**9. Designing for “-ilities”**

Shivakumar R Goniwada[**1**](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_9_Chapter.xhtml#Aff2)

(1)

Bangalore, India

So far, we have discussed cloud native principles, patterns, and elements of the cloud. In this chapter, I will explain the quality attributes for designing a system, i.e., how to design for your “-ilities” in a cloud native environment.

Developing the functional requirements in your application means addressing the business use cases, but what about the nonfunctional requirements? How will you address them? Developing your application to meet the nonfunctional requirements is as important as for the functional ones. In the cloud native world, you must prioritize the “-ilities” and address them at the beginning of the project like with the functional user stories.

You are responsible for cross-cutting concerns and making sure that the individual components of a system can work together seamlessly to meet the overall objectives.

In this chapter, I will explain what you need to consider when designing your system for the following “-ilities”:

* *Designing for security*: This is vital for a modern system in the digital economy, with the exponential growth across the globe, strict regulatory environments, and cyberthreats.
* *Designing for resiliency*: This is vital for modern-day, distributed applications, where any individual component could fail. The overall application should remain functional.
* *Designing for integrated observability*: This means providing the required behavior of an application across applications, infrastructures, and threat landscapes.
* *Designing for portability*: This helps you to design for multicloud and hybrid cloud environments across containers, VMs, and FaaS platforms.
* *Designing for sustainability*: The whole world is talking about global warming. In IT, we need to help the world reduce its carbon footprint by minimizing the compute usage or utilizing more sustainable cloud hosting options.
* *Designing for availability*: The application must be available to serve your customer, either from the application logic or database storage or from the deployment environment.
* *Designing for reliability*: The designed application must be reliable to serve all the requests in the stipulated time.
* *Designing for business*: However you design an application, you need to ask the question, for whom are you designing this application and for what reason?

**Why Do You Need “-ilities”?**

Developing an application has always been a complex task. Cloud native, modern-day architecture and distributed systems that are built using microservices, with event-driven architecture, and that are deployed onto a container with a serverless infrastructure yield many benefits but also introduce several new challenges.

Decoupling allows teams to iterate faster by adopting agility and automation, which provides flexibility to increase the quality, release faster to market, etc. However, there is an accompanying increase in the number and level of code changes, testing, and deployment required.

Along with the domain requirements of your system, you need to consider numerous factors, some explicit and some implicit, of a system and balance all concerns optimally.

The following “-ilities” help you to develop quality systems, offer customers a great user experience, implement security, and meet customer vision.

**Partial List of “-ilities”**

Business and domain requirements exist along with “-ilities.” These “-ilities” can alter the decision process for what and how to develop a system. Table [9-1](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_9_Chapter.xhtml#Tab1) is the partial list of “-ilities.” When developing a system, you must determine the most important of these “-ilities.” However, many of these “-ilities” oppose one another. For example, achieving both high performance and high security can be difficult.

Along with the traditional “-ilities,” I have given equal importance to sustainability and ethics to support our planet and humans.

***Table 9-1***

List of “-ilities”

| Security | Scalability | Availability | Operability | Sustainability |
| --- | --- | --- | --- | --- |
| Performance | Fault tolerance | Integrity | Testability | Maintainability |
| Extensibility | Usability | Portability | Agility | Debuggability |
| Interoperability | Simplicity | Ethics | Flexibility | Stability |
| Resilience | Inspectability | Robustness | Efficiency | Tolerance |
| Modularity | Coupling | Cohesion | Degradability | Cloudability |
| Self-healing | Self-sustainability | Observability | Autonomy | Auditability |
| Learnability | Changeability | Provability | Durability | Composability |

**Designing for Security**

Designing and developing cloud native systems that are secure is of vital importance. A system that does not follow secure practices creates vulnerabilities that can be exploited by various threats. The result of a threat can be unauthorized access to your system. A secure system can prevent and protect against malicious attacks and unauthorized access to the system. It is your responsibility to design a system that protects against malicious attacks.

Cloud native is fundamentally new, and using an existing approach to designing and building applications raises a security challenge, because different systems have different security requirements. It is important to understand the security needs of the system, so your approach should be radically different.

Cloud native security necessitates a refocusing on security that operates in step with the overall cloud native strategy of your organization. The key emphasis of cloud native security needs to ensure that vulnerabilities are identified and remediated during development. The approach you are adopting must be holistic and should be baked in through the software engineering lifecycle including operations. You should strive to create cloud native applications that are secure by design.

Using the following proven and new concepts of security principles and methods can make your application more secure.

**Defense in Depth**

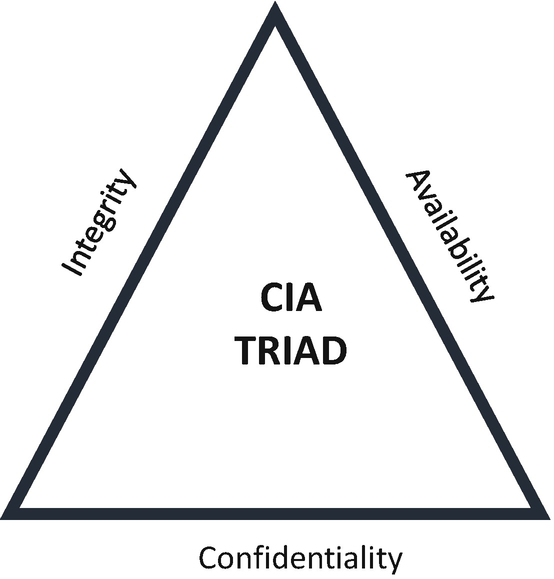
Security is more effective when each layer of the system architecture implements its own security techniques, so your strategy should be to adopt different methods for each layer. This redundancy ensures that if an outer layer is breached, then the subsequent layer can potentially thwart an attack.

In cloud native architecture, all components have asynchronous and synchronous communication. Therefore, each layer must be configured with well-matured security controls. If one security control fails, a threat may be prevented by another security control. Security controls in each layer with independent methods will make it much more difficult to exploit a vulnerability.

**The CIA Triad**

The CIA triad is the governing principle for information security and the protection of assets. The CIA triad, as shown in Figure [9-1](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_9_Chapter.xhtml#Fig1), summarizes the attributes that you want your system to exhibit. CIA stands for confidentiality, integrity, and availability.

* **Confidentiality**
* Systems must protect confidentiality. The information that a system manages has value to users, and you must prevent unauthorized individuals from accessing information. The system must protect the data, APIs, etc.



***Figure 9-1***

CIA triad

* **Integrity**
* The system must ensure integrity. The objective is to prevent unauthorized individuals from modifying the information. You must design your application to ensure that the data has not been tampered with by unauthorized individuals.
* **Availability**
* Systems need to maintain availability, and you must design a system that allows authorized individuals access to information in a timely and reliable way. Securing data serves no purpose if authorized users cannot access it.

**Policy as Code**

The technology landscape is becoming more complex and agile, and manually managing each policy has become more erroneous. Therefore, security needs to be implemented with automation and well-defined engineering practices. When you are developing a new cloud native system, you need to take into consideration the security policies, which are the rules and procedures to protect your systems from threats and disruptions.

Policy as code codifies the security policies, for example, access control, so this policy can be treated as a test. When I say code, you do not need to write the code yourself in a programming language; you write the code in a configuration file and apply practices such as keeping the code under version control, configuring these files as part of your DevSecOps pipeline, automatically deploying by using infrastructure as code, and configuring observations to your software artifacts.

Tools such as HashiCorp, Open Policy Agent (OPA), istio, and all the major cloud vendors support policy as code.

**Zero-Trust Security**

Traditional network security is based on the castle-and-moat concept, where external individuals are restricted and everyone inside the network is trusted by default. The downside of this approach is that sometimes internal individuals become offenders. Or, if external resources gain access, then they can get access to your application and code. In cloud native applications, your systems are decoupled into multiple microservices and communicate synchronously and asynchronously. To secure your systems from all kinds of network threats, you need a zero-trust model.

A zero-trust model is a network security model, based on a strict identity verification process. This model doesn’t trust anyone either internally (inside your network) or externally (outside your network) or a machine to access your application and code. In this model, you need to create a strict identity verification for every individual and device.

