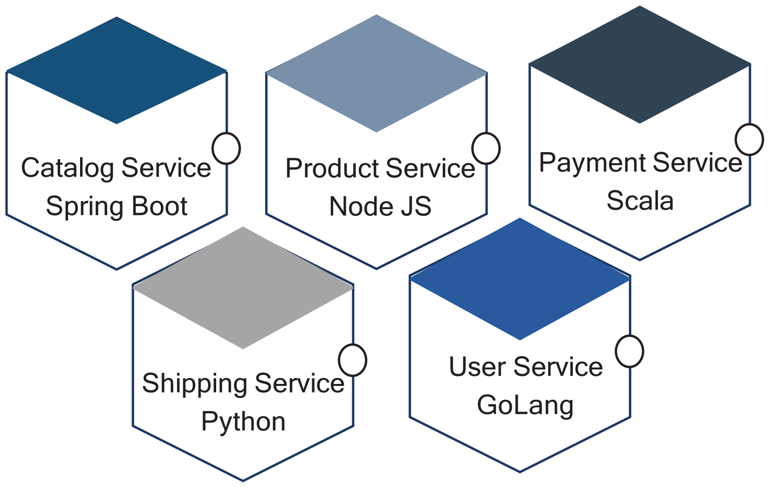


Figure 3-3. *Microservices with polylihic programming languages*

Properties of Polylithic Principles

The simplicity of a domain-based service makes for good building blocks of code. But the architecture approach will be incomplete without a discussion about the essential properties that enable, deliver, and sustain operations.

- *Encapsulation*: Services hide their implementation and expose only their signature.
- *Simplicity*: Services have a single responsibility.
- *Stateless*: Services are just code; they don't contain state or instances.
- *Purity*: Services can be pure, which makes them easy to understand, reuse, test, and parallelize.

The polylithic principle refers to an approach of a building system as a composition of multiple granular subsystems, each of which has its specialized architectures selected to suit specific needs on a best-fit basis.

- Each granular subsystem will be housed in its container environment and isolated from other subsystems.
- Each subsystem will take exclusive ownership of data and provide access through a well-defined published interface.

To support change management, polylithic principle will also make backward compatibility and interface versions aware of first-class architectural concerns, which means that each subsystem will support the coexistence of multiple versions of the same service.

Polyglot Persistence Principle

Neal Ford coined the term *polyglot* in 2006 to express the idea that applications should be written in a mix of languages to take advantage of the fact that different languages are suitable for different problems. The polyglot persistence principle is about is choosing the way data is stored based on the way data is being used by individual applications. In short, you need to pick the right storage for the right kind of data.

Applying the Polyglot Persistence Principle in Architecture

Let's take the example of Martin Fowler's ecommerce application, as shown in Figure 3-4 (Amazon, Flipkart, JioMart, etc.), that can be broken down into many microservices such as catalog, user, audit, inventory, etc. Storing all this data in one single monolithic database would be a nightmare. Instead, use the appropriate database technologies for the respective use cases.

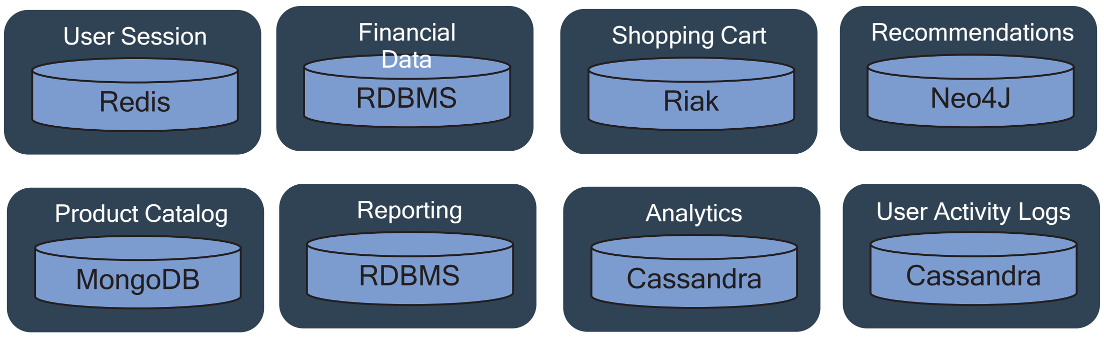


Figure 3-4. *Polyglot persistence*

Modeled with Business Domain Principle

The *modeled with business domain principle* (MBDP) is about using domain-driven design (DDD), which will be explained in Chapter 10.

DDD is an approach for developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concept.

DDD is needed to decouple the existing system that you do not have any knowledge of or a large enterprise with a complex map of departments and systems, for which you are asked to implement a solution that is coherent and works seamlessly.

When you are applying this principle, follow the nine steps shown in Figure 3-5 of event storming to identify the microservices.

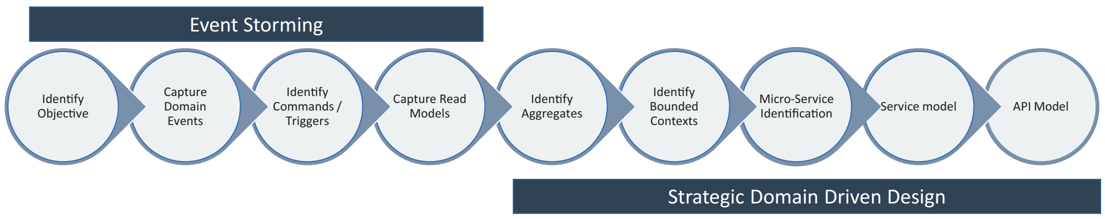


Figure 3-5. *DDD with event storming*

When you are identifying microservices by using this principle, some designers try to separate the parts by business domain and domain entities, users, and individual requests from the UI, but this leads to your design becoming data-oriented and technical-centric and doesn't help you to design your microservices across business capabilities. Always think of each request or service as a collection of capabilities.

Consumer First Principle

The *consumer first principle* (CFP) is about designing your API services to the consumers, before starting any design activity. The first thing is that you need to analyze what a consumer wants. It is not about consumer rights advocacy, but it's about recognizing that when we create all these services for the consumer, the services need to be called by all types of consumers.

Before the start any services design, you need to ask questions like, do you know who your consumers are? Do you know where they are in an organization? Do you have any collaboration with which you can interact? In nutshell, you need to have the full request details before initiating a design.

Every design starts with the basics, meaning consumer-driven contracts, as those contracts define for your consumers the process in your microservices.

Next, you need to decide on standards and consistency across all APIs. In the consumer-first approach, each person or design team defines their APIs differently. Some teams define them with nouns, some teams define them with verbs, and some teams handle user search, error handling, and pagination in different ways. To address this, you need to define an organizational standard for API design that will be useful for API governance and operations.

You need to make sure you define the documentation for APIs. For anyone in a client organization who wants to consume an API, knowing what the API does is important. There are many ways to create a document, but nowadays developers use Swagger effectively to design and document an API. It allows you to define metadata about API endpoints and expose them in multiple ways. There are various tools available to annotate metadata on your endpoints and have exposed Swagger documentation. For better traceability between consumer requirements, design, and documentation, integrate Confluence, JIRA, and Swagger or create a developer portal if you are using API management software.

Decentralize Everything Principle

The *decentralize everything principle* (DEP) is about providing self-direction, self-sufficiency, and self-reliance to cloud native development, deployment, and governance; this provides much freedom to the development community to think, develop, and deploy each service.

When you're thinking about decentralization, you need to provide autonomy to solve the problem without necessarily having to coordinate with lots of other people, but coordination is required but not at the extent of snatching freedom from the core problem-solving team. Teams building microservices prefer a different approach to standards too, rather than using a set of the standard defined by a centralized team. Netflix is a good example of an organization that follows this philosophy.

In software development, the decentralization to be adopted is as follows; note that not every problem is a nail, and not every solution a hammer:

- Decentralize microservices that are isolated from other microservices to help the team to achieve concerns such as testability, extensibility, scalability, etc. Apply domain-driven design and bounded context to decentralize domain-based microservices.
- Deploy microservices independently on any environment without affecting other services, use containers, and use Kubernetes technology by using infrastructure as code.
- Decentralize governance, popularized by Amazon. This promotes innovation and speed to market.
- Decentralize DevOps, and let each team have its pipeline with the various self-service tools. Centralizing the tools doesn't help.

- Decentralize data management, and use the polyglot principle to decentralize data for each microservice. But be cautious about licenses, data dependency, transactions, etc.

In the end, you need to know that decentralizing everything doesn't ease your problem. Sometimes it can get out of control. To mitigate this, you need to audit each team regularly.

Culture of Automation Principle

The *culture of automation principle* (CAP) states that it's imperative for organizations to first create a foundation that is conducive for automation. Automation must be threaded into the company culture and fully embraced across the business at all levels.

Apparently, 75 percent of an IT professional's time is spent "keeping the lights on," with the remaining 25 percent focused on innovation that moves their business forward. Everyone wants to flip those percentages.

Look at how Netflix, Amazon, Google, etc., are embracing cloud native, and the time it took them to get up to speed moving from a few hundred services to thousands of services into production: all of that work was centered around the culture of automation, tooling, and discipline. A few teams in your organization are probably pretty good at automating their everyday work, but the challenge is to apply a culture of automation across the entire enterprise so that the organization can drive toward the common goal of developing applications faster and more efficiently.

The most important thing in automation is developer mindset and quality; when developers check in the code in SCM tools, they should be confident that the code can go into production. Modeling the process from check-in through production, I get my release candidate, move my code through my pipeline, and think it's good enough for the build; if it fails a test, I move the next version through, and hopefully, I can move it into the production environment. This sort of automation and visibility of the quality of the software is key in enterprises because we want to move software as quickly as possible without human intervention. Once base automation available, you can leverage the AI-driven development principle to take it further and develop a foundation for streamlining processes, accelerating application production and deployment, and allowing everyone to learn from each other.

The following best practices should be helpful to adopt a culture of automation:

- Change the mindset.
- Create an automation community of practice.
- Have a common repository for automation code.
- Create a product mindset, not a project mindset.
- Treat automation as a product, not a project.
- Embrace AI in your automation process.

Always Be Architecting Principle

One of the core objectives of cloud native applications is the *always be architecting principle* (AbAP), which means always keep evolving. You should always use this principle when you are architecting the system as your application seeks to refine, simplify, and improve the architecture to support business disruption, organization change, system change, and technology disruption. Dead, rigid IT systems bring the organization to a standstill and are unable to support business disruption.

Cloud native architecture does not replace traditional architecture, but it is better adapted to the very different environment of the cloud.

Interoperability Principle

The interoperability principle is an enterprise architecture principle that states that software and hardware should conform to defined standards that promote interoperability for data, applications, and technology platforms.

Enterprise architecture frameworks state the principle as follows:

- The ability of a system to use the parts of another system
- The ability of a business entity to use functionality or information provided by another business entity

Interoperability improvement across applications and business can be realized through the following objectives:

- Design your application based on open industry best practices; this helps your application interoperate across any public, private, or hybrid cloud infrastructure.
- Design your application with industry best practices and standards; therefore, the information and services are shared across various other applications in an enterprise.

Here's how to manage the interoperability across various architecture segments:

- At the architecture level, you need to specify and/or define how you exchange or share information across various modules or systems.
- At the data level, you need to specify and/or define information exchange model details and the content of the information exchange.

Here's how to apply interoperability in architecture:

- *User experience integration*: A common look-and-feel approach is used to access the underlying functionality of the applications.
- *Information integration*: A commonly accepted corporate ontology is followed for seamlessly sharing information across applications.
- *Application integration*: Use choreography or the orchestration principle to seamlessly link functionality to avoid duplication.
- *Technical integration*: Use common methods to share data across application platforms and communication infrastructure domains.

Digital Decoupling Principle

The *digital decoupling principle* (DDP) was coined by Accenture and is as follows

> *"A process of using new technologies, development methodologies and migration methods to build systems that execute strategy on top of legacy systems. The organization can decouple the rapid execution of their business strategy from the lengthy and gradual transformation of the enterprises."*
>
> —Accenture

When applied to the enterprise landscape, digital decoupling leads to exponential IT, a scalable, flexible, and resilient architecture that gives companies the agility to innovate.

A few examples of DDP include data meshes, APIs, agile, DevSecOps, journey to cloud, microservices, RPA, and automation, as shown in Figure 3-6. Using these approaches, enterprises can gradually decouple their core systems, migrating critical customer-facing functionality and data to new service-based platforms.

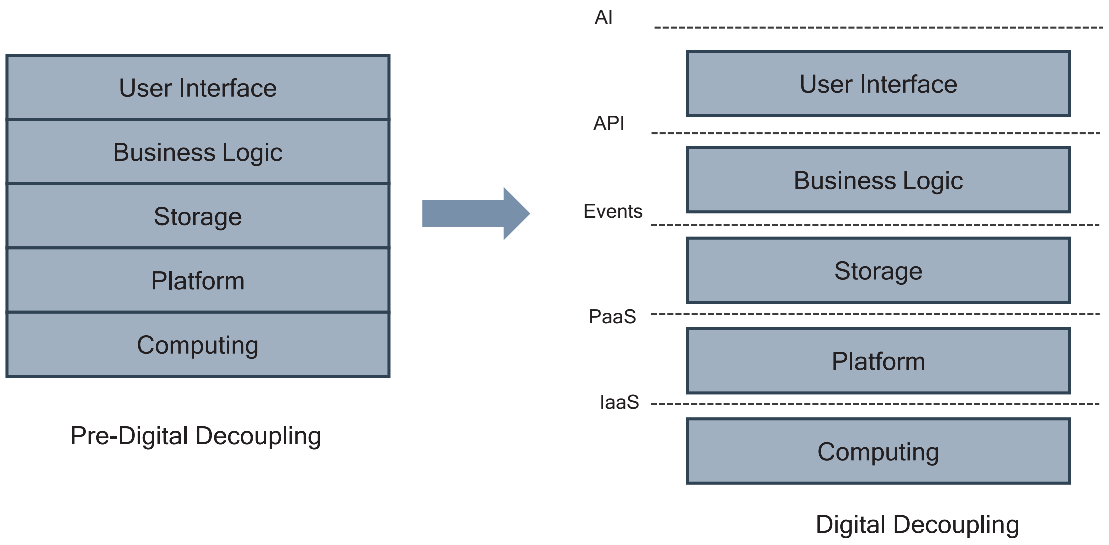


Figure 3-6. *Digital decoupling*

Here are some tips to achieve digital decoupling in your enterprise:

- Automate using RPA.
- Utilize cloud native to quickly build microservices.
- Use a data lake or data mesh with real-time eventing capabilities.
- Adopt API first and consumer first principles.
- Use interactions that react in real time to use behavior.
- Use systems of intelligence to enable smart interactions.
- Remove conflicts of interest and increase agility and enable future replacement.
- Do not use batches; the systems go straight through with minimal human interaction.
- Leverage cloud capabilities to isolate the infrastructure and platform.

