

EXPERT INSIGHT

Solutions Architect's Handbook

Kick-start your career with architecture design principles, strategies, and generative AI techniques

Forewords by:

Rajesh Sheth, VP
EBS and AWS Backup

Rohan Karmarkar, MD
Solution Architecture, AWS

Kamal Arora, Director
Solution Architecture, AWS

Third Edition



Saurabh Srivastava
Neelanjali Srivastav

packt

mail2divakar@gmail.com

Solutions Architect's Handbook

Third Edition

Kick-start your career with architecture design principles, strategies, and generative AI techniques

Saurabh Srivastava

Neelanjali Srivastav



Solutions Architect's Handbook

Third Edition

Copyright © 2024 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Senior Publishing Product Manager: Rahul Nair

Acquisition Editor – Peer Reviews: Gaurav Gavas

Project Editor: Amisha Vathare

Content Development Editors: Rebecca Youé and Shikha Parashar

Copy Editor: Safis Editing

Technical Editor: Karan Sonawane

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Presentation Designer: Ganesh Bhadwalkar

Developer Relations Marketing Executive: Maran Fernandes

First published: March 2020

Second edition: January 2022

Third edition: March 2024

Production reference: 2240524

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN: 978-1-83508-423-6

www.packt.com

mail2divakar@gmail.com

To our beloved children, Sanvi and Shubh, whose boundless joy and happiness illuminate our lives.

— Saurabh and Neelanjali

Forewords

A solutions architect role requires a unique skill set across the breadth and depth of technology and the ability to tie this back to the business and derive a return on investment. As cloud adoption accelerates, enterprises are looking for a solutions architect skill set to help their digital journey, keeping the cloud at the core of their strategy. Cloud has a very different value proposition and comes with several tools and services, replacing costly third-party licensed software. You can achieve desired reliability and scalability in the cloud in minutes to capture high growth and seasonality. You can use cloud native services to build high-performance applications while keeping your costs low. As per my 25+ years, tenure in the IT industry building complex and highly scalable applications, I strongly believe that the right architectural choice enables customers to get the most from their cloud strategy.

Solutions Architect's Handbook fills the skill gap by providing architecture best practices through the lens of cloud architecture. The book starts by clarifying the roles and responsibilities of solutions architects, and then helps build a strong foundation by walking through the design principles of the architecture. Keeping cloud strategy at the core, the book covers a broad range of topics, from migration to the cloud, to designing cloud native solution architecture. The book does a good job at addressing 50+ core application development architecture design patterns with reference architectures to visualize them. Solutions architects need to look at all aspects of application design and dive deep into each of these topics to optimize architecture for security, reliability, performance, cost, and operational excellence.

I particularly appreciate the third edition of this book for its in-depth exploration of cloud-native architectural patterns and its extensive coverage of generative AI (GenAI). The discussion on GenAI architecture and foundational models is thorough, offering insights into a range of GenAI tools and their potential applications. Additionally, the book delves into various storage technologies, which are crucial for developing a robust infrastructure for advanced analytics and machine learning. Moreover, it offers valuable information on machine learning architecture, CloudOps, and MLOps, providing a comprehensive guide on transitioning GenAI and cloud applications from prototype to production. This edition not only broadens your knowledge base but also equips you with the practical know-how to implement these advanced technologies effectively.

I often see people looking to upgrade their skills or change their career paths to become solutions architects. Saurabh has put his years of experience into this book, making it very easy for anyone looking to upskill in their current role or explore future technologies. In the cloud era, for a new or existing solutions architect who is looking to keep their skills sharp, this book hits all the key areas.

Rajesh Sheth, VP
AWS Elastic Block Store and AWS Backup

The third edition of *Solutions Architect's Handbook* arrives at a crucial juncture, as the COVID-19 pandemic accelerated digital transformation and cloud technology adoption. This edition adeptly addresses the urgent need for solutions architects to embrace cloud-native architectures and microservices, providing essential guidance on scaling, operational resiliency, disaster recovery, and business continuity. It offers a forward-looking perspective on distributed application development and the rapid embrace of cloud-native solutions, reflecting the shifting paradigms in technology architecture.

The book's comprehensive coverage, from cloud migration and modernization to the cutting edge of machine learning and GenAI, makes it an indispensable resource. It skillfully bridges legacy modernization, including mainframe considerations, with the latest in GenAI foundation models and tools, offering a deep dive into the trends shaping the future of architectural design.

Solutions Architect's Handbook stands out for its holistic approach, covering a gamut of crucial aspects from functional architectures and integration to extensibility, reusability, usability, accessibility, cost management, and security. Its breadth of topics and practical insights make it a must-read, serving as an invaluable reference for both aspiring and seasoned architects navigating the complexities of today's cloud-centric environment. Authored by Saurabh and Neelanjali, drawing from their rich experience, this handbook is a beacon for those striving to excel in the evolving landscape of solution architecture.

*Rohan Karmarkar, MD
Solution Architecture, AWS*

The technology realm has always been fast-moving and in order to keep growing in their careers, IT professionals need to incrementally acquire new skills over time. However, in the last decade this trend has become dominant, with cloud computing becoming the ‘new normal.’ Now almost every day there are new announcements, features, and service updates by the cloud providers, which has necessitated the focus on a continuous learning culture for everyone. Along with this, now the typical boundaries between the usual roles of developer, database administrator, security professional, build/release engineer, and so on have started to blur, resulting in new roles being created to focus on big-picture and end-to-end ownership. One such role is that of a ‘Solutions Architect,’ which started to evolve from existing roles in industry like ‘Application Architect’ and ‘IT Architect’ and has now become mainstream. There are also variations of this role; however, the most common avatar is that of ‘Cloud Solutions Architect,’ which is a pretty dynamic role in itself.

Often, IT professionals want to switch roles; however, they lack direction on how to be successful on that path. This book focusses on this very aspect DASH; an effective transition from an existing IT role to that of a solutions architect. It explains in a very logical manner the steps to embark on that journey. It starts off with a simple, very relatable explanation of what this role entails and how it differs from a few of the other similar type of profiles. Then, it goes into the technical skills and knowledge needed to be a successful solutions architect. This begins with basic design pillars and architectural principles (like high availability, reliability, performance, and security and cost optimizations), followed by a dive deep into each one of those. The book also covers some key concepts around cloud native architectures, DevOps, and the data engineering and machine learning domains, which are the cornerstone of any modern-day architecture. In the latest revision of the book, Saurabh and Neelanjali have also included very insightful details on cloud native architecture, generative AI, deep learning, CloudOps, advanced analytics, and cloud migration. All these areas are slowly becoming pivotal to the enterprise IT landscape, and it’s essential for solutions architects to be aware of them to stay ahead.

I have personally been through this journey of being a solutions architect as a development team-lead, and so has Saurabh. We always wished there was a handbook available that could help us then. So, to fill that major gap in the industry, Saurabh has created this very detailed book, which is based on personal experiences and learnings that make it a very relatable read for anyone. I highly recommend you read this book and keep it as a handy reference always, as in it you will find very important nuggets of knowledge that will help you be a successful solutions architect, opening up a new world of infinite possibilities!

*Kamal Arora, Director
Solution Architecture, AWS
<https://www.amazon.com/Kamal-Arora/e/B07HLTSNRJ/>*

Contributors

About the authors

Saurabh Srivastava is a technology leader, author, inventor, and public speaker with over 20 years of experience in the IT industry. He currently works at **Amazon Web Services (AWS)** as a Global Solutions Architect Leader and enables AWS partners and customers on their journey to the cloud. Saurabh led the AWS global technical partnerships, set his team's vision and execution model, and nurtured multiple new strategic initiatives.

In addition to his work at AWS, Saurabh is the author of Packt's bestselling book, *AWS for Solutions Architects, Second edition*, and has authored various blogs and white papers on a diverse set of technologies such as big data, IoT, machine learning, and cloud computing. He is passionate about the latest innovations and their impact on our society and daily life. Saurabh holds patents in cloud platform automation and has worked as an enterprise solutions architect, software architect, and software engineering manager in Fortune 50 enterprises, start-ups, and global product and consulting organizations. With his vast experience and expertise, Saurabh is a valuable resource for anyone looking to learn about cloud computing and its various applications.

Neelanjali Srivastav's extensive experience in the software industry as a technology leader, product manager, and agile coach brings a wealth of knowledge to the field. Currently, she is working as Technical Portfolio Leader for Aya Healthcare. Before that she worked as a Senior Product Manager at AWS, where she enabled AWS customers and partners in AWS database, analytics, and machine learning services.

Neelanjali is also the author of Packt's best-selling book, *AWS for Solutions Architects*, which is a valuable resource for those looking to kick-start their careers as AWS solutions architects. With her experience leading teams of software engineers, solutions architects, and systems analysts to modernize IT systems and develop innovative software solutions for large enterprises, Neelanjali is well-equipped to provide insights into challenges and opportunities in the field of technology.

Neelanjali's expertise in enterprise application management, agile coaching, cloud service management, and enterprise merger and acquisition makes her a sought-after speaker and thought leader in the industry. She is dedicated to helping others learn and grow in their careers, and her contributions to the field are sure to make a lasting impact.

About the reviewers

Wayne R. Vincent has 40 years of experience in information technology with over 3 decades in pre-sales roles as an Enterprise Architect. He holds a Bachelor's and master's degree in computer science from the University of New Haven, where he is also an advisor for the university's Executive Micro MBA program.

Wayne has worked for start-ups in high-end book publishing and data virtualization as well as large corporations, including Sun Microsystems, Oracle, AWS, and Google. While at these companies Wayne has worked with customers across many industry verticals, including Manhattan Financial Services, publishing, manufacturing, retail, and insurance.

Nigel Harris is an accomplished technology leader with more than 25 years of experience helping enterprises meet business objectives by designing, deploying, and operating secure IT infrastructure. Nigel has worked across a variety of technology domains including data and analytics, security, DevOps automation, and microservices. He works with AWS customers to provide guidance and technical assistance on AWS architectures.

Nigel is currently a senior leader at AWS where he leads a solution architecture organization helping enterprises in North America. Previously, Nigel worked as a principal solutions architect at Amazon where he directly assisted executive and technical stakeholders in planning, migrating, and operating cloud infrastructure. Nigel has passed 10 AWS certifications and is a contributing author to the AWS Professional and Security Specialty certifications. Prior to joining Amazon, Nigel worked at a Fortune 500 hi-tech global manufacturing organization in a variety of technical and management roles.

Viraj Padte has over 8 years of experience in software engineering and architecture. He is a senior solutions architect at AWS, where he leads strategic initiatives to architect and implement cloud solutions for enterprise clients, leveraging the full suite of AWS services. He collaborates closely with cross-functional teams, including sales, engineering, and product management, to deliver comprehensive solutions that exceed customer expectations. He also provides technical guidance and thought leadership to clients, helping them navigate complex cloud challenges and achieve their business objectives.

Viraj's core competencies include cloud-native architecture, microservices, serverless, Docker, Kubernetes, continuous integration (CI)/continuous deployment (CD), and infrastructure-as-code (IaC). He is also AWS certified as an architect and a developer, and he has spoken at re:Invent and edited publications on cloud topics. He is passionate about building autoscaling software systems, architecting cloud-native applications, and designing CI/CD pipelines for complex deployment environments.

mail2divakar@gmail.com

Join our book's Discord space

Join the book's Discord workspace to ask questions and interact with the authors and other solution architecture professionals: <https://packt.link/SAHandbook>

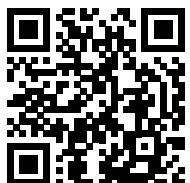


Table of Contents

Preface

xxvii

Chapter 1: Solutions Architects in Organizations	1
What is solution architecture?	2
The benefits of solution architecture • 2	
The solutions architect's role	4
Generalist solutions architect roles • 7	
<i>Enterprise solutions architect</i> • 7	
<i>Application Architect</i> • 7	
<i>Cloud Architect</i> • 8	
<i>Architect Evangelist</i> • 9	
Specialist solutions architect roles • 9	
<i>Infrastructure Architect</i> • 10	
<i>Network Architect</i> • 11	
<i>Data Architect</i> • 11	
<i>ML Architect</i> • 12	
<i>GenAI architect</i> • 13	
<i>Security Architect</i> • 14	
<i>DevOps architect</i> • 14	
<i>Industry Architect</i> • 15	
Understanding a solutions architect's responsibilities	16
Analyze functional requirements (FRs) • 18	
Define NFRs • 18	
Understand and engage stakeholders • 21	
Understand architecture constraints • 21	
Make technology selections • 23	
Develop a POC and prototype • 24	

Solution design and delivery • 24	
<i>Ensuring post-launch operability and maintenance</i> • 25	
Solution scaling and technology evangelism • 25	
Solutions architect in an Agile organization	26
Common challenges in the solutions architect role	26
Career path and skill development for solutions architects	28
Summary	30
Chapter 2: Principles of Solution Architecture Design	31
Building scalable architecture design	32
Scaling static content • 34	
<i>Session management for application server scaling</i> • 35	
<i>Database scaling</i> • 36	
<i>Building elastic architecture</i> • 36	
Building a highly available and resilient architecture	38
Highly available architecture • 38	
Resilient architecture • 38	
Achieving redundancy • 39	
Addressing component failure • 40	
Making your architecture fault-tolerant	41
Designing for performance	42
Creating immutable architecture	43
Think loose coupling	44
Think service, not server	46
Think data-driven design	48
Adding security everywhere	48
Making applications usable and accessible	50
Achieving usability • 50	
Achieving accessibility • 50	
Building future-proof extendable and reusable architecture	51
Ensuring architectural interoperability and portability	53
<i>Making applications interoperable</i> • 53	
<i>Making applications portable</i> • 53	
Applying automation everywhere	54
Plan for business continuity	55
Design for operation	57

Overcoming architectural constraints	57
Taking the MVP approach • 58	
Summary	59
Chapter 3: Cloud Migration and Cloud Architecture Design	61
Public, private, and hybrid clouds	62
Solution architecture in the public cloud	63
The public cloud architecture • 63	
Popular public cloud providers • 65	
Cloud-native architecture • 65	
Designing cloud-native architecture • 66	
Creating a cloud migration strategy	69
Lift and shift migration • 71	
<i>Rehost</i> • 71	
<i>Replatform</i> • 71	
<i>Relocate</i> • 72	
The cloud-native approach • 72	
<i>Refactor</i> • 73	
<i>Repurchase</i> • 74	
Retain or retire • 74	
<i>Retain</i> • 74	
<i>Retire</i> • 75	
Choosing a cloud migration strategy	75
Steps for cloud migration	77
Discovering your portfolio and workloads • 79	
Analyzing the information • 81	
Creating a migration plan • 82	
Designing the application • 83	
Executing application migration to the cloud • 87	
<i>Data migration</i> • 87	
<i>Server migration</i> • 89	
Integrating, validating, and cutover • 89	
<i>Validation</i> • 90	
<i>Integration</i> • 90	
<i>The cutover process</i> • 90	
Operating the cloud application • 92	
Application optimization in the cloud • 93	

Creating a hybrid cloud architecture	94
Taking a multi-cloud approach	97
Implementing CloudOps	97
CloudOps pillars	99
Summary	100
Further reading	101

Chapter 4: Solution Architecture Design Patterns	103
---	------------

Building an n-tier layered architecture	104
The web layer • 105	
The application layer • 105	
The database layer • 106	
Creating a multi-tenant SaaS-based architecture	106
Understanding service-oriented architecture	108
RESTful web service architecture • 109	
Building a RESTful-architecture-based e-commerce website • 111	
Building a cache-based architecture	112
Cache distribution pattern in a three-tier web architecture • 114	
Rename distribution pattern • 116	
Cache proxy pattern • 117	
Rewrite proxy pattern • 118	
App caching pattern • 119	
Memcached versus Redis • 120	
Model-View-Controller (MVC) architecture	121
Applying MVC to design an online bookstore • 122	
Building Domain-Driven Design (DDD)	123
Understanding the circuit breaker pattern	125
Implementing the bulkhead pattern	125
Creating a floating IP pattern	126
Deploying an application with a container	128
The benefit of containers • 129	
Container deployment • 130	
Building container-based architecture • 131	
Database handling in application architecture	133
High-availability database pattern • 135	
Clean Architecture	136
Avoiding anti-patterns in solution architecture	137

Summary	139
<hr/> Chapter 5: Cloud-Native Architecture Design Patterns 141	
What is cloud-native architecture?	142
Building serverless architecture	144
Considerations for serverless architecture • 145	
Building stateless and stateful architectural designs	147
Stateful architecture • 148	
Stateless architecture • 148	
Creating a microservice architecture	150
Saga pattern • 153	
Fan-out/fan-in pattern • 155	
Service mesh pattern • 157	
Reactive architecture	161
Building queue-based architecture	164
Queuing chain pattern • 164	
Job observer pattern • 166	
Pipes-and-Filters Architecture	167
Creating Event-Driven Architecture	168
Publisher/subscriber model • 169	
Event stream model • 170	
Backend for Frontend pattern	171
Cloud-native architecture anti-patterns	172
Single point of failure • 172	
Manual scaling • 172	
Tightly coupled services • 173	
Ignoring security best practices • 173	
Not monitoring or logging • 173	
Ignoring network latency • 173	
Lack of testing • 173	
Over-optimization • 173	
Not considering costs • 174	
Summary	174
<hr/> Chapter 6: Performance Considerations 177	
Design principles for high-performance architecture	177
Reducing latency • 178	

Improving throughput • 179	
Handling concurrency • 181	
Applying caching • 182	
Technology selection for performance optimization	183
Making a computational choice • 184	
<i>Working with containers</i> • 186	
<i>Going serverless</i> • 189	
Making a storage choice • 190	
<i>Working with block storage and storage area network</i> • 191	
<i>Working with file storage and network area storage</i> • 191	
<i>Working with object storage and cloud data storage</i> • 191	
<i>Storage for databases</i> • 193	
Making a database choice • 194	
<i>Online transactional processing</i> • 194	
<i>Nonrelational databases</i> • 195	
<i>Online analytical processing</i> • 195	
<i>Building a data search functionality</i> • 196	
Improving network performance • 197	
<i>Using edge computing</i> • 197	
<i>Defining a DNS routing strategy</i> • 198	
<i>Applying a load balancer</i> • 199	
<i>Applying auto-scaling</i> • 199	
Performance considerations for mobile applications	201
Optimization of load times • 201	
Efficient use of resources • 201	
Responsive user interface (UI) • 201	
Network efficiency • 202	
Battery consumption • 202	
Cross-platform compatibility • 202	
User experience (UX) design • 202	
Effective data management • 202	
Testing and quality assurance • 202	
Performance testing	203
Types of performance testing • 203	
Managing performance monitoring	204
Summary	205

Chapter 7: Security Considerations	207
Design principles for architectural security	207
Implementing authentication and authorization control • 208	
Applying security everywhere • 208	
Reducing the blast radius • 209	
Monitoring and auditing everything all the time • 209	
Automating everything • 210	
Protecting data • 210	
Responding to security incidents • 211	
Selecting technology for architectural security	211
User identity and access management • 211	
<i>Federated identity management and single sign-on</i> • 213	
<i>Kerberos</i> • 214	
<i>Microsoft Active Directory</i> • 215	
<i>Security Assertion Markup Language</i> • 216	
<i>OAuth</i> • 218	
<i>JWT</i> • 220	
Handling web security • 221	
<i>Cyberattacks</i> • 221	
<i>Web security mitigation</i> • 224	
Securing an application and its infrastructure • 226	
<i>Application and operating system hardening</i> • 226	
<i>Software vulnerability mitigation and secure code</i> • 227	
<i>Network security</i> • 227	
<i>Intrusion detection system and intrusion prevention system</i> • 230	
Data security • 232	
<i>Data classification</i> • 232	
<i>Data encryption at rest</i> • 232	
<i>Data encryption in transit</i> • 234	
Securing APIs • 235	
Security and compliance certifications	237
The cloud's shared security responsibility model	238
Securing threat modeling	241
Summary	242

Chapter 8: Architectural Reliability Considerations	245
Design principles for architectural reliability	245
Making systems self-healing by applying automation • 246	
<i>Quality assurance</i> • 246	
Creating a distributed system • 247	
Monitoring and adding capacity • 248	
Performing recovery validation • 248	
Technology selection for architectural reliability	248
Planning the RPO and RTO • 249	
Replicating data • 250	
<i>Synchronous versus asynchronous replication</i> • 250	
<i>Replication methods</i> • 251	
Planning disaster recovery • 252	
<i>Backup and restore</i> • 253	
<i>Pilot light</i> • 255	
<i>Warm standby</i> • 257	
<i>Multi-site</i> • 260	
Applying best practices for DR • 261	
Improving reliability with the cloud	262
Summary	263
Chapter 9: Operational Excellence Considerations	265
Design principles for operational excellence	266
Automating manual tasks • 266	
Making incremental and reversible changes • 267	
Predicting failures and responding • 267	
Learning from mistakes and refining • 267	
Keeping the operational runbook updated • 268	
Selecting technologies for operational excellence	268
Planning for operational excellence • 268	
IT asset management • 269	
<i>Configuration management</i> • 271	
The functioning of operational excellence • 273	
<i>Monitoring system health</i> • 274	
Improving operational excellence • 282	
<i>IT operations analytics</i> • 282	

Root Cause Analysis • 284	
Auditing and reporting • 284	
Achieving operational excellence in the public cloud	285
Driving efficiency with CloudOps	286
Summary	288
Chapter 10: Cost Considerations	289
Design principles for cost optimization	290
Calculating the total cost of ownership • 290	
Planning the budget and forecast • 292	
Managing demand and service catalogs • 293	
Keeping track of expenditure • 294	
Continuous cost optimization • 295	
Understanding techniques for cost optimization	296
Reducing architectural complexity • 296	
Increasing IT efficiency • 298	
Applying standardization and governance • 299	
Resource cost tagging • 301	
Monitoring cost usage and reports • 302	
Driving cost optimization in the public cloud	307
Green IT and its influence on cost considerations	309
Cost-effective and green application hosting on AWS • 310	
Summary	312
Chapter 11: DevOps and Solution Architecture Framework	313
Introducing DevOps	314
Understanding the benefits of DevOps • 315	
Understanding the components of DevOps	316
Continuous integration/Continuous deployment • 316	
Continuous monitoring and improvement • 318	
Infrastructure as code • 319	
Configuration management • 322	
Introducing DevSecOps for Security	323
Combining DevSecOps and CI/CD	324
Implementing a CD strategy	326
In-place deployment • 327	
Rolling deployment • 327	

Blue-green deployment • 327	
Red-black deployment • 329	
Immutable deployment • 329	
Best practices for choosing the right deployment strategy	330
Implementing continuous testing in the CI/CD pipeline	331
A/B testing • 332	
Using DevOps tools for CI/CD	334
Code editor • 334	
Source code management • 334	
CI server • 335	
Code deployment • 336	
Code pipeline • 338	
Implementing DevOps best practices	339
Building DevOps and DevSecOps in the cloud	341
Summary	343
Chapter 12: Data Engineering for Solution Architecture	345
What is big data architecture?	346
Designing big data processing pipelines	348
Data ingestion, storage, processing, and analytics	350
Data ingestion • 350	
<i>Technology choices for data ingestion</i> • 351	
<i>Ingesting data to the cloud</i> • 352	
Storing data • 353	
<i>Technology choices for data storage</i> • 354	
<i>Structured data stores</i> • 355	
<i>NoSQL databases</i> • 357	
<i>Search data stores</i> • 359	
<i>Unstructured data stores</i> • 359	
<i>Object storage</i> • 359	
<i>Vector Database (VectorDB)</i> • 360	
<i>Blockchain data stores</i> • 361	
<i>Streaming data stores</i> • 362	
Data storage in the cloud	363
Processing data and performing analytics • 364	
<i>Technology choices for data processing and analysis</i> • 366	
<i>Data processing in the cloud</i> • 367	

Visualizing data	369
Technology choices for data visualization • 369	
Designing big data architectures	370
Data lake architecture • 372	
Lakehouse architecture • 375	
Data mesh architecture • 376	
Streaming data architecture • 378	
Choosing the right big data architecture • 379	
Big data architecture best practices	381
Summary	383

Chapter 13: Machine Learning Architecture	385
--	------------

What is machine learning?	386
Types of machine learning • 387	
<i>Supervised learning</i> • 387	
<i>Unsupervised learning</i> • 388	
<i>Semi-supervised learning</i> • 388	
<i>Reinforcement learning</i> • 388	
<i>Self-supervised learning</i> • 389	
<i>Multi-instance learning</i> • 389	
Working with data science and machine learning	389
Evaluating ML models—overfitting versus underfitting • 392	
Popular machine learning algorithms • 393	
<i>Linear regression</i> • 393	
<i>Logistic regression</i> • 393	
<i>Decision trees</i> • 393	
<i>Random forests</i> • 394	
<i>K-Nearest Neighbours (k-NNs)</i> • 394	
<i>Support vector machines (SVMs)</i> • 394	
<i>Neural networks</i> • 394	
<i>K-means clustering</i> • 395	
<i>XGBoost</i> • 395	
Popular machine learning tools and frameworks • 395	
Machine learning in the cloud	399
Building machine learning architecture	401
Prepare and label • 401	
Select and build • 402	

Train and tune • 402	
Deploy and manage • 403	
ML reference architecture • 404	
Design principles for machine learning architecture	406
Organizing the machine learning system into modules • 406	
Ensuring scalability • 406	
Ensuring reproducibility • 406	
Implementing data quality assurance • 407	
Ensuring flexibility • 407	
Ensuring robustness and reliability • 407	
Ensuring privacy and security • 407	
Ensuring efficiency • 408	
Ensuring interpretability • 408	
Implementing real-time capability • 408	
Ensuring fault tolerance • 409	
MLOps	409
MLOps principles • 410	
MLOps best practices • 411	
Deep learning	413
Deep learning in the real world • 415	
<i>Healthcare: diagnosis and prognosis</i> • 416	
<i>Autonomous vehicles: navigation and safety</i> • 416	
<i>Manufacturing: quality control and predictive maintenance</i> • 416	
NLP	416
Chatbots and virtual assistants • 416	
Sentiment analysis • 417	
Text summarization • 417	
Machine translation • 417	
Summary	418
Chapter 14: Generative AI Architecture	421
What is generative AI?	422
Generative AI use cases	423
Customer experience transformation • 423	
Employee productivity enhancement • 424	
Optimizing business operations • 424	
The basic architecture of generative AI systems	425

Types of generative models • 425	
<i>Generative Adversarial Networks (GANs)</i> • 426	
<i>Variational Autoencoders (VAEs)</i> • 428	
<i>Transformer-based generative models</i> • 430	
<i>Other important generative models</i> • 432	
Importance of hyperparameter tuning and regularization in architectures • 432	
<i>Hyperparameter tuning</i> • 432	
<i>Regularization</i> • 433	
Popular generative AI FMs 433	
How to start with generative AI 436	
For end users • 436	
For builders • 437	
Using generative AI FMs in your applications with public cloud providers • 440	
<i>Choosing the right FM</i> • 443	
Preventing model hallucinations • 444	
Generative AI reference architecture for building a mortgage assistant app 446	
Challenges in implementing generative AI 448	
Training stability issues • 448	
Mode collapse • 449	
Latent space interpolation challenges • 449	
Ethical concerns and misuse • 450	
Summary 450	
Chapter 15: Rearchitecting Legacy Systems 453	
Learning the challenges of legacy systems 454	
Difficulty in keeping up with user demand • 454	
Higher cost of maintenance and updates • 455	
Shortage of skills and documentation • 456	
Vulnerability to corporate security issues • 456	
Incompatibility with other systems • 457	
Defining a strategy for system modernization 457	
Assessment of a legacy application • 458	
Defining the modernization approach • 459	
Benefits of system modernization • 460	
Looking at legacy system modernization techniques 462	
Encapsulation, rehosting, and replatforming • 463	
Refactoring and rearchitecting • 463	

Redesigning and replacing • 464	
Defining a cloud migration strategy for legacy systems	465
Documentation and support • 467	
Mainframe migration with the public cloud	467
Challenges of mainframe modernization • 468	
Migrating standalone applications • 469	
Migrating applications with shared code • 470	
<i>Application decoupling using a standalone API</i> • 470	
<i>Application decoupling using a shared library</i> • 471	
<i>Application decoupling using message queues</i> • 471	
Benefits of using the public cloud for mainframe modernization • 473	
Modernizing legacy code with generative AI	474
Summary	475
Chapter 16: Solution Architecture Document	477
Purpose of the SAD	477
Views of the SAD	478
Structure of the SAD	480
Solution overview • 482	
Business context • 484	
Conceptual solution overview • 486	
Solution architecture • 487	
<i>Information architecture</i> • 487	
<i>Application architecture</i> • 488	
<i>Data architecture</i> • 489	
<i>Integration architecture</i> • 490	
<i>Infrastructure architecture</i> • 491	
<i>Security architecture</i> • 491	
Solution implementation • 492	
Solution management • 492	
Appendix • 493	
Life cycle of the SAD	493
SAD best practices and common pitfalls	494
IT procurement documentation for a solution architecture	495
Summary	497

Chapter 17: Learning Soft Skills to Become a Better Solutions Architect	499
Importance of soft skills in solution architecture	500
Acquiring pre-sales skills	500
Key skills • 500	
Presenting to C-level executives • 501	
Taking ownership and accountability	503
Defining strategy execution with OKRs • 503	
Thinking big • 505	
Being flexible and adaptable	505
Design thinking	506
Being a builder by engaging in coding hands-on	509
Becoming better with continuous learning	509
Being a mentor to others	511
Becoming a technology evangelist and thought leader	512
Summary	512
Other Books You May Enjoy	517
Index	521

Preface

The *Solutions Architect's Handbook* guides readers to create robust, scalable, highly available, and fault-tolerant solutions by learning different aspects of solution architecture and next-generation architecture design in the cloud environment.

This book will start by detailing solution architecture and the role of solutions architect. It will take the reader through the journey of solution architecture design by providing a detailed overview of design pillars, advanced design patterns, and cloud-native aspects of modern software design. The reader will further deep dive into solution design performance optimization, security, compliance, reliability, cost optimization, and operational excellence.

This book provides an in-depth understanding of the automation of security, infrastructure, DevOps, disaster recovery, and the documentation of solution architecture. It also explains how to future-proof architecture design with data engineering, machine learning, and generative AI. As a bonus, this book also provides guidance on soft skills for solutions architects, and continuous learning techniques.

Who this book is for

This book is for software developers, system engineers, DevOps engineers, architects, and tech leaders working in the IT industry who aspire to become solutions architects and for seasoned architects to design secure, reliable, highly performant, and cost-effective architectures.

What this book covers

This book contains the following chapters:

Chapter 1: Solutions Architects in Organizations, explores the varied roles of solutions architects within an organization, detailing their responsibilities and illustrating how they align with different organizational structures.

Chapter 2: Principles of Solution Architecture Design, discusses the foundational principles that guide the creation of scalable, secure, and efficient solution architectures, emphasizing best practices in design methodology.

Chapter 3: Cloud Migration and Cloud Architecture Design, offers a roadmap for transitioning to the cloud, including strategies for migration, the benefits of cloud adoption, and the principles of cloud architecture design.

Chapter 4: Solution Architecture Design Patterns, reviews common architectural patterns, providing insights on when and how to apply them effectively to solve different architectural challenges.

Chapter 5: Cloud-Native Architecture Design Patterns, delves into design patterns specifically tailored for cloud-native environments, emphasizing scalability, resilience, and maintainability.

Chapter 6: Performance Considerations, focuses on optimizing the performance of IT systems, discussing key metrics, and strategies to enhance speed and efficiency in architectural designs.

Chapter 7: Security Considerations, addresses the critical aspect of security in solution architecture, covering best practices, patterns, and strategies to safeguard systems.

Chapter 8: Architectural Reliability Considerations, examines the principles and practices essential for building reliable systems, including disaster recovery planning and high availability design.

Chapter 9: Operational Excellence Considerations, highlights strategies and practices to achieve operational excellence, ensuring that systems are efficient, reliable, and maintainable.

Chapter 10: Cost Considerations, provides guidance on managing and optimizing costs associated with architectural solutions, focusing on maximizing value while minimizing expenses.

Chapter 11: DevOps and Solution Architecture Framework, integrates DevOps practices with solution architecture, illustrating how this synergy enhances deployment, monitoring, and maintenance.

Chapter 12: Data Engineering for Solution Architecture, introduces the fundamentals of data engineering within the context of solution architecture, emphasizing the design of data systems and workflows.

Chapter 13: Machine Learning Architecture, covers the architectural considerations for integrating machine learning into solutions, from data preparation to model deployment.

Chapter 14: Generative AI Architecture, explores the architecture behind generative AI systems, discussing the components and processes that enable AI to create novel content.

Chapter 15: Rearchitecting Legacy Systems, provides strategies and insights for modernizing legacy systems, focusing on how to transition them to contemporary, cloud-based environments.

Chapter 16: Solution Architecture Document, outlines the importance and structure of a comprehensive solution architecture document, guiding architects on how to effectively communicate their designs.

Chapter 17: Learning Soft Skills to Become a Better Solutions Architect, emphasizes the essential soft skills needed for solutions architects to succeed, including communication, problem-solving, and leadership.

To get the most out of this book

Prior experience of software architecture design will be helpful to follow this book. It's good to have a basic understanding of any popular public cloud provider such as AWS. However, there are no specific prerequisites to understand this book. All the examples and relevant instructions are provided in the various chapters. This book takes you through the concepts of solution architecture design and does not require knowledge of any particular programming language, framework, or tool.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://packt.link/gbp/9781835084236>.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and X handles. For example, “IoT platforms need to support SigV4, X.509 and custom authentication, while providing fine-grained access control with IoT policies down to the MQTT topic level.”

A block of code is set as follows:

```
def sum_of_squares_of_odd_numbers(numbers):
    # initialize the sum to zero
    sum = 0
    # Loop through the list of numbers
    for number in numbers:
        # check if the number is odd
        if number % 2 == 1:
            # square the number and add it to the sum
            sum += number ** 2
    # return the sum
    return sum
```

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes. For example: “Cloud providers such as AWS, Microsoft Azure, and GCP provide many options out of the box that can help you to modernize your system.”



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com, and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name.

Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packtpub.com>.

Share your thoughts

Once you've read *Solutions Architect's Handbook - Third Edition*, we'd love to hear your thoughts! Please [click here to go straight to the Amazon review page](#) for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/978-1-83508-423-6>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

1

Solutions Architects in Organizations

This book is your ultimate guide to solution architecture, designed to help you become a skilled solutions architect. In this chapter, you will explore the meaning of solution architecture and its significance as the foundation of solutions development in organizations. Solution architecture involves designing a robust framework that covers important areas like IT infrastructure, application security, reliability, and operational considerations.

Solutions architects work closely with stakeholders, analyzing requirements and considering constraints such as cost, budget, timeline, and regulations to create a comprehensive solution.

Solutions architects also actively engage in post-launch to ensure scalability, availability, and maintainability. Additionally, they collaborate with the sales team to promote the product and its technological benefits.

In this chapter, you will learn about the following topics:

- What is solution architecture?
- The solutions architect's role
- Understanding a solutions architect's responsibilities
- Solutions architects in an Agile organization
- Common challenges in the solutions architect role
- Career path and skill development for solutions architects

By the end of this chapter, you'll gain valuable insights into the role of solutions architects and the challenges they face. You'll discover how solutions architects handle constraints and contribute to an organization's technical vision and overall success.

What is solution architecture?

The concept of solution architecture may vary depending on the perspective of different professionals and organizations. However, at its core, solution architecture involves defining and envisioning various aspects of a business solution, considering both strategic and transactional considerations.

From a strategic standpoint, a solutions architect is responsible for developing a long-term vision for a software application. This vision ensures that the solution remains relevant and adaptable to future changes, with the ability to accommodate extensions to meet evolving user demands and workload.

On the other hand, from a tactical perspective, solution architecture focuses on the immediate needs of the business. It involves designing an application that can handle the current workload and effectively address the day-to-day challenges faced by the organization.

However, solution architecture goes beyond software alone. It encompasses the entire system, including aspects such as system infrastructure, networking, security, compliance requirements, system operation, cost considerations, and reliability.

By considering these various elements, a solutions architect creates a comprehensive blueprint that guides the development and implementation of the solution. This blueprint not only ensures that the solution meets the present needs of the business but also lays the groundwork for its future growth and success.

The benefits of solution architecture

Solution architecture is of utmost importance for several reasons. Firstly, it provides a solid foundation for the development of enterprise software solutions. As projects grow in size and teams become geographically distributed, having a well-defined solution architecture ensures long-term sustainability and effective collaboration.

Solution architecture addresses the diverse needs of a solution while maintaining alignment with the overall business context. It encompasses crucial elements such as technology platforms, application components, data requirements, resource needs, and essential **non-functional requirements (NFRs)**. These NFRs include scalability, reliability, performance, availability, security, and maintainability. By considering these aspects, solution architecture ensures that the developed solution meets the necessary standards and expectations.

Figure 1.1 shows a breakdown of the potential benefits bestowed upon an organization when employing the role of solutions architect in the business.



Figure 1.1: A solution architecture's beneficial attributes

The preceding diagram highlights the following attributes of a good solution architecture:

- **Technology in alignment with business requirements:** The solutions architect evaluates which technologies an organization or project should adopt in order to meet business requirements and achieve long-term sustainability, maintainability, and the team skill set.
- **Market opportunity evaluation:** Solution architecture involves the process of analyzing and continuously evaluating the latest trends in the market to ensure the solution developed meets customer needs, as well as business needs. It also helps with building and promoting new products.
- **Minimizing target date slippage:** A solutions architect continuously works with all stakeholders, including the business team, customers, and the development team. They make sure that the overall solution is in alignment with the business objective and launch timeline, to ensure minimal chances of target date slippage.
- **Facilitation of effective collaboration:** Solution architecture serves as a common reference point and communication tool for stakeholders involved in the project. It facilitates effective collaboration between business teams, developers, designers, and other stakeholders. Clear documentation and visualization of the solution architecture enable better understanding, alignment, and decision making among team members, ensuring everyone is on the same page and working towards shared objectives.
- **Scalability and flexibility:** A well-designed solution architecture considers scalability and flexibility as key factors. It allows the solution to adapt and grow seamlessly as the business evolves and the user workload increases. By anticipating future growth and incorporating scalability measures, the solution architecture ensures that the system can handle expanding demands without significant disruptions or costly rework.

- **Meets business goals:** The primary responsibility of a solution architecture design is to accommodate the needs of the stakeholders and adapt it to their requirements. Solution architecture converts business goals into a technical vision by analyzing market trends and implementing best practices. Solution architecture needs to be flexible enough to meet new, challenging, demanding, and rapidly changing business requirements.
- **Better resource planning:** With a clear solution architecture, organizations can precisely determine the type and amount of resources required. This facilitates strategic planning of human resources, ensuring appropriate financial resources and time, ensuring that projects are adequately staffed and resources are optimally utilized, leading to smoother project execution and adherence to timelines.
- **Better budget forecasting:** Investing in accurate estimation is crucial for effective budget forecasting. A well-defined solution architecture provides clear insights into the resources needed for project completion. Understanding the scope and requirements in detail enables organizations to predict costs more accurately and reduces the risk of budget overruns.
- **Risk mitigation:** A good solution architecture includes risk assessment and mitigation strategies. By identifying potential risks early on, solutions architects can implement measures to mitigate them. This proactive approach helps minimize the impact of risks on project timelines, budgets, and overall success. Risk mitigation strategies can include backup plans, redundancy measures, security considerations, and disaster recovery plans.
- **Increased ROI:** Solution architecture determines the ROI and helps to measure the success of the project. It forces a business to think about how to reduce costs and remove process wastage by applying automation in order to improve the overall ROI.
- **Defining a project timeline:** Defining an accurate project timeline is critical for solution implementation. A solutions architect determines the resources and effort that will be required during the design phase, which should help define the schedule for solutions development.

Now that you have had a high-level overview of solution architecture and its benefits, let's learn about the solutions architect role and how it helps to build a good solution architecture.

The solutions architect's role

If you wish to know how a solution should be organized and delivered, then a solutions architect plays an essential role in this context. A solutions architect designs the overall system and how different systems integrate across different groups. A solutions architect defines the expected outcome by working with business stakeholders and providing a clear understanding of the delivery objective on the part of the technical team.

Figure 1.2 contains a flowchart that shows the solution delivery life cycle. The solutions architect is involved in all the phases of solution design and delivery.

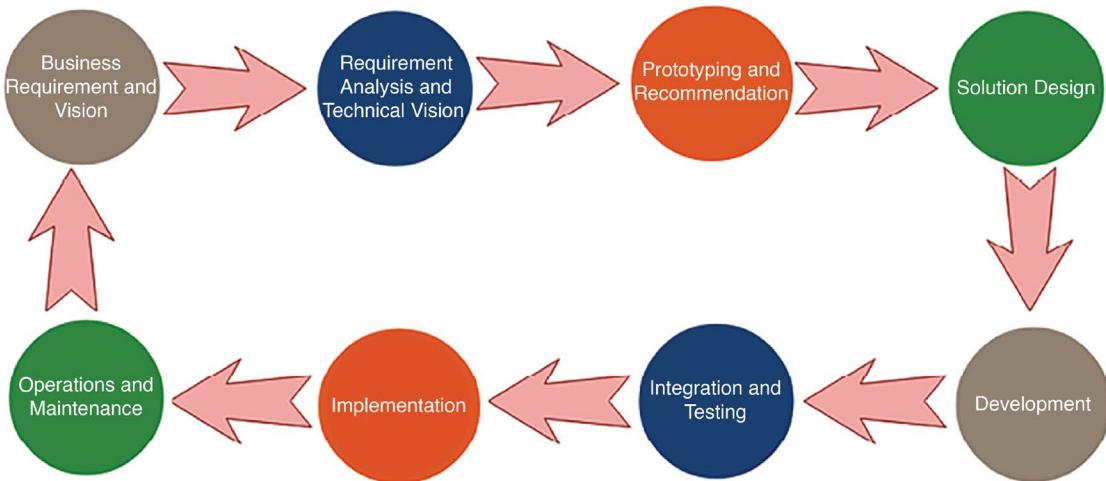


Figure 1.2: Solution delivery life cycle

As shown, the solution delivery life cycle includes the following, with the ways in which a solutions architect is involved:

- **Business Requirement and Vision:** A solutions architect works with business stakeholders to understand their vision.
- **Requirement Analysis and Technical Vision:** Analysis of the requirements, defining a technical vision in order to execute the business strategy.
- **Prototyping and Recommendation:** A solutions architect makes a technology selection by developing proof-of-concept (POC) and showcasing prototypes.
- **Solution Design:** A solutions architect develops solution designs in line with an organization's standards and in collaboration with other impacted groups.
- **Development:** They work with the development team on solutions development, and as a bridge between the business and technical team.
- **Integration and Testing:** They make sure that the final solution is working as expected with all functional requirements and NFRs.
- **Implementation:** They work with the development and deployment team for smooth implementation and guide them through any issues.
- **Operation and Maintenance:** They ensure logging and monitoring are in place and guide the team on scaling and disaster recovery as required.

The overall life cycle is an iterative process. Once the application goes into production and customers start using it, more requirements may be discovered from customer feedback, which will drive the product vision for future enhancements.

The solutions architect has major ownership during solution design, in which they do the following:

- Document solution standards
- Define high-level design
- Define cross-system integration
- Define different solution phases
- Define an implementation approach
- Define a monitoring and alert approach
- Document the pros and cons of design choices
- Document audit and compliance requirements

Solutions architects are not only responsible for solution design; they also help project managers with resource and cost estimation, defining the project's timeline and milestones, the project's release, and its support plan. The solutions architect works through different phases of the solution life cycle, from design to delivery and launch. The solutions architect helps the development team overcome obstacles and hurdles by providing expertise and a broad understanding.

Depending on the project's size and complexity, there may be a need for multiple solutions architects within a team. In general, this book explores the role of a solutions architect generically. Still, you often see solutions architects with different titles, as per the organization's structure; for example, enterprise solution, software, or technical architects. In this section, you will find some distinct attributes related to the various titles. However, the responsibilities of the solutions architects may overlap, depending on an organization's structure.

Solutions architects can be categorized as either generalists or specialists. Generalist solutions architects possess broad knowledge across multiple technical domains. They have a comprehensive understanding of various aspects of solution architecture and can provide holistic guidance. On the other hand, **specialist solutions architects (SSAs)** have deep expertise in specific areas such as big data, security, networking, or industry domains. They possess specialized knowledge and can provide in-depth guidance in their respective areas of expertise.

In many cases, a generalist solutions architect collaborates with SSAs to align the project's requirements and complexity. This collaboration allows for the leveraging of the specialized knowledge of specialists while ensuring that the overall solution architecture remains cohesive and well integrated.

The presence of both generalist solutions architects and SSAs within an organization allows for a balanced and comprehensive approach to solution architecture. It ensures that the architectural decisions and recommendations align with the project's needs, encompassing both breadth and depth of knowledge.

By combining the skills and expertise of different types of solutions architects, organizations can effectively address the unique challenges and requirements of their projects, leading to the successful design and implementation of robust solutions.

Generalist solutions architect roles

Generalist solutions architects play a critical role in solution architecture by having a broad understanding of multiple technical domains. They possess a comprehensive knowledge base that allows them to provide guidance and make informed decisions across various aspects of solution design and implementation. The following are the various types of generalist solutions architect roles.

Enterprise solutions architect

Do you ever think about how products launch in the information technology industry? This is where an enterprise solution role comes into the picture—they define best practices, culture, and suitable technologies. An enterprise architect works closely with stakeholders, subject matter experts, and management to identify organizational strategies for information technology and make sure that their knowledge aligns with company business rules.

Enterprise architects handle solution design across the organization; they create long-term plans and solutions with stakeholders and leadership. One of the most important aspects is to finalize which technologies should be used by the company and ensure the company is using these technologies with consistency and integrity.

Another important aspect of the enterprise architect's role is defining the business architecture. In some organizations, you may see *business architect* as the job title. Business architecture fills the gap between organizational strategy and its successful execution. It helps convert a map strategy into executable action items and takes this to a tactical level for implementation.

The key difference between a solutions architect and an enterprise solutions architect lies in their scope of work and focus. A solutions architect focuses on specific projects or solutions, designing and guiding the implementation of applications or systems in line with business and technology requirements. Their role is often project centric, concentrating on particular technologies or functional areas. In contrast, an enterprise solutions architect operates at a more strategic level, overseeing the organization's overall IT infrastructure and strategy. They ensure alignment of the IT strategy with business goals, integrating various solution architectures across departments. This role covers a wider range of technologies and business processes, focusing on the holistic technological landscape and strategic direction of the organization.

Overall, enterprise architects are more aligned with company visions and responsibilities when it comes to defining organization-wide standards for the successful implementation of the business' vision.

Application Architect

An application architect, sometimes referred to as a software architect, plays a vital role in software design and development. They collaborate with the organization to define technical details for software development projects. The application architect focuses on ensuring that the software aligns with industry best practices and adheres to the organization's standards. They work across different teams to understand how integration with other software modules will occur.

For example, a healthcare organization might ensure that a new patient management system integrates seamlessly with existing electronic health record systems, adhering to both healthcare regulations and internal protocols; or in a financial institution, they might oversee the development of a new banking application, ensuring it integrates securely with existing transaction processing systems and complies with financial industry standards. In both cases, the application architect ensures the software not only meets functional needs but also adheres to critical industry and organizational standards.

One of the key responsibilities of an application architect is managing the technical aspects of software development. They oversee API design, ensuring that it is well designed and performs optimally. They also consider scalability requirements, ensuring that the software can handle increasing workloads. Additionally, the application architect ensures seamless integration with other software components, making sure they can easily interact with each other.

The application architect serves as a point of contact for technical inquiries from the engineering team. They troubleshoot issues and provide guidance to ensure smooth system operation.



While smaller software development projects may not have a dedicated application architect, a senior engineer often takes on this responsibility and works on software architecture design.

In addition to technical expertise, the application architect plays a mentoring role. They support and guide the software engineering team, addressing any obstacles that arise during cross-team integration or due to evolving business requirements. Their close collaboration with the team ensures a cohesive and successful software development process.

Overall, the application architect contributes to the overall success of software projects by providing technical leadership, ensuring adherence to best practices, and supporting the engineering team throughout the development life cycle.

Cloud Architect

The cloud architect role has only come into existence in the last decade, but as cloud adoption is increasing among enterprises, this is one role that is in great demand. The role of cloud architect emerged in response to the increasing adoption of cloud technology by enterprises. As organizations move towards cloud computing, the demand for skilled professionals to plan, design, and manage cloud environments has soared.

Cloud architects are responsible for developing and implementing the company's cloud computing strategies. They possess in-depth knowledge of various cloud services and can design solutions that leverage the full potential of cloud-native capabilities.

The use of the cloud is now very popular, and it has become the norm for organizations to move onto a public cloud. With the popularity of public cloud platforms like **Amazon Web Services (AWS)**, **Microsoft Azure**, and **Google Cloud Platform (GCP)**, cloud architects play a crucial role in guiding organizations through the cloud adoption process. You will learn more about cloud architectures in *Chapter 3, Cloud Migration and Hybrid Cloud Architecture Design*.

One of the key tasks of a cloud architect is assisting organizations in migrating their existing workloads to the cloud. They develop comprehensive cloud migration strategies and design hybrid cloud architectures that seamlessly integrate on-premises applications with cloud-based resources. This enables organizations to take advantage of scalability, cost efficiency, and ease of management offered by the cloud.

For start-ups and businesses starting from scratch in the cloud, a cloud architect can design cloud-native architectures that are optimized for the cloud environment. These architectures leverage the pay-as-you-go model to optimize costs and capitalize on the automation capabilities provided by the cloud platform.

In today's business landscape, the cloud has become an integral part of enterprise strategies. To thrive in this modern era and keep up with the rapid pace of innovation and automation, having a skilled cloud architect is essential. They play a vital role in helping companies succeed by harnessing the power of the cloud and unlocking its potential for scalability, efficiency, and business growth.

Architect Evangelist

The role of an architect evangelist, also known as a technology evangelist, has emerged as a game changer in marketing, particularly in the context of complex solution platforms. In a competitive environment, people seek guidance from experts who possess in-depth knowledge and can address their queries, enabling them to make informed decisions. This is where architect evangelists step in with their expertise in specific subjects.

An architect evangelist plays a crucial role in designing architectures that cater to customer requirements and resolve their pain points. By becoming a trusted advisor for customers and partners, they possess a deep understanding of architectural concepts, issues, and market trends. This expertise helps secure platform adoption and contributes to revenue growth through increased market capture.

To drive platform adoption within the target audience, architect evangelists create public content such as blogs, whitepapers, and articles. They also actively participate in public platforms, including industry summits, technical talks, and conferences. Conducting technical workshops and publishing tutorials are also part of their repertoire, allowing them to spread awareness and generate interest in their products. Excellent written and verbal communication skills are essential for architect evangelists, and it is not uncommon to see solutions architects taking on technology evangelism as an additional responsibility.

Overall, architect evangelists serve as influential figures who promote their products and solutions to a wider audience. They leverage their expertise and communication skills to engage with customers, partners, and the community, ultimately driving adoption, growth, and market success.

Specialist solutions architect roles

In addition to the generalist solutions architects, there are specialized roles within the realm of solution architecture, depending on the organization's structure and project complexity. These SSAs focus on specific areas of expertise to address unique challenges and requirements.

The specific roles and titles of SSAs may vary across organizations. Depending on the project and organizational complexity, a solutions architect may take on multiple roles, or different solutions architects may have overlapping responsibilities. The key is to ensure that the organization has the necessary expertise and skills in each specialized area to effectively address the unique challenges and requirements of the project. Let's learn about some of the most common specialist architect roles.

Infrastructure Architect

An infrastructure architect is a specialist architect role heavily focused on enterprise IT infrastructure design, security, and data center operation. They work closely with solutions architects to make sure that the organization's infrastructure strategy is aligned with its overall business requirements, and they allocate appropriate resource capacity to fulfill this need by analyzing both the system requirements and the existing environment. They help reduce capital expenditure that could be utilized for operational spending to increase organizational efficiency and ROI.

The infrastructure architect plays a critical role in defining and planning the organization's IT resources, ranging from storage servers to individual workspaces. They develop detailed plans for procuring and setting up IT infrastructure, establish software standards, and coordinate system updates and patching across the organization. Security is a key aspect of their responsibilities, as they ensure that all environments are safeguarded against potential virus attacks. Disaster recovery planning and system backups are also part of their focus, ensuring continuous business operations.

For example, in most e-commerce businesses, planning for periods when demands will peak, such as Thanksgiving in the USA, Boxing Day in Canada and the UK, or Diwali in India, when most consumers start shopping, presents a challenge to infrastructure architects. They need to prepare enough server and storage capacity to accommodate the peak season, whose workload may be ten times higher than normal, thus increasing the cost of IT infrastructure. This system will be sitting idle for most of the year outside of the peak season.

They need to plan for cost optimization and better user experience, which is another reason they may use the cloud to fulfill additional capacity and scale on demand to reduce the cost. They need to ensure that systems are occupied while supporting the growth of new features.

In the context of the cloud, a **cloud infrastructure architect** is a specialized role within the field of infrastructure architecture that focuses specifically on designing and managing cloud-based IT infrastructures. They have in-depth knowledge of cloud platforms and services offered by major providers such as AWS, Microsoft Azure, and GCP.

Cloud infrastructure architects work closely with organizations to determine the optimal cloud architecture that meets their specific needs, considering factors such as scalability, cost efficiency, security, and performance. They design and implement cloud-based solutions, ensuring seamless integration with existing systems and applications.

Cloud infrastructure architects are responsible for planning resource allocation, managing cloud security measures, and optimizing the cloud environment for optimal performance and cost-effectiveness. Their expertise in cloud technologies enables organizations to leverage the benefits of cloud computing while ensuring a reliable and scalable infrastructure.

Overall, an infrastructure architect needs to have a good understanding of data center operation and the components involved, such as heating, cooling, security, racking and stacking, server, storage, backup, software installation and patching, load balancers, and virtualization.

Network Architect

Have you ever wondered how large enterprises with multiple offices or stores are able to connect and communicate seamlessly? Well, that's where the network architect comes into play, orchestrating the organization's network communication strategy and bringing the IT infrastructure to life.

A network architect is responsible for designing the computer network, **local area network (LAN)**, **wide area network (WAN)**, internet, intranet, and other communication systems. They manage organizational information and network systems and ensure low network latency and high network performance are available for users to increase their productivity. They establish secure connectivity between user workspaces and the internal network using **virtual private network (VPN)** connectivity.

The network architect works closely with the infrastructure architect; you may sometimes see this as an overlapping role to ensure all IT infrastructures are connected. They work with the security team and design the organization's firewall to protect against unethical attacks. They are responsible for monitoring and protecting the network via packet monitoring, port scanning, and putting an **intrusion detection system (IDS)** and **intrusion prevention system (IPS)** into place. You will learn more about IDS/IPS systems in *Chapter 7, Security Considerations*.

A network architect must stay up to date with the latest network strategies, operations, and secure connectivity techniques using VPNs. They configure load balancers, fine-tune **domain name system (DNS)** routing, and master the art of IT infrastructure connectivity. It's like building a complex web of connections, ensuring that data flows smoothly and efficiently across the organization.

Data Architect

In the age of data explosion, the role of a data architect has become increasingly vital. Think about it—every solution design revolves around data, whether it's customer information, product details, or insights derived from complex datasets. As data continues to grow exponentially, from gigabytes to terabytes and beyond, the need for effective data management and architecture is paramount. A data architect may have a different title, including analytics architect or big data architect. (I am not including the title database architect as their scope is limited to structured data inside relational databases like Oracle and Amazon Relational Database System (RDS).)

Traditionally, data was stored in structured relational databases. However, with the rise of unstructured data from sources like social media, the **Internet of Things (IoT)**, and application logs, the landscape has shifted. Enter the data architect, the visionary behind the organization's data strategy. Their role is to define rules, policies, standards, and models that govern the type of data collected and used within the organization's database. They design, create, and manage the data architecture, ensuring consistent performance and quality.

The data architect collaborates with various stakeholders, including business executives, analysts, data engineers, data scientists, and development teams. Their customers range from executives using **business intelligence (BI)** tools for data visualization to data scientists leveraging **machine learning (ML)** techniques. The data architect's goal is to meet the organization's data needs and empower users with valuable insights.

To fulfill these needs, a data architect tackles a wide range of responsibilities. They choose the right database technology, determine the storage options for structured and unstructured data, manage streaming and batch data processing, and design data lakes as centralized data stores. They also ensure data security, compliance, and encryption to protect sensitive information. Data warehousing, datamart design, and data transformation are additional areas where their expertise shines.

With the growing prominence of ML in enterprises, dedicated ML architect roles are emerging. These specialists work closely with data architects to design and implement ML algorithms and models, taking data-driven insights to the next level.

In an ever-evolving technological landscape, data architects must stay up to date with the latest database technologies, BI tools, and security measures. Their expertise in data engineering and architecture paves the way for effective data utilization, empowering organizations to unlock the full potential of their data assets.

ML Architect

In the era of **artificial intelligence (AI)** and ML, the role of ML architect has gained tremendous importance. With organizations increasingly adopting ML in their solutions, the need for experts who can design and implement robust ML architectures has become crucial.

ML architects are responsible for applying systems thinking to implement and adopt ML in the enterprise software stack. They analyze and identify the most suitable tools and technologies for ML and AI implementation based on the organization's requirements. They architect the information and data architecture to support ML, ensuring efficient data ingestion, processing, and storage for training and inference.

One of the key responsibilities of an ML architect is to modify the existing software stack and infrastructure to integrate ML capabilities seamlessly. This involves incorporating ML frameworks, libraries, and APIs into the existing ecosystem, enabling efficient data preprocessing, model training, and deployment.

Operationalizing ML solutions is another critical aspect of the ML architect's role. They establish mechanisms for continuous monitoring and improvement of ML models, ensuring optimal performance, accuracy, and reliability over time. They work closely with data scientists, data engineers, and software developers to enable the seamless deployment and scaling of ML models in production environments.

ML architects must have a deep understanding of architecture best practices, performance optimization techniques, security considerations, compliance requirements, cost optimization strategies, and operational excellence in the context of AI and ML solutions. They design architectures that adhere to these principles while considering the cloud-native aspects of modern ML technology stacks.

In *Chapter 13* of this book, you will dive deeper into the world of ML architecture, exploring design pillars, advanced design patterns, anti-patterns, and the cloud-native aspects of modern AI and ML technology stacks. It will equip you with the knowledge and skills needed to architect and deploy robust and scalable ML solutions.

ML is transforming industries and driving innovation across various domains. As organizations continue to leverage the power of ML, the role of an ML architect becomes indispensable in helping organizations harness the full potential of AI and ML for business success.

GenAI architect

In addition to ML, another emerging field that has gained significant attention is **generative artificial intelligence (GenAI)**. GenAI focuses on creating intelligent systems that possess human-like cognitive abilities and can perform a wide range of tasks across multiple domains.

GenAI architects are responsible for designing and developing advanced AI systems that go beyond specific use cases and can exhibit general intelligence. They explore cutting-edge technologies such as deep learning, reinforcement learning, natural language processing, and computer vision to build intelligent systems capable of reasoning, learning, and adapting in real time.

GenAI architects leverage their expertise in neural networks, cognitive science, and computational models to create architectures that enable machines to understand complex data, make decisions, and solve problems in a manner that simulates human intelligence. They work closely with interdisciplinary teams, including data scientists, computer scientists, and domain experts, to shape the overall GenAI solution.

Designing a GenAI architecture involves addressing challenges such as ethical considerations, and the ability to handle uncertainty and ambiguity. GenAI architects focus on building systems that can learn from limited data, transfer knowledge across domains, and exhibit robust performance in dynamic and unpredictable environments.

In *Chapter 14* of this book, you will delve into the fascinating world of GenAI architecture, exploring the principles, techniques, and challenges associated with building intelligent systems that can achieve GenAI. You will gain insights into the latest advancements, architectural paradigms, and ethical considerations in GenAI, empowering you to design and develop intelligent systems that push the boundaries of AI capabilities.

As the field of AI continues to advance, GenAI offers immense potential for transforming industries, revolutionizing automation, and enabling machines to perform complex tasks that were previously thought to be the sole domain of human intelligence. The role of GenAI architects is pivotal in driving this transformation and shaping the future of intelligent systems.



The integration of ML and GenAI into solution architecture has brought about exciting possibilities for intelligent automation, personalized experiences, and breakthrough innovations in various industries.

Security Architect

In today's digital landscape, ensuring the security of organizational data and systems is of paramount importance. The role of a security architect becomes crucial in designing and implementing robust security measures to protect against potential threats and vulnerabilities.

A security architect collaborates with various teams and external vendors to prioritize security across the organization. They are responsible for designing and deploying network and computer security solutions, safeguarding information systems, and securing company networks and websites. They also play a key role in vulnerability testing, risk analysis, and security audits to identify potential weaknesses and develop mitigation strategies.

As part of their responsibilities, security architects review and approve the installation of firewalls, VPNs, routers, and other security measures. They conduct thorough testing of security processes to ensure their effectiveness and provide technical guidance to security teams. Compliance with industry standards and regulations is a crucial aspect of their role, ensuring that applications adhere to necessary security protocols and that data is appropriately encrypted and accessible.

Security architects possess a deep understanding of security technologies, tools, and techniques, and are skilled in designing comprehensive security architectures that cover data, network, infrastructure, and applications. Their expertise and knowledge play a critical role in protecting organizations from cyber threats and ensuring the confidentiality, integrity, and availability of sensitive information.

In *Chapter 7* of this book, you will delve into the security considerations, exploring the principles, best practices, and emerging trends in security architecture. You will gain insights into the methodologies for assessing risks, implementing security controls, and fostering a culture of security within organizations. By understanding the role of security architects and the intricacies of security design, you will be equipped to create robust security architectures that fortify organizations against potential threats and safeguard their valuable assets.

DevOps architect

In today's fast-paced and highly competitive landscape, organizations are seeking ways to streamline their development and operations processes to deliver applications faster, more efficiently, and with higher quality. This is where the role of a DevOps architect becomes crucial.

DevOps is a collaborative approach that bridges the gap between development and operations teams, enabling them to work together seamlessly. A DevOps architect plays a pivotal role in driving this collaboration and implementing practices and tools that automate various aspects of the software delivery life cycle.

One of the key responsibilities of a DevOps architect is to establish and optimize **continuous integration and continuous deployment (CI/CD)** pipelines. They automate the build, testing, and deployment processes to ensure that code changes are thoroughly tested and seamlessly deployed to production environments. By automating these processes, organizations can reduce errors, accelerate release cycles, and deliver software more reliably.

Infrastructure as code (IaC) is another important aspect of a DevOps architect's role. They leverage tools like Chef, Puppet, Ansible, and Terraform to define and automate the provisioning and configuration of infrastructure resources. This enables the development and operations teams to easily create, replicate, and manage environments, providing greater flexibility and scalability.

Monitoring and alerting are essential components of a robust DevOps architecture. A DevOps architect plans and implements monitoring solutions that continuously monitor the application, infrastructure, and security incidents. Automated alerts are set up to promptly notify the appropriate teams in case of any issues or significant changes, allowing for quick response and resolution.

Disaster recovery is also a critical consideration for a DevOps architect. They design and implement deployment strategies that ensure organizations can recover from failures or disasters with minimal data loss (**recovery point objective (RPO)**) and downtime (**recovery time objective (RTO)**). By planning for disaster recovery in advance, organizations can minimize the impact of potential disruptions and maintain business continuity.

In *Chapter 11* of this book, you will dive deeper into the world of DevOps in the aspect of the solution architecture framework. You will explore the principles, methodologies, and tools used in DevOps, and understand how to integrate DevOps practices into your solution architecture. By embracing DevOps under the guidance of a skilled DevOps architect, organizations can enhance collaboration, accelerate delivery, and achieve greater agility in today's dynamic technology landscape.

Industry Architect

Industry architect is a specialist role that focuses on designing solutions specifically tailored to a particular industry or vertical. They possess deep knowledge and expertise in the specific domain and understand the unique challenges, requirements, and regulations associated with that industry.

The role of an industry architect is to collaborate closely with stakeholders, including business executives, subject matter experts, and technology teams, to understand the specific needs and goals of the industry. They analyze industry trends, emerging technologies, and best practices to develop architectural strategies that align with the industry's objectives.

Industry architects are responsible for translating business requirements into technical solutions that address industry-specific challenges. They design and develop industry-specific software applications, systems, and platforms that cater to the specific needs of the industry. This includes considering factors such as compliance, data privacy, security, scalability, and interoperability.

In addition, industry architects play a crucial role in staying up to date with the latest innovations and advancements within the industry. They continuously evaluate new technologies, tools, and frameworks that can enhance the industry's operations and drive competitive advantage.

Collaboration and communication skills are essential for industry architects as they need to work closely with diverse stakeholders, including business leaders, developers, data analysts, and regulatory authorities. They act as trusted advisors, providing guidance and recommendations on technology adoption, architectural decisions, and digital transformation initiatives within the industry.

By leveraging their industry expertise and architectural knowledge, industry architects contribute to the growth, efficiency, and digital transformation of organizations operating in specific sectors. Their role is critical in ensuring that technology solutions align with industry standards, regulations, and best practices, ultimately driving innovation and success within the industry.

Here are some examples of industry architects in specific sectors:

- **Finance industry architect:** They specialize in technology solutions for financial institutions, understanding complex regulations and security needs. They develop solutions for risk management, fraud detection, and compliance in the finance sector.
- **Manufacturing industry architect:** They design solutions for manufacturing sectors like automotive and consumer goods, focusing on supply chain optimization, production planning, and industrial IoT for enhanced efficiency and productivity.
- **Retail industry architect:** They develop technology solutions for retail, including POS systems, CRM, and omnichannel experiences. They address data security and integrate physical and digital retail channels.
- **Healthcare industry architect:** They focus on healthcare solutions, designing systems for EHR, patient management, and telemedicine. They address privacy, security, and compliance with healthcare regulations.

These are just a few examples of industry architects and the sectors they specialize in. Each industry has its unique challenges, requirements, and technology landscape, and industry architects play a critical role in designing tailored solutions that meet the specific needs of that industry.

The SSA role extends beyond industry and technology domains to include specific SaaS providers like Salesforce, ServiceNow, Databricks, and Snowflake, as well as enterprise workloads from SAP, VMware, Microsoft, Oracle, and cloud platforms like AWS, GCP, and Azure. Covering every variant of the SSA role in one section is challenging; therefore, this section focuses on the general concept of the SSA role, emphasizing its diversity and the breadth of specialization within the field.

As you have learned about various solutions architect roles, let's now dive deep into their responsibilities.

Understanding a solutions architect's responsibilities

Now that we have broken down the various roles of a solutions architect, we will next take a look at the details of the responsibilities of a solutions architect. A solutions architect is a technical leader in a customer-facing role, which comes with many responsibilities. The primary responsibility of a solutions architect is to convert an organization's business visions into a technical solution and work as a liaison between businesses and technical stakeholders. A solutions architect uses broad technological expertise and business experience to ensure the success of the solution's delivery.



A solutions architect's responsibilities may differ slightly based on the nature of the organization. Often, in a consulting organization, a solutions architect may be dedicated to a particular project and customer, while in a product-based organization, a solutions architect may be working with multiple customers to educate them on a product and review their solution design.

A solutions architect carries various responsibilities at different stages of the application development cycle, even before the project is kicked off. During the project incubation phase, the solutions architect works with business stakeholders to prepare and evaluate the **request for response (RFR)** document.

Once a project is kicked off, the solutions architect analyzes the requirements to decide on the feasibility of technical implementation, while defining NFRs such as scalability, high availability, performance, and security. A solutions architect understands various project constraints and makes the technology selection by developing a POC.

Once development starts, the solutions architect mentors the development team and adjusts both technical and business needs.

After the application has launched, the solutions architect makes sure that the application performs as per the defined NFRs, and identifies the next iteration based on user feedback.

You will learn more about the solutions architect role at various stages of the product development life cycle in this section. Overall, a solutions architect holds the following primary responsibilities, detailed in *Figure 1.3*.

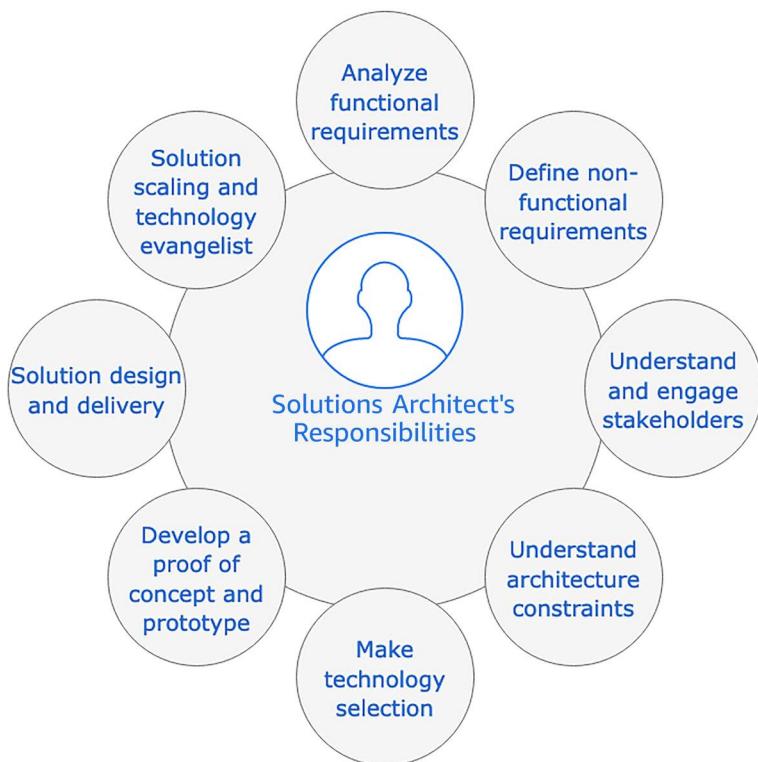


Figure 1.3: A solutions architect's responsibilities model

As shown, there are various significant responsibilities for a solutions architect. In the following sections, you will learn about the various aspects of the solutions architect's responsibilities.