You need to adopt the following methods to implement zero-trust security:

* Implement a strong verification mechanism by using identity and access management for both system and network access.
* For the principle of least privilege; each access is granted only much as needed. To implement this, you need to categorize your application components based on access privileges.
* Segment your network into microservices by breaking up security perimeters into smaller zones to maintain separate access for separate parts of the network.
* Use multifactor authentication to require more evidence to authenticate a user.
* Use strict control for device access and implement the strict monitoring and auditing of every access.

**Decentralized Identity**

Challenges of the current centralized identity provider are all-access ties to the single centralized identity and access management with the top-down approaches. This was good for traditional IT, where your applications are deployed in a single monolithic and one data center.

In a cloud native architecture, elements like microservices, data meshes, and event meshes enable decentralized implementations, but identity management remains centralized. In a decentralized approach, discrete identifiable units such as people, organizations, and things are free to use any shared root of trust.

This is an emerging concept in a cloud native world; here you will give back control of identity to consumers through the use of an identity wallet in which they collect verified information about themselves from certified identity issuers such as a Social Security number (SSN) in the United States, Aadhar number in India, etc. The Decentralized Identity Foundation is leading the way to conceptualize the implementation.

**Validating Input**

A software vulnerability can be avoided by being diligent about validating input from any untrusted sources. Whether it is user input or APIs or events, your application must validate all the input before processing it further. In the case of external integration, that system may have security policies and standards that differ from yours, so the system much check the data that it receives from external integration or extended enterprises.

**Design for Threats**

Threat modeling is a structured approach for analyzing security in your system. A threat is the most vulnerable to the system and may cause serious harm to it. Therefore, you need to create a threat model during design time. Threat modeling is a process that identifies and prioritizes potential security threats so that you can develop and test potential threats. Threat modeling evaluates threats to reduce a system’s overall security risks.

For cloud native and modern architecture, use threat modeling to focus on security from the attacker’s viewpoint.

**Naive Password Complexity Requirements**

All enterprises are set up with a password policy that has a mix of capitals, lowercase, and special characters, a length from 8 to 15, and more. In this policy, people will forget to remember this complexity and end up using a more insecure password. According to the National Institute of Standards and Technology (NIST), password length is the primary factor in characterizing password strength. Passwords that are too short or too common will encounter brute-force attacks as well as dictionary attacks using words.

**Compliance as Code**

This is about building compliance into development and operations and writing compliance policies and checks and auditing into the DevSecOps pipeline so that regulatory compliance becomes an integral part of automation. How to implement compliance is described by James DeLuccia and his team in “DevOps Audit Defense Toolkit.” This toolkit provides many details with real implementation scenarios.

The Chef Compliance tool from Chef scans infrastructure and reports on compliance issues, security risks, and outdated software, etc.

**Shift-Left Security**

*Shift-left security* applies 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).

Shifting the security review process left requires a new way of developing the application compared to the traditional approach. These changes are not significant deviations. You need to add the following process for shift-left security:

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

Configure static application security testing (SAST) and dynamic application security testing (DAST) as part of the DevSecOps pipeline, and implement container security to check the vulnerability at the early stage of the software development lifecycle.

**Single Pane of Glass for Audit**

Logs are essential components for helping to secure cloud native applications. Design your application for integrated centralized log management, operations, searching, and analysis. With this, you can use logs for detecting security threats, alerts, and notifications in an environment.

Through a single pane of glass, the tools provide you with holistic, business-level visibility across all environments. I will explain more about integrated monitoring in Chapter [19](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_19_Chapter.xhtml).

**Homomorphic Encryption**

Most encryption schemes such as Advanced Encryption Standard (AES), Rivest-Shamir-Adleman (RSA), Triple Data Encryption Security (DES), and Twofish consist of key generation, encryption, and decryption. Symmetric key encryption schemes use the same secret key for both encryption and decryption, and asymmetric key encryption schemas use a public key for encryption and a secret key for decryption. Both symmetric and asymmetric encryption can be used to secure data at rest and transit. Any outsourced computation will require such encryption layers to be removed before computation can take place. Therefore, cloud services providing outsourced computation capabilities must have access to secret keys and implement access policies to prevent unauthorized employees from getting access to these keys.

*Homomorphic encryption* (HE) refers to encryption schemes that allow the cloud to compute directly on the encrypted data, without requiring the data to be decrypted first. The result of such encrypted computations remains encrypted and can be decrypted only with the secret key of the data owner. Do not use HE for everything. It is a generic technology, so use only wherever computation data is possible on encryption. You can consider using federated machine learning results. Still, the industry is researching to standardize it. There are various software available for HE like SEAL, Lattigo, and HElib.

**Fail Securely**

Failures are bound to happen for any kind of system; therefore, you need to design your system to fail securely. This involves several things, such as using secure defaults, restoring to a security state, and always checking return values for failure. The confidentiality and integrity of your system should remain even though availability has been lost. Access must be restricted to privileged objects during failure. Application code should be written in such a way that there is proper exception and error handling and predictive analysis to alert required stakeholders.

**Secure APIs**

Your APIs are exposed outside of your network to transfer data. Broken, exposed, or hacked APIs are behind the major data breaches. You need to secure your API, but not all APIs require some kind of security. Open Authorization (OAuth 2) or Open ID Connect are open standards for authorization. They allow an application to be granted access to resources from the consumer. Open ID is an identity layer that sits on top of OAuth2 and OpenID Connect to enable authentication and authorization. Use a JSON Web token (JWT) along with your APIs to securely transmit information between the provider and the consumer as a JSON object.

There are no silver bullets when it comes to implementing security; however, there are proven principles and practices you need to secure your application. There are more techniques other than these; refer to the respective techniques in more detail while designing an application.

**Designing for Elasticity**

Elasticity is the degree to which a system can adapt to changes in demand by provisioning or releasing resources autonomously. The microservices, containers, and Kubernetes are built for elasticity. You need to design microservices to enable a view on resources as an infinite pool and give the ability to scale the deployed containers out and in depending on demand. To keep the costs to a minimum and quality objectives as promised in your client SLOs, an adaption process must exist that alters the number of container instances based on demand.

While you design an application, you should adopt the following principles:

* Design for stateless.
* Adopt a sidecar pattern.
* Make it independently deployable.
* Use the sharding principle for a database.
* Use autoscaling options from cloud providers.

You can find more information about elasticity in Chapter [5](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_5_Chapter.xhtml).

**Designing for Resilience**

*Resiliency* refers to the ability of a solution to absorb the impact of a problem in one or more parts of services while continuing to provide an acceptable service level to your business. A resilient application must thrive even when the unexpected happens. In other words, it provides the required capabilities despite excessive stresses that can cause disruptions. The residual defects in the software or hardware will eventually cause the system to fail to correctly perform a required function or cause it to fail to meet one or more quality attributes of microservices such as availability, security, performance, reliability, usability, etc. An unknown or uncorrected security vulnerability will enable an attacker to compromise the system.

The question is how to design for automatic self-healing and application resiliency. As mentioned in Chapter [5](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_5_Chapter.xhtml), microservices are always be on partial failure with more load. But how can the designer approach microservices resilience? Approaching for resilience is not a one-time activity but is a continuous plan, culture, and work during the entire lifecycle of a microservices.

The following patterns can help you to design resilient microservices:

* Circuit breaker pattern
* Bulkhead pattern
* Stateless services
* Retry
* Fail fast
* Timeout
* Throttling

You can find more detailed information on resilience in Chapter [6](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_6_Chapter.xhtml).

**Designing for Sustainability**

*“Sustainability consists of the strategies and actions your enterprise takes to reduce its carbon footprint and consumption of the planet’s resources so that it is not sacrificing the health and happiness of future generations to meet its own needs today.”—Forrester Research*

Designing for sustainability is an innovator trend, and people are realizing the software industry is responsible for a high level of carbon usage comparable to the transportation industry. Some of our day-to-day activities are directly measurable, as compute usage is highly correlated to energy consumption.