By adopting DDP, enterprises can focus on continuous modernization without the pain of wholesale migration of legacy systems. The more systems are decoupled, the more enterprises can evolve toward an even greater service-based exponential IT architecture that maximizes agility. This approach helps manage costs, diminishes the accumulation of technical debt, and significantly reduces legacy transformation risk.

Single Source of Truth Principle

The *single source of truth principle* (SSOTP) is not a tool but a practice of aggregating the data from many sources in an enterprise to a single location. In an enterprise, data exists everywhere, and this data exists in silos and does not help a business to make data-driven decisions. Without a single source, how can an organization improve the efficiency and effectiveness of its operational environment, its transparency, and its future growth?

If the company does not have any single authentic source of information, it often spends far too much time debating the accuracy of numbers, and this hinders the decision-making ability and loses competition to their peers.

Use various tools and techniques to aggregate data from across systems in an enterprise to a single location in near real time so the team can run business intelligence tools to generate the required information.

Evolutionary Design Principle

The main idea of the *evolutionary design principle* (EDP) is that design elements are changeable later. When you build in an evolutionary change in your architecture, changes will become cheaper and easy.

Traditionally, software architecture and design phases have been considered as an initial discovery phase. In this approach, the architecture and design decisions were considered valid for the entire life of the system.

In a modern system architecture and design, you need to assume that you don't have all the required details up front. As a result, having a detailed design phase at the beginning of the project is impractical. The domain services must evolve through iteration, and services mature as they progress. This evolution is necessary for modern-day architecture, which necessitates a different set of approaches in the direction of continuous planning, continuous integration, integrated monitoring, and tools thus providing guiderails for the system to evolve.

As a result of this principle, the team can build a minimum viable product (MVP) with a set of features and rollout to the users. The development team doesn't need to cover all the design features to roll out features; instead, the development team can focus on the needed pieces and evolve the design as customer feedback comes in. You can freeze initial feedback, refactor, and complete the service.

The following software design patterns (more details in Chapter 4) can be used to achieve evolutionary design:

- Sidecar extends and enhances the main service.
- Ambassador creates helper services that send network requests on behalf of the consumer service or application.
- The chain provides a defined order of starting and stopping containers.
- The proxy provides surrogates or placeholder.
- An iterator is a way to access the elements of aggregate objects.

Infrastructure as code provides additional automation for container images and deploys automatically in any place at any given point of time.

Cloud Native Runtime Principles

These are the cloud native runtime principles.

Isolate Failure Principle (IFP)

Embracing a cloud native architecture doesn't automatically make your system more stable. Designing to isolate failure in your microservices can ensure that your microservices don't become fragile. Microservices are not reliable by default; therefore, you can't assume that your microservices become more resilient or scalable by default.

For example, say you have five microservices in one system, as shown in Figure 3-7. For this system to work, all five services have to be up and running. If any service is down, it may impact the whole process; therefore, all five must be available at any given point of time or the system stops processing all requests. In other words, if one service goes down, it takes them all down.

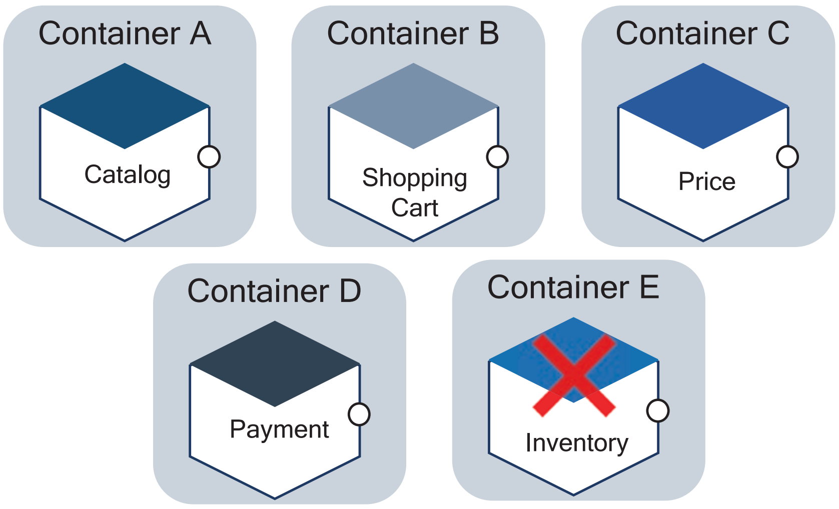


Figure 3-7. *Microservice failure*

If any one of the services fails, the system stops working; if any one of the networks between services stops, it fails, and your services stop working. Therefore, your services are less reliable.

You must consider the failure of microservices so you can avoid the single point of failure. Ask yourself, what happens if one of your services fails? Can your system keep running? Do you even know what happens when your users talk to your services? For example, the user clicks Catalogs, and your application invokes the catalog microservice, and the catalog microservice depends on the inventory microservice for an inventory, but your catalog microservice cannot invoke the inventory services to show the catalog because the inventory microservice is in a failed status. In this case, do you want to stop the whole system, or do you want to allow users to buy a product without availability in the inventory?

A particularly subtle sort of failure that can happen in a distributed system is the cascading failure, where all the way down the chain fails (service A calls service B and service B calls service E); this ripples all the way down the whole system.

A cascading failure can hurt a whole system, and you need to design your system to protect against this. You need to isolate the failure in every part of your system.

Deploy Independently Principle

The *deploy independently principle* (DIP) says that every service should be deployed independently in an infrastructure as a service (IaaS) by using containers and Kubernetes.

When you bundle more services into a single machine, you limit your ability to change things independently; this is the reason why you should deploy one service per container. The reason is the side effects; when you deploy a service in the same container with other services on it, what happens if one service fails? You need to forcefully stop other services also. Therefore, always consider one service per container.

The container provides a great way (as defined in the container principle) to deploy microservices. The following are the key tips you need to consider when deploying microservices in a container:

- Bundle the microservices into a container image.
- Deploy each service instance as a container.
- Deploy state and storage outside of the container.

Be Smart with State Principle

The *be smart with state principle* (BSSP) states when and how you store state in your design. Storing the state is the hardest aspect of architecting a distributed, cloud native architecture. Therefore, architect your system as stateless wherever it is possible.

Stateless means that any state must be stored outside of a container, and this external state can be stored in various storage. By storing data externally, you remove data from the container itself, meaning that the container can be cleanly shut down and destroyed at any time without fear of data loss. If a new container is created to replace the old one, you just connect the new container to the same datastore or bind it to the same disk.

Stateless components are as follows:

- *Easy to destroy and easy to create*: The stateless components have no dependency on the state to carry; therefore, the application in a container can be destroyed and created easily with no hassle.
- *Easy to repair*: If you want to repair failed instances in your deployment, simply terminate gracefully and spin up a replacement.

- *Auto scale/horizontal scale*: To scale more instances, just add more copies; the orchestrator can manage the scale-up and down. This scale can be managed automatically based on load or CPU usage.
- *Rollback*: If you have a wrong deployment, the stateless containers are much easier to replace with new ones without any human intervention.

Load balancing across services is much easier since any instances can serve any request from the requestor. If you have a state for an instance, you need to send a request to the same instance, and this can be managed with sticky sessions.

Location-Independent Principle

The *location-independent principle* (LIP) is about abstracting the physical location of the data from the logical representation that an application on a server uses to access data. In the cloud native application, the location of your deployment does not matter to the end customer or user, but they both should be able to access services ubiquitously and responsively regardless of location.

In a cloud native application, your services do not require you to define where you want to deploy a service, and one service doesn't need to know another service as both services are loosely coupled in nature if the services are required to communicate, though only in terms of API and events, as shown in Figure 3-8.

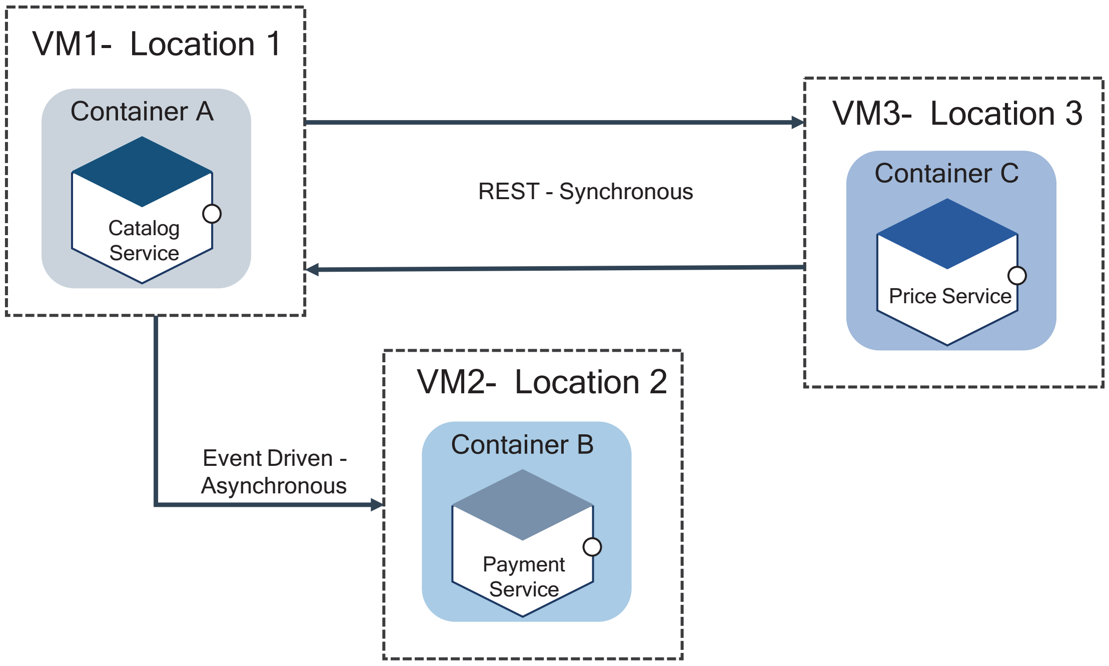


Figure 3-8. *Microservice deployments*

Follow these best practices when implementing location independence:

- Design your services based on a domain model and bounded context, which helps to avoid intercommunication.
- A distributed cloud provides public cloud options to a different physical locations, which helps latency and data privacy and regulations that require certain data to remain in a specific geographic location.
- Use the automation principle to deploy your services on any location.

Design for Failure Principle

The *design for failure principle* (DFFP) states that applications need to be designed so that they can tolerate the failure of services. Since services can fail at any time, it is important to be able to detect the failures quickly and, if possible, automate to restore quickly. Designing a failure means testing the design and watching services cope with deteriorating conditions. Design of failure yields a self-healing application and infrastructure.

Any call to microservices could fail due to the unavailability of the service; the client code must respond to the user as gracefully as possible. This emphasizes the real-time integrated monitoring of the application. Semantic monitoring can provide an early warning system of something going wrong that triggers stakeholders to follow up and investigate.

Designing for failure will help your services have greater availability and customer confidence on your application. Here are the key factors from the 12-factor app pattern methodology (more details in the "Architecture and Design of microservice Chapter 5") that provide best practices when designing for failure:

- *Disposability*: Maximize robustness with fast startup and graceful shutdown. Use lean container images and strive for processes that can start and stop in a matter of seconds.
- *Logs*: Treat logs as event streams. If a system fails, ensure you have collected all the integrated logs to troubleshoot.
- *Dev/prod parity*: Keep development, staging, and production as similar as possible.

Implement the failure as a service model to test all your services; for example, Netflix uses Simian Army or Chaos Monkey to test the failure of services. Amazon's use of a microservices architecture for its application means that the application never goes down, but there could be a problem in individual services. Amazon has built a user interface to gracefully degrade in the face of service failures.

Security Principles

These are security principles.

Defense in Depth Principle

The *defense in depth principle* (DiDP) provides a series of security mechanisms, and controls are layered throughout a computer network to protect the confidentiality, integrity, and availability of services.

Services in a cloud native architecture deploy and process requests for Internet applications, and there will always be a threat from external and internal attacks. Always use an authentication mechanism between services, that increases the trust between those services, whether it is an internal or external service.

You will apply this principle not just for authenticating the services to avoid rate limiting or script injection, but also you should protect your services from any threat. This makes your architecture more resilient and easier to deploy and creates more trust for your services. The DiDP principle ensures network security is redundant, preventing any single point of failure.

An effective DiDP strategy may include the following security best practices, tools, and policies:

- Strong credentials management
- Firewalls
- Intrusion prevention or detection system
- Endpoint detection and response
- Network segmentation
- Patch management
- APIs authentication
- Auditing and accounting

Security by Design Principle

The *security by design principle* (SBDP) means that the product has been designed from the ground up to be secure. The alternate security patterns are researched, and the best are selected and enforced by the architecture design.

Most attacks of any Internet-facing services either in a private or public cloud are performed because of software vulnerabilities. Software vulnerabilities are often found in the design and development lifecycle, so if you ignore any findings, you leave your service exposed to the hands of cybercriminals.

As I mentioned in Chapter 2, cloud applications are made up of IaaS, PaaS, and SaaS. In IaaS, a cloud vendor provides the physical or virtual infrastructure; you are responsible for the administering of network and system infrastructure, applications, and data. With the PaaS model, the cloud provider manages the infrastructure and managed components such as databases, middleware, etc., and you are responsible for the application and data security. In a SaaS model, the cloud provider provides everything from the infrastructure to the application, and you are responsible for access and data.

In a cloud native application, you are responsible for most of your application and data security; therefore, you need to provide utmost importance for security. There are various techniques and best practices available to secure your application.