Analyze functional requirements (FRs)

At the onset of any project, defining business requirements is fundamental to solution design. These requirements, initially presented in basic form, necessitate the involvement of a varied group from the start, including those with technical expertise, to accurately identify and understand these requirements. Business stakeholders initially set these requirements, but as the project technologically evolves, frequent adjustments are often required. This is where the role of a solutions architect becomes pivotal, not just in designing the application but also in shaping the overall business outcome.

Solutions architects go beyond technical expertise, integrating deep business insights to align technology with business goals. They work closely with product managers and stakeholders, bridging FRs with technical solutions and becoming trusted advisors. Their role is crucial in visualizing the end product and its implementation, guiding projects to not only meet technical specifications but also fulfill strategic business objectives and meet user expectations.

In essence, the role of a solutions architect transcends traditional boundaries of technical expertise. They are pivotal in bridging the gap between technical possibilities and business realities, ensuring that the final solution not only adheres to technical specifications but also delivers real business value. Their ability to work with a diverse set of stakeholders, understand the nuances of business needs, and foresee potential challenges makes them indispensable in the journey from conceptualization to the realization of a project. The success of a project often hinges on their capacity to translate complex requirements into a coherent, functional, and efficient solution architecture.

FRs specify what a system should do, detailing the behaviors, functions, and features the application must support. They are directly related to the user interactions and tasks the application will perform. NFRs, on the other hand, define how a system performs certain functions, outlining the system's quality attributes, such as performance, usability, reliability, and security. These requirements describe the system's operation conditions and constraints that affect the user experience but not the specific behaviors. Let's learn more about NFRs and how solution architects help to uncover them.

Define NFRs

NFRs may not be visible to users and customers directly, but their absence may impact the overall user experience in a negative way, and hamper the business. NFRs include critical aspects of the system, such as performance, latency, scalability, high availability, and disaster recovery. The most common NFRs are shown in *Figure 1.4*:

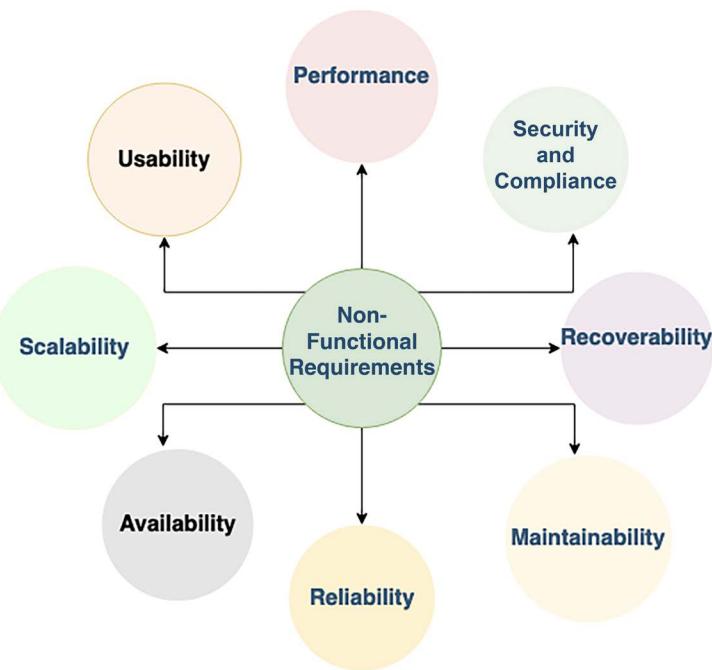


Figure 1.4: NFRs in a solution design

In relation to NFRs, solutions architects ask themselves the following questions:

- **Performance:**
 - What will the application load time be for users?
 - How can we handle network latency?
- **Security and compliance:**
 - How can we secure an application from unauthorized access?
 - How can we protect an application from malicious attacks?
 - How do we adhere to local laws and audit requirements?
- **Recoverability:**
 - How can we recover an application from an outage?
 - How can we minimize recovery time in the event of an outage?
 - How can we recover lost data?

- **Maintainability:**
 - How can we ensure application monitoring and alerts?
 - How can we ensure application support?
- **Reliability:**
 - How can we make sure the application performs consistently?
 - How do we inspect and correct glitches?
- **Availability:**
 - How can we ensure the high availability of an application?
 - How can we make an application fault-tolerant?
- **Scalability:**
 - How can we meet the increasing demand for resources?
 - How can we accommodate scaling for a sudden spike in utilization?
- **Usability:**
 - How can we simplify an application's use?
 - How can we achieve a seamless user experience?
 - How can we make the application accessible to a diverse set of users?



Depending on the nature of the project, however, there may be certain NFRs that are suitable only for that particular project (for example, voice clarity for a call center solution).

You will learn more about these attributes in *Chapter 2, Principles of Solution Architecture Design*.

The solutions architect becomes engaged in a project from a very early stage, which means they need to design a solution by gauging requirements across the stakeholders within an organization. The solutions architect needs to ensure consistency in solution design across system components and requirements. They are responsible for defining NFRs across these different components and across different groups since they make sure that the desired usability of a solution is achieved across the board.

NFRs are an integral and essential aspect of solution design that tends to slip when teams are too focused on business requirements, which can impact the user experience. A good solutions architect has the primary responsibility of conveying the importance of NFRs and ensuring that they are implemented as part of solution delivery.

Understand and engage stakeholders

A stakeholder is anyone who has an interest in the project, whether directly or indirectly. As well as the customer and user, stakeholders may also be the development, sales, marketing, infrastructure, network, or support team, or the project funding group. Stakeholders can also be internal or external to the project. Internal stakeholders include the project team, sponsors, employees, and senior management; external stakeholders include customers, suppliers, vendors, partners, shareholders, auditors, and the acting government of a country.

Stakeholders often have a different understanding of the same business problem as per the context they find themselves in; for example, a developer may look at a business requirement from a coding perspective, while an auditor may look at it from one of compliance and security.

A solutions architect's role involves working with both technical and non-technical stakeholders to ensure the success of a project. They need to understand the project requirements from various perspectives, which necessitates engaging with a diverse group of stakeholders. This includes translating complex technical concepts for non-technical stakeholders and ensuring the technical team understands the business objectives. By collaborating with all involved parties, the solutions architect ensures that the technical solution aligns with the broader business goals. This broad collaboration is essential for developing a comprehensive and effective solution that meets everyone's needs.

Solutions architects possess excellent communication skills and negotiation techniques, which help them to ascertain the optimal path for a solution while keeping everyone on board. A solutions architect works as a liaison between technical and non-technical resources and fills the communication gap. Often, those communication gaps between a businessperson and the technical team become a reason for failure. The businessperson tries to look at things from more of a feature and functionality perspective, while the development team strives to build a more technically compatible solution, which may sometimes lean toward the non-functional side of the project.

The solutions architect needs to make sure both teams are on the same page, and that the suggested features are also technically compatible. They mentor and guide the technical team as required and put their perspective into simple language that everyone can understand.

Understand architecture constraints

Architecture constraints are one of the most challenging attributes of solution design. Architecture constraints significantly challenge solution design because they limit flexibility and innovation. Ensuring new solutions are technically compatible with existing systems within these constraints requires considerable effort and resources. Additionally, constraints related to budget, resources, and timelines can impact the quality and scope of the solution. Complying with industry standards and regulatory requirements while meeting functional needs is a delicate balance to maintain.

A solutions architect needs to manage architectural constraints carefully and be able to negotiate between them to find an optimal solution. Often, these constraints depend on each other, and emphasizing one limitation can inflate others. The most common constraints are presented in *Figure 1.5*.

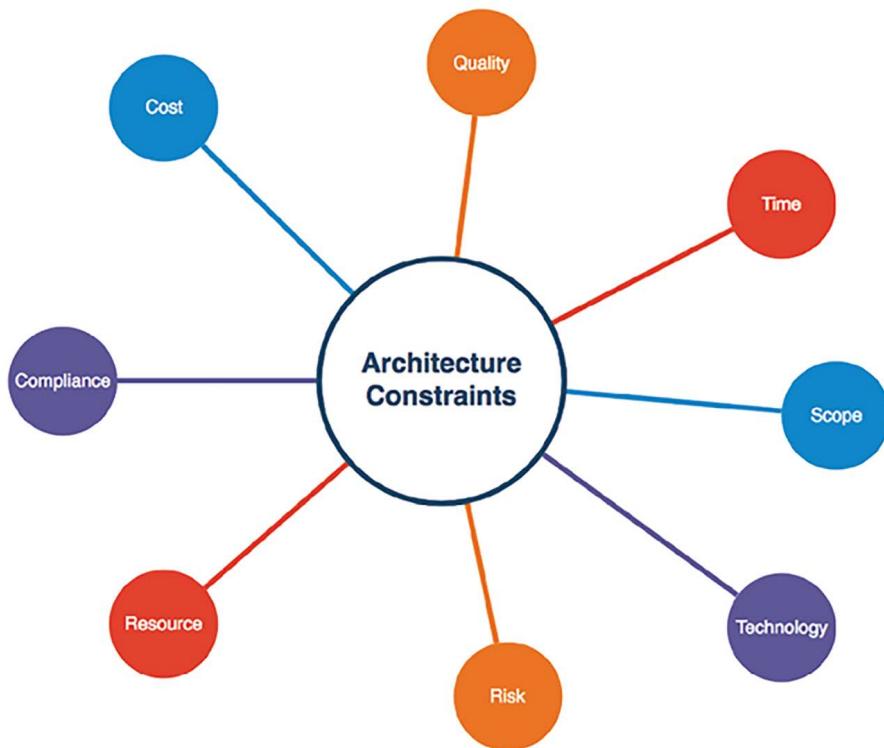


Figure 1.5: Architectural constraints in a solution design

The solution design should consider the following constraints:

- **Cost:**
 - How much funding is available for solution implementation?
 - What is the expected ROI?
- **Quality:**
 - How closely should outcomes match FRs and NFRs?
 - How can we ensure and track the quality of the solution?
- **Time:**
 - When should the output be delivered?
 - Is there any flexibility regarding delivery time?
- **Scope:**
 - What are the exact expectations from business and customer requirements?

- How does the requirement gap need to be handled and accommodated?
- **Technology:**
 - What technology can be utilized?
 - What flexibility does using legacy versus new technologies provide?
 - Should we build in-house or source from a vendor?
- **Risk:**
 - What can go wrong and how can we mitigate it?
 - What is the risk tolerance of stakeholders?
- **Resource:**
 - What is required to complete solution delivery?
 - Who will work on the solution's implementation?
- **Compliance:**
 - What are the local legal requirements that might impact the solution?
 - What are the audit and certification requirements?

A solutions architect needs to balance constraints and analyze the trade-off of each; for example, saving costs by reducing resources may impact the delivery timeline.

Achieving a schedule with limited resources may affect quality, which in turn increases costs due to unwanted bug fixes. So, finding the balance between cost, quality, time, and scope is significant. **Scope creep** is one of the most challenging situations that a solutions architect may be faced with, as it can negatively impact all other constraints and increase the risks to solution delivery.



Scope creep refers to the gradual expansion of a project's objectives and deliverables, often without corresponding increases in resources, time, or budget.

It is essential for a solutions architect to understand all the aspects of every constraint and to be able to identify any resulting risk. They must put risk mitigation plans into place and find a balance between them. Handling any scope creep can help a lot in delivering the project on time.

Make technology selections

Technology selection is the key aspect of a solutions architect's role and may involve the most complexity. There is a broad range of technologies available, and a solutions architect is required to identify the right ones for the solution.

The solutions architect needs to have breadth and depth in their knowledge of technologies to make the best decision since the chosen technology stack can impact the overall delivery of the product.

Each problem can have multiple solutions and an available range of technologies. To make the right selection, a solutions architect needs to keep FRs and NFRs in mind, and define selection criteria while making a technology decision. The selected technology needs to consider different perspectives, whether the goal is the ability to integrate with other frameworks and APIs or to meet performance requirements and security needs.

A solutions architect should be able to choose the technology that not only satisfies current requirements but also scales for future needs.

Develop a POC and prototype

Creating a prototype is probably the most fun part of being a solutions architect. To choose a proven technology, a solutions architect needs to develop a POC in various technology stacks to analyze their fit for the FRs and NFRs of the solution. The solution design POC is when a solutions architect is trying to figure out the building blocks of the solution.

The idea of developing a POC is to evaluate technology with a subset of critical functional implementations, which can help us decide on a technology stack based on its capabilities. It has a short life cycle and is limited to being reviewed by experts within a team or organization.

After evaluating multiple platforms using a POC, the solutions architect may proceed with prototyping to a technology stack. A prototype is developed for demonstration purposes and given to the customer so that it can be used to secure funding. POCs and prototyping are far from being production ready; solutions architect builds have limited functionality, which can prove to be a challenging aspect of solution development.

Solution design and delivery

Solutions architects work on solution design after understanding different aspects of FRs, NFRs, solution constraints, and technology selection. In an agile environment, this is an iterative approach where the requirements may change over time and need to accommodate the solution design.

The solutions architect needs to design a future-proof solution, which should have strong building blocks and be flexible enough to adjust to changes that can occur due to user demands or technology enhancements. For example, if the user demands increase ten times, then an application should be able to scale and accommodate user demands without significant changes to the architecture. Similarly, if new technology, such as ML or blockchain, gets introduced to solve a problem, your architecture should be able to accommodate them; for example, using AI to build a recommendation system on top of existing data for an e-commerce application.



The solutions architect needs to be careful about drastic changes to the requirements and apply a risk mitigation plan.

For future-proof design, you can take the example of a loosely coupled microservice architecture based on RESTful APIs. These architectures can be extended to new requirements and have the ability to integrate easily. You will learn more about different architecture designs in *Chapter 4, Solution Architecture Design Patterns*, and *Chapter 5, Cloud-Native Architecture Design Patterns*.

Ensuring post-launch operability and maintenance

The solutions architect plays an integral role after the solution's launch with respect to product operability. To handle the increasing user base and product utilization, a solutions architect should know how to scale the product to meet demands and ensure high availability without impacting the user experience.

In unforeseen events such as outages, a solution architecture guides infrastructure, IT support, and software deployment teams to execute a disaster recovery plan for business process continuation. The solutions architect satisfies the organization's RPOs and RTOs. RPOs define how much data loss an organization can tolerate in terms of the volume of data lost during the outage interval—for example, a loss of 15 minutes of data. RTOs define how much time the system should take to get back up and running. You will learn more about RTOs and RPOs in *Chapter 11, DevOps and Solution Architecture Framework*.

In the event of performance issues due to an increase in demand, the solutions architect helps scale the system horizontally to mitigate application bottlenecks, or vertically to alleviate database bottlenecks. You will learn more about different scaling mechanisms and self-healing in *Chapter 8, Architectural Reliability Considerations*.

The solutions architect plans to accommodate any new requirements in an existing product that arise from usage patterns or for any other reason. They can make changes to NFRs based on monitoring user behavior; for example, users may leave a page if it takes more than three seconds to load. The solutions architect works through this and guides the team in handling issues that may occur post release.

Solution scaling and technology evangelism

Being an evangelist is the most exciting part of the solutions architect role. The solutions architect increases product and platform adoption by spreading the word through public forums. They write blogs about solution implementation and conduct workshops to showcase the potential benefits and the use of technology platforms.

They build mass support for technologies and help establish a standard. A solutions architect should be passionate about technology. They should be an excellent public speaker and possess excellent writing skills to perform the technology evangelist role.

Solutions architect in an Agile organization

The Agile model is becoming very popular. It represents a significant shift from traditional project management methodologies. Unlike the traditional waterfall model, which follows a linear and sequential approach, Agile emphasizes flexibility, collaboration, and adaptability. It involves iterative development, where projects are divided into small, manageable units allowing for frequent reassessment and adaptation. This approach encourages continuous feedback and customer involvement throughout the project life cycle, contrasting with the traditional model's rigid structure and tendency to incorporate feedback only at specific stages. Agile's dynamic nature makes it particularly suitable for projects where requirements are expected to evolve or are not fully defined at the outset.

What comes to mind when thinking about the solutions architect in an agile model? There are many myths, such as thinking that solution architecture is a very complex activity, and with agile you will be asked to submit your design right away or in the next sprint cycle. Another myth is that agile architecture will not be robust to such architecture design and development, or that testing is not possible.

A solutions architect in an agile environment needs to follow an iterative re-architect concept by inspecting and adapting the approach. It's about choosing the right solution for enterprises, communicating well, taking continuous feedback, and modeling in an agile way. The development team needs a solid architectural foundation and the ability to adapt to changing requirements; they need guidance and mentoring from a solutions architect.

The foundation of the agile architecture should involve reducing the cost of changes, reducing unnecessary requirements by challenging them, and creating a framework to reverse incorrect requirements rapidly. The agile architect builds prototypes to minimize risk and plans for change by understanding them. They design the prototypes while balancing the needs of all stakeholders and creating a loosely coupled architecture that can easily integrate with other modules.

Agile architecture advocates designing decoupled and extendable interfaces, automation, rapid deployment, and monitoring. Solutions architects can build decoupled designs using microservice architecture and rapid deployment using test framework automation with a continuous deployment pipeline. You will learn more about various loosely coupled architecture patterns in *Chapter 4, Solution Architecture Design Patterns*.

Common challenges in the solutions architect role

While it is an exciting and dynamic position, the role of a solutions architect comes with its fair share of challenges. Understanding and addressing these challenges is crucial for success in the role.

Here are some of the common challenges in the solutions architect role:

- **Balancing business and technical requirements:** Solutions architects need to strike a balance between meeting the business objectives and ensuring the technical feasibility of the solution. This requires understanding both the business needs and the technical capabilities and finding the optimal solution that satisfies both aspects.

- **Managing complexity:** Solutions architects often work with complex systems and technologies, which can be challenging to understand and integrate. They need to navigate through intricate technical landscapes, incorporating different components and ensuring seamless interoperability.
- **Keeping up with technology advancements:** The technology landscape is constantly evolving, with new tools, frameworks, and methodologies emerging regularly. Solutions architects must stay up to date with the latest advancements and industry trends to provide innovative and effective solutions.
- **Stakeholder management:** Solutions architects work with various stakeholders, including business leaders, developers, project managers, and end users. Managing different expectations, requirements, and priorities can be demanding. Effective communication, collaboration, and negotiation skills are essential to address diverse stakeholder needs.
- **Addressing scalability and performance:** Solutions architects must design solutions that can handle increasing data volumes, user loads, and evolving business requirements. Ensuring scalability, performance, and reliability are critical challenges, as solutions need to accommodate future growth without sacrificing efficiency.
- **Security and compliance:** Data security and regulatory compliance are major concerns in today's digital landscape. Solutions architects must incorporate robust security measures, encryption techniques, and compliance frameworks into their designs to protect sensitive data and ensure adherence to industry standards.
- **Resolving conflicting requirements:** Different stakeholders often have conflicting requirements or priorities. Solutions architects must navigate these conflicts, identify trade-offs, and find the best compromise that satisfies the overall goals of the solution.
- **Managing project constraints:** Solutions architects need to work within the constraints of budgets, timelines, and resources. They must make informed decisions, optimize resource allocation, and adapt to changing project dynamics to ensure successful solution delivery.
- **Adoption of cloud technologies:** With the growing popularity of cloud computing, solutions architects often face the challenge of effectively leveraging cloud platforms and services. They need to understand the intricacies of cloud architecture, deployment models, and vendor-specific tools to design scalable and cost-effective cloud-based solutions.
- **Continuous learning and skill development:** Given the fast-paced nature of technology, solutions architects must invest in continuous learning and skill development. They need to acquire new knowledge, enhance their technical expertise, and stay updated with industry best practices to remain effective in their role.

By recognizing these challenges and proactively addressing them, solutions architects can navigate the complexities of their role and deliver successful solutions that meet business objectives while aligning with technical requirements.

Career path and skill development for solutions architects

The career path of a solutions architect can vary depending on the organization, industry, and individual aspirations. Here, we look at a general outline of the career path and skills development for solutions architects.

Career path

The career path of a solutions architect typically involves a series of progressive roles, starting from an educational foundation:

- **Educational foundation:** A bachelor's degree in computer science, software engineering, or a related field is typically required to start a career as a solutions architect. Building a strong foundation in software development, system design, and IT concepts is essential.
- **Professional experience:** Solutions architects usually begin their careers as software developers, systems analysts, or technical consultants. Gaining hands-on experience in designing and implementing software solutions helps develop a deep understanding of practical application development and IT infrastructure.
- **Solution design and architecture:** As professionals progress in their careers, they transition into solution design and architecture roles. They work closely with stakeholders, analyze business requirements, and design scalable, reliable, and cost-effective solutions. Developing expertise in solution architecture frameworks and methodologies, such as **The Open Group Architecture Framework (TOGAF)** or the Zachman Framework, is beneficial.

Skill development

To enhance their career prospects, solutions architects should focus on developing skills in the following areas:

- **Technical expertise:** Solutions architects need a broad range of technical skills across different domains, such as application development, database management, networking, cloud computing, and security. They should continuously enhance their technical knowledge to stay updated on the latest technologies and industry trends.
- **Communication and collaboration:** Effective communication and collaboration skills are crucial for solutions architects. They must be able to translate technical concepts into understandable terms for non-technical stakeholders, facilitate discussions, and build consensus. Developing strong interpersonal and leadership skills is essential to effectively work with cross-functional teams.
- **Business acumen:** Solutions architects need to align technology solutions with business objectives. Developing business acumen helps them understand organizational strategies, industry dynamics, and customer needs. They should be able to analyze the impact of technology decisions on the overall business and make recommendations accordingly.
- **Leadership and management:** As solutions architects progress in their careers, they may take on leadership and management roles, overseeing teams of architects or managing solution delivery projects. Developing skills in project management, team leadership, and strategic planning enhances their ability to drive successful outcomes.

- **Continuous learning:** The field of technology is constantly evolving, and solutions architects need to be proactive in their learning journey. Staying updated on emerging technologies, industry best practices, and new architectural patterns is essential. Pursuing certifications and attending industry conferences and workshops can help in continuous professional development.

In today's digital landscape, cloud computing has become integral to solution architecture. Cloud platforms offer scalability, flexibility, and cost efficiency, enabling rapid deployment and scaling of applications. They also provide access to advanced technologies like AI, big data analytics, and IoT, which are integral to digital transformation strategies. Therefore, proficiency in cloud solutions is essential for solutions architects to design effective, future-proof, and competitive technology solutions.

Here are some key points regarding cloud knowledge and certifications for solutions architects:

- **Cloud platforms:** Solutions architects should be familiar with major cloud platforms such as **Amazon Web Services (AWS)**, **Microsoft Azure**, and **Google Cloud Platform (GCP)**. They should understand the core services, architectural patterns, scalability options, and security features offered by these platforms.
- **Cloud architecture:** Solutions architects need to be proficient in designing cloud-based architectures that leverage the capabilities of cloud platforms effectively. This includes designing highly available and scalable solutions, implementing fault-tolerant systems, and optimizing cost and performance in the cloud environment.
- **Cloud security:** Security is a critical aspect of cloud computing. Solutions architects should have knowledge of cloud security best practices, encryption mechanisms, identity and access management, and compliance standards specific to the cloud environment. Understanding how to design and implement secure cloud architectures is essential.
- **Cloud storage and databases:** Solutions architects should have a good understanding of cloud storage options such as object storage, block storage, and file storage, and how to choose the appropriate storage solution based on specific requirements. Additionally, knowledge of cloud-based database services like **Amazon RDS**, **Azure SQL Database**, and **Google Cloud Spanner** is beneficial.
- **Cloud certifications:** Cloud certifications validate an individual's expertise in cloud technologies and provide credibility in the industry. Popular cloud certifications for solutions architects include **AWS Certified Solutions Architect**, **Microsoft Certified: Azure Solutions Architect Expert**, and **Google Cloud Certified – Professional Cloud Architect**. These certifications demonstrate proficiency in designing and implementing cloud-based solutions.

Having cloud knowledge and certifications not only enhances the skill set of solutions architects but also demonstrates their ability to design and implement scalable, reliable, and secure cloud-based solutions. It boosts their professional credibility and increases their marketability in an increasingly cloud-centric industry. You can learn more about developing a career as an AWS-focused cloud solutions architect by referring to the book *AWS for Solutions Architects* (<https://www.amazon.com/gp/product/180323895X/>).

Summary

In this chapter, we explored the role of a solutions architect within an organization, providing a comprehensive overview of their responsibilities, skills, and challenges. It began by defining solution architecture and tracing its evolution over time, highlighting its significance in driving successful project outcomes and aligning technical solutions with business goals.

The chapter introduced various roles within the solutions architect domain, both generalist and specialist. Each role was described, shedding light on their unique responsibilities and expertise.

A deep dive into the responsibilities of a solutions architect followed, covering essential aspects such as analyzing user requirements, defining NFRs, engaging with stakeholders, managing architecture constraints, selecting appropriate technologies, developing POCs, designing and delivering solutions, ensuring post-launch operability and maintenance, and acting as a technology evangelist.

The chapter also acknowledged the integration of solutions architects in agile teams, emphasizing the importance of collaboration, adaptability, and continuous improvement within the agile development process.

Common challenges faced by solutions architects were addressed, providing insights into overcoming these hurdles effectively. The chapter emphasized the need for continuous career development and skill enhancement to stay abreast of evolving technologies and industry trends.

By delving into the core aspects of the solutions architect role, this chapter served as a comprehensive guide for aspiring and practicing professionals. The provided overview sets the stage for a comprehensive exploration of solution architecture.

Subsequent chapters delve into the principles of designing scalable, resilient, and high-performance architectures. They cover crucial aspects such as applying security measures, navigating architectural constraints, and implementing changes through testing and automation.

Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below to get a free eBook of your choice.



2

Principles of Solution Architecture Design

This chapter throws light on the most important and common design principles and attributes of solution architecture. While the focus in this chapter is on the most pivotal design elements, it's worth noting that additional design aspects might arise, contingent on the complexity of the product and the specific industry domain. As you advance on your journey toward becoming a solutions architect through this book, you'll see these foundational principles and attributes applied in more depth, including in the crafting of various design patterns tailored to diverse scenarios and challenges.

In this chapter, you will learn the principles of designing architectures that are scalable, resilient, and optimized for performance, while ensuring robust security measures are in place to safeguard your application. You will explore strategies to navigate architectural constraints and embrace change through testing and automation, emphasizing a data-driven approach. By understanding and applying these principles, you'll be equipped to think critically and make informed decisions that enhance the effectiveness and reliability of your architectural designs.

You will learn about the following topics in this chapter:

- Building scalable architecture design
- Building a highly available and resilient architecture
- Design for performance
- Creating immutable architecture
- Think loose coupling
- Think service, not server
- Think data-driven design
- Adding security everywhere
- Making applications usable and accessible
- Building future-proof extendable architecture
- Ensuring architectural interoperability and portability

- Applying automation everywhere
- Design for operation
- Overcoming architectural constraints

Let's embark on exploring the foundational elements of architectural design. By the end of this chapter, you'll have gained insights into various essential design aspects to consider when constructing an architecture. This knowledge will serve as a crucial stepping stone on your journey to understanding and implementing effective and robust architectural solutions.

Building scalable architecture design

Scalability has always been a primary factor while designing a solution. If you ask any enterprise about their solutions, scalability will be one of the key considerations. **Scalability** refers to allowing your system to handle growing workloads, which can apply to multiple layers, such as the application server, web app, and database. Scalability helps you fulfill user demand without impacting application performance, resulting in higher business returns.

As most applications nowadays are web-based, let's also talk about **elasticity**. This is about growing your system by adding more capabilities and shrinking it to save on unnecessary costs. With the adoption of the public cloud, it has become easy to quickly grow and shrink your workload, with elasticity now replacing scalability.

Traditionally, there are two modes of scaling:

- **Horizontal scaling:** Horizontal scaling is becoming increasingly popular as computing power has become an exponentially cheaper commodity in the last decade. In horizontal scaling, the team adds more servers to handle increasing workloads, as in *Figure 2.1*:

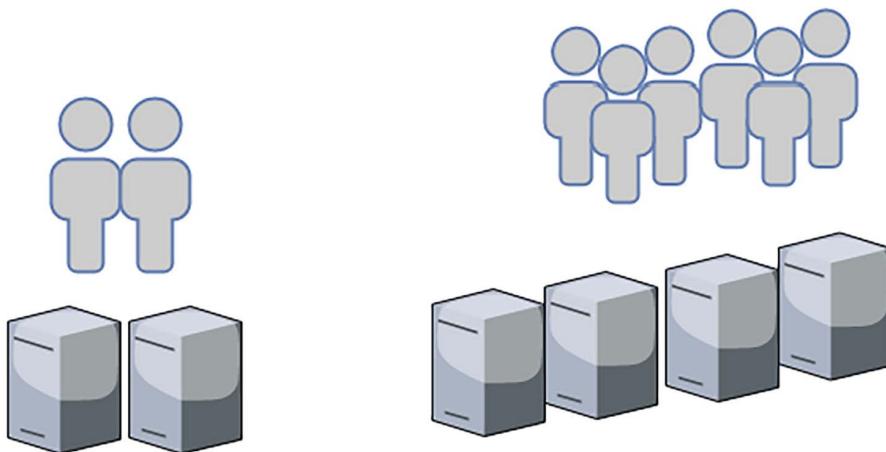


Figure 2.1: Horizontal scaling

Let's say your application can handle *1,000 requests per second* with *two server instances*, for example. As your user base grows, the application receives *2,000 requests per second*, which means you may want to double your application instances to four to handle the increased load.

- **Vertical scaling:** This has been around for a long time. It is a practice in which the team adds additional computer storage capacity and memory power to the same server to handle increasing workloads. As shown in *Figure 2.2*, during vertical scaling, you will get a larger server—rather than adding more servers—to handle the increased workload:

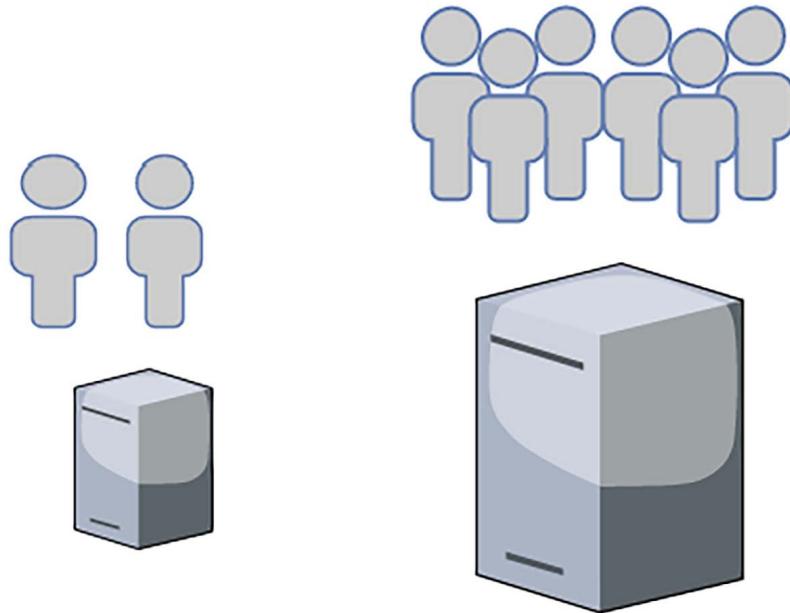


Figure 2.2: Vertical scaling

The vertical scaling model may be less cost-effective, however; when you purchase hardware with more computing power and memory capacity, the cost increases exponentially. You want to avoid vertical scaling after a certain threshold unless it is required to handle an increasing workload due to high costs and limitations on server capacity.

Vertical scaling is most commonly used to scale relational database servers. However, you need to think about database sharding here as if your server hits the limits of vertical scaling, it cannot grow beyond a specific memory and computing capacity.



Sharding is a technique used to scale databases by dividing and distributing data across multiple servers. Data is partitioned based on a shard key, which determines how data is distributed across the shards. In vertical sharding, the shard key can be a specific column or a set of columns in a table.

Scaling could be predictive if you are aware of your workload, which is often the case, or it could be reactive if you get a sudden spike or if you have never handled that kind of load before.

Predictive scaling is an advanced approach to managing application workloads, particularly useful for scenarios with predictable traffic patterns, like those seen on e-commerce websites. By analyzing historical data, organizations can forecast traffic trends and adjust their resources accordingly.

For instance, an e-commerce site might experience varying traffic based on the day of the week, time of day, or specific shopping holidays, necessitating a scaling strategy that preemptively adjusts resources to handle expected load increases. This approach not only optimizes resource usage but also enhances user experience by reducing latency and preventing outages, which is especially important during traffic spikes when resource allocation might lag behind demand.

Reactive scaling, on the other hand, is essential for dealing with unforeseen traffic surges, which might be significantly higher than the norm and could be triggered by events like flash sales. Understanding the unique traffic patterns for different pages of a website, as well as the user navigation path, is crucial for effectively managing these spikes. By identifying which pages can be cached or which queries are read-intensive, organizations can strategically offload traffic from the web layer, utilizing content distribution networks to manage static content.

This combination of predictive and reactive scaling ensures that applications remain resilient and responsive, regardless of traffic fluctuations. For example, the following **Auto Scaling group** has a maximum of six instances and a minimum size of three instances. During regular user traffic, three servers will be up and running to handle the workload, but the number of servers can reach six to handle a traffic spike. Your server fleet will increase based on the scaling policies you define to adjust the number of instances. For example, you can add one server when CPU utilization exceeds 60% in the existing servers' fleet but doesn't spin up more than six servers.

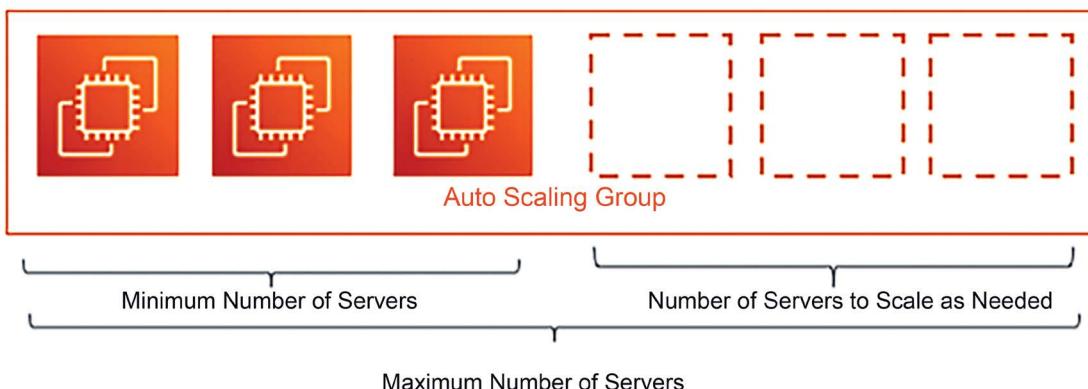


Figure 2.3: Server Auto Scaling

Regardless of scaling being reactive or predictive, you need to monitor the application and collect data to plan for your scaling needs.

Scaling static content

Static content, such as images and videos, plays a crucial role in engaging users on your website. However, if not managed properly, these elements can significantly slow down your application's performance. To maintain optimal speed and user experience, it's essential to scale and distribute static content efficiently.

Let's take the example of an e-commerce website. Each product will likely have multiple images—and perhaps even videos—to show a product's texture and demos, which means the website will have a significant amount of static content with a read-heavy workload since, most of the time, users will be browsing products. In addition, users may upload multiple images and videos for product review.

Storing static content in a web server means consuming lots of storage space, and as product listings grow, you have to worry about storage scalability. The other problem is that static content requires large file sizes, which may cause significant load latency on the user's end. The web architectural tier must utilize the **Content Distribution Network (CDN)** to solve this issue. CDNs help cache this content closer to the users, reducing latency and speeding up loading times. Properly scaling static content ensures that your application remains fast and responsive, providing a seamless experience for your users even as traffic volumes increase.

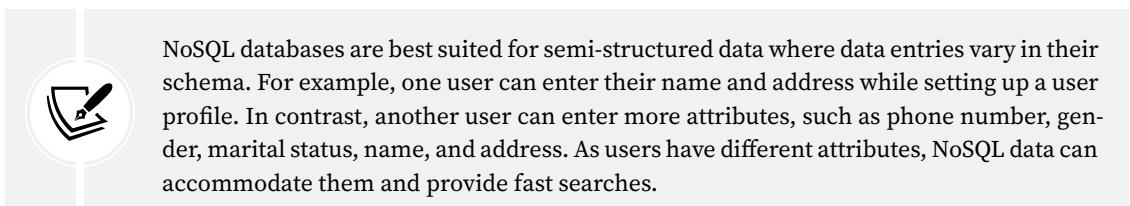
CDN providers (such as Akamai, Amazon CloudFront, Microsoft Azure CDN, and Google CDN) provide locations across the globe where static content can be cached from the web server near the user's location, reducing latency. *Chapter 4, Solution Architecture Design Patterns*, will teach you more about caching.

To scale static content storage, it is recommended to use object storage, such as Amazon S3, or an on-premises custom origin, which can grow independently of memory and computer capabilities. Additionally, scaling storage independently with popular object storage services saves on cost. These storage solutions can hold static HTML pages to reduce the load of web servers and enhance the user experience by reducing latency through the CDN.

Session management for application server scaling

The application architectural tier collects user requests from the web tier and performs the heavy lifting of calculating business logic and talking to the database. When user requests increase, the application tier needs to scale to handle them and then shrink back as demands decrease. In such scenarios, users are tied to the session, where, for example, they may browse from their mobile and purchase from their desktop. Performing horizontal scaling without handling user sessions may cause a bad user experience, as it will reset the user's shopping progress.

Here, the first step is to take care of user sessions by decoupling them from the application server instance, which means you should consider maintaining the user session in an independent layer, such as a NoSQL database where you can store semi-structured data.



NoSQL databases are best suited for semi-structured data where data entries vary in their schema. For example, one user can enter their name and address while setting up a user profile. In contrast, another user can enter more attributes, such as phone number, gender, marital status, name, and address. As users have different attributes, NoSQL data can accommodate them and provide fast searches.

NoSQL databases such as Amazon DynamoDB or MongoDB offer exceptional partitioning capabilities, enabling effortless horizontal scaling at levels that surpass the scalability of other database types.

Once you start storing your user session in NoSQL databases, your instance can scale horizontally without impacting the user experience. You can add a load balancer in front of a fleet of application servers, which can distribute the load among instances; with the help of autoscaling, you can automate adding or removing instances on demand.

Database scaling

Most applications use relational databases to store their transactional data. These have been around for decades and provide the robust transactional consistency required for many applications. However, the main problem with relational databases is that they cannot scale horizontally until you plan for other techniques, such as sharding, and modify your application accordingly. This will be a lot of work.

For databases, taking preventive care and reducing their load is better. Using a combination of storage methods, such as storing user sessions in separate NoSQL databases, storing static content in an object store, and applying an external cache, helps to offload the master database. It's better to keep the master database node for writing and updating data and use an additional read replica for all read requests. For example, Amazon RDS for MySQL provides up to 15 read replicas for relational databases. Read replicas may have milliseconds of delay while syncing with the leader node, and you need to plan for that while designing your application. Using a caching engine such as Memcached or Redis is recommended to cache frequent queries and thus reduce the load on the leader node.

If your database starts growing beyond its current capacity, you need to redesign and divide it into shards by applying partitions.

Each shard can grow independently, and the application needs to determine a partition key to store user data in a respective shard. For example, if the partition key is `user_name`, then usernames from A to E can be stored in one shard, names from F to I can be stored in a second shard, and so on. The application needs to direct user records to the correct partition per their name's first letter.

So, as you can see, scalability is a significant factor while designing a solution architecture, and it can impact the overall project budget and user experience significantly if it's not planned correctly. A solutions architect always needs to consider elasticity while designing applications and optimizing workloads for the best performance and least cost.

A solutions architect needs to evaluate different options, such as CDNs for static content scaling and load balancing, autoscaling options for server scaling, and various data storage options for caching, object stores, NoSQL stores, read replicas, and sharding.

Building elastic architecture

While focusing on scalability to enhance application performance, it is crucial to construct a cost-aware architectural design. This implies that as you expand your server infrastructure to meet rising user demand, the system should also contract when the server load diminishes. Elasticity is required to right-size your architecture, which involves scaling your server infrastructure to match the current demand accurately. It's a balancing act between ensuring there's enough capacity to handle peak loads efficiently without over-provisioning resources that would lie idle during off-peak times.

Let's continue with the e-commerce website example by considering a modern three-tier architecture and see how we can achieve elasticity at a different Application Layer. Here, we only target the elasticity and scalability aspects of architecture design. *Figure 2.4* shows a three-tier architecture diagram of the AWS cloud tech stack:

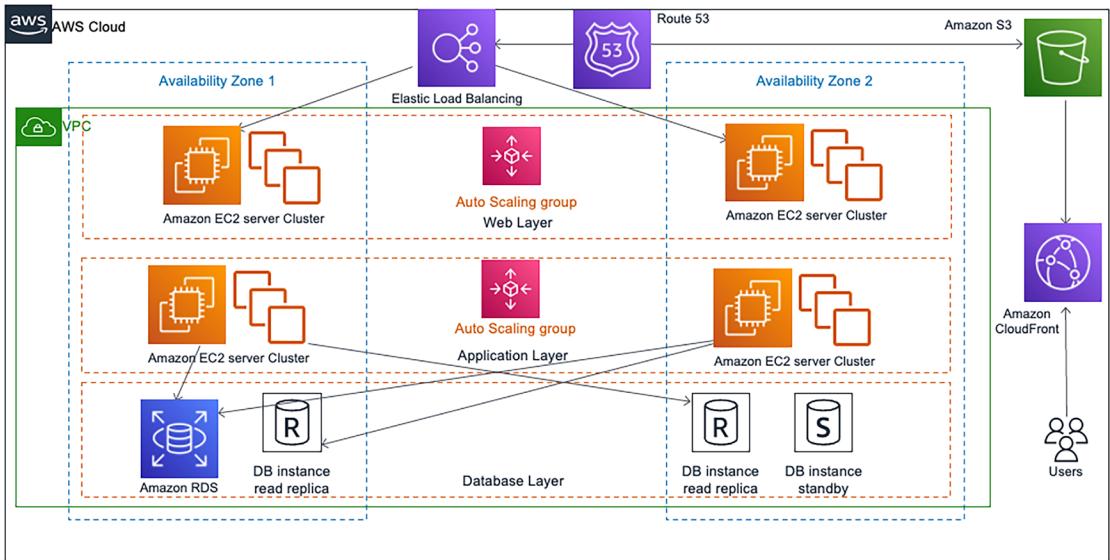


Figure 2.4: Scaling three-tier architecture

This diagram depicts a three-tier architecture designed for elasticity and high availability, focusing on building an elastic server fleet to manage variable loads efficiently.

The following are the architectural components:

- Elastic Load Balancing automatically distributes incoming application traffic across multiple targets, such as Amazon **Elastic Compute Cloud (EC2)** instances, containers, IP addresses, etc., in multiple Availability Zones. This increases the fault tolerance of the e-commerce application.
- The web layer consists of an Auto scaling group of EC2 instances designed to serve the application's dynamic content. This fleet can automatically scale out (add instances) or scale in (remove instances) based on defined criteria like CPU utilization, ensuring that the fleet can adapt to the incoming traffic and maintain consistent performance.
- The Application Layer also has an autoscaling group of EC2 instances responsible for executing the business logic of the application. Like the web layer, this layer can dynamically adjust its size to meet the demands of the application workload.
- At the bottom, the Database Layer includes Amazon **Relational Database System (RDS)** instances, which provide managed relational databases. The setup includes a primary DB instance and a read replica to handle read-heavy operations, improving performance and reducing load on the primary instance. There is also a standby instance in a different Availability Zone for high availability and failover support.

This architecture allows for a flexible, scalable application environment that can handle variable workloads with high availability across multiple zones. It's designed to expand and contract automatically in response to the application's needs, ensuring users experience consistent, responsive performance.

When users access and interact to the application using website or mobile application. Their requests are routed through Amazon Route 53, which is a highly available and scalable **Domain Name System (DNS)** web service. Amazon CloudFront, a CDN, is used to distribute static content like images, stylesheets, and JavaScript files efficiently. This reduces the load on the web servers and improves the user experience by decreasing latency.

In this section, you have discovered the various scaling methods and how to inject elasticity into the different layers of your architecture. Scalability is an essential factor in ensuring that there is high application availability to make your application resilient. We will learn more about high availability and resiliency in the next section.

Building a highly available and resilient architecture

Creating a highly available and resilient architecture involves designing systems that can tolerate failures of individual components without disrupting the overall system functionality.

Highly available architecture

The one thing an organization wants to avoid is *downtime*. Application downtime can cause a loss of business and user trust, making **high availability** a primary factor while designing the solution architecture. The principle of high availability is “design for failure, and nothing will fail.”

The requirement for application uptime varies from application to application. If you have an external-facing application with a large user base, such as an e-commerce website or social media platform, 100% uptime becomes critical. In the case of an internal application (accessed by an employee, such as an HR system or a company intranet), it can likely tolerate some downtime. Achieving high availability is directly associated with cost, so a solutions architect must always plan for high availability per the application requirements to avoid over-architecting.

To achieve a high-availability architecture, it is better to plan workloads in an isolated physical location so that, should an outage occur in one place, your application replica can operate from another location. The high-availability architecture goes hand in hand with self-healing, where you can ensure your application is up and running, but you also need a quick recovery to maintain the desired user experience.

Resilient architecture

A resilient architecture means your application should be available for customers while recovering from failure. Making your architecture resilient includes applying best practices to recover your application from increased loads due to more user requests, malicious attacks, and architectural component failure. Resiliency needs to be used in all architectural layers, including infrastructure, application, database, security, and networking. A resilient architecture should recover from failure within a desired amount of time.

To make your architecture resilient, you need to define the time of recovery and address the following points:

- Identify and implement redundant architectural components wherever required.
- Understand when to fix versus when to replace architectural components. For example, fixing a server issue might take longer than replacing it with the same machine image.

Achieving redundancy

Redundancy is a crucial aspect of building resilient systems. Building a resilient architecture requires a multi-layered redundancy strategy. It involves deploying server clusters across different racks within a single data center, extending to multiple data centers within the same region, and further, across various geographic regions. This geographical distribution ensures protection against localized and regional disasters and reduces latency for a global user base.

Incorporating intelligent load balancing and global traffic management, such as DNS-based routing with health checks, ensures that users are always served from the optimal location. Database resiliency is achieved through strategic replication, with automated failover mechanisms to maintain database availability and integrity.

If servers are spread across different physical locations, the first level of traffic routing can be handled using the DNS server before it reaches the load balancer. This way, in the case of an entire region failure, your application can still run.

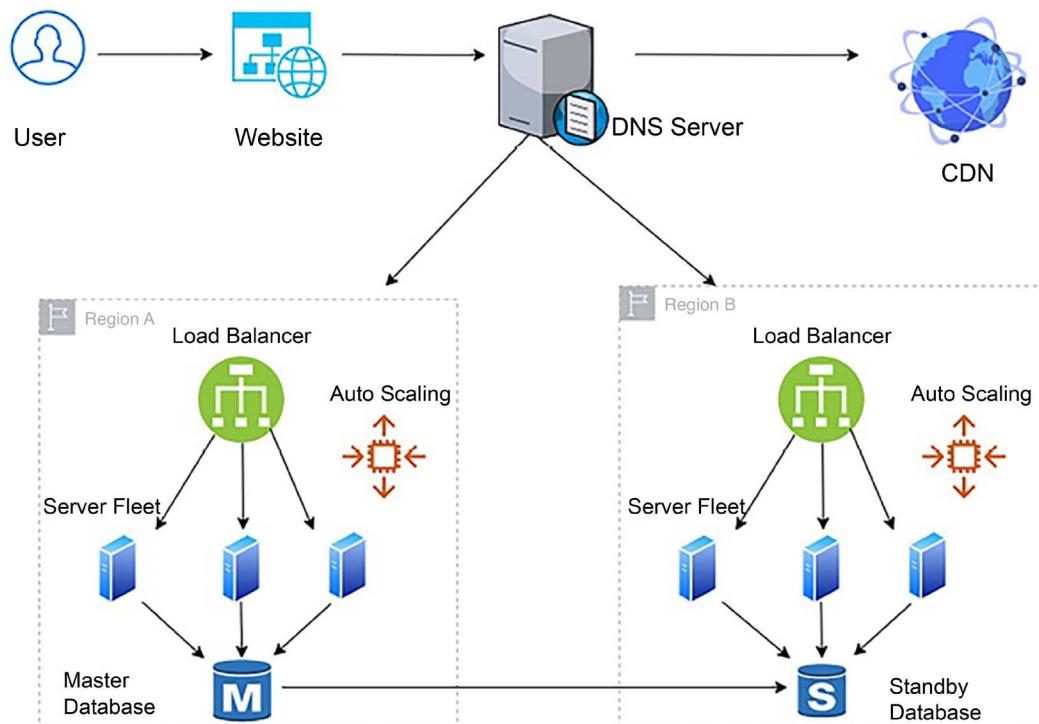


Figure 2.5: Application architecture resiliency using a DNS server

As you can see in the preceding architecture, resiliency must be applied in all the critical layers that affect the application's availability in order to implement a design that can sustain failure. To achieve resiliency, in addition to using a DNS server to route traffic between different physical locations, the following best practices need to be applied to create a redundant environment:

- Use the CDN to distribute and cache static content such as videos, images, and static web pages near the user's location so that your application will still be available.
- Once traffic reaches a region, use a load balancer to route traffic to a fleet of servers so that your application can still run even if one location fails within your region.
- Use autoscaling to add or remove servers based on user demand. As a result, your application should not be impacted by individual server failures.
- Create a standby database to ensure the high availability of the database, meaning that your application should be available in the event of a database failure.

Addressing component failure

If any components fail, you should have a backup to recover them and achieve architecture resiliency. The load balancer and routers at the DNS server perform a health check to ensure that the traffic is only routed to healthy application instances. You can configure this to perform a shallow health check, which monitors local host failures, or a deep health check, which can also take care of dependency failure. However, a deep health check takes more time and is more resource-intensive than a shallow health check. You will learn more about resilient architecture in *Chapter 8, Architectural Reliability Considerations*.

At the application level, it is essential to avoid **cascade failure**, where the failure of one component can bring down the entire system. To mitigate the risk of cascading failures in a system, various mechanisms can be employed:

- **Timeouts:** Setting a maximum time limit for operations and requests can prevent waiting indefinitely for a response, which can lead to resource exhaustion.
- **Traffic rejection:** When the system is overwhelmed, it can proactively reject new requests to prevent overload and maintain stability for existing processes.
- **Idempotent operations:** Ensuring that operations can be repeated without causing unintended effects can help recover from intermediate failures without duplicating actions or causing inconsistencies.
- **Circuit breakers:** Implementing a circuit breaker pattern can detect failure patterns and open the “circuit” to stop further requests to a failing service, allowing it to recover and preventing the failure from spreading to other parts of the system.

By incorporating these strategies, systems can become more resilient, maintaining functionality in the face of individual component failures and preventing those failures from escalating into widespread system outages.

While high availability and resiliency ensure your system is up and available for users, it is also essential to maintain performance where fault tolerance comes into play. Let us now turn to the subject of fault tolerance.

Making your architecture fault-tolerant

High availability means your application is available to the user but it can result in degraded performance. Suppose you need four servers to handle users' traffic. For this, you put two servers in two different physically isolated data centers. If there is an outage in one data center, user traffic can be served from another. But now you have only two servers, which means only 50% of the original capacity is available, and users may experience performance issues. In this scenario, your application has 100% high availability but is only 50% fault tolerant.

As shown in *Figure 2.6*, to achieve 100% fault tolerance, you need full redundancy and have to maintain the double count of the servers so that the user doesn't encounter any performance issues during the outage of one zone.

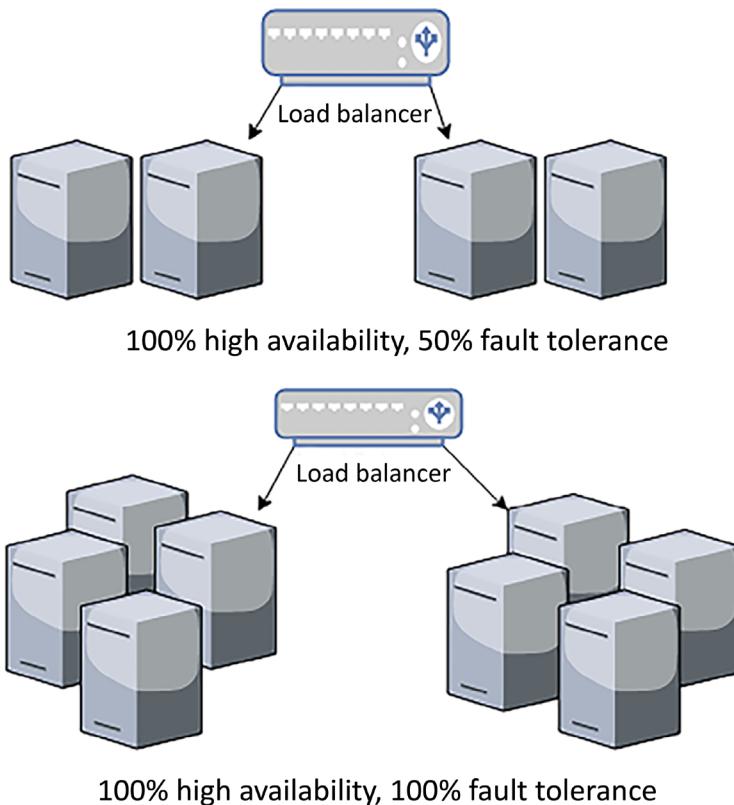


Figure 2.6: Fault tolerance architecture

Fault tolerance is handling workload capacity if an outage occurs without compromising system performance. A fully fault-tolerant architecture involves high costs due to increased redundancy. Whether your user base can live with degraded performance for the period of application recovery depends on your application's criticality.

While designing the application architecture, a solutions architect needs to determine the nature of the application's users and whether a fault tolerance of 100% is required, which will inevitably come with a cost implication. For example, an e-commerce website may need 100% fault tolerance, as degraded performance directly impacts business revenue. At the same time, an internal payroll system, which employees use at the end of the month to check their salary slips, can tolerate reducing performance for a short period. Let's dive deep into building high-performance architecture next.

Designing for performance

With the availability of fast internet, customers are seeking high-performance applications with minimal load time. Organizations have noticed that a direct revenue impact is proportional to application performance, and slowness in application load time can significantly impact customer engagement. Modern companies are setting high expectations when it comes to performance, resulting in high-performance applications becoming necessary to stay relevant in the market.

Like resiliency, the solutions architect needs to consider performance at every layer of architecture design. The DevOps team needs to put monitoring in place to check that the solution continues to perform effectively and work to improve it continuously. Better performance means increased user engagement and return on investment.

High-performance applications are designed to handle application slowness due to external factors such as a slow internet connection. For example, you may have designed your blog web page to load within 500 milliseconds where there is good internet availability. However, where the internet is slow, you can have it load text first and engage the user with this content while images and videos are still loading.

In an ideal environment, as your application workload increases, automated scaling mechanisms start handling additional requests without impacting application performance. But in the real world, your application latency goes down for a short duration when scaling takes effect. To understand how it will behave in a real-world situation, it's better to test your application for performance by increasing the load and understanding whether you can achieve the desired concurrency and user experience.

You need to choose the right kind of server at the server level, depending on your workload. For example, choose the right amount of memory and compute to handle the workload, as memory congestion can slow down application performance and, eventually, the server may crash. You should choose the correct **input/output operations per second (IOPS)** for storage. You need high IOPS for write-intensive applications to reduce latency and increase disk write speed.



IOPS is a performance measurement used to benchmark the speed at which storage devices like hard drives, solid-state drives, and storage area networks can read and write data. Each input or output operation could be a data read or a data write.

To achieve higher performance, apply caching at every layer of your architecture design. Caching makes your data locally available to users or keeps data in memory to serve an ultra-fast response.

The following are considerations for adding caching to various layers of your application design:

- Use the browser cache on the user's system to load frequently requested web pages.
- Use the DNS cache for quick website lookup.
- Use the CDN cache for high-resolution images and videos that are near the user's location.
- At the server level, maximize the memory cache to serve user requests.
- Use cache engines such as Redis and Memcached to serve frequent queries from the caching engine.
- Use the database cache to serve frequent queries from memory.
- Take care of cache expiration, which is the process by which data stored in the cache becomes outdated and is marked for update or removal. Cache eviction, on the other hand, is the process by which data is removed from the cache, typically to make room for new data.

As you can see, keeping your application performant is an essential design aspect and is directly related to organizational profitability. The solutions architect needs to think about performance when creating a solution design and should work relentlessly to keep improving the performance of the application. In *Chapter 6, Performance Considerations*, you will dive deeper into this and learn techniques to optimize your application for better performance.

Creating immutable architecture

Organizations make a significant capital investment in hardware and develop the practice of regularly refreshing it with a new version of the application and configuration. Over time, this can lead to different servers running in varied configurations, and troubleshooting them becomes tedious. Sometimes organizations must keep running unnecessary resources when they are not needed, as they are unsure which server to shut down, which may cause application failure. The inability to replace servers makes rolling out and testing any new updates in your server fleet challenging. These problems can be solved by treating your server as a replaceable resource, which enables you to move more quickly to accommodate changes, such as upgrading applications and underlying software, resulting in less downtime and fixing application issues quickly. That is why, while designing your application, you should always think of immutable infrastructure. This means that, during application upgrades, you will not only replace software but hardware, too.

Adopting a mindset where servers are treated like cattle, not pets, is fundamental in modern cloud architecture. This approach means that individual servers are not meticulously maintained or customized to the point where they become irreplaceable. Instead, servers are designed to be provisioned quickly, managed consistently, and disposed of or replaced without significant impact on the overall system. This methodology enhances scalability and resiliency, as it allows for rapid adaptation to changes in demand or recovery from failures.

To create replaceable servers, making your application stateless to retain user experience and avoiding the hardcoding of any server IP or database DNS name is advisable to avoid failures during replacement. You need to apply the idea of treating your infrastructure as code instead of hardware, and not apply updates to the live system.

Creating immutable infrastructure becomes more viable with a virtual machine. You can create a golden image of your virtual machine and deploy it with the new version of your infrastructure rather than trying to update an existing version. You should always spin up new server instances from the golden machine image, which acts as a template and has all the necessary security and software in place. This deployment strategy is also beneficial for server troubleshooting, where you can dispose of the server with an issue and spin up a new server from a golden image.

You should take a backup of logs for root cause analysis before disposing of the server with issues. This approach also ensures consistency across the environment, as you use the same baseline server image to create all your environments.

Loose coupling is another critical design principle that complements the “cattle not pets” approach. It involves designing system components so they interact with each other through well-defined interfaces and are independent enough that changes in one component do not necessitate changes in others. This separation enhances flexibility and scalability, allowing individual components to evolve, scale, or recover from failures independently. Let’s learn more about loose coupling.

Think loose coupling

A traditional application is deployed on a tightly integrated server fleet where each server has a specific responsibility. Often, applications depend on multiple servers for completeness of functionality.

As shown in the following diagram, in a tightly coupled architecture, the web server fleet has a direct dependency on all application servers, and vice versa:

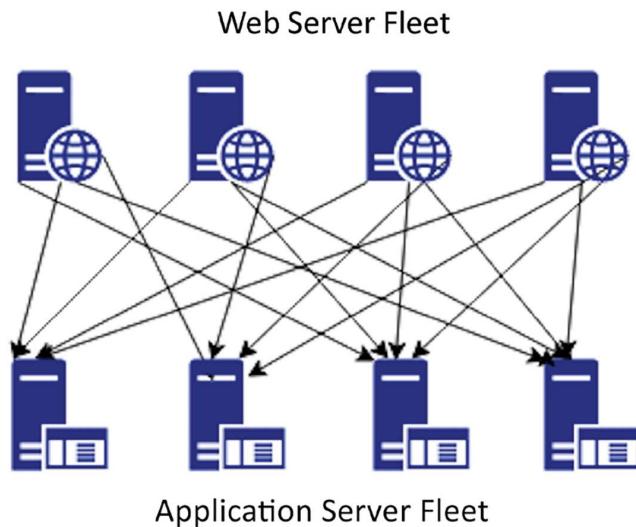


Figure 2.7: Tightly coupled architecture

In the preceding architecture diagram, if one application server goes down, then all web servers will start receiving errors, as the request will route to an unhealthy application server, which may cause a complete system failure. With tightly coupled architecture, if you want to scale by adding and removing servers, it requires lots of work, as all connections need to be set up appropriately.

With loose coupling, you can add an intermediate layer, such as a load balancer or a queue, which automatically handles failures or scaling for you.

In the following architecture diagram, there is a load balancer between the web server and the application server fleet, which makes sure that user requests are always served from a healthy application server:

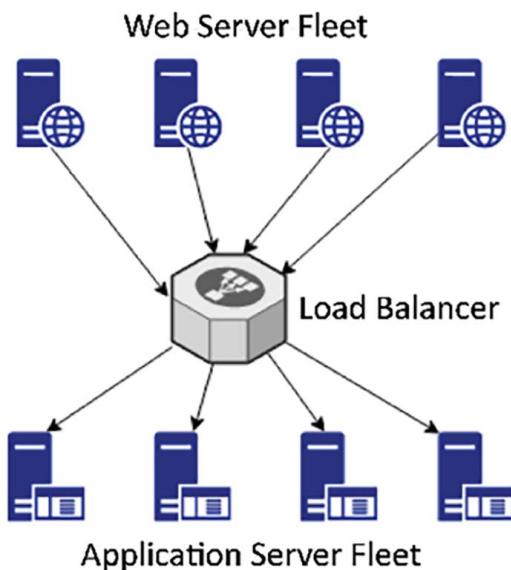


Figure 2.8: Load balancer-based, loosely coupled architecture

If one of the application servers goes down, the load balancer will automatically start directing all the traffic to the other three healthy servers. Loosely coupled architecture also helps you to scale your servers independently and replace unhealthy instances gracefully. It makes your application more fault-tolerant as an error radius is limited to a single instance only.

Loosely coupled architecture can also be queue-based; take the example of an image-processing website, where you need to store an image and then process it for encoding, thumbnails, and copyright. The following architecture diagram has queue-based decoupling. With this, loose coupling of systems is achieved by using queues between systems and exchanging messages that transfer jobs through these.

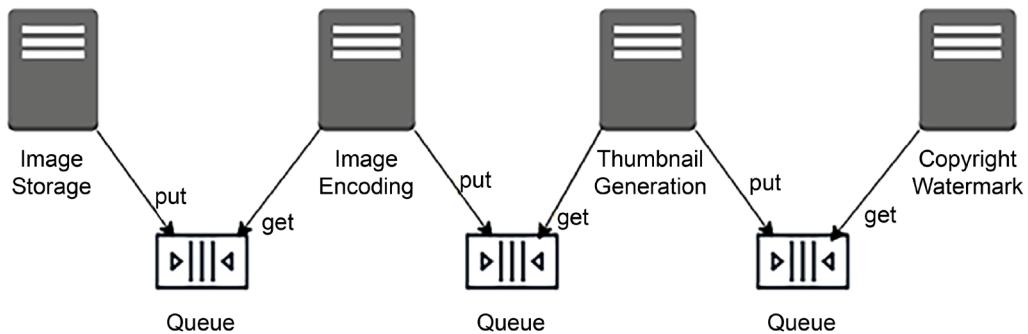


Figure 2.9: Queue-based, loosely coupled architecture

Queue-based decoupling enables the asynchronous linking of systems, where one server is not waiting for a response from another server and is working independently. This method lets you increase the number of virtual servers that receive and process the messages in parallel. You can configure autoscaling to terminate the excess servers if, for example, there is no image to process.

In a complex system, a loosely coupled architecture is achieved by creating a microservice architecture where independent services contain a complete set of functionalities and communicate with each other over a standard protocol. In modern design, event-driven design like this is becoming highly popular, facilitating the decoupling of an application component. The loosely coupled design has many benefits, from scalability and high availability to ease of integration.

Think service, not server

In the previous section, you learned about loose coupling and how important it is for our architecture to be loosely coupled for scalability and fault tolerance. Developing service-oriented thinking will help to achieve a loosely coupled architecture (as opposed to server-oriented thinking, which can lead to hardware dependency and a tightly coupled architecture). Microservice-based event-driven architecture helps us to achieve ease of deployment and maintenance for our solution design.

In a RESTful architecture, you can format a message in XML, JSON, or plain text and send it over Internet using a simple HTTP protocol. RESTful architecture is popular as it is very lightweight. Microservices are based on RESTful architecture and are independently scalable, which makes it easier to expand or shrink one component of your application without impacting others.

As you can see in the following diagram, in a monolithic architecture, all components are built into a single service, thus deployed in a single server and tied up with a single database, which creates a hard dependency. In contrast, in a microservice architecture, each component is independent, with its framework and database, which allows them to be scaled independently:

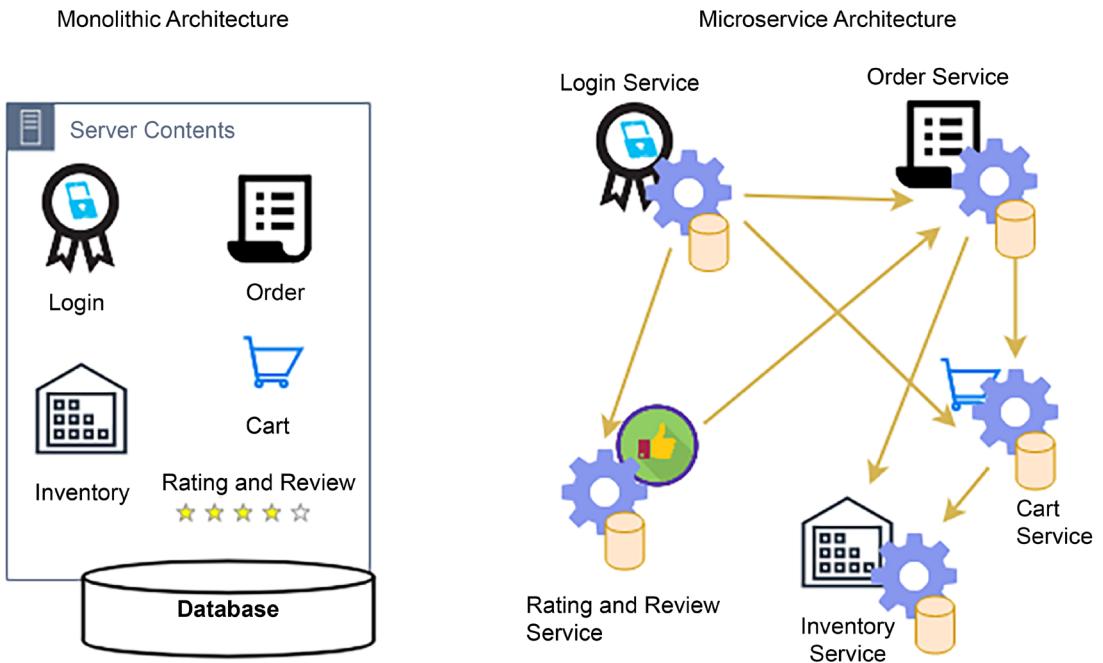


Figure 2.10: Monolithic and microservice architectures

In the preceding diagram, you can see an example of an e-commerce website as both a monolithic architecture and a microservice architecture, where customers can log in and place an order, assuming the items they want are available, by adding items to the cart.

To convert a monolithic architecture into a microservice-based architecture, you can create applications made of small, independent components, which constitute smaller parts to iterate.

The *modularization approach* reduces the *cost, size, and risk of change*. In the preceding case, each component is created as a service. Here, the **Login** service can independently scale to handle more traffic, as the customer may log in frequently to explore the product catalog and order status. In comparison, the **Order** and **Cart** services may have less traffic, as a customer may not place an order very often.

Solutions architects need to think of microservices while designing a solution. The clear advantage of services is that you have a smaller surface area of code to maintain, and services are self-contained. However, monitoring microservices involves a more granular approach compared to traditional monolithic applications due to the distributed nature of microservices. Each microservice functions independently, which means monitoring must be implemented at the individual service level as well as the system level to ensure a comprehensive view of the application's health and performance.

You can build microservices with no external dependencies. All prerequisites are included in the service, which enables loose coupling and scaling, and reduces the blast radius in case of failure.

Any application design revolves around data, and working backward from data helps to build the best architecture. Let's learn more about data-driven design.

Think data-driven design

Any software solution revolves around the collection and management of data. Take the example of an e-commerce website; the software application is built to showcase product data on the website and encourage the customers to buy the products. It starts by collecting customer data when they create a login, adding a payment method, storing order transactions, and maintaining inventory data as the product gets sold. Another example is a banking application, which stores customer financial information and handles all financial transaction data with integrity and consistency. The most important thing for any application is handling, storing, and securing data appropriately. Data heavily influences solution design, and by keeping data in mind, you can apply the right design-driven solution for your needs.

It's not just application design that revolves around data but operational maintenance and business decisions, too. You need to add monitoring capabilities to ensure that your application and your business are running without any issues. For example, for application monitoring, you collect log data from the server and create a dashboard to visualize the metrics. Continuous data monitoring and sending alerts in the case of issues help you to recover quickly from failure by triggering the auto-healing mechanism.

As a solutions architect, you are thinking about application design and the overall business value proposition, including ways to collect data and utilize it around the application, which can help to increase customer satisfaction and maximize the return on your investment. Data is gold, and getting insights into data can significantly impact an organization's profitability.

Adding security everywhere

Security is one of the essential aspects of solution design; any gap in security can have a devastating effect on a business or organization's future. Many organizations are compromised by security breaches, resulting in a loss of customer trust and damaging businesses' reputations. Industry-standard regulations, such as **PCI (Payment Card Industry)**, **HIPAA (Health Insurance Portability and Accountability Act)**, **GDPR (General Data Protection Regulation)**, and **SOC (System and Organization Controls)** compliance are pivotal frameworks securing data across distinct domains. PCI secures credit card information in finance, HIPAA safeguards patient data in healthcare, GDPR enhances data privacy in the EU, and SOC ensures data management security in service organizations, enforce security safeguards to protect consumer data while providing standard guidance to the organization. Depending on your industry and region, you must comply with local legislation by adhering to compliance needs such as these.