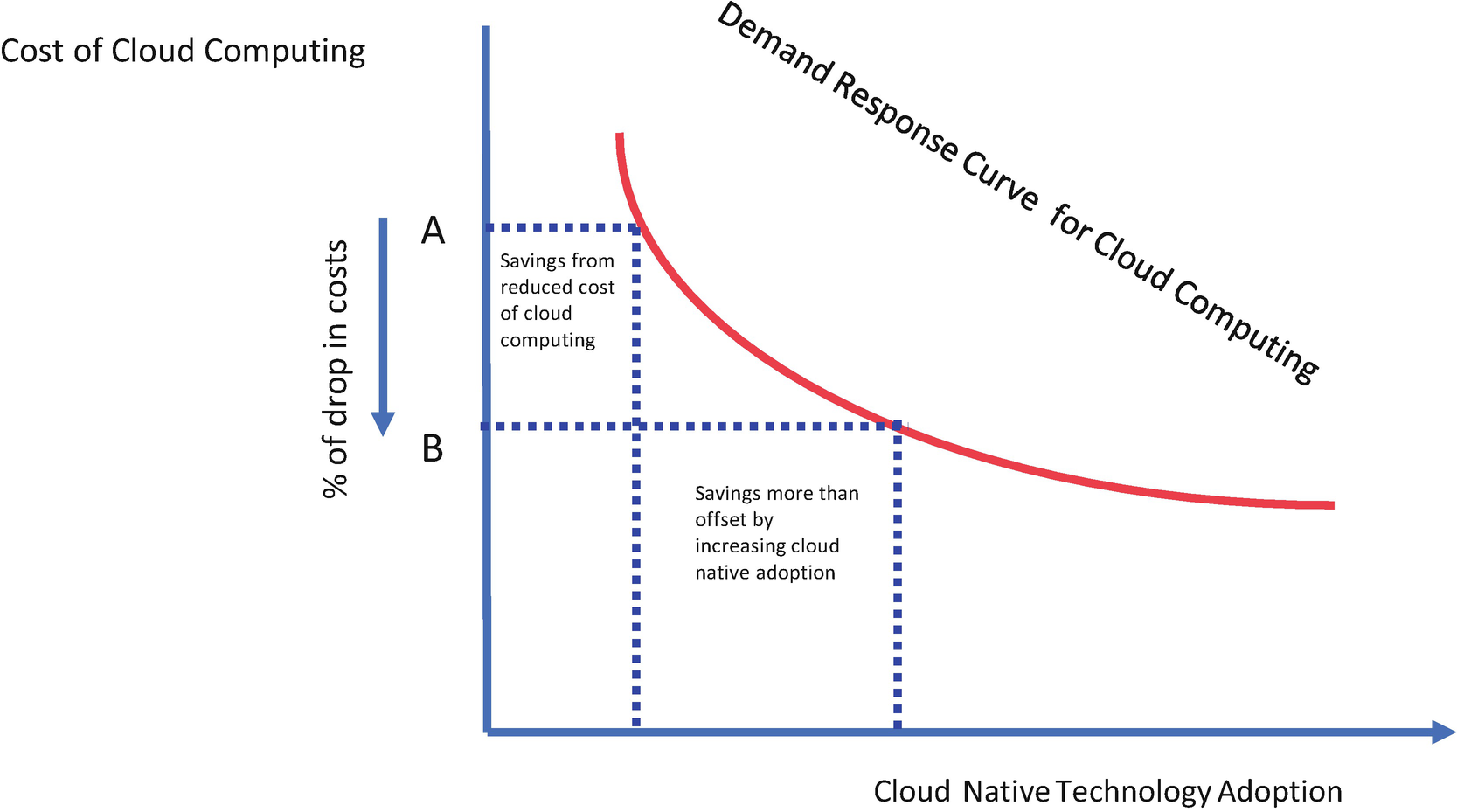
In IT, energy is needed in the following areas:

* Creating, testing, launching, and maintaining applications
* Hosting and serving applications
* Interacting with users in applications

**The JEVONS Paradox in Cloud Native**

The Jevons paradox is an economic term coined in the 19th century by economist William Stanley Jevons.

The Jevons paradox, as shown in Figure [9-2](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_9_Chapter.xhtml#Fig2), occurs when technological progress or government policy increases the efficiency with which a resource is used and increases the demand and subsequent consumption due to what he called the *rebound effect*: when something is cheap and convenient, more people want it. This theory was for coal usage, and he observed that technical advancement increased efficiency and reduced prices.



***Figure 9-2***

JEVONS paradox theory

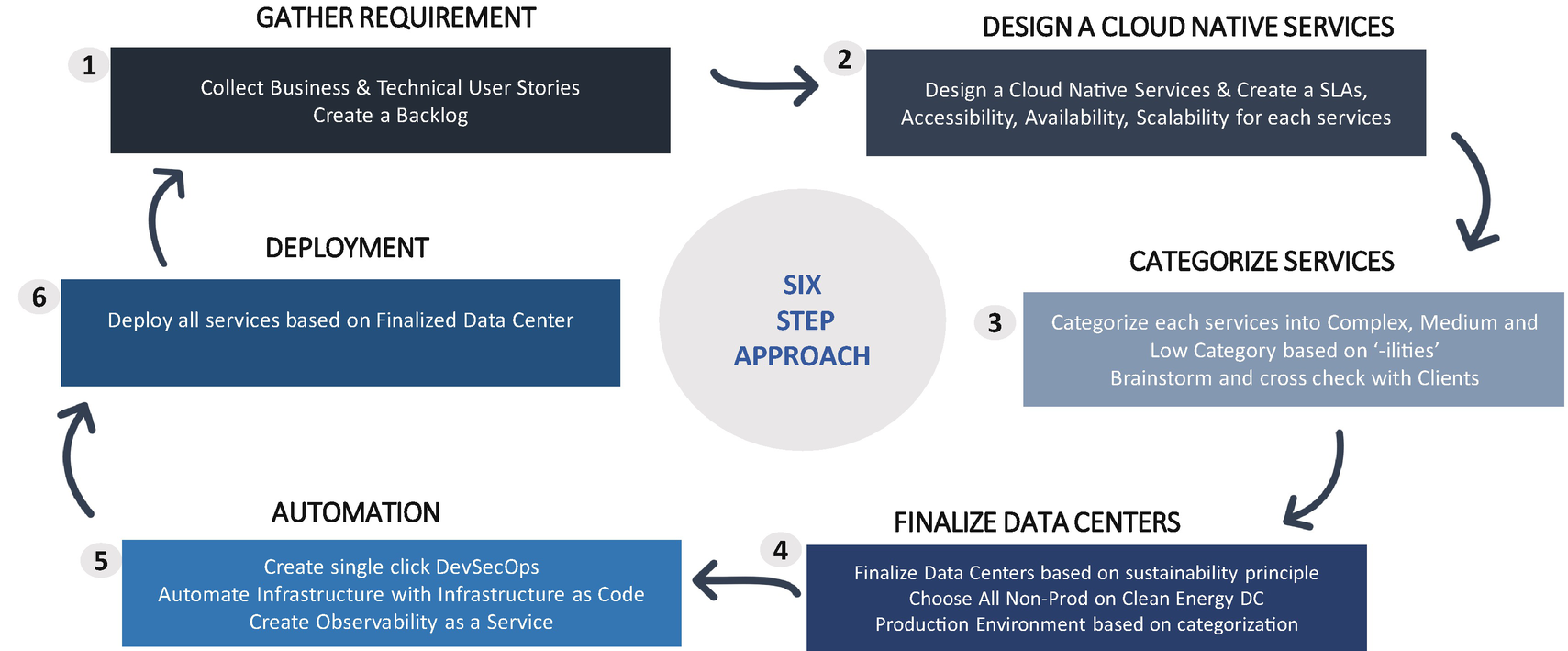
His original theory was about coal, but his paradox can be applied to almost all resources and is especially relevant in present-day cloud native architecture.

**Sustainability Approaches**

The following approaches will help you to design your system for sustainability, as shown in Figure [9-3](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_9_Chapter.xhtml#Fig3):

* *Net zero transitions*: Net zero carbon targets are no longer optional. The challenge is making them real and visible. Your organization must be rapidly progressing toward goals. You unleash the potential of cloud native to transform business models for the better.
* *Sustainable IT and technologies*: Cloud native technology is a true enabler of sustainability, but its energy consumption footprint is vast. You must use technology more sustainably, as well as use technology as a vehicle for being more sustainable.
* *Sustainable consumer experience*: Today, consumers demand sustainability. Your organization combines deep technology experience to help clients deliver consumer experience without compromising the user experience.
* *Culture of sustainability*: This means creating the mechanisms and cultures that bake sustainability into everything we do. Your organization must use the design thinking workshops and transform the way people design, work, and deliver systems.
* *Sustainable assessment*: Use tools, techniques, and methodologies to help organizations understand where they stand and help them to realize the sustainability journey.

Figure [9-3](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_9_Chapter.xhtml#Fig3) shows the six-step approach to build a sustainability system.