The following practices help when designing and developing an application:

- *Minimize attack surface area*: This restricts the services that a user can access.
- *Establish secure defaults*: Implement strong security rules for how users are registered to access your services.
- *The principle of least privilege*: The user should have the minimum set of privileges required to perform a special task.
- *The principle of defense in depth*: Add multiple layers of security validations.
- *Fail securely*: Failure is unavoidable; therefore, fail in a secure way.
- *Don't trust services*: Don't trust third-party services without implementing a security mechanism.
- *Separation of duties*: Prevent individuals from acting fraudulently.
- *Avoid security by obscurity*: There should be sufficient security controls in place to keep your application safe without hiding core functionality or source code.
- *Keep security simple*: Avoid the use of very sophisticated architecture when developing security controls.
- *Fix security issues correctly*: Developers should carefully identify all affected systems.
- *Implement shift-left security*: Implement security from the developer box.

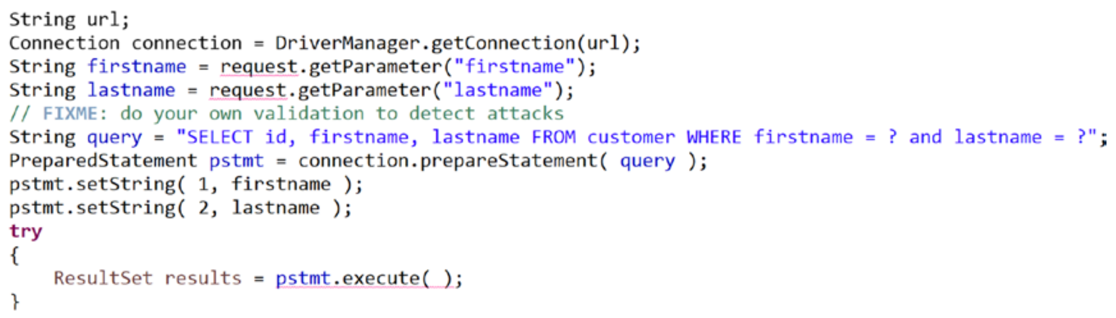
The Open Web Application Security Project (OWASP) provides security design techniques and best practices that designers should adopt while designing services. The OWASP updates the list of vulnerabilities often and rates them based on the security reports. You need to well aware of the implementation and adherence of the security risks. The following are the few implementation of OWASP security risks.

SQL Injection

SQL injection is a security risk where a SQL query is input to your query. If an attacker can exploit your SQL query and can read sensitive data from your databases and even modify the data, the consequences are confidentiality, authentication, authorization, and integrity.

- *Standard SQL syntax*: `select id, firstname, lastname from customer;`
- *With query string*: `select id, firstname, lastname from customer where firstname='Peter's' and lastname ='john'`

The database tries to run this example but provides incorrect syntax. Figure 3-9 shows the correct implementation.



```
String url;
Connection connection = DriverManager.getConnection(url);
String firstname = request.getParameter("firstname");
String lastname = request.getParameter("lastname");
// FIXME: do your own validation to detect attacks
String query = "SELECT id, firstname, lastname FROM customer WHERE firstname = ? and lastname = ?";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, firstname );
pstmt.setString( 2, lastname );
try
{
    ResultSet results = pstmt.execute( );
}
```

Figure 3-9. *SQL injection implementation*

Cross-Site Scripting (XSS)

XSS attacks are type of injection like SQL injection; here the attacker injects malicious scripts into a web application. Flaws in your user experience code like Angular, JavaScript, etc., allow these attacks to succeed. XSS attacks occur when:

- Data enters a web application through an untrusted source.

- The data included in the dynamic content is sent to a web user without any proper validation for request.

Implement the following best practices to avoid XSS in your web application:

- Use the OWASP XSS prevention sheet from the OWASP community (`https://cheatsheetseries.owasp.org/cheatsheets`).
- Turn off HTTP trace, or an attacker can steal cookie data.
- Use the proper syntax in your code, don't hard-code, and use variables and parameters.

Here I have provided a few examples. You can find more details and implementation best practices on the OWASP.org community website.

Software Engineering Principle

These are software engineering principles.

Products Not Projects Principle

Amazon states that the core benefit of treating software as a product is an improved end-user experience. When an enterprise treats its software as an always improving product rather than a one-off project, like with the *products not projects principle* (PNPP), it will produce code that is better architected for future work.

Traditionally, enterprises and service organizations delivered software as a project with a set of resources and start and end dates with a list of predefined features. A product-centric development lives for an indefinite period and evolves and has no fixed predefined features.

In the project-centric approach, there will be a little room for iteration and improvement as the software spends a small amount of time in the hands of end users before the budget is exhausted. But in the product-centric approach, you will adopt the MVP approach, where the smallest increment is delivered to real users as soon as possible, so the team can get early feedback that sets the future direction.

The core benefits of treating software as a product are the following:

- Improved end-user experience
- Matured architecture
- Automation and innovation culture
- MVP approach
- Easier to extend, maintain, and test
- More visibility into how their software is performing in real-world scenarios
- Accelerates feedback loop

The following concepts are crucial for adopting a product approach:

- *Automated provisioning with cloud-enabled*: Use the infrastructure automation principle.
- *Self-service, self-healing*: Configure own dependencies and better configuration management by adopting the separation of concerns principle.
- *DevSecOps pipeline with infrastructure as code*: Adopt automation culture.

Shift-Left Principle

The *shift-left principle* (SLP) refers to a practice in software engineering development in which scrum teams can focus on quality, work on problem anticipation instead of detection, and begin testing from the developer system.

This principle in DevOps is a set of a process aimed at the following:

- Finding and preventing defects early in the software delivery lifecycle
- Beginning testing, security, and performance earlier than ever before
- Focusing on quality

The idea of this principle is to improve quality by moving tasks to the left as early in the lifecycle as possible, thus reducing the technical debt and cycle time.

Shift-Left Security

SLP will be applicable to functional, security, and performance testing and related processes, techniques, and tools to be integrated as part of the DevSecOps and developer integrated development environment (IDE).

The shifting left of the security review process requires a new way of developing the application compared to the traditional approach; these changes are not a significant deviation. You need to follow these tips for shift-left security:

- Involve an information security expert early in the lifecycle of the project.
- Use security tools.
- Integrate security tools as part of the continuous integration and as part of the developer IDE.

Shift-Left Performance

Shifting performance testing means enabling developers and testers to conduct performance testing in the early stages of the development lifecycle. Performance means not just a request or stress; actual performance starts with the code and therefore involves practices at the developer level to prevent performance-related issues. To implement the shift-left approach, implement best practices, tools, and techniques as part of the continuous integration pipeline and as part of the developer environment. The following are the best practices to be adopted for shift left:

- Implement performance testing with or in parallel to development activities.
- Include performance testing along with the unit, system, and integration test lifecycles.
- Create performance attributes.
- Integrate tools as part of DevSecOps.

Container Principles

The following are the container principles.

Single Concern Principle

In many ways, the *single concern principle* (SCP) is like the single responsibility principle from SOLID, which says that a module or class must have only one responsibility.

In a cloud native architecture, SCP highlights higher level of single of responsibility. The single responsibility enables you to define a clear boundary for every microservices.

The main motivation for the single responsibility principle is to have a single reason for a change; the main objective of the SCP is for container image reuse and replaceability. You can create a container that addresses a single responsibility with the common feature, and then you can reuse the same container image in different applications without modification and testing.

The SCP principle objective is that every container must address a single resposibility with a microservices architecture style. Always use a single responsibility in the container even though your microservice provides multiple resposnibility. If you have microservices with multiple resposibility, use sidecar and init-containers patterns as explained in Chapter 4 to combine multiple containers into a single deployment unit (pod), where each container still holds single responsibility, as shown in Figure 3-10. You can swap a container that addresses the same responsibility. For example, replace service A container with service C by using infrastructure as code.

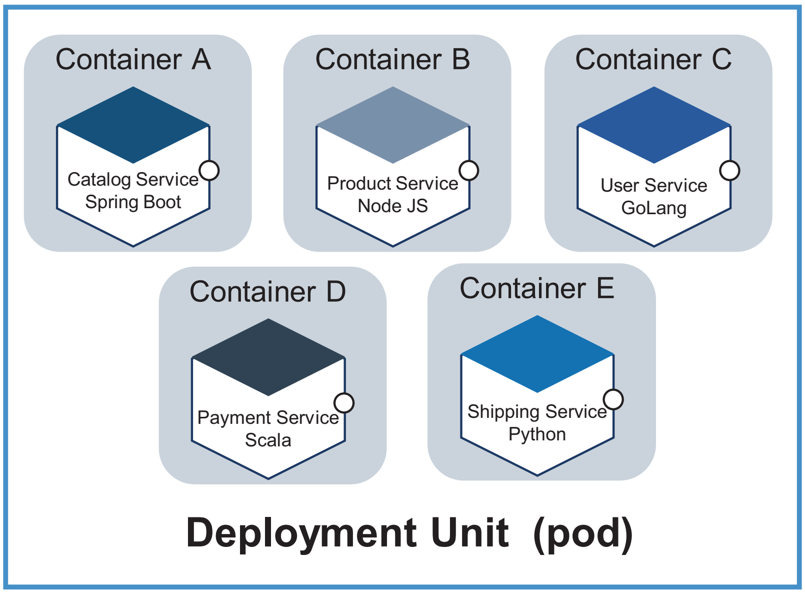


Figure 3-10. *Microservices deployed in separate containers*

High Observability Principle

Observability is a measure of how well internal states of microservices can be derived from external outputs. The concept of observability was introduced by Rudolf E. Kalman for linear dynamic systems.

The observability principle states that an application is said to be observable if one can determine the behavior of the entire application from the application output.

Logs, metrics, traces, liveness, readiness, and process health are known as the pillars of observability, as shown in Figure 3-11, in a cloud native architecture. While having access to these pillars doesn't make your application more observable, you need to create interfaces to access these pillars for further analysis.

Containers provide a unified way of packaging and running microservices by treating the application as a black box. You need to configure containers with APIs to access runtime environments to observe the container health and act accordingly. These are the prerequisite for automating container updates and lifecycles in a unified way, which in turn improves the system's resilience and user experience.

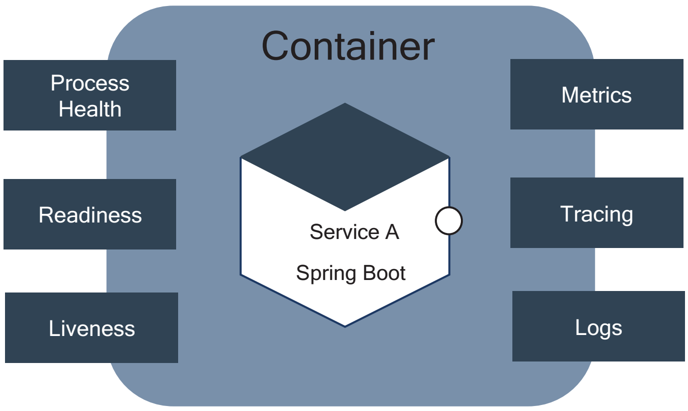


Figure 3-11. *Observability in cloud native application*

You need to design your container and application with APIs for the different kinds of health checks. The microservices should log events into the standard error (STDERR) and standard output (STDOUT) for log aggregation by using tools such as FluentD, Logstash, Nagios, etc., and should integrate with tracing and metrics-gathering libraries such as Zipkin, open tracing, etc.

At runtime, your application is a black box to you; implement the necessary APIs to help the platform observe and manage your application in the best way possible.

Lifecycle Conformance Principle

The *lifecycle conformance principle* (LCP) states that a container should have a way to read the events coming from the platform and conform by reacting to those events.

All kinds of events are available for managing platforms that are intended to help you to manage the lifecycle of the container and microservices, based on all types of available events; it is up to you to decide which events to handle and whether to react to those events or not.

By looking into all sorts of events, you need to pick important events, as shown in Figure 3-12, for example.

- Graceful shutdown process
- Terminate message (SIGTERM)
- Forceful shutdown (SIGKILL)

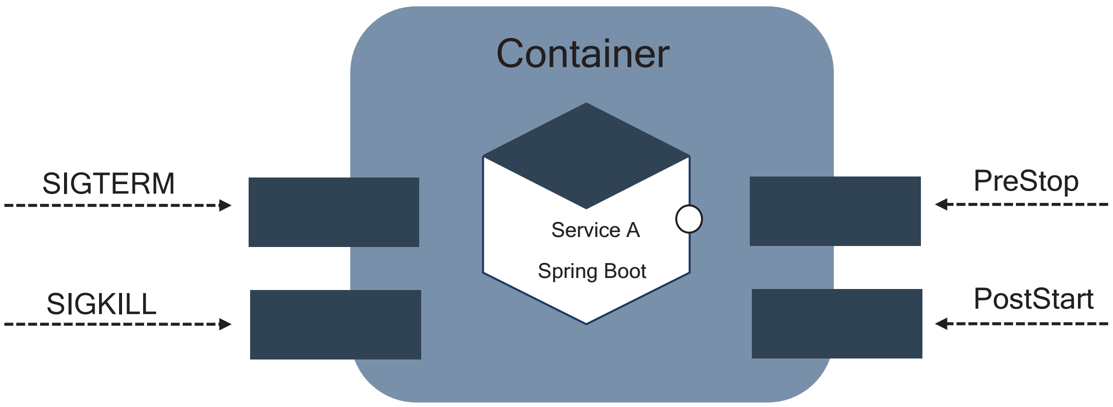


Figure 3-12. *Container lifecycle*

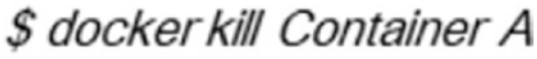
When you issue a `docker stop` command, Docker will wait for 10 seconds to stop the process; if there no action in 10 seconds, then it will forcibly kill the process.