Security can significantly impact solution design, so you need to understand your security needs before starting your design. Security needs to be included in platform readiness at the hardware level and in application development at the software level.

The following are the security aspects that need to be considered during the design phase:

- **Physical security of data center:** All IT resources in data centers should be secure from unauthorized access.
- **Network security:** The network should be secure to prevent any unauthorized server access.

- **Identity and Access Management (IAM):** Only authenticated users should have access to the application, and they can do the activity as per their authorization.
- **Data security in transit:** Data should be secure while traveling over the network or the internet.
- **Data security at rest:** Data should be secure while stored in the database or any other storage.
- **Security monitoring:** Any security incident should be captured, and the team should be alerted to act.

Application design needs to balance security requirements, such as encryption, and other factors, such as performance and latency. Data encryption always has a performance impact as it adds a layer of additional processing because data needs to be decrypted to be utilized. Your application needs to accommodate the overhead of additional encryption processing without impacting overall performance, so consider use cases where encryption is required while designing your application. For example, if the data is confidential, you need to encrypt it.

The other aspect of application design to consider related to security is regulatory compliance for adherence to local law. Compliance is essential if your application belongs to a regulated industry such as healthcare, finance, or the federal government. Each type of compliance has its requirements, which commonly include data protection and recording each activity for audit purposes. Your application design should include comprehensive logging and monitoring, which will fulfill the audit requirement.

Security is one of the most important aspects of application resiliency. From a security perspective, a **Distributed Denial of Service (DDoS)** attack can potentially impact the availability of services and applications. A DDoS attack usually puts fake traffic in your server and makes it busy, meaning legitimate users cannot access your application. This can happen at the network layer or the Application Layer. It's essential to take a proactive approach to prevent DDoS attacks. Keep as much of the application workload as possible in the private network and avoid exposing your application endpoints to the internet wherever possible.

Security automation is another factor you should consistently implement along with your design to reduce and mitigate any security incidents. Automation in security involves utilizing technology to perform security tasks without human intervention, streamlining the detection, analysis, and remediation of security incidents. By integrating automated security measures, you can achieve continuous monitoring and real-time threat detection, enabling quicker responses to vulnerabilities and breaches.

In this section, you have learned how to apply security thinking while designing and considering any regulatory needs. However, you have a high-level overview here. You will learn more details in *Chapter 7, Security Considerations*.

You might create a feature-rich product, but it may not appeal broadly to users until they find it easy to navigate and access. Your application's usability and accessibility play a significant role in product success. Let's learn more about this in the next section.

Making applications usable and accessible

Ensuring applications are both usable and accessible is a critical aspect of design that impacts the user experience significantly. Usability refers to how easy and intuitive an application is for users to interact with, which involves a user-friendly interface, clear navigation, and efficient task-completion processes. Accessibility, on the other hand, ensures that applications are usable by people with various disabilities. Let's learn more about these.

Achieving usability

You want your users to have a seamless experience browsing through the application. It should be so smooth that they don't even notice how easily they can find things without any difficulties. You can do this by making your application highly usable.

Usability is how quickly the user can learn navigation logic when using your application for the first time. It's about how quickly they can bounce back if they make a mistake and whether they can perform the task efficiently. Complex and feature-rich applications have no meaning if they can't be used effectively. The goal is to create an intuitive and user-friendly interface that enhances the user experience, ensuring that the application's functionality is accessible and straightforward for all users.

User research and testing are essential to defining usability that can satisfy user experience.

Achieving accessibility

When designing your application, you often want to target a global audience or significant geographic region. Your user base will be diverse regarding technical amenities and physical abilities. Accessibility is about inclusion; you want your application to be accessible to everyone, regardless of whether a user has a slow internet connection, uses an old device, or has physical limitations.

While designing an application, a solutions architect must ensure that accessibility is considered. Sometimes, a different version of the application may need to be created altogether to achieve that.

Accessibility design should include design components, such as voice recognition and voice-based navigation, screen magnifiers, and the ability to read content aloud to aid those who cannot easily access and use applications due, for example, to vision or hearing impairment.

Localization helps the application become available in a language that's specific to a region (for example, Spanish, Mandarin, German, Hindi, or Japanese), enabling global users across the world to navigate your application in their local languages.

As shown in *Figure 2.11*, customer satisfaction is a key component of usability and accessibility.

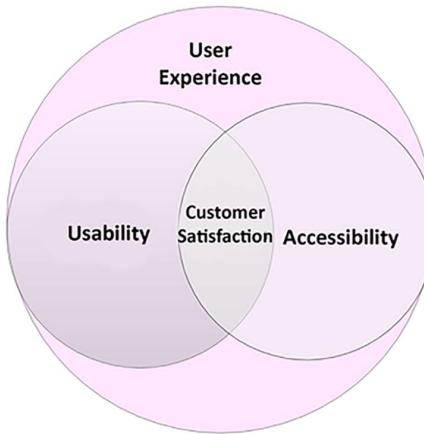


Figure 2.11: Customer satisfaction with usability and accessibility

You must know your users to achieve usability and accessibility—where accessibility is a component of usability—as they go hand in hand. Before starting the solution design process, a solutions architect should work alongside a product owner to research users by conducting interviews and surveys and gathering feedback on the mock frontend design. You need to understand the users' limitations and empower them with supporting features during application development.

When the product is launched, the team should plan for A/B testing by routing a small portion of user traffic to new features and understanding user reactions. A/B testing involves comparing two versions of an application to evaluate their performance and determine the superior option. After launch, the application must have a mechanism to collect continuous feedback (by providing a feedback form or by launching customer support) to make the design better.

As your users keep evolving, your architecture should be able to keep up with increasing demands. For that, you need to design extendable and future-proof architecture. Let's learn how to make your architecture future-proof.

Building future-proof extendable and reusable architecture

Businesses evolve as they grow; applications scale to handle an increased user base and add more features to stay ahead and attain a competitive edge. A solution design needs to be extendable and flexible enough to modify an existing feature or add new functionality.

To achieve solution extensibility, a solutions architect must use a loosely coupled architecture wherever possible. At a high level, creating a RESTful or queue-based architecture can help develop loosely coupled communication between different modules or across applications. You will learn more about the other kinds of architecture in *Chapter 4, Solution Architecture Design Patterns*. In this section, we will take a simple example to explain the concept of architectural flexibility.

To modularize their application, organizations often want to build a platform with a group of features and launch them as separate applications. This is only possible with a reusable design.

Figure 2.12 shows an API-based architecture in an e-commerce application. Here, you have independent services, such as product catalog, order, payment, and shipping, utilized by an end user application in a pick-and-choose manner. The customer uses mobile and browser applications to place an online order. These applications need a product catalog service to allow the customer to browse the products on the web, an order service to allow them to place an order, and a payment service to process a payment.

In turn, the product catalog and order service communicate with the shipping service to send ordered items to the customer's doorstep. On the other hand, brick-and-mortar stores use point-of-sale systems, where a customer representative scans barcodes, places orders on behalf of the customer, and takes payment. No shipping service is required here, as the customer picks up the item in-store.

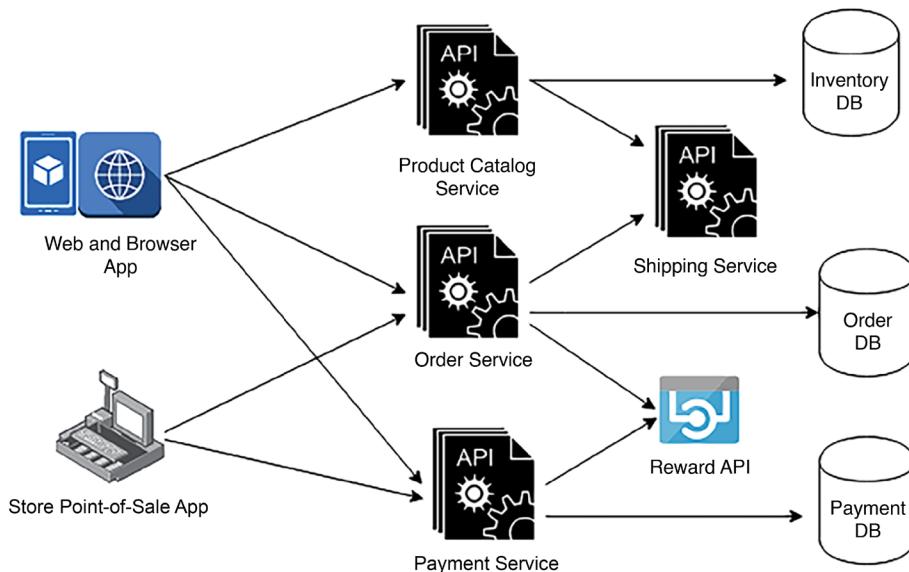


Figure 2.12: Extensible API-based architecture

In *Figure 2.12*, you can see the Reward API used for third-party API integration. This architecture allows you to extend the current design to integrate the Reward API for customer retention and to attract new customers by providing benefits when they purchase an item. Here, you can see how payment services are reutilized by both online and store ordering. Another service can reuse payment service if the organization wants to take payments for a gift card service, food services, and so on.

Extensibility and reusability are not limited to the service design level—they go deep into the actual API framework level, where software architects should use **object-oriented analysis and design (OOAD)** concepts, such as inheritance, to create an API framework. This can be extended and reutilized to add more features to the same service.



OOAD is a foundational approach in software engineering that helps developers plan and model applications more effectively, ensuring that the software is modular, scalable, and maintainable.

In order to extend your application feature, it needs to work seamlessly with other products where it can extend data and transactions. Making your application interoperable to the ecosystem helps to add new features by utilizing other adjacent applications. Let's learn more about making compatible architecture.

Ensuring architectural interoperability and portability

Architectural interoperability and portability are crucial aspects of modern software architecture, ensuring that applications can work across different environments and interact seamlessly with other systems. Let's look at these concepts.

Making applications interoperable

Interoperability is the ability of one application to work with others through a standard format or protocol. Often, an application must communicate with various upstream systems to consume data and downstream systems to supply data, so it is essential to establish that communication seamlessly.

An e-commerce application, for example, needs to work with other applications in the supply chain management ecosystem. This includes enterprise resource planning applications to record all transactions, transportation life cycle management, shipping companies, order management, warehouse management, and labor management.

All applications should be able to exchange data seamlessly to achieve an end-to-end feature from customer order to delivery. You will encounter similar use cases everywhere, whether it is a healthcare application, manufacturing application, or telecom application.

A solutions architect needs to consider application interoperability during design by identifying and working with various system dependencies. An interoperable application saves a lot in terms of cost, as it depends on systems that can communicate in the same format without any data messaging effort.

Each industry has its standard data exchange size that needs to be understood and adhered to. In general, for software design, the architect may choose a popular format, such as JSON or XML, for different applications so that they can communicate with each other. Both formats are supported out of the box in modern RESTful API design and microservice architecture.

Making applications portable

System portability allows your application to work across different environments without or with only minimal changes. Any software application must work across various operating systems and hardware to achieve higher usability.

Since technology changes rapidly, you will often see that a new version of a software language, development platform, or operating system is released and you will need to ensure your application can adapt to these changes. Today, mobile applications are an integral part of any system design, and your mobile apps need to be compatible with major mobile operating system platforms, such as iOS and Android.

During the design phase, the solutions architect needs to choose a technology that can achieve the desired portability of the application. For example, if you aim to deploy your application across different operating systems, programming languages such as Java may be a good choice, as all operating systems often support it, and your application will work on a different platform without needing to be ported across. For mobile applications, an architect may choose a JavaScript-based framework such as React Native, which can provide cross-platform mobile app development.

Interoperability enriches system extensibility, and portability increases the usability of an application. Both are critical attributes of architecture design and may add exponential costs if not addressed during solution design. A solutions architect must carefully consider both aspects per industry requirements and system dependencies.

Automation is key to reducing errors and improving efficiency. We will look at this next.

Applying automation everywhere

Most accidents happen due to human error, which can be avoided using **automation**. Automation not only handles jobs efficiently but also increases productivity and saves costs. Anything identified as a repeatable task can be automated to free up valuable human resources so team members can spend their time on more exciting work and focus on solving a real problem. It also helps to increase team morale.

When designing a solution, think about what can be automated. Think about automating any repeatable task. Consider the following components to be automated in your solution:

- **Application testing:** You need to test your application every time you make any changes to ensure nothing breaks. Also, manual testing is very time-consuming and requires lots of resources. Automating repeatable test cases is better to speed up deployment and product launch. Automate your testing on a production scale and use rolling deployment techniques, such as canary and A/B testing, to release changes. Canary testing involves releasing changes to a small group of users to assess the impact and detect issues before a full rollout, acting as an early warning system for potential problems. A/B testing, or split testing, compares two versions of an application to determine which one performs better with users, guiding decisions based on data.
- **IT infrastructure:** You can automate your infrastructure by using *infrastructure as code* scripting, for example, Ansible, Terraform, and Amazon CloudFormation. The automation of infrastructure allows environments to be created in minutes compared to days. Automating infrastructure as code helps avoid configuration errors and creates a replica of the environment.

- **Logging, monitoring, and alerting:** Monitoring is critical, and you want to monitor everything every time to make sure all parts of your application are functioning properly and you can take proactive measures to fix any issues. You can only monitor the vast system by using automation. You need to automate all activity monitoring and logs to ensure that your application is running smoothly and functioning as desired. Also, based on monitoring, you should take automated action, such as scaling up your system or alerting your team to act.
- **Deployment automation:** Deployment is a repeatable task that is very time-consuming and delays the last-minute launch in many real-time scenarios. Automating your deployment pipeline by applying **continuous integration and continuous deployment (CI/CD)** helps you to be agile and iterate quickly on product features with frequent launches. CI/CD helps you make small, incremental changes to your application.
- **Security automation:** While automating everything, remember to add automation for security. If someone tries to hack your application, you want to know immediately and act quickly.
- You want to take preventive action by automating any incoming or outgoing traffic in your system boundary and setting alerts for suspicious activity.

Automation provides peace of mind by helping to ensure the product functions without a glitch. When designing an application, always think from an automation perspective and consider that as a critical component. You will learn more about automation in *Chapter 9, Operational Excellence Considerations*.

Plan for business continuity

There may be a situation when the entire region where your data center is located goes down due to massive power grid outages, earthquakes, floods, or a security attack but your global business should continue running. In such situations, you must have a disaster recovery plan in which you will plan your business continuity by preparing sufficient IT resources in an entirely different region, perhaps even in different continents or countries, so that your business can get back up and running quickly or experience no downtime at all.

When planning disaster recovery, a solutions architect must understand an organization's **Recovery Time Objective (RTO)** and **Recovery Point Objective (RPO)**. RTO measures how much downtime a business can sustain without significant impact; RPO indicates how much data loss a business can tolerate. Reducing RTO and RPO means incurring higher costs, so it is essential to understand whether the business is mission-critical and needs minimal RTO and RPO. For example, a stock trading application cannot afford to lose a single data point, and a railway signaling application cannot be down for a single second, as human life depends on it.

The architecture diagram in *Figure 2.13* shows a multi-site disaster recovery architecture. The primary data center location is in Ireland, Europe, and the disaster recovery site is in Virginia, USA, hosted on the AWS public cloud. In this case, a business can continue operating even if something happens to the European region or the public cloud. The fact that the disaster recovery plan is based on a multi-site model to achieve minimal RTO and RPO means minimal to no outage and no data loss.

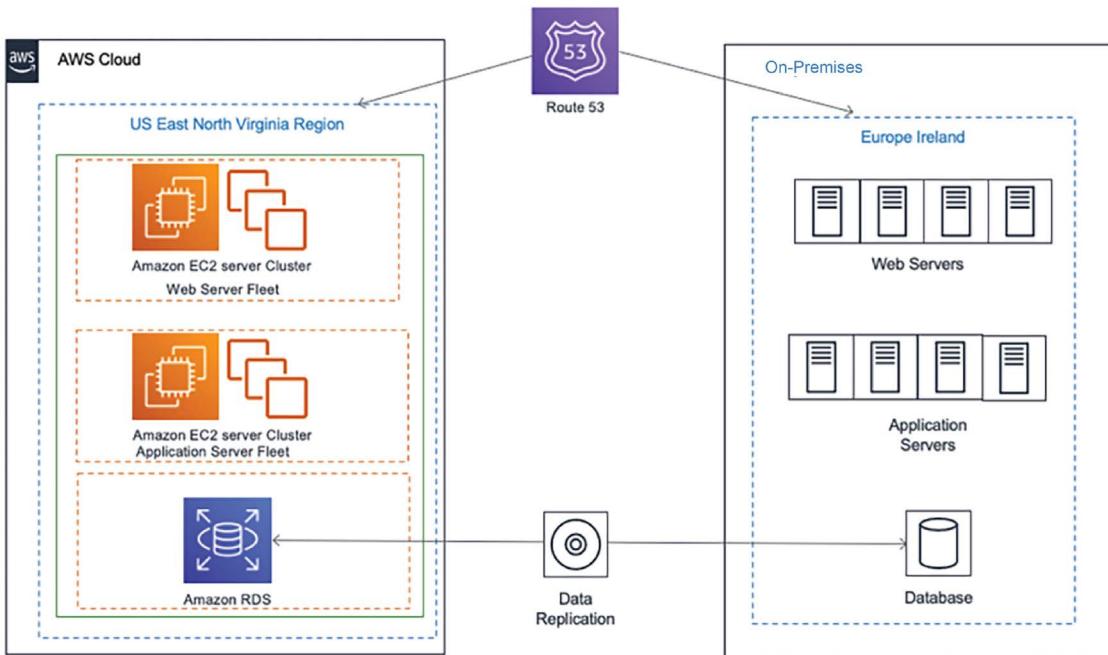


Figure 2.13: Hybrid multi-site disaster recovery architecture

The following are the most common disaster recovery plans, all of which you will learn about in *Chapter 11, DevOps and Solution Architecture Framework*:

- **Backup and Store:** This plan is the least costly but has the maximum RTO and RPO. In this plan, all the server's machine images and database snapshots should be stored in the disaster recovery site. The team will try to restore the disaster site from a backup in a disaster.
- **Pilot lite:** In this plan, all the server's machine images are stored as a backup, and a small database server is maintained in the disaster recovery site with continual data synchronization from the leading site. Other critical services, such as Active Directory, may be running in small instances. In a disaster, the team will try to bring up the server from the machine image and scale up a database. Pilot lite is more costly but has lower RTO and RPO than Backup and Store.
- **Warm standby:** In this plan, all the application and database server (running at low capacity) instances in the disaster recovery site continue to sync up with the leading site. In a disaster, the team will try to scale up all the servers and databases. Warm standby is costlier than the pilot lite option but has lower RTO and RPO.
- **Multi-site:** This plan is the most expensive and has a near-zero RTO and RPO. This plan maintains a replica of the leading site in a disaster recovery site with equal capacity that actively serves user traffic. In a disaster, all traffic will be routed to an alternate location.

Often, organizations choose a less costly option for disaster recovery, but it is essential to perform regular testing to ensure the failover works. The team should make operational excellence a routine checkpoint to ensure business continuity during disaster recovery. Running your application in production and maintaining it for years is very important. Let's learn about the principles of making your application maintainable and operational.

Design for operation

Operational excellence can be a great differentiator for your application by providing a high-quality service to customers with minimal outages. Applying operational excellence proactively also helps support and engineering teams increase productivity. Maintainability goes hand in hand with operational excellence. Easily maintainable applications help reduce costs, avoid errors, and let you gain a competitive edge.

A solutions architect needs to design for operation, including how the workload will be deployed, updated, and operated in the long term. Planning for logging, monitoring, and alerting is essential to capture all incidents and take quick action for the best user experience. Apply automation wherever possible, whether deploying infrastructures or changing the application code, to avoid human error.

Including deployment methods and automation strategy in your design is very important, as this can accelerate the time to market for any new changes without impacting existing operations. Operational excellence planning should consider security and compliance elements, as regulatory requirements may change over time, and your application must adhere to them to operate.

Maintenance can be proactive or reactive; for example, once a new version of an operating system becomes available, you can modernize your application to switch platforms immediately or monitor system health and wait until the end of the life of the software before making any changes. In any case, changes should be made in small increments with a rollback strategy. To apply these changes, you can automate the entire process by setting up a CI/CD pipeline. For the launch, you can plan for A/B testing or blue-green deployment.

For operational readiness, architecture design should include the appropriate documents and knowledge-sharing mechanisms—for example, creating and maintaining a runbook to document routine activity and creating a playbook to guide your system process through issues. This allows you to act quickly in the event of an incident. You should use *root cause analysis* for post-incident reporting to determine why the issue occurred and ensure it doesn't happen again.

Operational excellence and maintenance are ongoing; every operational event and failure is an opportunity to improve your operation by learning from previous mistakes. You must analyze the operation's activities and failures, experiment, and improve. You will learn more about operational excellence in *Chapter 9, Operational Excellence Considerations*.

In *Chapter 1, Solutions Architects in Organizations*, you learned about the various constraints a solution architecture needs to handle and balance. It is a key architectural principle to take care of constraints, and we will look at this next.

Overcoming architectural constraints

When designing an application architecture, the significant limitations are cost, time, budget, scope, schedule, and resources. Overcoming these constraints is a significant factor that must be considered while designing a solution. You should look at the limitations as challenges that can be overcome rather than obstacles, as challenges always push you to the limit of innovation.

A solutions architect needs to make suitable trade-offs while considering the constraints. For example, a high-performance application results in more cost when you need to add additional caching in multiple layers of architecture. However, sometimes, cost is more important than performance, primarily if a system is used by internal employees, which may not directly impact revenue. Sometimes, the market is more important than launching a fully featured product, and you need to make the trade-off between scope versus speed. In such scenarios, you can take the **minimum viable product (MVP)** approach; you will learn more about this in the next section.

Technology constraints become evident in a large organization, as bringing changes across hundreds of systems will be challenging. When designing applications, you need to use the most common technique across the organization. You also need to make sure that the application is upgradable to adopt new technology and be able to plug in components built on a different platform.

A RESTful service model is popular when teams are free to use any technology for their development. They only need to provide a URL with which their services can be accessed. Even legacy systems such as mainframes can be integrated into the new system using an API wrapper around it, and this can help overcome technology challenges.

Throughout this book, you will learn more about handling various architectural constraints. An MVP approach helps you overcome constraints and build a customer-centric product.

Taking the **MVP** approach

For a successful solution, always put the customer first, think backward from the customer's needs, determine what is critical for them, and plan your solution delivery in an agile way.

MVP is a development strategy used to build a new product or website with the minimum amount of features necessary to satisfy early adopters and validate a product idea early in the product development cycle. In this approach, the initial version of the product includes only the core features that allow the product to be deployed and no more. The goal is to provide immediate value, minimize development costs, and gather feedback from customers as quickly as possible to iterate and improve the product over time.

One popular method of prioritizing customer requirements is **MoSCoW**, where you divide requirements into the following categories:

- **Mo (must have):** Requirements that are critical for your customers, without which the product cannot launch
- **S (should have):** Requirements that are the most desirable to the customer once they start utilizing the application
- **Co (could have):** Requirements that are nice to have, but their absence will not impact the desired functionality of the application
- **W (won't have):** Requirements that customers may not notice if they are not there

You need to plan an MVP for your customer with must-have requirements and go for the next delivery iteration with must-have requirements. With this phased delivery approach, you can thoroughly utilize your resources and overcome the challenges of time, budget, scope, and resources. The MVP approach helps you to determine customer needs. You are not trying to build everything without knowing if your features have added value for the customer. This customer-focused approach helps to utilize resources wisely and reduces the waste of resources.

In the following diagram, you can see the MVP evaluation for a truck manufacturing delivery, where the customer wants a delivery truck that gets delivered initially, and you evolve the process based on the customer's requirements and feedback:

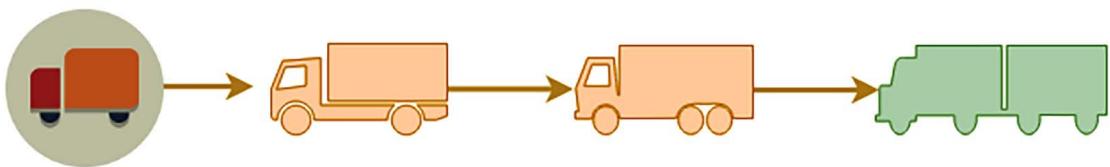


Figure 2.14: MVP approach to building the solution

Once a customer gets the first delivery truck, which is fully functioning, they can determine if they need a stronger or larger truck to handle a more significant load. Based on that, the manufacturer can build a 6-wheel, a 10-wheel, and an 18-wheel truck trailer. This stepwise approach provides working products with essential features that the customers can use, and the team can build upon them as per customer requirements.

You can see how the MVP approach helps to utilize limited resources in an efficient way, which helps to buy more time for quality product development and clarify the scope, in comparison to an approach where we turn up the first time with an 18-wheel truck only to find out we only needed a 6-wheeler. Putting the working product in the customer's hands early gives you an idea of where to invest. As your application has already started generating revenue, you can present use cases to ask for more resources as required.

Summary

In this chapter, you explored an in-depth overview of design principles required to architect effective and efficient systems. Initially, we delved into scalable architecture design, detailing predictive and reactive scaling strategies, and discussing techniques for scaling architecture, including strategies for static content, session management for application server scaling, and database scaling. We also looked at the importance of elasticity.

The chapter then explored constructing a highly available and resilient architecture, emphasizing the necessity of fault tolerance and using replaceable resources for robust system design. A separate section was dedicated to performance, underscoring how to build systems that perform optimally under various conditions.

The principle of loose coupling was discussed next, highlighting its importance in modern design, followed by the approach of “services, not servers,” which is central to the serverless computing paradigm. The chapter also underscored the importance of data-driven design, using data to make informed decisions about system architecture, and explored the need for robust security throughout the architecture. The importance of usability and accessibility in application design was also covered.

Building future-proof, extendable architecture was next on the agenda, focusing on architectural interoperability and portability to guarantee that systems can evolve and adapt to changing needs. The application of automation in all aspects of system architecture was discussed as a means to improve efficiency and reduce error rates.

Designing for operation was emphasized, emphasizing ease of system maintenance and updates. Lastly, the chapter addressed the challenge of overcoming architectural constraints, providing strategies to identify and mitigate the limitations of a given system design. The MVP approach was also explored as a tool for rapidly validating architectural choices.

In the next chapter, you’ll delve into various strategies and methodologies essential for cloud migration, highlighting how businesses transition their infrastructure, applications, and data to the cloud. Additionally, the chapter will cover the intricacies of designing and implementing hybrid cloud architecture, which combines on-premises infrastructure with cloud services, offering a flexible and scalable solution.

Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below to get a free eBook of your choice.



3

Cloud Migration and Cloud Architecture Design

Organizations must continuously acquire new customers, satisfying their needs while working in a fiercely competitive environment. Today's organizations must be more agile to respond to increasing customer demands, which requires quickly scaling up to millions of customers and scaling down as needed without impacting the budget. Cloud migration could be the answer to achieving agility and speed. The cloud enables frequent application releases and reduces costs by applying automation and data center consolidation.

The cloud is becoming essential for every enterprise strategy. Most organizations decrease spending by moving into the public cloud and, in addition to saving costs, they convert upfront capital expenditure into operational expenditure. Many start-ups born in the last decade started in the cloud and were fueled by *cloud infrastructure* for rapid growth. As enterprises move to the cloud, they must focus on cloud migration strategies and the hybrid cloud.

Public clouds such as **Amazon Web Services (AWS)**, **Microsoft Azure**, and **Google Cloud Platform (GCP)** are becoming the primary destinations to host applications, so learning about strategies and methods to migrate to the cloud is essential. In this chapter, you will learn about the various aspects of the cloud and develop “cloud thinking,” which will also help you understand the upcoming chapters better.

This chapter covers the following topics:

- Public, private, and hybrid clouds
- Solution architecture in the public cloud
- Cloud-native architecture
- Creating a cloud migration strategy
- Choosing a cloud strategy
- Steps for cloud migration
- Application optimization in the cloud
- Creating a hybrid cloud architecture

- Taking a multi-cloud approach
- Implementing CloudOps

By the end of this chapter, you will have learned about the benefits of the cloud and you will understand different cloud migration strategies and steps. You will also have learned about hybrid cloud design, taking a multi-cloud approach, and implementing CloudOps.

Public, private, and hybrid clouds

There are three different types of cloud models: public, private, and hybrid.

The **public cloud** is based on the standard computing model in which a service provider makes resources such as **virtual machines (VMs)**, applications, and storage available to customers over the internet. In the cloud computing model, a public cloud vendor provides on-demand availability of IT resources, such as the server, database, network, and storage, which organizations can use with secure web-based interfaces or through application programs over the internet. Public cloud services offer a *pay-as-you-go* model and, in most cases, the customer only pays for the services they are using for the duration of utilization, saving them costs by optimizing IT resources to reduce idle time.

You can think of the public cloud as an electric power supply model, where you switch on the light and pay only for the amount of electricity you use in units. You stop paying for it from the moment you switch it off. It abstracts you from the complexity of power generation using turbines, resources to maintain the facility, and a significant infrastructure setup, and you use the entire service in a simplified way.

In addition to cost benefits, major public cloud providers, such as AWS, GCP, Microsoft Azure, Alibaba, and Oracle Cloud Platform (OCP), help to bring innovation by extending their technology platforms through the cloud. These public cloud providers have mastered scalability and future-looking architecture with comprehensive machine learning and analytics. With the public cloud, you get access to these cutting-edge technologies and the option of using them to advance your architecture.

A **private cloud**, or **on-premises cloud**, is registered to a single organization that owns and accesses it. Private clouds act as a replication or extension of the company's existing data center. In comparison, a public cloud has a shared tenancy, which means virtual servers from multiple customers share the same physical server; however, they offer dedicated physical servers if the customer wants them for a license or compliance need.

The third model is a **hybrid cloud**, used by large enterprises who are moving their workload from on-premises to a cloud, where they still have a legacy application that cannot move to the cloud directly or maybe they have a licensed application that needs to stay on-premises—or sometimes, due to compliance reasons, they need to secure data on-premises. In such a situation, the hybrid model helps when the enterprise has to maintain a partial environment on-premises and move other applications to the public cloud. Sometimes, an organization moves to test and develop the environment to the public cloud and keep production environments on-premises. A hybrid model can vary depending on the organization's cloud strategy.

As multiple public cloud providers are in the market, you may start seeing **multi-cloud** trends, where enterprises choose to distribute their workload between different public cloud vendors to get the most out of each cloud technology or provide options to their team depending on their skill set.

Let's understand more about the public cloud and how it is becoming an essential technology platform for businesses.

Solution architecture in the public cloud

Solution architecture in the cloud has become increasingly important and is becoming the “new normal” as more enterprises choose to migrate their workload to it. The public cloud has been a critical factor fueling start-up organizations’ growth, as they only need a small upfront investment rather than investing upfront in costly on-premises solutions. It allows organizations to be run as an experiment, and to be agile and innovative.

The great thing about cloud computing architecture is that you have an end-to-end view of all architecture components, including the frontend platforms, the application development platform, servers, storage, database, automation, delivery, and the networks that are required to manage the entire solution landscape.

Let's learn more about the public cloud architecture.

Public cloud architecture

A typical definition of the public cloud is a fully virtualized environment accessible over the internet and through a private network. However, public cloud vendors have recently started offering an on-premises physical infrastructure for better hybrid cloud adoption. The public cloud provides a multi-tenancy model, where IT infrastructure, such as storage and computational power, is shared between multiple customers; however, it is isolated at the software and logical network levels and does not interfere with each other’s workloads. Organizations can create network-level isolation in the public cloud to have their virtual private cloud equivalent to the logical data center. Looking at organizations’ regulatory needs, the public cloud also provides dedicated physical instances; however, those are also accessible over the web, but this is a less common option.

Public cloud storage achieves high durability and availability by creating a redundancy model using multiple data centers and robust data replication. This makes it achieve architecture resiliency and easy scalability.

There are three major types of public cloud computing models, as shown in *Figure 3.1*. On-premises solutions are also shown for comparison purposes.

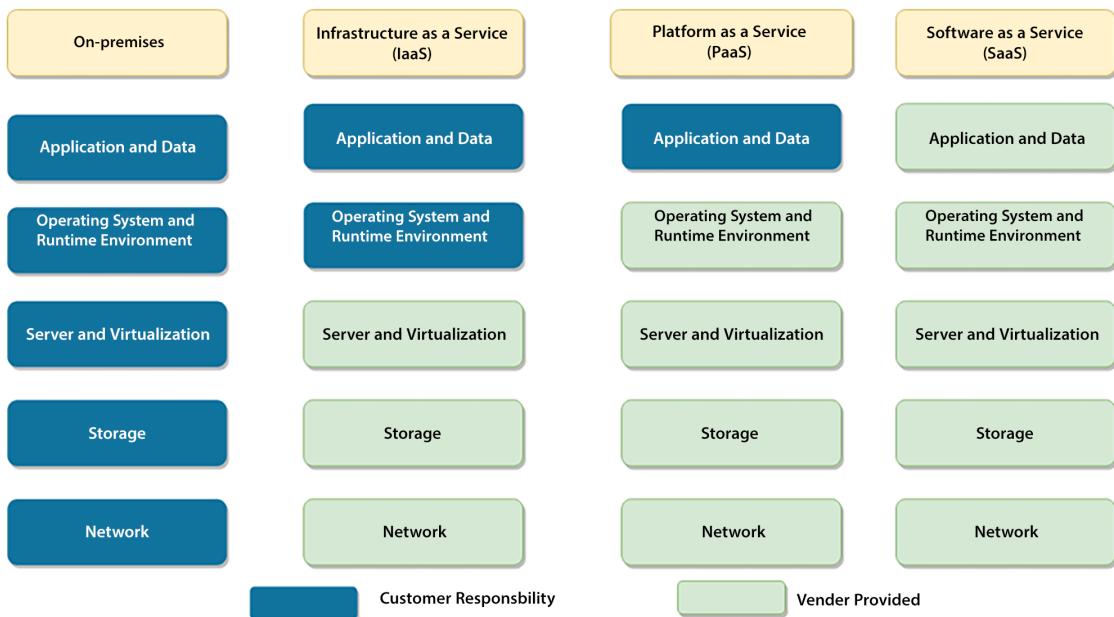


Figure 3.1: Types of cloud computing models

In *Figure 3.1*, you can compare customer responsibilities in the on-premises environment with the cloud computing service model. In the on-premises environment, the customer has to manage everything, while in the cloud computing model, customers can offload responsibilities to the vendor and focus on their business needs. The following points are high-level details of services that are provided under different cloud computing models:

- **Infrastructure as a Service (IaaS):** Here, a cloud vendor provides infrastructure resources, such as a compute server, networking components, and data storage space, as managed services. It helps customers use IT resources without worrying about handling data center overheads, such as heating and cooling, racking and stacking, physical security, and so on.
- **Platform as a Service (PaaS):** The PaaS model adds a layer of service where the cloud vendor takes care of the resources that are required for your development platform, such as the **operating system (OS)**, software maintenance, and patching, along with infrastructure resources. The PaaS model facilitates your team's focus on writing business logic and handling data by taking care of the burden of platform maintenance for you.
- **Software as a Service (SaaS):** The SaaS model adds one more layer of abstraction on top of the PaaS and IaaS models, wherein the cloud or software vendor provides ready-to-use software, and you pay for the service. For example, you use email services such as Gmail, Yahoo Mail, AOL, and so on, where you get your own space for emails as a service and you don't have to worry about underlying applications or infrastructures.

A fourth emerging model is the **Function-as-a-Service (FaaS)** model, which is becoming popular in the building of serverless architecture through using services including AWS Lambda. You will learn more details about serverless architecture in *Chapter 5, Cloud-Native Architecture Design Patterns*. Let's look at a quick overview of the public cloud providers.

Popular public cloud providers

Four major cloud providers primarily dominate the global cloud market. As per Statista's 2023 report, AWS leads the pack with a market share of 32%, offering an extensive array of cloud services encompassing computing, storage, networking, databases, analytics, machine learning, and AI. AWS is used as an example throughout this book.

Microsoft Azure follows closely with a 24% market share, excelling in enterprise applications and hybrid cloud computing. GCP holds an 11% market share and is rapidly growing, particularly renowned for its machine learning and AI prowess. Alibaba Cloud secures the fourth spot with a 4% market share, excelling in the Asia-Pacific region. These four providers command over 70% of the global cloud market. Other significant players include Oracle, IBM Cloud, Tencent Cloud, and Salesforce. You can refer to the detailed report here: <https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/>.

As the public cloud functionality and the cost model are very different, let's learn how to develop a cloud-native approach to architecture design.

Cloud-native architecture

With increasing cloud adoption, cloud-native (or cloud-based) architecture optimizes system architectures for cloud capabilities. A typical on-premises architecture is usually built for fixed infrastructure, as adding new IT resources such as servers and computing power can add a considerable amount of time, cost, and effort. However, the cloud is charged based on usage and provides ease through automation, such as scaling servers up and down, on demand, without worrying about a long procurement cycle. Cloud-native architecture focuses on achieving on-demand scale, distributed design, and replacing failed components rather than fixing them.

In cloud-native architecture, you continually create automated operations for recovery, scalability, self-healing, and high availability using the cloud capabilities of **continuous integration (CI)**, deployment, and infrastructure automation. It encourages the continuous optimization of your application in terms of cost and performance, using new cloud capabilities that are released and improved upon every day.

Public cloud providers allow global infrastructure to spread across the world, which helps an application to be scaled globally near your user base. To encourage adoption, all cloud services provide a free-tier service with lots of learning resources, so you can try your hand and develop your knowledge of them.

The *cloud-native approach* helps employees develop innovative thinking and implement their ideas immediately rather than waiting for the long cycle of infrastructure.

With the cloud, customers don't need to plan excess capacity to handle their peak season, such as the holiday shopping season for retailers; they have the elasticity to provision resources to meet demand instantly. This significantly helps reduce costs and improve the customer's experience. For any organization to stay in the competition, they have to move quickly and innovatively.

The cloud enables enterprises to get their infrastructure quickly across the globe and access various technologies that were not available to them previously. These include access to cutting-edge technologies such as the following:

- Big data and analytics
- Machine learning and AI
- Internet of Things (IoT)
- Blockchain
- Generative AI

Building a solution architecture for the cloud differs from regular enterprise architecting. While moving to the cloud, you have to develop cloud thinking and understand how to leverage the built-in capabilities of the cloud. For cloud thinking, you follow the *pay-as-you-go* model, which means you need to make sure that you optimize your workload correctly and run your servers only when required.



In the cloud, the solutions architect must have a holistic view of each component regarding performance, scaling, high availability, disaster recovery, fault tolerance, security, and automation.

Other areas of optimization are **cloud-native monitoring and alerting mechanisms**. You may not need to bring your existing third-party monitoring tool from on-premises to the cloud, as you can utilize native cloud monitoring better and eliminate costly third-party licensing software. Also, now, you can have deployment capabilities to any part of the world in minutes. Don't restrict yourself to a particular region; utilize the global deployment model to build better high-availability and disaster recovery mechanisms.

The cloud provides excellent deals for automation; *you can automate everything*. Automation reduces errors, speeds up time to market, and saves lots of costs by utilizing human resources efficiently and freeing them from performing tedious and repetitive tasks.

The cloud works on a *shared responsibility model* where cloud vendors are responsible for securing physical infrastructure. However, the security of an application and its data is entirely the customer's responsibility. Therefore, it's important to lock down your environment and keep tabs on security by utilizing cloud-native tools for monitoring, alerts, and automation.

Designing cloud-native architecture

Each organization may have a different opinion on cloud-native architecture, but at the center of it, becoming cloud-native is about utilizing all the cloud capabilities in the best way possible. True cloud-native architecture is about designing your application from its foundations to be built in the cloud.

Cloud-native doesn't mean hosting your application on the cloud platform; it's about leveraging services and features provided by the cloud. This may include the following:

- Containerizing your monolithic architecture in a microservice and creating a CI/CD pipeline for automated deployment.
- Building a serverless application with technology such as AWS Lambda FaaS and Amazon DynamoDB (a managed NoSQL database in the cloud).
- Creating a serverless data lake, by using Amazon S3 (a managed object storage service), AWS Glue (a managed Spark cluster for ETL), and Amazon Athena (a managed Presto cluster for ad hoc queries) for example.
- Using a cloud-native monitoring and logging service, for example, Amazon CloudWatch.
- Using a cloud-native auditing service, for example, AWS CloudTrail.

The following diagram is an example of a cloud-native serverless architecture for a micro-blogging application:

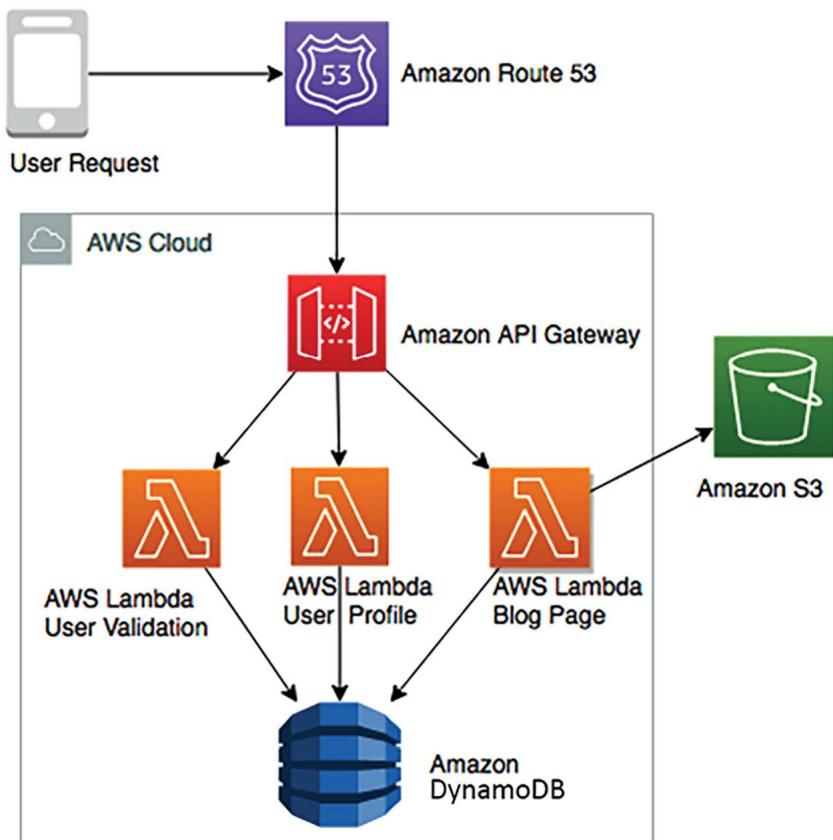


Figure 3.2: Cloud-native micro-blogging application architecture

The preceding diagram depicts utilizing cloud-native serverless services in the AWS cloud. Here, Amazon Route 53, which manages the DNS service, is routing user requests. Lambda manages to function as a service to handle the code for user validation, user profiles, and blog pages. All the blog assets are stored in Amazon S3, which manages object storage services, and all user profile data is stored in Amazon DynamoDB, which the NoSQL store manages.

As users send requests, AWS Lambda validates the user and looks at their profile to ensure they have a subscription in Amazon DynamoDB; after that, it picks blog assets such as pictures, videos, and a static HTML writeup from Amazon S3 and displays them to the user. This architecture can be scaled in an unlimited manner as all services are cloud-native managed services, and you are not handling any infrastructure.

Crucial factors such as high availability, disaster recovery, and scalability are taken care of by these cloud-native services so that you can focus on your feature development. In terms of cost, you will only pay if a request goes to a blogging application. If no one is browsing the blog at night, you don't pay anything for hosting your code; you only pay a nominal storage fee.

The benefit of cloud-native architecture is that it enables fast-paced innovation and agility in the team. It simplifies building out a complex application and infrastructure. As you focus strictly on designing and building your networks, servers, file storage, and other computing resources, you can leave the physical implementation to your cloud computing provider.

Other cloud-native architecture benefits include:

- **Fast scale-out on demand:** You can request the resources you need when you need them. You only pay for what you use.
- **Replicate quickly:** Infrastructure-as-code means you can build once and replicate more. Instead of building your infrastructure by hand, you can structure it as a series of scripts or applications. Building your infrastructure programmatically allows you to build and rebuild it on demand when needed for development or testing.
- **Tear up and tear down easily:** In the cloud, services are provided on demand, so building up a large experimental system is easy. Your system may include a cluster of scalable web and application servers, multiple databases, terabytes of capacity, workflow applications, and monitoring. You can tear it all down after the experiment is completed and save costs.

There are many more examples in the area of storage, networking, and automation for building cloud-native architecture. You will learn more about this architecture in *Chapter 5, Cloud-Native Architecture Design Patterns*.

Throughout this book, you will learn about the cloud perspective of solution architecture and get an in-depth understanding of cloud architecture. In the next section, you will learn about various strategies for cloud migration.

Creating a cloud migration strategy

Your strategy for the cloud helps you to determine a migration strategy and prioritize applications. These are some of the reasons that can trigger an initiative for cloud migration and hybrid cloud strategies:

- The data center needs a technology refresh
- The data center's lease is ending
- The data center has run out of storage and computing capacity
- Modernization of an application
- To leverage cutting-edge technologies such as Generative AI, advanced analytics, machine learning, IoT, and so on
- The need to optimize IT resources to save on operational costs
- Disaster recovery planning and operational resilience
- To utilize a content distribution network for the website
- To reduce upfront capital expenditures and eliminate maintenance costs
- To increase workforce efficiency and productivity
- To improve business agility

Every organization has a different strategy, and one size does not fit all when it comes to cloud adoption. The frequent use cases are putting development and testing environments in the cloud to add agility for developers to move faster. As hosting web applications is becoming more economical and more straightforward with the cloud, organizations are using the cloud for digital transformation by hosting their websites and digital properties in the cloud.

For application accessibility, it is essential to not only build an application for the web browser but also to ensure that it is accessible through *smart mobiles* and *tablets*. The cloud is helping with such transformations. Data processing and analytics is another area where enterprises utilize the cloud since it is less expensive and faster to collect, store, analyze, and share data with the cloud.

Cloud adoption is not just about choosing the platform, security design, and operation; you also need to consider people, processes, and culture in addition to technology. For cloud migration success, you must align leaders and earn team commitment by upskilling them. You need to define the vision across the organization to ensure a successful cloud transition.

Often, migration projects adopt multiple strategies and utilize different tools accordingly. The migration strategy will influence the time it takes to migrate and how the applications are grouped for the migration process. The following diagram shows some of the commonly used strategies for migrating existing applications to the cloud:

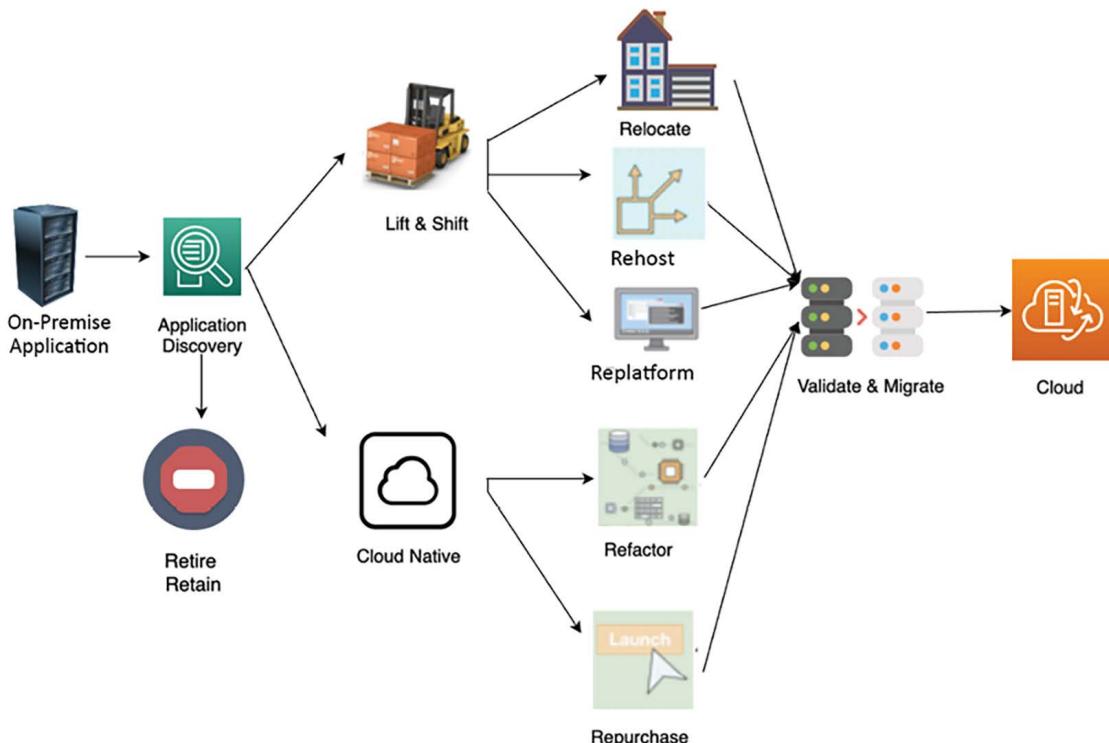


Figure 3.3: Cloud migration strategy

As shown in the preceding diagram, you can do a **lift and shift** of the server or application from the source environment to the cloud. Migrating a resource requires minimal changes for it to work in the cloud. To take a more **cloud-native** approach, you can refactor your application to utilize the cloud-native feature fully, for example, converting monolithic applications into microservices.

If your application is a legacy application that cannot be moved or is not cloud-compatible, you may want to retire it and replace it with a cloud-native SaaS product or third-party solution.

An organization can take a mix of migration strategies; for example, if an application-hosted OS is at its end of life, you need to upgrade it. You can take this opportunity to migrate to the cloud for better flexibility. In this case, you will likely choose the **replatform** method to recompile your code into a new version of the OS and validate all its features. After you've finished testing, you can migrate the application to the OS in the server provided by the cloud. If you want to buy a new platform, for example, replacing your old **customer relationship management (CRM)** solution with SaaS-based solutions provided by Salesforce, you can choose a **retire and repurchase** strategy. If you want to rebuild your application from monolithic to microservices to add more agility, you can **refactor** it.

Your business objectives will drive your decision to migrate applications and define the strategy for migration as per their priority. For example, when cost efficiency is the primary driver, the migration strategy typically involves mass migration with a heavy focus on the **lift and shift** approach. However, if the main goal is to enable agility and innovation, the cloud-native approach (such as rearchitecting and refactoring) plays a crucial role in the cloud migration strategy. Let's learn more about each strategy in the following subsections.

Lift and shift migration

Lift and shift is the fastest migration mode; it takes minimal work to move your application. However, this mode needs to take advantage of the cloud-native features. Let's look at the most common migration strategies (**rehost**, **replatform**, and **relocate**) that are often utilized to lift and shift.

Rehost

Rehosting is fast, predictable, repeatable, and economical, which makes it the preferred method for migrating to the cloud. Rehosting is one of the quickest cloud migration strategies, where the server or application is lifted and shifted from the source on-premises environment to the cloud. Minimal changes may be made to the resources during the migration process.

Customers often use rehosting to quickly migrate their applications to the cloud and then focus on optimization when the resources run in the cloud. This technique allows them to realize the cost benefits of using the cloud.

Customers typically use the rehosting technique for the following:

- A temporary development and testing environment
- When servers are running packaged software, such as SAP and Microsoft SharePoint
- When an application doesn't have an active roadmap

While rehosting is intended to be applied to packaged software and helps you move quickly into the cloud, you may need to upgrade underlying application platforms such as operating systems. In such a situation, you can use the replatform approach of cloud migration.

Replatform

When an OS, server, or database version gets to its end of life, it can trigger a cloud migration project, for example, upgrading the OS of your web server to Microsoft Windows 2022, upgrading your database to Oracle 23c, and so on. The strategy involves upgrading the platform as a part of the cloud migration project without changing the application architecture. You can decide to update your OS or application to a newer release as part of the migration.

When using the replatform migration strategy, you may need to reinstall your application on the target environment, which triggers application changes. This requires thorough testing on your application after replatforming to ensure and validate its post-migration operational efficiency.

The following common reasons warrant the use of the replatform technique:

- Changing the OS from 32-bit to 64-bit
- Changing the database engine
- Updating the latest release of the application
- Upgrading the OS
- Upgrading the database engine
- To get the benefits of managed services that are available from cloud vendors, such as managed storage, databases, application deployment, and monitoring tools

Replatforming helps you advance your application's underlying platform while migrating to the cloud. You can relocate your application to the cloud if deployed in containers or VMware. Now, let's learn more about the relocation strategy.

Relocate

You may deploy your application using containers or VMware appliances in your on-premises data center. You can move such workloads to the cloud using the accelerated migrations strategy: **relocate**. Relocation helps you move hundreds of applications in days. You can quickly relocate applications based on VMware and container technologies to the cloud with minimal effort and complexity.

The relocation strategy only requires a little upfront developer investment or a test schedule since it provides the agility and automation you expect from the cloud. You need to determine existing configurations and use VMotion or Docker to relocate your servers to the cloud. You will learn more about Docker in *Chapter 6, Performance Considerations*.



VMotion is known for live migration. It's a VMware technology that enables a virtual instance to be moved from one physical host server to another without any interruption to the service.

While migrating your application to the cloud, you may want to take the opportunity to rebuild and rearchitect your entire application to make it more cloud-native. The cloud-native approach allows you to use the full capability of the cloud. Let's learn more about the cloud-native approach.

The cloud-native approach

When your team decides to move to cloud-native, in the short term, it seems like more upfront work and a slower migration to the cloud. This is a bit costly, but it pays off in the long term when you use all the cloud benefits with an agile team to innovate.

With the cloud-native approach, you will see a drastic decrease in costs over time, as you can optimize your workload for the right price while keeping performance intact with the *pay-as-you-go* model. The cloud-native approach includes containerizing your application by rearchitecting it as a microservice or opting for a purely serverless approach.

Let's learn more about the refactor and repurchase methods for the cloud-native migration approach.

Refactor

The refactor method involves rearchitecting and rewriting an application before migrating it to the cloud to make it a cloud-native application. In refactoring, you change the application to a more modular design, such as from monolithic to microservices. Refactoring to microservices helps organizations create small independent teams that can take complete ownership, thus increasing the speed of innovation.

Cloud-native applications are applications that have been designed, architected, and built to perform efficiently in a cloud environment. The benefits of these cloud-inherent capabilities include *scalability*, *security*, *agility*, and *cost-efficiency*.

Refactoring requires more time and resources before migration to recode the application and architecture. This approach is commonly used by organizations with extensive cloud experience or a highly skilled workforce. An alternative option for refactoring is to migrate your application to the cloud and then optimize it. You can use cloud-native serverless technologies to reduce the admin overhead of a modular design.

Common examples of refactoring include the following:

- Changing platforms, such as from AIX to Unix
- Database transition from traditional to cloud databases
- Replacing middleware products
- Rearchitecting the application from monolithic to microservices
- Rebuilding application architecture, such as containerizing or making it serverless
- Recoding application components
- Data warehouse modernization to connect organizations to customers

Deciding whether to rearchitect an application to microservices or rebuild it for containerization or serverless architecture requires careful consideration of the organization's strategic goals, cost implications, and technical capabilities. Rearchitecting to microservices offers enhanced scalability and flexibility, allowing each service to be developed, deployed, and scaled independently, which can significantly benefit long-term agility and efficiency. However, this approach might involve substantial refactoring and could introduce complexity in terms of service coordination and network communication. On the other hand, rebuilding the architecture to adopt containerization or serverless models can streamline operations, reduce infrastructure overhead, and improve deployment speed, albeit with an initial investment in development and a potential learning curve for the team.

The decision should factor in the application's compatibility with the new architecture, the expertise of the team, and the potential impact on performance and reliability. While microservices can improve fault isolation and facilitate more granular resource management, containerization and serverless architectures can optimize resource use and potentially reduce costs. Security and compliance are also critical considerations, as each architectural style introduces unique challenges that must be proactively addressed.

Sometimes, a significant effort is made to rebuild an application. As an architect, you should evaluate whether purchasing a SaaS product helps you get a better **return on investment (ROI)**. Let's explore the repurchase strategy in more detail.

Repurchase

When your IT resources and projects are migrated to the cloud, you may need servers or applications requiring you to purchase a cloud-compatible license or release. For example, the current on-premises license for your application might need to be validated when you run the application in the cloud.

There are multiple ways to address such scenarios of licensing. You can purchase a new license and continue using your application in the cloud, or drop the existing one and replace it with another one in the cloud. This replacement could be a SaaS offering of the same application.

The cloud may not be the answer to all of your problems and, sometimes, you will find a legacy application that may not benefit from cloud migration or discover rarely used applications that can be retired.

Workload rationalization

Workload rationalization is a strategic process in cloud migration and architecture design, focusing on the consolidation of similar services to streamline operations and enhance efficiency. This approach involves evaluating and amalgamating disparate systems, such as multiple CRM systems, into a unified solution. The goal is to eliminate redundancy, reduce complexity, and optimize resource utilization across the organization's IT landscape.



In many firms, workload rationalization is a critical initiative, guiding decisions on which applications to keep, update, retire, or consolidate. This process not only aids in simplifying the technological stack but also aligns IT resources with business objectives, ensuring that each service is necessary, efficient, and contributes to the overarching goals of the organization. Through rationalization, companies can achieve a more agile, cost-effective, and scalable IT infrastructure, particularly beneficial in the context of cloud migration and hybrid cloud architecture, where efficiency and adaptability are key.

Let's learn about the *retain or retire strategy* in more detail next.

Retain or retire

When you are planning a cloud migration, it may only be necessary to move some applications. You may need to retain some applications due to technology constraints; for instance, there may be legacy applications coupled with an on-premises server that cannot move. On the other hand, you may retire some applications and use cloud-native capabilities, for example, third-party application monitoring and alerting systems. Let's learn more about the retain or retire strategy.

Retain

You might encounter a few applications in your on-premises environment that are essential for your business but unsuitable for migration because of technical reasons, such as the OS/application not being supported on a cloud platform. In such situations, your application cannot be migrated to the cloud but you can continue running it in your on-premises environment.

For such servers and applications, you may need to perform only an initial analysis to determine their suitability for cloud migration. However, the server or application may still have dependencies with migrated applications. Therefore, you may have to maintain the connectivity of these on-premises servers to your cloud environment. In this chapter's *Creating a hybrid cloud architecture* section, you will learn more about on-premises to cloud connectivity.

You may want to retain complex legacy systems on-premises and prioritize them so they can be moved later; however, during discovery, organizations often find applications that are not in use but are still sitting around and consuming infrastructure space. You can choose to retire such applications. Let's explore more about the retirement strategy.

Retire

While migrating to the cloud, you may discover the following:

- Rarely used applications
- Applications consuming an excessive amount of server capacity
- Applications that may not be required due to cloud incompatibility

In such a situation, you can retire the existing workload and take a fresh approach that is more cloud-native.

A retirement strategy can be applied to hosts and applications that will soon be decommissioned. This can also be applied to unnecessary and redundant hosted applications. Depending on your business needs, such applications can be decommissioned on-premises without migrating to the cloud. Hosts and applications that are commonly suited for retirement include the following:

- On-premises servers and storage for disaster recovery purposes
- Server consolidation to resolve redundancies
- Duplicate resources due to mergers and acquisitions
- Alternative hosts in a typical high-availability setup
- Third-party licensed tools (such as workload monitoring and automation) that are available as in-built capabilities in the cloud

Most migration projects employ multiple strategies, and different tools are available for each strategy. The migration strategy will influence the time it takes to migrate and how the applications are grouped for the migration process. Cloud migration is a good time to examine your overall inventory and eliminate any ghost servers going unaccounted for. In this section, you learned about various cloud strategies. Let's take a quick look at comparing them in the next section.

Choosing a cloud migration strategy

Choosing the right migration strategy for cloud adoption for your business drivers is critical. Consider various constraints such as financial, resources, time, and skills. You can compare the effort required for the different strategies covered in the previous section in the following table. The bars in the table depict the amount of time and cost required for each strategy and the level of the optimization opportunity.

Migration Strategy	Description	Time and Cost	Optimization Opportunities
Refactor	Rearchitect applications into a more modularized form to make them cloud-native		
Replatform	Migrate applications to the upgraded platform without changing the core architecture		
Repurchase	Replace your current environment by purchasing a cloud-based solution		
Rehost	Quickly lift and shift your applications to the cloud without architecture changes		
Retain	Leave the application on-premises, for now at least		N/A
Relocate	Quickly relocate applications to the cloud without changing them		N/A
Retire	Identify the assets that are no longer useful and remove them entirely	N/A	N/A

Table 3.1 – Comparison of cloud migration strategies

Cloud migration, while offering numerous benefits such as scalability, flexibility, and cost-efficiency, also comes with its set of risks. Organizations must be aware of these potential pitfalls to mitigate them effectively. Possible risks include:

- **Data loss and leakage:** During the migration, sensitive data can be exposed to risks if not properly encrypted and managed. Ensuring data integrity and security during the migration process is crucial to prevent data breaches.
- **Downtime:** Migration can lead to system downtime, affecting business operations. Planning and executing the migration in phases or during off-peak hours can minimize the impact on business continuity.
- **Cost overruns:** Without the proper planning and understanding of cloud pricing models, organizations can face unexpected costs. It's essential to have a clear roadmap and budget for the migration process.
- **Performance issues:** Applications may not initially perform as expected in the cloud due to differences in the architecture or unforeseen compatibility issues. Rigorous testing and optimization are necessary post-migration.

- **Skill gaps:** The lack of expertise in cloud computing within the organization can hinder the migration process and future operations. Investing in training and possibly hiring specialists can mitigate this risk.
- **Interoperability and integration challenges:** Ensuring that existing systems and applications work seamlessly with cloud services can be complex, requiring robust integration and testing strategies.
- **Compliance:** Adhering to industry regulations and compliance standards can be challenging in the cloud environment, especially if the organization operates in a highly regulated sector.
- **Vendor lock-in:** Relying too much on a single cloud provider's technologies and services can lead to difficulties in switching providers in the future, potentially affecting flexibility and cost-efficiency.

To reduce the risks associated with cloud migration, it's always recommended to take a phased approach when migrating applications to the cloud. First, prioritize business functionality and optimize applications to realize the difference in cost-saving, performance improvement, and resource productivity. Try to migrate first; then, in subsequent phases, you can go for optimization. For example, if you are migrating an application that uses an MS SQL database and replacing it with a cloud-native database such as Amazon Aurora or Azure SQL, the best approach is to migrate the application in the first phase, followed by migrating the database while monitoring risk and application stability in the second phase. You can optimize your application in subsequent steps by using a cloud-native serverless tech stack such as AWS Lambda, Amazon API Gateway, and Amazon DynamoDB.

The migration strategy should be defined to execute quickly by allowing teams to work independently. The cloud migration strategy can impact other organizational factors, such as building engineering functions within the organization rather than outsourcing. Migrating to the cloud also offers an excellent opportunity to adopt or enhance a DevOps culture within the organization. This culture emphasizes collaboration between development and operations teams, streamlining workflows, and boosting efficiency.



Organizations often see the unexpected advantage of optimizing the workload and tightening security while running application discovery to prepare for migration.

There are multiple phases involved in cloud migration. In the next section, you will learn about the steps for cloud migration.

Steps for cloud migration

In the previous section, you learned about different migration strategies and you may have started to group your applications to apply the appropriate migration technique. These strategies are also known as the 7 Rs (retain, retire, relocate, rehost, repurchase, replatform, and refactor), and some or all of them could be part of your cloud journey.

Since you may need to migrate and manage multiple applications in the cloud, it's advisable to set up a cloud **Center of Excellence (CoE)** and standardize this process with a cloud migration factory. The cloud CoE includes experienced people from various IT and business teams across the organization who act as a dedicated cloud team focused on accelerating the building of cloud expertise in the organization. The cloud migration factory defines migration processes and tools, as well as the steps that need to be taken, as shown in the following diagram:

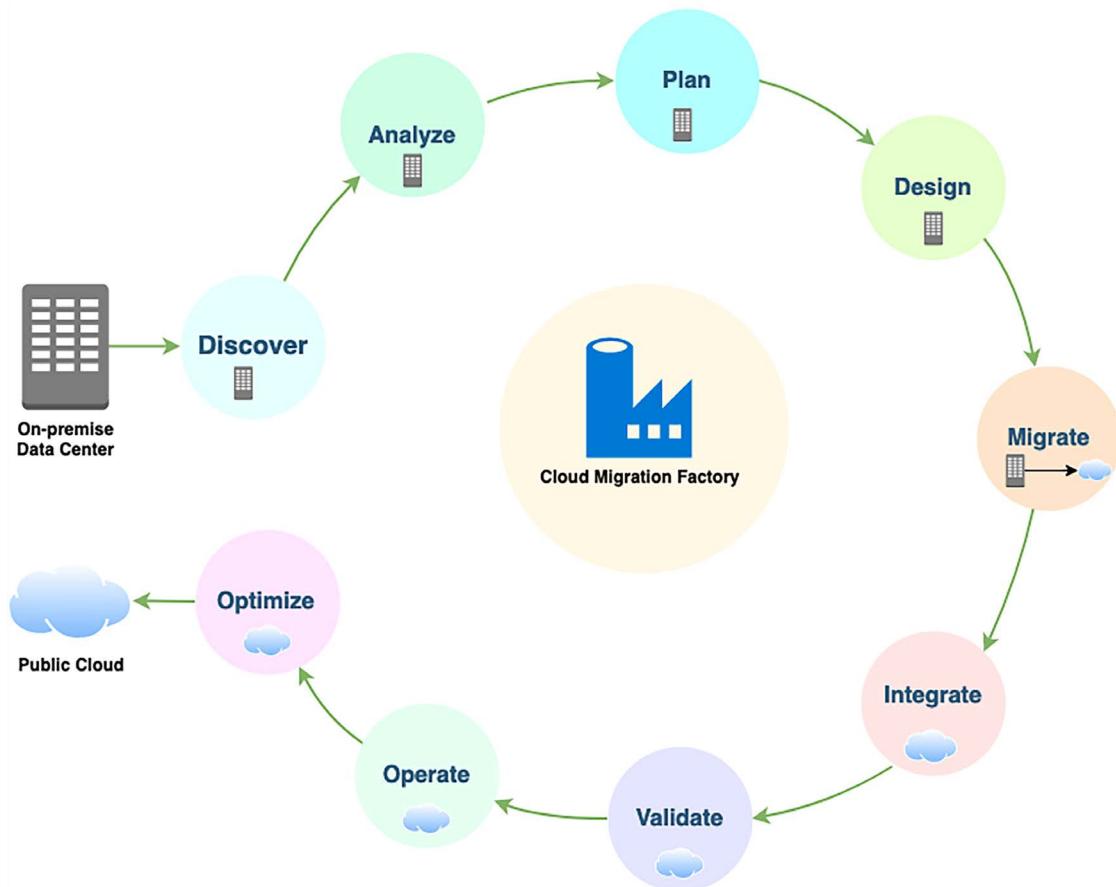


Figure 3.4: Cloud migration steps

As shown in the previous diagram, the cloud migration steps are the following:

- **Discover:** Discovery of cloud migration portfolios and on-premises workloads
- **Analyze:** Analyze discovered data and workloads
- **Plan:** Plan the migration to the cloud and define the migration strategy
- **Design:** Design the application as per the migration strategy
- **Migrate:** Execute the migration strategy
- **Integrate:** Integrate with other application and system dependencies
- **Validate:** Validate functionality after migration

- **Operate:** Plan to operate in the cloud
- **Optimize:** Optimize your workload for the cloud

One of the initial steps of a cloud migration project is to assess and prioritize the applications for migration. To accomplish this, you need to get a complete inventory of the IT assets in your environment to determine which servers, applications, and business units are suitable for migrating to the cloud, prioritize the migration plan, and determine a migration strategy for these applications. Let's drill down into each step and learn more about them.

Discovering your portfolio and workloads

In the discovery phase of your migration project, you discover and capture detailed data about your *cloud migration portfolio*, for example, the scope of your migration project. You identify servers and applications in your portfolio, as well as their interdependencies and current baseline performance metrics. In addition, workload discovery includes understanding the following components:

- **Storage:** Identifying the amount and type of data stored, such as databases and files. For example, discovering whether an application uses 1 TB of block storage for database operations.
- **Networking configurations:** Understanding network topology, including subnets, firewalls, and load balancers. For instance, identifying that an application is spread across multiple subnets with specific firewall rules.
- **Security and compliance needs:** Documenting security policies, data protection mechanisms, and compliance requirements, like identifying that an application must comply with GDPR and use encryption for data at rest.
- **Application release frequency:** Knowing how often new releases are deployed, such as determining whether an application follows a bi-weekly release schedule.
- **DevOps model:** Understanding the integration and deployment processes, tools used, and automation level. For example, noting that the organization uses Jenkins for CI/CD with a high degree of pipeline automation.
- **Escalation path:** Documenting the process for handling incidents and outages, like identifying key contacts and procedures in case of a service disruption.
- **OS maintenance and patching:** Understanding how and when OS updates are applied, such as if servers are auto-patched monthly or manually updated.
- **Licensing requirements:** Identifying any software licenses that need to be maintained or updated, like checking whether the application uses licensed middleware that needs to be accounted for in the cloud.
- **Other associated assets:** Noting additional components tied to the workload, such as external dependencies, third-party services, or integrated tools. For example, recognizing an application's dependency on an external payment gateway service.

In addition to helping you design and architect the target cloud environment and identifying the cost, detailed discovery can also help identify any issues in the current state of the application that might need mitigation before you migrate to the cloud.

Portfolio discovery identifies all the IT assets involved in your cloud migration project, including servers and applications, their dependencies, and performance metrics.

You will also need to gather business details about your resources, such as the current state of the resource, the refresh cycle of the application, the roadmap of the application, and the business criticality of the server or application. These details will help you determine your migration strategy and create a migration plan. In most organizations, these details are maintained across multiple business units and teams. Therefore, during the discovery process, you may have to interact with various teams, such as business, development, data center, network, and finance.

It is essential to understand that your discovery landscape will depend on various factors:

- What has already been migrated to the cloud?
- What application dependencies are there, along with resources and assets?
- What are the business drivers for cloud migration?
- What is the estimated duration for the entire migration project?
- How many phases is the migration process going to happen in?