***Figure 9-3***

Six-step approach

**Deployment Environment**

Where you deploy your application is a significant part of sustainability. In a cloud native architecture, usually the cloud environment will be a deployment environment. All the major cloud vendors have data centers across the globe, but not all data centers are running with clean energy. So, you need to design your application to use the respective cloud environment appropriately and choose clean energy data centers.

In your application, not all use cases are highly critical, so you need to categorize the use cases as critical or noncritical with higher latency. The noncritical use case can be deployed in any data center that runs with clean energy, and the critical use cases that require low latency can be deployed in nearby data centers, which may or may not run on clean energy. For effective deployment management, you need to create infrastructure as code to automate your service based on criticality.

**Software Engineering**

The software engineering methodology plays a critical role in sustainability; I am talking about how you develop, test, and maintain your system. Use agility, next-generation automation, and AI-driven development to minimize the time and resource usage for software development.

**UI Architecture**

* *Web pages*: According to the HTTP Archive and its page weight report, the average size of a website is around 2MB, and the average load time is 4.7 seconds for desktop and 11.4 seconds on mobile apps. When served up on a sluggish Internet connection with 2G, 3G, and 4G or mobile devices with slower processors, these pages waste time and energy and frustrate users. We are inclined toward high resolutions and multi-image carousels.
* *Social media*: According to Statista, the average daily time spent on social media by users is around 145 minutes per day.
* *Video streaming*: With more than 4.5 billion Internet users in the world as of this writing, YouTube streaming around 260MB/hour worldwide, and Netflix accounting for around 12.6% of total Internet traffic as of writing, the subscriptions for OTT platforms are increasing daily.

The total energy consumed by cloud computing is more than many countries in the world. You need to design your application effectively based on what you need to display and how much data you need to show to the user.

**Sustainability Assessment**

You can consider the previous design adoption for new projects. What about existing applications? How can you assess and improve the sustainability?

Assess the existing IT estate with the existing technology portfolio to establish comprehensive technology-driven sustainability. Follow these four steps to conduct an assessment:

1. 1.

*Current state assessment*: The objective of this initial step is to identify the current system landscape and deployment model: interview stakeholders, consolidate systems and criticality, and write a regulatory landscape report.

1. 2.

*Gap analysis*: The objective of this step is to consolidate the report of the existing landscape and map against a sustainability chart: list and define projects and criticality, map them against the sustainability data centers, and list automation gaps.

1. 3.

*Financial constraints and sustainability*: The objective of this step is to consolidate the sustainability report and financial constraints: list the carbon and sustainability measures per workstream, benchmark reports, list ambitions and target values, and set a sustainability transformation scope.

1. 4.

*Roadmap, project charter, and recommendations*: The objective of this step is to define a path for a roadmap to project realization, including project cost estimates and project organizations.

You need to conduct an audit by asking the following questions:

* What are you trying to accomplish? List the objectives and SLAs and SLOs for each business use case.
* What are we trying to assess? Do an inventory of all use cases and categorize them.
* What is the impact of our inventory? Do the impact assessment. List the present inventory and what changes are needed to apply sustainability, include how much CO**2** is generated based on where it is deployed, and check whether the hosted environment uses fossil fuels or green energy.
* What does the data tell us? Create a target architecture and create a plan to move from fossil fuels to a clear energy hosting platform.

**Designing for Failure**

A cloud native architecture might fail for a variety of reasons, such as bugs in your code, unstable deployment, poor underlying infrastructure, resources saturated by load, unhealthy underlying nodes, faulty data center, or network between services failing. Lastly, human error can lead to major failures. You might have seen recent outages on Google services, Azure India availability zones, etc.

It is impossible to eliminate failure in a cloud native architecture; the cost of that would be infinite! Your focus should be on designing services that are tolerant of dependency failures and that are able to gracefully recover from them to mitigate the impact of those failures on their responsibility. You need to understand the different types of failures they might be susceptible to. Understanding the nature of these risks and their likelihood is fundamental to both architecting the appropriate mitigation strategies and reacting rapidly when an incident occurs.

These are the following areas of failure you need to consider while designing your services:

* *Infrastructure*: The underlying infrastructure on which your service operates such as containers and VMs
* *Communication*: Collaboration and coordination between various services through APIs and events
* *Dependencies*: Failure independent services
* *Internal*: Errors within your service

**Infrastructure**

Regardless of where your services are deployed, the reliability of services depends on the infrastructure that underpins them. The sources of failures in the infrastructure are hosts, data centers, networks, operating systems, etc. The failure in the infrastructure may affect the operations of multiple services in an application. You need to design your application with redundancy to mitigate infrastructure failure that might happen in one availability zone. You need to balance the redundancy because it incurs additional costs to your project.

**Communication**

Communication between services or external third-party services may fail. The source of communication failure might be firewalls, messaging, network, etc. These failures are common. You need to design your service to maximize availability, correct operation, and recovery when it occurs. To mitigate these errors, you need to configure your services with a retry mechanism for asynchronous or proper error mechanism for API implementation. Along with this, consider using circuit breaker, communication brokers, fallback, and other patterns.

**Dependencies**

Failure can occur in other dependent services or databases. Failures are related to timeouts, external dependencies, overload on other services, etc. You need to use various patterns such as a circuit breaker, timeout, retry, etc., to mitigate dependencies on other services and to provide the consumer experience as a whole system.

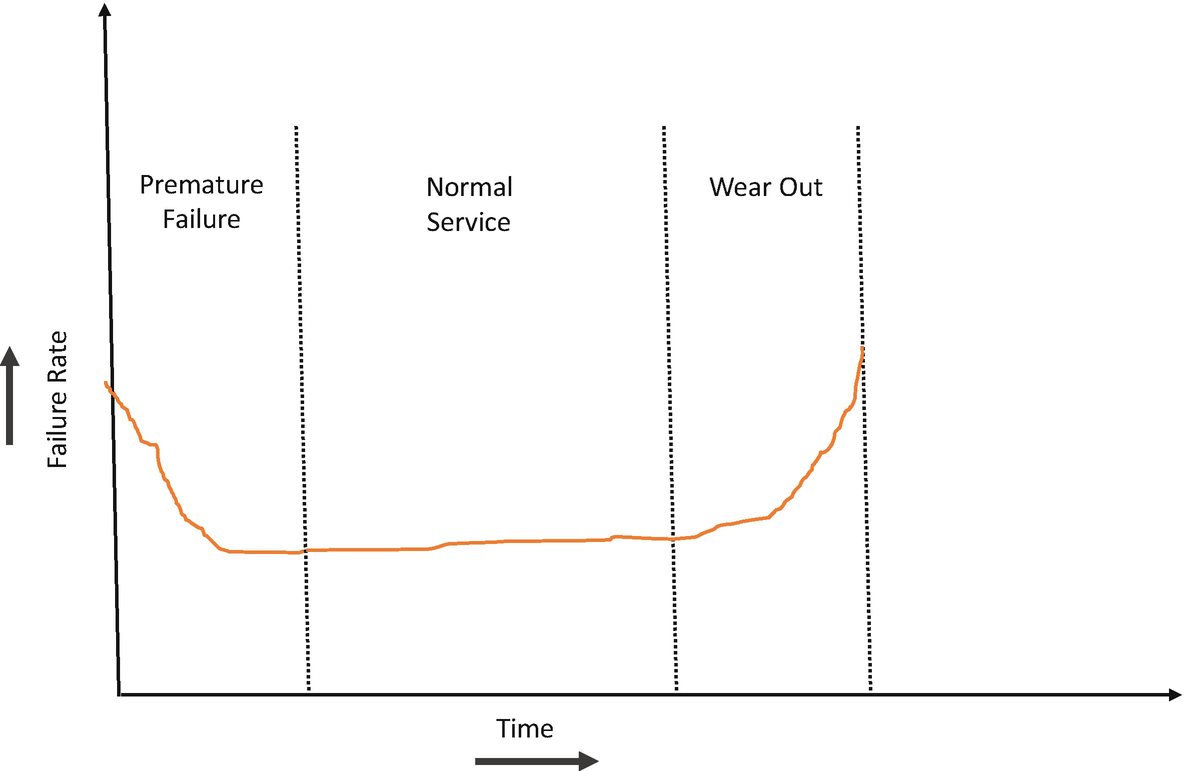
**Internal**

Inadequate software engineering practices might lead to failure. Services might be poorly designed or developed, inadequately tested, have improper deployments, etc. This leads to memory leaks, improper CPU usage, erroneous programs, etc., which leads to performance degradation. You need to design your service with software engineering best practices like shift-left, automation, testing, etc., and also adopt self-healing, graceful degradation, etc.