Command to stop process:

```
$ docker stop Container A
```

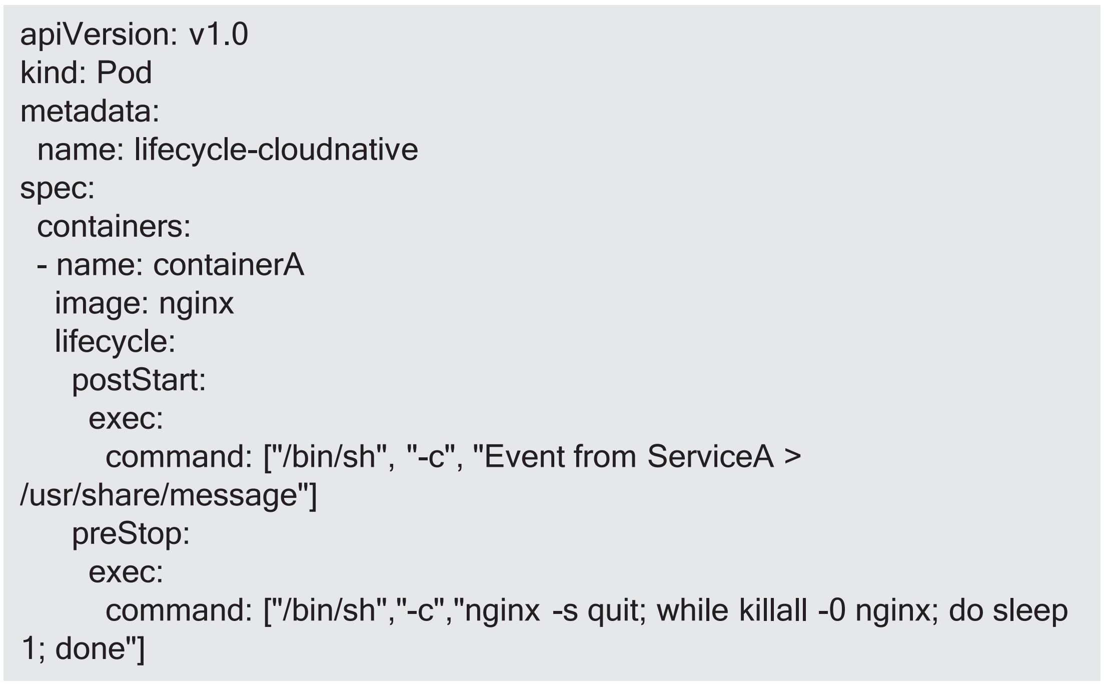
The `docker stop` command attempts to stop running the container by sending a SIGTERM signal to the root process in the container; if the process hasn't exited within the timeout period, a SIGKILL signal will be sent.

Command to kill process:



```
$ docker kill Container A
```

There are other events such as `PreStop` and `PostStart`, which might be significant in your application lifecycle management. For example, some applications need to warm up before a service request, and some need to release resources before shutting down clearly, as shown in Figure 3-13.



```
apiVersion: v1.0
kind: Pod
metadata:
  name: lifecycle-cloudnative
spec:
  containers:
  - name: containerA
    image: nginx
    lifecycle:
      postStart:
        exec:
          command: ["/bin/sh", "-c", "Event from ServiceA >
/usr/share/message"]
      preStop:
        exec:
          command: ["/bin/sh","-c","nginx -s quit; while killall -0 nginx; do sleep
1; done"]
```

Figure 3-13. *Configuration file*

In this configuration file, you can see how to use the `PostStart` and `PreStop` command to write a message file to the container's `/usr/share` directory. The `presto` command shuts down Nginx gracefully

Image Immutability Principle

The *image immutability principle* (IIP) states an image is unchangeable once it is built and requires creating a new image if changes need to be made. Container applications like microservices are meant to be immutable. Once you have developed applications, they aren't expected to change between different environments except for runtime data like environment configuration and variables such as listening port, runtime options, etc. You need to store configurations and variables external to the container. For each image change, you need to build a new image and reuse it across various environments in your development lifecycle.

Immutability makes deployments safer and more repeatable. If you need to roll back, you simply redeploy the old image. This approach allows you to deploy the same container image in all your environments. Containers are usually configured with environment variables or configuration files mounted on a specific path. You can use secrets and config maps to inject configurations in containers as environment variables or files of Kubernetes. If you need to update a configuration, deploy a new container (based on the same image) with the updated configuration, as shown in Figure 3-14.

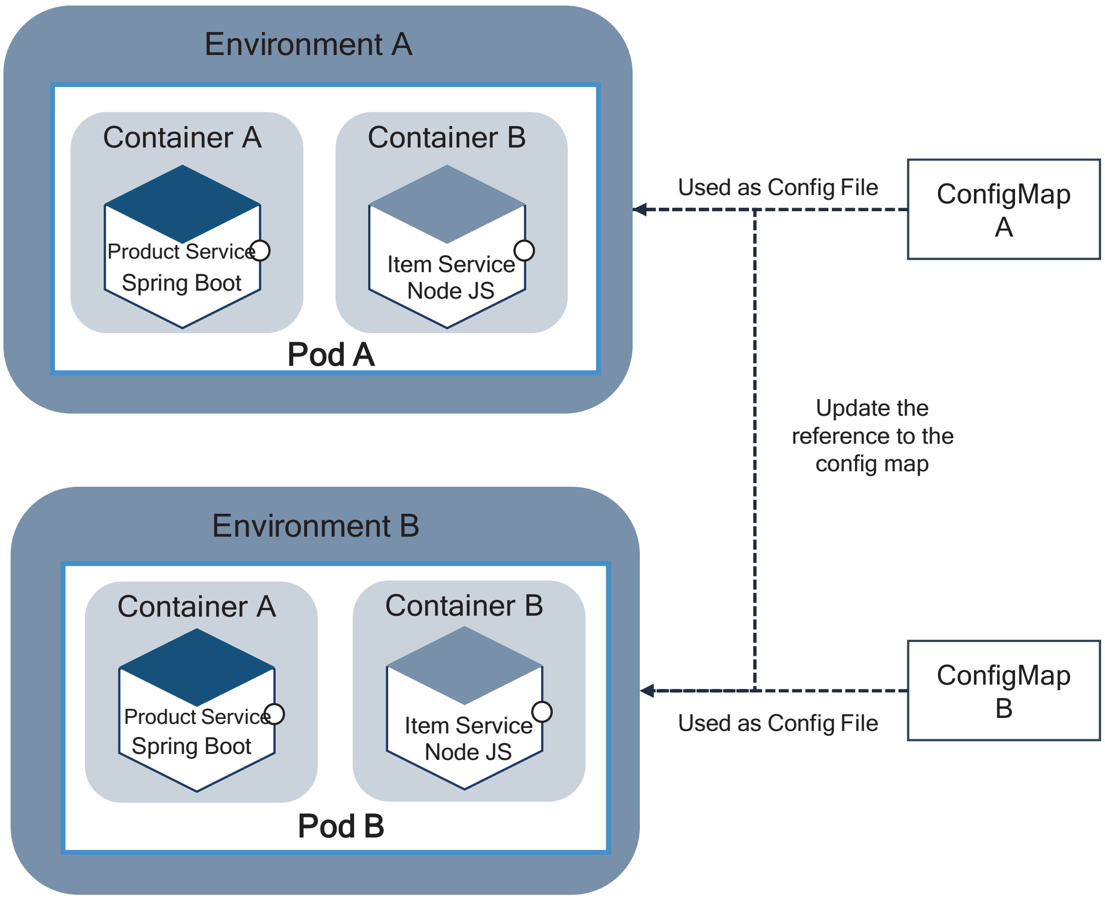


Figure 3-14. *Immutable container images across all environments*

Immutability is one of the best qualities of container-based infrastructure. Immutability along with statelessness allows you to automate deployments and increase their frequency and reliability.

Process Disposability Principle (PDP)

The *process disposability principle* (PDP) is a container runtime principle and states applications must be ephemeral as possible and ready to be replaced with container instances at any point of time by using infrastructure as code, as shown in Figure 3-15.

Usually, you may not replace containers regularly except for a few circumstances such as the following:

- Container not responding to health checks
- Autoscaling down the application with CPU utilization or load
- Migrating the container to a different host
- Platform resource starvation

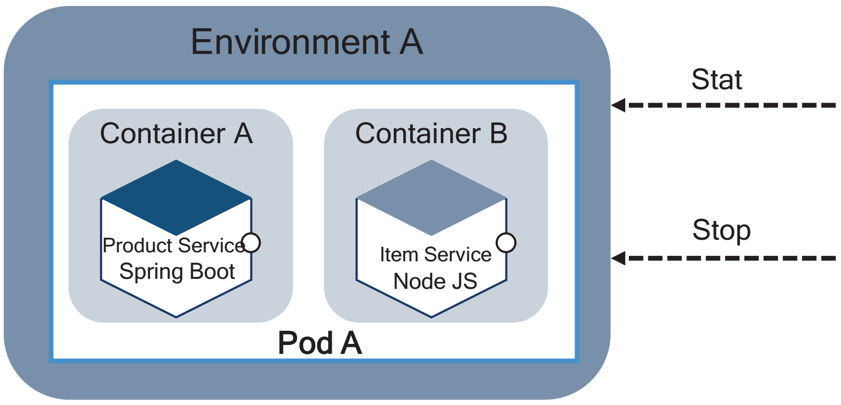


Figure 3-15. *Container replacement based on load*

If you store the state within the container, then it is difficult to replace in a distributed environment; therefore, you should keep their state externalized or distributed and redundant.

Figure 3-16 illustrates how the PDP principle is applied.

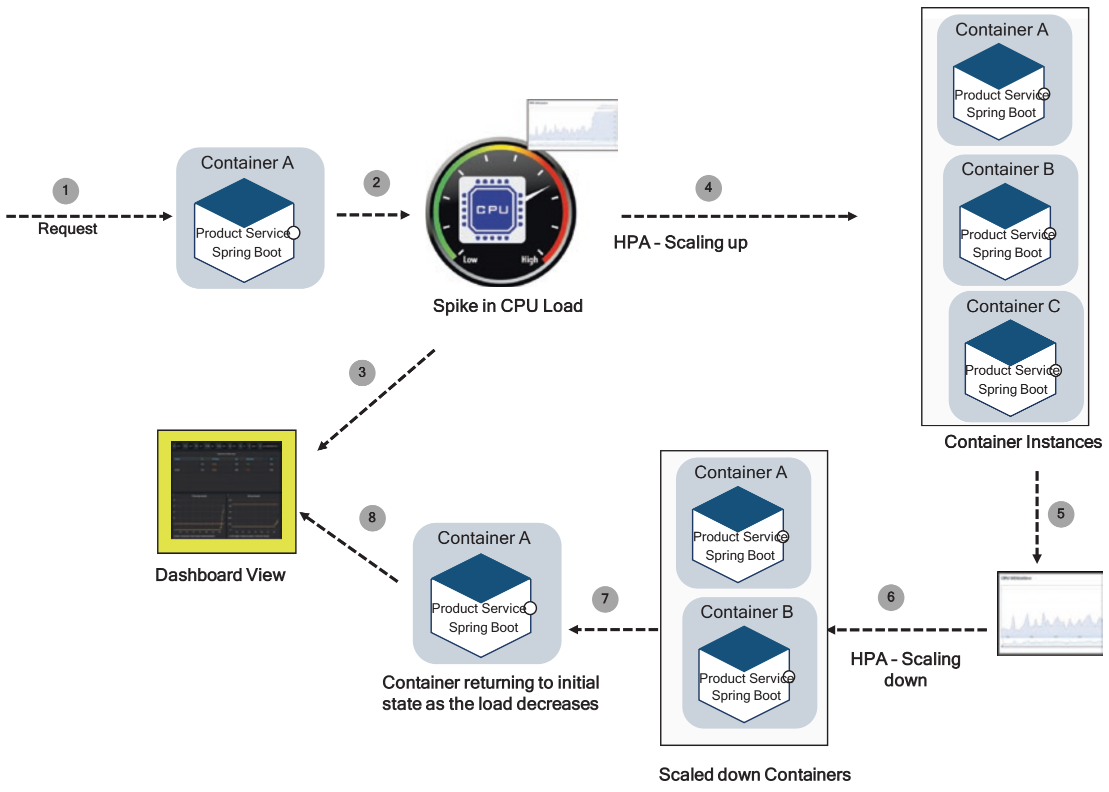


Figure 3-16. Container scale-up and down based on Spile in CPU

At the beginning of your day, service A has only one container instance, but as the day progresses and the load increases, the containers autoscale to three instances to meet the demand. The container instances dispose gradually as and when the load decreases, and finally it reaches the original state. This can be achieved by using the PDP.

You need to follow best practices for the size of containers and functionality of microservices. For example, it is better to create small containers, which leads to quicker start and stops because, before the spin of the new container, the containers need to be physically copied to the host system.

Self-Containment Principle

The *self-containment principle* (SCP) addresses the build time concern, and the objective of this principle is that the container must contain everything that it needs at build time. The container relies on the presence of the Linux kernel or Windows silos and any additional libraries. The Windows silos are the Microsoft variant for the Linux namespace. With silos, Windows kernel objects such as files, registry, and pipes can be isolated into separate logical units.

Along with the container's Linux kernel or silos, the following should be added at the time of build:

- Dependent libraries
- Language runtime
- Application platform

The configuration and state are not part of the build time; they should be externalized at runtime through `ConfigMap`, as shown in Figure 3-17.

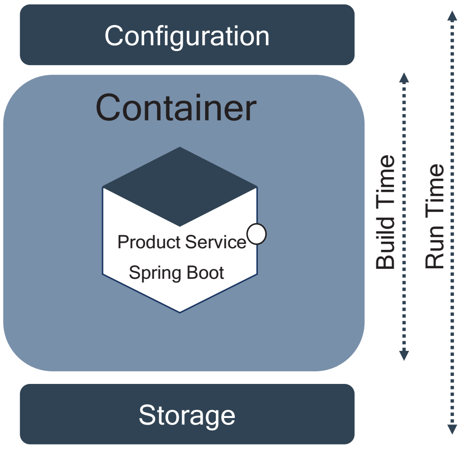


Figure 3-17. *Containers with build and runtime environments*

Some of your applications require multiple container components. For example, your containerized microservices may also require a database container. This principle does not suggest merging both containers; instead, this principle suggests each container requires a dependent configuration to run respective containers.

Runtime Confinement Principle

The *runtime confinement principle* (RCP) states that every container should declare its resource requirements and pass that information to the hosted platform.