One of the top challenges of a migration project is *determining interdependencies among applications*, particularly since they pertain to **input/output (I/O)** operations and communications. Cloud migration becomes even more challenging as organizations expand due to mergers, acquisitions, and growth. Organizations often do not have complete information about the following:

- The inventory of the number of servers
- Server specifications such as the type and version of OS, RAM, CPU, and disk
- Server utilization and performance metrics
- Server dependencies
- Overall networking details

Performing thorough portfolio discovery helps in answering questions such as the following:

- Which applications, business units, and data centers are good candidates for migration?
- How suitable are the applications for migrating to the cloud?
- What are known or possible unknown risks associated with migrating an application to the cloud?
- How should the applications be prioritized for migration?
- Which other IT assets is the application dependent on?
- What are the best migration strategies for the application?
- Is it better to have some downtime for the application than to perform a live migration due to its dependencies and risks?

Several tools are available in the market to help automate the discovery process and provide more detailed information in various formats. These tools can be classified based on various characteristics, such as deployment type, operation, support, and type of data discovered and reported. Most of the available solutions can be broadly classified into two categories:

- **Agent-based solutions**, which require their software client to be installed on a server to gather the necessary details. For example, installing a monitoring agent on all servers in an environment to track performance metrics, software inventory, and system logs.
- **Agentless solutions**, which may be able to capture this information without any additional installations. An example could be using network-based scanning tools that remotely check servers for open ports, running services, and vulnerabilities by interfacing with the existing network protocols and management interfaces.

Some solutions perform *port scanning* to probe a server or host for open ports. In contrast, others perform *packet scanning*, often involving capturing and analyzing network packets to decode the information. The tools also vary based on the granularity of the discovered data, the storage types, and the reporting options. For example, some tools can provide a higher stack of intelligence beyond the network and can also determine the type of applications that are running.

The complexity of the discovery process depends on the organization's workload and whether it already has a well-maintained inventory in place. Discovery processes are typically run for at least a couple of weeks to gather more holistic information about your environment. Once you discover all the necessary information, you need to analyze it. Let's look at the analysis step in more detail.

Analyzing the information

Analyzing information is crucial for cloud migration as it provides a detailed understanding of the current IT landscape, enabling informed decision-making. This analysis helps identify which applications and workloads are suitable for migration, assesses their compatibility with cloud environments, and determines the optimal cloud services and architecture. It also uncovers dependencies between applications and infrastructure, ensuring a smooth transition without disrupting business operations. Moreover, a thorough analysis aids in anticipating and mitigating potential risks, optimizing resource allocation, and forecasting costs, thereby ensuring a cost-effective and successful migration to the cloud.

To identify server and application dependencies, you need to analyze the network connectivity data, port connections, system, and process information on the hosts. Depending on your tool, you can visualize all the contacts from a server to identify its dependencies, or you can run queries to list all the servers running a specific process, using a particular port, or talking to a specific host.

To group your servers and applications for migration scheduling, you need to identify patterns in your host configurations. Often, some prefixes are embedded in the server hostnames to signify their association with a particular workload, business unit, application, or requirement. Some environments also use tags and other metadata to associate such details with the host.

To right-size your target environment, you can analyze the performance metrics for your servers and applications:

- If a server is *over-provisioned*, you can revise your right-size mapping information. You can also optimize this process by leveraging the utilization data for the server/application instead of the server specifications.
- If a server is *under-provisioned*, you might assign a higher priority to the server to migrate to the cloud.

You can combine the insights you gain with the availability of your resources and business requirements to prioritize your cloud migration workload. This can help you determine the number of servers included in each cloud migration sprint.

Based on the discovery and analysis of your cloud migration portfolio, you can determine an appropriate cloud migration strategy for your applications. For instance, servers and applications that are less complex and run on a supported OS might be suitable candidates for a lift and shift strategy. Servers or applications that run on an unsupported OS might need further analysis to determine an appropriate strategy.

The next phase in your migration project is *planning cloud migration*. In this phase, you will use the information you gathered during the portfolio discovery phase and analysis to create an efficient migration plan. Let's look at migration planning in more detail.

Creating a migration plan

In a cloud migration project, discovery, analysis, and planning are tightly integrated. You fully discover your cloud migration portfolio and analyze the data to create a migration plan. By the end of the analysis phase, based on your analysis and the details you've gathered from business owners, you should be able to do the following for each server/application that is part of your cloud migration portfolio:

- Choose a migration strategy, depending on your organization's cloud adoption strategy. You may be limited to specific choices within the retain, retire, relocate, repurchase, rehost, re-platform, and refactor strategies.
- Assign a priority for migrating the resources to the cloud. Eventually, all the resources that are part of the cloud migration portfolio may migrate to the cloud, but this priority will determine the urgency of that migration. A higher-priority resource might move earlier in the migration schedule.
- Document the business driver for migrating the resources to the cloud, which will drive the need and priority for migrating the resources to the cloud.

Planning utilizes the information collected in the discovery and analysis phase to create migration waves. Waves are logical groupings of resources that can be sequentially deployed into production and test/dev environments during cloud migration.

By the end of this phase in your migration project, you should be able to create an ordered backlog of applications that can migrate to the cloud.

In addition to choosing a migration strategy, the main goals of the migration planning phase include the following:

- Defining the success criteria for the migration
- Determining the right size of the resources in the cloud
- Determining the priority of applications to migrate to the cloud
- Identifying migration patterns

- Creating a detailed migration plan, checklist, and schedule
- Creating migration sprint teams
- Identifying tools for migration



Sprints and backlogs are part of the continuous delivery methodologies Agile and Scrum.

In preparation for the migration planning phase, you need to have performed a detailed discovery and analysis of all the IT assets that are part of your cloud migration portfolio. Migration planning includes determining the cloud account structure and creating a network structure for your application. It is also essential to understand hybrid connectivity with the target cloud environment. Hybrid connectivity will help you plan for applications that depend on resources that are still running on-premises.

The order of application migration can be determined through three high-level steps:

1. Evaluate each application across several business and technical dimensions associated with a potential migration to accurately quantify the environment.
2. Identify the dependencies for each application with locked, tightly coupled, and loosely coupled qualifications to identify any dependency-based ordering requirements.
3. Determine the desired prioritization strategy of the organization to determine the appropriate relative weighting of the various dimensions.

If the organizational strategy is to minimize risk, then business criticality will have more weight in identifying the applications. If ease of migration is the strategy, applications that can be migrated using the rehost method will have higher priority, as rehosting is a more straightforward process than other strategies. The outcome of planning should be an ordered list of applications that can be used to schedule the cloud migration.

Designing the application

During the design phase, your focus should be on successfully migrating applications and ensuring your application design meets the success criteria identified in the planning phase. You should also ensure your application is up to date after it has been migrated to the cloud. For example, if you are maintaining user sessions in the on-premises application server (so that it can scale horizontally), make sure that a similar architecture is implemented in the cloud after the migration, which defines the success criteria.

During the design phase, you will identify the architecture gap and enhance your architecture as per your application requirements. When you have multiple accounts, each account may have some level of relationship or dependency; for example, you can have a security account to ensure that all your resources comply with company-wide security guidelines.

When thinking about your application's network design, you need to consider the following:

- Network packet flows entering the boundaries of your application
- External and internal traffic routing
- Firewall rules for network protection
- Application isolation from the internet and other internal applications
- Overall network compliance and governance
- Network log and flow audit
- Separation of application risk levels, as per their exposure to data and users
- DDoS attack protection and prevention
- Network requirements for production and non-production environments
- SaaS-based multi-tenancy application access requirements
- Network boundaries at the business unit level in an organization
- Billing and implementation of the shared services model across the business unit

Depending on your connectivity needs, you can consider hybrid connectivity options with an on-premises system.



To build and maintain a secure, reliable, performant, and cost-optimized architecture in the cloud, you need to apply best practices. Review your cloud foundational architecture against cloud best practices before migrating to the cloud.

When migrating to the cloud, you can design your application architecture to benefit from the global cloud infrastructure, increase the proximity to your end users, mitigate risk, improve security, and address data residency constraints. Systems expected to grow over time should be built on top of a scalable architecture that can support growth in users, traffic, or data with no drop in performance.

You could make specific architecture components stateless for applications that need to maintain some user state information. If any layers in the architecture need to be stateful, you could leverage techniques such as session affinity to scale such components. Leverage a distributed processing approach for applications that process vast amounts of data.

Another approach to reducing the operational complexity of running applications is using serverless architecture. This type of architecture can reduce costs because you are neither paying for underutilized servers nor provisioning redundant infrastructure to implement high availability. In *Chapter 5, Cloud-Native Architecture Design Patterns*, you will learn more about serverless architecture.

The following diagrams show an example migration design from on-premises to the AWS cloud. The on-premises design is as follows:

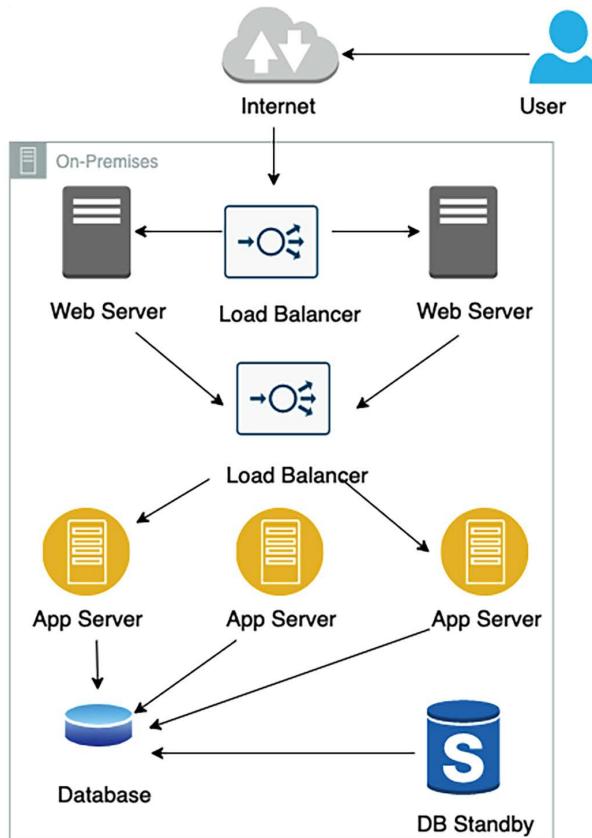


Figure 3.5: On-premises architecture mapping

The depicted architecture outlines a high-availability setup for a web application that is spread across multiple tiers, each with a specific role in processing user requests. Users access the application through the internet, with their requests being evenly distributed by load balancers to a cluster of web servers. These servers serve the frontend content and may also perform some initial request processing.

Subsequently, deeper business logic is handled by a separate layer of application servers, which might interact with a database for data retrieval and storage. To ensure data integrity and continuity, a standby database is maintained, ready to take over in case the primary database encounters issues.

This multi-tiered approach, with redundancy at both the web and application layers, plus a failover strategy for the database, aims to provide robustness against server failures and optimal performance even under high traffic conditions.

Now we transition to an AWS cloud design:

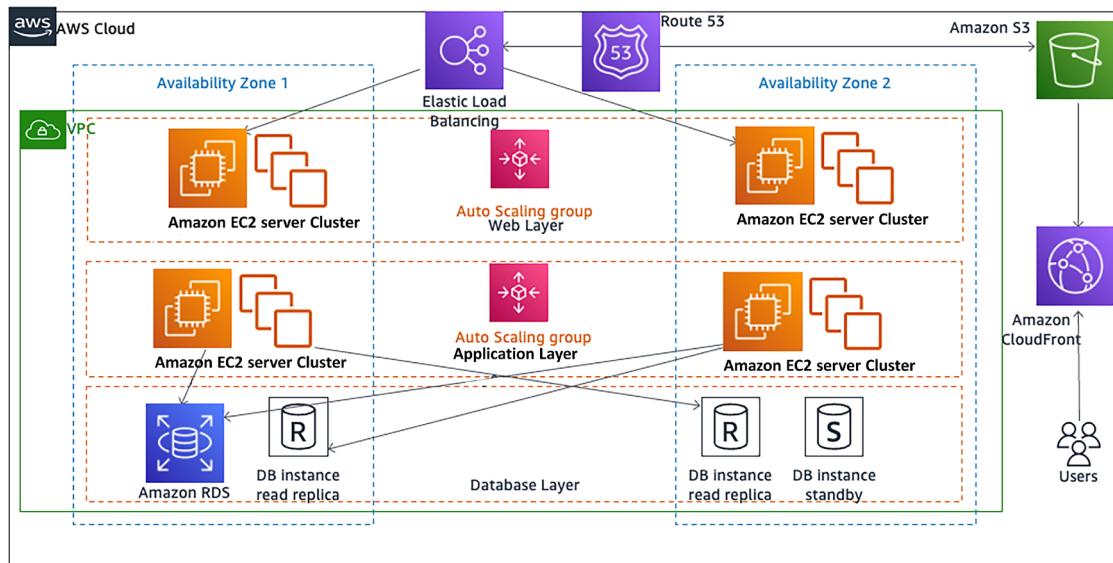


Figure 3.6: On-premises to AWS cloud architecture mapping

In the preceding diagram, as part of the cloud migration strategy, it was determined to rehost the web servers and introduce auto-scaling to provide the elasticity that can help meet spikes in demand. Elastic load balancers were also added to distribute the incoming traffic to the web server instances. The application servers were migrated using the refactor method, and the platform for the database tier changed from the traditional database to a cloud-native **Amazon Relational Database Service (Amazon RDS)**. The entire architecture is distributed across multiple Availability Zones, and the database replicates to a standby instance in the second Availability Zone to provide high availability.

As an output of your design phase, you should create a detailed design document for the architecture of your application in the cloud. The design document should include details such as:

- **User account migration:** The user accounts that will be transferred during the migration process
- **Network configuration:** The network setup required for the application in the new environment
- **Access control lists:** A comprehensive list of users, groups, and applications requiring access to the migrated data
- **Hosting details:** Where and how the application will be hosted post-migration
- **Backup requirements:** The backup strategies and requirements specific to the application
- **Licensing needs:** Any licensing requirements associated with the application
- **Monitoring protocols:** The monitoring systems and protocols that will be in place
- **Security measures:** The security measures and compliance standards that the application must adhere to
- **Maintenance and patching:** Procedures for regular maintenance and patching schedules

Ensure that you create a design document for each application. You must perform basic cloud and application functionality checks during the migration validation phase.

Executing application migration to the cloud

The migration execution step brings your plans to fruition. In the execution phase, you must define steps and configurations, as you will repeat them during the dev/test and production waves. Before executing the migration, ensure that you have a migration plan and that you have identified the sprint teams and migration waves and schedules, created a prioritized backlog, and notified all the application stakeholders about the migration schedule, timelines, and their roles and responsibilities.

You must also ensure that the target environment in the cloud has already been set up with the foundational architecture and core services. You might have some application-specific pre-steps, such as performing a backup or sync before migration, shutting down the servers, or unmounting disks and devices from the server. Ensure you put your essential components in place, such as networking and firewall rules, authentication and authorization, and accounts. All need to be configured appropriately. You need to test your applications on the infrastructure to ensure they have access to required servers, load balancers, databases, authentication servers, and so on. You must focus on application logging and monitoring to measure performance post-migration.

Ensure you have good network connectivity with the cloud environment during migration. A good estimate of the amount of data that needs to be migrated also helps you properly estimate the time it will take to migrate your data to the cloud, given other factors such as bandwidth and network connectivity. You also need to understand the tools available to perform the migration. Given the number of available devices in the market, you might have to narrow down the selection criteria based on your requirements and other constraints.

As you know, rehosting is often the fastest way to migrate your application to the cloud. When the application runs in the cloud, you can further optimize it to leverage all its benefits. By migrating your applications to the cloud by applying the lift and shift approach, you may realize the cost and agility benefits sooner.

Depending on the migration strategy, you typically migrate either the entire server, including the application and the infrastructure that the application is running on, or just the data that belongs to the application. Let's look at how to migrate data and servers.

Data migration

Cloud data migration refers to moving existing data to a new cloud storage location. Most applications will require data storage throughout their progression into the cloud. Storage migration typically aligns with one of two approaches, but organizations may perform both at the same time:

- First, a single lift and shift move. This may be required before new applications can be started in the cloud.
- Second, a hybrid model weighted toward the cloud, which results in newly architected cloud-native projects with some legacy on-premises data. The legacy data stores may shift toward the cloud over time.

However, your approach to migrating data will vary. It depends on factors such as the amount of data, network and bandwidth constraints, the data classification tier (such as backup data, mission-critical data, data warehouses, or archive data), the level of data security, and the amount of time you can allocate for the migration process.

Suppose you have extensive archives of data or data lakes in a situation where bandwidth and data volumes are unrealistic. In that case, you should lift and shift the data from its current location straight into a cloud provider's data center. You can do this by using dedicated network connections to accelerate network transfers or by physically transferring the data over the hard drive.

If your data stores can gradually migrate over time or when new data is aggregating from many non-cloud sources, consider methods that provide a friendly interface to the cloud storage service. These migration services can leverage or complement existing installations, such as backup and recovery software or a **storage area network (SAN)**.

For a small-scale database, one-step migration is the best option, which requires you to shut down the application for a couple of hours to a few days as per the complexity of the workload. During the downtime, all information from the database is extracted and migrated to the destination database in the cloud. Once the database has been migrated, it must be validated with the source database for no data loss. After that, a final cutover can be completed.

In the opposite case, if a system requires minimal downtime, a two-step migration process is more commonly used for databases of any size:

1. Information is extracted from the source database.
2. Data is migrated while the database runs. You can configure **change data capture (CDC)** to ensure all data is migrated and the application works during migration.

In the entire process, there is no downtime. After completing the migration task, you can perform functionality and performance tests for connectivity to external applications or any other criteria as needed.

During this time, as the source database is still up and running, changes will need to be propagated or replicated before the final cutover. At this point, you would schedule downtime for the database, usually a few hours, and synchronize the source and destination databases. After all the change data has been transferred to the target database, you should perform data validation to ensure a successful migration and route application traffic to a new cloud database.

You might have mission-critical databases that cannot have any downtime. Performing such zero-downtime migrations requires detailed planning and the appropriate data replication tools. You will need to use continuous data replication tools for such scenarios, such as AWS DataSync, Oracle GoldenGate, or NetApp SnapMirror. It is important to note here that source database latency can be impacted in the case of synchronous replication as it waits for data to be replicated everywhere before responding to the application while the replication is happening.

You can use asynchronous replication if your database downtime lasts only a few minutes. With zero-downtime migration, you have more flexibility regarding when to perform the cutover since the source and target databases are always in sync.

Server migration

There are several methods you can use to migrate a server to the cloud:

- The **host or OS cloning** technique involves installing an agent on the source system that will clone the OS image of the system. A snapshot is created on the source system and sent to the target system. This type of cloning is used for a one-time migration.
- With the **OS copy** method, all OS files are copied from the source machine and hosted on a cloud instance. For the OS copy method to be effective, the people and/or tool that executes the migration must understand the underlying OS environment.
- The **disaster recovery** replication technique deploys an agent on the source system to replicate data to the target. However, the data is replicated at the file system or block level. A few solutions continuously replicate the data to target volumes, offering a continuous data replication solution.
- The disk volume is copied in its entirety with the **disk copy** method. Once the disk volume has been captured, it can be loaded into the cloud as volumes, which can then be attached to a cloud instance.
- For VMs, you could use agentless techniques to export/import your VM into the cloud. With the **VM copy** method, the on-premises VM image is copied. If the on-premises servers are running as VMs, such as VMware or OpenStack, then you can copy the VM image and import it into the cloud as a machine image. The main benefit of this technique is having server backup images that can be launched repeatedly.
- Only the application's user data is copied with the **user data copy** method. Once the data has been exported from the original server, you can choose one of three migration strategies—*repurchase, replatform, or refactor*. The user data copy method is only viable for those who know the application's internals. However, because it only extracts user data, the user data copy method is an OS-agnostic technique.
- You can containerize your application and then redeploy it in the cloud. The **containerization method** copies both the application binary and user data. Once the application binary and user data have been copied, it can be run on a container runtime that is hosted on the cloud. Because the underlying platform is different, this is an example of the platform migration strategy.

Several migration tools in the market can help you migrate your data and/or server to the cloud. Each major public cloud provides its own tool for migration; however, you can also use other popular cloud migration tools such as **CloudEndure**, **NetApp**, **Dynatrace**, **Carbonite**, **OpenText**, and so on. Some tools take a disaster recovery strategy for migration, and some disaster recovery tools also support continuous replication to facilitate live migrations. Some specialize in lifting and shifting your servers, performing database migrations across platforms, or database schema conversion. The tool must support business processes that you are comfortable with and have the operational staff to manage it.

Integrating, validating, and cutover

Migration, integration, and validation go hand in hand, as you want continuous validation while performing various integrations with your application in the cloud.

Validation

The team starts by performing the necessary cloud functionality checks to ensure that the application is running with proper network configuration (in the desired geolocation) with some designated traffic flow. Instances can start or stop as desired when the basic cloud functionality check is complete. It's advisable to validate that the server configuration (such as RAM, CPU, and hard disk) is the same as intended.

Some knowledge of the application and its functionality is required to perform these checks. When the primary check is complete, you can perform integration tests for the application.

Integration

Integration tests include checking integration with external dependencies and applications, such as ensuring the application can connect to Active Directory, CRM services, patch or configuration management servers, and shared services. For example, your application might have to communicate with an Active Directory server, a configuration management server, or shared services resources that are all external to the application. Your application may also need to be integrated with external applications that belong to your clients or vendors, such as a supplier receiving a feed from your APIs after a purchase order placement.

When the integration process is complete, you need to validate the integration by performing unit tests, smoke tests, and **user acceptance tests (UATs)**. The results from these tests help you get approval from the application and business owners.

The final step of the integration and validation phase includes a sign-off process from the application and the business owner of the application, which will allow you to cut over the application from on-premises to the cloud.

The cutover process

The next phase of cloud migration is the **cutover process**. In this phase, you take the necessary steps to redirect your application traffic from the source on-premises environment to the target cloud environment. Depending on the type of data or server migration (one-step, two-step, or zero-downtime migration), the steps in your cutover process may vary. Some factors to consider when determining a cutover strategy include the following:

- Acceptable downtime for the application
- The frequency of the data update
- Data access patterns such as read-only or static data
- Application-specific requirements such as database syncs, backups, and DNS name resolutions
- Business constraints, such as the day or time during which the cutover can happen and the criticality of the data
- Changing management guidelines and approvals

Live migration is most popular for business-critical workload migration. Let's learn more about it.

Live migration cutover

In this method, the data is continuously replicated to the destination, and you perform most of the functional validation and integration testing at the destination while the application is still up and running. The following diagram illustrates a cutover strategy for live zero-downtime migration.

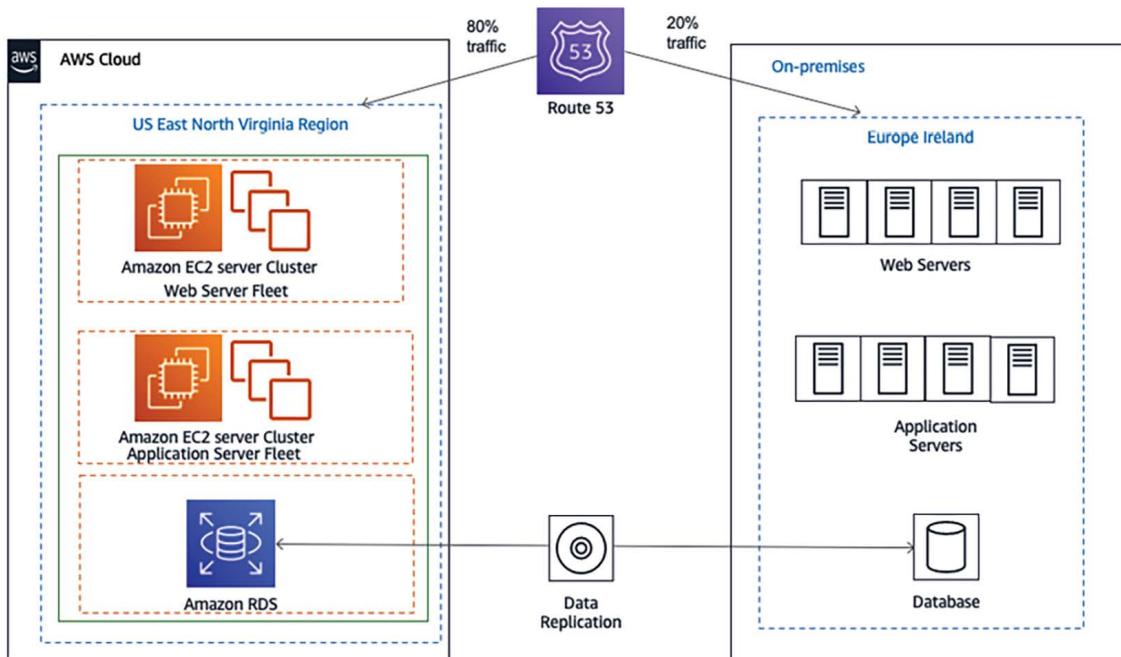


Figure 3.7: Live migration cutover using blue-green deployment

The preceding diagram depicts a hybrid cloud architecture used in a blue-green deployment strategy for a live migration cutover.

 The idea behind blue-green deployment is that your blue environment is your existing production environment carrying live traffic. In parallel, you provision a green environment, which is identical to the blue environment other than the new version of your code. You will learn more about blue-green deployments in *Chapter 11, DevOps and Solution Architecture Framework*.

Here's how it works in the context of the diagram:

- Current setup (blue environment):** The on-premises data center, located in Europe (Ireland), consists of web servers, application servers, and a database. It handles a certain percentage of user traffic (20%, as indicated).
- Target setup (green environment):** The AWS cloud setup in the US East (North Virginia) region is the new environment being prepared to take over the entire traffic. This includes a cluster of Amazon EC2 instances for the web server fleet and another cluster for the application server fleet. Amazon RDS is used for the database.

3. **Traffic routing and distribution:** Amazon Route 53, a DNS service, is used to route user traffic between the on-premises and AWS cloud environments. Initially, it is configured to send the majority of traffic (80%) to the AWS cloud environment, while the remaining traffic still goes to the on-premises data center.
4. **Data replication:** Data replication is continuously happening from the on-premises database to Amazon RDS in the AWS cloud to ensure data consistency and up-to-date information in the cloud environment.
5. **Live migration cutover:** During the cutover phase in a blue-green deployment, the new (green) environment in AWS is fully operational and is handling most of the traffic.
6. After thorough testing and confirmation that the new environment is stable and performing as expected, Route 53 will gradually shift 100% of the traffic away from the on-premises (blue) environment to the AWS cloud (green) environment.
7. The on-premises environment remains on standby during this phase. If any critical issues arise in the AWS cloud setup, traffic can be rerouted back to the on-premises servers to ensure service continuity.
8. **Completion:** Once the cutover is successfully completed and the AWS cloud environment is handling all the traffic, the on-premises infrastructure can be decommissioned or repurposed as needed.

This approach minimizes downtime and risk because the new environment is fully tested with live traffic before the old environment is decommissioned. It also provides an easy rollback strategy if issues arise during the cutover.

Initially, the application continues to run both on-premises and in the cloud, distributing traffic between the two sides. You can gradually increase traffic to cloud applications until all the traffic is directed to the new application, thus resulting in a cutover with no downtime.

The other most commonly used cutover strategies involve some downtime. You schedule downtime for the application, pause the traffic, take the application offline, and perform the final sync by applying the CDC process.

After the final sync, performing a quick smoke test on the destination side might be a good idea to check that all critical features are working as desired. At this point, you can redirect the traffic from the source to the application running in the cloud, thus completing the cutover.



Data is critical to sync and cut over during migration as it changes continuously when an application is live. You can use data migration tools such as **AWS Database Migration Service (DMS)** and Oracle GoldenGate to perform one-time data migration of CDC data.

Operating the cloud application

The operation phase of the migration process helps you allow, run, use, and operate applications in the cloud to the level agreed upon with the business stakeholders.

Most organizations typically already have guidelines defined for their on-premises environments. This operational excellence procedure will help you identify the process changes and training that will allow operations to support cloud adoption goals.

The following are the IT operations that you would want to address in the cloud:

- Server patching
- Service and application logging
- Cloud monitoring
- Event management
- Cloud security operations
- Configuration management
- Cloud asset management
- Change management
- Business continuity with disaster recovery and high availability

IT organizations typically follow standards such as **Information Technology Infrastructure Library (ITIL)** and **Information Technology Service Management (ITSM)** for most of these operations. ITSM organizes and describes the activities and processes involved in planning, creating, managing, and supporting IT services, while ITIL applies best practices to implement ITSM. You need to modernize your ITSM practices to take advantage of the agility, security, and cost benefits provided by the cloud.

Your job is not over after just migrating to the cloud; continuous optimization is required to utilize the full potential of the cloud. Let's learn more about it.

Application optimization in the cloud

Optimization is an essential aspect of operating in the cloud, and this is a continuous process of improvement. In this section, you will learn about the various optimization areas. There are chapters dedicated to each optimization consideration in this book. The following are the major optimization areas:

- **Performance:** Optimize for performance to ensure that a system is architected to deliver efficient performance for a set of resources, such as instances, storage, databases, and space/time. You will learn more about architecture performance considerations in *Chapter 6, Performance Considerations*.
- **Security:** Continuously review and improve security policies and processes for the organization to protect data and assets in the cloud. You will learn more about architecture security considerations in *Chapter 7, Security Considerations*.
- **Reliability:** Optimize applications for reliability to achieve high availability and defined downtime thresholds for applications, which will aid in recovering from failures, handling increased demand, and mitigating disruptions over time. You will learn more about architecture reliability considerations in *Chapter 8, Architectural Reliability Considerations*.

- **Operational excellence:** Optimize operational efficiency and the ability to run and monitor systems to deliver business value and continually improve supporting processes and procedures. *Chapter 9, Operational Excellence Considerations*, will teach you more about architecture operational considerations.
- **Cost:** Optimize the cost efficiency of an application or a group of applications while considering fluctuating resource needs. You will learn more about architecture cost considerations in *Chapter 10, Cost Considerations*.

As a quick overview of some of the major elements to consider, you need to understand what is currently being deployed in your cloud environment and the price of each of those resources to optimize costs. You can proactively monitor your costs in the cloud by using detailed billing reports and enabling billing alerts.

You need to maintain and scale the infrastructure and pay for less as you offload more. Another way to optimize costs is by designing your architecture for *elasticity*. Make sure you right-size your resources, use auto-scaling, and adjust your utilization based on price and need. For example, it might be more cost-efficient for an application to use more small instances than fewer large instances.

Several application architectural modifications can help you improve the performance of your application. One way to improve the performance of your web servers is to offload your web page through caching. You can write an application that lets you cache images, JavaScript, or even entire pages to provide a better experience to your users.

You can design n-tier and service-oriented architectures to scale each layer and module independently, which will help optimize performance. *Chapter 4, Solution Architecture Design Patterns*, will teach you more about this architectural pattern.

Customers may want to retain an on-premises workload during cloud migration due to a phased approach or inability to migrate to the cloud due to application complexity or licensing issues. In such scenarios, you must build a hybrid cloud where the on-premises workload can interact with the cloud workload and exchange information seamlessly. Let's learn about creating a hybrid cloud architecture in more detail.

Creating a hybrid cloud architecture

The value of the cloud is growing, and many large enterprises are moving their workload to the cloud. However, often, it's not possible to entirely move to the cloud on day one, and for most customers, this is a journey. Those customers seek a hybrid cloud model where they maintain a part of the application in an on-premises environment that needs to communicate with the cloud module.

In a hybrid deployment, you must establish connectivity between the resources running in the on-premises and cloud environments. The most common method of hybrid deployment is between the cloud and existing on-premises infrastructure to extend and grow an organization's infrastructure into the cloud while connecting cloud resources to the internal system. The common reasons for setting up a hybrid cloud may include the following:

- You want to have operating legacy applications in an on-premises environment while you refactor and deploy in the cloud with a blue-green deployment model.
- A legacy application such as a mainframe may not have a compatible cloud option and has to continue running on-premises. It would be best if you had time to refactor the tech stack.
- You need to keep part of the application on-premises due to compliance requirements.
- To speed up migration, you want to keep the database on-premises and move the application server to the cloud.
- You want to have more granular control of part of the application.
- You want to ingest data in the cloud from on-premises for analytics.

Public cloud vendors provide a mechanism for integrations between a customer's existing infrastructure and the cloud so that customers can easily use the cloud as a seamless extension of their current infrastructure investments. These hybrid architecture functionalities allow customers to do everything, from integrating networking, security, and access control to powering automated workload migrations and controlling the cloud from their on-premises infrastructure management tools.

As shown in the following diagram, with AWS Direct Connect, you can establish high-speed connectivity between your data center and the AWS cloud to achieve a low-latency hybrid deployment:

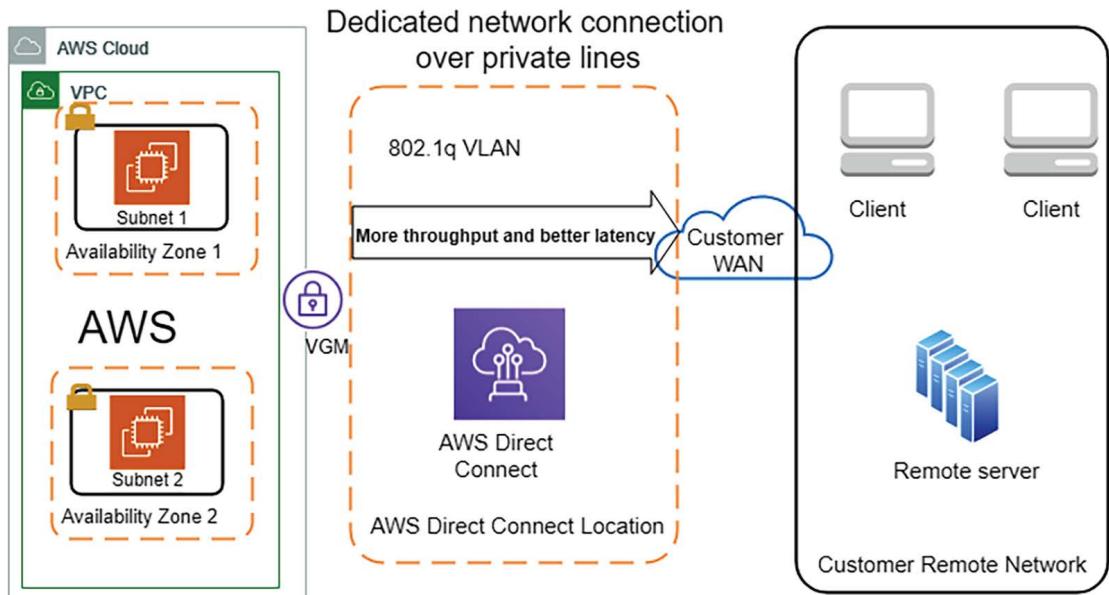


Figure 3.8: Hybrid cloud architecture (on-premises-to-cloud connectivity)



In the diagram, VPC refers to Amazon Virtual Private Cloud. A VLAN is a virtual local area network, VGM is Virtual Private Gateway and WAN is a wide area network.

As shown in the preceding diagram, the **AWS Direct Connect location** establishes the connectivity between the on-premises data center and the AWS cloud. This helps you achieve the customer need of having dedicated fiber-optic lines to an AWS Direct Connect location; the customer can opt for this fiber-optic line from a third-party vendor such as AT&T, Verizon, T-Mobile, or Comcast in the USA. AWS has a Direct Connect Partner in each region of the world.

At the AWS Direct Connect location, the customer's fiber-optic line is connected to an AWS private network, which provides dedicated end-to-end connectivity from the data center to the AWS cloud. These optic lines can provide speeds of up to 10 GB/s. To secure traffic over a direct connection, you can set up a VPN to apply IPSec encryption to the traffic flow.

To effectively balance the risks and benefits of a hybrid cloud model, a comprehensive evaluation is necessary.

The benefits of a hybrid cloud model include:

- **Flexibility and control:** Hybrid clouds offer the ability to leverage the scalability of public clouds while retaining critical workloads on-premises for better control and performance.
- **Scalability:** Businesses can scale their IT resources on-demand, ensuring that they can handle peak loads without the need for significant capital investment in physical infrastructure.
- **Enhanced resilience:** By distributing resources across multiple environments, a hybrid cloud strategy can improve overall system resilience and business continuity.
- **Innovation and experimentation:** A hybrid cloud model enables organizations to test new cloud technologies and services without disrupting core business applications that remain on-premises.

However, there are also risks, including:

- **Complexity:** A hybrid cloud environment is inherently complex, requiring sophisticated orchestration and networking capabilities to manage workloads across multiple platforms seamlessly.
- **Security concerns:** The increased surface area for potential attacks (due to the interconnection of public and private clouds) demands stringent security measures and constant vigilance.
- **Compliance challenges:** Adhering to regulatory standards becomes more challenging when data and applications are spread across various cloud environments.
- **Cost management:** Without careful planning and monitoring, the operational costs of a hybrid cloud can quickly surpass budgeted expectations due to underutilized resources or shadow IT.

The decision to implement a hybrid cloud strategy should weigh these risks against the potential benefits, aligning with the organization's specific needs, capabilities, and strategic objectives.

As more cloud offerings become available in the market from prominent vendors, organizations may choose to take a multi-cloud approach. Let's learn about the multi-cloud strategy next.

Taking a multi-cloud approach

Before the cloud existed, organizations used multiple vendors to use the best of the breed and avoid vendor lock-in. As more public cloud players enter the market, organizations seek to create a multi-cloud approach. A multi-cloud approach utilizes two or more public cloud providers to serve organization infrastructure and technology needs. The multi-cloud strategy could be a mix of major public cloud providers such as AWS, GCP, Microsoft Azure, Oracle Cloud, IBM, and so on. Organizations can share their workload between clouds based on geographical availability, technical capabilities, and cost. They can also combine the multi-cloud approach with on-premises.

Some of the significant advantages of adopting a multi-cloud strategy are as follows:

- **Vendor flexibility:** With multi-cloud, you can choose between vendors and retain your negotiation power, agility, and flexibility. In the event of a missed **service-level agreement (SLA)**, you can switch to a better cloud provider.
- **Disaster recovery:** Another advantage is planning disaster recovery in the same region when one cloud provider has an outage; you can rely on other providers. Each cloud provider has its strengths, and you can pick the best services available across the cloud.

While the multi-cloud approach provides a competitive advantage to organizations, it also comes with challenges:

- One of the most prominent challenges is skill sets. It would help if you had people who understand multiple clouds while creating a workload hosting strategy and, more than that, you need to replicate teams to dive deep into each cloud tech stack. Consider hiring a consultant or outsourcing your cloud management to global system integrators with a pool of human resources across the cloud.
- The other major challenge is coordinating data availability, security, and performance across multiple clouds. While each cloud vendor provides built-in security, cross-region applications, and cloud-native tools for performance, this area becomes the organization's responsibility regarding the cloud. You need to implement consistent data management across the cloud, taking data from one cloud, feeding it to another, and ensuring consistent performance.

As you have seen, the multi-cloud approach has advantages and disadvantages that you need to consider when choosing such a strategy.

You can build cloud-native applications once you have started your cloud journey. Let's learn more about building cloud-native architecture.

Implementing CloudOps

The cloud operations model, called CloudOps, is a framework of rules and guidelines that organizations establish, monitor, and adjust to manage costs, enhance efficiency, and mitigate security risks in their cloud infrastructure, security, and operations. This operational model is a guiding principle for aligning people, processes, and technology with cloud-related tasks, including security, budget management, and compliance across cloud workloads.

The CloudOps model provides several key benefits:

- **Unlocking speed and agility:** Organizations can harness the agility and rapid response capabilities inherent in cloud services, accelerating cloud adoption and application modernization efforts as part of their digital transformation journey.
- **Leveraging automation for efficiency:** Automation reduces manual errors and interventions by streamlining routine tasks, freeing up valuable resources and time.
- **Consistent governance at scale:** Cloud governance is uniformly maintained across different environments, ensuring standardization and compliance throughout the organization.
- **Effective utilization of skilled personnel:** CloudOps allows skilled personnel to focus on delivering business outcomes rather than repetitive, manual tasks.
- **Avoiding cost overruns:** By utilizing automation and governance, organizations can avoid unexpected cost overruns and optimize cloud spending.

Effective management and governance are essential for businesses transitioning to the cloud to uphold best practices in cloud IT operations. Cloud management and governance services offer faster innovation and robust control over cost, compliance, and security.

Cloud automation plays a pivotal role in enabling organizations to develop efficient cloud operations models by automating the creation, modification, and removal of cloud resources. The concept of on-demand cloud services promises flexibility but, in reality, many organizations still rely on manual processes to provision, test, identify the need for, and decommission cloud resources. These manual workflows can lead to labor-intensive tasks, potential errors, and increased costs.

Cloud automation may require initial effort but the benefits become evident as complex tasks can be performed swiftly with just a few clicks. Beyond the apparent reduction in manual work, cloud automation offers additional advantages, including:

- **Improved security and resiliency:** Automation helps enhance security by minimizing human error and ensuring proper implementation of security measures, such as setting up security credentials for newly added development environments. Additionally, automation enables automated recovery in case of capacity-related issues, ensuring resiliency and avoiding downtime.
- **Streamlined backup processes:** Automated backups ensure business continuity and customer trust during disaster recovery events, minimizing business losses. Automated backups eliminate the reliance on individuals to initiate backups, reducing the risk of data loss.
- **Enhanced governance:** Automation ensures comprehensive monitoring of activities across environments, providing better governance by tracking servers, databases, and access controls effectively.

Cloud providers offer a range of services and third-party tools to support modern enterprises in implementing the CloudOps model, facilitating innovation, improved application performance, and faster response times to customer feedback, all while maintaining governance and compliance.

The CloudOps model encompasses several pillars that cover various aspects of IT workload automation. Let's look at these next.

CloudOps pillars

In the process of planning your CloudOps strategy, it is crucial to adopt a comprehensive 360-degree perspective. The goal is to provision and manage your cloud environment with a focus on business agility and governance control. Establishing a robust CloudOps model, irrespective of your cloud migration journey, allows you to achieve consistent governance and streamlined operations across diverse infrastructure environments. This strategic approach enables you to optimize critical resources, leading to accelerated delivery of business outcomes, faster time-to-market, and improved safety, efficiency, and cost control.

The diagram below illustrates the key pillars of the CloudOps model, encompassing complete IT workload automation coverage.

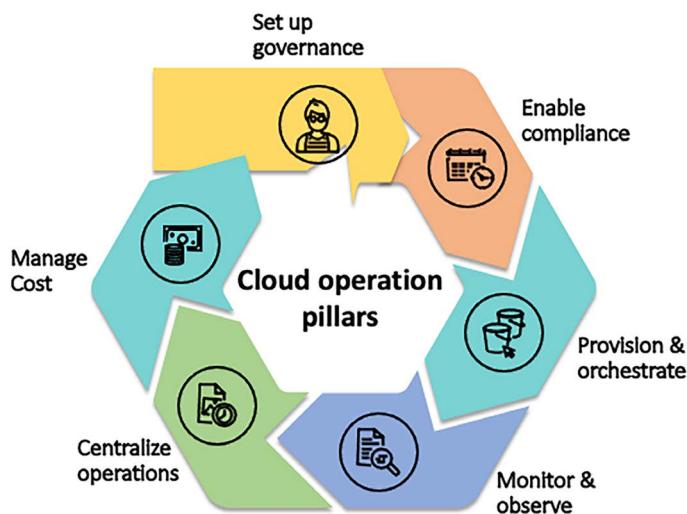


Figure 3.9: CloudOps pillars

The following are the fundamental pillars of CloudOps, as depicted in the diagram:

- Set up governance:** Create a robust, well-architected, and multi-account cloud environment with guardrails to form the foundation for governance. Adhering to the Well-architected Framework checklist ensures comprehensive monitoring and alerting mechanisms for security, operational excellence, cost optimization, reliability, and performance.
- Enable compliance:** Continuously monitor compliance and configurations for your cloud resources, promptly remediate failures using automated methods, and gather evidence for audits to maintain adherence to industry regulations and internal policies.
- Provision and orchestrate:** Accelerate application and resource provisioning using Infrastructure-as-Code (IaC) principles while maintaining consistency and compliance throughout your cloud environment.
- Monitor and observe:** Effectively measure and manage your applications and cloud resources, swiftly identifying and resolving issues to ensure optimal performance and reliability.

5. **Centralize operations:** Streamline and automate operational actions across your entire application portfolio, ensuring seamless execution while upholding safety, security, and compliance standards.
6. **Manage cost:** Transform your business operations by implementing cost transparency, control measures, forecasting capabilities, and optimization strategies to optimize cloud spending.

You can learn about each pillar in detail by referring to our other book: *AWS for Solutions Architects: The Definitive Guide to AWS Solutions Architecture for Migrating to, Building, Scaling, and Succeeding in the Cloud, 2nd Edition*.

Summary

In this chapter, you explored the fundamental aspects of solution architecture in the public cloud. You also learned about cloud-native and hybrid architecture, gaining a comprehensive understanding of cloud computing and its benefits.

We began with a comparison between public, private, and hybrid clouds, helping you to grasp the different cloud deployment models and their respective use cases.

We then defined the public cloud concept in more detail through its architecture and introduced some popular public cloud providers.

Furthermore, you delved into cloud-native architecture, gaining insights into the advantages of adopting a cloud-native architecture, such as enhanced scalability, flexibility, and cost-efficiency.

With a solid public cloud fundamentals foundation, we discussed creating a cloud migration strategy. Different migration approaches were explored in detail, including lift and shift, rehost, replatform, relocate, refactor, repurchase, retain, and retire. You explored guidance on choosing the most suitable cloud migration strategy based on business requirements.

The subsequent section outlined the essential steps for cloud migration, starting with workload discovery and analysis. You learned how to create a comprehensive migration plan and design the application for seamless migration to the cloud. Furthermore, we covered the critical aspects of application migration, including data migration, server migration, integration, validation, and cutover, and you were introduced to application optimization techniques in the cloud to ensure optimal performance and cost-effectiveness.

As organizations often deal with complex infrastructures, you learned how to create a hybrid cloud architecture and adopt a multi-cloud approach to leverage the best of multiple cloud providers.

The chapter culminated with a focus on designing a cloud-native architecture, emphasizing the principles of CloudOps, which help to operationalize your cloud workload.

In the next chapter, you will dive deep into various architectural design patterns and reference architectures, such as multi-tier, service-oriented, serverless, and microservices.

Further reading

To learn more about the major public cloud providers, please refer to the following links:

- **Amazon Web Services (<https://aws.amazon.com>):**
 - *AWS for Solutions Architects* by Saurabh Srivastava, Neelanjali Srivastav, Alberto Arta-sanchez, and Imtiaz Sayed: <https://www.amazon.com/gp/product/180323895X>
 - *AWS Well-Architected Framework*: <https://aws.amazon.com/architecture/well-architected/>
- **Google Cloud Platform (<https://cloud.google.com>):**
 - *GCP Cloud Architecture Framework*: <https://cloud.google.com/architecture/framework>
- **Microsoft Azure (<https://azure.microsoft.com>):**
 - *Azure Well-Architected*: <https://azure.microsoft.com/en-us/solutions/cloud-enablement/well-architected#reliability>
- **Oracle Cloud Infrastructure (OCI)**: <https://www.oracle.com/cloud/>
- **Alibaba Cloud**: <https://us.alibabacloud.com>
- **IBM Cloud**: <https://www.ibm.com/cloud>

Almost every cloud provider extends the chance to learn to new users, which means you can sign up with your email and try their offerings out before you choose which one to go with.

Join our book's Discord space

Join the book's Discord workspace to ask questions and interact with the authors and other solution architecture professionals: <https://packt.link/SAHandbook>



4

Solution Architecture Design Patterns

Have you ever wondered how large enterprises design scalable systems? Before starting application development, solutions architects work across organizations and weigh multiple options to develop architecture designs to handle their business needs.

There are multiple ways to design a solution. A solutions architect needs to take the right approach based on user requirements and the architecture constraints of cost, performance, scalability, and availability. In this chapter, you will learn about various solution architecture patterns, reference architectures, and how to apply them in real-world scenarios.

In the previous chapters, you learned about the principles of solution architecture design. This chapter is exciting and essential as you can apply your learning to various architectural design patterns. In this chapter, you will gain an understanding of some of the significant solution architecture patterns, such as layered, event-driven, microservice, loosely coupled, service-oriented, and RESTful architectures.

You will learn about the advantages of various architectural designs and look at examples demonstrating when to utilize them. You will also gain an understanding of architecture design anti-patterns in addition to the following architecture design patterns:

- Building an n-tier layered architecture
- Creating a multi-tenant SaaS-based architecture
- Understanding service-oriented architecture
- RESTful web service architecture
- Building a cache-based architecture
- **Model-View-Controller (MVC) architecture**
- Building **Domain-Driven Design (DDD)**
- Understanding the circuit breaker pattern
- Implementing the bulkhead pattern

- Creating a floating IP pattern
- Deploying an application with a container
- Database handling in application architecture
- Clean Architecture
- Avoiding anti-patterns in solution architecture

By the end of the chapter, you will know how to optimize your solution architecture design and apply best practices, making this chapter the center point and core of your learning.

Building an n-tier layered architecture

In *n*-tier architecture (also known as **multitier architecture**), you need to apply loosely coupled design principles and attributes of scalability and elasticity. In *n*-tier architecture, you divide your product functions into multiple layers, such as presentation, business, database, and services, so that each layer can be implemented and scaled independently.

With *n*-tier architecture, adopting new technologies and making development more efficient is easy. This layered architecture provides the flexibility to add new features in each layer without disturbing the features of other layers. In terms of security, you can keep each layer secure and isolated from the others, so if one layer gets compromised, the other layers won't be impacted. Application troubleshooting and management also become manageable as you can quickly pinpoint where an issue is coming from and which part of the application needs to be troubleshooted.

The most common architecture in multilayer design is **three-tier architecture**, so let's learn more about it. The following diagram shows an AWS example architecture that allows you to interact with a web application from the browser and perform the required functions, for example, ordering your favorite T-shirt or reading a blog and leaving a comment:

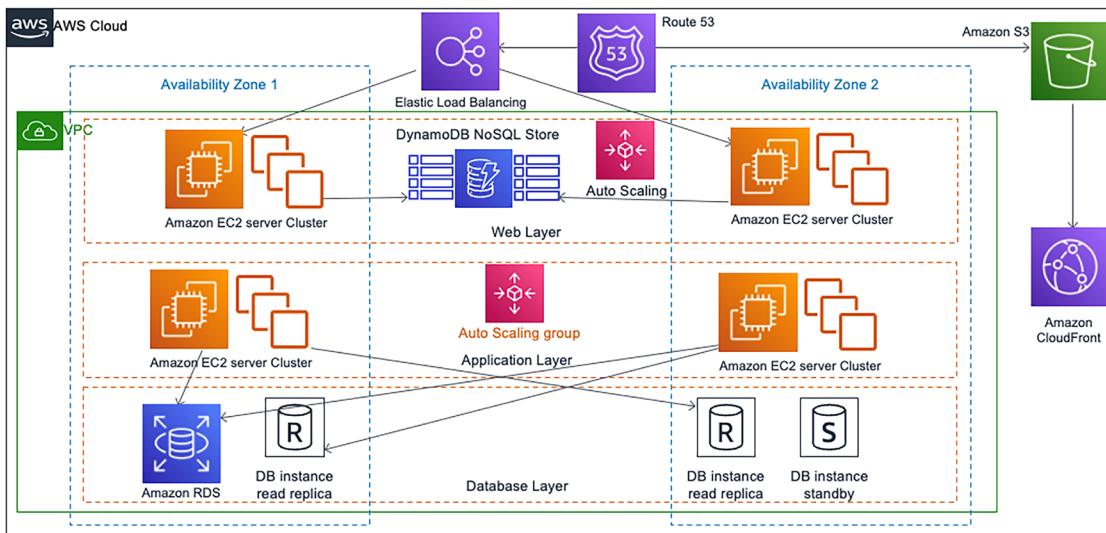


Figure 4.1: Three-tier website architecture

In the preceding architecture, you have the following three layers:

- **Web Layer:** The web layer is the user-facing part of the application. End users interact with the web layer to collect or provide information.
- **Application Layer:** The application layer mainly contains business logic and acts upon information received from the web layer.
- **Database Layer:** All kinds of user data and application data are stored in the database layer.

Let's take a look at these layers in more detail.

The web layer

The web layer is also known as the **presentation tier**. The web layer provides a user interface that helps the end user to interact with the application. The web layer is your user interface (in this case, the website page), where the user enters information or browses it. Web developers may build a presentation tier user interface in technologies such as HTML, CSS, Angular, React, JavaServer Pages (JSP), and Active Server Pages (ASP). This tier collects the information from the user and passes it to the application layer.

The web layer is user-facing, so organizations spend most of their time improving the user experience. Many organizations have dedicated **User Experience (UX)** teams researching various areas to understand how users interact with applications.

Also, the solutions architect must ensure the architecture design includes UX input and page load performance. There should be a seamless information flow between the web and application layers to return the correct information to users within the expected timeframe, such as user login, profile loading, etc.

Let's look at the application layer.

The application layer

The application layer is also known as the **logic tier**, as this is the core of the product where all the business logic resides. The presentation tier collects the information from the user and passes it to the logic tier to process it and get a result. For example, on an e-commerce website such as Amazon, users can enter a date range on the website's order page to find their order summary. In return, the web layer passes the data range information to the application layer. The application layer processes the user input to perform business logic, such as the count of orders, the sum of amounts, and the number of items purchased. This returns information to the web layer to render it for the user.

Generally, in a three-tier architecture, all algorithms and complex logic live in the application tier, including creating a recommendation engine or showing personalized pages to the user as per their browsing history. You may add layers such as a domain layer, data access layer, or presentation layer to make a four- or five-tier architecture. Developers may choose to implement this layer in a server-side programming language, for example, C++, Java, .NET, or Node.js. The application layer is the center of system design and requires most of the design effort. Most of the application features depend on logic built at the application layer. The application layer performs logic on the data stored in the database layer. Let's look at the database layer in more detail.

The database layer

The database layer, also known as the **data tier**, stores all the information related to user profiles and transactions. Essentially, it contains any data that needs to persist in being stored in the data tier. This information is sent back to the application layer for logic processing, and then, eventually, is rendered to the user in the web layer. For example, suppose the user is logged in to a website with their ID and password. In that case, the application layer verifies the user credentials with information stored in the database. If the credentials match the stored information, the user is allowed to log in and access the authorized area of the website.

The architect may choose to build a data tier in relational databases, for example, PostgreSQL, MariaDB, Oracle Database, MySQL, Microsoft SQL Server, Amazon Aurora, or Amazon RDS. The architect may add a NoSQL database such as Amazon DynamoDB, MongoDB, or Apache Cassandra.

The data tier is used to store transaction information and hold user session information and application configuration. An architect may consider adding caching databases such as Memcached and Redis to meet performance needs. You will learn more about various databases in *Chapter 12, Data Engineering for Solution Architecture*.

The data tier needs special attention in terms of security. You must protect user information by applying data encryption at rest and in transit. In the n -tier layered architecture diagram, you will notice that each layer has its own auto-scaling configuration, which means it can be scaled independently. Also, each layer has a network boundary, which means having access to one layer doesn't allow access to other layers. You will learn more about security considerations in *Chapter 7, Security Considerations*.

The architect needs to decide on the number of tiers based on application complexity and user requirements. For example, you might add additional tiers, such as a data access layer for database access logic, and keep the data storage layer for the database engine. You can add more layers to reduce complexity by defining logical separation, which can help increase the application's general maintainability and the ability to scale and achieve performance.

Creating a multi-tenant SaaS-based architecture

In the previous section, you learned about multtier architecture, also called a **single tenancy**, when built for a single organization. Multi-tenant architecture is becoming more popular as organizations welcome the digital revolution while keeping the overall application and operational cost low.

The **Software-as-a-Service (SaaS)** model is constructed upon a multi-tenant architecture, where an instance of the software and the accompanying infrastructure caters to numerous customers. Within this framework, each customer utilizes the application and database in a shared manner. With each tenant being isolated by their unique configuration, identity, and data, they remain invisible to each other while sharing the same product.

As multi-tenant SaaS providers are responsible for everything from the hardware to the software, SaaS-based products offload an organization's responsibilities to the application's maintenance and updates, as the SaaS provider takes care of this.

Each organization that is a buyer of the SaaS product is considered a tenant. These tenants can customize their user interface using a configuration without code changes. As multiple customers share a common infrastructure, they benefit from scale, which further lowers the cost. Some of the most popular SaaS providers are Salesforce CRM, Jira Software, Slack, Google Workspace, and Amazon Connect.

As shown in the following architecture diagram, two organizations (tenants) use the same software and infrastructure. The SaaS vendor provides access to the application layer by allocating a unique tenant ID to each organization:

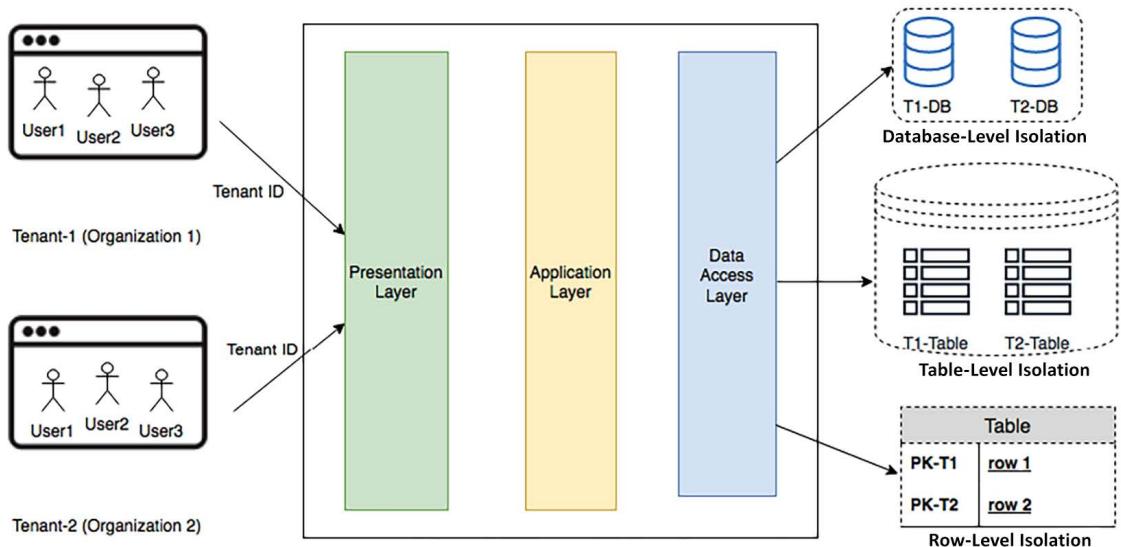


Figure 4.2: Multi-tenant SaaS architecture

The preceding architecture design shows that the presentation layer provides a user interface, and the application layer holds the business logic. At the data access layer, each tenant will have data-level isolation with one of the following methods:

- **Database-Level Isolation:** In this model, each tenant has its database associated with its tenant ID. When each tenant queries data from the user interface, they are redirected to their database. This model is required if the customer doesn't want a single shared database for compliance and security reasons.
- **Table-Level Isolation:** This isolation level can be achieved by providing a separate table for each tenant. In this model, tables need to be uniquely assigned to each tenant, for example, with the tenant ID prefix. When each tenant queries data from the user interface, they are redirected to their tables as per their unique identifier.
- **Row-Level Isolation:** All tenants share the same table in a database in this isolation level. There is an additional column in a table where a unique tenant ID is stored against each row. When an individual tenant wants to access their data from the user interface, the application's data access layer formulates a query based on the tenant ID to the shared table. Each tenant gets a row that belongs to their users only.



For enterprise customers, a careful assessment should be carried out to understand whether a SaaS solution fits them based on their unique features' requirements. This is because, often, a SaaS model has limited customization capabilities.

The selection of isolation approaches is based on considerations around the organization's compliance, security, cost, and tenant contractual requirements.

It's important to find the cost value proposition if many users need to subscribe. The cost comparison should be calculated based on the total cost of ownership when making a *build versus buy* decision. This is because building software is not the primary business of most organizations, so the SaaS model is becoming highly popular as organizations can focus on their business and let the experts handle the IT side of it.

Service-oriented architecture (SOA) is a popular approach for designing and building applications, especially when organizations have unique, customized requirements. Let's learn more about it.

Understanding service-oriented architecture

In SOA patterns, different application components interact using a communication protocol over the network. Each service provides end-to-end functionality, for example, *fetching an order history*. SOA is widely adopted by large systems to integrate business processes, for example, taking your payment service from the main application and putting it as a separate solution.

In a general sense, SOAs take monolithic applications and spread some of those operations into individual *services* that operate independently. The goal of using an SOA is to loosen the coupling of your application's services. Sometimes, an SOA includes splitting services apart from one another and splitting resources into separate instances of that service. For example, while some choose to store their company's data in a single database split by tables, an SOA would consider modularizing the application by function into separate databases altogether. This allows you to scale and manage throughput based on the individual needs of tables for each database.

SOA has multiple benefits, for example, the parallelization of development, deployment, and operation. It decouples the service so that you can optimize and scale each service individually.

However, it also requires more robust governance to ensure work performed by each service's team meets the same standard. With SOA, the solution could become complex enough to increase the overhead to balance that, so you need to choose tools and implement automation for service monitoring, deployment, and scaling.

There are multiple ways to implement SOA.

Simple Object Access Protocol (SOAP) was initially the most popular messaging protocol, but it is heavyweight as it relies entirely on XML for data interchange. **Representational State Transfer (REST)** architecture is becoming more popular as developers need to build more lightweight mobile and web applications. As of the time of writing, SOAP architecture is considered legacy architecture, so in this edition of the book, we will focus on learning REST architecture.

RESTful web service architecture

A REST or RESTful web service offers better performance due to its lightweight architecture. It allows different messaging formats, such as JSON, plaintext, HTML, and XML, compared to SOAP, which only allows XML. REST is an architecture style that defines the standard for loosely coupled application design using the HTTP protocol for data transmission.

JavaScript Object Notation (JSON) is a more accessible format for data exchange in REST architecture. JSON is also lightweight and language-independent. It contains a simple key-value pair that makes it compatible with data structures defined in most programming languages.

RESTful web services, also known as REST web services, establish a framework with specific rules to design web services. They are aimed at ensuring compatibility between various computer systems connected through the internet. With RESTful web services, systems can communicate by accessing and modifying text-based data on the web using a consistent and predefined set of operations that don't rely on past interactions or states. Here are some fundamental principles of RESTful web service architecture, along with an illustration of the principles of RESTful web service architecture using the example of an e-commerce website:

- **Stateless:** Every request from the client to the server must contain all the information the server needs to understand and process. Every request made by a client includes all the necessary information to fulfill that request, and there is no need to maintain any session-related information on the server; instead, it is entirely managed on the client side. Taking the example of an e-commerce website, each request from a client, like viewing a product or adding it to the cart, must contain all the information needed for processing. If a user wants to view their cart, the request must include the user's ID or any other relevant details so the server can identify and respond with the appropriate card details.
- **Client-server architecture:** In this design, there are two distinct parts, the client and the server, which communicate with each other through a network. The client is responsible for managing the user interface and interacting with the user, and the server is responsible for the backend and data processing. They can evolve separately without affecting each other. The client (browser or app) manages user interaction, like selecting products, while the server handles data retrieval, cart management, and checkout processing. They interact through HTTP requests and responses.
- **Uniform interface:** REST uses a uniform interface, simplifying and decoupling the architecture. For RESTful APIs, interactions are facilitated through a set of standard HTTP methods that correspond to **CRUD (Create, Read, Update, Delete)** operations. These methods include:
 - **GET:** This method is used to retrieve data from the server. For example, when a user wants to view a list of products on example.com, their browser sends a GET request to the server. The URL might look like <https://example.com/api/products>, and the server responds with the list of products in a structured format like JSON or XML.

- **POST:** This method is used to create a new resource on the server. Suppose a user wants to add a new product to their shopping cart on `example.com`. They might fill out a form with the product details and click **Add to Cart**. This action would trigger a POST request to `https://example.com/api/cart`, including the product details in the request body. The server then processes this data and adds the new product to the user's cart.
- **PUT:** This method is used to update an existing resource on the server. If the user wants to update the quantity of a product in their cart, a PUT request would be sent to a specific URL like `https://example.com/api/cart/{productId}`. The request body would include the updated quantity, and the server would update the corresponding item in the user's cart.
- **DELETE:** This method is used to remove a resource from the server. For instance, if the user decides to remove a product from their cart, their browser would send a DELETE request to a URL like `https://example.com/api/cart/{productId}`. The server would then remove the specified product from the cart.

By adhering to these standard methods, APIs provide a consistent way for developers to interact with web services, enabling them to perform basic operations on resources without needing to understand the underlying implementation details.

- **Resource-based:** In REST, everything is considered a resource, and a specific URL can access each resource. Resources are the key abstractions in REST, and a resource can represent a single object or a collection of objects. Resources like products, users, orders, and cart items are all identified by URLs. For example, a specific product could be accessed at `www.amazon.com/products/{product_id}`.
- **Representation of resources:** Resources can have different representations, such as JSON, XML, HTML, etc. Clients interact with resources by obtaining representations of them and manipulating these representations. When a client holds a representation of a resource, it has enough information to modify the resource on the server. The same product resource might be rendered differently for a web browser than a mobile app.
- **Layered system:** The architecture allows for an intermediate layer (such as a load balancer or a cache layer) to be introduced without affecting how the client interacts with the server. Each layer can provide a specific set of functionalities, improving the scalability and maintainability of the system. An e-commerce website can have various layers, like load balancers, caching layers, or authentication layers. The client does not need to know about these layers. A request to view a product might pass through a cache layer to improve response time without the client's awareness.
- **Code on demand:** Servers can provide executable code to the client to be executed in the client's context. This allows for part of the application logic to be shifted to the client. An e-commerce website could send JavaScript code to the client's browser to execute certain functionalities like client-side validation or enhance interactivity within the user's browsing experience.

The RESTful architectural style uses standard HTTP methods, and by adhering to these principles, RESTful web services aim to be simple, scalable, and maintainable. Many modern web APIs are developed following the RESTful principles, using standard conventions to perform CRUD operations on resources. Let's learn about a reference architecture based on RESTful architecture.

Building a RESTful-architecture-based e-commerce website

An e-commerce website such as www.amazon.com has users worldwide and a vast catalog with millions of products. Each product has multiple images, reviews, and videos. Maintaining such an extensive catalog for a global user base is a very challenging task.

This reference architecture in AWS follows RESTful principles. The services are operating as independently as possible from each other:

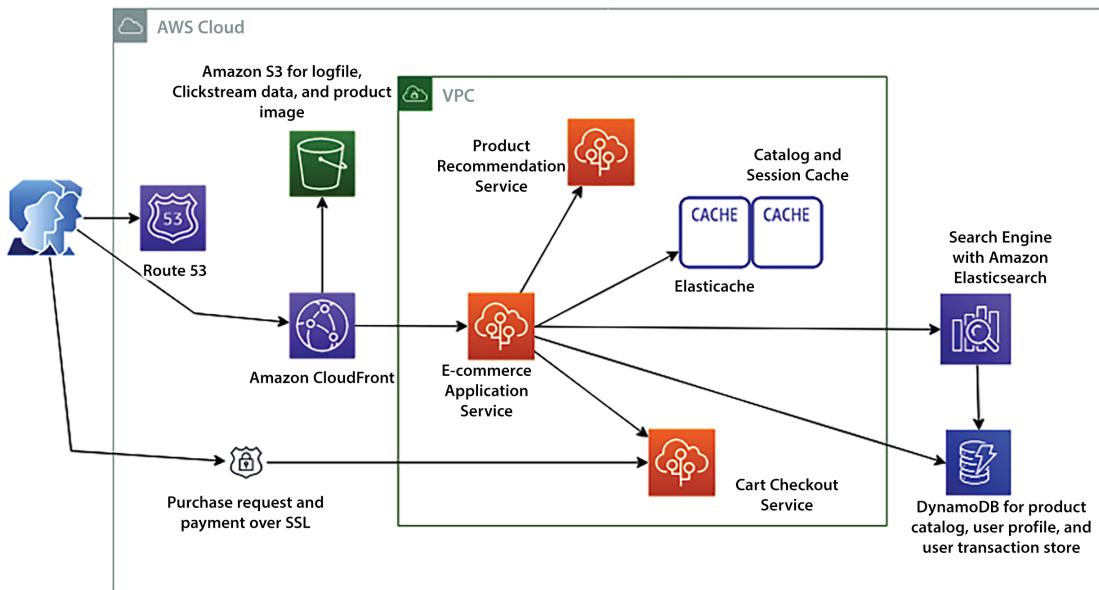


Figure 4.3: RESTful architecture for an e-commerce website

As shown in the preceding architecture diagram, we can take note of the following:

- When a user types a website address into the browser, the user request reaches out to the DNS server to load the website. The DNS requests for the website are routed by Amazon Route 53 to the server where the web applications are being hosted.
- The user base is global, and users continue browsing for products to purchase as the website has an extensive product catalog with static images and videos. A content distribution network like Amazon CloudFront caches and delivers static assets to users.
- The catalog contents, such as static product images and videos, and other application data, such as log files, are stored in Amazon S3.
- Users will browse the website from multiple devices; for example, they will add items to a cart from their mobile and then make a payment on a desktop. A persistent session store, such as DynamoDB, is required to handle user sessions. DynamoDB is a NoSQL database where you don't need to provide a fixed schema, so it is a great storage option for product catalogs and attributes.