**Designing for Reliability**

Reliability is the probability that your system will continue to work normally over a specified interval of time, under specified conditions. For example, your Payment service might have a reliability of 99 percent during business hours; it has a 99 percent chance of working normally during this time. A more reliable system requires less maintenance. The reliability is design-centric, i.e., how reliable your system is comes from how you design your system. This is the reason we collect the reliability measurement at the start of the architecture and design it as a nonfunctional requirement.

Failures in your services are normally distributed as shown in Figure [9-4](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_9_Chapter.xhtml#Fig4). Different services and failure rates will apply to different kinds of services, but generally, all services behave like Figure [9-4](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_9_Chapter.xhtml#Fig4) irrespective of what kind of services you have.



***Figure 9-4***

Bath tub curve

As shown in Figure [9-4](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_9_Chapter.xhtml#Fig4), the graph goes by the name the *bathtub curve* because of its characteristic shape. The highest failure rates correspond to premature failure and end-of-life wear-out. Your services might fail after some time due to various conditions; this is understandable, but what about premature failure? The premature failure is the result of bad design and development. This can be eliminated by identifying and adopting good design practices.

The premature failure can occur as mentioned here:

* Services are designed and developed well but inappropriately deployed.
* Services are not built properly in the DevSecOps pipeline.
* Services are not managed by the operation team.
* Overall, service design is not good and introduces unnecessarily high throttling.

The normal service phase failure can occur as mentioned here:

* Failure due to natural calamities
* A sudden spike in a request, for example, during Black Friday time

The wear-out service phase failure can occur as mentioned here:

* Aging of infrastructure services
* The old version of adopted software

Simply identifying a failure by using observability is a waste of time and process. Eliminating failures saves money and time.

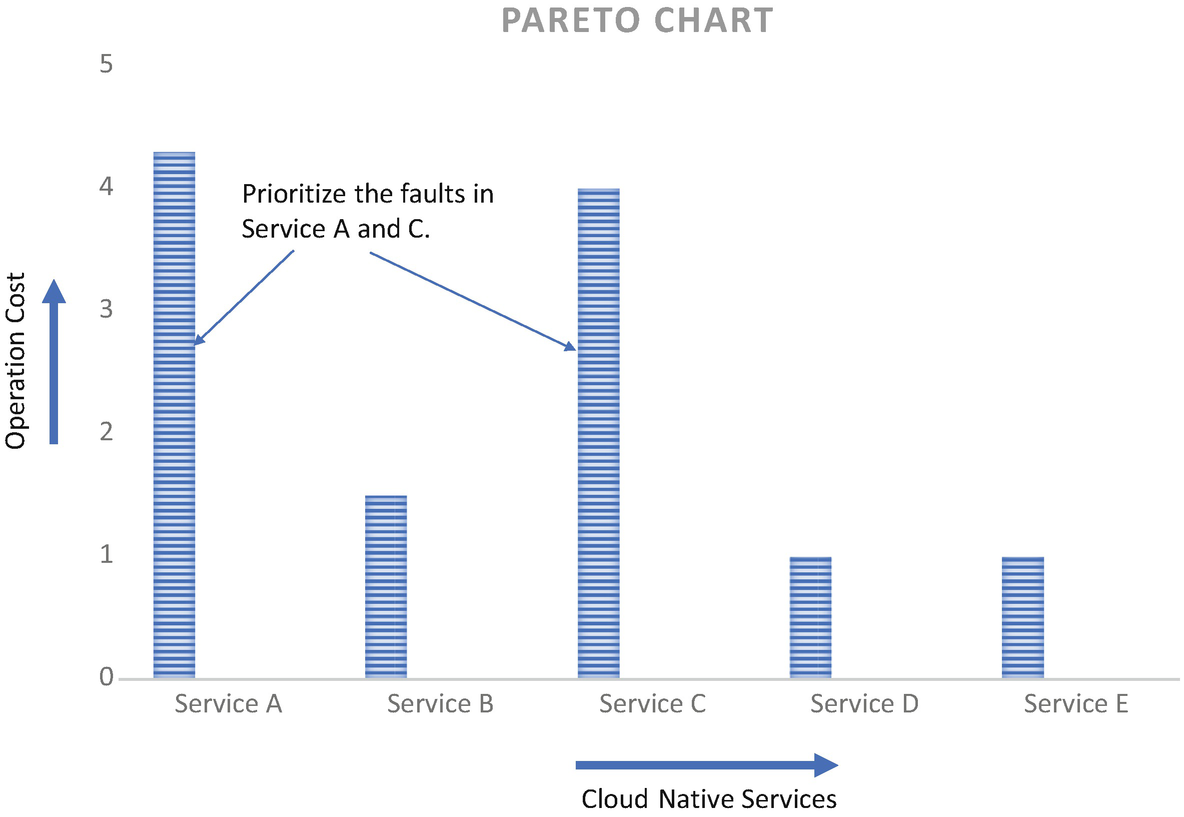
Good architectural design is the key to extending the service’s operational lifetime, which is the bottom of the bathtub. The best approaches are as follows:

* Follow a minimum viable product (MVP) to evaluate your design before industrializing the design across all services.
* Apply this knowledge and create a template and best practices for all the services’ agile Product Oriented Delivery (POD) teams.
* Create automation in every step; this helps to quickly pivot if any failure occurs.
* Create redundancy like a disaster recovery (DR) and replica set for databases.

**Pareto Chart**

Apply Pareto analysis (the 80/20 rule), where 20 percent of service faults in the system are responsible for 80 percent of the failure cost. Prioritize to identify this 20 percent and provide an early remedy. Use a Pareto chart.

Fault rates under specific headings are tabulated and calculated and converted to graphical form, as shown in Figure [9-5](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_9_Chapter.xhtml#Fig5), so you can examine the individual cost of running the services. The individual faults that are responsible for the highest operation cost are the ones to remedy first, either rectifying errors or creating new ones. One of the useful principles of cloud native services is that it is easy to create a new microservices if the operation cost is more than the new development cost.



***Figure 9-5***

Pareto chart analysis

Although the software services are not subject to wear-and-tear, the bathtub curve and Pareto chart provide us with insight into the operational lifecycle. The bathtub curve and Pareto chart enable software systems to understand the reliability of your services. This helps you to strategy your operationalization of services.

**Designing for High Availability**

High availability configuration is an approach for defining the services of your system, which ensures optimal operation performance, even at times of high loads. Although there are no fixed rules for implementing high availability (HA) systems, there are a few good practices that you need to follow to make your system highly available.

For any kind of system, there are two types of downtime: scheduled and unscheduled. Scheduled downtime is a result of maintenance like a software update or patch update. You can’t avoid this. Unscheduled downtime is caused by some unforeseen event, like hardware, software, or network failure.

The main objective of implementing an HA architecture is to make sure your system can handle a variety of loads and provide a great customer experience with minimal or no downtime.

The availability of a system is measured using the following formula:

**Availability = uptime/ (uptime + downtime)**

The result of this formula simply refers to a system that is continuously operational for a desirable long period of time. The result is expressed as 99.99 percent (“four nines”) or 99.999 percent (“five nines”), etc.

An HA solution is not just adding servers or containers to the existing stack, but actually, it is the opposite as more servers add a higher probability of failure. The cloud native modern architecture allows for the distribution of workloads across multiple instances of services in a cluster, which helps in optimizing resource use.

Cloud platforms are essentially built to tolerate failures and provide features to help build reliable, highly scalable, and highly available systems. Such features include the following:

* Infrastructure as a service (IaaS) is available across geographic locations.
* Availability zones are engineered to be isolated from failures in other zones.
* Your services can be deployed across availability zones across geographic locations to provide HA.

However, you cannot leverage the previous benefits just by moving your application to the cloud. To achieve HA in a cloud, consider these best practices:

* Design your application to be cloud native.
* Design your application for availability and recoverability.

To achieve HA, you need to strategize all the layers of the application equally. Let’s examine each component of your application.