The SCP addresses the build-time perspective, and RCP addresses the runtime perspective. The container is not just a single black box, but it has multiple dimensions as follows:

- CPU usage dimension
- Memory usage dimension
- Resource consumption dimension
- Control groups dimension

The container, as shown in Figure 3-18, shares the resource profile of a container to a hosted platform in terms of CPU, memory, networking, and disk influence to specify how the platform performs scheduling, autoscaling, capacity management, and SLAs of the container.

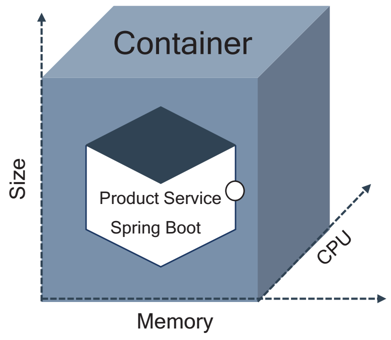


Figure 3-18. *Container runtime characteristics*

In addition to passing the resource requirements to the host platform, it is important that the application stay confined to the indicated resource requirements. If the application stays confined, the platform is less likely to consider it for termination and migration when resource starvation occurs.

Principles of Orthogonal

In mathematics, orthogonality describes the property of two vectors. As shown in Figure 3-19, they are perpendicular, or 90°, to each other. Each vector will advance indefinitely into space, never to intersect.

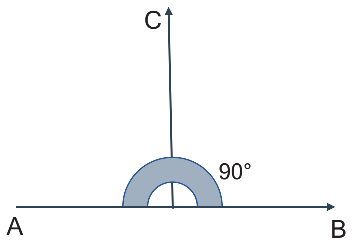


Figure 3-19. *Orthogonal*

Well-architected software is orthogonal, and each of its components or modules can be modified without affecting another. By considering agility in both business and technology, the software applications undergo many changes to support business disruption. The cost of applying orthogonal principles is a little high, but by considering the cost at the end, the overall cost will be managed by considering changeability, testability, extensibility, etc.

The orthogonal design is based on two principles, as shown in Figure 3-20.

- Cohesion
- Coupling

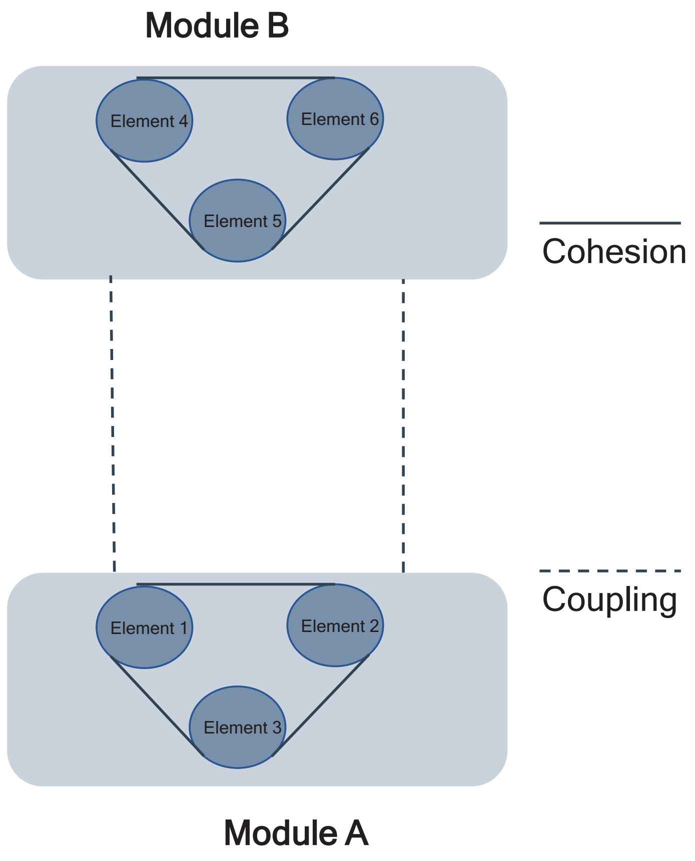


Figure 3-20. *Orthogonal principle*

Cohesion

Cohesion is the degree to which the elements inside a module belong together. It is the strength of the relationship of elements within the module. It is the internal glue that keeps the module together.

It is a measurer that defines the degree of intradependability within elements of a module. The greater the cohesion, the better the program design. It is a natural extension of the information hiding concept.

A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. We always strive for high cohesion, but sometimes the middle path of the spectrum is always acceptable, as shown in Figure 3-21.

Cohesion is an ordinal type of measurement and is generally described as *high cohesion* and *low cohesion*.

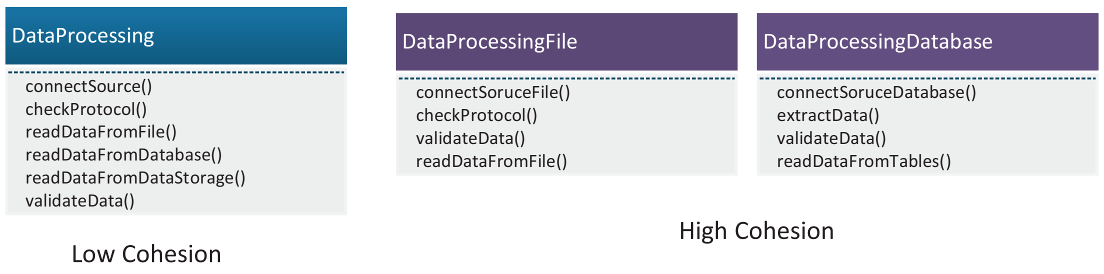


Figure 3-21. *High and low cohesion*

High cohesion is where you have a module that does a well-defined job with similar elements; it gives us a better-maintaining facility and reflects a better quality of a design. Reusability is high as all elements in the module work together as a logical unit of work with clear functionality. This makes it easier to do the following:

- Understand what class or method does
- Use descriptive names
- Reuse classes or methods

Low cohesion is where you have a module that does a lot of unrelated jobs and results in a monolithic module that is difficult to maintain, extend, and test. The extra complexity in modules with low cohesion makes it more likely that defects may be introduced and leads to high technical debt. Reusability is reduced for modules as it performs diverse functionality.

Types of Cohesion

Cohesion is a qualitative measure; the cohesion is measured based on the level of cohesion in a module, as shown in Figure 3-22. Let's examine the type of cohesion.

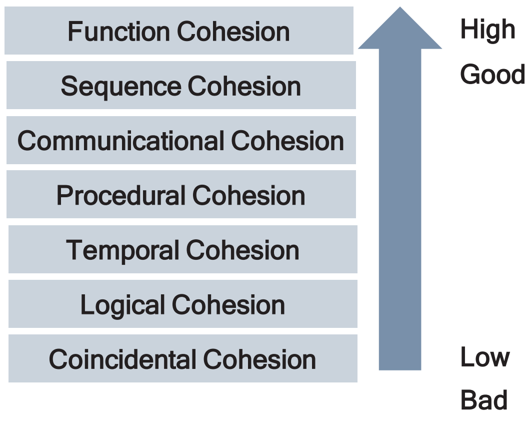


Figure 3-22. *Types of cohesion*

Function Cohesion

This is the highest degree of cohesion. Every essential element for a single computation is contained in the component because they all contributed to a single well-defined function. It can also be reused. Modules with functional cohesion perform exactly one action.

Here are some examples:

- Lexical analysis of XML. Converting a sequence of characters or elements in XML into a sequence of tokens. The group of elements is grouped together to analyze XML.
- Assign a seat to train passengers.
- Calculate the interest rate; calculate the sales commission.

Sequence Cohesion

Sequential cohesion is like a sequential operation. The elements of a module are grouped because of the output from one element and input to another element. This type of cohesion you can see in streaming data or file or ETL jobs.

Here are some examples:

- In an ETL application, the extract, transfer, and load functions are grouped into one module for each data element.
- In streaming, it is the continuous transmission, validation, storage, and display of audio or video of data files.

Communication Cohesion

A module is said to have communicational cohesion if all functions of the module refer to or update the same data structure, or a cohesive module is one whose elements perform different functions, but each function references the same input information or output, as shown in Figure 3-23. This cohesion is not flexible as it lacks the reusability principle.

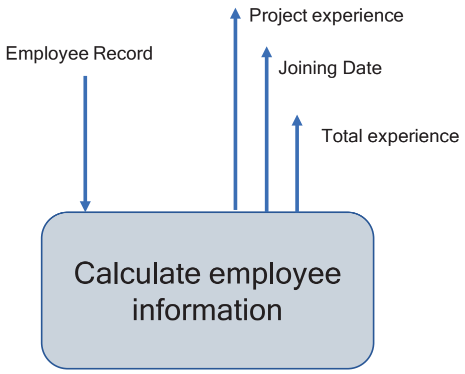


Figure 3-23. Communication cohesion example

Procedural Cohesion

Procedural cohesion is when elements of a module are grouped as they always follow a certain sequence of execution and are commonly found at the top of the hierarchy such as the main program. It is like sequential cohesion, as shown in Figure 3-24, except for the elements in the sequence are unrelated in procedural cohesion.

The weakness of procedural cohesion is that actions in a sequence are weakly connected and modules are unlikely reusable.

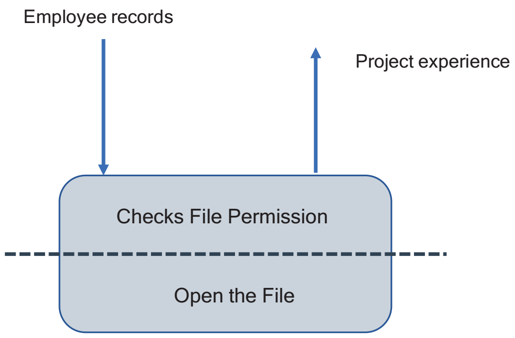


Figure 3-24. Procedural cohesion example

These two separate elements in a module are cut along the dotted line. We could do separate activities in each element. Checking the file permission operation can be used for another file also, and we can open the file if no checks are available.

Temporal Cohesion

The elements in this cohesion are related to the time; all the tasks must be executed in the same period.

The actions of this module are weakly related to one another but strongly related to actions in other modules. The elements are not reusable in this cohesion.

For example, consider a module in a digital twin that invokes the factory tasks that are not functionally similar or logically related, but all tasks are needed to happen at the moment when the failure occurs. The module might do the following:

- Cancel all outstanding requests for services.
- Cut power to all assembly line machines.
- Notify the operator console.
- Make an entry in the database.
- Invoke an alarm if a catastrophic failure occurs.

Logical Cohesion

Logical cohesion is when elements of a module are grouped because they are logically categorized to do the same thing, even if they are different by nature.

The following are the drawbacks of logical cohesion:

- The interface is difficult to understand.
- Code for more than one action may be intertwined.
- Reusability is lessened.

The actions of this module are all logically read as input content.

The type of input, as shown in Figure 3-25, tells the module what part of its internal logic to apply to the particular transaction data coming in for each specific invocation.

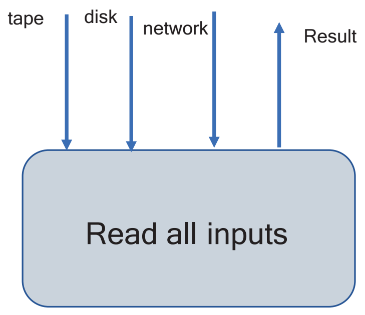


Figure 3-25. *Logical cohesion example*

Coincidental Cohesion

Coincidental cohesion is when elements of a module are grouped; the only relationship between the parts is that they have been grouped.

- Elements contribute to activities with no meaningful relationship to one another.

The drawbacks of this cohesion are degraded overall application maintainability and that modules are not reusable in nature.

Helper or utility classes in your application, usually utility classes, contain many functions that are unrelated and accessible from various other classes or modules. Changes in one function in the utility class affect the utility class and also the calling class.

Applying High Cohesion to Software Design

Design your application by keeping high cohesion in mind. Each module should have a single well-defined functionality. The elements within the module must be related and perform on the same set of data.

There are ancillary elements in the module that are not directly related, and they work on a different set of variables; consider moving nonrelated functionality into other related modules that have the same purpose.

Coupling

Coupling is the degree of interdependence between software modules or microservices; a coupling measures how closely connected two modules or microservices are and the strength of the relationship between modules or microservices. Coupling tells at what level the modules interface and interact with each other, as shown in Figure 3-26, Figure 3-27, and Figure 3-28. The coupling can be low or weak and high or strong or tight. The degree of the coupling between modules reflects the quality of the design.

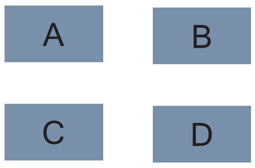


Figure 3-26. *No dependencies*

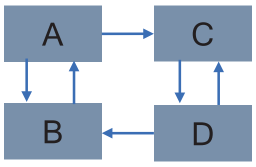


Figure 3-27. *Loosely coupled with some dependencies*

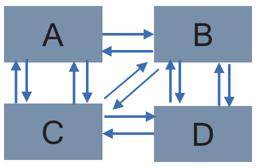


Figure 3-28. *Highly coupled with many dependencies*

Coupling is the measure of the interdependence of one module to another. Modules should have low coupling; low coupling minimizes the ripple effect where changes in one module cause an error in the other module.

Software modules that are *tightly coupled* are more complex; it is the degree to which one module is connected to another module. If a module is tightly coupled, then you are bound to use/edit the rest of the connected modules where editing only one module could have served the purpose. This impacts the principle of maintainability, extensibility, and testability. You need to carry out the full suite on the entire connected modules irrespective of modification, which increases the cost and effort.

The *loose coupling* design is to reduce the dependency that a change made within one module or microservice will create unanticipated changes within other elements. Individual modules can be altered or extended without the need to consider a lot of information from other modules. Errors of data flow can be pointed out easily. The loose coupling supports the principle of maintainability, extensibility, and testability.

Types of Coupling

There are different types of coupling, as shown in Figure 3-29. This section covers the details of these types in order from lowest to highest coupling. The coupling between the modules can be more than one way.