- Amazon ElastiCache is used as a caching layer for the product to reduce read and write operations on the database to provide high performance and reduce latency.
- A convenient search feature is vital for product sales and business success. Amazon CloudSearch helps to build scalable search capability by loading the product catalog from DynamoDB. You can also use Amazon Kendra for an AI-powered search engine.
- A recommendation can encourage users to buy additional products based on browsing history and past purchases. A separate recommendation service can consume the log data stored on Amazon S3 and provide potential product recommendations to the user.
- The e-commerce application can also have multiple layers and components that require frequent deployment. AWS Elastic Beanstalk handles the auto-provisioning of the infrastructure, deploys the application, handles the load by applying auto-scaling, and monitors the application.

You learned about RESTful architecture in this section. Let's learn more about the critical aspect of cache-based architecture design.

Building a cache-based architecture

Caching involves temporarily storing data or files in an intermediate location positioned between the requester and the permanent storage. This practice is aimed at speeding up future requests and minimizing network bandwidth usage. Caching increases application speed and lowers cost. It allows you to reuse previously retrieved data. To increase application performance, caching can be applied at various layers of the architecture, such as the web layer, application layer, data layer, and network layer.

Typically, the server's **random-access memory (RAM)** and in-memory cache engines are utilized to support application caching. However, if caching is coupled to a local server, then the cache will not persist data in case of a server crash. Most applications are in a distributed environment, so it's better to have a dedicated caching layer that should be independent of the application life cycle. If you apply horizontal scaling to your application, all servers should be able to access the centralized caching layer to achieve the best performance.

The following diagram depicts the mechanism of caching in various layers of solution architecture:

Solution Architecture Layers

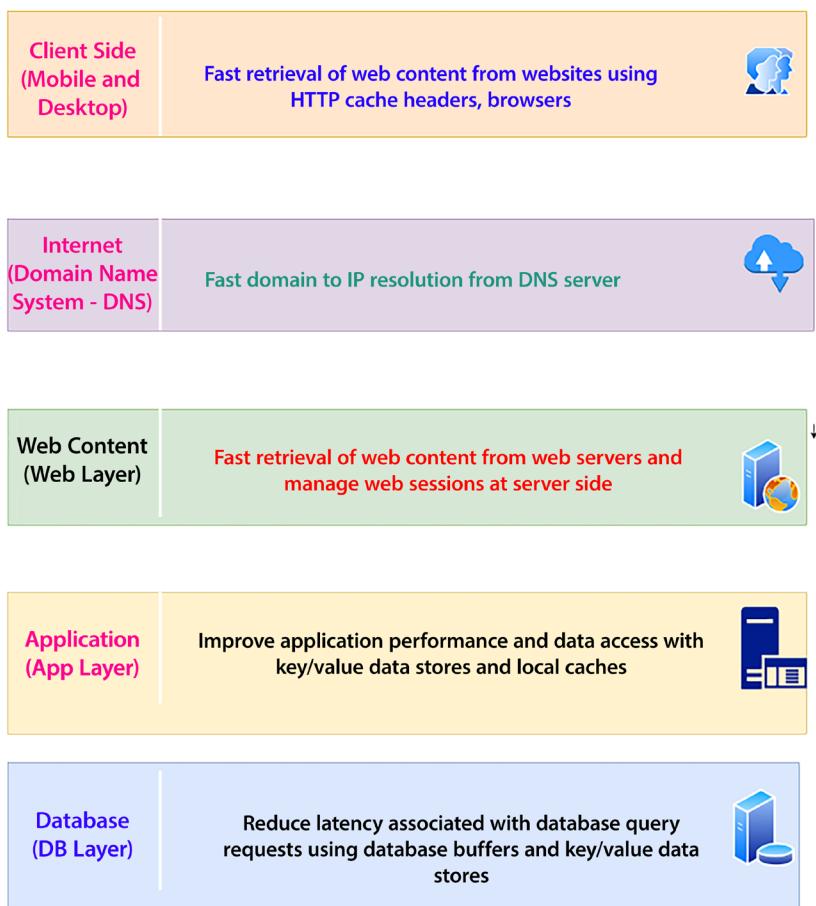


Figure 4.4: Caching at the architecture layers

As shown in the preceding diagram, the following are the caching mechanisms at each layer of the architecture:

- **Client-side caching:** Client-side caching is applied to user devices such as mobile and desktop. Client-side caching caches the previously visited web content to respond faster to a subsequent request. Each browser has its own caching mechanism. HTTP caching makes the application faster by caching content in the local browser. The cache-control HTTP header defines browser caching policies for client requests and server responses. These policies define where the content should be cached and how long it will persist, known as the **time to live (TTL)**. Cookies are another method used to store information on the client's machine to respond to the browser faster.
- **Internet DNS cache:** When a user types the website address over the Internet, the public **Domain Name System (DNS)** server looks up the IP address. Caching this DNS resolution information will reduce the website's load time. DNS information can be cached to a local server or browser after the first request, and any further requests to that website will be faster.
- **Web content caching:** Much of the request involves retrieving web content such as images, videos, and HTML pages. Caching these assets near the user's location can provide a much faster response for a page load. This also eliminates disk read and server load time. A **content distribution network (CDN)** provides a network of edge locations where static content, such as high-resolution images and videos, can be cached. It's beneficial for reading heavy applications such as games, blogs, e-commerce product catalog pages, and more. The user session contains lots of information regarding user preference and their state. It provides a great user experience by storing the user's session in its own key-value store for quick user response.
- **Application caching:** At the application layer, caching can be applied to store the result of a complex repeated request to avoid business logic calculations and database hits. Overall, it improves application performance and reduces the load on the database and infrastructure.
- **Database caching:** Application performance highly depends upon the speed and throughput provided by the database. Database caching increases database throughput significantly and lowers data retrieval latency. A database cache can be applied in front of any relational or non-relational database. Some database providers integrate caching, while applications handle local caching.

Redis and **Memcached** are the most popular caching engines. While Memcached is faster (it is suitable for low-structure data and stores data in a key-value format), Redis is a more persistent caching engine and is capable of handling complex data structures required for an application such as a gaming leaderboard; you will learn more details in the *Memcached versus Redis* section in this chapter. Let's learn about a few more caching design patterns.

Cache distribution pattern in a three-tier web architecture

Conventional web hosting architecture follows a common three-tier web application model, which divides the architecture into presentation, application, and persistence layers.

As shown in the following AWS architecture diagram, caching is applied at the web, application, and database layers:

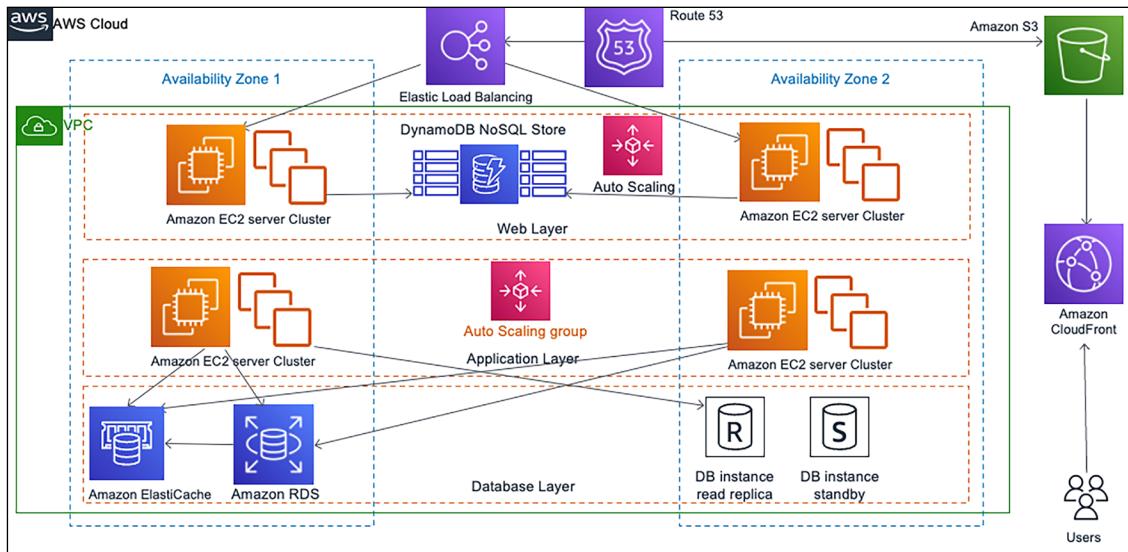


Figure 4.5: Cache distribution pattern architecture

In caching patterns, your goal is to try to hit the backend as little as possible. You can write an application where you can cache images, JavaScript, or even full pages to provide a better experience for your users. In the diagram above, caching is strategically implemented across various layers of the architecture:

- Amazon Route 53 plays a role in caching DNS-to-IP mappings, simplifying domain management.
- Amazon S3 serves as the storage location for static content, including high-resolution images and videos.
- Amazon CloudFront offers edge caching for high-traffic content, utilizing cache-control headers to determine update frequency from the origin.
- Amazon DynamoDB is used for session stores, helping web applications manage user sessions through caching.
- Elastic Load Balancing evenly distributes traffic among web server Auto Scaling groups.
- Amazon ElastiCache offers caching services for the application, effectively reducing the load on the database tier.

Typically, static content is cached, but there are scenarios where caching dynamic or unique content can improve application performance. The decision depends on specific usage patterns and demands.

Let's take a look at a more specific pattern.

Rename distribution pattern

When using a CDN such as Amazon CloudFront, you store frequently used data in an edge location near the user for fast performance. Often, you set up the TTL in the CDN for your data, which means the edge location will not query back to the server for updated data until the TTL expires. TTL is when an object is stored in a caching system before it's deleted or refreshed. You may have situations where you need to update CDN cached content immediately, for example, if you need to correct the wrong product description.

You can't wait for the file's TTL to expire in such a situation. The rename distribution pattern helps you update the cache as soon as new changes are published so the user can get updated information immediately. The following diagram shows this pattern with AWS:

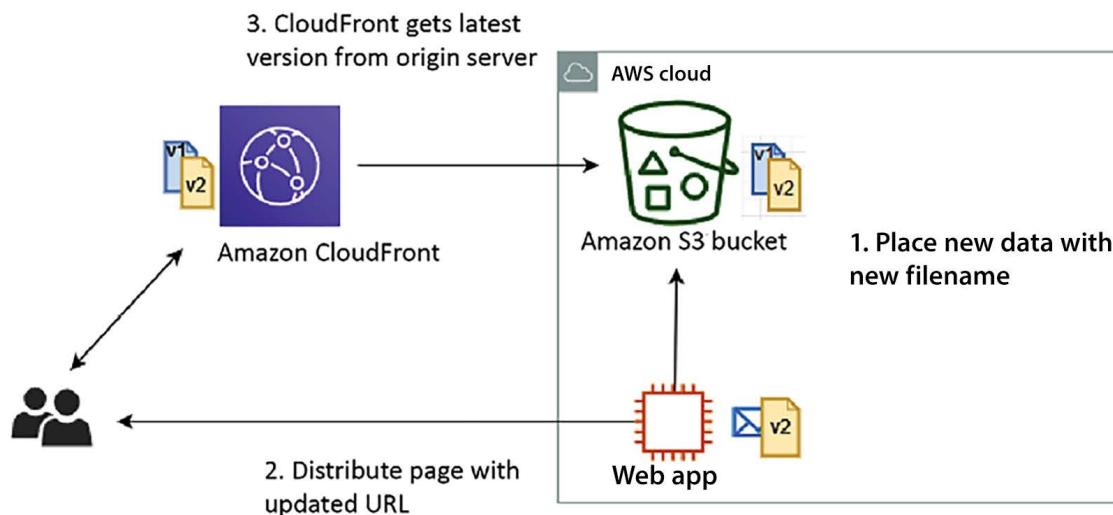


Figure 4.6: Rename distribution pattern architecture

As shown in the preceding diagram, using the rename distribution pattern with the cache distribution pattern helps to solve the update issue. With this pattern, instead of overwriting the file in the origin server and waiting for the TTL in CloudFront to expire, the server uploads the updated file with a new filename and then updates the web page with the new URL. When the user requests original content, CloudFront has to fetch it from the origin and can't serve the obsolete file that's already cached.

However, you can invalidate the old file immediately, but that will cost more, so it's better to put a new version of the file for the CDN to pick immediately. Again, you must update the URL in the application to pick up a new file, adding some overhead compared to the invalidation option. It would be best to decide based on your business requirements and budget.

You can use the proxy cache server to use something other than a CDN for a user base distributed across the country. Let's learn more about it in the next section.

Cache proxy pattern

You can increase your application performance significantly by adding a cache layer. In a cache proxy pattern, static or dynamic content is cached upstream of the web app server. As shown in the following architectural diagram, you have a caching layer in front of the web application cluster:

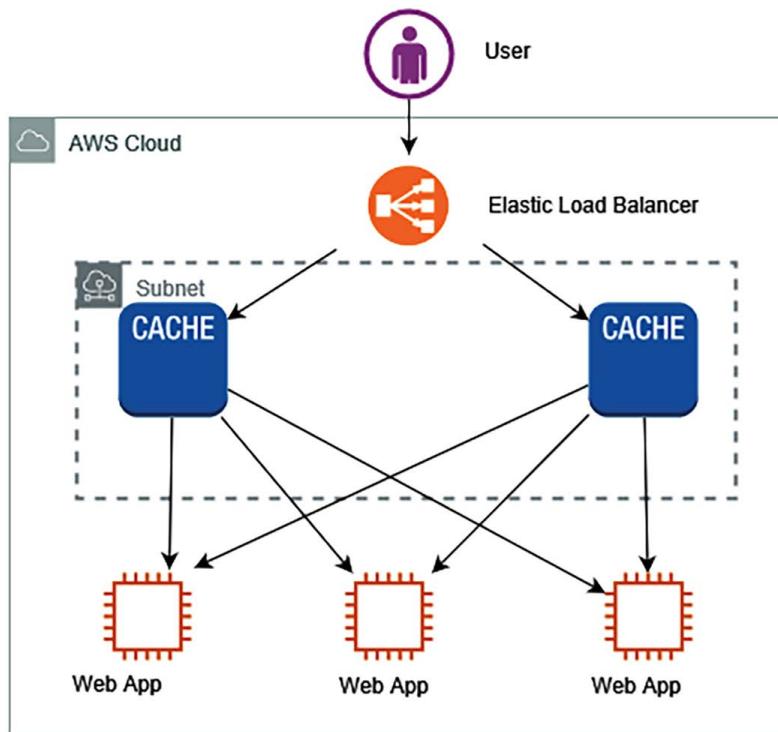


Figure 4.7: Cache proxy pattern architecture

In the preceding diagram, for high-performance delivery, cache content is delivered by the cache server. A few benefits of cache proxy patterns are as follows:

- Cache proxy patterns help you to deliver content using the cache, which means no modification is needed at the web server or application server level.
- They reduce the load of content generation for dynamic content.
- You can set up a cache at the browser level, such as in HTTP headers, URLs, cookies, and more. Alternatively, you can cache information in the cache layer if you don't want to store it at the browser level.

In the cache proxy pattern, you must maintain multiple copies of the cache to avoid a single point of failure. Sometimes, you may want to serve your static content from both the server and CDN, each requiring a different approach. Let's dive deep into this hybrid situation in the next section.

Rewrite proxy pattern

Sometimes, you want to change the access destinations of static website content, such as images and videos, but you don't want to change the existing systems. You can achieve this by providing a proxy server using rewrite proxy patterns. To change the destination of static content to other storage, such as a content service or internet storage, you can use a proxy server in front of the web server fleet. As shown in the following architecture diagram, you have a proxy server in front of your application layer, which helps to change the content delivery destination without modifying the actual application:

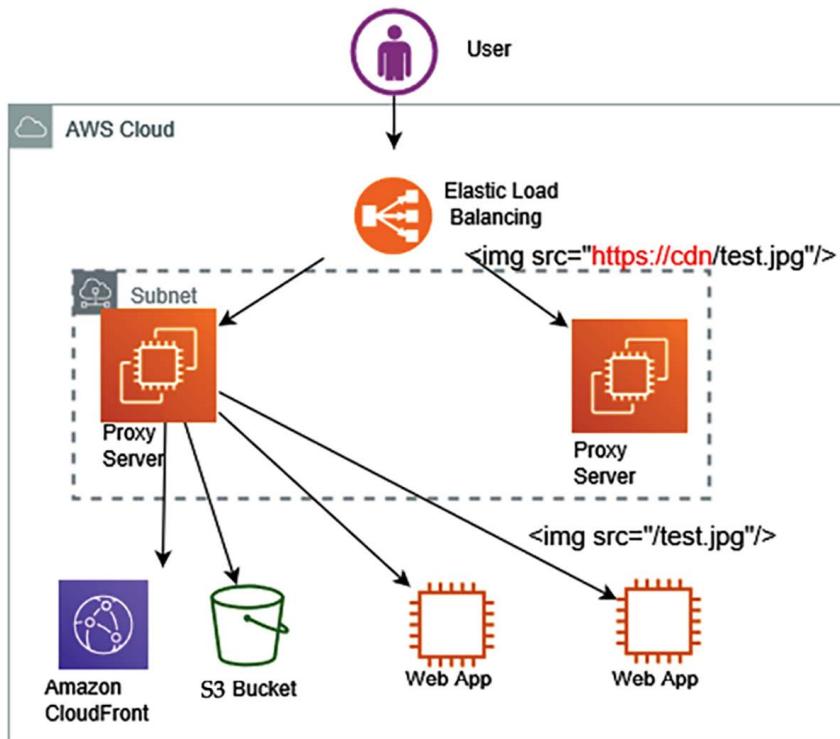


Figure 4.8: Rewrite proxy pattern architecture

As shown in the preceding diagram, place the proxy server in front of the currently running system to rewrite the proxy pattern. You can construct a proxy server using software such as Apache or NGINX. The following are the steps to build a rewrite proxy pattern, using AWS as the example:

1. Put a running proxy server on an EC2 instance, which can overwrite the content between the **load balancer** and the storage service, such as **Amazon S3**, which stores the static content.
2. Add to the proxy server rules for overwriting URLs within the content. These rules will help **Elastic Load Balancing (ELB)** to point to a new location, as shown in the preceding diagram, which redirects the proxy server rule from `https://cdn/test.jpg` to `/test.jpg`. ELB is a service provided by AWS that automatically distributes incoming application traffic across multiple targets, such as Amazon EC2 servers, containers, and IP addresses.

- As required, apply auto-scaling to the proxy servers by configuring minimum and maximum proxy server amounts as per the application load.

In this building a cache-based architecture section, you learned how to handle caching for static content distribution over the network. However, caching at the application layer is very important for improving application performance for the overall user experience. Let's learn more about the app caching pattern to handle the dynamic user data delivery performance.

App caching pattern

When it comes to applying caching to applications, you want to add a cache engine layer between your application servers and the database. The app caching pattern allows you to reduce the load on the database as the most frequent query is served from the caching layer. The *app caching pattern* improves overall application and database performance.

As shown in the following diagram, you can see the caching layer applied between the application layer and the database layer in AWS:

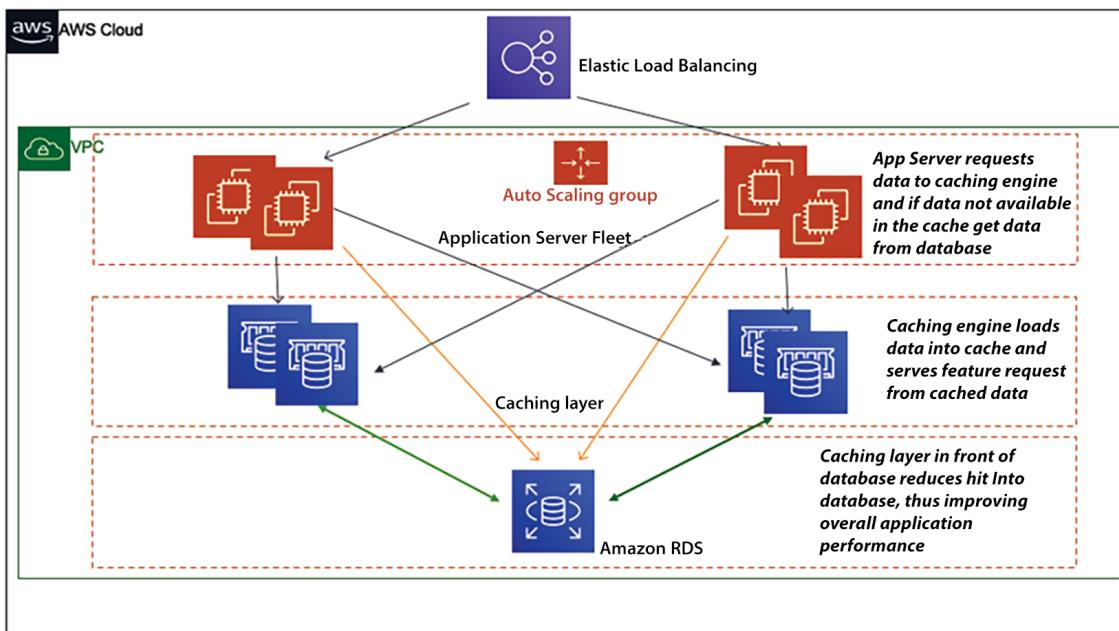


Figure 4.9: App caching pattern architecture

As shown in the preceding diagram, you can use either **lazy caching** or **write-through** based on your data access pattern. In lazy caching, the cache engine checks whether the data is in the cache and, if not, gets it from the database and keeps it in the cache to serve future requests. Lazy caching is also called the **cache aside pattern**. In the **write-through** method, data is written in the cache and in the data store at the same time. If the data gets lost from the cache, it can get it again from the database.

Choose lazy caching when you have a read-heavy application with acceptable stale data and choose write-through caching when dealing with write-heavy operations requiring immediate data consistency. For example, you can use lazy caching in a product catalog in an e-commerce website where product details are read frequently but updated less often. When a user accesses a product detail that is not in the cache, it's fetched from the database and stored in the cache for subsequent access, reducing database load. You might want to use write-through for a user review section on an e-commerce website, where user-generated reviews are instantly displayed on the product page. When a user submits a review, it's written to both the cache and the database simultaneously, ensuring that any subsequent read requests fetch the most current data.

Let's learn more about the popular caching engines *Redis* and *Memcached*.

Memcached versus Redis

Redis and Memcached are two popular caching engines used in application design. The Redis cache engine is often required for more complex application caching needs, such as creating a leaderboard for a game. However, Memcached is more high-performing and helps handle heavy application loads. Each caching engine has its pros and cons. Let's take a look at the significant differences between them, which will help you to make a decision about which to use:

Memcached	Redis
Offers multithreading	Single-threaded
Able to use more CPU cores for faster processing	Unable to utilize multi-core processor, which results in comparatively slow performance
Supports key-value style data	Supports complex and advanced data structures
Lacks data persistence; loses the data stored in cache memory in the event of a crash	Data can persist using built-in read replicas with failover
Easy maintenance	More complexity is involved owing to the need to maintain the cluster
Good to cache flat strings such as flat HTML pages, serialized JSON, and more	Good to create a cache for a gaming leaderboard, a live voting app, and more

Table 4.1 – Memcached versus Redis comparison

If you need to decide which engine to use, base it on a use case that can justify using Redis or Memcached. Memcached is simple and has lower maintenance, and it is typically preferred when your cache doesn't need the advanced features that Redis offers. However, Redis is the best solution if you need the advantage of data persistence, advanced data types, or any of the other features listed.

When implementing caching, it's essential to understand the validity of data that needs to be cached. If the **cache hit** rate is high, the data is available in the cache when required. For a higher cache hit ratio, offload the database by reducing direct queries, improving the overall application performance. A **cache miss** occurs when data is not present in the cache, which increases the load in the database. The cache is not a large data store, so you need to set the TTL and evict the cache as per your application needs.

As you have seen in this section, applying caches has multiple benefits, including application performance improvement, the ability to provide predictable performance, and the reduction in database cost.

Let's learn about some more application-based architecture that demonstrates the principle of loose coupling and constraint handling, MVC architecture.

Model-View-Controller (MVC) architecture

MVC is one of the most popular design patterns for developing software applications. It separates an application into three interconnected components: Model, View, and Controller. This separation enables more modular development, easier testing, and excellent maintainability. Let's explore these components in detail:

- **Model:** The Model represents the internal state of the application, along with the rules, logic, and data that govern and manipulate that state. The model does not depend on the View or Controller, meaning changes to the UI or business logic do not affect the data handling. It ensures that the application's data remains consistent across different parts of the application. Its responsibilities include:
 - **Managing data:** It contains all the data-related logic. Whether it's retrieving data from a database or an API, the model handles it.
 - **Implementing business rules:** Implements business logic, such as calculations or data transformations.
 - **Notifying of changes:** Informs associated Views and Controllers when the data has changed so they can update themselves accordingly.
- **View:** The View is the visual representation of the Model's data. It defines exactly how the application's data is presented to the user. The View automatically updates when the underlying Model data changes, ensuring the user always sees the most current data. Multiple Views can be created from the same Model data, allowing different representations (e.g., tables, charts, detailed views). Its responsibilities include:
 - **Displaying data:** Takes the data from the model and presents it in a format that's easy to understand.
 - **Handling the user interface (UI):** Handles all the UI logic of the application, such as user input fields, buttons, display screens, etc.
- **Controller:** The Controller mediates between the Model and the View. It takes user inputs from the View, processes them (with potential updates to the Model), and returns the output display to the View. The controller ensures that the View and the Model are always in sync with each other. It acts as a centralized handler for all user interactions, making the management of those interactions more systematic and organized. Its responsibilities include:
 - **Handling user input:** Takes and interprets user commands, translating them into actions to be performed by the model
 - **Updating the Model:** Modifies the data in the Model by sending commands to it

- **Updating the View:** Changes what is presented in the View based on user input and Model changes

The following are the key advantages of applying the MVC pattern:

- **Separation of concerns:** By isolating the application's data, user interface, and control logic, MVC promotes modular development.
- **Reusability:** Components can be reused across different parts of an application or even different applications.
- **Maintainability:** It makes updating, testing, and debugging different parts of the application easier.
- **Flexibility:** Enables developers to change one part of the system without affecting others, such as changing the UI without altering the underlying data processing.

MVC is a powerful architectural pattern that offers robust data management, user interface, and business logic management. It's widely used in various application development environments, from web development frameworks to desktop applications, to create scalable and maintainable software. By following the principles of MVC, application architects can create organized, efficient, and flexible applications that are easier to update and maintain. Let's consider an example to understand MVC better.

Applying MVC to design an online bookstore

When designing an online bookstore, for example, the MVC architecture efficiently handles complex interactions between book data, user interface, and user inputs, leading to a more robust and scalable system. Let's look at the details of each module:

- **Model:** Managing the data related to books, authors, categories, customer reviews, etc. Examples of operations are:
 - Retrieving details of a specific book
 - Updating the inventory after a purchase
 - Adding a new book to the catalog
- **View:** Displaying information to the user in a readable and interactive format. Examples of Views are:
 - **Book listing page:** Displays a list of books with their titles, covers, and prices
 - **Book detail page:** Shows detailed information about a specific book, including author, description, reviews, etc.
 - **Shopping cart page:** Allows users to view, add, or remove items from their shopping cart
- **Controller:** Handling user interaction, updating the model as needed, and updating the View to reflect changes. Examples of actions are:
 - **Searching for a book:** The user enters a search term. The Controller queries the model for matching books and updates the View to display the results.

- **Adding to cart:** The user clicks **Add to Cart**, the Controller updates the model to reflect the new item in the cart, and the View is updated to show the new cart status.
- **Checkout:** The user decides to purchase. The Controller processes the transaction, updates the model (including inventory), and redirects to a confirmation View.

The MVC pattern provides a clean separation of concerns, making extending, maintaining, and testing applications easier.

Building Domain-Driven Design (DDD)

Domain-Driven Design (DDD) is a methodology and set of practices aimed at understanding and solving complexity at the heart of software. This approach is used to design and model software based on the “domain,” or the business’s core logic and key concepts. Using a common language and dividing the system into clear contexts, DDD promotes a deep understanding of the problem space and leads to a design that accurately reflects the underlying business needs. It’s particularly valuable in complex domains, where aligning the software closely with the real-world concepts it represents is vital.

Let’s delve into DDD using a concrete example and use case. For this, we’ll consider the domain of a **Healthcare Management System (HMS)**. Imagine we’re developing a system that manages patient records, appointments, medical treatments, billing, etc. for a healthcare provider. Here’s how we could apply DDD concepts to this domain:

- **Domain:** A “domain” refers to a specific problem area the software intends to address. The application logic revolves around the sphere of knowledge and activity. Understanding the domain is essential for creating a system that truly meets the needs of the business. For HMS, the domain will be healthcare management, focusing on patients, medical staff, appointments, treatments, and billing.
- **Ubiquitous Language:** Ubiquitous Language is a shared language between developers and non-technical stakeholders that describes the domain. This common language ensures that all team members understand the key terms and concepts in the same way, reducing misunderstandings and promoting clear communication for the HMS, creating a shared language that both medical professionals and developers understand, for example, **Patient**, **Appointment**, **Treatment**, **Medical Staff**, etc.
- **Bounded contexts:** In DDD, the application is divided into different bounded contexts, each representing a specific responsibility or functionality within the overall domain. A bounded context encapsulates all the terms, definitions, and logic for that specific part of the domain, and it is explicit about what is inside and outside its boundaries. For example, the **Patient Management** bounded context handles patient records, personal information, medical history, etc. An **Appointment Scheduling** bounded context includes managing appointments, scheduling, cancellations, rescheduling, etc., and the **Billing** bounded context includes processing payments, insurance, invoices, etc.
- **Entities:** These objects have a distinct identity that runs through time and different states, for example, patients (with a unique ID) and medical staff (with unique credentials).

- **Value objects:** Objects that describe characteristics of a thing but have no conceptual identity. They are immutable and can be easily replaced. For example, **Address**, **Date of Birth**, and **Medical History** (as these don't have individual identities).
- **Aggregates:** An aggregate is a cluster of associated objects treated as a single unit for data changes. One entity within the aggregate is the root, and external references are restricted to this root to ensure integrity. For example, in an online healthcare management system, a medical appointment can be modeled as an aggregate. The aggregate might include entities and value objects like **Patient** (who the appointment is for), **Medical Staff** (who will attend the patient), **Treatment Room** (where the appointment will take place), and **Time Slot** (when the appointment is scheduled). Here, the **Appointment** entity would be the aggregate root. Any changes to the **Patient**, **Medical Staff**, **Treatment Room**, or **Time Slot** related to a specific appointment would be made through the **Appointment** entity. This ensures that the appointment aggregate maintains consistency and enforces all business rules related to medical appointments.
- **Repositories:** Repositories are used to retrieve aggregates from the underlying persistence layer. They provide an abstraction allowing the rest of the application to interact with the data store in a way consistent with the domain model. For example, the **Patient** repository is used to fetch and manage Patient entities, and the **Appointment** repository is used to retrieve and store Appointment aggregates.
- **Factories:** Factories are responsible for encapsulating the logic of creating complex objects and aggregates. They ensure that an object or aggregate is created in a consistent and valid state. For example, the **Patient** factory is used to create a new **Patient** entity with a valid initial state, and the **Appointment** factory is used to create a new **Appointment** aggregate with the required details.
- **Services:** When an operation doesn't logically belong to a value object or entity, it can be defined as a service. Services are part of the domain model and contain business logic that operates on the domain's concepts. For example, in the **Billing** context, the billing service contains operations like calculating total charges, applying insurance discounts, generating invoices, etc.
- **Domain events:** Domain events capture the fact that something significant has happened within the domain. They can trigger other activities within the system or in other systems. For example, an appointment scheduled event triggered when a new appointment is scheduled may notify relevant staff members and a payment processed event occurs after successful payment, which might initiate a receipt generation process.
- **Anti-corruption layer:** This layer translates between different parts of the system that use different languages or models. It ensures that each model's integrity is maintained, and inconsistencies are handled. If the **Billing** system must interact with an external third-party payment gateway, an anti-corruption layer could translate between the domain model in the **Billing** context and the model used by the external system.

In this HMS, DDD ensures that complex domain logic is carefully modeled and organized. It encourages collaboration between healthcare professionals (domain experts) and developers to create a shared understanding and language.

The system's design closely aligns with real-world healthcare operations by defining clear bounded contexts, entities, aggregates, and other DDD concepts. This alignment ensures that the software provides a robust and flexible solution tailored to the specific needs of the healthcare domain.

This example shows how DDD can be an essential tool in crafting complex, well-structured systems by focusing on the core domain and facilitating collaboration between different stakeholders.

Dependency handling is an important aspect when working in complex systems. Let's learn about how to handle dependency between different services via a circuit breaker to make sure an error in one service does not bring the entire system down.

Understanding the circuit breaker pattern

It's common for a distributed system to make a call to other downstream services, and the call could fail or hang without a response. You will often see code that retries the failed call several times. The problem with a remote service is that it could take minutes or even hours to correct, and an immediate retry might end up in another failure. As a result, end users wait longer to get an error response while your code retries several times. This retry function would consume the threads and potentially induce a cascading failure.

The circuit breaker pattern is about understanding the health of downstream dependencies. It detects when those dependencies are unhealthy and implements logic to gracefully fail requests until it detects that they are healthy again. The circuit breaker can be implemented using a persistence layer to monitor healthy and unhealthy requests over the past request interval.

If a defined percentage of requests observe an unhealthy behavior over the past interval or a total count of exceptions, regardless of percentage, the circuit is marked as open. In such a situation, all requests throw exceptions rather than integrate with the dependency for a defined timeout period. Once the timeout period has subsided, a small percentage of requests try integrating with the downstream dependency to detect when the health has returned. Once a sufficient percentage of requests are healthy again over an interval, or no errors are observed, the circuit closes again, and all the requests are allowed to thoroughly integrate as they usually would.

The implementation decisions involve the state machine tracking/sharing the healthy/unhealthy request counts. The states of services can be maintained in DynamoDB, Redis/Memcached, or another low-latency persistence store.

Let's next turn to the bulkhead architectural pattern, which helps to reduce dependency between services and mitigate the situation in the case of a service getting an error.

Implementing the bulkhead pattern

Bulkheads are structural partitions used in ships to create individual watertight sections. The primary purpose is to contain the consequences of any breach in the ship's hull, thereby preventing water from spreading throughout the vessel in the event of damage. This design serves as a crucial safety measure, aiming to minimize the risk of the entire ship sinking if one area is compromised.

The same concept is helpful to limit the scope of failure in the architecture of large systems where you want to partition your system to decouple dependencies between services. The idea is that one failure should not cause the entire system to fail, as shown in the following diagram:

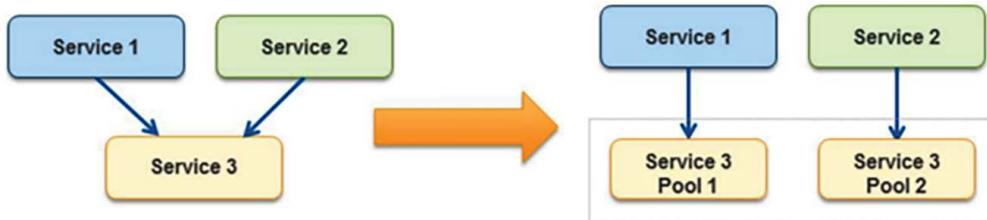


Figure 4.10: Bulkhead pattern

In the bulkhead pattern, it's better to isolate the element of the application into the pool for service, which has a high dependency; so, if one fails, others continue to serve upstream services. Service 3 is partitioned into two pools from a single service in the preceding diagram. Here, if Service 3 fails, then the impact of either Service 1 or Service 2 depends on their dependency on the pool, but the entire system does not go down. The following are the significant points to consider when introducing the bulkhead pattern in your design, especially for the shared service model:

- Save part of the ship, which means your application should not shut down due to the failure of one service.
- Decide whether less efficient use of resources is okay. Performance issues in one partition should be fine for the overall application.
- Pick a useful granularity. Make sure to make the service pools manageable; make sure they can handle the application load.
- Monitor each service partition performance and adhere to the SLA. Ensure all moving parts are working together and test the overall application when one service pool is down.

You should define a service partition for each business or technical requirement. It would be best if you used this pattern to prevent the application from cascading failure and isolating critical consumers from the standard consumer.

Often, legacy application servers have a configuration with hardcoded **Internet Protocol (IP)** addresses or **Domain Name System (DNS)** names. Making any server change for modernization and upgrade requires changing and revalidating the application. In these cases, you want to keep the server address the same. In the next section, let's learn how to handle such a situation with a floating IP.

Creating a floating IP pattern

Commonly, monolithic applications have many dependencies on the server where they are deployed. Application configuration and code often have hardcoded parameters based on the server's DNS name and IP address. Hardcoded IP configuration creates challenges if you want to bring up a new server in case of an issue with the original server. Additionally, you don't want to bring down the entire application for the upgrade, which may cause significant downtime.

To handle such a situation, you need to create a new server keeping the same server IP address and DNS name. This can be achieved by moving the network interface from a problematic instance to the new server. The network interface is generally based on a **Network Interface Card (NIC)**, which facilitates communication between servers over a network. It can be in the form of hardware or software. Moving the network interface means that now your new server assumes the identity of the old server. Your application can live with the same DNS and IP address. It also allows easy rollback by moving the network interface to the original instance.

The public cloud (for example, AWS) made it easy by providing an **Elastic IP (EIP)** and **Elastic Network Interface (ENI)**. If your instance fails and you need to push traffic to another instance with the same public IP address, then you can move the EIP address from one server to another, as shown in the following architecture diagram:

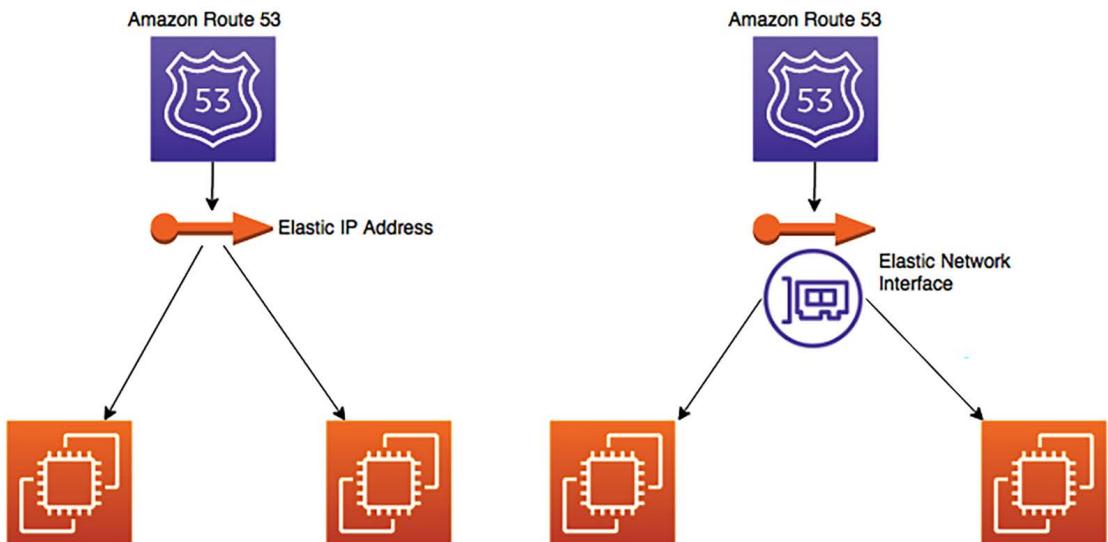


Figure 4.11: Floating IP and interface pattern

Since you are moving EIP, the DNS may not need to be updated. EIP can move your server's public IP across instances. If you need to move public and private IP addresses, use a more flexible approach, such as ENI, as shown on the right of the preceding diagram. ENI can move across instances, and you can use the same public and private address for traffic routing or application upgrades.

So far, you have learned about multiple architecture patterns where applications are deployed in the virtual machine. However, you may need help to utilize the virtual machine in many cases. To optimize your utilization further, you can deploy your application in containers. Containers are most suitable for microservice deployment. Let's learn more about container-based deployment in the next section.

Deploying an application with a container

As many programming languages are invented, and technologies evolve, this creates new challenges. Different application stacks require different hardware and software deployment environments. Often, there is a need to run applications across different platforms and migrate from one platform to another. Solutions require something that can run anything everywhere and is consistent, lightweight, and portable.

Just as shipping containers standardize the transport of freight goods, software containers standardize the transport of applications. Docker creates a container that includes everything a software application would need to run its files, such as filesystem structure, daemons, libraries, and application dependencies.

Containers provide isolation for software within its respective development and staging environments. This isolation is essential because it prevents conflicts from arising when multiple teams are running various software applications on the same underlying infrastructure.

VMs are isolated at the operating system level, and containers isolate at the kernel level. This isolation allows several applications to run on a single-host operating system and yet still have their filesystem, storage, RAM, libraries, and, mostly, their own *view* of the system:

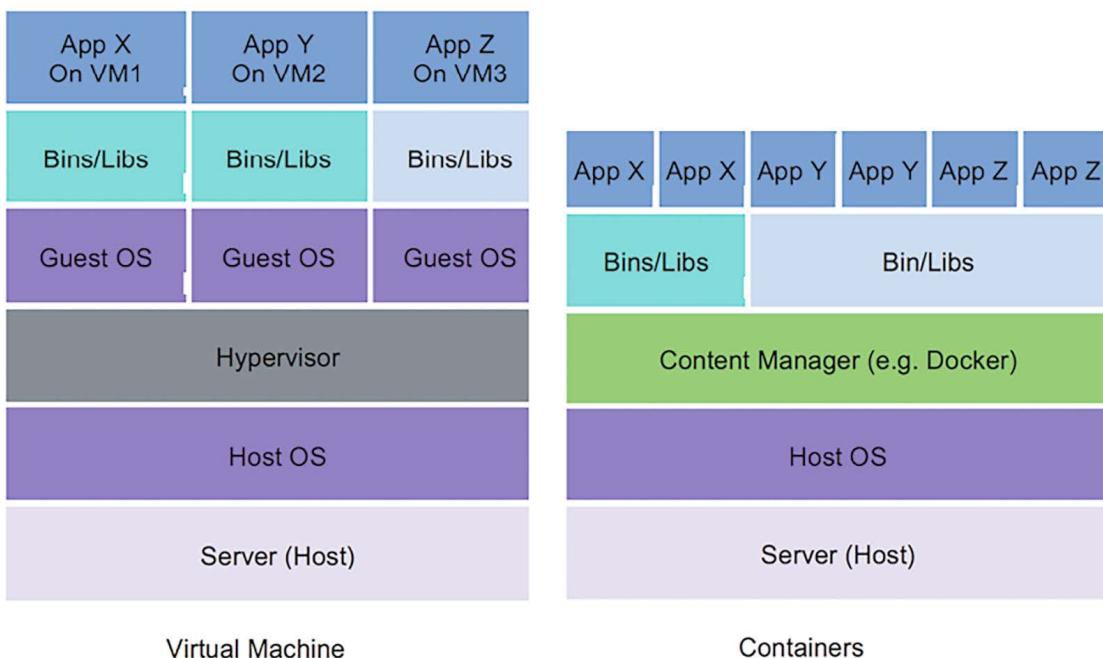


Figure 4.12: Virtual machines and containers for application deployment

As shown in the preceding diagram, containers deploy multiple applications in a single virtual machine. Each application has its runtime environment, so you can run many individual applications while keeping the same number of servers. Containers share a machine's operating system kernel. They offer advantages like quick startup times and efficient use of computing resources such as RAM. Container images are built using layers from the filesystem, and they can share common files. This shared resource approach reduces disk usage and speeds up the process of downloading container images.

Let's look at why containers are becoming more popular, along with their benefits.

The benefit of containers

These questions are often asked when it comes to containers:

- Why do we need containers when we have instances?
- Don't instances already provide us with isolation from the underlying hardware?

While the preceding questions are valid, several benefits accrue from using a system such as Docker. One of the key benefits of Docker is that it allows you to fully utilize your virtual machine resources by hosting multiple applications (on distinct ports) in the same instance.

Docker uses certain features of the Linux kernel, namely kernel namespaces and groups, to achieve complete isolation between each Docker process, as indicated in the following architecture diagram:

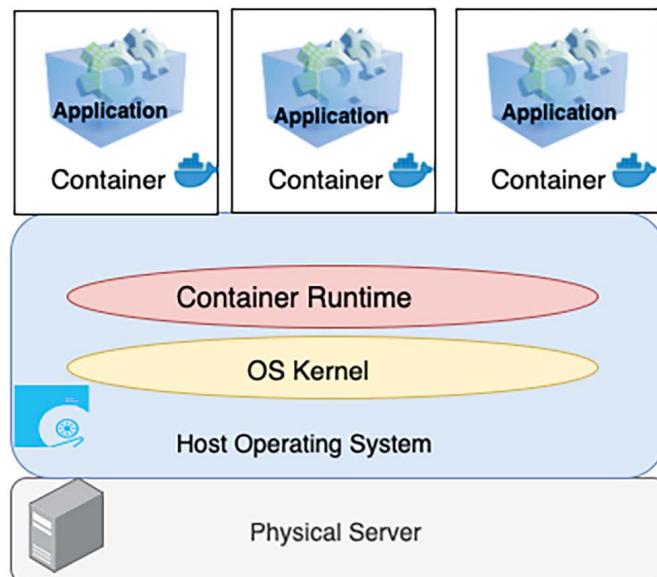


Figure 4.13: Container layer in application infrastructure

As shown in the preceding diagram, it's possible to run two or more applications that require different versions of the Java runtime on the same machine, as each Docker container has its version of Java and the associated libraries installed. In turn, the container layer in the application infrastructure makes it easier to decompose your applications into microservices that can run side by side on the same instance. Containers have the following benefits:

- **Portable runtime application environment:** Containers provide platform-independent capabilities, where you build your application once and deploy it anywhere regardless of the underlying operating system.
- **Faster development and deployment cycles:** Modify the application and run it anywhere with a quick boot time, typically within seconds.
- **Package dependencies and application in a single artifact:** Package the code, library, and dependencies together to run the application in any operating system.
- **Run different application versions:** Applications with different dependencies run simultaneously in a single server.
- **Everything can be automated:** Container management and deployment are done through scripting, which helps to reduce cost and the risk of human error.
- **Better resource utilization:** Containers provide efficient scaling and high availability, and multiple copies of the same microservice container can be deployed across servers for your application.
- **Easy to manage the security aspect:** Containers are platform-specific rather than application-specific.

Container deployment is becoming very popular due to its benefits. There are multiple ways to orchestrate containers. Let's look at container deployment in more detail next.

Container deployment

Complex applications with multiple microservices can be quickly deployed using container deployment. The container makes building and deploying the application more manageable as the environment is the same. Build the container in development mode, push it to test, and then release it to production. For hybrid cloud environments, container deployment is very useful. Containers make it easier to keep environments consistent across microservices. As microservices aren't always very resource-consuming, they can be placed together in a single instance to reduce cost.

Sometimes, customers have short workflows that require a temporary environment setup. Those environments may be queue systems or continuous integration jobs, which don't always utilize server resources efficiently. Container orchestration services such as Docker and Kubernetes can be a work-around, allowing them to push and pop containers onto the instance.

Docker's lightweight container virtualization platform provides tools to manage your applications. Its standalone application can be installed on any computer to run containers. Kubernetes is a container orchestration service that works with Docker and another container platform. Kubernetes allows automated container provisioning and diligently handles security, networking, and scaling aspects.

Containers help the enterprise to create more cloud-native workloads, and public cloud providers such as AWS extend services to manage Docker containers and Kubernetes.

The following diagram shows Docker's container management using Amazon Elastic Container Service (ECS), providing a fully managed elastic service to automate the scaling and orchestration of Docker containers:

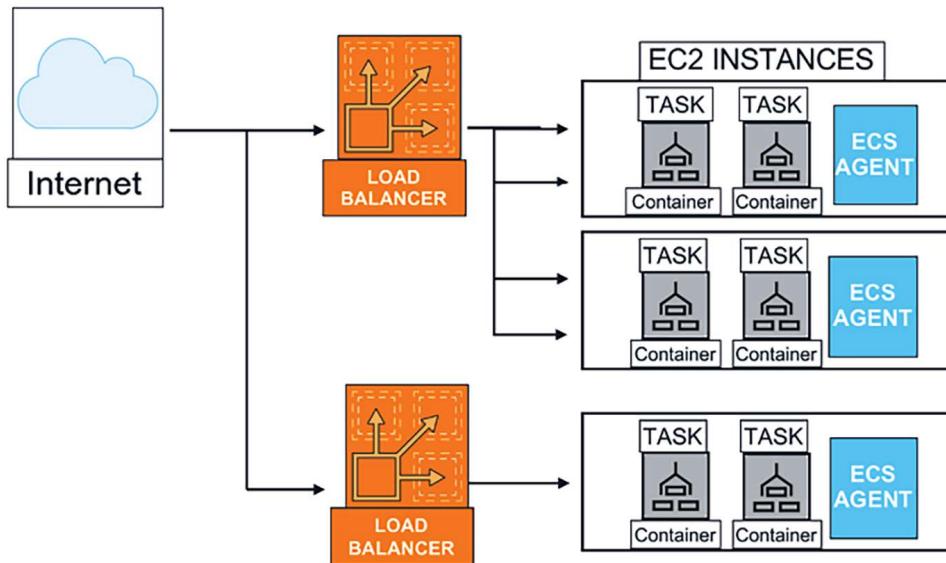


Figure 4.14: Container deployment architecture

In the preceding diagram, multiple containers are deployed in a single Amazon EC2 virtual machine managed through Amazon ECS, which facilitates the agent communication service and cluster management. All user requests are distributed using a load balancer among the containers. Similarly, AWS provides Amazon Elastic Kubernetes Service (EKS) to manage containers using Kubernetes.

Containers are a broad topic, and as a solutions architect, you must be familiar with all the available options. This section provides an overview of containers. However, you will need to dive further if you utilize containers for your microservice deployment. Let's look at container-based architecture in the next section.

Building container-based architecture

As you learned in the previous section, containerization helps create environments for repeatable and scalable applications. To start container adoption, you need to identify a pilot workload managed through container orchestration. You can take existing microservice components and deploy them in containers. After identifying gaps and operational needs, you can define a migration strategy to move your workload to containers.

Container migrations can be challenging if your applications are not originally designed to operate in a containerized environment. This is because many applications typically store files locally and rely on stateful sessions. When migrating to containers, it's essential to address these specific requirements and ensure that your applications can function smoothly within the container environment.

For container platforms, you can make choices; you can choose Docker, OpenShift, Kubernetes, and so on. However, Kubernetes is becoming an increasingly popular open-source container orchestrator. Public cloud vendors like AWS provide a platform to manage containers, such as Amazon ECS for Docker and Amazon EKS for Kubernetes. These cloud services provide a control plane to choose various compute options to select self-managed nodes, managed nodes, or serverless options with AWS Fargate. The control plane serves as the central management interface, allowing for the orchestration and operational oversight of containerized applications and their resources. If you are utilizing Amazon EKS for deploying a microservices-based application, for example, the Kubernetes control plane, managed by AWS, takes care of orchestrating container deployments, managing state, and maintaining desired configurations. This setup allows you to focus on application development rather than managing infrastructure.

The following architecture diagram shows running a stateful service on Amazon EKS in your programming languages of choice, such as Java or .NET. Given the architecture, you can manage the session state in a Redis database.

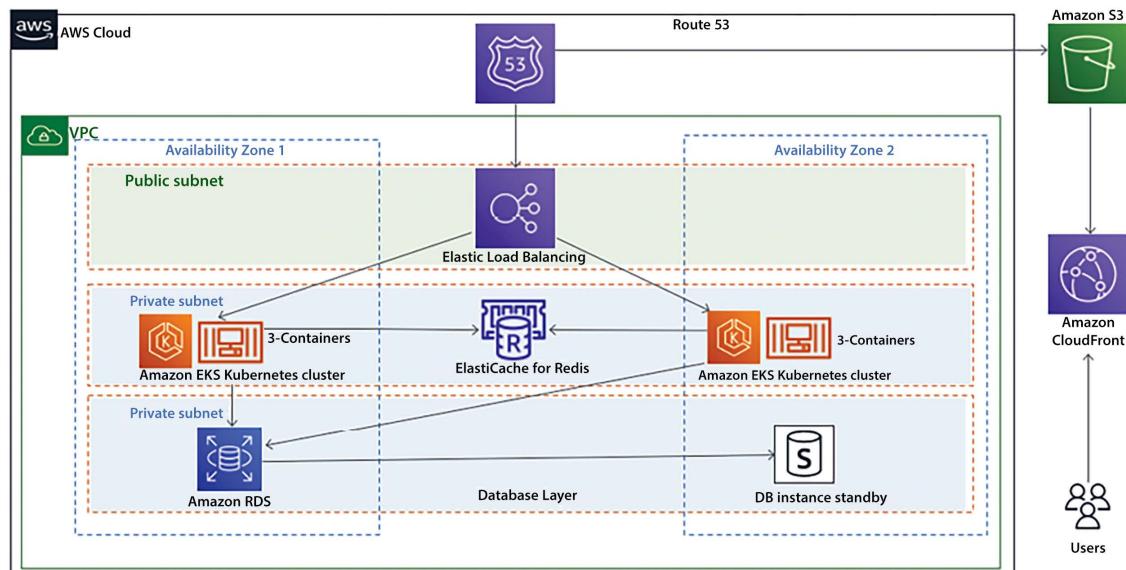


Figure 4.15: Deploying a stateful application on a container

As you can see in the preceding diagram, the container-based architecture comprises several key components:

- An Amazon Virtual Private Cloud (VPC) with specific subnets:
 - Public subnet: Hosts the load balancer
 - Private subnets: Used for deploying the application and database
- An Application Load Balancer, responsible for providing access to the website hosted within the containers.
- An Amazon Elastic Kubernetes Service (EKS) cluster featuring a managed node group within Kubernetes. These nodes are responsible for running multiple application containers.
- An Amazon ElastiCache Redis database, utilized to store user session state.

This architecture allows for the scalability of the application by storing user sessions in a Redis database. However, please note that implementing this solution may require modifications to the application code, which may not be feasible in certain scenarios.

Now, you have learned about various architecture patterns focusing on application development. Data is an integral part of any architectural design, and most of the architecture revolves around collecting, storing, and processing data visualization. In the next section, let's learn more about handling data in application architecture.

Database handling in application architecture

Data is always central to any application development, and scaling data has always been challenging. Handling data efficiently improves application latency and performance. In the earlier section *Building a cache-based architecture*, you learned how to handle frequently queried data by putting a cache in front of your database under the app caching pattern. You can put either a Memcached or Redis cache in front of your database, reducing the many hits on the database and improving database latency.

In application deployment, as your application's user base grows, you need to handle more data with your relational database. You need to add more storage or vertically scale the database server by adding more memory and CPU power. Often, horizontal scaling is more complex when it comes to scaling relational databases. If your application is read-heavy, you can achieve horizontal scaling by creating a read replica. Route all read requests to database read replicas while keeping the master database node to serve write and update requests. As a read replica has asynchronous replication, it can add some lag time. You should choose the read replica option if your application can tolerate some milliseconds of latency. You can use read replicas to offload reporting.

You can use database sharding to create a multi-master for your relational database and inject the concept of horizontal scaling. The sharding technique is used to improve writing performance with multiple database servers. The database is structured and segmented into identical sections, with appropriate table columns serving as keys for distributing the writing processes.

As demonstrated in the following architecture diagram, the customer database can be divided into multiple shards:

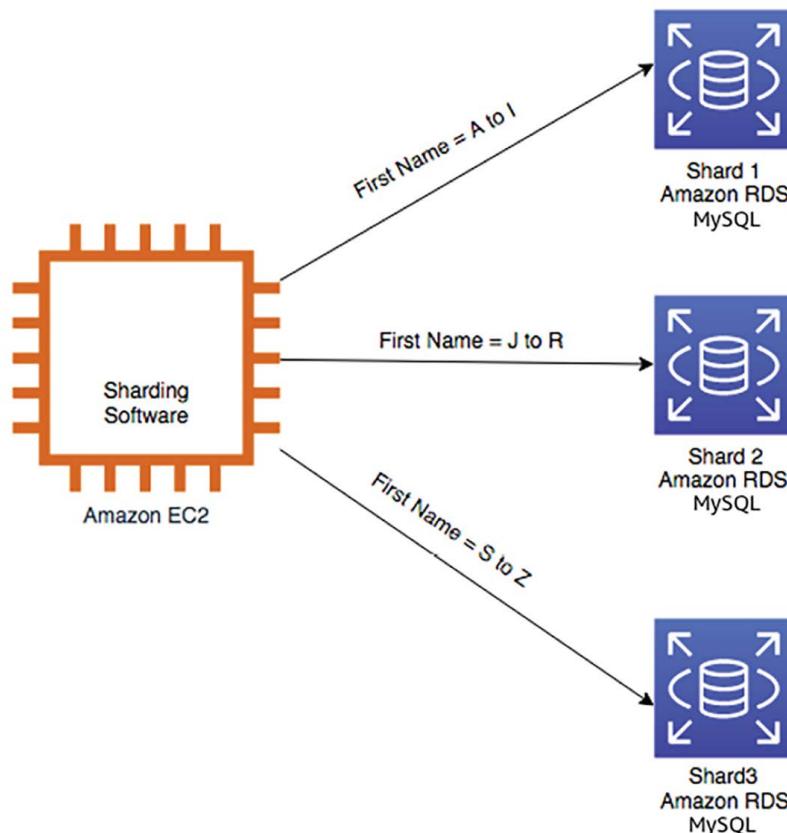


Figure 4.16: Relational database sharding

As shown in the preceding diagram, without *shards*, all data resides in one partition, for example, the first names of all users being in one single database. With sharding, data is split into large chunks called shards. For example, all users' first names beginning with **A to I** are in one database, **J to R** in another, and **S to Z** in a third database. In many circumstances, sharding gives you higher performance and better operating efficiency.

Utilizing Amazon RDS for sharding backend databases involves installing sharding software, such as MySQL, along with a Spider storage engine on an Amazon EC2 instance. Subsequently, you can begin by setting up multiple RDS databases and employing them as backend databases for sharding.

However, what if your master database instance goes down? In that case, you need to maintain high availability for your database. Let's take a closer look at the high-availability database pattern.

High-availability database pattern

For the high availability of your application, it is critical to keep your database up and running all the time. As horizontal scaling is not a straightforward option in the relational database, it creates additional challenges. To achieve high database availability, you can have a standby replica of the master database instance, as shown in the following diagram:

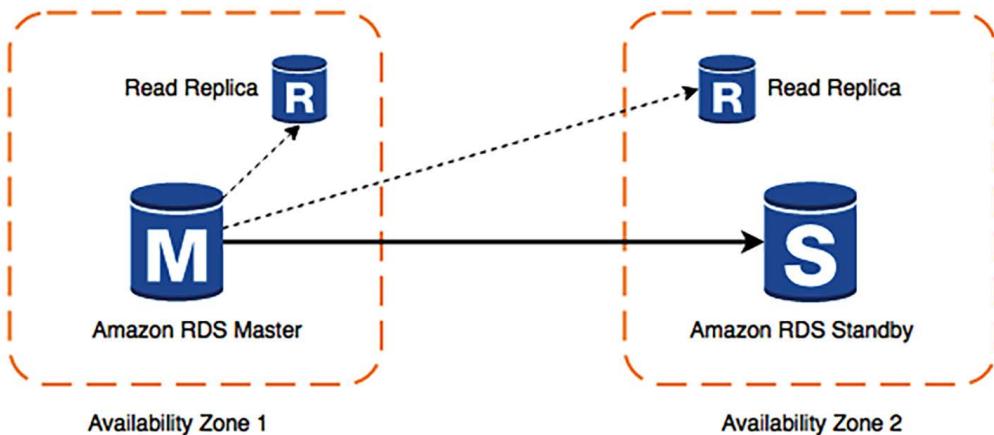


Figure 4.17: High-availability database pattern

As shown in the preceding diagram, your application server switches over to the standby instance if the primary instance goes down. A read replica takes the load off the primary instance to handle latency. The primary and standby are located in different **availability zones**, so your application will still be up even when an entire availability zone is down. This architecture also helps to achieve zero downtime, which may be caused during the database maintenance window. When a primary instance is down for maintenance, the application can fail over to a secondary standby instance and continue serving user requests.

For disaster recovery, you will want to define the database backup and archival strategy, depending on your application's **recovery point objective** (RPO) of how frequently you want to take backups. You will learn about RPOs and RTOs in depth in *Chapter 8, Architectural Reliability Considerations*.

If your RPO is 30 minutes, it means your organization can only tolerate 30 minutes' worth of data loss. In that case, you should take a backup every half an hour. While storing the backup, you need to determine how long the data can be stored for customer query purposes. You can store data for six months as an active backup and then in an archival store as per the compliance requirement.

Consider how quickly you need to access your backup and determine the type of network connection needed to meet your backup and recovery requirements as per the company's **recovery time objective** (RTO).

For example, if your company's RTO is 60 minutes, it means you should have enough network bandwidth to retrieve and restore your backup within an hour. Also, define whether you are backing up snapshots of complete systems or volumes attached to systems.

You may also need to classify your data, for example, if it has customer-sensitive information such as email, addresses, personally identifiable information, and more. It would be best if you defined the data encryption strategy accordingly. You will learn more about data security in *Chapter 7, Security Considerations*.

Depending on your application's growth and complexity, consider migrating from an RDBMS to a NoSQL database. NoSQL can provide greater scalability, management, performance, and reliability than most relational databases. However, the process of migrating to NoSQL from an RDBMS can be time-consuming and labor-intensive.

There is a lot of data to process in any application, for example, clickstream data, application log data, rating and review data, social media data, and more. Analyzing these datasets and getting insight can help you to grow your organization exponentially. *Chapter 12, Data Engineering for Solution Architecture*, will teach you more about these use cases and patterns.

Now, let's learn about building a maintainable system using Clean Architecture.

Clean Architecture

Clean Architecture, also known as Hexagonal Architecture or Ports and Adapters, is an architectural pattern used in designing business applications. Robert C. Martin proposed it and it emphasizes the separation of concerns, maintainability, and testability. Clean Architecture aims to create a flexible, adaptable, and maintainable system over time.

Clean Architecture divides your application into five key components; let's understand them through the example of an online bookstore:

1. **Entities (innermost layer):** Entities are the business objects that encapsulate the core business rules. They are independent of any specific technology, database, or framework. Entities represent the “things” in the system and what they can do. In an online bookstore, a Book entity might have properties like title, author, price, and methods to check availability or apply discounts.
2. **Use cases:** Use cases contain the application-specific rules and define how the entities interact to fulfill specific scenarios or user stories. They coordinate the flow of data and actions between entities and external interfaces. They are also technology-agnostic, focusing only on business logic. The checkout use case might involve validating the shopping cart, applying discounts, calculating shipping, and processing payment, for example.
3. **Interfaces (ports):** Interfaces define contracts for how different layers of the system interact with each other. They create a boundary that separates the inner layers (entities and use cases) from the outer layers (adapters, frameworks, and drivers). This separation enables flexibility and maintainability. There might be an interface for payment processing that defines methods like processing payments and refunds.
4. **Adapters:** Adapters implement the interfaces and translate between the inner and outer layers. They allow the application to interact with external components like databases, APIs, or third-party libraries. Adapters allow the core logic to remain isolated from technological changes or external dependencies. A database adapter might implement a data access interface to handle interaction with a specific database technology.

5. **Frameworks and drivers (outermost layer):** This layer comprises all the technical details and tools used to build the application. It includes web servers, databases, UI frameworks, third-party libraries, etc. This layer interacts with the adapters to connect the core application to the outside world. This could include implementing a RESTful API using a specific web framework, setting up a connection to an SQL database, or integrating with a third-party payment gateway.

In Clean Architecture, each layer is independent of the others, allowing changes in one layer without affecting the others. You can switch databases, change the UI framework, or modify business logic without causing ripple effects throughout the system. Since your architecture has well-defined interfaces, it's easier to create mocks or stubs for testing. Core business logic can be tested independently from databases, UI, or other external dependencies.

While using Clean Architecture, make sure to avoid over-engineering. For simple or small projects, the complexity and overhead of Clean Architecture might need to be revised. It requires careful consideration of whether the benefits outweigh the increased complexity and development time.

Clean Architecture provides a robust and flexible foundation for developing software that can adapt to changing technologies and requirements. Focusing on separating concerns and clear boundaries between layers promotes maintainability, scalability, and testability. It's a robust pattern that can serve well in complex systems but must be applied with an understanding of the needs and context of the specific project to avoid unnecessary complexity.

Now, you have learned about various architectural patterns and best practices. Let's learn about key anti-patterns that you should be careful of when designing application architecture.

Avoiding anti-patterns in solution architecture

In this chapter, you have learned about a different way of designing solution architecture with various design patterns. Often, teams can drift away from best practices due to timeline pressure or the unavailability of resources. It is advised to try and avoid the following architecture design anti-patterns. An anti-pattern serves as an example of a poorly designed system:

- In an anti-pattern, scaling is handled reactively and manually. When application servers reach their maximum capacity and have no more resources available, users face disruptions in accessing the application. It's only when users start reporting issues that the administrator becomes aware of the problem. The admin then initiates the process of launching a new server instance to alleviate the load on existing servers. However, there's a drawback to this approach as there's typically a delay of a few minutes between the instance launches and its actual availability. During this intervening period, users experience service interruptions and are unable to access the application. You should take a proactive approach and use auto-scaling to add processing power when servers reach a certain threshold, like 60% CPU or 60% memory utilization.
- With anti-patterns, automation is missing. When application servers crash, the admin manually launches and configures the new server and notifies the users manually. Automating the detection of unhealthy resources and launching replacement resources can streamline operations. Furthermore, it's possible to implement automated notifications when such resource changes occur.

- With anti-patterns, the server is kept for a long time with hardcoded IP addresses, which prevents flexibility. Over time, server configurations can become inconsistent, leading to the inefficient allocation of resources, with some resources running when they are not needed. It would help if you kept all of the servers identical and had the ability to switch to a new IP address. You should automatically terminate any unused resources.
- With anti-patterns, an application is built monolithically, where all layers of the architecture, including web, application, and data layers, are tightly coupled and server-dependent. If one server crashes, it brings down the entire application. Keep the application and web layers independent by adding a load balancer in between. In the event that one of the application servers becomes unavailable, the load balancer automatically redirects all traffic to the remaining healthy servers.
- With anti-patterns, the application is server-bound, and the servers communicate directly with each other. User authentication and sessions are stored in the server locally, and all static files are served from the local server. You should create a service-oriented RESTful architecture, where the services talk to each other using a standard protocol such as HTTP. User authentication and sessions should be stored in low-latency distributed storage to scale the application horizontally. The static asset should be stored in centralized object storage decoupled from the server.
- With anti-patterns, a single database is used for all kinds of needs. You use a relational database for all needs, which introduces performance and latency issues. You should use the right storage for the right need, such as the following:
 - NoSQL to store the user session
 - Cache data storage for low-latency data availability
 - Data warehouse for reporting needs
 - Relational database for transactional data
- With anti-patterns, you will find a single point of failure by having a single database instance to serve the application. Whenever feasible, remove single points of failure from your architecture. Establish a secondary server (standby) and replicate the data. In the event of a primary database server failure, the secondary server can take over the workload.
- With anti-patterns, static content such as high-resolution images and videos are served directly from the server without caching. It would be best if you considered using a CDN to cache heavy content near the user location, which helps to improve page latency and reduce page load time.
- With anti-patterns, you can find security loopholes that open server access without a fine-grained security policy. You should always apply the principle of least privilege, which means starting with no access and only giving access to the required user group.

The preceding points provide some of the most common anti-patterns. Throughout this book, you will learn the best practices for avoiding them in solution design.

Summary

This chapter delved into constructing robust and scalable software architectures through various architectural paradigms. It began with exploring n-tier layered architecture, dissecting the essential components comprising the web, application, and database layers. The discussion transitioned into the intricate world of multi-tenant **Software-as-a-Service (SaaS)** architecture, delving into the complexities and benefits of accommodating diverse user bases within a unified framework.

As for web services, the chapter dove into the RESTful architectural style, elucidating its principles and applications. This was followed by a journey through constructing a RESTful e-commerce architecture, offering practical insights into real-world implementation.

Cache-based architectures were then discussed, with a comprehensive exploration of cache distribution, proxy patterns such as cache proxy and rewrite proxy, and efficient caching strategies like app caching. A comparative study of Memcached and Redis shed light on selecting the optimal caching solution.

The significance of architectural patterns was underscored by exploring the **Model-View-Controller (MVC)** approach and **Domain-Driven Design (DDD)** methodology, empowering architects to create structured, adaptable, and maintainable systems.

Architectural resilience was covered through an in-depth section on the circuit breaker pattern and on implementing the bulkhead pattern for enhanced system stability. Coverage of the floating IP pattern further enriched your toolkit for achieving high availability.

The chapter delved into containerization, unearthing the manifold benefits of containers and providing a roadmap for effective container deployment. Database handling strategies were examined within application architecture, looking at high-availability patterns to ensure data integrity and continuous operation.

The chapter concluded by spotlighting the Clean Architecture principles and imparting strategies for avoiding detrimental anti-patterns in solution architecture.

By embarking on this architectural expedition, you have gained profound insights into the intricacies of building resilient, scalable, and future-ready software systems, and you are now armed with the knowledge needed to navigate the dynamic landscape of modern technology.

Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below to get a free eBook of your choice.



5

Cloud-Native Architecture Design Patterns

In an era of rapid digital transformation, businesses increasingly turn to the cloud to deliver scalable, resilient, and cost-effective solutions. Adopting cloud-native architectures is becoming a strategic necessity for organizations seeking agility, innovation, and operational efficiency. This chapter will guide you through the journey of designing and implementing cloud-native architectures, focusing on patterns, designs, and best practices.

The chapter will comprehensively cover various cloud-native design patterns, including design principles and real-world examples. In addition to architecture design patterns, you'll also acquire knowledge on the anti-patterns of cloud-native architecture design, providing you with insights into the practices to avoid.

You will learn about the following topics in this chapter.

- What is cloud-native architecture?
- Building serverless architecture
- Building stateless and stateful architectural designs
- Creating a microservice architecture
- Reactive architecture
- Building queue-based architecture
- Pipes-and-Filters Architecture
- Creating Event-Driven Architecture
- Backend for Frontend (BFF)
- Cloud-native architecture anti-patterns

By the end of this chapter, you will have a solid understanding of cloud-native architecture patterns and be well equipped to design, build, and optimize your cloud-native solutions.