* **High Availability of Databases**
* You use can both SQL databases and NoSQL databases in your architecture, but they will run on a separate server. You need to configure databases for redundancy. This can be achieved with a master and slave strategy. If the master fails, the voting technique will be carried out to choose a master and also can be made highly available by using a sharding strategy. More details of horizontal scalability are explained in Chapter [4](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_4_Chapter.xhtml). I have extensively covered database partitioning with various patterns such as horizontal partitioning or sharding, list partition, round-robin partition, vertical partitioning, leader-based replication, and quorum-based replication.
* **High Availability of Services**
* Applications can scale automatically by using containers and Kubernetes based on the load across multiple availability zones and geography. You can find more details in Chapter [16](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_16_Chapter.xhtml) Cloud Native Infrastructure.

To design an HA cloud native solution, you must remove the single point of failure at each layer, all the way from the infrastructure to software applications. The HA is usually accomplished with the redundant deployment of your systems. The deployment strategy can be chosen as either active-active or active-passive. Each strategy comes with cost and effort.

**Active-Active Deployments**

In an active-active deployment, application instances are actively running simultaneously.

* The load balancer distributes incoming requests across application instances.
* Instances are always running and ready to receive a request.
* You can achieve a recovery time objective (CTO) close to zero.
* Write messages simultaneously for both the data center for event-driven architecture by using messages queues or Kafka streaming.

**Active-Passive Deployments**

In active-passive deployments, one instance will be active, and the other will be in standby mode.

* The request sends only to primary instances.
* If the primary fails, then it routes to secondary instances.

Follow these best practices to make a system highly available:

* *Data backups, recovery, and replication*: Plan your databases to take regular snapshots and create read replicas to server requests to help recover if the primary fails.
* *Clustering*: You cannot avoid failure. HA is all about serving consumers regardless of failure. An HA cluster includes multiple nodes and shares information across nodes depending on the type of your architecture, like event-driven or synchronous calls. In an event-driven use, message queues or Kafka clusters share information across nodes as mentioned for active-active deployments.
* *Load balancer*: Use a load balancer to route traffic to the available instances. You can configure the load balancer to route request with the percentege or near location.
* *Geographic redundancy*: You can use a cloud location to deploy your application across geographies to make your application highly available.
* *Self-healing*: Apply a self-healing mechanism to heal your services automatically without human intervention; you can find more details about self-healing in Chapter [6](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_6_Chapter.xhtml).
* *Design your systems with stateless as much as possible*: Service states are not stored in one instance; the loss of an instance will impact availability and performance. Always store the state outside of the container.
* *Design your application to handle disruption gracefully, without customer impact*: Deploy the application in multiple AZs and automate the load balancer at every layer.
* *Observability*: Implement integrated observability across all services to monitor the health of services.
* *Prediction machine learning model*: Use ML models to predict your service health and the load; therefore, you can manage your system effectively.

**Designing for the Customer**

In the age of the digital economy, we are undergoing a lot of business and technology disruption. The customer can become impatient and want something new. Therefore, as an architect, you need to always think about the customer and business while designing your system.

With cloud native, IT services that can quickly build and deliver solutions in response to customer needs will attract and retain your customers and build enduring success. The methodology that you adopt must enable your engineering team to iterate faster and release software rapidly so that you can respond more effectively to customer needs and events.

When you designing your system, you need to focus on the customers and their needs in each phase of the design process. The team must involve users throughout the design process through a variety of research and design techniques to create highly usable and easily accessible applications.

Adopt the following best practices for customer-centric design:

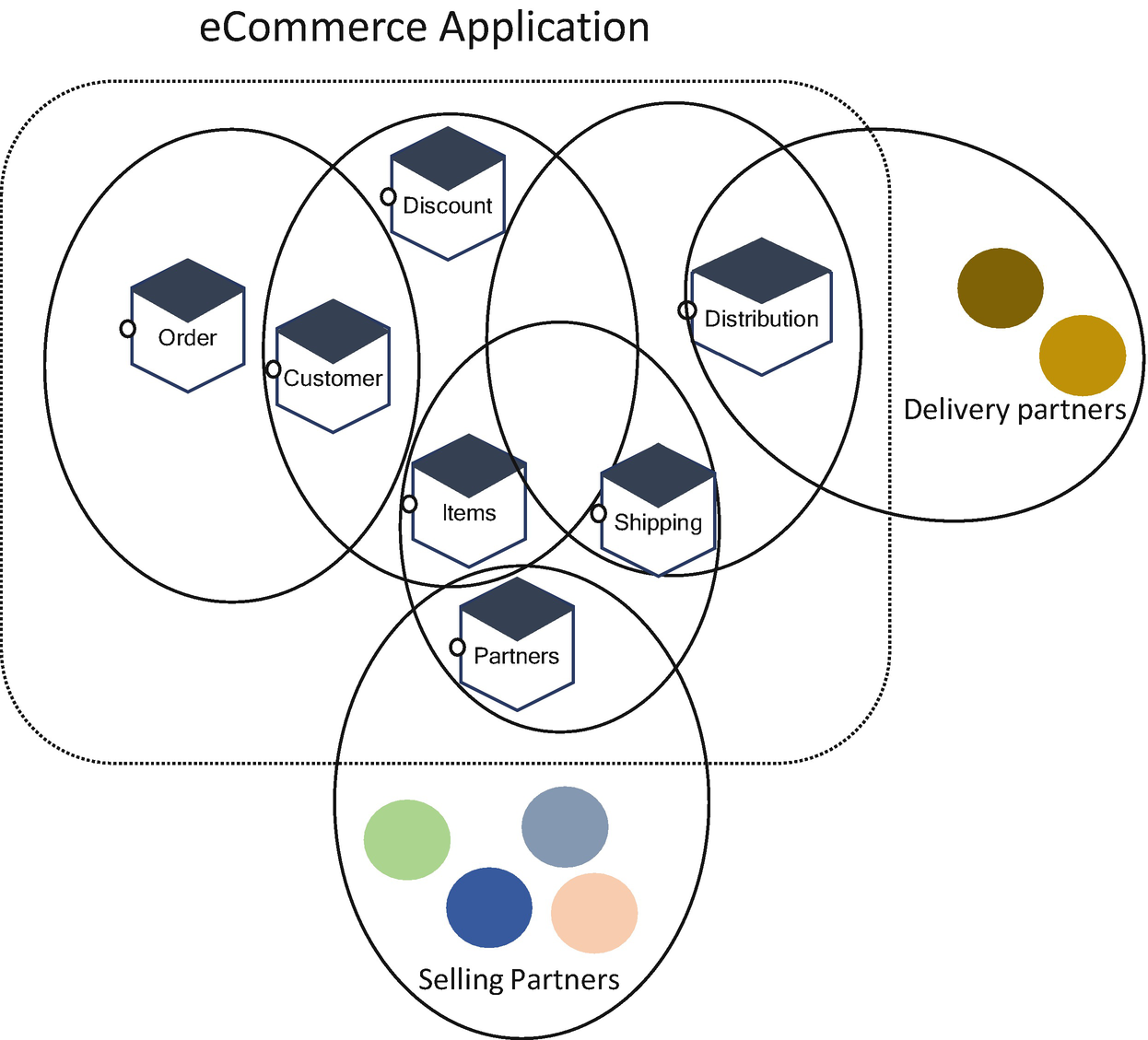
* Cloud native microservices allow you to deliver continuously and be agile. There is no final version of services. It’s an approach where you see action every day.
* You need to constantly update the code based on customer feedback, regardless of whether it’s voiced directly, comments on social media, etc.
* Rather than the software engineering team working in a silo, you need to enable the engineering team to engage proactively and regularly with the business team through design thinking workshops.
* Adopt hypothesis-driven development to get early feedback from customers and use their feedback to make valuable real-time improvements.
* Adopt A/B testing, which helps you to discover if a customer-suggested change represents the majority opinion and whether any services can be modified and deployed in production without a business impact.
* Adopt continuous integration and continuous delivery to turn around quickly on changes and also create infrastructure as code to automate infrastructure.
* Follow a decentralized governance approach, which allows agile Pod teams to work independently and embrace customer-centric innovation.
* Follow canary deployment. Test a new version of the software before the version is introduced in the main production environment. Make small releases available to a small group of people to get early feedback.
* Collect customer data, analyze it instantly, and apply changes to your software.

**Designing for Interoperability**

Interoperability can be understood in multiple ways, like data transfer from one system to another without transformation loss or the ability of different applications to interact with each other dynamically, facilitating the smooth exchange of information. In this section, I am use the second interoperability definition.