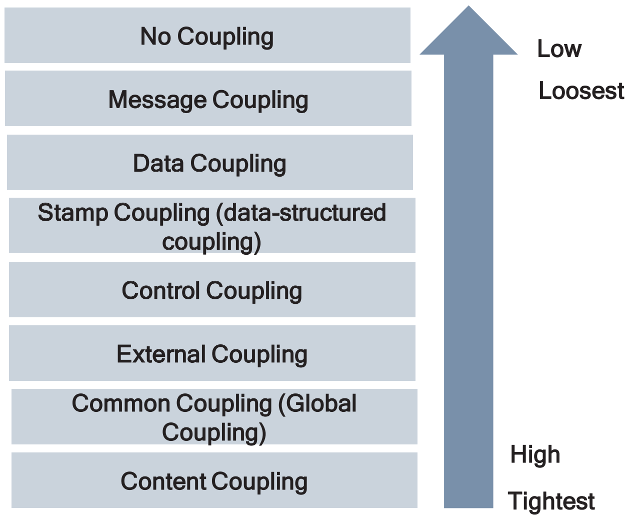


Figure 3-29. *Types of coupling*

No Coupling

In this coupling, the modules are isolated and do not communicate with each other.

Message Coupling

This is the loosest type of coupling. Modules are not dependent on each other; instead, one module calls a method or interface on another and does not pass any parameters. They are coupling only on the name of method or interface.

Example: Dependency injection and observable. Figure 3-30 depicts how message coupling helps to interact between two modules.

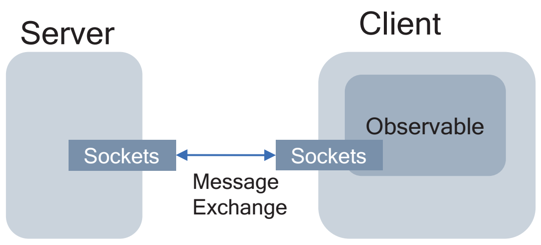


***Figure 3-30.** Message coupling*

In this example, the server and clients are loosely coupled and exchange details over the socket.

Data Coupling

When data of one module is shared with another module, this condition is said to be data coupling.

Data coupling occurs when methods share data regularly through parameters. The two modules, Module1 and Module2, exhibit data coupling if Module1 calls Module2 directly, and they communicate using parameters. Each parameter is an elementary piece, and the parameter is the only data shared between Module1 and Module2.

Example: As shown in Figure 3-31, the two modules Calculate EMI and Calculate Total Loan are data coupled as they communicate by passing the parameters.

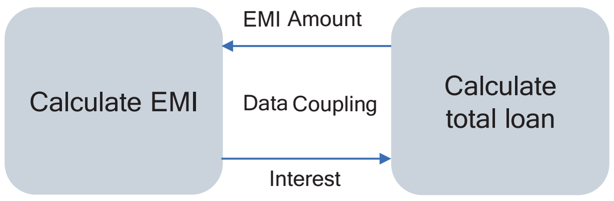


***Figure 3-31.** Data coupling*

Stamp Coupling (Data-Structured Coupling)

Stamp coupling occurs when modules share a composite data structure. If the module interacts by sharing or passing a data structure that contains more information than the information required to perform their actions, then these modules are said to be stamp coupled.

Two modules, module A and module B, exhibit stamp coupling if module A passes directly to module B a composite piece of data such as record, array, tree, or list.

Modules A and B will share a data structure and use only part of the whole data structure.

For example, ss shown in Figure 3-32, three modules are stamp coupled if they communicate via passed data structure, which contains more information than necessary for the modules to perform their functions.

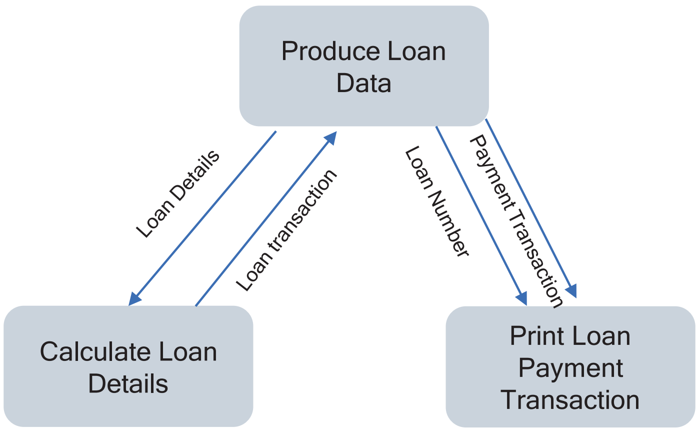


Figure 3-32. *Stamp coupling*

Here we assume Loan Number contains the loan number, date, address, etc. We are sending more information than what it requires. In this scenario, Calculate Loan Details requires only Loan Number to perform required functionality.

Control Coupling

Control coupling means to control data sharing between modules; in other words, control coupling occurs when one module controls the flow of another module by passing control information.

Two modules exhibit control coupling if module A passes to module B, a part of the information that is intended to control the internal logic of module B.

For example, as shown in Figure 3-33, the two modules Error Module and Notification Module are control coupled if they communicate using at least one control flag, denoted as Notification Flag.

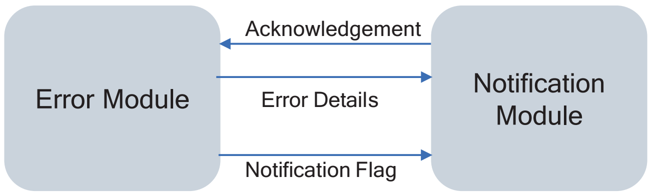


Figure 3-33. *Control coupling*

When an error occurs in an application, the error module captures the error and sends the notification flag to the notification module to send a notification to the stakeholders. Here the error module controls the notification module with the notification flag as a control flag.

External Coupling

External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface. This coupling is related to the communication to external tools and devices such as printers, IoT devices, etc.

For example, module A and module B exhibit external coupling if both modules share direct access to the same I/O devices or are tied to the same external IoT devices in some other way.

Common Coupling (Global Coupling)

Common coupling occurs when two or more modules share global data. Any changes to them have a ripple effect on all the modules; in other words, changing the shared resources implies changing all the modules using them.

For example, as shown in Figure 3-34, three modules are commonly coupled if they both share the same global data area.

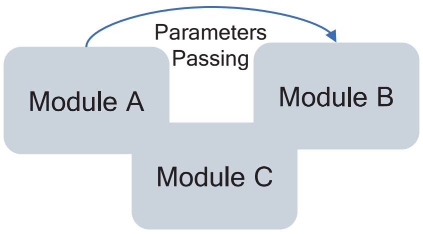


Figure 3-34. *Common coupling*

One of the design principles we have been using for many years is this: don’t use global data; it impacts security.

Content Coupling (Pathological Coupling)

When a module can directly access or modify or refer to the content of another module, it is called content-level coupling. Changing the inner workings will lead to the need of changing the dependent module. Module A refers to or changes the module B internal data or statement directly. This type of coupling is very high or tight in nature.

Module A and module B are content coupled if:

- Module A changes a statement in module B
- Module A references or alters data contained inside module B
- Module A branches into module B

For example, the search method that adds an object that is not found in the internal structure of the data structure is used to hold information.

Law of Demeter (LoD) or Principle of Least Knowledge

Introducing coupling increases the instability of a system. The law of Demeter is the important principle to reduce the coupling between modules. This law is a specific case of loose coupling. This law says:

- Each module or microservice has knowledge about only other modules or microservices closely related to the current module or microservices.
- Each module or microservices should talk only to its immediate friends; don’t talk to strangers.

The advantage of LoD is that resulting software tends to be more testable, maintainable, extensible, etc.

For example, module A could call module B’s interface for any intercommunication, but module A should not call module B to communicate module C. If module A needs to intercommunicate with C, then A calls directly to C.

Applying Loose Coupling to Software Design

Coupling is unavoidable; we need to have coupled. Otherwise, each class in a module or microservices would be its module. However, achieving a low coupling should be one of the primary objectives in system design, such that individual module or microservices can be studied and altered without the need of taking into account a lot of information from other module or microservices and applying domain design concepts while designing a module or microservices.

Loose coupling leads to high cohesion and together leads to a highly maintainable, extensible, and testable system.

Software Quality Principles

> *"A good architecture is important; otherwise it becomes slower and more expensive to add new capabilities in the future. Good architecture is something that supports its evolution."*
>
> —Martin Fowler

As architects, designers, or programmers, we spend a lot of time analyzing, designing, and developing code, but we spend even more time maintaining that developed code. How often do we go back and find that the application has become a tangled mess? Sometimes we park that system as a legacy application.

The purpose of quality principles is to reduce complexity in a manageable way. Complexity can never be eliminated; however, architects and designers can reduce it by using quality principles.

Several problems lead to a highly complex and unmanageable system.

- The architect team does not analyze the business problems properly.
- The architect team does not have a clear view of what the end user wants from our application.
- The architect team does not have full visibility of the enterprise or business unit applications.
- The architect team may not get sufficient time to analyze the architecture.

- The architecture team is to embrace business and technology disruption.
- There are too many software programming languages and platforms with diverse features.

These complexities lead to several problems in software while creating an architecture.

- May cause the software to behave in an unanticipated state
- May create security vulnerabilities that could raise the management of application to an enterprises
- May lead to big operation team and end up with more cost
- May lead to a schedule overrun

Minimizing the complexity and improving the quality of software helps to eliminate or manage the difficulties. Some of the principles related to improving the quality and reducing the complexity are covered next.

KISS Principle

The *keep it short and simple* (KISS) principle was created by the late Kelly Johnson, who was the lead engineer at Lockheed Skunk Works. Kelly's version of the phrase was "Keep it simple, stupid." This phrase was embraced by Lockheed designers. There are many variants of KISS: "Keep it simple and straightforward," "Keep it super simple," etc.

We are using the phrase "short and simple" in this book of cloud native architecture. The objective of this principle is to deliver the simplest possible outcome.

Some of the famous quotes related to KISS are:

> *"Among competing hypotheses, the one with the fewest assumption should be selected"* —Occam's Razor
>
> *"Make everything as simple as possible but not simpler"* —Albert Einstein

This principle has been key for many years, typically when an architect or developer is breaking down an application into smaller pieces to address the business problems and then they think they understood the business problem and try to design and develop a particular problem but end up with complexity. Based on my experience. it is a complex process on how and where to break complex into simple.

Applying KISS to Software Design

Simplicity is a highly desirable quality in software applications. Making software more complicated than it needs to be lowers the overall quality of software. The maintainability, testability, and supporting the business disruption are reduced when complexity increases.

Here are some ways to follow the KISS principle in your day-to-day work:

- Focus on a simple solution that meets the requirements.
- Avoid the "Rolls Royce" solution when you need a low-end car.
- Break down your problems into many small problems. Each problem should be able to be solved.
- Apply design methodologies to solve the problem and then code it.
- Design the problem as easy to develop and easy to throw away; sometimes throwing away and re-creating is simpler and cheaper than maintaining it.
- Make it easier for the developer to visualize the various aspects of the application, mentally mapping the possible effects of any change. This involves knowing the dependencies and state of the application.
- Avoid abstraction and dependencies.
- Avoid flaunting. Most architects and designers flaunt their skills and knowledge, which makes design unnecessarily complicated.

Try to keep it as simple as possible. This is the hardest behavior pattern to apply, but once you have it, you'll look back and will say " I can't imagine how I was doing work before."

Don't oversimplify a design. Stop breaking things down when you reach a point that negatively affects the design of the application.

Don't Repeat Yourself

The *don't repeat yourself* (DRY) principle aims to reduce repetition in the software application. It says that every piece of knowledge must have a single, unambiguous, authoritative representation within a system."

This principle applies at the code level and architecture level. When code is duplicated across many packages in the application, it makes maintainability harder, and this leads to a bigger codebase that is difficult to modify. Finally, it becomes a technical debt. In an architecture decision, you don't need to build everything; use the packages or software already available on the market instead of building on your own.

Duplication Is Waste

Every line of code that goes into the system must be organized and maintained or it will be a potential source of future bugs. Duplication needlessly bloats the codebase, resulting in more opportunities for bugs and adding accidental complexity into the system. The maintainability of the code becomes a nightmare. It can lead to technical debt, and enterprises need to spend effort and time on refactoring the codebase.

The DRY Principle in Polylithic and Polyglot Architecture

When designing microservices, the DRY rule applies here also, as Sam Newman said in his book *Building Microservices*: "Don't repeat yourself inside microservices. The dilemma is about reusing across microservices. The basic principle of microservices is to "avoid dependencies between microservices." Even though there is a dependency, but it should be very minimal. As part of the microservices design, we need to reuse some utility or generalized functionality across microservices, but the challenge is how to find the right balance to apply the DRY principle.

As shown in Figure 3-35, one of the well-known approaches is to create a package as a library for reusing code and maintain the package separately outside of the microservices code, and then use well-structured build pipeline to include relevant libraries into the microservice package.

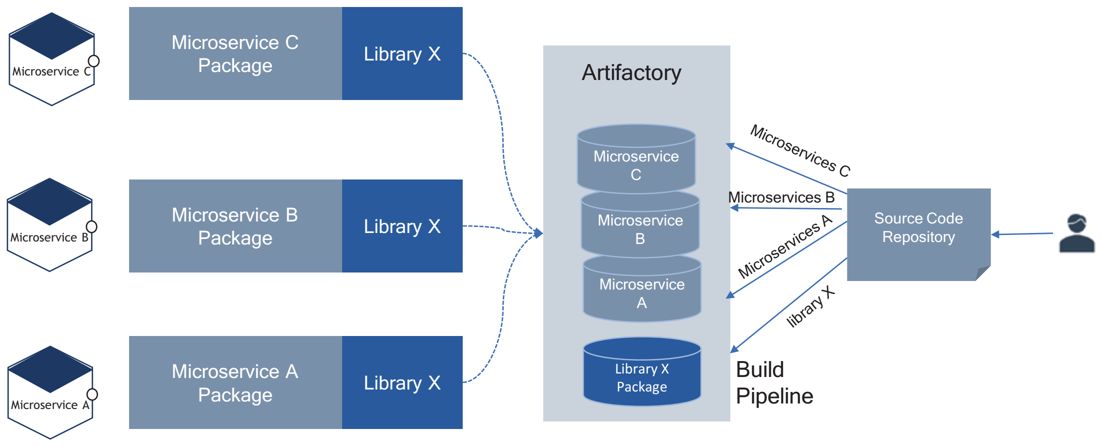


Figure 3-35. *DRY principle in microservices*

Let's examine several considerations when you are applying the DRY principle in the context of microservices.