What is cloud-native architecture?

In *Chapter 3, Cloud Migration and Hybrid Cloud Architecture Design*, you were introduced to different strategies for cloud migration, including lift and shift, replatform, repurchase, retire, and others. To fully leverage the advantages and pricing models of the cloud, it's crucial to adopt cloud-native architecture. Cloud-native architecture refers to a design approach for building and running applications that harness the benefits and capabilities of cloud computing to their fullest extent. It involves crafting applications to be efficient, scalable, and resilient in dynamic cloud environments.

Cloud-native applications are developed with principles that tap into cloud services, automation, and modern development practices. The key characteristics of cloud-native architecture include:

- **Microservices:** Cloud-native applications are often composed of smaller, loosely coupled services known as microservices. Each microservice handles a specific business capability and can be developed, deployed, and scaled independently.
- **Serverless computing:** Cloud-native applications frequently leverage serverless computing to achieve seamless scalability and cost reduction. This approach allows developers to focus on their code and application logic without worrying about managing servers, enabling automatic scaling and efficient resource use, which can significantly lower operational costs. Serverless architecture packages applications and their dependencies, ensuring consistency across different environments. It facilitates seamless deployment, scaling, and portability of applications.
- **Elasticity and scalability:** Cloud-native applications can scale up or down based on demand, enabling efficient resource utilization and cost savings. This is achieved through automatic scaling and load balancing.
- **Resilience and fault tolerance:** Cloud-native applications are designed to be resilient to failures. They incorporate practices such as redundancy, automated recovery, and fault tolerance mechanisms to ensure continuous operation even in the face of failures.
- **Automation:** Cloud-native architectures emphasize automation for various processes, including deployment, scaling, monitoring, and recovery. Automation reduces manual intervention, increases efficiency, and reduces the risk of human errors.
- **DevOps practices:** Cloud-native development encourages close collaboration between development and operations teams, promoting a culture of continuous integration, continuous delivery, and rapid iteration.
- **Statelessness:** Cloud-native applications are designed to be stateless, meaning that each component does not rely on the local state of a server. This enhances scalability and allows for easier horizontal scaling.
- **API-first:** APIs (application programming interfaces) are crucial in cloud-native architecture. Applications are designed with clear and well-documented APIs, enabling communication between microservices and promoting integration with other services.
- **Continuous monitoring and improvement:** Cloud-native applications are continuously monitored to ensure optimal performance and reliability. Data-driven insights are used to identify areas for improvement and optimization.

When transitioning applications to the cloud, it's not merely about moving them as they are. Instead, it's an opportunity to optimize and utilize the cloud's features for maximum advantage. First and foremost, the pay-as-you-go model in the cloud is a game-changer. It means you pay only for the resources you use, aligning costs directly with your actual consumption. This provides elasticity and cost efficiency, as you can scale up or down based on demand without investing in fixed infrastructure. Planning resource provisioning carefully is essential to avoid over-provisioning and unnecessary costs.

The global infrastructure available in the cloud is another significant benefit. You can deploy your application closer to your users in various regions, reducing latency and improving user experience. This global reach empowers you to cater to a broader audience without investing in physical data centers worldwide.

The shift from **capital expenditure (CapEx)** to **operational expenditure (OpEx)** is a significant financial advantage in the cloud. Rather than upfront investments in hardware and maintenance, costs are spread out over time. This aligns better with budget planning and allows you to allocate resources more efficiently. However, with distributed teams and applications, cost management becomes a challenge. It's crucial to establish effective cost-control measures across different teams.

Cloud-native architecture allows organizations to fully harness the benefits of cloud computing, including scalability, flexibility, and cost-effectiveness. Consider an example of a media streaming application to highlight the distinctions and advantages of cloud-native architecture with a serverless approach compared to an on-premises architecture.

In a cloud-native architecture, the media streaming application is designed using microservices and serverless computing. Different aspects of the application, such as user authentication, content recommendation, video encoding, and storage, are each developed as separate microservices. These microservices are encapsulated in serverless functions, allowing them to execute in response to specific events or triggers. For instance, video encoding functions can be automatically invoked when a new video is uploaded, and content recommendation functions can respond to user interactions. Managed cloud services handle databases, storage, authentication, and even the execution of serverless functions.

The media streaming application is hosted on the company's servers and infrastructure in an on-premises architecture. The monolithic application handles all tasks, including authentication, content serving, and video processing. Scaling requires manual intervention and additional hardware procurement.

When adopting cloud-native development, it's important to be aware of the potential for provider lock-in. This means that designing your architecture with the native tools and services of a specific cloud provider, such as AWS, might not seamlessly transfer to another provider due to the unique, proprietary nature of each platform's offerings. Services across platforms may have different names, and the methods for invoking these services can vary significantly. While cloud-native features offer powerful capabilities tailored to optimize your operations on a specific platform, they can also introduce challenges if you later decide to migrate to a different cloud provider. Carefully consider the balance between leveraging these advanced features and maintaining some level of platform independence.

Embracing a cloud-native architecture with a serverless approach offers numerous advantages over traditional on-premises setups. The combination of microservices and serverless computing enables applications to deliver exceptional performance, scalability, cost efficiency, and rapid innovation while ensuring resilience and real-time responsiveness to users' dynamic demands.

Let's look into serverless architecture in more detail.

Building serverless architecture

In a traditional scenario, if you want to develop an application, you need to have a server where your desired operating system and required software can be installed. While writing your code, you need to ensure your server is up and running. During deployment, you need to add more servers to keep up with user demand and add scaling mechanisms such as auto-scaling to manage the desired number of servers to fulfill users' requests. In this situation, much effort goes into infrastructure management and maintenance, which has nothing to do with your business problem.

Serverless means no server is required to host your code, freeing you from auto-scaling and decoupling overheads while providing a low-cost model. Going serverless lets you focus on your application and write code for feature implementation without worrying about underlying infrastructure maintenance.



In relation to AWS, when you talk about serverless, the first thing that comes to mind is AWS Lambda functions, a **Function as a Service (FaaS)** provided by the AWS cloud. To make your application service-oriented, Amazon API Gateway offers you the ability to put RESTful endpoints in front of your AWS Lambda functions, helping you to expose them as microservices. Amazon DynamoDB provides a highly scalable NoSQL database, an entirely serverless NoSQL data store, and Amazon **Simple Storage Service (S3)** provides serverless object data storage.

Let's take a look at an example of serverless architecture for the delivery of a secure survey with AWS in the following diagram:

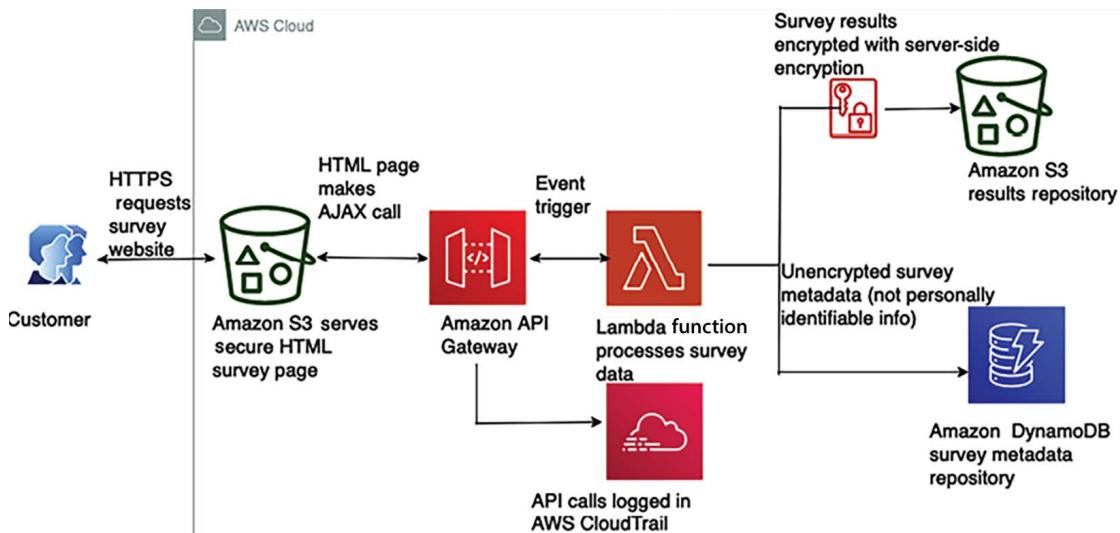


Figure 5.1: AWS Serverless architecture example for a secure survey delivery

The preceding diagram illustrates the flow of a secure serverless architecture used for a customer survey application hosted on AWS:

1. A customer makes a secure HTTPS request for the survey website. The static web page, including any client-side scripts for AJAX calls, is served directly from an Amazon S3 bucket, which is configured for web hosting.
2. Upon completing the survey, the customer submits their responses. This triggers an AJAX call from the client's browser to Amazon API Gateway. API Gateway is configured to expose the necessary endpoints for receiving survey data and is secured to ensure only authorized calls are processed.
3. Amazon API Gateway has built-in integration with AWS CloudTrail, which logs all requests made to the API. This means every survey submission is recorded, providing an audit trail that can be useful for troubleshooting lost data or investigating suspicious activities.
4. API Gateway converts the incoming AJAX call into an event that triggers an AWS Lambda function. This serverless function is responsible for processing the survey data, which may include validation, transformation, and applying business logic specific to the survey's requirements.
5. After processing the data, the Lambda function securely sends the survey results to another Amazon S3 bucket dedicated to storing these submissions. The results are encrypted using server-side encryption, ensuring that the data at rest is protected against unauthorized access.
6. Alongside the encrypted survey results, any non-sensitive metadata (excluding personally identifiable information) is concurrently stored in an Amazon DynamoDB table. This metadata could include timestamps, survey version information, or other contextual data relevant for future queries, reporting, or analytical purposes.

Due to the increasing popularity of serverless architecture, you will see more example architectures using serverless services as we move forward with this book. Now AWS SAM (**S**erverless **A**pplication **M**odel) offers straightforward syntax for creating functions, APIs, and databases tailored for serverless environments. Let's learn more about design considerations for serverless architecture.

Considerations for serverless architecture

When crafting a serverless architecture, it's crucial to account for key factors that ensure the successful deployment and functioning of your application. Serverless architecture is highly suitable for designs that can be broken down into more modular components. This approach shines when you can delineate your application into discrete, independently scalable services. However, if your project involves constructing a large, complex logic within a single, monolithic module, it might be more advantageous to opt for a traditional server-based approach.

Serverless architecture, while offering numerous benefits, often encounters the challenge of cold starts, which can affect application start latency. Although the infrastructure appears serverless to the user, cloud providers like AWS operate by creating an abstraction layer in the background, dynamically spinning up servers as needed. This process can sometimes take time, leading to a delay—or “cold start”—when the function is invoked after being idle. It's important to be mindful of cold start issues when designing with serverless architecture and to implement strategies to mitigate this, ensuring that your application remains responsive and performs efficiently.

Let's explore this through an example: developing a real-time notification system for a social media platform. The system must send instant notifications to their devices whenever users receive likes, comments, or new friend requests. Here are some critical considerations for serverless architecture for our notification system:

- **Granular function design:** Break down your application logic into small, discrete functions. Each function should perform a specific task or handle a particular event. This granularity ensures efficient resource usage and better scalability. You might have separate functions for sending likes, comments, and friend requests.
- **Statelessness:** Serverless functions are designed to be stateless. Any required state should be managed externally, such as in a database or storage service. This ensures that functions can scale and be easily replaced without affecting application behavior. Ensure that each function is stateless and doesn't rely on local memory. All necessary data, such as user preferences or notification history, should be stored in a database.
- **Event-driven design:** Serverless architecture is well suited for event-driven applications. Design your functions to trigger in response to specific events, such as user actions or changes in data. For instance, when a user receives a new friend request, an event should trigger the corresponding function.
- **Cold starts:** Serverless functions can experience a delay when invoked for the first time, known as a "cold start." This could delay the delivery of notifications, so the architecture should be designed to minimize the impact of cold starts, such as by using provisioned concurrency to keep a certain number of function instances warm and ready to handle incoming requests.
- **Scalability:** Serverless platforms automatically scale functions based on demand. This allows your application to handle sudden spikes in traffic without manual intervention. The system will handle more notifications without manual intervention as user activity increases.
- **Performance considerations:** Understand the limitations of serverless platforms, such as execution time limits and memory constraints. Optimize your functions for performance to ensure your notification system remains responsive, even during high-traffic periods.
- **Distributed tracing and monitoring:** Implement monitoring and distributed tracing to gain visibility into the performance of your serverless functions. This will be crucial for identifying bottlenecks and diagnosing issues in delivering notifications.
- **Security:** Implement security best practices for serverless applications to avoid unauthorized access to notifications. This includes proper authentication, authorization, and data encryption at rest and in transit.
- **Cost management:** While serverless can be cost-effective, monitoring usage and costs is essential. Set up budget alerts and use cloud provider tools to analyze spending patterns. With serverless, you pay for execution time, so optimize code to reduce this and consider using cost analysis tools to monitor usage.
- **Data storage and persistence:** Choose appropriate storage solutions for your data, such as managed databases, object storage, or data warehouses. Ensure data persistence across function invocations. For our notification system, we'll store user preferences and notification history in a managed database, ensuring data persistence across function invocations.

- **Dependencies:** Be mindful of dependencies in your functions. Including unnecessary libraries or components can increase the size of your deployment package and impact performance. Minimize dependencies to keep the function deployment package small and efficient.
- **Testing and debugging:** Develop effective testing strategies for your serverless functions. Use local emulators and debugging tools provided by the cloud provider.
- **Leveraging managed services:** Serverless doesn't mean every component must be a function. Use managed services for other parts of your application architecture, such as databases, queues, and authentication.
- **Compliance and regulations:** Consider any compliance or regulatory requirements that apply to your application, especially when dealing with sensitive data or industries with strict regulations. Ensure the architecture complies with data protection regulations, especially when handling personal information.

By carefully addressing these considerations, you can create a well-architected serverless application that benefits from auto-scaling, cost efficiency, and simplified management. The serverless architecture ensures scalable, cost-effective, and responsive notification delivery without worrying about managing infrastructure.

When developing a serverless architecture, emphasizing statelessness is crucial. By designing stateless applications, you reduce the dependency on server-managed session states, which in turn aids scalability. Stateless architecture is key to scaling cloud-native architecture. Let's learn more about it.

Building stateless and stateful architectural designs

Stateless and stateful architectural designs represent two different approaches to managing client-server interactions within software applications. Stateless architectures treat each client request as a separate, independent transaction, requiring no knowledge of previous interactions; this simplifies design and enhances scalability, as any server can respond to any request without needing to maintain session information. On the other hand, stateful architectures retain client session information across multiple requests, allowing for more personalized and context-aware interactions but at the cost of increased complexity in managing session data and challenges in scaling, as the state must be consistently available and synchronized across server instances.

While designing a complex application such as an e-commerce website, you need to handle the user state to maintain activity flow, where users may be performing a chain of activities such as adding to the cart, placing an order, selecting a shipping method, and making a payment. Users can use various channels to access an application, so there is a strong possibility that they will be switching between devices—for example, adding items to the cart from their mobile and then completing checkout and payment from a laptop. To cater to this situation, you should persist user activity across devices and maintain their state until the transaction is complete. Therefore, your architecture design and application implementation must plan for user session management to fulfill this requirement.

To persist user states and make applications stateless, user session information needs to be stored in persistent database layers such as the NoSQL database. This user state can be shared between multiple web servers or microservices.

Traditionally, a monolithic application uses stateful architecture, storing user session information in the server rather than via any external persistence database storage.

The key distinction between stateless and stateful application designs lies in how they handle session storage. In stateful applications, session information is stored locally on the server, which means it cannot be easily shared with other servers. This setup poses a challenge for scalability and is not well suited for modern microservice architectures, as it requires all subsequent requests from the same user to be routed to the original server that handled the first request. This can significantly restrict the application's ability to scale across multiple servers or instances. On the other hand, stateless designs do not store session data on the server, allowing any server to handle any request, which enhances the application's scalability and flexibility. The choice between adopting a stateless or stateful approach hinges on the application's requirements, specifically how it balances the need for scalability with the desire for a continuous, personalized user experience.

Stateful architecture

In a stateful application, state information is handled by the server, so once users establish a connection with a particular server, they have to stick with it until the transaction completes. You can put a load balancer in front of the stateful application, but to do that, you have to enable sticky sessions in a load balancer.

Sticky sessions are a technique used to ensure that all requests from a particular user session are directed to the same server that handled the initial request. This approach is necessary in stateful applications to maintain session consistency, as it prevents session data from being lost when subsequent requests are routed to different servers. By using sticky sessions, the load balancer deviates from its standard practice of distributing requests evenly among servers, typically done via a round-robin method, and instead, routes a user's requests to a specific server where their session information resides. While this method supports session persistence, it introduces challenges, such as the potential for overloading a single server with too many persistent connections. To mitigate this, implementing a session timeout mechanism becomes essential, ensuring that sessions do not indefinitely consume server resources.

Often, a stateful application doesn't support horizontal scaling very well, as the application state persists in the server, which cannot be replaced. The stateful application works well early on when the user base is small. However, as the internet becomes increasingly widespread, it is reasonable to assume that you will have millions of users active on a web application. Therefore, efficient horizontal scaling is essential for handling a large user base and achieving low application latency.

Stateless architecture

Using the stateless method, your design approach should focus more on the shared session state, as it allows horizontal scaling.

The following diagram shows an architecture that depicts a stateless application for an example web application with AWS:

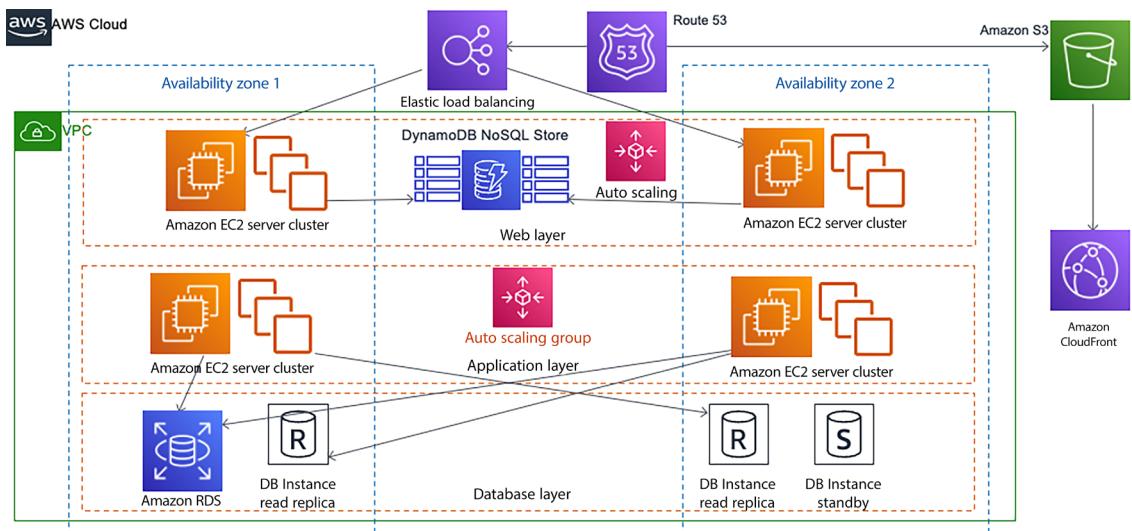


Figure 5.2: A stateless application architecture

The depicted AWS architecture provides a secure, highly available, and scalable environment for a three-tier application across two Availability Zones for fault tolerance. It uses Elastic Load Balancing to distribute traffic across EC2 server clusters, which are dynamically scaled with Auto Scaling to meet changing demands. The database layer, powered by Amazon RDS, includes a read replica for query scaling and a standby instance for failover, ensuring data durability and high availability. Static content is served through Amazon S3 and delivered efficiently via Amazon CloudFront, with AWS Route 53 managing DNS services to optimize user traffic routing. This setup ensures operational resilience, cost-efficiency, and performance optimization for the application. To make applications loosely coupled and scalable, all user sessions are stored persistently in the NoSQL database, for example, Amazon DynamoDB.

For the session ID, you should use client-side storage, such as cookies. This architecture lets you scale the application horizontally by adding more servers without worrying about losing user state information. A stateless architecture removes the overhead of creating and maintaining user sessions and allows consistency across the application's modules. A stateless application has performance benefits, too, as it reduces memory usage from the server side and eliminates the session timeout issue.

Implementing a stateless architecture involves complexities such as integrating additional database components for storing user sessions and creating a supplementary layer to retrieve the correct user session across servers. However, with the right approach, you can achieve a rewarding experience for your user base. You can develop applications using the microservice approach with REST design patterns and deploy them in containers. For this, use authentication and authorization to connect users to the server.

In the following sections, you will learn more about microservices and REST design patterns. As access to user session information from multiple web servers focuses on a single data storage location, you must use caution to prevent the performance of the data store from becoming a bottleneck.

Creating a microservice architecture

In cloud-native architecture, microservices play a vital role in breaking down extensive features into smaller, manageable chunks that can scale independently. This approach allows for specific components to be scaled up or down as needed without affecting the entire system. By using microservices, a system is designed to be fault-tolerant, meaning it's constructed with potential failures in mind, allowing for the graceful degradation of application availability and preventing widespread system failures.

The clear advantage of microservices is that you have to maintain a smaller code surface area. Microservices should always be independent. You can build each service with no external dependencies where all prerequisites are included, which reduces the inter-dependency between application modules and enables loose coupling.

The other overarching concept of microservices is **bounded contexts**, which are the blocks that combine to make a single business domain. A business domain could be retail, car manufacturing, bookselling, or social network interactions involving a complete business process. An individual microservice defines boundaries in which all the details are encapsulated. For example, let's consider an e-commerce platform. In such a system, you would have several microservices handling different aspects of the business. Here are a few bounded contexts within this platform:

1. **User account context:** This microservice handles everything related to user accounts, including user registration, profile management, login, and authentication. Its boundary encompasses user information and the operations that can be performed on this data, such as updating a profile or resetting a password. No other microservice will manage these operations.
2. **Product catalog context:** This microservice is responsible for managing the product listings, categories, and product details. It operates independently of the user account context, focusing solely on the products, their organization, and their presentation to the user.
3. **Order processing context:** This microservice handles the checkout process, order tracking, and payment processing. It uses information from the product catalog context (e.g., product IDs, prices) and the user account context (e.g., customer details) to fulfill its functions but maintains its separate operations, such as updating order status or processing returns.

Each bounded context is a self-contained system with its own domain logic and database, communicating with others via well-defined APIs. These boundaries allow each microservice to be developed, deployed, scaled, and updated independently, making the overall system more resilient and adaptable to change.

By defining these boundaries, the e-commerce platform can ensure that changes in one context, such as adding new payment methods in the order processing context, do not affect the user account or product catalog contexts, leading to a more maintainable and scalable system.

Scaling each service is essential while dealing with the large-scale access of applications, where different workloads have different scaling demands.

Let's learn about some best practices for designing microservice architecture:

- **Create a separate data store:** Adopting a separate data store for each microservice allows the individual team to choose the best database for their service. For example, the website traffic team can use a scalable NoSQL database to store semi-structured data. The team handling order services can use a relational database to ensure data integrity and the consistency of transactions. This also helps to achieve loose coupling where changes in one database do not impact other services.
- **Keep servers stateless:** As you learned in the previous section, *Building stateless and stateful architecture designs*, keeping your server stateless helps in scaling. Servers should be able to go down and be replaced easily, with minimal or no need for storing state on the servers.
- **Create a separate build:** Creating a separate build for each microservice makes it easier for the development team to introduce new changes and improve the agility of the new feature release. This helps to ensure that the development team is only building code required for a particular microservice and not impacting other services.
- **Deploy in a container:** Deploying in a container gives you the tool to deploy everything in the same standard way. Using containers, you can choose to deploy all microservices in the same way, regardless of their nature. You can use serverless container deployment services like Amazon Fargate to manage your container without worrying about infrastructure.
- **Go serverless:** Try to use a serverless platform or a leveraging function with service capability, such as AWS Lambda, when your microservices are simple enough. Serverless architecture helps you to avoid infrastructure management overhead.
- **Blue-green deployment:** For application deployment, the best approach is to create a copy of the production environment. Deploy the new feature and route a small percentage of the user traffic to ensure the new feature is working as expected in a new environment. After that, increase the traffic in the new environment until the entire user base can see the new feature. You will learn more about blue-green deployment in *Chapter 11, DevOps and Solution Architecture Framework*.
- **Monitor your environment:** Good monitoring is the difference between reacting to an outage and proactively preventing an outage with proper rerouting, scaling, and managed degradation. To prevent application downtime, you want services to offer and push their health status to the monitoring layer because what knows more about status than the service itself? Monitoring can be done in many ways, such as with plugins or by writing to a monitoring API.

While microservice architectures have various advantages, a modular approach comes with the overhead of managing more infrastructure. You must carefully choose the tools to help you manage and scale multiple modules in parallel. While designing microservice architecture, try to use serverless platforms wherever possible, which will help mitigate the infrastructure and operation overhead. Let's look at a microservice-based example architecture for a real-time voting application.

In the diagram below, we show a design that uses microservices for a live voting app. This app works by having small, separate services that handle and count votes from users. When someone votes using their mobile device, the app records each vote and then saves all these votes together in a NoSQL database, Amazon DynamoDB.

There is application logic in the AWS Lambda function, which aggregates all of the voting data cast by users to their favorite actor and returns the final results:

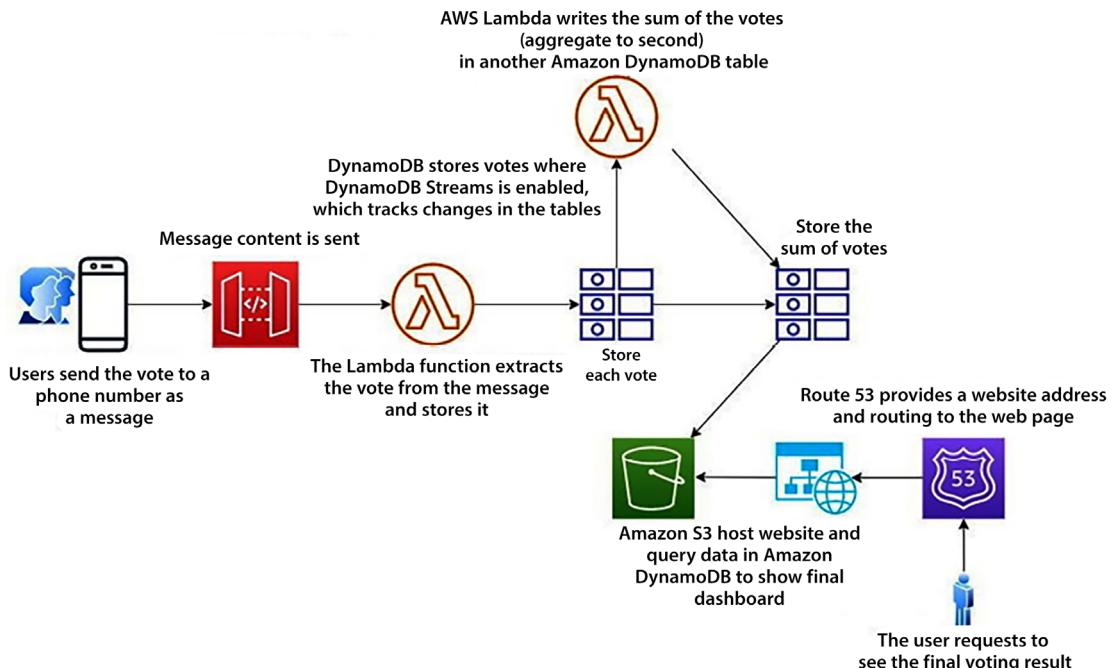


Figure 5.3: Microservice-based real-time voting application architecture with AWS

In the preceding architecture, the following things are happening:

1. Users text a vote to a phone number or a short code provided by a third party such as *Twilio*.
2. The third party is configured to send the content of the message to an endpoint created by Amazon API Gateway, which then forwards the response to a function built in AWS Lambda.
3. This function extracts the vote from the message content and writes the result and any metadata into a table in Amazon DynamoDB.
4. This table has DynamoDB Streams enabled, which tracks changes to your tables on a rolling basis.
5. After the update, DynamoDB Streams notifies a second AWS Lambda function with the application logic to aggregate the votes (to every second) and write them back to another DynamoDB table. The second table only stores the sum of the votes for each category.
6. A dashboard to display a summary of votes is created using HTML and JavaScript and hosted as a static website in Amazon S3. This page uses the AWS JavaScript SDK to query the aggregate Amazon DynamoDB table and display the voting results in real time.
7. Finally, Amazon Route 53 is a DNS provider for creating a hosted zone pointing to a custom domain name in the Amazon S3 bucket. This allows you to host static websites in S3 buckets in a cost-effective serverless manner.

This architecture is not only microservice-based but also serverless. Using microservices, you can create applications made of small independent components, which constitute smaller parts to iterate. Microservice-based architecture means that the cost, size, and risk of change are reduced, increasing the rate of change.

Coordinating between multiple services becomes critical if your system is distributed using microservices. Let's learn how to orchestrate multiple microservices next.

Saga pattern

The Saga pattern is a design pattern used to manage long-running, complex business transactions. It's beneficial in microservice architectures, where a single business transaction might involve multiple microservices. Instead of using a traditional two-phase commit, the Saga pattern divides the transaction into multiple smaller, isolated transactions. A different service handles each of these smaller transactions, and they are coordinated to ensure data consistency across services. If one of the smaller transactions fails, compensating transactions are executed to undo the previous steps.

In complex systems where multiple services need to work together to fulfill a single operation, such as processing an order or booking a flight, the Saga pattern helps ensure that if something goes wrong at any point, the entire operation can be either fully completed or rolled back.

Here's how the Saga pattern works:

- **Decomposition:** The operation that needs to be performed is broken down into smaller, isolated steps or transactions. Each step corresponds to an action performed by a specific microservice.
- **Compensation actions:** For every step, a corresponding compensation action is defined. If a step fails or an error occurs, the compensation action is executed to reverse the effects of the previous steps. This brings the system back to a consistent state.
- **Coordinator:** A coordinator is responsible for orchestrating the sequence of steps and their corresponding compensation actions. It initiates the saga, monitors its progress, and ensures that all steps are completed or the necessary compensation actions are taken.
- **Local transactions:** Each step and its compensation action are encapsulated within a local transaction within their respective microservices. This allows for the atomicity of operations within each microservice.
- **Eventual consistency:** The Saga pattern embraces eventual consistency, which means that even if a failure occurs, the system will eventually reach a consistent state by either completing the entire operation successfully or rolling back to a consistent state.

Imagine an e-commerce application where a customer places an order. The Saga pattern could be used to handle the entire order processing flow:

1. **Initiation:** The order service starts a new saga for order processing.
2. **Steps:** The saga involves multiple steps performed by different microservices: check product availability, charge the customer, update inventory, and notify the customer, for example.
3. **Compensation actions:** Corresponding compensation actions are defined, for example, if the item is out of stock: release the charged amount, restock the product, and send an apology email to the customer.

4. **Coordinator:** A coordinator oversees the saga, ensuring that each step is successfully executed or compensated. For example, steps flow from checking product availability to placing an order, charging the customer, and fulfilling the order for delivery.
5. **Eventual consistency:** If a step fails at any point (for example, if charging the customer fails), the compensation actions are triggered to bring the system back to a consistent state.

Each service involved in the Saga produces and listens for events, as shown in the following diagram:

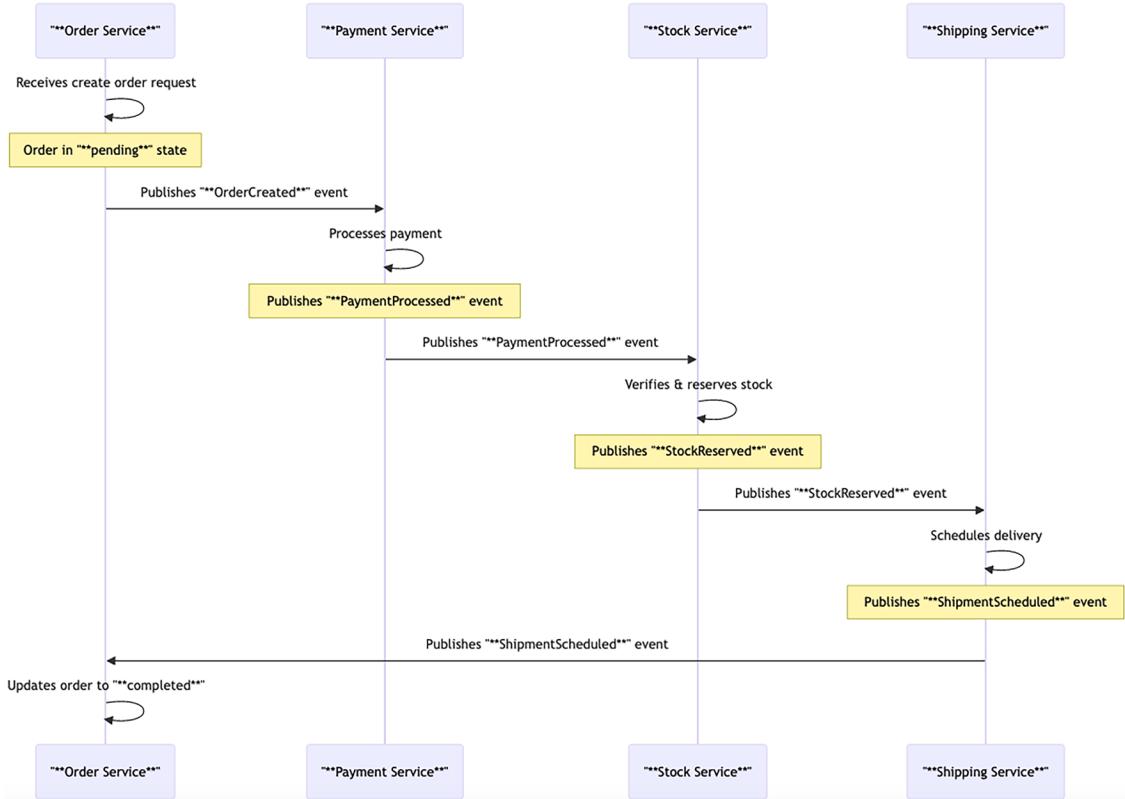


Figure 5.4: Saga pattern sequence diagram for e-commerce application architecture

As depicted in the preceding diagram, when a service completes its part of the transaction, it produces an event that triggers the next service in the saga. For example:

1. The Order Service receives a request to create an order.
2. The Order Service starts the saga by creating the order in a **pending** state and publishing an **OrderCreated** event.
3. The Payment Service listens for the **OrderCreated** event, processes the payment, and publishes a **PaymentProcessed** event.
4. The Stock Service listens for the **PaymentProcessed** event, verifies that the items are in stock, reserves the stock, and publishes a **StockReserved** event.
5. The Shipping Service listens for the **StockReserved** event, schedules the delivery, and publishes a **ShipmentScheduled** event.

6. The Order Service listens for the **ShipmentScheduled** event and updates the order to a **completed** state.

If any of the services fail to complete its part of the transaction, it publishes a compensating event to trigger the rollback of the previous steps. For example, if the Stock Service finds out there's insufficient stock, it could publish a **StockInsufficient** event. The Payment Service would listen for this event and initiate a refund. The Order Service would listen for the **StockInsufficient** event and update the order to a **failed** state.

The Saga pattern is a design solution that addresses the challenge of data consistency in distributed systems, particularly when working with microservices. Instead of relying on a single, large-scale transaction to ensure data consistency across different services, the Saga pattern breaks the transaction into a series of local transactions for each service. Each local transaction updates the database and publishes an event or message indicating the transaction's success or failure. However, the Saga pattern introduces the concept of eventual consistency, which means the system's state will become consistent over time, but not necessarily immediately. Additionally, implementing the Saga pattern can be complex because it requires handling failure scenarios and ensuring that compensating transactions correctly undo previous operations. This often involves intricate coordination and robust messaging systems to manage the asynchronous communication between services.

The Saga pattern allows complex operations to be broken down into manageable steps, with a safety net to handle failures and maintain data integrity. It promotes better resilience in distributed systems by ensuring that the system remains coherent and eventually consistent even if failures occur. However, implementing the Saga pattern requires careful design and coordination to handle various failure scenarios effectively. What if you have extensive information that needs to be processed by multiple microservices, but it needs to be consolidated to create meaningful insights? In such scenarios, the fan-out/fan-in pattern can rescue you. Let's learn more about it.

Fan-out/fan-in pattern

The fan-out/fan-in pattern is a design pattern commonly used in distributed systems to process requests efficiently and aggregate data from multiple sources. It's beneficial for scenarios where data must be collected, processed, and consolidated from various input streams or sources. The pattern gets its name from how data fans out from multiple sources and then fans back in for aggregation.

Consider a real-time analytics system for a social media platform. The fan-out/fan-in pattern can be applied to collect and process data from various user activities. Let's see how the fan-out/fan-in pattern works:

- **Fan-out phase:**
 - In the fan-out phase, data is collected from multiple sources, including different microservices, APIs, or data streams. Each source sends its data to a separate processing component. User posts, comments, likes, shares, and followers generate real-time data streams.

- The processing component for each source operates independently and simultaneously. This allows for efficient parallel processing, reducing the time to gather data from various sources. Each type of activity has a dedicated processing component that calculates statistics such as engagement rates, popular content, and trending topics.
- **Fan-in phase:** Once individual processing is complete, the results from each processing component are aggregated or combined, in this case to calculate overall platform engagement metrics. This aggregation can involve calculations, summarizations, or any other operation needed for the final result. The aggregated data generates the desired outcome or final report. This could be a single report, a summarized analysis, or any other form of consolidated data. For our example, this is presented to administrators as a dashboard that displays real-time engagement insights.

In this example, the fan-out/fan-in pattern allows the analytics system to process and consolidate data from multiple user activities efficiently, providing administrators with real-time insights into platform engagement.

Benefits of the fan-out/fan-in pattern

The fan-out/fan-in pattern is a strategic approach in distributed systems that significantly enhances the way data is managed and processed. Here are the key benefits of employing this pattern:

- **Parallelism:** The pattern leverages parallel processing, allowing faster data collection and aggregation from multiple sources.
- **Efficiency:** Instead of processing data sequentially from each source, the pattern optimizes processing time by working on multiple sources concurrently.
- **Scalability:** Each source can be processed independently, enabling the system to scale efficiently as the number of sources increases.
- **Modularity:** The pattern encourages modular design by separating the data collection (fan-out) phase from the aggregation (fan-in) phase. This makes it easier to maintain and extend the system.

The fan-out/fan-in pattern, while beneficial for parallel processing and enhancing efficiency in distributed systems, introduces specific challenges that must be navigated carefully. Implementing this pattern adds complexity due to the need for meticulous coordination between the numerous parallel tasks it initiates and their subsequent aggregation. Error handling becomes more intricate, as the system must account for potential failures in any of the fan-out tasks and ensure robust mechanisms for recovery to maintain data consistency.

This pattern can also be resource-intensive, as it may require significant computational power to manage the parallel processes, potentially leading to higher operational costs and necessitating advanced scaling strategies. Moreover, the aggregation stage can become a bottleneck, particularly if it involves processing large volumes of data, which may delay the overall data processing timeline. Additionally, the system might only achieve eventual consistency, posing challenges for applications that require real-time processing. Finally, the distributed nature of this pattern complicates debugging and monitoring, requiring comprehensive tools to ensure visibility across all tasks. Despite these challenges, with careful design and management, the fan-out/fan-in pattern remains a potent strategy for improving data processing efficiency in distributed architectures.

Overall, the fan-out/fan-in pattern is valuable for managing and processing data from various sources in distributed systems, enabling efficient parallel processing and streamlined aggregation.

Increasing the number of microservices requires careful orchestration, which is where the service mesh comes into the picture. Let's learn more about it.

Service mesh pattern

In modern software development, microservices have become a go-to approach for building flexible and scalable applications. However, as the number of microservices increases, managing their communication and reliability can become more challenging than navigating a busy road intersection. This is where the concept of the service mesh enters the picture, simplifying microservices' communication while enhancing their robustness.

Imagine you're at a bustling city intersection with multiple lanes of traffic. Each vehicle represents a microservice, serving a specific purpose. To ensure smooth traffic flow and prevent collisions, traffic lights, signs, and road rules are essential. Similarly, a service mesh acts as the traffic controller for microservices, regulating their interactions and ensuring they work harmoniously.

A service mesh is a layer of the infrastructure that manages communication between different services in a cloud application. It ensures reliable message delivery among these services. Builders can focus on core application programming, while the service mesh takes care of networking and security in the system's infrastructure.

The following diagram illustrates a service mesh infrastructure with AWS services as an example.

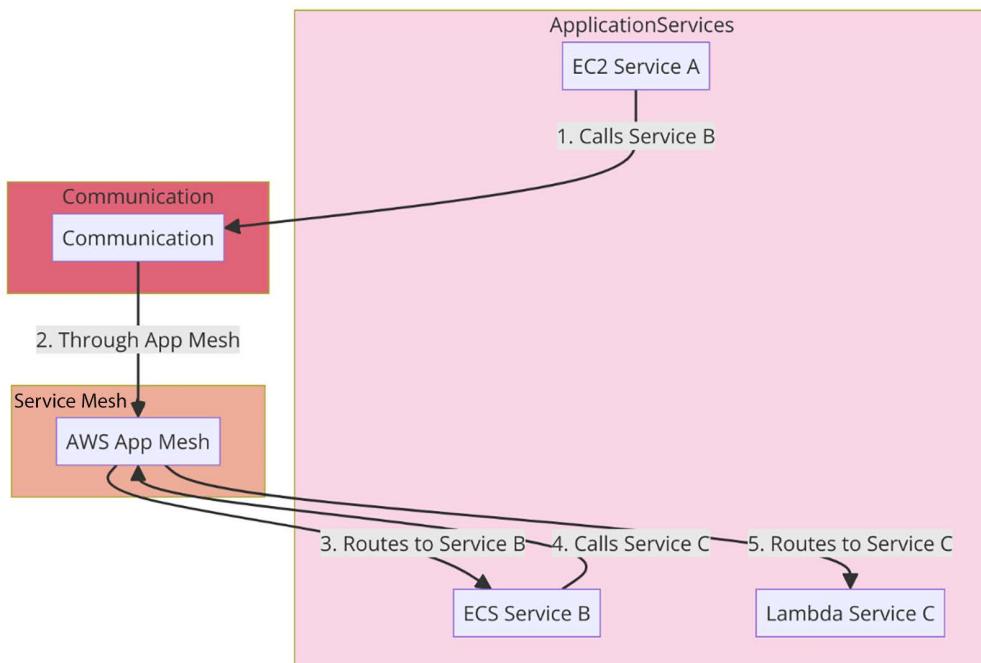


Figure 5.5: Service mesh pattern architecture in AWS cloud

Let's walk through each step illustrated in the service mesh diagram:

1. **EC2 Service A:** This represents an Amazon EC2 instance running a service (**Service A**). EC2 instances provide scalable computing capacity in the **Amazon Web Services (AWS)** cloud.
2. **Calls Service B:** Service A initiates a call to **Service B**. This is the beginning of an inter-service communication process.
3. **Communication:** This block represents the communication layer where Service A's request is captured to be routed through the service mesh.
4. **Through App Mesh:** The request from Service A goes through AWS App Mesh, which is a service mesh that provides application-level networking. App Mesh standardizes how services communicate, giving end-to-end visibility and ensuring high availability for applications.
5. **Routes to Service B:** AWS App Mesh routes the request to the appropriate service, in this case, Service B.
6. **ECS Service B:** This represents an Amazon **Elastic Container Service (ECS)** task running Service B. ECS is a highly scalable, high-performance container management service that supports Docker containers.
7. **Calls Service C:** After Service B completes its processing, it calls **Service C**. This could be part of a larger transaction that involves multiple microservices.
8. **Routes to Service C:** Again, AWS App Mesh routes the call from Service B to Service C.
9. **Lambda Service C:** This represents an AWS Lambda function for Service C. AWS Lambda lets you run code without provisioning or managing servers. It executes your code only when needed and scales automatically.

The architecture abstracts the complex interplay of services within a service mesh, illustrating the role of AWS App Mesh in managing, routing, and controlling the communication between different services.

Here are the primary features provided by a service mesh:

- **Traffic management:** Service meshes provide detailed control over traffic behavior with rich routing rules, retries, failovers, and fault injection.
- **Observability:** They give you deep insights into your applications through visualizations, tracing, monitoring, and logging traffic between services.
- **Security:** Service meshes offer automated **mutual TLS (mTLS)** traffic encryption between your services.
- **Policy enforcement:** They allow you to define and enforce policies consistently across all your services, regardless of where they run.
- **Resilience:** Service meshes enable advanced load balancing, timeouts, and retries, helping you create more resilient applications.

A popular way to implement a service mesh is by using sidecar proxies. Each service instance in a microservices application is paired with a sidecar proxy, which handles all the network communication to and from the service. All these proxies are networked into a mesh, hence the name “service mesh.”

Service meshes are becoming an essential part of modern, cloud-native application architectures, offering a variety of implementations tailored to different needs and environments. Among the most popular service mesh implementations are:

- **Istio:** This comprehensive service mesh solution provides a robust way to control service-to-service communication within a microservice architecture. It allows developers to define detailed routing rules and policies, implement resilience patterns like retries and circuit breakers, and gather insights into application traffic flows. Istio's ability to enforce policies and collect metrics helps in securing and observing communications between services, thereby enhancing the network's reliability and performance.
- **Linkerd:** Known for its focus on simplicity and performance, Linkerd is an open-source service mesh that provides critical features such as service discovery, routing, failure handling, and visibility to modern application infrastructures. It's designed to be lightweight and easy to install, with a minimal footprint that makes it an attractive choice for teams looking to adopt service mesh technology without significant overhead.
- **AWS App Mesh:** Specifically designed for AWS users, App Mesh is a managed service mesh service that makes it easy to manage and control communications between microservices across AWS services. It supports application-level networking, enabling the application services to communicate over the network with more visibility and control. AWS App Mesh simplifies the configuration of service communication, providing application-level insights and ensuring high availability for your applications.
- **Consul Connect:** Part of HashiCorp Consul, Consul Connect focuses on securing service-to-service communication with automatic TLS encryption and identity-based authorization. It's built to be platform-agnostic, providing a consistent, unified method of securing and configuring communication across services, regardless of the underlying platform. With its emphasis on security, Consul Connect ensures that only authorized services can communicate with each other, thereby reducing the risk of internal threats.

While service meshes offer a range of benefits for microservice architecture, such as improved service-to-service communication, enhanced security, and better observability, it's crucial to consider the complexity they introduce to your infrastructure. Incorporating a service mesh involves additional components to manage, monitor, and maintain, which could increase the operational overhead for your team. This added layer of infrastructure requires careful planning, skilled personnel to manage it, and a clear understanding of its impact on your system's performance and complexity. Therefore, evaluating the specific needs of your application and weighing the advantages against the potential increase in infrastructure complexity is essential before deciding to implement a service mesh. This cautious approach ensures that the benefits of adopting a service mesh align with your application's requirements and your team's capacity to manage the additional complexity.

AWS App Mesh is a service that normalizes communication across your services, offering comprehensive monitoring and promoting consistent availability. The following architecture diagram depicts the implementation of the service mesh pattern using AWS cloud services:

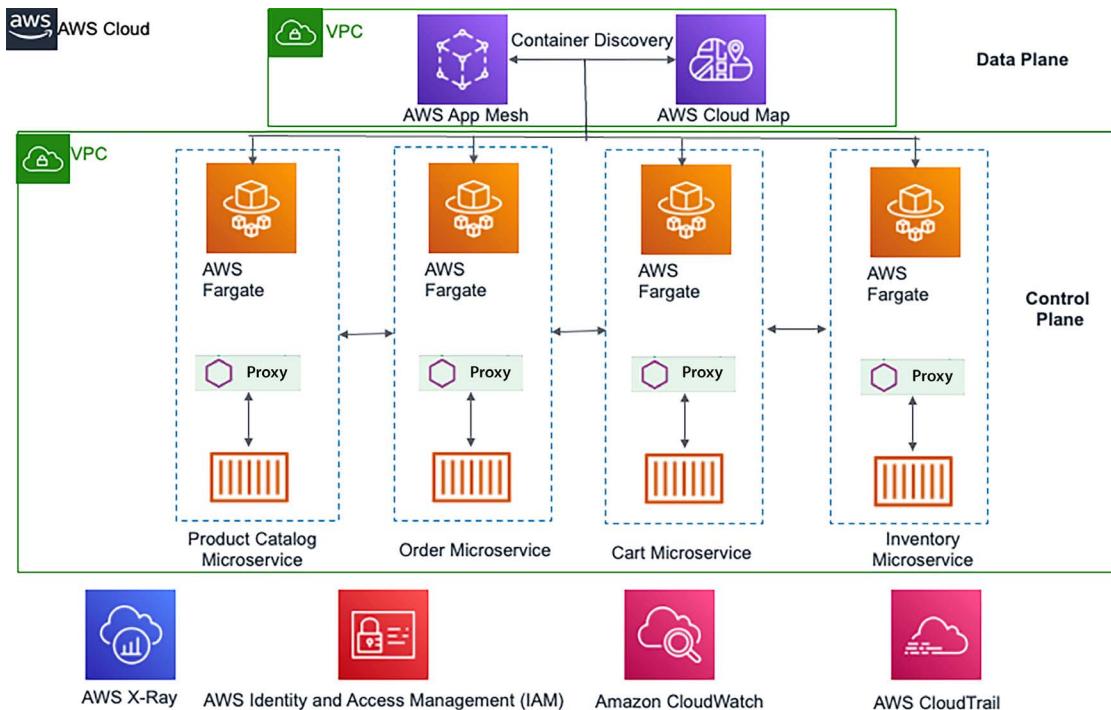


Figure 5.6 – An e-commerce application managed by App Mesh in AWS

As shown in the preceding diagram, Amazon Fargate operates as a serverless engine for container computing, compatible with Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS). The following are the steps to implement an e-commerce application managed by App Mesh:

- Create Fargate services:** Define each microservice (User, Order, Payment, Product Catalog, and Authentication) as an Amazon Fargate on EKS with the required task definitions.
- Set up AWS App Mesh:** Create a mesh that serves as a logical boundary for the network traffic between services.
- Define virtual nodes:** Create a virtual node for each ECS service in App Mesh. A virtual node acts as a logical pointer to a particular ECS service.
- Create virtual routers and routes:** Define virtual routers and routes to control the traffic flow between virtual nodes.
- Configure virtual services:** Virtual services route traffic to virtual nodes, enabling the discovery of services within the mesh.
- Deploy sidecar proxies:** Attach an Envoy proxy to each ECS task definition as a sidecar container. Envoy proxies intercept and manage the traffic between microservices.

7. **Monitor and log:** Use AWS CloudWatch and AWS X-Ray to monitor and log the traffic flowing through the mesh.

Implementing a service mesh can enhance service-to-service communication, security, and observability. This approach allows you to manage a microservice architecture more efficiently and effectively, providing a robust and scalable solution for complex applications. Recovery from failure is an important aspect of building large-scale architecture. Let's learn about reactive architecture to solve this problem.

Reactive architecture

As cloud-native architecture can have various moving parts due to multiple microservices and small modules, they need to be protected from failure. Reactive architecture is a design approach for building software that can efficiently handle changes and stay responsive under various conditions. It benefits large-scale and distributed systems that must maintain high availability and responsiveness, even in the face of failures or high demand.

The principles of reactive architecture are based on the Reactive Manifesto, a document that outlines the core traits of reactive systems: responsive, resilient, elastic, and message-driven. You can find details on the Reactive Manifesto by visiting: <https://www.reactivemanifesto.org/>:

- **Responsive:** Reactive systems prioritize responsiveness, ensuring they respond to user requests promptly regardless of the system's load or state.
- **Resilient:** Reactive systems are designed to handle failures gracefully. They can recover quickly and operate, even when some components fail.
- **Elastic:** Reactive systems can scale up or down based on demand, efficiently utilizing resources and maintaining responsiveness under varying workloads.
- **Message-driven:** In reactive systems, components communicate using messages that are passed asynchronously. This approach allows for components to be loosely connected, independently isolated, and accessible from different locations.

The reactive architecture style leans heavily on microservices, which segment functionality into smaller, independently scalable services for improved scalability, maintainability, and faster deployment cycles. Communication within reactive systems is event-driven, meaning components interact and react through asynchronous events, leading to more efficient use of resources and better system performance.

To manage data, reactive architectures adopt a decentralized approach, where each microservice manages its own data, minimizing dependency and contention over shared data resources. This not only bolsters the system's resilience but also its ability to recover swiftly from failures. Isolation and autonomy are central to reactive systems, ensuring that components can fail independently without impacting the overall system's availability, thus enhancing fault tolerance.

Scalability is achieved through horizontal scaling, where the system can grow to accommodate increased loads by adding more instances of services rather than upgrading the capacity of existing instances.

Additionally, reactive architectures implement resilience patterns such as circuit breakers, timeouts, and retries. These mechanisms help in managing and recovering from failures, preventing one component's issues from cascading into system-wide disruptions. Together, these principles facilitate the creation of systems that are more responsive to user demands, resilient to failures, and capable of graceful degradation under load.

Reactive architecture benefits large-scale, distributed systems that need to handle varying workloads, recover from failures quickly, and provide a responsive user experience.

Imagine an online gaming platform with thousands of players simultaneously interacting in virtual worlds. Reactive architecture can be applied here to ensure a seamless and responsive gaming experience:

- **Responsive:** The system quickly responds to players' actions, allowing characters to move, cast spells, and interact with objects in real time.
- **Resilient:** If a server experiences a sudden crash due to a technical glitch, the architecture automatically redistributes the load to healthy servers, ensuring uninterrupted gameplay for other players.
- **Elastic:** As more players join the game during peak hours, the architecture dynamically allocates additional server resources to handle the increased load. When player numbers decrease, surplus resources are released to save costs.
- **Message-driven:** Player actions, such as casting spells or trading items, are communicated through messages. This asynchronous communication minimizes bottlenecks and ensures smooth gameplay despite many concurrent actions.

To implement the reactive architecture, you can take the following steps:

- Design components to communicate asynchronously using message queues. This prevents blocking and enhances responsiveness.
- Implement the Actor model, where components (actors) communicate through messages. Each actor processes messages sequentially, avoiding concurrency issues.
- Integrate resilience patterns like Circuit Breaker and Bulkhead to handle failures and prevent cascading errors. You learned about these patterns in *Chapter 4, Solution Architecture Design Patterns*.
- Utilize auto-scaling mechanisms to allocate resources based on load dynamically. Cloud platforms like AWS provide tools for this purpose.
- Leverage reactive libraries or frameworks like Akka, Spring WebFlux, or ReactiveX, which offer abstractions for building reactive systems.

Let's explore how to implement reactive architecture using AWS services for an ad-tracking use case. The following diagram demonstrates reactive architecture for an ad tech company:

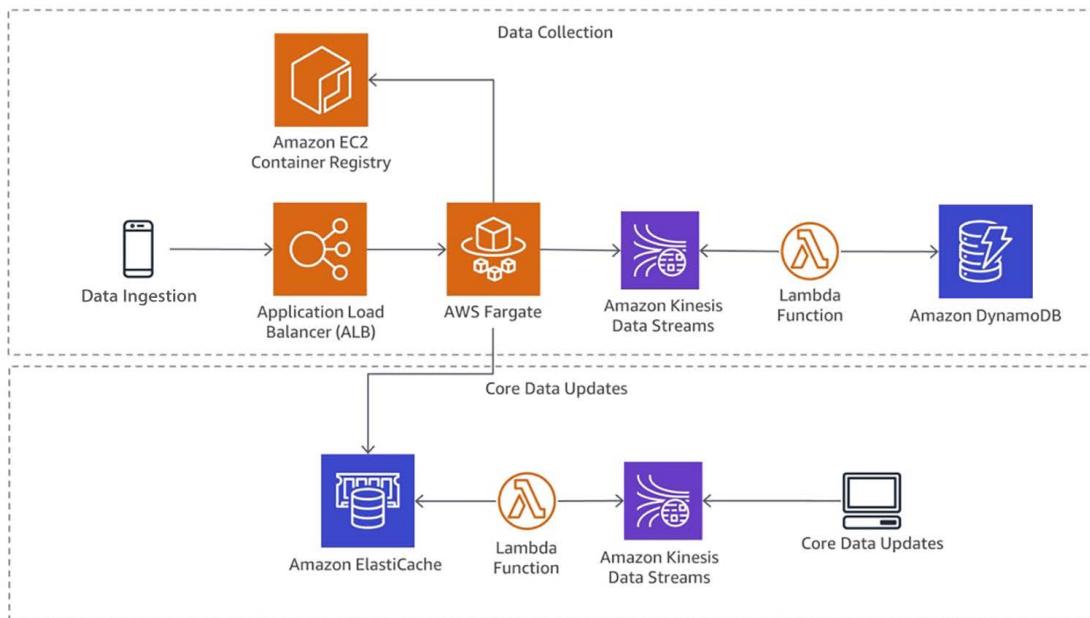


Figure 5.7 – Reference architecture for an ad-tracking application

The architecture depicted in the preceding diagram demonstrates an ad-tracking application using AWS's architecture for both real-time and batch processing.

In the given architectural layout, when a user views or clicks an ad, the application load balancer captures this request and forwards it to the appropriate service within the primary application. The application independently processes each request in a timely and robust manner, avoiding immediate database writes. Instead, Amazon Kinesis Data Streams collects these events, and an AWS Lambda function is then responsible for recording the information into an Amazon DynamoDB table. Amazon Kinesis Data Streams is a highly scalable and durable real-time data streaming service designed to collect, process, and analyze streaming data. This setup of data streams serves as a protective intermediary, ensuring no data is lost during high-traffic periods.

To optimize access speed to essential data, Amazon ElastiCache for Redis acts as the primary cache. Core data updates are synchronized through a message-passing architecture, using event streams to capture and communicate changes from all contributing systems. This arrangement allows the handling of varying request volumes, with Lambda functions processing stream data and refreshing the primary cache to ensure system integrity and performance.

Integrating these AWS services allows you to build a reactive architecture for your online ad platform. The services provided by AWS align with the core principles of responsiveness, resilience, elasticity, and message-driven communication that define a reactive system. Loosely coupled architecture plays a key role in building highly scalable cloud-native architecture, and message queues play a pivotal role in that, so let's learn about some queue-based architectural patterns.

Building queue-based architecture

In the previous sections, you learned about microservice design using RESTful architecture. The RESTful architecture helps your microservice to be easily discoverable, but what happens if your service goes down? RESTful is a contemporary architecture where your client service waits for a response from the host service, which means that the HTTP request blocks the API. Sometimes, your information may be lost due to the unavailability of a downstream service. You must implement some retry logic to retain your information in such cases.

A queue-based architecture solves this problem by adding message queues between services, which hold information on behalf of services. The queue-based architecture provides fully asynchronous communication and a loosely coupled architecture. In a queue-based architecture, your information is still available in the message. If a service crashes, the message can get the process as soon as the service becomes available. Let's learn some of the terminology of a queue-based architecture:

- **Message:** A message has two parts—the header and the body. The header contains metadata about the message, while the body contains the actual message.
- **Queue:** The queue holds the messages that can be used when required.
- **Producer:** A service that produces and publishes a message to the queue.
- **Consumer:** A service that consumes and utilizes the message.
- **Message broker:** This helps to gather, route, and distribute messages between the producer and consumer.

Let's explore some typical queue-based architecture patterns to understand how they work.

Queuing chain pattern

A queuing chain pattern is applied when sequential processing needs to run on multiple linked systems. Let's understand the queuing chain pattern using the example of an image-processing application. In an image-processing pipeline, sequential operations of capturing the image and storing it on a server, running a job to create different-resolution copies of the image, watermarking the image, and thumbnail generation are tightly linked to each other. A failure in one part can cause the entire operation to be disrupted.

You can use queues between various systems and jobs to remove a single point of failure and design true loosely coupled systems. The queuing chain pattern helps you to link different systems together and increases the number of servers that can process the messages in parallel. You can configure auto-scaling to terminate the excess servers if there is no image to process.

The following diagram shows the queuing chain pattern architecture for our image-processing application. Here, the queue provided by AWS is called Amazon Simple Queue Service (SQS):

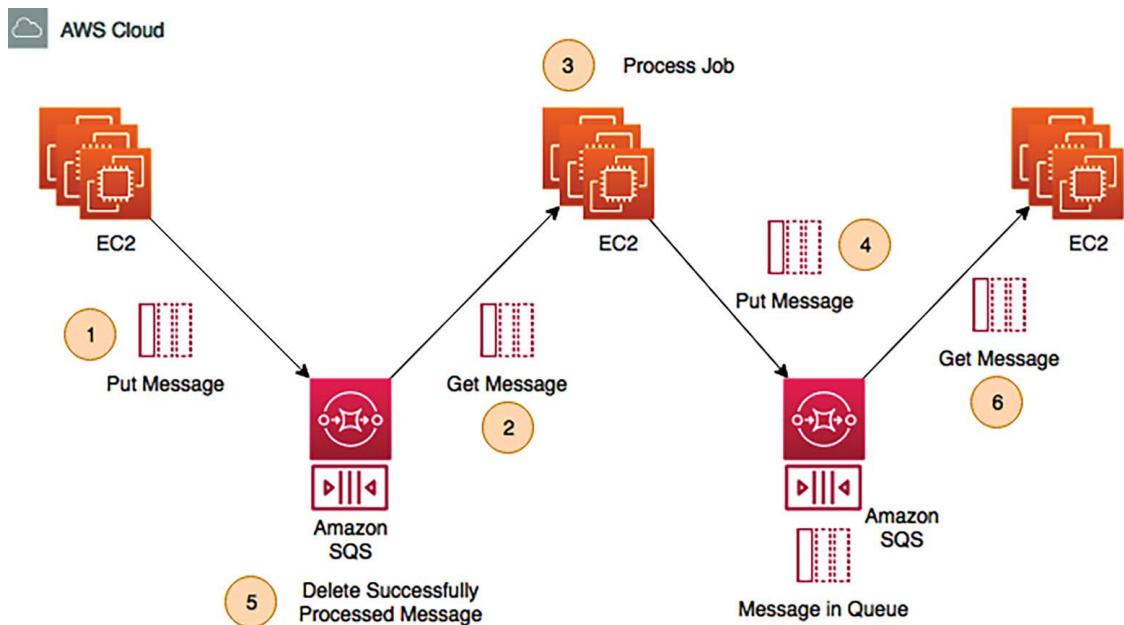


Figure 5.8: Queuing chain pattern architecture

The preceding architecture has the following steps:

1. When the raw image is uploaded to the server, the application must watermark all images with the company's logo. A fleet of Amazon EC2 (**Elastic Cloud Compute**) servers runs batch jobs to watermark all the images and push the processed image into the Amazon SQS queue.
2. The second fleet of Amazon EC2 servers pulls the watermarked images from the Amazon SQS queue.
3. The second fleet of EC2 workers processes the image and creates variations with different resolutions.
4. After encoding the images, the EC2 workers push the message into another Amazon SQS queue for thumbnail creation.
5. As the image is processed, the job deletes the message from the previous queue to make space.
6. The final fleet of EC2 servers gets encoded messages from the queue and creates thumbnails along with the copyright.

The benefits of this architecture are as follows:

- You can use loosely coupled asynchronous processing to return responses quickly without waiting for another service acknowledgment.
- You can structure the system by loosely coupling Amazon EC2 instances or containers using Amazon SQS.

- A message in the queue service remains intact even if there's a failure with the Amazon EC2 instance. This is crucial for maintaining data integrity and system robustness, as it ensures that processing can resume once the server is back online. This design creates a resilient system that can withstand and recover from server failures without losing critical data.

You may get fluctuations in application demand that can cause unexpected message loads. Automating your workload using the queuing chain pattern will help you handle any fluctuations. Let's learn more about using the job observer pattern to handle sudden workload fluctuations.

Job observer pattern

Queuing chain patterns help you design a loosely coupled architecture, but how will you handle workload spikes? In the case of request fluctuation, you need to adjust your processing power based on user demand, which the job observer pattern can address.

In the job observer pattern, you can create an auto-scaling group based on the number of messages in the queue to process. The job observer pattern helps you to maintain performance by increasing or decreasing the number of server instances used in job processing.

The following diagram depicts the job observer pattern:

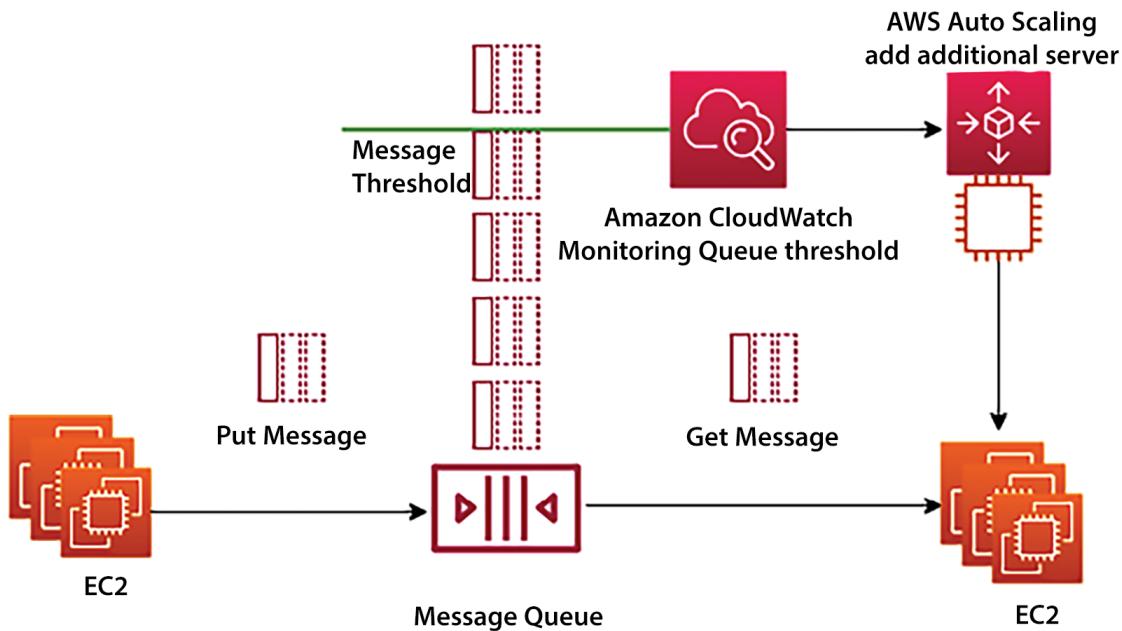


Figure 5.9: Job observer pattern architecture

In the preceding architecture, the first fleet of Amazon EC2 servers, which is the virtual server of AWS, is on the left-hand side, running batch jobs and putting messages in the queue, such as image metadata. The second fleet of EC2 servers on the right-hand side is consuming and processing those messages, for example, image encoding. As the message reaches a certain threshold, Amazon CloudWatch triggers auto-scaling to add the additional server in the consumer fleet to speed up the job processing. Auto-scaling also removes additional servers when the queue depth goes below the threshold.

The job observer pattern computes scale with job size, providing efficiency and cost savings. The job observer pattern architecture allows the job to be completed quickly. The process is resilient, which means job processing doesn't stop if a server fails.

While queue-based architecture provides loose coupling, it works mainly on the **asynchronous pull** method, where the consumer can pull messages from the queue when they are available.

In cloud-native architecture, it often helps if you build smaller independent steps between various architecture components, where one event should trigger other events. To implement this, let's learn more about the Pipes-and-Filters architecture in the next section.

Pipes-and-Filters Architecture

Pipes-and-Filters architecture is a software design pattern that divides complex tasks into a sequence of smaller, independent processing steps or stages. Each stage performs a specific operation on the input data and passes the transformed data to the next stage through a “pipe.” The stages are called “filters,” and the connectors are called “pipes.” Let's take a closer look at the primary components of this architecture:

- **Filters:** These processing units perform specific operations on the data. Filters read input data, process it, and produce output data. Each filter works independently and can be implemented and tested separately.
- **Pipes:** Pipes are the connectors that transport data between filters. They can be simple data streams or more complex mechanisms, such as message queues, that provide buffering, synchronization, and data format conversion.

The primary advantage of this architectural pattern is that it is a robust structure that promotes separation of concerns and modularity, making it easier to understand, modify, and maintain complex systems. It is favored for its reusability, composability, sequential processing, and scalability. Individual filters, which perform discrete processing tasks, can be reused across various applications, ensuring consistency and reducing development time. The composability of these filters allows for the construction of complex processing chains that can be easily modified by rearranging the filters as needed. Data flows through the pipeline in a clear, sequential manner, allowing each filter to transform the data step by step, which simplifies understanding and maintaining the system. Moreover, this pattern supports scalability as filters can run in parallel and be distributed across multiple computational nodes, enabling the system to handle increasing workloads effectively.

Let's understand this by taking an example. Imagine a text processing pipeline that reads a text file, removes stop words, performs stemming (reducing words to their root form), and counts the occurrences of each word. This can be implemented using the Pipes-and-Filters architecture:

1. **Filter 1—Read File:** Reads the text file and outputs lines of text
2. **Filter 2—Tokenize:** Splits lines into individual words
3. **Filter 3—Remove Stop Words:** Removes common words like “and,” “the,” “is,” etc.
4. **Filter 4—Stemming:** Reduces words to their root form (e.g., “walking” to “walk”)
5. **Filter 5—Count Words:** Counts the occurrences of each word

The filters are connected with pipes that transport data between them. The pipeline reads the text file, processes it step by step, and outputs the word frequencies.

The Pipes-and-Filters architecture is a powerful design pattern for building modular and easily extensible systems. Architects can create flexible, maintainable, scalable applications by dividing complex tasks into a sequence of smaller, independent filters connected by pipes.

Next, let's learn more about **Event-Driven Architecture (EDA)**, a design paradigm in which the flow of the program is determined by events such as user actions or messages from other programs. These events are processed asynchronously by independent components, allowing systems to be highly responsive and adaptable to changes or fluctuations in workload.