Each system in your enterprise is different, and systems do not interact with each other out of the box. You need to create an integration mechanism between two systems to work. In cloud native architecture, each domain is designed as a microservice, and you need to implement interoperability across multiple microservices to complete a unit of work. For example, completing payment processing with a debit or credit card requires multiple services to perform. The degree of interoperability of a system or service can be measured as its cost of integration. The cost of integration of your services should be considered over its lifetime, not just at its point of first use. Changing to a service interface implies a need to re-integrate it with other services. The lifetime cost of a service whose interface changes will be considerably more than its initial cost. A service should have well-defined interfaces that do not change over time and are backward compatible.

For an ecommerce application, as shown in Figure [9-6](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_9_Chapter.xhtml#Fig6), you are required to develop various services and also required to integrate with various third-party applications to fulfill the orders. In Figure [9-6](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_9_Chapter.xhtml#Fig6), ecommerce services like order, customer, discount, item, partner, shipping, and distributions are part of your application. The delivery partners and selling partners are extended enterprises where you need to partner with them to complete the order lifecycle.



***Figure 9-6***

Interoperability of services in an enterprise

For your services to be interoperable, they must be able to exchange data and subsequently present the data in a way that is understood by other services.

In the exchange of messages among services, communication can be weaved through many services, across many security domains. Your service must be agile in nature, and your system must be able to modify interoperable conversation across involved services. The robustness of the conversation depends on well-formed message exchanges. For a well-formed message, the services must know each other for a well-defined contract. As messages are exchanged among services, the services must cross boundaries of local knowledge.

The interface of your service must be clearly described, and the description should be human-readable and also machine-readable. The human-readable description is essential for the developer to understand and integrate. Machine-readable descriptions enable dynamic discovery and composition of your software components.

**Designing for Events**

Anything that happens in enterprises or systems is an event. Examples include customer requests, batch updates, data changes, employees swiping a card, the customer swiping a credit or debit card, the customer buying a product in a retail ecommerce application, or someone checking in for a flight. Events exist everywhere and are constantly happening, no matter what the application is and what industry it is. Events are pervasive across any business. There is value in knowing about an event and being able to react to it quickly. The more quickly you can get information about events, the more effectively your business can react to them. The event is separate from the message because the event is an occurrence, and the message is the carrier of the information that relays the occurrence. In an event-driven architecture, an event likely commands one or more actions or processes in response to its occurrence. The following are the decisions you need to be aware of while designing an event-driven architecture:

* Prefer domain events to technical events, consider only technical events within a domain, and use domain events within and across domains.
* You should consider event-driven architecture for everything by replacing HTTP calls.
* Replicate data with ownership. Create governance for who is responsible for what; have only one data owner per domain entity (explained in data mesh) so that only the responsible person may write and changes to an entity and must be requested by the data owner.
* Do use distributed tracing, because distributed systems are difficult to observe and they use conversation IDs to track business-relevant interactions over time and in multiple bounded contexts.
* Use the Cloud Events specification for the interoperability of events across enterprises.
* Use event sourcing and write events to a journal table in the same transaction; instead of publishing them, use sidecar track events from that journal and publish them.
* Design with security and privacy concerns in mind from the beginning.

**Designing for Observability**

Observability is the extent to which you can understand the internal state of services based on the behavior. Making the system observable involves the practice of combining context, information, and specific knowledge about the system to create the conditions for understanding. In cloud native architecture, all services are distributed across various containers, which increases the need for observability because such architecture can fail due to interaction between multiple services.

The term *observability* originates in the mathematics of control theory, in which observability is a measure of how well the internal states of a system can be incidental knowledge of external outputs.

You might use monitoring tools to track the performance of infrastructure, networks, and services that support business use cases. As the organization is enabling cloud native architecture, the monitoring tools have shown limitations in their ability to adapt to the volatility of these architectures. Your existing static dashboards and manual thresholds do not scale, are not able to provide behaviors of systems, and are inflexible in assisting the resolution of unforeseen events. Using these tools, the business is unable to determine the state of its services with a high degree of certainty and to understand how its services impact key business indicators (KPIs).

Observability has evolved to solve the problems of present-day modern architecture. There is a need for observing techniques throughout the software development lifecycle. It encourages a shift-left approach starting from the developer laptop. It is an evolution of established monitoring, emphasizing visibility of the behavior of your distributed services in the traditional monitoring that focuses on individual services. To fully realize the promise of modern development methodologies, the application must be built with observability-driven development.

The monitoring relies on building dashboards and alerting to escalate known problem scenarios when they occur. The monitoring dashboards may not be able to provide the exact behavior of your services, such as when unknown problems happen frequently especially during request spikes. In these circumstances, the monitoring dashboards cannot get the entire picture of your services. Observability enables quick interrogation of services to identify the underlying cause of performance degradation.

Observability enables your system to reduce the time it takes to identify the root causes of performance issues. You can find more details of observability in subsequent chapters. The benefits of observability include the following:

* It improves the time to identify the issues, which helps to improve the application uptime and performance.
* The shift-left approach helps the developer to code for observability by implementing right configuration in a code.

Observability emphasis on collection and prediction of monitoring and, logging data.

**Designing for Portability**

Portability is the capability of running an application on the various platforms without any changes. Nowadays, many organizations are prioritizing a multicloud strategy and require you to move the application from one cloud provider to another cloud provider automatically without changing the application. But to achieve this, you need to design your application to be portable. Adopt a multicloud strategy for several reasons, including vendor lock-in, optimal utilization, reduced cost, SLA issues, etc.

Portability can be categorized into three ways.

* *Functional portability*: This is realized by describing the application’s functionality details in a vendor-agnostic manner.
* *Data portability*: This is realized when the customer is able to access and save application data from the provider and to input this into a corresponding application hosted by another provider.
* *API enhancements*: API enhancement metadata is added through annotations; metadata provides information about other data.

In application portability, there are four areas of concerns that you need to take into consideration while designing an application.

* *Programming language and framework*: To build an application, the programming language plays an important role, and all cloud platforms have certain languages and frameworks that they support. For example, Google App Engine supports Java, Python, PHP, GO, and Node.js.
* *Platform-specific services*: Cloud platforms provide services through specific APIs, etc.
* *Data storage services*: There are two types of storage, SQL and NoSQL. The data that you designed for SQL will not work for NoSQL, and the data you designed for AWS Dynamo DB will not work directly Azure Cosmos DB, etc.
* *Platform-specific configuration files*: Platform-specific configuration files also exist on the cloud. For example, Google AppEngine, for instance, uses the “app engine web XML file.” Adapting the configuration file to each target cloud platform affects application portability.

Along with these considerations, follow these techniques for your application design to adopt portability:

* Choose the right programming language.
* Containerize your services.
* Use a unified cloud API.

More important, create infrastructure as code for your application; this helps to deploy your application to any cloud vendors.

**Designing for Ethics**

You can find technology everywhere. Our society is more reliant on technology than ever before. Currently, without technology, nothing is moving. Therefore, it is everyone’s responsibility to consider ethics when making decisions. Some in the society are misusing the advancement of technology and so we have fake news, cyberattacks, and technology wars against each other. The average person spends around 145 minutes per day on social media, and the average person usage of mobile is around 7 hours per day. Those fixated eyes never leave screens, which creates stress and anxiety.

As software engineers, it is natural that we spend most of the time focusing on how best we can serve users and how we can better compare to peers, which is perfectly fine. In some ways, we need to consider how we can use technology to create a better world. For example, Facebook never realized that it would grow to become a home of algorithmic propaganda and filter bubbles, YouTube didn’t expect to become a conspiracy theorist, and Twitter hadn’t anticipated the hate speech or trolling.

Let’s look into another example that may occur if a company like Facebook purchases a major bank and becomes a social credit provider. What happens if artificial intelligence becomes a mainstream tool, spawning across terrorism, theft, and more?

While designing your system, you need to anticipate the long-term social impact and unexpected uses of the tech you create today. Your job is not only to create a fancy architecture but also remain ethical. So, ask yourself these questions before you make any design decisions:

* If the technology you’re building will someday be used in unexpected ways, how can you prepare for this?
* What are the new categories of risk that you should give special attention?
* How can you react if any unforeseen risks occur?

There are various toolkits available for you to think through some of the future implications of the software you are building. They are the Ethical OS and Tarot Cards of Tech. Let’s explore briefly what they offer.