- Use semantic versioning from the beginning of the project.
- Limit the code and functionality in libraries by design.
- Design a package such a way that the functionality of the code doesn't change often.
- Standardize on a naming convention so that other microservices teams can discover these packages.
- You need to set up proper governance to manage these packages

How does the DRY principle reduce maintenance costs?

If the code is duplicated and needs to be changed, you need to find all the places where it is duplicated and apply changes to all of them. This is more difficult than modifying in one place, and this leads to more errors and technical debt. You can think of it like you accidentally apply it differently in one location than another location, or you can modify code that happens to be the same. Duplicate code tends to obscure the structure and intent of your code, making it harder to understand and modify.

In some places, the DRY principle is good to follow, but some places need to maintain duplication due to unavoidable situations, but in that duplication, make sure you follow proper packaging such as using a library and governance to manage such code.

Isolate

When building a cloud native architecture, one of the primary goals is to achieve a degree of isolation between services within one business application.

When we design an application that is cloud native, the application should be fragmented into multiple, independently executing services. By physically separating services, we will introduce isolation between application modules, which allows us to reduce the coupling between modules or microservices and potentially increase each module's scalability.

What do we mean by isolation?

The concept of isolation means that changes to one module of the architecture generally don't impact or affect elements of another module. The change is isolated to the elements within the module, which does not have any knowledge of the inner workings of another module.

Isolation in Cloud Native Applications

When working with cloud native architecture, we focus on three dimensions of isolation: state, space, and failure.

One of the primary characters of a cloud native application is the state. The individual module or microservices are wholly responsible for maintaining the state; any access to this state from other modules is through REST APIs.

Space refers to the location in which modules are deployed. The deployment strategy changes radically for cloud native applications. In a cloud native architecture, the modules are deployed independently and execute with the separate process in containers. This allows each service to be managed independently. The ability to manage application elements independently allows remediating defects and new features to be deployed automatically with infrastructure as code.

Failure refers to how the application modules isolate the failure between modules. Each module in a cloud native architecture executes independently; the failure will no longer crash the entire application. During application design, avoid propagation of failure and try to use the Bulkhead pattern to both isolate and mitigate failure; the patterns are explained in Chapter 4.

Applying Isolation to Software Design

Software architects should design modules by following isolation principles. These best practices will help you while designing an application:

- Limit or restrict unneeded interactions or dependencies.
- Protect system integrity by preventing one process from interfering with another.
- Provide boundaries so individual failures do not compromise the whole system.
- Limit exposure to a particular area of the system.

Separation of Concern

Separation of concern (SoC) is a design principle that manages the quality and complexities of an application by decoupling the software system so that each isolated module is responsible for a separate concern, minimizing the dependency as much as possible. At a low level, this principle is closely related to the single responsibility principle.

The SoC involves decoupling larger problems into smaller manageable problems. It improves the quality of software by reducing complexity.

The term *separation of concern* was probably coined by Edsger W. Dijkstra in his 1974 paper "On the role of scientific thought."

In 1989, Chris Reade in his book *Elements of Functional Programming* describes SoC.

The programmer has to do several things at the same time, namely, the following:

- Describe what is to be computed.
- Organize the computation sequencing into small steps.
- Organize memory management during the computation.

Applying SoC to Software Design

SoC is achieved by establishing boundaries. A boundary is any logical or physical constraint that delineates a given set of responsibilities. Some examples of boundaries include the use of modules, methods, layers, and services.

At the design level, the application can follow an SoC by separating different elements such as user interfaces, APIs, database, business logic, etc. An example of a pattern is the Model-View-Controller pattern.

Use Layering

The most common principle is the use layering principle. This pattern was the de facto standard principle for all the web applications since the MVC pattern and has been in use for quite some time. In today's world, this principle is still relevant in cloud native architecture.

Elements within a layered architecture are organized into horizontal and vertical layers, and each layer within an application performs specific functionality. Although this principle does not specify the number and types of layers that exist, it all depends on what type of application you are developing.

Layering in Traditional Application

As shown in Figure 3-36, traditional software architecture consists of four standard layers: presentation, business, persistence, and database. A smaller application may have only three layers, and a large and complex application may have four to five layers.

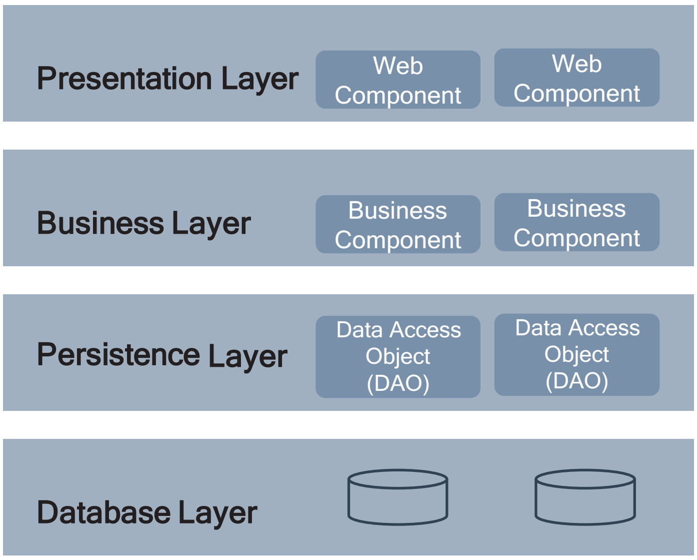


Figure 3-36. *Traditional architecture layering approach*

The presentation layer is responsible for handling all user interfaces, whereas the business layer is responsible for executing specific business functionality, the persistence layer is responsible for connecting and managing database access, and the database layer is responsible for storing information.

Layering in Cloud Native Application

A cloud native application is composed of various logical layers, as shown in Figure 3-37, and grouped according to responsibility and deployment. Each layer in a cloud native application runs specific tasks, and each task can be a separate microservice.

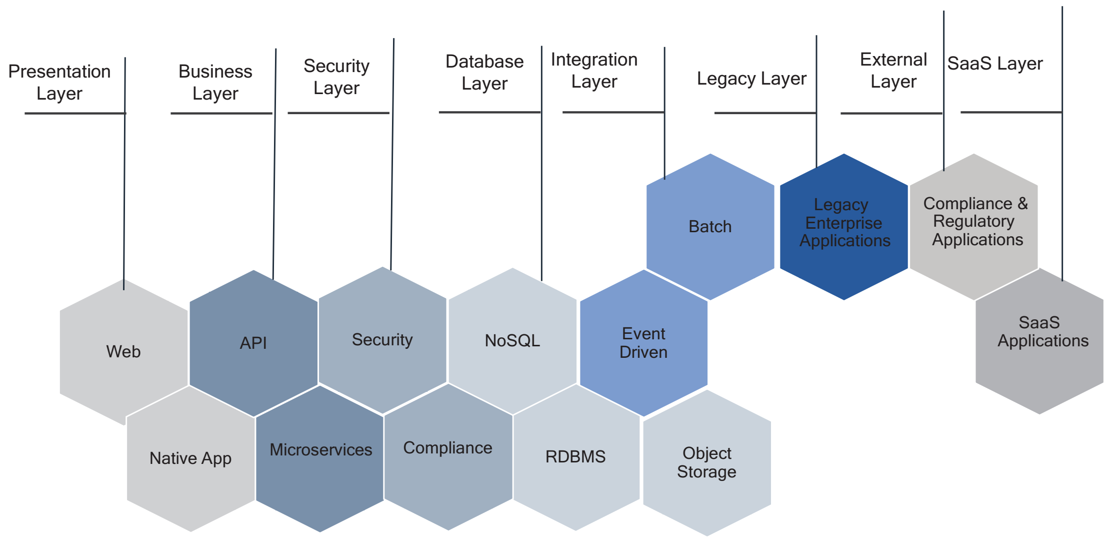


Figure 3-37. *Cloud native architecture layering approach*

The presentation layer provides a user experience through the web application and mobile native application.

The business layer runs stateless services that expose the API; this layer can dynamically expand and shrink depending on the usage at runtime by using the autoscaling option of the cloud.

The security layer provides security as a service to the entire application including access, security at rest, and security at transit.

The database layer has stateful services that are backed by polyglot persistence. Stateful services rely on traditional RDBMS, NoSQL, object storage, graph storage, etc.

The integration layer has an event-driven architecture and batch jobs; the event-driven architecture can use a variety of services in the cloud that connects various internal legacy and external third-party applications. The interconnection may be synchronous or asynchronous or batch.

Cloud native applications interoperate with the existing enterprise applications at the legacy layer; these legacy applications may host in the cloud or on-premises.

Cloud native applications interoperate with various third-party applications such as payment, regulatory systems, etc.

Some cloud native applications can interoperate with third-party SaaS providers; these SaaS applications may host in the same cloud or multicloud environment.

Applying Layering to Software Design

The layered architecture principle is general-purpose, making it a good starting point for most applications. You need to consider the following few points when you are applying the layering principle in your architecture:

- Divide and conquer by decomposing the system in different logical layers.
- Apply SoC when designing the layering approach.

Information Hiding

Information hiding focuses on hiding the nonessential details of functions and code in a program so that they are inaccessible to other components of the software. Software designers and developers apply information hiding in software design and coding to hide unnecessary details from the rest of the modules. The objective of the information hiding is to minimize complexities among different modules of the application.

The information hiding principle suggests the architecture be designed as modules or microservices in such a way that they hide implementation details from the consumers.

D.L. Parnas introduced the term *information hiding* in 1972 in "On the Criteria to be Used in Decomposing Systems into Modules." The idea was that each module should hide some design decisions from the rest of the system, especially decisions that would have cross-cutting effects if changed.

A well-designed system means that it needs to be well-organized. We presented various principles to achieve. You do not need everything in your system to know about everything else. So, how do you limit the information the various modules can have access to? Information hiding allows elements of the module to give accessors the minimum amount of information needed to use them correctly and hide everything else. Information hiding is often associated with encapsulation.

Why Information Hiding?

Information hiding is relevant in all levels of application; exposing only the details that are required improves the quality of the software and reduces the complexity. Most importantly, it improves maintainability and security. Hiding implementation reduces potential coupling and dependent modules, which will reduce the effect of the change on your implementation.

Applying Information Hiding to Software Design

Information hiding can be useful in designing your module and APIs. The gap between theory and practice in module design is wide, and among many designers, the decision about what to put into an API amounts to deciding what interface would be easiest to write internal code to, which results in exposing as much of the elements in the APIs as possible. I have seen that most programmers would rather expose all the elements and write extra lines of excess code to keep module secrets intact.

Asking about what needs to be hidden supports good design decisions at all levels. It promotes the use of named constants instead of literals at the implementation level. Get into the habit of asking "What does a consumer want?" or "What should I hide?" You'll be surprised at how many decisions vanish before you.

You Aren't Gonna Need It

You Aren't Gonna Need It YAGNI is an acronym that stands for "You Aren't Gonna Need It" or "You Ain't Gonna Need It." It is a principle from the Extreme Programming methodology. YAGNI states that you should prioritize the functionality in a backlog until it is completed.

Idea of YAGNI

The idea of YAGNI is that you should only implement features that are required and not just because you think you may require them sometime later. Ron Jeffries, the author and cofounder of XP, said this:

> *"Always implement things when you need them, never when you just foresee that you need them."*

Even if you are sure that you will need a feature or piece of code later, do not implement it now. Implement it when the feature required. Most likely, you will not need it after all, or what you need is quite different from what you foresaw needing earlier.

The reason you may consider building presumptive features is that you think it will be cheaper to build it now rather than build it later. Before making a decision, the cost and time comparison must be made against the cost of delay. Spending time and money on a feature you don't need now takes away time and money that are required for other immediate features. This doesn't mean you should avoid building flexibility into your application. It means you shouldn't overengineer something based on what you think you might need later.

The idea of YAGNI is that you save time because you avoid writing code that you do not need; our code is better because you avoid polluting it with guesses or assumptions that turn out be wrong and end up with technical debt and require refactoring.

How to Decide What You Need

Martin Fowler wrote in his blog, "YAGNI only applies to capabilities built into the software to support a presumptive feature; it does not apply to effort to make the software easier to modify." YAGNI is a viable strategy only if the code is easy to change, so expending effort on refactoring isn't a violation of YAGNI because refactoring makes the code more malleable.

In cloud native architecture, we are building loosely coupled independently deployable software with the principle of ease of maintenance, ease of test, and ease of extension. By considering this, the YAGNI principle is very relevant now. It means you can add any feature at any time without affecting the existing implementation. What you need is to manage backlog smartly so that the features can be mapped to the particular microservices features, this helps the team to pick easily for development.

SOLID Design Principles

SOLID principles are an object-oriented approach that is applied to software design and coding. It was conceptualized by Robert C. Martin in 2000, and the acronym was coined by Michael Feathers. These five principles are the de facto standard for OO programming.

The idea of the SOLID principle is to reduce dependencies so that developers can change one area of software without impacting others. These principles are intended to make designs easier to understand, maintain, and extend. Ultimately, using these principles makes it easier for software development to avoid issues and to build adaptive, effective, and cloud native software.

These principles have become important in cloud native applications. When followed correctly, you can achieve maintainability, extensibility, and testability of software design.

The SOLID principle is a framework consisting of complementary principles that are generic and open for interpretation but still give enough direction for creating a good object-oriented design. The SOLID is a mnemonic acronym for five design principles intended to make software designs more understandable, flexible, and maintainable.

SOLID stands for the following:

- Single responsibility principle
- Open-closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

Single Responsibility Principle

The single responsibility principle is one of the most tried and tested in software design. Every module or microservices should do one thing.

This principle states that each module should have a single responsibility or a single job or a single purpose. The responsibility of a module should have one and only one reason to change, meaning that a module should have only one job. This means each of your modules or microservices should serve only one greater purpose and change only if the greater purpose changes. It doesn't mean each module or microservices doesn't require you to stick just one task or contain only one unit of work, but these tasks or units or elements all need to cohesively relate to the greater purpose of the microservice.