Creating Event-Driven Architecture

When EDA is implemented into a cloud-native architecture, it enhances the system's ability to react to real-time data and events. This combination can lead to highly efficient, scalable systems that can respond quickly to changes. The cloud-native environment supports the dynamic allocation of resources to handle the variable loads of event-driven systems, while the EDA provides the mechanism for immediate and reactive processing.

EDA helps you chain a series of events to complete a functional flow. For example, when you are making a payment to buy something on a website, you expect to get your order invoice generated and an email as soon as the payment is complete. Event-driven architecture helps to rope in all of these events so that making a payment can trigger another task to complete the order flow. Often, you will see message queues, which you learned about in the previous section, as the central point while talking about EDA. EDA can also be based on the publisher/subscriber or event stream models.

Publisher/subscriber model

In the publisher/subscriber (pub/sub) model, when an event is published, a notification is sent to all subscribers, and each subscriber can take the necessary action as per their requirements for data processing.

Let's look at an example of a photo studio application, which enriches a photo with different filters and sends a notification to the user. The following architecture depicts this pub/sub model:

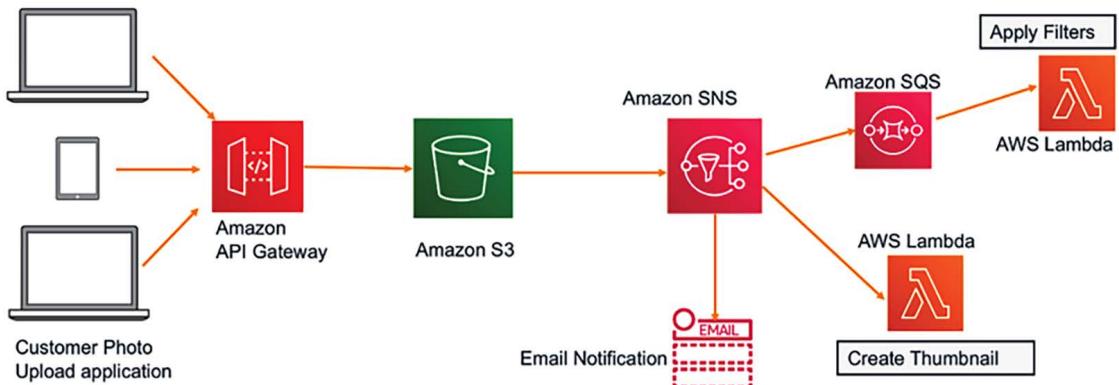


Figure 5.10: Photo studio application pub/sub event-driven architecture

In the preceding diagram, you will notice the following things:

1. The user first uploads the picture to an **Amazon S3** bucket using a web/mobile application.
2. The **Amazon S3** bucket then sends a notification to **Amazon Simple Notification Service (SNS)**. **Amazon SNS** is a message topic with the following subscribers:
 - Here, the first subscriber is using the email service, and as soon as the photo upload is complete, an email is sent to the user.
 - The second subscriber uses an **Amazon SQS** queue, which gets the message from the **Amazon SNS** topic and applies various filters in code written in **AWS Lambda** to improve the image quality.
 - The third subscriber uses the direct **AWS Lambda** function, which creates the image thumbnail.

In this architecture, **Amazon S3** publishes the message to the **SNS** topic as a producer, which multiple subscribers consume. Additionally, as soon as the message comes to **SQS**, it triggers an event for the **Lambda** function to process images.

Event stream model

In the event stream model, the consumer can read the continuous flow of events from the producer. For example, you can use the event stream to capture the continuous flow of a clickstream log and also send an alert if there are any anomalies detected, as shown in the following architecture diagram:

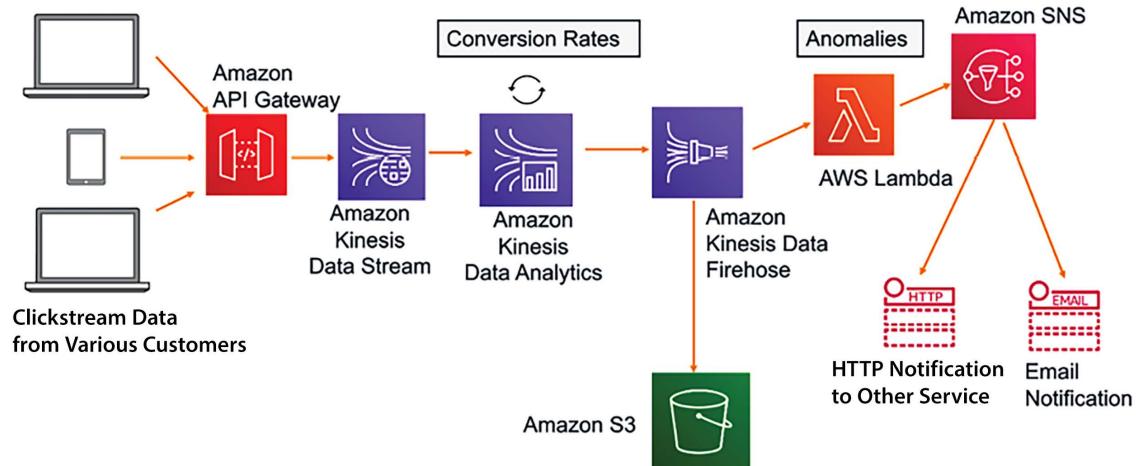


Figure 5.11: Clickstream analysis event stream architecture

Amazon Kinesis is a service used to ingest, process, and store continuous streaming data. In the preceding diagram, various customers clicking on e-commerce applications from web and mobile applications produce a stream of click events.

These clickstreams are sent to analytics applications using **Amazon API Gateway** for real-time analytics. In this analytics application, **Amazon Kinesis Data Analytics** calculates **conversion rates** over a certain period, for example, the number of people that made a purchase in the last five minutes. After aggregating data in real time, **Amazon Kinesis Data Analytics** sends the results to **Amazon Kinesis Data Firehose**, which stores all the data files in **Amazon S3** storage for further processing as needed.

A Lambda function reads from the event stream and starts examining the data for **anomalies**. As anomalies in the conversion rates are detected, the **AWS Lambda** function sends a notification via email for the campaign team to be notified. In this architecture, the event stream occurs continuously, and **AWS Lambda** reads from the stream for a specific event.

In EDA, producers and consumers operate independently, with events acting as the communication medium. This decoupling means that producers can send events without knowing which consumers will process them, and consumers can listen for events they are interested in without knowing who produced them. This leads to a flexible and extendable system where new consumers can be added to process events without modifying existing producers, facilitating scalability and adaptability. However, with the benefits of EDA, there are also challenges that need to be addressed:

- **Avoiding duplicate processing:** In distributed systems, the same event might be delivered more than once due to network retries or service outages. Implementing idempotency within event consumers ensures that processing an event multiple times does not lead to incorrect behavior or data inconsistencies.
- **Error message handling:** A robust EDA must have a mechanism to handle errors effectively. This can include dead-letter queues where events that cannot be processed are stored for later inspection or retried, and error handling logic within consumers to manage exceptions without disrupting the entire system.
- **Event ordering:** Ensuring that events are processed in the correct order can be critical. This might involve sequencing patterns or using event sourcing to maintain the order of events for each entity.
- **Event tracking and monitoring:** As the system scales, tracking the flow of events and monitoring the health of the system become essential. Implementing proper logging, tracing, and alerting mechanisms ensures visibility into the system's operation and quick diagnosis of issues.
- **Event schema management:** As systems evolve, event schemas may change. Managing these changes without disrupting the system requires a schema registry and versioning strategy, allowing consumers to understand different versions of an event.

While EDA fosters a highly scalable and extendable cloud-native architecture, it necessitates meticulous design and operational considerations to ensure that the system is resilient, consistent, and maintainable.

When discussing modular architecture, it's essential that modularity extends across all architectural layers to truly achieve scalability. Let's explore the BFF design pattern, which champions this approach.

Backend for Frontend pattern

The BFF pattern is a cloud-native architectural approach that tailors backend services for each specific type of frontend application. BFF is a design pattern that emerged as a response to the growing complexity of modern web and mobile applications. It involves creating separate backend services tailored to each frontend or user experience. By doing so, BFF aims to simplify frontend development and optimize backend responses according to each frontend's unique requirements.

Here's an overview of the key aspects of the BFF pattern:

- **Tailored APIs:** Each frontend (e.g., web, mobile, or smart TV) has its own backend service (BFF) tailored to its specific needs. BFFs provide APIs that deliver only the data the frontend requires in a suitable format. This approach reduces the need for data transformation on the frontend and results in optimized API responses.
- **Simplified frontend development:** Frontend developers can work closely with BFF, enabling better collaboration and faster development cycles. BFFs can be written in the same language as the frontend, making it easier for frontend developers to understand and modify the backend.
- **Delegated complexity:** BFFs can handle tasks that would otherwise burden the frontend, such as authentication, data aggregation, and error handling. This delegation of complexity reduces the frontend's workload and leads to a smoother user experience.

- **Independent evolution:** Each BFF can evolve independently, making it easier to roll out updates and features for specific frontends without affecting others. BFFs act as adapters between frontends and core backend services, minimizing the impact of changes in either layer.

Let's consider an e-commerce application with web, mobile, and intelligent TV frontends:

- **Web BFF:** Provides product details, user reviews, and recommendations for the web frontend. Aggregates data from multiple backend services, such as product information, user profiles, and recommendation engines. Transforms the data into a format suitable for web display.
- **Mobile BFF:** Offers a simplified product view, user reviews, and recommendations optimized for mobile devices. Handles tasks like image resizing to fit smaller screens. Aggregates data and adapts it for the mobile frontend's specific needs.
- **Smart TV BFF:** Delivers product information, user reviews, and recommendations tailored for smart TV display. Transforms data to fit the larger screen and simpler navigation options of smart TVs. Aggregates data and adapts it for the smart TV frontend's requirements.

By having separate BFFs for each frontend, the e-commerce application can provide optimized user experiences across different platforms while simplifying frontend development and reducing the complexity of backend services.

The BFF design pattern is powerful for building modern web and mobile applications, offering tailored APIs, simplified frontend development, delegated complexity, and independent evolution. Architects use BFFs to create more efficient, responsive, and user-friendly applications across multiple platforms.

So far in this chapter, you've learned about various cloud-native architecture design patterns. Now, let's learn about some anti-patterns to avoid.

Cloud-native architecture anti-patterns

In cloud-native architecture, as with any system design, certain practices are considered anti-patterns. An anti-pattern is a method that seems beneficial but typically falls short and can even be detrimental to your application. Here are some common anti-patterns to avoid in cloud-native architecture.

Single point of failure

A single point of failure occurs when a single component's failure can bring down the entire system. Design your cloud-native architecture with redundancy and failover mechanisms to handle such failures gracefully. A cloud application relying on a single database instance without any backup or replication is vulnerable to system-wide failure if that database instance goes down. Implementing a redundant database setup with replication and automatic failover prevents this scenario.

Manual scaling

Manual scaling involves manually adding or removing resources to accommodate changes in demand. It can be time-consuming, error-prone, and inefficient. Use a serverless service and auto-scaling features that automatically adjust the number of running instances based on demand.

If a streaming service experiences a sudden surge in viewers during a popular event, for example, manually scaling up the infrastructure might not be fast enough. Using serverless services or auto-scaling allows the service to quickly scale up resources to meet demand and scale down once the demand subsides.

Tightly coupled services

In a microservice architecture, services should be loosely coupled to operate independently. Tightly coupled services can lead to a fragile system that's hard to maintain and evolve. For example, if a payment service and a shipping service in an e-commerce platform are tightly coupled, changes to one service can unintentionally impact the other. Designing these services with clear boundaries and APIs allows them to evolve independently.

Ignoring security best practices

Security should be a top priority in any cloud-native architecture. Ignoring security best practices can result in data breaches, unauthorized access, and other security incidents. An application that stores user passwords in plain text is vulnerable to data breaches. Implementing proper password hashing, salting, and other security measures can prevent such incidents.

Not monitoring or logging

Monitoring and logging are necessary to diagnose issues, optimize performance, and understand system behavior. Implement monitoring tools to track application health and logs to diagnose problems. If a cloud application experiences performance issues, detailed monitoring and logging can help identify the cause, such as increased network latency, resource constraints, or application errors.

Ignoring network latency

Network latency can impact application performance in a distributed system. Design your system to handle network latency gracefully. For example, in a microservices-based e-commerce platform, network latency between services can slow down user interactions like browsing products or checking out. Implementing techniques like caching, data replication, and asynchronous communication can mitigate these effects.

Lack of testing

Proper testing ensures your application functions as expected and helps identify issues before they reach production. A cloud-native application that processes user data should have comprehensive unit, integration, and end-to-end tests to ensure data processing works correctly, prevent data loss or corruption, and verify proper integration between services.

Over-optimization

Over-optimizing your applications prematurely can make the code complex and hard to maintain. Implementing a highly optimized, custom data structure for a cloud application may improve performance slightly but can also make the code harder to understand, maintain, and adapt to future changes.

Not considering costs

Cloud services can be expensive if not managed properly. Monitor and optimize your cloud resource usage to avoid unexpected costs. Running large virtual machine instances 24/7 for an application with fluctuating demand, for example, is costly. Implementing auto-scaling and using serverless services can optimize costs by adjusting to demand.

You can create a robust, scalable, and maintainable cloud-native architecture by avoiding these anti-patterns. By adhering to best practices, avoiding anti-patterns such as these, and utilizing a microservices approach, you can ensure your cloud applications are scalable, robust, and secure. As you continue to build and evolve your applications, remain vigilant about potential challenges and strive to improve your designs, operations, and monitoring strategies continuously.

Summary

In this chapter, you explored cloud-native architectures comprehensively, uncovering the essential concepts, patterns, and practices vital for designing resilient, scalable, and efficient systems.

You started by unraveling the essence of cloud-native architecture and gaining an appreciation for its transformative potential in modern software development. You learned about its core benefits, including scalability, resiliency, and agility, which have made it indispensable in today's dynamic software landscape.

You delved deep into serverless architecture, discovering how it offers cost savings, seamless scalability, and operational simplicity. You learned about the contrasts and nuances between stateless and stateful designs, understanding their respective use cases, challenges, and implementation strategies. You ventured into the microservice architecture domain, grasping its inherent advantages of scalability, fault tolerance, and ease of deployment.

You encountered the Saga pattern, gained insights into its use for managing long-running transactions, and learned about the considerations for its effective implementation. You explored the fan-out/fan-in pattern, understanding its power in parallel data processing and subsequent aggregation. You learned about the service mesh pattern, appreciating its contributions to decentralized service management, enhanced observability, and traffic management.

You immersed yourself in reactive architecture, grasping its asynchronous and event-driven nature and recognizing its potential for improving responsiveness and scalability. You explored the territory of queue-based architectures, learning about their benefits in decoupling and asynchronous processing. You studied the queuing chain pattern, gaining insights into its applications and strategies for crafting robust sequential workflows.

You were introduced to the job observer pattern, understanding its utility in efficiently monitoring and managing jobs. You discovered the Pipes-and-Filters architecture, appreciating its flexibility and composability for processing data streams. You ventured into event-driven architectures, learning about their scalability, responsiveness, and decoupling benefits. You explored the publisher/subscriber model, understanding its potential for scalable and loosely coupled event distribution, and you delved into the event stream model, recognizing its advantages in handling continuous event streams.

You explored the BFF pattern, learning about its ability to tailor backends to specific user interfaces for greater flexibility and performance. Lastly, you uncovered common cloud-native architecture anti-patterns, learning how to avoid these pitfalls and following best practices.

With this newfound understanding of cloud-native architecture, you are better equipped to design robust, scalable, and efficient cloud-native systems that align with your unique requirements and objectives.

While you learned about various architecture patterns in this chapter, you will learn about architecture design principles for performance optimization in the next chapter. Additionally, you will deep dive into technology selection in computing, storage, databases, and networking, which can help improve your application's performance.

Join our book's Discord space

Join the book's Discord workspace to ask questions and interact with the authors and other solution architecture professionals: <https://packt.link/SAHandbook>



6

Performance Considerations

Experiments show that every second of application load delay causes a significant loss in an organization's revenue. Therefore, an application's performance is one of the most critical attributes of solution design that can impact the growth of your product's adoption.

In the previous chapter, we discussed various solution architecture design patterns that can be used to solve a complex business problem. In this chapter, we will explore some best practices to optimize the performance of your application that need to be done at every layer and with every architecture component. You will learn how to choose the right technology for the various layers of your architecture to continuously improve your application's performance. We will focus on the following topics in this chapter:

- Design principles for high-performance architecture
- Technology selection for performance optimization
- Performance considerations for mobile applications
- Performance testing
- Managing performance monitoring

By the end of the chapter, you will understand important attributes of performance improvement, such as latency, throughput, and concurrency. You will be able to make better decisions regarding your choice of technology, which will help you improve performance at the various layers of architecture, such as compute, storage, database, and networking.

Design principles for high-performance architecture

Architectural performance efficiency focuses on using application infrastructure and resources to meet increasing demand and technological evolution. Performance efficiency guides architects to create systems that not only meet current needs but are also agile enough to scale and evolve, ensuring that performance remains robust and responsive as user expectations and technological landscapes shift. Let's examine some vital design principles for your workload performance optimization.

Reducing latency

Latency can significantly affect your product adoption because users are looking for the fastest applications. It doesn't matter where your users are located; you need to provide an efficient and reliable service for your product to grow. Latency is a measure of the time taken for a packet of data to get from one designated point to another. In simpler terms, it's the delay or lag you experience between initiating an action and seeing the response on your device or system. This delay can be influenced by various factors, including the physical distance between the client and server, the speed of the transmission medium (such as fiber-optic cables or wireless signals), and how busy the network is.

For example, consider you're browsing a website. When you click a link or press a button, a request is sent from your device to the website's server. This server could be located in the same city as you, or it might be halfway around the world. The time it takes for your request to travel to the server, for the server to process that request, and then to send back a response, is what we call latency. You may not achieve zero latency, but the goal should be for the response time to be within the users' tolerance limit.

As shown in the following diagram, imagine a scenario where it takes 600 milliseconds (**ms**) for a message to travel from your device to the server (this could be because of the physical distance the data needs to travel or because the data packets are being routed through multiple intermediary devices, such as routers and switches). If the server takes another 900 ms to process the request and send back a response, the total latency experienced would be 1.5 seconds (1,500 ms). During this time, you might notice a delay in the web page loading.

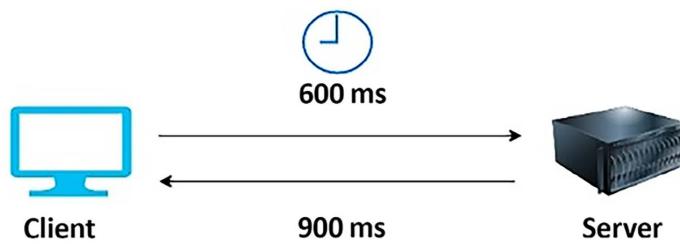


Figure 6.1: Request-response latency in a client-server model

Now, any application needs to access the internet to have a diverse set of global users around the world as consumers. These users expect consistency in performance, regardless of their geographical location. It is sometimes challenging because moving data over the network from one part of the world to another takes time.

Various factors, such as the **network transmission medium**, **router hops**, and **propagation**, can cause network latency. Enterprises commonly use a fiber-optic line to set up connectivity between their corporate network and the cloud, which helps to prevent inconsistency. Organizations can also utilize a **content distribution network (CDN)** to store heavy image and video data near user locations to reduce network latency and improve performance. With edge locations, it is easier to deploy workloads close to the user base.

In addition to the problems caused by the network, latency can occur in various architecture components. Your compute server can have latency issues at the infrastructure level due to memory and processor problems, where the data transfer between the CPU and RAM is *slow*. The disk can have latency due to slow read and write processes. Latency in a **hard disk drive (HDD)** depends on the time it takes to select a disk memory sector and for that sector to position itself under the head of the disk for reading and writing. The disk memory sector is the physical location of data in the memory disk. In an HDD, data is distributed in memory sectors during write operations. Since the disk is continuously rotating, data can be written randomly. During read operations, the head needs to wait for the rotation to bring it to the appropriate disk memory sector.

At the database level, latency can be caused by slow data reads and writes due to hardware bottlenecks or slow query processing. Reducing the database load by distributing the data with partitioning and sharding can reduce latency.

Low latency means *higher throughput*, as latency and throughput are directly related, so let's learn more about throughput.

Improving throughput

Network throughput refers to the amount of data that is successfully transferred over a network in a given period of time. This measure is crucial for understanding how well a network performs under specific conditions and loads. Throughput can be influenced by various factors, including the network's capacity (bandwidth), the quality of the connection, and the protocols used for data transfer. Bandwidth determines the maximum amount of data that can be transferred over the network.

Throughput and latency have a direct relationship. Low latency means high throughput because more data can be transferred in less time. To understand this better, let's use the analogy of a country's transportation infrastructure.

Let's say that highways with lanes are network pipelines and cars are data packets. Suppose a highway has 16 lanes between 2 cities. Not all vehicles can reach their destination at the desired time; they may get delayed because of traffic congestion, lanes closing, or accidents. Here, latency determines how fast a car can travel from one city to another, while throughput tells us how many cars can reach their destinations. Using full bandwidth is challenging for a network because of errors and traffic congestion.

Network throughput is measured by the amount of data sent over the network in bits per second (bps). Network bandwidth is the maximum size of the network's pipeline that can be processed. The following diagram illustrates the amount of data transferred between the client and the server:

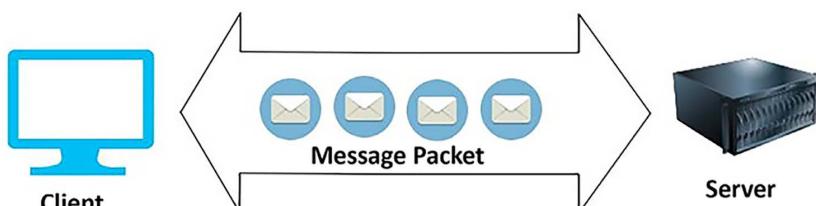


Figure 6.2: Throughput in a network

In addition to the network, throughput is applicable at the disk level. Disk throughput is a crucial metric that describes how fast data can be read from or written to a storage device. It's measured in terms of **megabytes per second (MB/s)** and is influenced by two main factors: the **Input/Output Operations Per Second (IOPS)** and the size of each I/O operation (average I/O size).

The formula to calculate disk throughput is:

$$\text{Throughput (MB/s)} = \frac{\text{Average I/O Size (Bytes)} \times \text{IOPS}}{1024^2}$$

Here's a breakdown of the formula:

- **Average I/O Size** is the average size of each read or write operation, measured in bytes. This can vary depending on the workload; for instance, database operations might have smaller I/O sizes compared to streaming large video files.
- **IOPS** (Input/Output Operations Per Second) measures how many read or write operations the storage can handle in one second. A high IOPS value indicates a fast storage system capable of handling a lot of operations in parallel.
- **Throughput in MB/s** provides a measure of the actual data transfer rate of the storage device. It combines the IOPS and the average I/O size to reflect how much data can be moved in and out of the storage system per second.

To convert the result into MB/s, we divide the product of average I/O size and IOPS by $1,024^2$ (since there are 1,024 bytes in a kilobyte and 1,024 kilobytes in a megabyte).

Given a disk IOPS of 20,000 and an I/O size of 4 KB (which is $4 * 1,024$ bytes = 4,096 bytes), you can calculate the throughput as follows:

1. First, convert the I/O size from bytes to megabytes:

- $\text{I/O size in MB} = \frac{4,096 \text{ bytes}}{1024 \times 1024} = 0.00390625 \text{ MB}$

2. Then, multiply by the IOPS to get the throughput in MB/s:

- Throughput = I/O size in MB × IOPS
- Throughput = $0.00390625 \text{ MB} \times 20,000$
- Throughput = 78.125 MB/s

So, with a disk IOPS of 20,000 and an I/O size of 4 KB, the throughput is approximately 78.125 MB/s. This calculation shows the total amount of data that can be read from or written to the disk per second under these conditions.

At the operating system level, throughput is determined by the amount of data transferred between the CPU and RAM per second. At the database level, throughput is determined by the number of transactions a database can process per second. At the application level, your code needs to handle transactions that can be processed every second by managing the application memory with the help of garbage collection handling and efficient use of the memory cache.

When you look at latency, throughput, and bandwidth, there is another factor called concurrency, which applies to the various architecture components and helps improve application performance. Let's learn more about concurrency.

Handling concurrency

Concurrency plays a vital role in designing scalable and efficient applications. It allows an application to perform multiple tasks simultaneously, making better use of system resources and improving overall application performance. By implementing concurrency, developers can ensure that their applications can handle multiple operations at the same time without waiting for one task to complete before starting another. This is particularly important in web applications that serve thousands of users and in data processing tasks that need to handle large volumes of data efficiently. Implementing concurrency can lead to significant improvements in response times and throughput, enhancing the user experience and the application's ability to scale.

Parallelism is another crucial concept in software design, complementing concurrency by executing multiple operations simultaneously across different processors or cores. While concurrency involves dealing with many tasks at once (for example, multitasking on a single-core CPU, where tasks are switched rapidly to give the appearance of simultaneous execution), parallelism takes this a step further by truly performing multiple operations at the same time via multi-core processors or distributed systems. This approach significantly speeds up processing times for compute-intensive tasks by dividing the work into small chunks that can be processed in parallel. Applications that process large datasets, complex calculations, or tasks that can be divided into independent units benefit greatly from parallelism, achieving higher throughput and efficiency.

As illustrated in the following diagram, concurrency is like a traffic light signal that controls where traffic flows between all four lanes. As there is a single thread along which you should pass all traffic, processing in other lanes must stop while traffic in one lane is in the *clearing process*. In the case of parallelism, there is a parallel lane available, and all cars can run in parallel without interrupting each other, as shown in the following diagram:

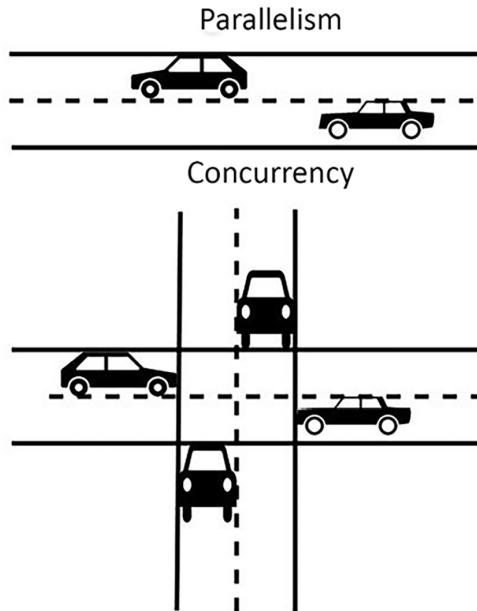


Figure 6.3: Concurrency versus parallelism

A database is always the central point of architectural design. Concurrency plays an essential role in data handling, as the database should have the ability to respond to multiple requests simultaneously. Database concurrency is more complicated as one user might be trying to read a record while another user is updating it simultaneously. The database should only allow data to be viewed when it is fully saved. Ensure the data is completely committed before another user tries to update it.

Caching can improve performance significantly; let's learn about some different cache types in architecture.

Applying caching

In *Chapter 4, Solution Architecture Design Patterns*, you learned how to apply caching at various levels of architecture in the *Cache-based architecture* section. Caching helps to improve application performance significantly. Although you have learned about the different design patterns to apply to the cache by adding an external caching engine and technology such as a **content delivery network (CDN)**, it's essential to understand that almost every application component and infrastructure has a cache mechanism. Utilizing the caching mechanism at each layer can help reduce latency and improve the application's performance.

The CPU has its hardware cache at the server level, which reduces the latency when accessing data from the main memory. The CPU cache includes the instruction and data cache; the data cache stores copies of frequently used data. The cache is also used at the disk level, but it is managed by operating system software (known as the **page cache**); however, the CPU cache is entirely managed by hardware. The disk cache originates from secondary storage, such as the HDD or **solid-state drive (SSD)**. Frequently used data is stored in an unused portion of the main memory (that is, the RAM as the page cache, which results in quicker access to content).

Often, databases have a cache mechanism that saves the results from the database in order to respond more quickly. They have an internal cache that gets data ready in the cache based on your usage patterns. They also have a query cache that saves data in the main server memory (RAM) if you make a query more than once. The query cache gets cleared in case any data changes inside the table. If the server runs out of memory, the oldest query result gets deleted to make space.

You have a DNS cache at the network level that stores the web domain name and corresponding IP address locally on the server. If you revisit the same website domain name, DNS caching allows a quick DNS lookup. The operating system manages the DNS cache and contains all recent website visits. You learned about client-side cache mechanisms such as the **browser cache** and caching engines such as **Memcached** and **Redis** in *Chapter 4, Solution Architecture Design Patterns*.

In this section on design principles for high-performance architecture, you learned about the original design factors, such as latency, throughput, concurrency, and caching, that need to be addressed in order to optimize architecture performance. Each component of the architecture (whether it is a network at the server level or an application at the database level) has a certain degree of latency and a concurrency issue that needs to be handled.

You should design your application for the desired performance because improving performance comes at a cost. The specifics of performance optimization may differ from application to application. The solution architecture needs to direct the effort accordingly—for example, a stock-trading application cannot tolerate even sub-millisecond latency. On the other hand, an e-commerce website can live with a couple of seconds' latency. To overcome performance challenges, let's learn about selecting technology for various architecture levels.

Technology selection for performance optimization

In *Chapter 4* and *Chapter 5*, you learned about various design patterns, including microservice, event-driven, cached-based, and stateless patterns. An organization may choose a combination of these design patterns depending on their solution's design needs. You can have multiple approaches to architecture design, depending on your workload. Once you have finalized your strategy and start to implement your solution, the next step is to optimize your application. To optimize your application, you must collect data by performing load testing and defining benchmarking as per your application's performance requirements.

Performance optimization is a continuous improvement process in which you need to be cognizant of optimal resource utilization from the beginning of solution design until after the application's launch. You need to choose the right resources as per the workload or tune the application and infrastructure configuration. For example, you might select a NoSQL database to store your application's session state and store transactions in a relational database.

For analytics and reporting purposes, you can offload your production database by loading data from the application database into data warehousing solutions and creating reports from there.

In the case of servers, you can choose a virtual machine or containers. You can take an entirely serverless approach to build and deploy your application code. Regardless of your approach and application workload, you need to choose the primary resource type: compute, storage, database, or network. Let's look in more detail at selecting these resource types for performance optimization.

Making a computational choice

In this section, you will see the use of the term *compute* instead of *server*, as software deployments are not limited to servers nowadays. A public cloud provider such as AWS has serverless offerings, where you don't need a server to run your application. One of the most popular FaaS offerings is AWS Lambda. Like AWS Lambda, other popular public cloud providers provide solutions in the FaaS space—for example, Microsoft Azure has Azure Functions, and GCP has Google Cloud Functions.

However, organizations still make the default choice to go for servers with virtual machines. Containers are also becoming popular as the need for automation and resource utilization has increased. Containers are becoming the preferred choice, especially in the area of microservice application deployment.

The optimal choice of computing—whether you want to choose server instances, containers, or go serverless—depends upon your application's use case. Let's look at the various computing choices.

The following table provides a snapshot of the differences between CPUs, **Graphics Processing Units (GPUs)**, **Field Programmable Gate Arrays (FPGAs)**, and **Application-Specific Integrated Circuits (ASICs)**, focusing on their primary uses, ease of programming, core structures, cost implications, and their suitability for parallel processing, among other features. Let's first define each of these terms:

- **CPU:** The primary processor within a computer that performs most of the data processing operations, often referred to as the “brain” of the computer
- **GPU:** A specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display
- **FPGA:** An integrated circuit designed to be configured by a customer or a designer after manufacturing – hence “field-programmable”
- **ASIC:** A chip customized for a particular use, rather than intended for general-purpose use

You might use one or more of these processing unit choices as your workload varies.

Feature	CPU	GPU	FPGA	ASIC
Primary Use	General-purpose applications	Graphics processing, large data use cases	Programmable hardware for specific tasks	Custom integrated circuits for specific applications
Programming Ease	Easy	Requires knowledge of specific libraries (e.g., CUDA)	Complex, requires hardware programming	N/A (hardware is custom-designed)
Multitasking	Yes	Limited by design focus on parallelism	Yes, with reconfiguration	No, single-purpose
Versatility	High	Moderate	Moderate, reconfigurable for tasks	Low, application-specific
Performance Measurement	GHz (billions of cycles per second)	TFLOP (trillions of floating-point operations per second)	Not typically measured in Flops (floating point operations per second)	Optimized for power consumption and performance
Core Structure	Few large cores	Thousands of small cores	Logic elements that can be reconfigured	N/A (custom-designed)
Parallel Processing	Limited	High, with Massively Parallel Processing (MPP)	Accommodates MPP, configurable as CPU	Optimized for specific applications
Cost	Low	Higher than CPUs	Higher than CPUs and GPUs, requires customization	Highest, due to custom design and long development cycle
Power Consumption	Moderate	High	Low	Optimized for the application
Flexibility	Versatile for a range of applications	Specific to compute-intensive applications	Reconfigurable but requires development	Fixed, requires redesign for changes
Development Cycle	Short	Short to moderate	Long, due to the need for customization	Longest, hardware-level redesign needed

Table 6.1: Comparison of the various processor types

The preceding table compares the types of processing. The ASIC is the most efficient but has a long development cycle to implement. ASICs provide the most optimal performance but have the least flexibility to reutilize, while CPUs are very flexible and can fit many use cases.

Today, the CPU has become a commodity and is used everywhere for general-purpose devices to keep costs low. The GPU has become famous for compute-intensive applications, and the FPGA has become the first choice where more customized performance is required. These processing choices are available from public cloud providers, such as AWS.

In this section, you learned about the most popular computing choices.

You may also hear about other types of processors, such as the accelerated processing unit (APU). The APU combines the CPU, GPU, and **digital signal processor (DSP)**, which is optimized to analyze analog signals and requires high-speed real-time data processing.

Let's learn more about other popular compute-type containers that are rapidly gaining popularity because of their ability to optimize the use of resources within the virtual machine.

Working with containers

In *Chapter 4, Solution Architecture Design Patterns*, you learned about the benefits of container deployment in the section titled *Deploying an application with a container*. The use of containers is becoming the norm for deploying complex microservice applications because of the ease of automation and resource utilization efficiency. There are various platforms available for container deployment.

Because of their popularity and platform-independent capabilities, containers have become the first choice for building cloud-agnostic platforms. You can deploy containers in your on-premises data center and manage them through your cloud. Also, you can take a relocation approach and move a container from on-premises to the cloud without having to make any changes.

You can build a multi-cloud platform with a container, and now, each major public cloud vendor provides tools for managing a container environment spread over multiple platforms. For example, AWS provides **Elastic Container Service (ECS) Anywhere**, enabling you to run and manage container workloads on customer-managed infrastructure easily. Similarly, GCP provides **Google Anthos**, which gives you container management across on-premises and other cloud platforms. Let's learn about some of the most popular choices in the container area, their differences, and how they work together.

Docker

Docker is one of the most in-demand technologies. It allows you to package an application and its related dependencies together as a container and deploy it to any operating system platform. Docker provides platform-independent capabilities to a software application, making the overall software development, testing, and deployment process simple and accessible.

Docker containers help you to build a more complex multilayer application. For example, you need to run the application server, database, and message queue together. In that case, you can run them side by side using different Docker images and establish communication between them. Each of these layers may have modified library versions, and Docker allows them to run on the same computing machine without conflicts.

Docker container images are portable from one system to another over a local network or across the internet using Docker Hub. You can manage and distribute your container using a Docker Hub container repository. Suppose you make changes in the Docker image that cause environmental issues. In that case, it's easy to revert to the working version of the container image, making overall troubleshooting easier.

When using Docker, the development team builds an application and packages it with the required dependencies into a container image. This application image is run in a container on the Docker host. Just as you manage code in a code repository such as GitHub, in the same way, a Docker image should be stored in a registry. Docker Hub is a public registry, and other public cloud vendors provide their own registries, such as **AWS Elastic Container Registry (ECR)** and **Azure Container Registry**. In addition, you can have a private registry on-premises for your Docker images.

Public cloud providers like AWS provide container management platforms, such as **Amazon ECS**. Container management manages Docker containers on top of the cloud virtual machine, Amazon EC2. AWS also provides a serverless option of container deployment using Amazon Fargate, where you can deploy containers without provisioning virtual machines.

Complex enterprise applications are built based on microservices that may span multiple containers. Managing various Docker containers as a part of one application can be complicated. Kubernetes helps solve the multi-container environment's challenges; let's learn more about Kubernetes.

Kubernetes

Kubernetes excels in managing and orchestrating multiple containers in production settings, functioning as a comprehensive container orchestration system. It supports hosting Docker containers on either physical servers or virtual machine nodes, which are commonly referred to as worker nodes. Kubernetes efficiently coordinates operations across a cluster of these nodes, automating tasks such as deployment, scaling, and the management of containerized applications, thus ensuring smooth and reliable application performance across the infrastructure.

Kubernetes makes your application self-healing by replacing unresponsive containers in the case of any application error. It also provides horizontal scaling capabilities and a blue-green deployment ability to prevent downtime. Kubernetes distributes incoming user traffic load between the containers and manages the storage shared by various containers.

The following diagram shows that Kubernetes and Docker work well together to orchestrate your software application. Kubernetes handles network communication between worker nodes and Docker containers:

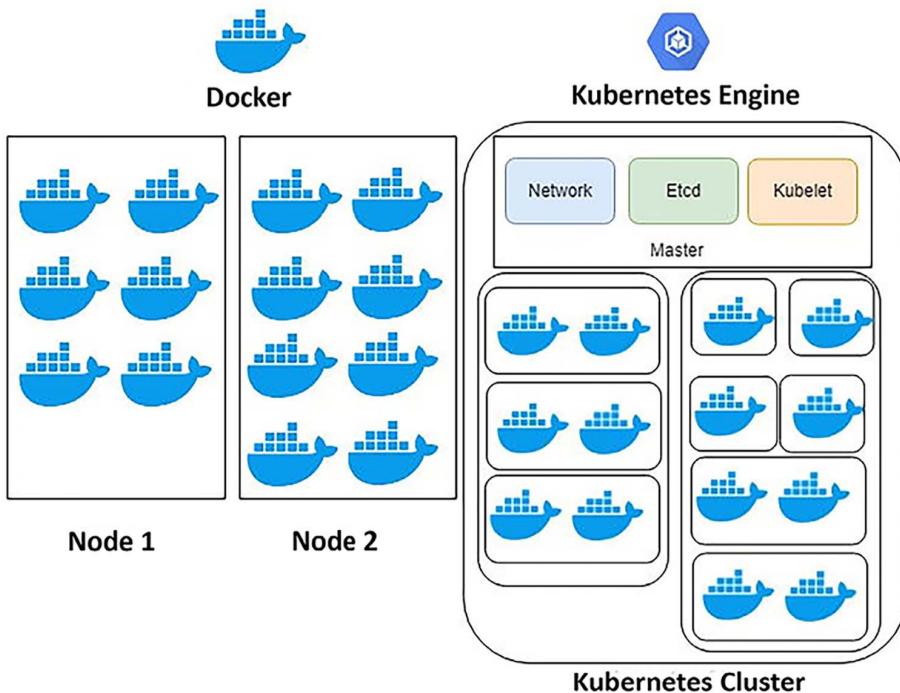


Figure 6.4: Docker and Kubernetes

Docker works as an individual piece of the application, and Kubernetes takes care of the orchestration to make sure all these pieces work together as designed. It's easy to automate overall application deployment and scaling with Kubernetes. In Docker, containers are hosted in nodes, and each Docker container in a single node shares the same IP space. In Docker, you must manage the connections between containers by handling any IP conflict. Kubernetes solves this problem by having a primary instance that keeps track of all nodes hosting Pods.

Kubernetes's primary node assigns an IP address and hosts a key-value store for container configuration and a **kubelet** to manage the containers. The kubelet is the primary "node agent" that runs on each node and ensures that the containers defined in the Pods are started and continue running. Docker containers are grouped into **Pods** that share the same IP address. This entire setup is called a **Kubernetes cluster**.

Kubernetes is complex to maintain. Cloud providers supply their own management tools for it. For example, AWS provides Amazon **Elastic Kubernetes Service (EKS)** to simplify the management of the Kubernetes cluster. OpenShift is another Kubernetes distribution managed by Red Hat and is offered as a **Platform as a Service (PaaS)**. Similarly, Microsoft Azure provides **Azure Kubernetes Service (AKS)**, and GCP provides **Google Kubernetes Engine (GKE)**, offering a simple way to deploy, scale, and manage Kubernetes automatically.

Overall, containers add a layer of virtualization to the whole application infrastructure. While they are helpful in resource utilization, choose a bare-metal physical machine for your application deployment if it requires ultra-low latency.

Going serverless

In recent years, serverless computing has become possible because of the popularity of public cloud solutions from cloud providers such as Amazon, Google, and Microsoft. Serverless computing allows developers to focus on their code and application development without worrying about underlying provisioning, configuring, and scaling infrastructure. Serverless solutions abstract server management and infrastructure decisions from developers and let them focus on their area of expertise and the business problem they are trying to solve. Serverless computing brings the relatively new **Function as a Service (FaaS)** concept.

FaaS offerings are available from AWS Lambda, Microsoft Azure Functions, and Google Cloud Functions. For example, you can write your code in the cloud editor, and AWS Lambda handles the computing infrastructure underneath to run and scale your function. You can design event-based architecture or RESTful microservices by adding an API endpoint using Amazon API Gateway and AWS Lambda functions. Amazon API Gateway is a maintained cloud system that adds RESTful APIs and WebSocket APIs as frontends for the Lambda functions and enables real-time communication between applications. You can further break down your microservice into small tasks that can be scaled automatically and independently.

In addition to focusing on your code, you never have to pay for idle resources with the FaaS model. Rather than scaling your entire service, you can scale the required functions independently with built-in availability and fault tolerance. However, this could be daunting if you have thousands of features to orchestrate, and predicting the auto-scaling cost can be tricky. It is perfect for scheduling jobs, processing web requests, and queuing messages.

In this section, you have learned about the various computing choices you can make for performance optimization. We looked at server instances, containers, and serverless options. You need to select the right computing services for your application's requirements. No rule forces you to choose a particular type of computing; it is all about your organization's choice of technology, the pace of innovation, and the nature of the software application.

However, you can generally stick to a virtual or bare-metal machine for a monolithic application, and for complex microservices, you can choose containers. For simple task scheduling or events-based applications, serverless functions are an obvious choice. Many organizations have built complex applications entirely on serverless computing, which helped them reduce costs and achieve high availability without managing any infrastructure.

Let's learn about another important aspect of your infrastructure and how it can help you to optimize performance.

Making a storage choice

Storage plays a pivotal role in the performance of any software application, and the concept of **data affinity** is crucial when discussing storage for an application. Data affinity refers to the strategic placement of data in proximity to the application consuming it, with the aim of reducing latency, improving performance, and ensuring efficient data retrieval.

In a multi-cloud or hybrid cloud environment, the notion that all storage must be in immediate proximity to the application server doesn't always hold true. Modern distributed systems are designed to allow data to reside in multiple locations, whether on-premises or on different cloud providers, while still maintaining acceptable levels of latency and performance. This flexibility is crucial for organizations that use various cloud services or for those with data residency requirements mandating certain data to remain within specific geographical or jurisdictional boundaries.

However, deciding where to store data—whether close to the application service or in a different location—requires careful consideration of several factors:

- **Latency requirements:** The acceptable delay between a request and a response can significantly influence where data should be stored. Applications requiring real-time access to data might necessitate storage solutions with minimal latency, often implying close physical or network proximity.
- **Data sovereignty and compliance:** Legal and regulatory requirements might dictate where data can be stored and processed, which means the architecture will need to align with compliance mandates.
- **Cost considerations:** Storing and accessing data across different locations or clouds can incur additional costs. It's essential to balance performance needs with budget constraints, especially when considering data egress fees in cloud environments.
- **Bandwidth and throughput:** The available network bandwidth and throughput between the application server and the data storage location can affect performance. High bandwidth and throughput can mitigate some latency issues, allowing for more flexible data storage options.
- **Data access patterns:** Understanding how your application accesses data (e.g., frequently accessed vs. rarely accessed data) can help you choose the right storage location. Frequently accessed data might benefit from being closer to the application to speed up access times.
- **Disaster recovery and availability:** Data resilience strategies might require data to be replicated across different geographical locations to ensure availability in case of failures.

In multi-cloud strategies, implementing caching, data replication, or edge computing solutions can help maintain performance standards by keeping a synchronized copy of critical data close to the application, regardless of the primary data storage location. These approaches allow applications to access data with minimal latency, even if the primary data source is geographically distant.

Selecting the appropriate storage depends on a thorough analysis of these factors. You should aim to strike a balance between operational requirements, performance, cost, and compliance. The ultimate goal is to architect a solution that meets the application's performance needs while adhering to organizational, legal, and budgetary constraints.

You first need to decide whether your data will be stored in block, file, or object storage. These are storage formats that store and present data differently. Let's look at this in more detail.

Working with block storage and storage area network

Block storage divides data into blocks and stores them as chunks of data. Each block has a unique ID that allows the system to place data wherever it is most easily accessible, as blocks don't store any metadata about files. Hence, a server-based operating system manages and uses these blocks in the hard drive. Whenever the system requests data, the storage system collects the blocks and gives the result back to the user.

Block storage deployed in a **storage area network (SAN)** stores data efficiently and reliably. It works well when a large amount of data needs to be stored and accessed frequently—for example, database deployment, email servers, application deployment, and virtual machines.

SAN storage is sophisticated and supports *complex, mission-critical applications*. It is a high-performance storage system that communicates block-level data between the server and storage; however, SANs are expensive and should be used for large-scale enterprise applications where low latency is required.

To configure your block-based storage, you must choose between an SSD and an HDD. HDDs are the legacy data storage system for servers and enterprise storage arrays. HDDs are cheap but slow and need a lot of power and cooling. SSDs use semiconductor chips and are faster than HDDs. They are much more costly; however, SSDs have become more affordable and popular as technology evolves because of their efficiency and lower power and cooling requirements.

Working with file storage and network area storage

File storage has been around for a long time and is widely used. In file storage, data is stored as a single piece of information and is organized inside folders. When you need to access the data, you provide the file path and get the data files; however, a file path can grow complicated as files become nested under multiple folder hierarchies.

Each record has limited metadata, including the filename, time of creation, and up-to-date timestamps. Consider the analogy of a library, where you store books on shelves and keep a log with the location of each book.

Network area storage (NAS) is a file storage system that is attached to the network and displays to the user where they can store and access their files. NAS also manages user privilege, file locking, and other security mechanisms that protect the data. NAS works well for file-sharing systems and local archives; however, there are better solutions for storing billions of files than NAS, given that it has limited metadata information and a complex folder hierarchy. To store billions of files, you need to use object storage. Let's learn more about object storage and its benefits compared to file storage.

Working with object storage and cloud data storage

Object storage bundles data with a unique identifier and metadata that is customizable. Object storage uses a flat address space, in contrast to the hierarchical addresses in file storage or addresses distributed over a chunk of blocks in block storage. A flat address space makes it easier to locate and retrieve data quickly, regardless of the data storage location. Object storage also helps the user to achieve unlimited scalability of storage.

Object storage metadata can have many details, such as object name, size, timestamp, and so on, and users can customize it to add more information than you get with tagging in file storage. A simple API call can access data, and it is very cost-effective to store. Object storage performs best for high-volume, unstructured data; however, objects cannot be modified but only replaced, making it not a good use case for a database.

Cloud data storage, such as **Amazon Simple Storage Service (S3)**, provides unlimited scalable object data storage with high availability and durability. You can access data with a unique global identifier and metadata file prefix.

The following diagram shows all three storage systems in a nutshell:

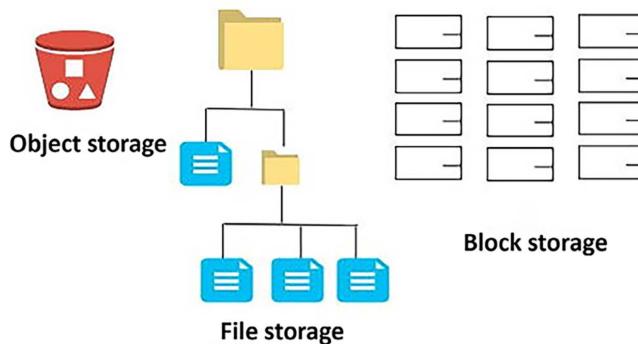


Figure 6.5: Data storage systems

As shown in the preceding diagram, block storage stores data in blocks. Block storage is ideal for scenarios where a single instance requires exclusive access to the storage, such as databases or applications that demand high performance and fast access to data. File storage stores data in a hierarchical folder structure with little overhead latency. You should use a file storage system when your data needs to be accessed by multiple instances simultaneously, much like how various people might need to access files in a shared room. Object storage stores data in buckets with a unique identifier for each object. It provides access over the web to reduce latency and increase throughput. You should use object storage to store and access static content, such as images and videos. You can store a high volume of data in object stores and perform big data processing and analysis.

Direct-attached storage (DAS) is another kind of data storage that is directly attached to the host server; however, it has very limited scalability and storage capacity.

The magnetic tape drive is another popular back up and archiving storage system. Because of its low cost and high availability, magnetic tape drives are used for archival purposes but have high latency, making them unsuitable for use in direct applications.

Often, you will need to increase throughput and data protection for a mission-critical application, such as a transactional database, where data is stored in SAN storage.

Select the storage solution that matches your access pattern to maximize performance. Various options are available with a cloud offering for your block, file, and object storage method. For example, the AWS public cloud provides **Amazon Elastic Block Store (EBS)** as SAN storage in the cloud and **Amazon Elastic File System (EFS)** as NAS storage in the cloud.

Amazon S3 is very popular for object storage. Similarly, Microsoft Azure provides Azure Disk Storage for SAN, Azure Files for NAS, and Azure Blob Storage for block storage.

Storage for databases

Choosing the right type of storage for database performance is crucial to ensure optimal operation and efficiency. The selection often depends on the specific requirements of the database workload, such as IOPS, database size, the geographical location of data access, and the nature of the database operations (**Online Transaction Processing (OLTP)** versus **Online Analytical Processing (OLAP)**). Below is a comparative table that outlines the selection criteria and the suitability of various storage types for databases.

Storage Type	IOPS Capability	Suitable DB Size	Location Consideration	Best Use Case	Suitability
SSD	High	Small to large	Near to application server	OLTP and OLAP	Highly suitable for most databases, especially where high IOPS and low latency are critical.
HDD	Moderate to low	Large	Near to application server	Large OLAP	Suitable for large, infrequently accessed databases or where cost is a concern, but not recommended for high-performance OLTP systems.
NAS	Low to moderate	Small to medium	Flexible, can be off-premises	OLAP and backup	Suitable for databases with moderate performance requirements or for backup/archival purposes.
SAN	High	Large	Flexible, preferably nearby	OLTP and OLAP	Very suitable for large databases requiring high IOPS, throughput, and scalability. Can be on-premises or cloud-based.
Cloud storage	Variable	Variable	On-premises or cloud	OLTP and OLAP	Suitable for a wide range of database sizes and types. Performance and suitability depend on specific cloud service offerings.

Table 6.2: Comparison of various storage types

In scenarios involving multi-cloud or hybrid setups, additional factors, such as data sovereignty and compliance (which dictate where data can be stored based on regulations), access patterns (whether the data is read or write-heavy), network latency, bandwidth, and cost considerations, also play a critical role. These aspects are especially crucial when databases are accessed over **Wide Area Networks (WANs)**, where delays can impede performance.

Now that you have learned about the compute and storage choices required to achieve optimal performance, let's look at the next critical component of application development: the database. Choosing the right database for the right need will improve your application's performance and reduce overall application latency. Different types of databases are available, and choosing the correct one is critical.

Making a database choice

Often, you will want to standardize a common platform and use a database for ease of management; however, consider using a different database solution as per your data requirements. Selecting the incorrect database solution can impact system latency and performance.

The database you choose will depend on your application's availability, scalability, data structure, throughput, and durability requirements. There are multiple factors to consider when choosing which database to use. For example, the access pattern can significantly impact database technology selection, depending on the number of users and data access frequency. It would be best if you optimized your database based on the access pattern.

Databases generally have a configuration option for workload optimization. You should consider the configuration for memory, cache, storage optimization, and so on. You should also explore the operational aspects of database technologies regarding scalability, backup, recovery, and maintenance. Let's look at the different database technologies that can be used to fulfill the database requirements of applications.

Online transactional processing

Most of the traditional relational databases are considered to use **online transactional processing (OLTP)**. The transactional database is the oldest and most popular method of storing and handling application data. Some examples of relational OLTP databases are Oracle, Microsoft SQL Server, MySQL, PostgreSQL, and Amazon RDS. The data access pattern for OLTP involves fetching a small dataset by looking up its ID. A database transaction means that either all related database updates were completed or none of them were.

The relational model allows the processing of complex business transactions in an application, such as banking, trading, and e-commerce. It enables you to aggregate data and create complex queries using multiple joins across tables.

While optimizing your relational database, you need to consider including the following optimizations:

- A database server that includes computing, memory, storage, and networking
- Operating system-level settings, such as the storage volume, volume management, and block size
- Database engine configuration and partition as required
- Database-related options, such as schema, index, and view

Scaling can be tricky for relational databases as they can scale vertically and hit the upper limit of system capacity. Utilize read replicas to distribute the read load. This allows you to offload the read queries from the primary database to one or more replicas, enhancing the system's read capacity. Implement partitioning (sharding) for scaling writes. By dividing a larger database into smaller, more manageable pieces (partitions or shards), each with a subset of the data, you can distribute the write load across multiple servers or instances, improving write performance.

In the previous chapter, you learned how to scale a relational database in the *Database handling in the application architecture* section.

OLTP databases are suitable for large and complex transactional applications; however, they need to scale better when a massive amount of data must be aggregated and queried. Also, with the internet boom, a lot of unstructured data is coming from everywhere, and relational databases cannot handle unstructured data efficiently out of the box. In this case, a nonrelational, or NoSQL database, comes to the rescue. Let's learn more about how to handle them.

Nonrelational databases

A lot of unstructured and semi-structured data is produced by applications such as social media programs, the Internet of Things (IoT), clickstream data, and logs, where you have very dynamic schemas. These data types may have different schemas for each set of records. Storing this data in a relational database could be a very tedious task. Everything has to be filed in a fixed schema, which can either cause lots of null values or data loss. Nonrelational databases, commonly referred to as NoSQL ("Not Only SQL" or "Non-SQL"), provide a flexible approach to data storage and management. Unlike traditional relational databases, which require a fixed schema before data can be stored, NoSQL databases allow you to store and manage data without predefined schema constraints. Records with a variable number of columns can be stored in the same table.

NoSQL databases can store a large amount of data and provide *low-access latency*. They are easy to scale by adding more nodes when required and can support horizontal scaling out of the box. They are an excellent choice for storing user session data and making your application stateless in order to achieve horizontal scaling without compromising user experience. You can develop a distributed application on top of the NoSQL database, which provides good latency and scaling, but query joining must be handled at the application layer because NoSQL databases don't support complex queries such as joining tables and entities.

There are various NoSQL database options available, such as Cassandra, HBase, and MongoDB, which you can install in a cluster of virtual machines. AWS provides a managed NoSQL database called **Amazon DynamoDB**, which provides high-throughput sub-millisecond latency with unlimited scaling.

You can use OLTP for a relational database, but it has limited storage capacity. It needs to respond better to queries for large amounts of data and perform aggregations as required for data warehouses. Data warehousing needs are more analytical than transactional. OLAP fills the gaps in OLTP's capabilities to query large datasets. Let's learn more about OLAP.

Online analytical processing

OLTP and NoSQL databases are helpful for application deployment but have limited capabilities for large-scale analysis. OLAP is primarily used in data warehouse technology. A query for a large volume of structured data for analytics purposes is better served by a data warehouse platform designed for faster access to structured data. Modern data warehouses leverage columnar storage formats and **Massively Parallel Processing (MPP)** architecture to significantly enhance data retrieval and analysis speeds. Unlike traditional row-oriented databases, where data is stored in rows, columnar storage organizes data in columns.

The columnar format means you don't need to scan the entire table when you need to aggregate only one column for data—for example, if you want to determine your inventory sales in a given month. There may be hundreds of columns in the order table, but you only need to aggregate data from the purchase column. With a columnar format, you will only scan the purchase column, which reduces the amount of data scanned compared to the row format and increases the query performance.

With MPP, you store data in a distributed manner between child nodes and submit a query to the leader nodes. Based on your partition key, the leader node will distribute queries to the child nodes. Each node then picks up part of a query to perform parallel processing. The leader node then collects the subquery result from each child node and returns your aggregated result. This parallel processing helps you to execute the query faster and process a large amount of data quickly.

You can use this kind of processing by installing software such as IBM Netezza or Microsoft SQL Server on a virtual machine, or you can go for a more cloud-native solution, such as Snowflake. AWS, as a public cloud, provides the petabyte-scale data warehousing solution Amazon Redshift, which uses the columnar format and MPP. You will learn more about data processing and analytics in *Chapter 12, Data Engineering for Solution Architecture*.

You need to store and search a large amount of data, especially when you want to find a specific error in your logs or build a document search engine. For this kind of capability, your application needs to create a data search functionality. Let's learn about data search functionality.

Building a data search functionality

Often, you need to search a large volume of data to solve issues quickly or get business insights. Searching your applications' data will help you access and analyze detailed information from different views. To search for data with low latency and high throughput, you need search engines.

Elasticsearch is one of the most popular search engine platforms; it is built on top of the **Apache Lucene** library. Apache Lucene is a free and open-source software library that is the foundation of many popular search engines. The **ELK** (short for **Elasticsearch, Logstash, and Kibana**) Stack is easy to use to discover large-scale data and index it for searching automatically. Because of its properties, multiple tools have been developed around Elasticsearch for visualization and analysis. For example, **Logstash** works with Elasticsearch to collect, transform, and analyze a large amount of an application's log data. **Kibana** has an in-built connector with Elasticsearch that provides a simple solution for creating dashboards and analyzing indexed data. Elasticsearch can be deployed in virtual machines and scaled horizontally to increase capacity by adding new nodes to the cluster. The AWS public cloud provides the managed **Amazon OpenSearch Service**, making it cost-effective and simple to scale and manage the Elasticsearch cluster in the cloud.

In this section, you learned about the various database technologies and where they are used. Your applications can use a combination of database technologies for their different components to achieve optimal performance. For complex transactions, you need to use a relational OLTP database, and to store and process unstructured or semi-structured data, you need to use a nonrelational NoSQL database. You should use a NoSQL database when very low latency is required over multiple geographical regions and when you need to handle complex queries at the application layer, such as in a gaming application. If you need to perform large-scale analytics on structured data, use a data warehouse OLAP database.

Let's look at another critical component of your architecture: **Networking**. Networking is the backbone of the entire application and establishes communication between the servers and the outside world. Let's learn about networking as it relates to application performance.

Improving network performance

In this era of fast internet availability in almost every corner of the world, applications are expected to have a global user reach. Any delay in the system's response time depends upon the request load and the distance of the end-user from the server. If the system is not able to respond to user requests promptly, it can have a ripple effect by continuing to engage all the system's resources and pile up a considerable request backlog, which will degrade the overall system performance.

To reduce latency, you should simulate the user's location and environment to identify any gaps. Using your findings, you should design the server's physical location and caching mechanism to reduce network latency; however, the network solution choice for an application depends upon the networking speed, throughput, and latency requirements. An application to handle a global user base needs fast connectivity with its customers, and location plays an important role. Edge locations provided by the CDN help to localize the rich content and reduce overall latency.

In *Chapter 4, Solution Architecture Design Patterns*, you learned how to use a CDN to put data near your user's location in the section titled *Cache-based architecture*. There are various CDN solutions available with an extensive network of edge locations. You can use a CDN if your application is static-content-heavy, for example, if you need to deliver a large amount of image and video content to your end-user. Some of the more popular CDN solutions are Akamai, Cloudflare, and Amazon CloudFront (provided by the AWS cloud).

Using edge computing

Edge computing is a distributed computing paradigm that brings computation and data storage closer to the location where it is needed to improve response times and save bandwidth. These are the mini data centers providing IT infrastructure near to the location at which they are used. Edge computing has emerged as a game-changing strategy for optimizing software application performance, mainly when latency, bandwidth, and real-time data processing are critical. You can leverage edge computing to boost your application's performance, which has a globally distributed user base.

Consider a scenario where a renowned global social media website such as Facebook, X, or TikTok experiences a surge in user activity due to a live event, such as a major sports game or a celebrity announcement. In the traditional model, centralized servers may need help to handle the massive influx of requests, resulting in slower load times and potential disruptions. This is where **Content Delivery Networks (CDNs)** come into play, with industry giants such as Akamai, Cloudflare, Imperva, and Amazon CloudFront leading the way.

Akamai, one of the pioneers in the CDN space, boasts a vast network of edge servers strategically located in numerous countries and cities worldwide. Akamai's edge servers in Tokyo spring into action when a user from, say, Tokyo, Japan, accesses their global social media site during a high-traffic event. These servers cache and deliver frequently accessed content, such as images, videos, and static files, from a location much closer to the user than the centralized data centers. As a result, the user experiences lightning-fast load times, reduced latency, and smooth content delivery.

Moreover, Akamai's edge servers also provide advanced security features, such as **Distributed Denial of Service (DDoS)** protection and **Web Application Firewall (WAF)** capabilities, ensuring that the social media website remains resilient against cyberattacks and unauthorized access. Amazon CloudFront, tightly integrated with **Amazon Web Services (AWS)**, also offers a robust edge computing solution for businesses of all sizes.

Beyond social media, edge computing is transforming various industries. In autonomous vehicles, for example, edge devices process data from sensors in real time to make split-second decisions, ensuring safety on the road. In the IoT realm, edge computing enables smart devices to analyze data locally, reducing latency and conserving bandwidth. For instance, a smart thermostat can adjust temperature settings based on local sensor data without the need for constant communication with centralized servers.

In healthcare, edge computing is utilized for remote patient monitoring. Wearable devices equipped with edge processing capabilities can analyze health data in real time and send alerts to healthcare providers or patients themselves if anomalies occur, allowing for timely interventions.

By bringing computation closer to data sources and end-users, edge computing enhances performance, responsiveness, and scalability, making it an important technology for improving application performance.

Let's look at some DNS routing strategies to achieve low latency if your application is deployed globally.

Defining a DNS routing strategy

You may deploy your application in multiple geographical regions to have a global reach. When it comes to user request routing, you need to route user requests to the nearest and fastest available server for a quick response from your application. The DNS router provides the mapping between the domain names and the IP addresses. It ensures that the requests are served by the correct server when the user types in the domain name—for example, when you type `amazon.com` in your browser to do some shopping, your request is always routed to the Amazon application server DNS service.

AWS provides a DNS service called **Amazon Route 53**, where you can define a different kind of routing policy as per your application's needs. Amazon Route 53 provides DNS services to simplify domain management. The following are the most common routing policies:

- **Simple routing policy:** As the name suggests, this is the most straightforward routing policy and doesn't involve any complications. It is helpful to route traffic to a single resource—for example, a web server that is used to deliver information to a specific website.
- **Failover routing policy:** This routing policy requires you to achieve high availability by configuring active-passive failover. If your application goes down in one region, all the traffic can automatically be routed to another region.
- **Geolocation routing policy:** If the user belongs to a particular location, you can use a geolocation policy. A geolocation routing policy routes traffic to a specific region.
- **Geoproximity routing policy:** This is like a geolocation policy, but you can shift traffic to nearby locations when needed.

- **Latency routing policy:** If your application runs in multiple regions, you can use a latency policy to serve traffic from the region where the lowest latency can be achieved.
- **Weighted routing policy:** A weighted routing policy is used for A/B testing, where you want to send a certain amount of traffic to one region and increase this traffic as your trial proves more and more successful.

Additionally, Amazon Route 53 can detect anomalies in the source and volume of DNS queries and prioritize requests from users that are known to be *reliable*. It also protects your application from a DDoS attack.

Once traffic passes through the DNS server, in most cases, the next stop will be a load balancer, which will distribute traffic among a cluster of servers. Let's learn some more about load balancers.

Applying a load balancer

Load balancers distribute network traffic across servers to improve concurrency, reliability, and application latency. Load balancers can be *physical* or *virtual*. It would be best to choose a load balancer that suits your application's needs. Commonly, two types of load balancers can be utilized by an application:

- **Layer 4 or network load balancer:** Layer 4 load balancing routes packets based on information in the packet header—for example, source/destination IP addresses and ports. Layer 4 load balancing does not inspect a packet's contents, making it less compute-intensive than Layer 7 or application load balancing, and, therefore, faster. A network load balancer can handle millions of requests per second.
- **Layer 7 or application load balancer:** Layer 7 load balancing inspects and routes packets based on the full contents of the packet. Layer 7 is used in conjunction with HTTP requests. The materials that inform routing decisions are factors such as HTTP headers, URI path, and content type. It allows for more robust routing rules but requires more compute time to route packets. An application load balancer can route requests to containers in your cluster based on their distinctive port number.

Depending on the environment, you can choose hardware-based load balancers, such as an F5 load balancer or a Cisco load balancer. You can also select a software-based load balancer, such as **Nginx**.

AWS provides a managed virtual load balancer called **Amazon Elastic Load Balancing (ELB)**. ELB can be applied at Layer 7 as an application load balancer and Layer 4 as a network load balancer.

A load balancer is an excellent way of securing your application, making it highly available by sending requests to healthy instances. It works together with auto-scaling to add or remove instances as required. Let's look at auto-scaling and learn how it helps to improve overall performance and ensure the high availability of your application.

Applying auto-scaling

You learned about auto-scaling in *Chapter 2, Principles of Solution Architecture Design*. You learned about predictive and reactive auto-scaling in the *Design for scale* section. The concept of auto-scaling became popular with the agility provided by cloud computing platforms. Cloud infrastructure allows you to quickly scale up or scale down your server fleet based on user or resource demand.

With a public cloud platform such as AWS, you can apply auto-scaling at every layer of your architecture. You can scale the web server fleet based on the number of requests in the presentation layer, and scale the application layer based on the server's memory and CPU utilization. You can also perform scheduled scaling if you know the traffic pattern when the server load increases. At the database level, auto-scaling is available for relational databases such as Amazon Aurora Serverless and Microsoft Azure SQL Database. A NoSQL database such as Amazon DynamoDB can be auto-scaled based on throughput capacity.

When auto-scaling, you need to define the number of desired server instances. You need to determine the maximum and minimum server capacity as per your application's scaling needs. The following screenshot illustrates the auto-scaling configuration in the AWS cloud as an example:

Edit details - ASG-SA

Launch Instances Using Launch Template Launch Configuration

Launch Configuration webserverCopy

Desired Capacity 3

Min 2

Max 5

Availability Zone(s) eu-west-1a eu-west-1b

Subnet(s) subnet-0be5c2b238624205e(10.0.0.0/24) | PublicSubnetA-saurabh | eu-west-1a
subnet-0a499ab52ff71bacd(10.0.1.0/24) | PublicSubnetB-saurabh | eu-west-1b

Classic Load Balancers testec2-SAelb-1OE7V07XZAQVB

Target Groups

Health Check Type EC2

Health Check Grace Period 300

Instance Protection

Cancel **Save**

Figure 6.6: Auto-scaling configuration

In the preceding auto-scaling configuration setting, if three web server instances are running, it can scale up to five instances if the CPU utilization of servers goes above 50% and scale down to two instances if the CPU utilization goes below 20%. If an instance becomes unhealthy in a standard scenario, the total number of instances will fall below the desired capacity. In this case, the load balancer will monitor the instance's health and use auto-scaling to provide new instances. The load balancer monitors instance health and triggers auto-scaling to provision new instances as required.

Auto-scaling is a good feature to have, but make sure you set up the appropriate configuration to limit the cost of a change in CPU usage. Auto-scaling can significantly increase costs in the case of unforeseen traffic due to a **distributed denial of service (DDoS)** attack. It will help to protect your system from such events. You will learn more about this in *Chapter 7, Security Considerations*.

In this section, you have learned about various networking components that can help to improve application performance. You can optimize your application network traffic according to your user location and application demand. Since mobile is becoming the go-to user interface for many applications, you should do proactive performance monitoring of mobile apps to improve the customer experience. Let's learn more about performance considerations for mobile apps next.

Performance considerations for mobile applications

Mobile applications have now become an essential part of many digital platforms. Nowadays, users often first check out the mobile application before visiting the desktop website. Moreover, a significant portion of user traffic is driven via mobile applications, making it crucial to ensure these apps are highly performant. As mobile apps increasingly become a staple of our digital interactions, it's paramount to guarantee their performance, security, and usability. Let's delve into some best practices for building efficient mobile applications.

Optimization of load times

In mobile applications, load times are a critical factor that can either augment user engagement or serve as a deterrent. Quick and efficient load times are indispensable, especially considering users often utilize applications on the go and require immediate responses. Some ways to enhance load times are to optimize image sizes, employ lazy loading for elements, and ensure that the initially visible content loads promptly.

Efficient use of resources

Limited resources, such as CPU, memory, and battery, constrain mobile devices. To ensure an app runs smoothly without draining the device's resources, developers need to prioritize minimizing the usage of these resources. Strategies include using efficient algorithms, reducing memory leaks by properly managing memory allocation, and optimizing queries to fetch only the necessary data.

Responsive user interface (UI)

The UI should be intuitive and highly responsive, ensuring that user inputs yield immediate feedback. To achieve this, any computationally intensive process, such as data retrieval or image processing, should be executed in the background, preventing it from interfering with UI interactions. Using asynchronous programming and multithreading can keep the UI agile and responsive.

Network efficiency

Considering the potential for unstable or slow network connections in mobile environments, an application should proficiently manage network requests. Implementing caching for data that does not change frequently, optimizing API calls, and gracefully handling network failures by providing appropriate user feedback can significantly enhance user experience and app performance.