The Ethical OS comes from a partnership between the Institute of the Future and Tech and Society Solutions Lab. It addresses social impact harms ranging from disinformation to a dopamine economy.

The Ethical OS toolkit helps manage the design process and manage risk around the existing technologies you are using and helps to identify dark spots of your architecture and design. It has risk zones and provides a checklist to identify the emerging areas of risk of your design. The following are the eight risk zones:

* *Risk Zone 1: Truth, Disinformation, Propaganda*: These risks have bad actors using the data and creating propaganda against individuals or companies, using the fake data to undermine the credibility of a company, etc.
* *Risk Zone 2: Addiction and the Dopamine Economy*: This risk zone is about addiction. As I mentioned, people are spending more hours on mobile and social media. This is not good for the mental or physical or social health of people. In recent times, we have seen many young have once lost their lives due to the PUBG game.
* *Risk Zone 3: Economic & Asset Inequalities*: This risk zone talks about inequalities in society. The people who don’t have access can encounter setbacks compared to those who have access to it.
* *Risk Zone 4: Machine Ethics & Algorithmic Biases*: This risk zone talks about how you use machine learning models and create a bias against individuals or the marketing of a product.
* *Risk Zone 5: Surveillance State*: This risk zone provides information about the use of technology by government bodies and military zones, for example, in the recent case of Philadelphia police action.
* *Risk Zone 6: Data Control & Monetization*: This risk zone is about data privacy, data share, and data monetization.
* *Risk Zone 7: Implicit Trust & User Understanding*: This risk is provided by collecting the data or use of technology without acceptance from the user.
* *Risk Zone 8: Hateful & Criminal Actors*: This risk zone helps to identify bullying, harassing, or stalking about people and financial fraud and illegal activities.

Each zone provides a checklist to evaluate your technology choice, tools, and features you’re working on and choose the risk zone that is relevant to you. You can start investigating these checklists and mitigating these risks. After you understand the risk zones, carry out the following activities to be ethical in your software design:

* Use the relevant questions in each risk zone and design your system to mitigate these risks.
* Use this selected checklist as part of your agile backlog.
* Socialize these questions across your project team and client.
* Collect relevant resources and brainstorm with the right subject matter experts.
* Fine-tune your design by implementing ethics in your system.

**Designing for Accessibility**

Accessibility is a design concept that means your application will include accommodations to the user interface or for slow networking so it can reach all people without any discrimination. Accessibility is all about supporting that flexibility for different user needs. The following are a few incidents:

* 3,500 web and app accessibility lawsuits were filed in 2020.
* 1 in 5 adults in the world live with a disability.
* Around 70 percent of web users with a disability will simply leave a website that is not accessible.
* 100 percent of humans in the world will face temporary and situational impairments at some point in our lives either in touch, sight, hearing, or speech.

In the digital world, you may use a variety of technologies and strategies in several ways to access and use digital content depending on your needs. Human-centered design focuses on the specific needs of individuals, including people with disabilities and elders, who most need accessible content. Designing and creating accessible content benefits all of us, while it is essential for some of us. Accessibility encompasses all disabilities that affect access and engagement to digital content including physical, speech, visuals, auditory, cognitive, learning, and neurological disabilities.

People across the organization use different technologies and strategies to access and navigate content based on their needs and preferences. You can adopt two approaches.

* *Assistive technologies*: This includes any technologies that aid in the usability, perception, comprehension, and navigation of digital content, such as screen readers that read content aloud, screen magnifiers, voice recognition software, and selective switches.
* *Adaptive strategies*: These are techniques that people use to improve interaction with digital content, such as increasing text size, reducing mouse speed, and turning on captions.

People who have multiple disabilities need a combination of assistive technologies and adaptive strategies to interact with content.

**Accessibility Guidelines and Standards**

The Web Accessibility Initiative (WAI) of the World Wide Web Consortium (W3C) develops guidelines that are widely regarded as the international standard for web accessibility. The Web Content Accessibility Guidelines (WCAG) is the set of technical standards and recommendations developed by WAI that defines requirements on how to make digital content such as text, images, multimedia, structure, and presentation accessible. The WCAG is organized around the POUR principle (Perceivable, Operable, Understandable, Robust). There are two WCAG standards; they are WCAG 2.0 and WCAG 2.1.

These standards are categorized into three levels of conformity.

* *Level A*: This is the basic level; you must consider all the guidelines included at this level as “MUST SUPPORT” requirements.
* *Level AA*: It is a midrange level that satisfies all Level A criteria and more; guidelines included in this level of conformance are considered “SHOULD SUPPORT” requirements.
* *Level AAA*: This is the most comprehensive level of conformance and also the most restrictive. You consider guidelines at this level as “MAY SUPPORT” requirements.

You need to consider the following areas for defining and designing the template of a document with an accessibility checklist:

* *Readability*: Good readability should be guaranteed to all users, in particular with disabilities. Ensuring clear and flawless readability is key for rendering a material accessible. Some of the readability elements are screen magnification, actionable elements, and movement and animation.
* *Use of color*: Contrast is the difference between text and the background immediately behind it. High-contrast text benefits users with low vision, color blindness, or other visual disabilities. Ensure that good color contrast with at least 4:5:1 is established between text and images and the background color, and ensure a contrast ratio of 3:0:1 between text and the neighboring text when color is used to denote status.
* *Text formatting*: The readability of text can be affected by how the text has been formatted. Ensure that you choose a typeface that emphasizes clarity and readability, and use a font size between 12 and 14 points and use a 1.5 line space.
* *Navigation and orientation*: Well-organized content helps users to orientate themselves and to navigate effectively.

There are many more checklists available such as headings, interaction and feedback, repeated elements, metadata, images, and links.

There are different types of tools that need to be used by your quality assurance team to test digital content for accessibility such as evaluation tools, assistive technologies, and authoring tools.

**Designing for Automation**

Automation is always required for software systems, but the cloud makes it easier for you to automate, test, and build infrastructure. These are some common areas for automation in cloud native applications:

* Continuous integration and continuous delivery (CI/CD)
* Infrastructure
* Observability and automated recovery

The lifecycle of CI/CD is continuous definition, continuous integration, continuous deployment and release, and continuous operation of your packages into the cloud environments by adopting the following principles:

* Have cohesive teams with shared objectives.
* Test early and often test right.
* Implement zero-touch deploy and configuration.
* Automate everything.
* Embrace failure, recover automatically, and degrade gracefully.

In the infrastructure, you need to automate the creation of infrastructure using infrastructure as code. It is a process that allows you to treat your infrastructure provisioning and configuration in the same manner that you handle application code. The infrastructure configuration is stored in the source code repository and uses a CI/CD pipeline to automate infrastructure.

In the case of observability and automated recovery, you should add user stories from inception. Logging and monitoring observe the behavior of the system to give a measure of the overall health of your system and automate your application by applying self-healing, resizing the disk, etc. You can read more details about DevSecOps in Chapter [14](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_14_Chapter.xhtml), Enterprise cloud native operation, which explains the end-to-end DevSecOps pipeline.

**Designing for Maintainability**

Maintainability focuses on the ease with which a software system can be maintained. Maintenance of a system takes place as changes are made to it after it is in production. Changes are constant in the present-day world; it is inevitable that the system experience will change. It is important to build a maintainable system.

In cloud native architecture, your systems are built with microservice principles. The core principle is to make it easy to maintain and enhance. To achieve this principle, your microservices must be well designed with the use of domain-driven design.

**Designing for Usability**

Usability describes how easy it is for users to perform the required tasks using the system. User satisfaction is directly correlated to its level of usability. Users are more satisfied with a system that is easy to use and that provides a good user experience. Adopt a hypothesis-driven development approach to design for usability.

**Summary**

The main principle of architecting for cloud native architecture focuses on how to optimize system architecture for the “-ilities.” In traditional architecture, we focus on the “-ilities” of a relatively small number of components, but in the cloud, that fixed infrastructure makes much less sense because it is easily available and on-demand usage. Therefore, cloud native architecture focuses on achieving the “-ilities.”

You must pay attention to designing for the “-ilities” as they influence the architecture and design of your system. The system must meet the designated “-ilities” while identifying and specifying them in a way that they can be measured and tested.

The success of the system depends on how you are approaching each “-ility.” Look at the holistic design of your system while designing your application.

Now you understand more about designing the “-ilities” and the fact that they influence your software design.