If a microservice has multiple responsibilities, there is a possibility that it is used all over the place. When one responsibility changes in the microservices, then we need to test the entire set of responsibilities whether it is changed or not.

This principle is related to the separation of concern principle; as concerns are separated from each other, it facilitates the creation of microservices that have a single responsibility.

Applying Single Responsibility to Microservice Design

Microservices serve a single responsibility in a domain. As shown in Figure 3-38, each domain like Customer Account, Payment, or Quote is considered to be a microservice, so these domains serve a single responsibility. This domain model is from the Auto Insurance domain.

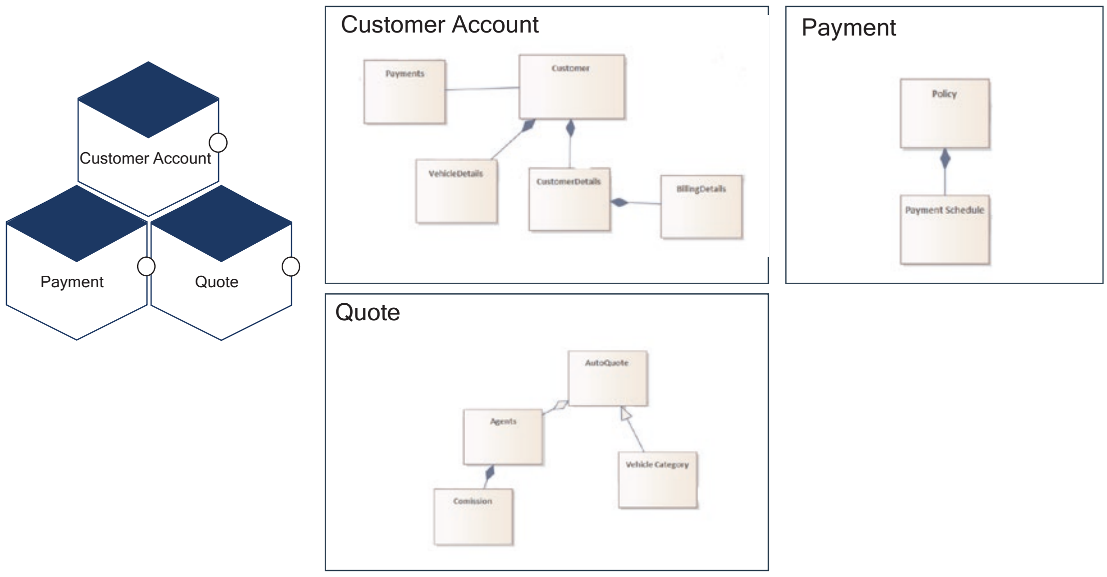


Figure 3-38. *Single responsibility principle in microservices design*

The Customer Account provides the functionality of customer account management like customer profile, vehicle details, payment details, customer details, etc. The Customer Account microservice invokes payment microservices to process a payment for insurance purchase based on the quote created.

Open-Closed Principle

The open-closed principle states that software entities should be open for extensions and closed for modification. When functionality changes, the entity can allow its code to be extended without modifying the existing code that has already been developed.

At a code level or class level, you should be able to extend the class's behavior without modifying it. This extension can be done by extending the class, either using inheritance or using composition.

At the architecture level, we are not modifying the functionality of an existing module but always add new elements by using the existing design.

Applying Open-Closed to Microservices

Even though this principle was created for object-oriented programming, this principle is still relevant in cloud native architecture.

In cloud native, you expose your functionality through either APIs or event-driven messaging. These APIs are contracted with the consumer, and you cannot modify the existing contract; instead, you extend it.

Let's move on to the specific example of insurance. As shown in Figure 3-39, imagine we work in an Auto Insurance domain, and you are building a new cloud native application. During the insurance process, the customer requests insurance by providing vehicle details and other customer details; your user experience invokes the customer microservices through APIs. You define an API contract between your customer account microservices and user experience (web and mobile native application) and to the third-party agent application.

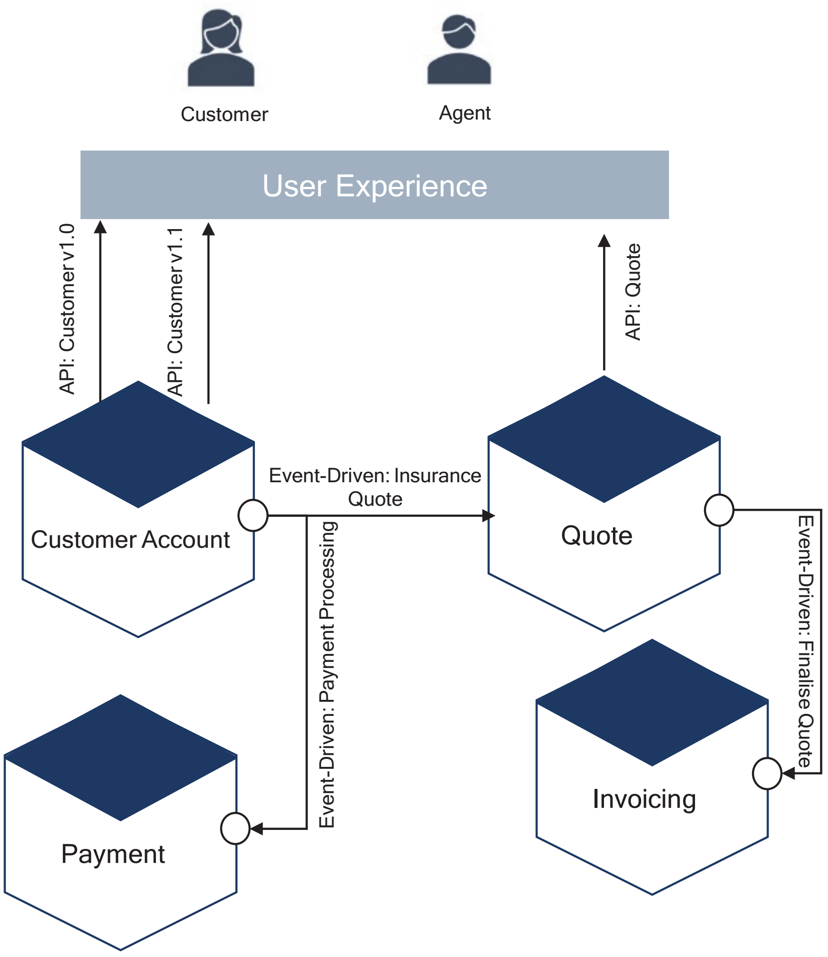


Figure 3-39. *Open-close principle in microservices*

Your business team would like to add new functionality for the existing one; in this case, are you going to modify a customer account microservices or add new functionality into the customer account? You will extend the functionality and provide a new API with the new version. Here you are doing open for extension and close for modification.

Liskov Substitution Principle

The *Liskov substitution principle* (LSP) defines that objects of a superclass will be replaceable with objects of its subclasses without breaking the application. This principle allows subclasses to inherit from a superclass, which includes the properties and methods of the superclass. This principle is like the design by contract concept defined by Bertrand Meyer.

In cloud native architecture, the design by contract is part of the API contract and relies on preconditions, postconditions, and invariants. The API contract is the contract of messages between your API provider and the consumer that will be used across channels.

Applying Liskov Substitution to Microservices Design

The LSP in OOP is to enable your code using type T1 to use type T2 instead, as T2 is a subtype of T1. In other words, you don't want to break existing code but alter behavior. If you apply LSP to microservices, you don't want to break existing clients of the service but replace them with better or enhanced ones.

We will use the same example as shown in the open-close principle with the modification of the API contract.

In the example shown in Figure 3-40, you need to find a way to replace the microservices Customer Account version 1.0 with version 1.1, not only breaking existing consumers but having them utterly unaware of these changes.

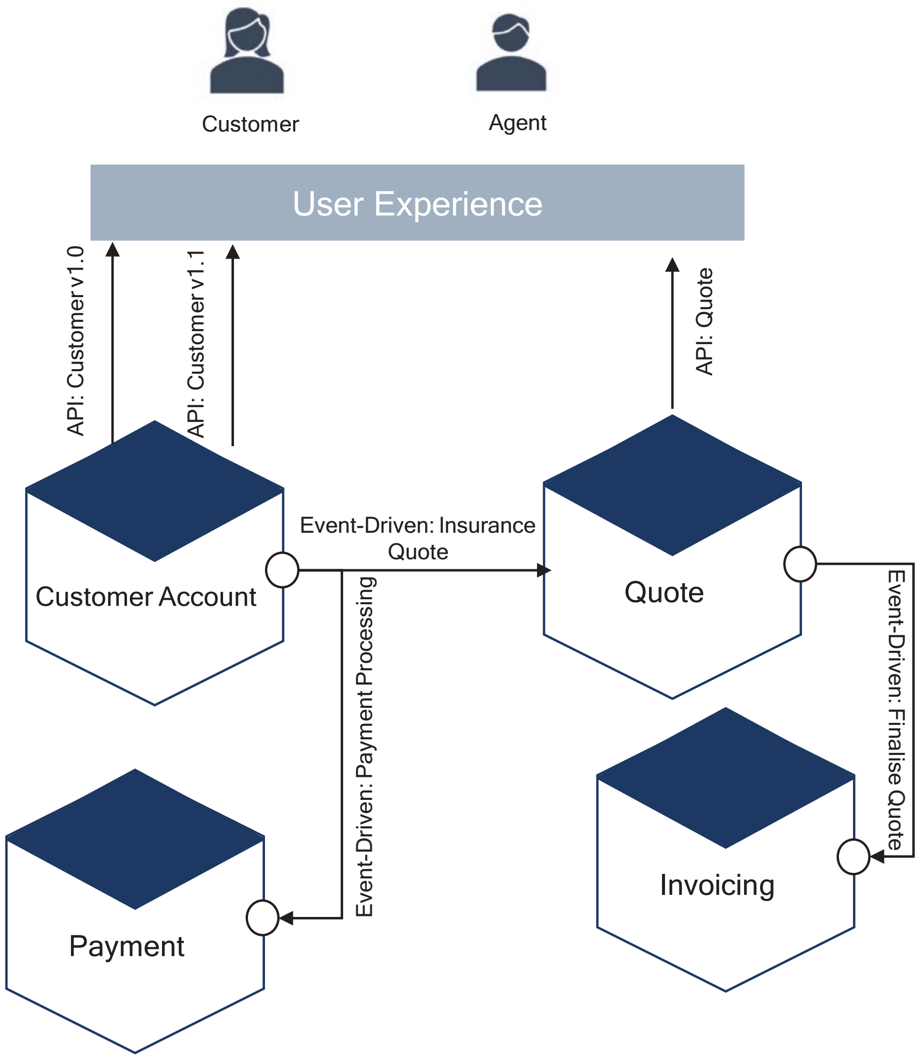


Figure 3-40. *Liskov substitution to microservices design*

The API contract for the Customer Account version 1.0 and the API version 1.0 of Customer Account is as follows:

GET https://mydomain/customer/resource-a
Accept: application/json; version 1.0

Some consumers want to add new features to the Customer Account microservice; here you need to extend it without affecting the existing customer. Here you need to introduce version 1.1, the API contract for the customer Account version 1.1, the API version 1.1 of customer Account as follows:

```
GET https://mydomain/customer/resource-a
Accept: application/json; version 1.1
```

Interface Segregation Principle

Interfaces in OOP define methods and properties but do not provide any implementations. Classes that implement interfaces provide an implementation. Interfaces define a contract, and consumers can use them without concerning themselves with their implementation details. The implementation can change, and if interfaces are not modified, the consumer does not need to change their logic.

In a cloud native architecture, an API is the interface between the consumer and implementation; the API provides the interface with properties and HTTP methods. Microservices that implement APIs provide an implementation.

The *interface segregation principle* (ISP) states that consumers should not be forced to depend on properties and methods that they do not use. This is exactly what an API implementation provides, you design APIs to provide an optional property with HTTP methods so that consumers can use only relevant properties and HTTP methods.

Dependency Inversion Principle

The *dependency inversion principle* (DSP) is a specific form of decoupling software modules for handling dependencies between modules and writing loosely coupled software systems.

The principle states the following:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

In cloud native architecture, you can use DSP to design your microservice's internal layers and decouple dependencies between the API, database, and infrastructure. It has nothing to do with your domain but is related to the application microservice design. This principle allows you to decouple the infrastructure layer from the application's deployment layers.

Summary

In this chapter, you learned various cloud native architecture principles and how to adopt these principles in a cloud native architecture.

To design the best cloud native architecture, several principles can be applied, such as API first, polylithic and polyglot, consumer first, a culture of automation, digital decoupling, evolutionary design principles, etc. After you design services, you must run these services in production. For effective runtime efficiency, several principles can be applied, such as isolate failure principle, deploy independently, be smart with the state, design for failure, etc.

Security is the most important part of any application, and cloud native architecture is no different. To implement effective security in an application, several principles can be considered, such as defense-in-depth, shift left in security, security by design, etc.

Once you design an application, the next most important part is how you develop and deliver the software, and a number of principles such as agility, shift-left, products not projects principles must be adopted.

The container is the de facto standard for cloud native applications; the effective configuration of containers in a cloud native architecture is important. Therefore, you need to apply container principles for your deployment using principles such as SCP, HOP, LCP, IIP, PDP, SCP, and RCP.

You learned that to design orthogonal software systems that can be extended while minimizing the impact on existing and new functionality, you need to focus on loose coupling and high cohesion. Complexity is an important concept in software application; architects and designer think they need to build the Taj Mahal or Eiffel Tower. But the customer wants something else; therefore, you need to apply these principles to make sure you deliver what the customer wants: KISS, DRY, information hiding, YAGNI, and SoC.

The SOLID design principles, which include SRP, OCP, LSP, ISP, and DSP, can be used to design and develop code that addresses maintainability, reusability, testability, and flexibility concerns. Several practices, such as agility, product centric, decentralization, and shift-left, improve the quality of software systems.

Cloud native architecture patterns are reusable solutions that can be used to solve recurring problems. In the next chapter, we will go over some of the common cloud native architecture patterns so that you will be aware of them and can apply them appropriately to your services.