Battery consumption

An app that excessively drains the battery will quickly fall out of favor with users. You should be mindful of optimizing processes and managing background tasks to minimize power consumption. Ensuring that GPS, Bluetooth, and other power-intensive processes are used judiciously and turned off when not needed is crucial.

Cross-platform compatibility

With a multitude of devices, operating systems, and screen sizes available, an app should maintain high performance across various platforms. Utilizing cross-platform development frameworks and thoroughly testing the app on diverse devices ensures a consistent and optimal user experience.

User experience (UX) design

Ensuring the UX is seamless and intuitive is central to any application's success. This involves crafting a user-friendly design, ensuring ease of navigation, and maintaining a logical flow throughout the app, ensuring users can perform their desired actions with minimal effort.

Effective data management

Managing data efficiently by utilizing local storage for frequently used data and ensuring smooth synchronization between local and remote data is essential for providing users with up-to-date information without compromising performance.

Testing and quality assurance

Implementing rigorous testing protocols, which include performance testing under various conditions and on diverse devices, ensures that the app can maintain optimal performance even under strain. Employing automated testing and continuous integration can assist in identifying and rectifying issues promptly during the development phase.

Constructing a high-performance mobile application involves a harmonious blend of user-centric design and technical proficiency. By meticulously optimizing every facet of an application, from the interface and load times to data management and security features, developers can ensure it performs optimally in varied conditions and across numerous devices. While you are implementing various strategies to improve application performance, it's always recommended to test it as well. Let's learn more about performance testing.

Performance testing

Performance testing is a critical subset of software testing that aims to ensure that a software application will perform well under expected workloads. It revolves around gauging an application's stability, speed, responsiveness, and scalability under various circumstances. Rather than identifying bugs or defects, performance testing determines how an application will react to different demand levels. Given the fact that today's users expect seamless and swift functionality, performance testing has never been more crucial.

The significance of a well-performing application can't be overstated in today's digital age. Primarily, it directly affects user satisfaction. Users are accustomed to swift and seamless interactions on their devices; thus, a sluggish or frequently crashing app can be a major turn-off. Nobody wants to waste time on an application that doesn't deliver promptly, especially during moments of high demand or peak usage. The frustration from such experiences can lead users to abandon the app entirely, favoring competitors that provide a smoother experience.

Consider a popular e-commerce website gearing up for a Black Friday sale. With thousands, if not millions, expected to visit the site, it's imperative to ensure the system won't crash, transactions will process swiftly, and the user experience will remain smooth, even with the surge of users. Performance testing, in this case, is not just helpful but essential.

Types of performance testing

There are several types of performance testing, each designed to address specific aspects of an application's performance. Here's a brief overview of the main types of performance testing:

- **Load testing:** This type of testing is about understanding how a system behaves under expected, real-world loads. It's the equivalent of testing a bridge by gradually adding weight until it carries the anticipated maximum number of vehicles. For example, if an e-commerce website expects 10,000 visitors during a holiday sale, load testing will simulate these 10,000 users to ensure the site works smoothly under such conditions.
- **Stress testing:** Imagine cramming too many people into an elevator, beyond its capacity, to see if it still operates or breaks down; that's the essence of stress testing. It's designed to push systems to their breaking point, ensuring that even in worst-case scenarios, failures don't lead to catastrophic results. For instance, a banking app might be stress tested to see how it behaves if a million users tried to log in simultaneously, far exceeding its regular traffic.
- **Endurance testing:** The question endurance testing answers is "Can the system run efficiently when subjected to an expected load for an extended period of time?" For example, a streaming service might be endurance tested to ensure it can stream movies and shows to users continuously over a weekend without any degradation in quality or speed.
- **Spike testing:** In the real world, user traffic can be unpredictable. Spike testing is like observing how a power grid reacts when everyone suddenly turns on their air conditioners simultaneously during a heatwave. An example might be testing a news website during a major event, like the Olympics, where sudden influxes of users might want to check results or updates.

- **Volume testing:** Here, the emphasis is on data. It's analogous to checking how well a library can organize and lend millions of books. For a database-driven application, volume testing might involve seeing how the system behaves when the database has billions of entries. A practical example is a global email service testing its system's responsiveness when searching through vast numbers of stored emails.

Many performance testing tools are available, such as **JMeter**, **LoadRunner**, and **WebLoad**. These tools simulate various scenarios and loads to test the application's performance.

Performance testing plays a crucial role in the software development lifecycle. Ensuring an application's robustness, reliability, and speed are vital for its success in the real world.

Performance testing and performance monitoring are two critical aspects of ensuring an application's efficiency and reliability, but they serve different purposes throughout the development and deployment lifecycle. Performance testing is about identifying potential performance issues before they affect users, while performance monitoring is about keeping an eye on the system's performance and quickly reacting to any issues that occur after deployment. Let's learn more about performance monitoring in the next section.

Managing performance monitoring

Performance monitoring is essential when you are trying to keep an eye on performance issues and proactively reduce end-user impact.

You should define your performance baseline and raise the alarm to the team in the case of a threshold breach—for example, an application's mobile app load time should not be more than three seconds. Your alarm should be able to trigger an automated action to handle poorly performing components—for example, adding more nodes to a web application cluster to reduce the request load.

There are multiple monitoring tools that measure application performance and overall infrastructure. You can use a third-party tool like Splunk or the AWS-provided Amazon CloudWatch to monitor any application.

Monitoring solutions can be categorized into **active monitoring** and **passive monitoring** solutions:

- With active monitoring, you must simulate user activity and identify performance gaps up front. Application data and workload situations constantly change, requiring continuous proactive monitoring. Active monitoring works alongside passive monitoring as you run the known possible scenarios to replicate the user experience. You should run active monitoring across all dev, test, and prod environments to catch any issue before reaching the user.
- Passive monitoring tries to identify unknown patterns in real time. For a web-based application, passive monitoring needs to collect essential metrics from the browser that can cause performance issues. You can gather metrics from users regarding their geolocation, browser types, and device types to understand your application's user experience and geographic performance. Monitoring is all about data, including the ingestion, processing, and visualization of lots of data.

Performance always comes at a cost, and, as a solution architect, you need to think about the trade-offs to take the right approach. For example, an organization's internal applications, such as the timesheet and HR programs, may not need as high performance as external products, such as e-commerce applications. An application that deals with trading (for example) needs very high performance, which requires more investment. You should balance durability, consistency, cost, and performance to suit your application's needs. You will continue to learn about the various monitoring methods and tools in upcoming chapters and dive deep into monitoring and alerts in *Chapter 8, Architectural Reliability Considerations*.

Tracking and improving performance are complex tasks where you need to collect lots of data and analyze patterns. An access pattern helps you to make the right choice for performance optimization. Applying continuous active monitoring in combination with passive monitoring helps you to maintain consistent performance for your application.

Summary

In this chapter, you learned about the various architecture design principles that impact the performance of applications. You learned about latency and throughput at different layers of architecture and how they relate to each other.

For highly performant applications, you need to have low latency and high throughput at every architecture layer. Concurrency helps to process a large number of requests. You also learned the differences between parallelism and concurrency and gained insights into how caching can help improve overall application performance.

Then, you learned about choosing your technology and its working models, which can help achieve your desired application performance. While looking at the compute options, you learned about the various processor types and the differences between them to help you make the right choice when selecting server instances. You learned about containers and how they can help you to utilize resources efficiently and, at the same time, help to improve performance. You also learned how Docker and Kubernetes work well with each other and fit into your architecture.

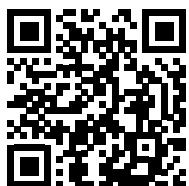
In the section on choosing storage, you learned about different kinds of storage, such as block, file, and object storage, and the differences between them. You also learned about the available storage choices in on-premises and cloud environments.

In the section on choosing a database, you learned about the various database types, including relational databases, nonrelational databases and data warehouses. You also learned about the different request routing strategies that can help you improve network latency for your globally distributed users, and you learned how load balancers and auto-scaling can help you manage many user requests without compromising application performance. As mobile apps are critical for any application, you learned about performance considerations for mobile applications. You also learned about the importance of performance testing and monitoring your application for performance.

In the next chapter, you will learn how to secure your application by applying authentication and authorization, which will help you to secure your data both at rest and in transit and ensure your application is protected from threats and attacks. You will also learn about compliance requirements and how to satisfy them when designing your application. You will learn about security audits, alerts, monitoring, and automation.

Join our book's Discord space

Join the book's Discord workspace to ask questions and interact with the authors and other solution architecture professionals: <https://packt.link/SAHandbook>



7

Security Considerations

Security is always at the center of architectural design. Many enterprises suffer financial losses due to security breaches when their customer data gets leaked. Organizations can, therefore, not only lose customer trust but also the entire business.

There are many industry-standard compliances and regulations out there to make sure your application is *secure* and protects customer-sensitive data. In the previous chapter, you learned about various aspects of performance improvement and technology choices for your architecture. In this chapter, you will gain an understanding of best practices to secure your application and make sure it is compliant with industry-standard regulations.

Security in architecture doesn't only mean protecting the edge of your IT workload. It also includes making sure that different parts of your application infrastructure are safe from each other. For example, on a server, you can use a firewall to control which data can enter or leave and where it can go. This way, if one part has a security issue, it won't affect the rest. You need to do the same for all parts, such as your data and programs. Security needs to be applied to every layer and every component of the architecture. This chapter also discusses different ways to keep your cloud system safe.

You will learn about the following security practices in this chapter:

- Design principles for architectural security
- Selecting technology for architectural security
- Security and compliance certifications
- The cloud's shared security responsibility model
- Security threat modeling

Design principles for architectural security

Security is all about the ability to protect your system and information while delivering business value for your customers. Lack of good security can have a serious impact on your customers and business.

You need to conduct an in-depth security risk assessment and plan a mitigation strategy for the continuous operation of your business. The following sections talk about standard design principles that will help you strengthen your architectural security.

Implementing authentication and authorization control

The purpose of **authentication** is to determine whether a user can access the system with the provided credentials, while **authorization** determines what a user can do once they are inside the system.

You should create a centralized system to manage your users' authentication and authorization. A centralized user management system helps you to keep track of users' activity so you can deactivate them if they are no longer a part of the system or are no longer using it appropriately. You can define standard rules to onboard a new user and remove access for inactive users. A centralized system eliminates reliance on long-term credentials and allows you to configure other security methods, such as password rotation.

For authorization, you should start with the **principle of least privilege**—this means users should not have any access to begin with and are assigned only the required access types according to their job role. Creating an access group according to job role helps to manage the authorization policy in one place and apply authorization restrictions across a large number of users. For example, you can restrict the development team to have full access to the development environment and read-only access to the production environment. If any new developer joins, they should be added to this dev group, where all authorization policies are managed centrally.

Enabling **single sign-on (SSO)** with a centralized user repository helps to reduce the hassle of remembering multiple passwords for your user base and eliminates any risk of password leakage. To further enhance security, integrating **multi-factor authentication (MFA)** with SSO adds an additional layer of protection. MFA requires users to provide two or more verification factors to gain access to a resource, such as a security token, a fingerprint, or facial recognition.

Large organizations use centralized user management tools such as **Active Directory (AD)** for employee authentication and authorization to provide access to internal enterprise applications such as the HR system, the expense system, and the timesheet application.

In a customer-facing application, such as e-commerce and social media websites, you can use an OpenID authentication system to maintain a centralized system. OpenID is an open standard authentication protocol. You will learn about large-scale user management tools in more detail in the *OAuth and OpenID Connect* section of this chapter.

Applying security everywhere

Often, organizations primarily focus on ensuring the physical safety of their data center and protecting the outer networking layer from any attack. Instead of just focusing on a single outer layer, ensure that security is applied at every application layer.

Apply the **defense-in-depth** (DiD) approach to layer security controls throughout an application; for example, a web application needs to be secured from external internet traffic by protecting the **Enhanced Data rates for Global Evolution (EDGE)** network and **Domain Name System (DNS)** routing. Apply security at the load balancer and network layers to block malicious traffic.

Secure every application instance by allowing only required incoming and outgoing traffic in the web application and database layer. Protect operating systems with antivirus software to safeguard against any malware attack. Apply both proactive and reactive measures of protection by putting an **intrusion detection system (IDS)** and **intrusion prevention system (IPS)** in front of your traffic flow and using a **web application firewall (WAF)** to protect your application from various kinds of attacks. You will learn more details about the various security tools to use in the *Selecting technology for architectural security* section of this chapter.

Reducing the blast radius

While applying security measures at every layer, you should keep your system isolated in a small pocket to reduce the blast radius. If attackers get access to one part of the system, you should be able to limit a security breach to the smallest possible area of the application. For example, in a web application, keep your load balancer in a separate network from other layers of the architecture, as that will be internet-facing. Further more, apply network separation at the web, application, and database layers. If an attack happens in one layer, you should prevent it from expanding to other layers of the architecture.

The same rules are applied to your authorization system, giving the least privilege to users and providing only the minimum required access. Implement MFA so that even if there's a breach in user access, the attacker always needs a second level of authentication to get into the system.

Provide minimal access to the system to ensure that you are not exposing the entire system, and provide temporary credentials to make sure access is kept open for a short time. Take precautions when providing programmatic access by putting a secure token in place with frequent key rotation.

Monitoring and auditing everything all the time

You need a logging mechanism for every activity in your system, and you should conduct a regular audit. Audit capabilities are often required from various industry-compliance regulations. Collect logs from every component, including all transactions and each API call, to put centralized monitoring in place. It is a good practice to add a level of security and access limitations to a centralized logging account so that no one can tamper with it.

Take a proactive approach and be ready to handle any incident before the user is affected. Alert capabilities with centralized monitoring help you take quick action and mitigate any incident. Monitor all user activity and application accounts to limit a security breach.

Automating everything

Automation is essential for applying quick mitigation for any security-rule violation. You can use automation to revert changes against desired configurations and alert the security team—for example, if someone added admin users to your system and an open firewall to an unauthorized port or IP address. You can apply automation to remove such undesired changes in the system.



Applying automation to security systems has become popular with the concept of DevSecOps. DevSecOps is about adding security to every part of application development and operations. You will learn more about DevSecOps in *Chapter 11, DevOps and Solution Architecture Framework*.

Create secure architectures and implement security controls that are defined and managed as code. You can version-control your security as a code template and analyze changes as required. Automated security mechanisms as software code help you scale security operations more rapidly and cost-effectively.

Protecting data

Data is at the center of your architecture, and it is essential to secure and protect it. Most of the compliance regulations in place are there to protect customer data and identity. Most attacks have the intention of stealing user data.

You should categorize your data by its sensitivity level and protect it accordingly. For example, customer credit card information should be the most sensitive data and should be handled with the utmost care. On the other hand, a customer's first name may not be that sensitive, while the card number is sensitive information.

Protecting data throughout its lifecycle is crucial for maintaining its confidentiality, integrity, and availability. Data can exist in three states, and each state requires specific security measures to ensure comprehensive protection:

- **Data at rest:** This refers to data stored on physical media, whether on a server's hard drive, a laptop, a USB stick, or cloud storage. One protection mechanism for data at rest is encryption, which ensures that even if the storage device falls into the wrong hands, the data remains inaccessible without the encryption key. Additionally, you need to have access controls and regular audits in place to ensure that only authorized users can access or modify the data.
- **Data in transit:** When data moves across a network—from a user's computer to a server, between servers, or across the internet—it's considered in transit. To protect data in transit, you can use encryption protocols such as **Transport Layer Security (TLS)**. This ensures that even if the data is intercepted during transmission, it remains unreadable to the attacker.
- **Data in use:** This is often the most challenging state to protect because the data is being processed or used by applications. Encryption can protect data at rest and in transit, but once it's loaded into memory and being used by an application, it's in plaintext and potentially vulnerable. New technologies such as **trusted execution environments (TEEs)** and homomorphic encryption are emerging to secure data in use, allowing operations on encrypted data without needing to decrypt it first.

Create mechanisms and tools that minimize the need for direct access to data. Avoid manual data processing by applying tool-based automation that eliminates human error, especially when handling sensitive data. Apply access restrictions to the data wherever possible to reduce the risk of data loss or data modification.

Once you categorize data by sensitivity, you can use the appropriate encryption, tokenization, and access control to protect the data. Data needs to be protected not only at rest but also in motion—when being transmitted over the network—as well. You will learn about various mechanisms to protect data in the *Data security* section of this chapter.

Responding to security incidents

Keep yourself ready for any security events. Create an incident management process as per your organizational policy requirements. Incident management can differ from one organization to another and from one application to another. For example, if your application handles the **personally identifiable information (PII)** of your customers, you need tighter security measures in your incident response. However, if the application is handling small amounts of sensitive data, such as an inventory management application, then it will have a different approach.

Make sure to simulate the incident response to see how your security team would recover from a situation.

Your team should use automation tools to speed up detection, investigation, and response to any security event. You will need to set up the alert, monitor, and audit mechanisms to do **root cause analysis (RCA)** to prevent such events from occurring again.

In this section, you learned about the general security principles to apply in your architecture for application security. In the next section, you will learn how to apply these principles using different tools and techniques.

Selecting technology for architectural security

The previous section focused on the general application security rules to consider while designing architecture. Still, the question is: *How do we apply these rules to make the application secure during implementation?* There are various tools and technologies available for each layer of your application to make it secure.

In this section, you will learn in detail about the multiple technology choices you have in the areas of user management and the protection of the web layer, infrastructure, and data of your application. Let's start with the first area, user identity and access management.

User identity and access management

User identity and access management are vital parts of information security. This is because it's best to ensure that only authenticated and authorized users can access your system resources in a defined manner.

User management could be a daunting task as your organization grows and your product is more widely adopted. User access management should differentiate and manage access to an organization's employees, vendors, and customers.

Enterprise or corporate users could be the organization's employees, contractors, or vendors. They are specialist users with special privileges to develop, test, and deploy the application. In addition to that, they probably require access to other corporate systems to do their daily job—for example, an **Enterprise Resource System (ERP)**, a payroll system, an HR system, a timesheet application, and so on. As your organization grows, the number of users can grow from hundreds to thousands.

The end users are the customers who use your applications and have enough access to explore and utilize the desired feature of the application—for example, players of a gaming application, users of social media applications, or customers of an e-commerce website. The number of these users could be from thousands to millions as the popularity of your product or application grows. You need to take special care of security when exposing the application to external-facing internet traffic to protect it from various threats.

Let's talk about corporate user management first. You need to have a centralized repository to enforce security policies such as strong password creation, password rotation, and MFA for better user management. MFA provides another means of validating someone's identity if a password may have already been compromised. Popular MFA providers include Google Authenticator, Gemalto, YubiKey, RSA SecurID, Duo, and Microsoft Authenticator.

From a user-access perspective, **role-based authentication (RBA)** simplifies user management; you can create user groups for the users' roles and assign an appropriate access policy. As illustrated in the following diagram, you can have three groups—admin, developer, and tester—with the corresponding access policy applied to each group. For example, an admin can access any system, including production, while developer access is limited to the dev environment, and the tester can only access the test environment:

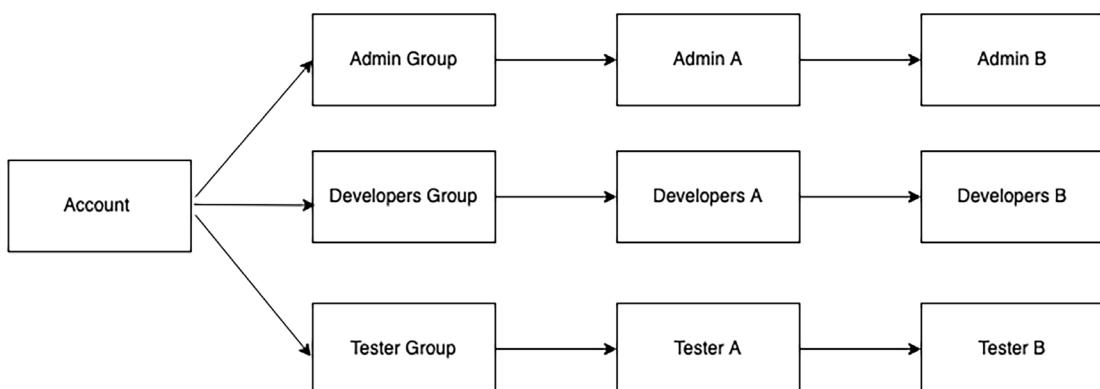


Figure 7.1: User group organization

As shown in the preceding diagram, when new users join the team, they get assigned to the appropriate group for their role. In this way, each user has a defined set of standard access. The user group can also update access in case a new development environment is introduced and all developers need to have access to it.

SSO is a standard process that helps to reduce security lapses and automate the system. SSO provides users with a login to the different corporate systems using a single user ID and password. **Federated Identity Management (FIM)** allows users to access the system without a password with a pre-authenticated mechanism. Let's look at some more details.

Federated identity management and single sign-on

FIM provides a way to connect the identity management system when user information is stored in a third-party **identity provider (IdP)**. With FIM, the user only provides authentication information to the IdP, which, in turn, already has a trusted relationship with the service.

As illustrated in the following diagram, when a user logs in to access a service, the service provider gets credentials from the IdP rather than getting them directly from the user:

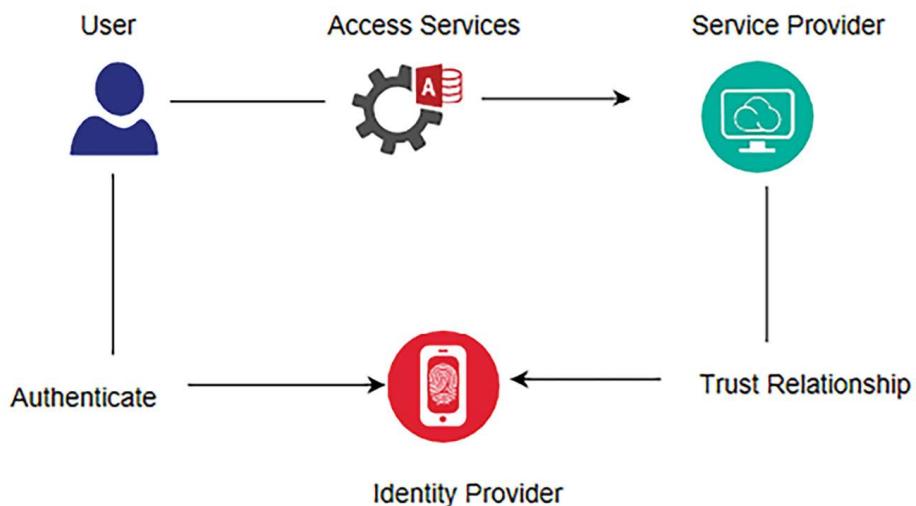


Figure 7.2: FIM authentication flow

SSO allows the use of a single set of sign-on details with which the user can access multiple services. Here, the service provider could target an environment to which you want to log in—for example, a **Customer Relationship Management (CRM)** application or your cloud application. The IdP could be a corporate AD. Federation allows authentication similar to an SSO but without a password, as the federation server knows users and allows them to access information.

There are various ways to implement FIM and SSO. Let's look at some of the popular **Identity and Access Management (IAM)** options.

Kerberos

Kerberos is an authentication protocol that allows two systems to identify each other securely and implements SSO. It works on the client-server model and uses a ticket system for user identity.

Kerberos has a **Key Distribution Center** (KDC) that facilitates authentication between two systems. The KDC consists of two logical parts—the **Authentication Server (AS)** and the **Ticket-Granting Server (TGS)**.

Kerberos stores and maintains the secret keys of each client and server in the data store. It establishes a secure session between two systems during their communication and identifies them with the stored secret key. The following diagram illustrates the flow of Kerberos authentication:

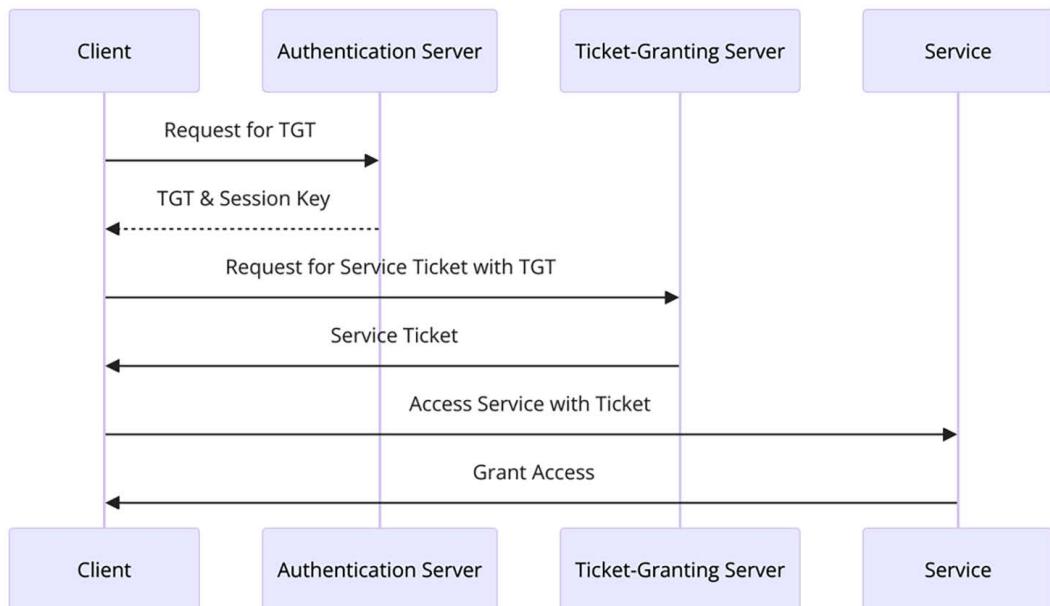


Figure 7.3: Kerberos authentication

As shown in the preceding diagram, when you want to access a service, the following steps are involved:

1. When you want to access a service on your computer network, your computer (the client) asks a special server called the **Authentication Server AS** for a ticket.
2. The AS checks if you're in its database. If you are, it creates a **Ticket-Granting Ticket (TGT)** and a session key, then sends them back to your computer. You can unlock the session key with your password, but you can't unlock the TGT because it's locked with a key that only the **Ticket-Granting Server** has.
3. Your computer takes this TGT and asks another server, the TGS, for a service ticket to access the service you want.
4. The TGS checks the TGT and, if everything looks good, sends back a service ticket that your computer can use to prove to the service that you have permission to access it.
5. Your computer shows this ticket to the service, and if the service agrees that the ticket is valid, you get access.

While Kerberos can be beneficial, it is an open-source protocol, and generally, large enterprises like to use more managed software with robust support, such as AD. Let's look at the working mechanism of one of the most popular user management tools, Microsoft AD, which is based on the **Lightweight Directory Access Protocol (LDAP)**.

Microsoft Active Directory

AD is an identity service developed by Microsoft for users and machines. AD has a domain controller, also known as **Active Directory Domain Services (AD DS)**, which stores the user's information, access credentials, and identity, as well as system information.

The following diagram illustrates a simple flow of the authentication process:

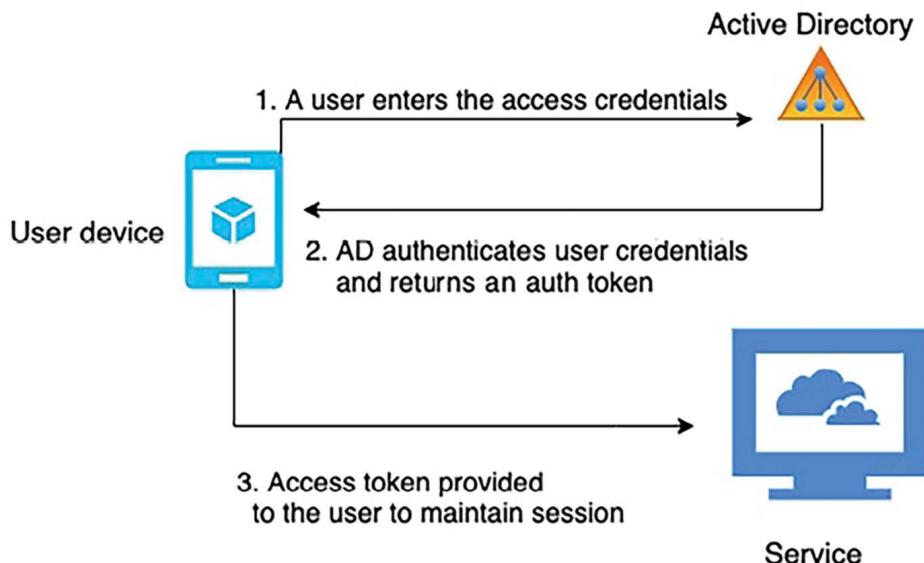


Figure 7.4: AD authentication flow

As shown in the preceding diagram, the user login is managed by AD on the domain networks. Users first send the request to the domain controller with their credentials and communicate with the **Active Directory Authentication Library (ADAL)**. The ADAL verifies the user credentials and sends back an access token with a continuous session for the requested service.

LDAP is the standard protocol that handles the tree-like hierarchical structure of information stored in directories. **Active Directory Lightweight Directory Services (AD LDS)** provides an LDAP interface to the directory of users and systems. For file encryption and network traffic encryption, **Active Directory Certificate Services (AD CS)** provides the key infrastructure functionality. **Active Directory Federation Services (AD FS)** provides access mechanisms for external resources, such as web app logins for many users.

As many organizations have started using cloud services, let's learn about the active directory service that AWS cloud provides.

Amazon Web Services Directory Service

Amazon Web Services (AWS) Directory Service connects AWS resources in your account with an existing on-premises user management tool such as AD. AWS Directory Service sets up a new user management directory in the AWS cloud. It facilitates a secure connection to the on-premises directory. After establishing the connection, all users can access cloud resources and on-premises applications with their existing credentials.

AWS AD Connector is another service that helps you connect the existing Microsoft AD to the AWS cloud; you don't need a specific directory synchronization tool. Admin users can manage AWS resources using AWS IAM.

AD Connector helps to enable MFA by integrating with your existing MFA infrastructure, such as YubiKey, a Gemalto token, or an RSA token.

For a small user base (fewer than 5,000 users), AWS provides Simple AD, which is a managed directory powered by *Samba 4 Active Directory Compatible Server*. Simple AD has common features such as user account management, user group management, SSO based on Kerberos, and user group policies.

Other directory services provided by major technology companies include Okta, Centrify, Ping Identity, and Oracle **Identity Cloud Service (IDCS)**.

Security Assertion Markup Language

Earlier in this section, in the *Federated identity management and single sign-on* section, you learned about IdPs and service providers. To access a service, the user gets validated by the IdP, which, in turn, has a trusted relationship with the service provider. **Security Assertion Markup Language (SAML)** can be used to establish a trusted relationship between an IdP and a service provider through **Extensible Markup Language (XML)**, which standardizes communication between an IdP and a service provider.

A SAML assertion is an XML document that the IdP sends to the service provider with user authorization. The following diagram illustrates the flow of the SAML assertion:

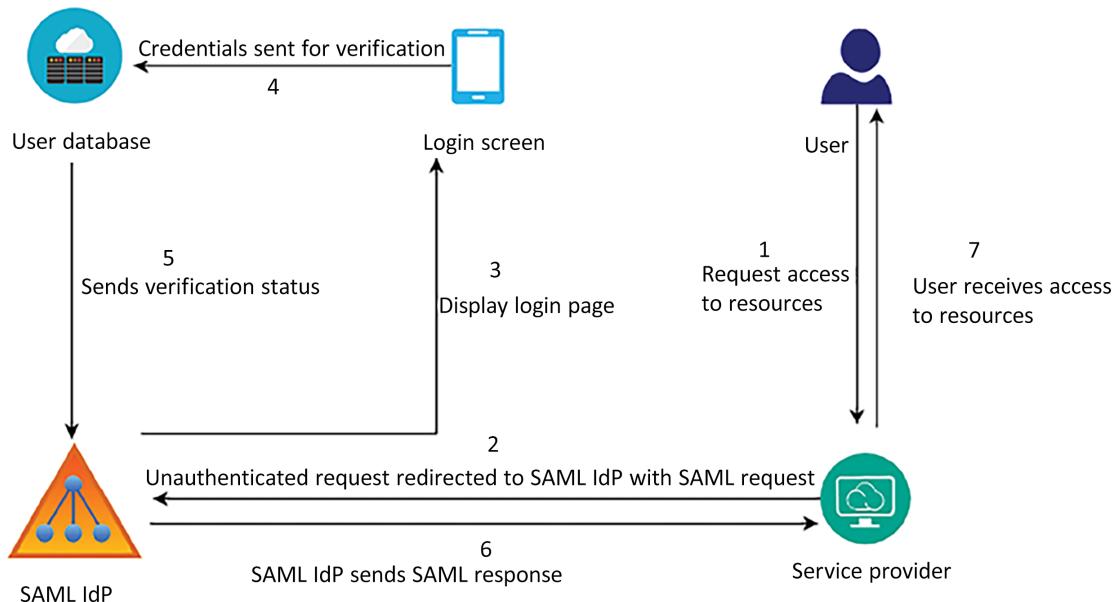


Figure 7.5: User authentication using SAML

As mentioned in the preceding diagram, the following steps are taken to implement user authentication using SAML:

1. A user sends a request to access the service—for example, the Salesforce CRM application—as a service provider.
2. The service provider (a CRM application) sends a SAML request with the user information to the SAML IdP.
3. The SAML IdP pops up the SSO login page, where users enter authentication information.
4. The user access credential goes to the user database, an identity store for validation. In this case, the user identity store is an AD.
5. The user identity store sends the user validation status to the SAML IdP, with whom the identity store has a trusted relationship.
6. The SAML IdP sends a SAML assertion to the service provider (a CRM application) with information about user verification.
7. After receiving the SAML response, the service provider allows application access to the user.

Sometimes, a service provider can act as an IdP as well. SAML is very popular for establishing a relationship between any identity store and service provider. All modern identity store applications are SAML 2.0-compatible, which allows them to communicate with each other seamlessly. SAML allows user identity to be federated and enables SSO for enterprise users.

For large user bases such as social media and e-commerce websites, **OAuth** (short for **Open Authorization**) and **OpenID** are more suitable than SAML. Let's learn about OAuth and **OpenID Connect (OIDC)**.

OAuth

OAuth is an open standard authorization protocol that provides secure access delegation to an application. OAuth doesn't share password data but uses an authorization token to establish the identity between service providers and consumers. Users of an application provide access to their information without giving login credentials.

While OAuth is mainly for authorization, many organizations have started adding their own mechanisms for authentication.

OIDC is a protocol that defines the authentication standard on top of the OAuth 2.0 authorization framework. While OAuth 2.0 provides a framework for authorization (granting access to resources), OIDC adds an additional layer to handle user authentication. This means that OIDC not only helps applications know what resources a user can access but also verifies the identity of the user accessing the service. It's a way for clients to verify a user's identity based on the authentication performed by an authorization server, as well as to obtain basic profile information about the user in an interoperable and REST-like manner.

Large technology companies such as Amazon, Facebook, Google, and X allow users to share information in their accounts with third-party applications. For example, you can log in to a new photo app using your Facebook login and authorize the new app to access only your Facebook photo information. The following diagram illustrates an OAuth access delegation flow, where a user requests the LinkedIn app to get their profile photo from Facebook:

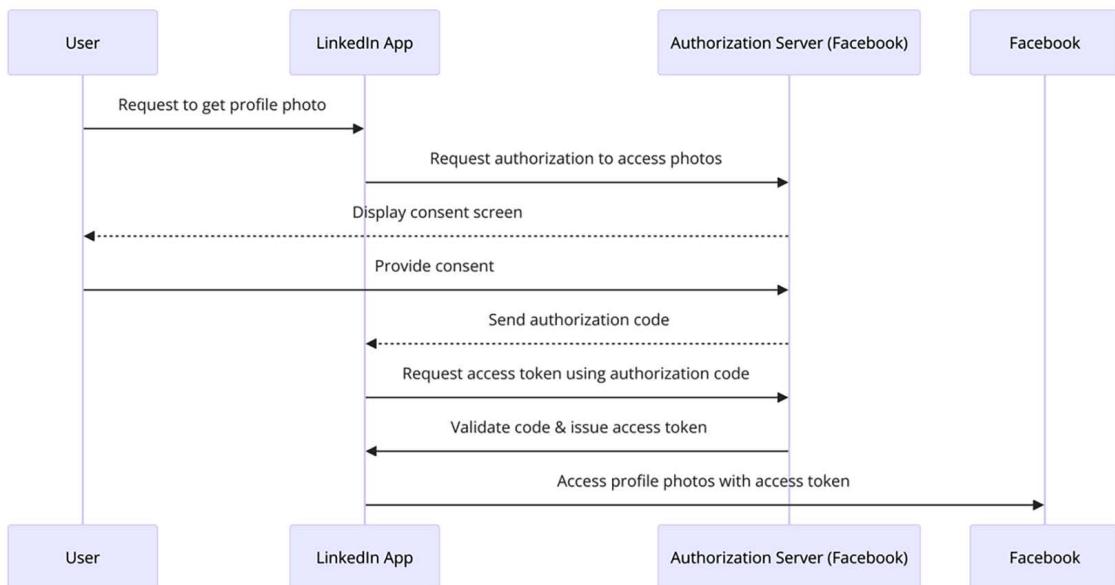


Figure 7.6: User access delegation with OAuth 2.0

As shown in the preceding diagram, the authentication flow follows these steps:

1. In this scenario, the user makes a request to the LinkedIn app to get your profile photo from Facebook.
2. The LinkedIn app requests authorization to access Facebook profile photos.
3. The authorization server (which is your Facebook account in this case) creates and displays a consent screen to the user.
4. The user provides consent to the request for the LinkedIn app to access only their Facebook profile photos.
5. After getting user approval, the Facebook authorization server sends an authorization code back to the requesting LinkedIn app.
6. The LinkedIn app then requests an access token from the authorization server (Facebook account) using the authorization code.
7. The authorization server identifies the LinkedIn app and checks the validity of the authentication code. If the access token is validated, the server issues an access token to the LinkedIn app.
8. The LinkedIn app can now access resources such as Facebook profile photos using the access token.



OAuth 2.0, which is faster than OAuth 1.0 and more comfortable to implement, is now most commonly used.

JSON Web Token (JWT) is a simple and accessible token format that can be used with OAuth and is popular with OpenID; we look at this next.

JWT

JWT is a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret or a public/private key pair.

A JWT has a JSON structure that has information about expiration time, issuer, subject, and so on. It is more robust than a **Simple Web Token (SWT)** and simpler than SAML 2.0. You can see a JWT in the following screenshot:

The screenshot shows a JWT decoding interface. At the top, there is a dropdown menu labeled "Algorithm" set to "HS256". Below this, there are two sections: "Encoded" and "Decoded".

Encoded: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxOTE5MjAyMCIsIm5hbWUiOiJTb2x1dGlvbiBBcmNoaXRlY3QgSGFuZGJvb2siLCJpYXQiOjIxMjExMDEwMX0.kjV743Dko6XciP3Kp3YrrRoVvSJvBXzK3JkHWqGKhIE

Decoded: EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

PAYOUT: DATA

```
{  
  "sub": "19192020",  
  "name": "Solution Architect Handbook",  
  "iat": 212110101  
}
```

VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  your-256-bit-secret  
)  secret base64 encoded
```

Figure 7.7: Sample JWT

The preceding screenshot depicts JWTs, which have three parts, each separated by a dot. The first part, the header, tells us the token type—JWT—and the algorithm it uses for signing, such as HS256 or RSA. The second part, the payload, holds claims, which are pieces of information about the user and other data. The last part is the signature, which makes sure the token hasn't been altered and confirms who sent the JWT.

JSON has a simpler structure than XML and is smaller, making JWT more compact than SAML. JWT is an excellent choice for passing information into HTML and HTTP environments. Due to their small size, JWTs are an ideal choice for passing the identity of authenticated users between services in a microservices architecture, or for providing access tokens that allow users to access resources. They're used in various authentication and authorization scenarios, particularly in web and mobile applications.

In this section, you learned about the most common user management tools and services. However, there are various other protocols and services available for user authentication and authorization. The implementation of the protocols mentioned previously can be complicated, and there is a large amount of packaged software available that makes the job easier.

Amazon Cognito is a user access management service provided by AWS that includes standard-based authorization such as SAML 2.0, OIDC, and OAuth 2.0, along with an enterprise user directory that provides the ability to connect with AD. Okta and Ping Identity provide enterprise user management and the ability to communicate with various service provider tools in one place.

Once your application is exposed to the internet, there are always various kinds of attacks that can happen. Let's learn about some of the most common attacks and how to set up the first layer of defense for web-layer protection.

Handling web security

As user demand is changing to require 24/7 availability of services, businesses are evolving to go into online mode, and to do so, they are adopting web application models. Web applications also help a company to gain a global customer base. Businesses such as online banking and e-commerce websites are always available, and they deal with sensitive customer data, such as payment information and payer identity.

Now, web applications are central to any business, and these applications are exposed to the world. Web applications can have vulnerabilities, exposing them to cyberattacks and data breaches. Let's explore some common types of cyberattack and how to mitigate them.

Cyberattacks

Web applications are vulnerable to security breaches. Hackers orchestrate cyberattacks from different locations and by various methods. Just as you lock and protect a physical building, in the same way, your web application needs to be protected from illicit activity. Let's explore some standard methods of attack that can cause security vulnerabilities in your web application.

Denial of service and distributed denial of service attacks

A Denial of Service (DoS) attack attempts to make your website unreachable to your users. To achieve a successful DoS attack, the attacker uses a variety of technologies that consume network and system resources, thus interrupting access for legitimate users. The attacker uses multiple hosts to orchestrate the attack against a single target.

A **Distributed Denial of Service (DDoS)** attack involves using many hijacked systems, often infected with malware, to flood a single target system with requests. This overwhelms the target, causing service disruptions. The attacker remotely controls the compromised systems to conduct the assault. As illustrated in the following diagram, a DDoS attack happens when multiple systems exhaust the bandwidth of resources of a targeted system:

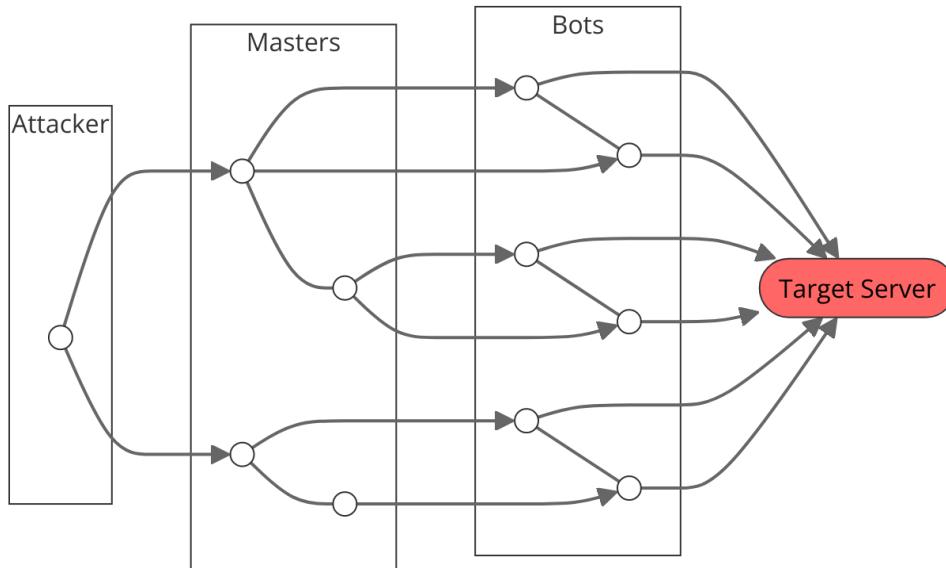


Figure 7.8: DDoS attack

The general concept of a DDoS attack is to leverage additional hosts to amplify the requests made to the target, rendering them overprovisioned and unavailable. A DDoS attack is often the result of multiple compromised systems, whereby a botnet puts a flood of traffic in the targeted system.



A botnet is a network of devices infected by and controlled through malware.

The most common DDoS attack happens at the application layer, using either a DNS flood or a **Secure Sockets Layer (SSL)** negotiation attack. In DNS floods, attackers exhaust the resources of a DNS server with too many requests. During SSL negotiations, attackers send a large amount of unintelligible data for computationally expensive SSL decryption. The attacker can perform other SSL-based attacks on the server fleet and overburden it with unnecessary task processing.

At the infrastructure layer, a typical DDoS attack happens in the following form:

- **User Datagram Protocol (UDP) reflection:** With UDP reflection, attackers spoof the target server's IP address and make a request that returns amplified significant responses from a hacked reflector server.

- **SYN floods:** With SYN floods, attackers exhaust the target server's **Transmission Control Protocol (TCP)** service by creating and abandoning high numbers of connections, blocking legitimate users from accessing the server.

Often, attackers try to get sensitive customer data, and for that purpose, they use a different kind of attack called **SQL injection (SQLi)**. Let's learn more about it.

SQLi attacks

As the name suggests, in a SQLi attack, attackers inject malicious **Structure Query Language (SQL)** to get control of a SQL database and fetch sensitive user data. The attacker uses SQLi to gain access to unauthorized information, take control of an application, add new users, and so on.

Take an example of a loan-processing web application. You have `loanId` as a field that customers can use to get all information related to their loan. If proper care is not taken, attackers can execute a query such as `SELECT * FROM loans WHERE loanId = 117 or '1=1'` and get access to the entire customer database, as this query will always return the true result.

The other common way to hack user data through script injection is **cross-site scripting (XSS)**, where a hacker impersonates a legitimate user. Let's learn more about it.

XSS attacks

You have probably encountered phishing emails that have links impersonating a website known to you. Clicking on these links may lead to compromised data through XSS. In an XSS attack, the attacker embeds their malicious code into a legitimate website. This code then executes when an unsuspecting user visits the web page.

The attacker can introduce this code in various ways, such as embedding it directly into a URL string or by inserting a snippet of JavaScript code onto the web page. When you load the web page, this client-side JavaScript code gets executed and steals your browser cookies.

These cookies often contain sensitive information, such as the access token and authentication to your banking or e-commerce websites. Using these stolen cookies, the hacker can enter your bank account, among other accounts, and take your hard-earned money.

Cross-site request forgery attacks

A **cross-site request forgery (CSRF)** attack takes advantage of user identity by creating confusion and tricking authenticated users into state-changing activities, for example, changing the password of a shopping website or requesting a money transfer to your bank.

It is slightly different from an XSS attack as, with CSRF, the attacker tries to forge a request rather than insert a code script. For example, the attacker can forge a request to transfer a certain amount of money from the user's bank and send that link in an email to the user. When users click on that link, the bank gets a request and transfers the money to the attacker's account. CSRF can be particularly harmful if attackers can get into an admin account.

Buffer overflow and memory corruption attacks

A software program writes data in a temporary memory area for fast processing, which is called a **buffer**. With a buffer overflow attack, an attacker can overwrite a portion of the memory connected with the buffer, deliberately causing a buffer overflow and accessing connected memory, where an application executable may be stored. The attacker can replace the executable with the actual program and take control of the entire system. Looking at the overall application, more security threats exist in the infrastructure layer, network layer, and data layer. Let's explore some standard ways to mitigate and prevent security risks at the web layer.

Web security mitigation

Security needs to be applied to every layer of your application, and special attention is required for the web layer due to its exposure to the world. For web protection, necessary steps include keeping up with the latest security patches, following the best software development practices, and ensuring that proper authentication and authorization are carried out.

Several methods exist to protect and secure web applications; let's explore two common methods.

Web application firewalls

WAFs are firewalls that apply specific rules to HTTP and HTTPS traffic (that is, ports 80 and 443). WAFs inspect your web traffic and verify that it conforms to the norms of expected behavior. They provide an additional layer of protection from web attacks.

WAF rate limiting is the ability to look at the amount or type of requests sent to your service and define a threshold that caps how many requests are allowed per user, session, or IP address. Approved and unapproved lists enable you to allow or block users explicitly.

AWS WAF is one example of a WAF that adds security to your web layer by creating and applying rules to filter web traffic. These rules are based on conditions that include HTTP headers, user geolocation, malicious IP addresses, custom **Uniform Resource Identifiers (URIs)**, and so on. AWS WAF rules block common web exploits such as XSS and SQLi. You can create a single set of rules for an environment that has various websites and web applications running and you can reuse rules across applications instead of recreating them.

Overall, a WAF is a tool that applies a set of rules to HTTP traffic. It filters web requests based on data such as IP addresses, HTTP headers, HTTP bodies, or URI strings. It can mitigate DDoS attacks by offloading illegitimate traffic. Let's learn more about DDoS mitigation.

DDoS mitigation

Resilient architecture can prevent or mitigate DDoS attacks. A fundamental principle in keeping your infrastructure secure is reducing the potential number of targets an attacker can hit. In short, if an instance doesn't need to be public, then don't make it public.

You can apply various strategies to minimize the attack surface area:

- Wherever possible, reduce the number of necessary internet entry points—for example, open incoming internet access to your load balancer, not web servers.

- Identify and eliminate any unnecessary entry points to the internet. For instance, you can set up file-share storage for vendors to upload data, but restrict the access to a limited group instead of making it accessible to the entire global internet traffic.
- Hide any required internet entry points from untrusted end users so that they cannot access them.
- Isolate the access point and apply a specific restrictions policy for end user traffic compared to application management traffic.
- Create a decoupled internet entry point to minimize the attack surface.

Your primary goal is to mitigate DDoS attacks at the edge location of the CDN. It's more challenging and costly to handle DDoS attacks if they get through to your application servers. The following diagram illustrates a DDoS mitigation example for an AWS cloud workload:

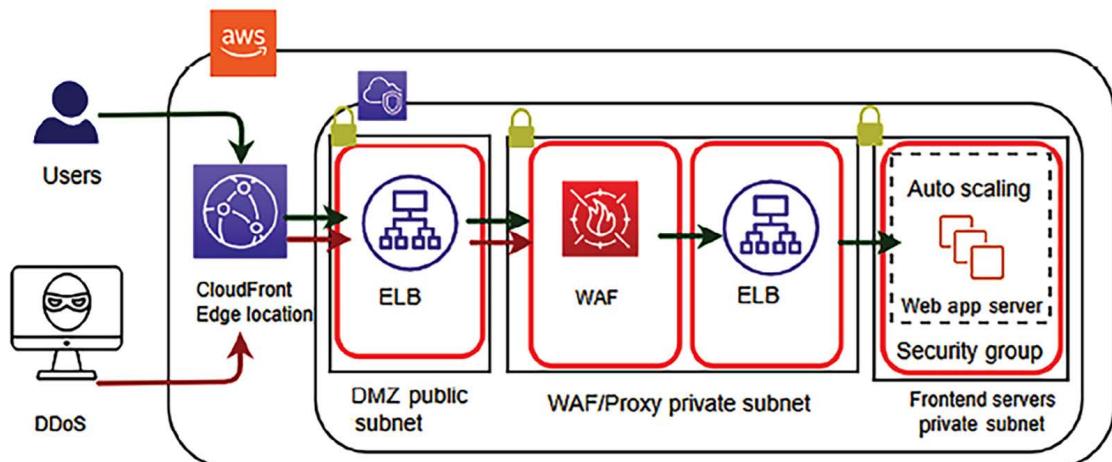


Figure 7.9: DDoS WAF sandwich mitigation strategy

The preceding diagram illustrates a **WAF sandwich architecture**, where the WAF appliance is staged between two load balancers to handle a DDoS attack.

Frequent DDoS attacks come from strategies such as SYN floods and UDP reflection, which Amazon CloudFront prevents by only accepting well-formed connections before the attacking strategy can reach your application servers. CDNs such as Amazon CloudFront help to tackle DDoS attacks by isolating them in a geographically isolated location and preventing the traffic from affecting other locations. Network firewall security helps you to control incoming and outgoing traffic at the individual server level.

As mentioned in the previous section, WAFs are used to protect web applications against exploit attacks such as XSS and SQLi attacks. In addition to this, WAFs also help to detect and prevent DDoS attacks at the web application layer.

To handle a DDoS attack, you can apply either horizontal or vertical scaling. You can take advantage of scaling in the following way:

1. First, select the right server size and configuration for your web application.
2. Second, use a load balancer to distribute traffic among the fleet of servers and add auto-scaling to add/remove servers as required.
3. Finally, use the CDN and DNS server, as they are built to handle traffic at scale.

Scaling for DDoS attacks is an excellent example of why it's essential to set reasonable maximum counts for your servers. A DDoS attack could scale your servers out to a number that would be extremely costly while still potentially not being able to prevent the server from becoming unavailable. Having reasonable maximum limits for the expectations of regular traffic spikes would prevent a DDoS attack from costing your company too much money.

In this section, you learned about various security risks and vulnerabilities at the web layer and some standard ways to protect them. As security needs to be applied to every layer, let's explore protecting the infrastructure layer in more detail.

Securing an application and its infrastructure

In the previous section, you learned about securing the web layer. As security needs to be applied at every layer of your workload, let's learn about securing the application and network layers of your architecture.

Application and operating system hardening

It is impossible to eliminate vulnerabilities in your application entirely but you can limit system attacks by hardening your application's operating system, filesystem, and directory. Once attackers can get into your application, they can get root access and orchestrate an attack on the entire infrastructure.

It is essential to limit attacks to the application level by restricting the directory by *hardening permission*. At the process level, restrict memory and CPU utilization to prevent a DoS attack.

Set the right permission at the file, folder, and file partition levels. Avoid giving root privileges to the application or its users. You should create a separate directory containing only the required access for each application so that only the required user has application access. Only use common access for some applications.

Automate application restart by using process tools such as **DAEMON Tools** and **Supervisord** to prevent a manual approach, whereby users need to log in to the server to start. For Linux operating systems, a utility such as **systemd** or **System V init** scripts can start/stop the application.

Software vulnerability mitigation and secure code

It is always recommended to apply the latest security patch provided by your operating system vendor. This helps to fill any security holes in the system and protect your system from vulnerabilities where attackers can steal your security certificate or run arbitrary code.

Keeping your system up to date with the latest security patches is very important. It is better to automate the process of the most recent patch installation as soon as it becomes available. However, running a security patch may sometimes break your working software, so it's useful to set up a **Continuous Integration and Continuous Deployment (CI/CD)** pipeline with automated testing and deployment. You will learn more about the CI/CD process in *Chapter 11, DevOps and Solution Architecture Framework*.

The AWS cloud provides a system manager tool that allows you to apply security patches and monitor your server fleet in the cloud. You can use a tool such as **auto-updates** or **unattended upgrades** to automate security patch installation. When you opt for managed services from cloud providers, you essentially free yourself from the underlying infrastructure's operational burden. The cloud provider takes care of the setup, management, operation, and optimization of the services. This includes regular maintenance tasks, such as patching, which is critical for security and performance.

Make sure to integrate secure coding best practices into your software development process, as recommended by the **Open Web Application Security Project (OWASP)**, details about which can be found here: owasp.org/www-project-secure-coding-practices-quick-reference-guide/stable-en/01-introduction/05-introduction.

Network security

When it comes to protecting your infrastructure, securing the network must be your first consideration. The physical security of your IT infrastructure in the data center must be taken care of by the data center providers. Let's talk about ensuring network security, which is your responsibility as an application owner.

Let's take an example from a public cloud provider such as AWS to help you understand network security. You can apply the same example to your on-premises or private cloud network infrastructure as well.

As illustrated in the following diagram, security should be applied at every layer, and trusted boundaries around each layer with minimal access should be defined:

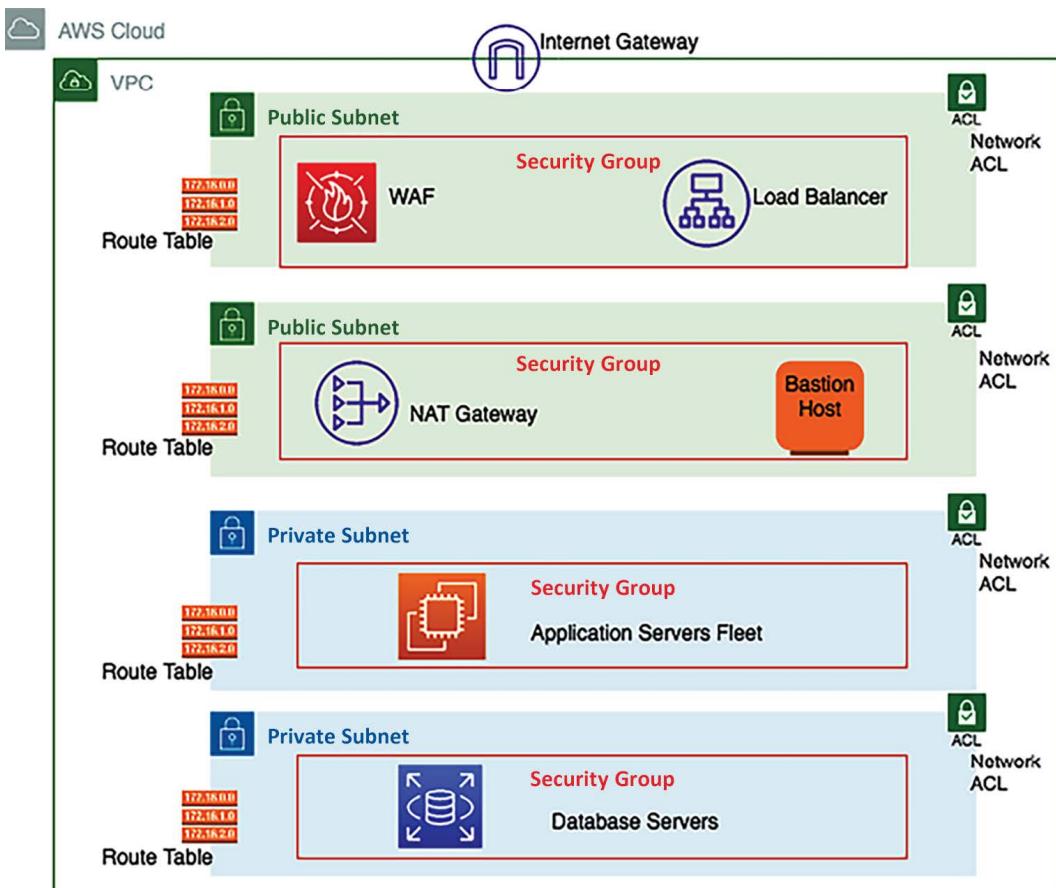


Figure 7.10: Network configuration for infrastructure security

In the preceding diagram, the load balancer is in a public subnet, which can accept internet traffic and distribute it to the application server fleet. WAF filter traffic is based on set rules and protects your application from various attacks, as you learned in the previous section. The application server fleet and database servers are in private subnets, which means direct internet access is not available.

Let's dive deep into the preceding architecture diagram and walk through each layer as follows:

- **Amazon Virtual Private Cloud (VPC)** provides a logically isolated network for your cloud infrastructure. It serves as your personalized network environment within the cloud, hosting various resources. Designed for enhanced control, it allows the segregation of different environments and resources. You can set up multiple VPCs within each AWS account or region. When setting up a VPC, you define its range of IP addresses using **Classless Inter-Domain Routing (CIDR)** notation. This notation is a concise way to represent a specific span of IP addresses. For instance, the CIDR block **10.0.0.0/16** encompasses all IP addresses from **10.0.0.0** to **10.0.255.255**. This range includes a total of 65,535 IP addresses available for use.

- **Subnets** are parts of a network, divided using CIDR ranges, that establish secure boundaries between private and public resources. Instead of organizing subnets by application or function (such as the web, app, or data tiers), it's more effective to arrange them based on internet access. This setup enables clear isolation at the subnet level, differentiating between public-facing and private internal resources.

In this setup, resources needing internet access, like public-facing load balancers, **network address translation (NAT)** instances, and bastion hosts, are placed in a **public subnet**. Other resources, such as databases and applications, reside in a private subnet. This creates distinct layers of separation between different resource tiers, with application instances and data resources each allocated to their own private subnets. On AWS, most resources can be hosted in **private subnets**, utilizing public subnets only when internet access is required. As a result, it's advisable to allocate a larger number of IP addresses to your private subnets than to your public subnets, which ensures ample space for the majority of your resources that will reside in the private network. Subnets provide basic separation between resources with **Network Access Control List (NACL)** rules, but security groups provide a more detailed level of traffic management. This approach prevents overly complex infrastructure and the inefficient use of IP addresses.

- A **routing table** consists of rules, known as routes, that dictate which application servers receive network traffic. For enhanced security, it's recommended to use a distinct custom route table for each subnet.
- **Security groups** function as virtual firewalls that manage both inbound and outbound traffic for one or more instances. These instances can be specified within a given CIDR block range or can be part of another designated security group. Following the principle of least privilege, security groups are set to deny all incoming traffic by default. You can then establish specific rules to filter traffic based on protocols such as TCP, UDP, and the **Internet Control Message Protocol (ICMP)**. This setup ensures that only the necessary and authorized traffic can access your instances, enhancing the security of your network.
- An **NACL** is an optional layer of security in the form of a virtual firewall, controlling both inbound and outbound traffic at the subnet level within your network. Unlike a security group, which is stateful, an NACL operates in a stateless manner. This statelessness means that each request, whether incoming or outgoing, is treated independently. For example, even if an incoming request is allowed through, the corresponding outbound response must also be explicitly permitted by the rules set in the NACL. This requires you to define both inbound and outbound traffic rules meticulously to ensure proper traffic flow and security at the subnet level.
- Internet traffic is routed through an **internet gateway (IGW)** to make a subnet public. By default, internet accessibility is denied for internet traffic in your environment. An IGW needs to be attached to your VPC, and the subnet's route table should define the rules of the IGW.
- A private subnet blocks all incoming and outgoing internet traffic but servers may need outgoing internet traffic for software and security patch installation. A **NAT gateway** enables instances in a private subnet to initiate outbound traffic to the internet and protects resources from incoming internet traffic.

- A **bastion host** acts like a jump server, which allows access to other resources in the private subnet. A bastion host needs to be hardened with tighter security so that only appropriate people can access it. To log in to the server, always use *public-key cryptography* for authentication rather than a regular user ID and password method.

Organizations often gather, store, and review network flow logs for multiple reasons. These include diagnosing connectivity problems, resolving security issues, and evaluating network access policies. You need to monitor traffic flow to your system VPC, which includes recording incoming and outgoing traffic information from your network. **VPC flow logs** enable you to capture that information, along with accepted and rejected traffic information for the designated resource, which helps you to understand traffic patterns better.

Flow logs serve as a security tool for monitoring traffic to instances. You can set up alarms for specific traffic types and create metrics to spot trends and patterns. Flow logs can be established for a VPC, a subnet, or a network interface. When created for a subnet or VPC, they monitor each network interface within that subnet or VPC. For example, consider a scenario where you have a VPC with multiple subnets. By setting up a flow log for the VPC, you can monitor all incoming and outgoing traffic across its network interfaces. If you notice unusual traffic patterns, like an unexpected spike in data requests from an unknown IP address, you can configure an alarm to alert you. This proactive monitoring helps to identify potential security threats or network inefficiencies early on.

As you can see, there are multiple layers of security available at the network layer that can help protect your infrastructure. Keeping resources in their isolated subnet helps to reduce the blast radius. If an attacker can penetrate one component, you should be able to restrict them to limited resources. You can use an IDS and an IPS in front of your infrastructure to detect and prevent any malicious traffic. Let's learn about them next.

Intrusion detection system and intrusion prevention system

An IDS detects any cyber attack happening through network traffic by recognizing an attack pattern. An IPS goes a step further and proactively helps to stop malicious traffic.

An IPS provides critical analysis of potential threats, positioned behind the firewall. It identifies dangerous content, like malicious packets, and can block traffic or reset connections. The IPS uses two primary detection methods:

- **Signature-based detection:** This method relies on a growing database of unique patterns, or “signatures,” associated with each known exploit.
- **Statistical anomaly-based detection:** This approach establishes a baseline for normal network performance and compares random samples of network traffic to this baseline. If traffic deviates significantly, the IPS intervenes.

You will need to determine the applicability of the IDS/IPS system to your application's requirements. An IDS can be host-based or network-based.

Host-based IDS

A host- or agent-based IDS runs on each host of your environment. It can review the activity within that host to determine if an attack has occurred and has succeeded. It can do this by inspecting logs, monitoring the filesystem, monitoring network connections to the host, and so on. The software or agent then communicates with a central/command application about the health or security of the host it monitors.

The pros of host-based solutions include that they can deeply inspect the activity inside each host. They can scale horizontally as required (each host gets its own agent) and do not need to impact the performance of running applications. The cons include the additional configuration management overheads that can be introduced if agents are managed on many servers, which are burdensome for an organization.

As each agent operates in isolation, widespread/coordinated attacks can be hard to detect. To handle coordinated attacks, the system should respond immediately across all hosts, which requires the host-based solution to cooperate with the other components, such as the operating system and the application interface, deployed on the host.

Network-based IDS

A network-based IDS inserts an appliance into the network, through which all traffic is routed and inspected for attacks.

The pros include a simple/single component that needs to be deployed and managed away from the application hosts. Also, it is hardened or monitored in a way that might be burdensome across all hosts. An individual/shared view of security exists in a single place so that the big picture can be inspected for anomalies/attacks.

However, a network-based IDS includes the performance hit of adding a network hop to applications. Having to decrypt/re-encrypt traffic to inspect it is both a massive performance hit and a security risk that makes the network appliance an attractive target. Any traffic that the IDS cannot decrypt cannot inspect/detect anything.

An IDS is a detection and monitoring tool and does not act on its own. An IPS detects, accepts, and denies traffic based on set rules. IDS/IPS solutions help to prevent DDoS attacks due to their anomaly-detection capabilities that make them able to recognize when valid protocols are used as an attack vehicle. An IDS and an IPS work by analyzing network packets and comparing their contents against a database of known threats. This process allows them to identify and respond to potential security risks. Continuous auditing and scanning are required for your infrastructure to secure it from any attack proactively.

In this section, you learned all about securing your infrastructure from various types of attacks. The goal of these attacks is to get hold of your data. You should secure your data in such a way that an attacker cannot acquire sensitive information even after getting hold of the data. Let's learn about data protection using security at the data layer, encryption, and backup.

Data security

In today's digital world, every system revolves around data. Sometimes, this data may contain sensitive information such as customer health records, payment information, and government identity. Securing customer data to prevent any unauthorized access is highly important. Many industries place significant stress on data protection and security.

Before architecting any solution, you should define basic security practices that align with your objective, such as complying with regulatory requirements.

There are several different approaches used to protect data. Let's look at them in the upcoming sections.

Data classification

One of the best practices is to classify your data, which provides a way to categorize and handle organizational data based on sensitivity levels.

According to data sensitivity, you can plan data protection, data encryption, and data access requirements.

By managing data classification as per your system's workload requirements, you can create the data controls and level of access needed for the data. For example, content such as a user rating and review is often public, and it's fine to provide public access, but user credit card information is highly sensitive data that needs to be encrypted and put under very restricted access.

At a high level, you can classify data into the following categories:

- **Restricted data:** This contains information that could harm the customer directly if it got compromised. Mishandling of restricted data can damage a company's reputation and impact a business adversely. Restricted data may include customer PII data, such as social security numbers, passport details, credit card numbers, and payment information.
- **Private data:** Data can be categorized as private if it contains customer-sensitive information that an attacker can use to plan to obtain their restricted data. Private data may include customer email IDs, phone numbers, full names, and addresses.
- **Public data:** This is available and accessible to everyone and requires minimal protection—for example, customer ratings and reviews, customer location, and customer username if the user made it public.

You can have a more granular category depending on the type of industry and the nature of the user data. Data classification needs to balance data usability versus data access. Setting different levels of access, as mentioned previously, helps to restrict only the necessary data and make sure sensitive data is not exposed. Always avoid giving direct human access to data and add some tools that can generate a read-only report for users to consume in a restrictive manner.

Data encryption at rest

Data at rest means data that is stored somewhere, such as a **storage area network (SAN)**, a **network-attached storage (NAS)** drive, or in cloud storage. All sensitive data must be protected by applying symmetric or asymmetric encryption, explained in this section, with proper key management.

In addition to encryption, there are other methods to protect data at rest, such as masking and tokenization. These methods provide additional layers of security and are particularly useful in situations where sensitive information needs to be used or shared without exposing the actual data.

Data encryption is a way of protecting your data whereby you convert your data from plaintext to encoded ciphertext format using an encryption key. This ciphertext needs to be decrypted using the decryption key before it can be read, and only authorized users will have access to the decryption key.

Commonly used key-based encryption falls into one of two categories of cryptography:

- **Symmetric-key encryption:** With symmetric encryption algorithms, the same key is used to encrypt and decrypt the data. Each data packet is self-encrypted with a secret key. Data is encrypted while saving and decrypted during retrieval. Symmetric encryption used to be applied as per the **Data Encryption Standard (DES)**, which used a 56-bit key. Now, the **Advanced Encryption Standard (AES)** is widely used for symmetric encryption, which is more reliable as it uses a 128-bit, 192-bit, or 256-bit key.
- **Asymmetric-key encryption:** With the help of asymmetric algorithms, two different keys can be used: one to encrypt data and one to decrypt data. In most cases, the encryption key is a public key and the decryption key is a private key. Asymmetric key encryption is also known as **public-key encryption**. The public and private keys are *unidentical* but they are paired together. The private key is only available to one user, while the public key can be distributed across multiple resources. Only the user who has a private key can decrypt the data. **Rivest–Shamir–Adleman (RSA)** is one of the first and most popular public key encryption algorithms used to secure data transmissions over a network.

Data encryption and decryption come with a performance price as they add an additional layer of processing. You need to make a careful trade-off while choosing data for encryption. It might be best to use encryption only where it's necessary to reduce performance and key management overheads.

If you encrypt your data with an AES 256-bit security key, it is almost impossible to break the encryption. The only way to decrypt it is by getting your hands on the encryption key, which means you need to secure your code and keep it in a safe place. Let's learn about some essential management methods to safeguard your encryption key.

Encryption key management

Key management is essential for effective encryption. It ensures that only authorized personnel can access and manage encryption keys. It involves the creation, storage, rotation, and deletion of keys, along with controlling who can access them. Envelope encryption is a specific key management technique used in symmetric encryption where a data key encrypts the plaintext, and a master key encrypts the data key. This method enhances security by requiring two keys for decryption, adding an extra layer of protection.

AWS Key Management Service (KMS) provides envelope encryption functionality. It provides a secure environment where data keys encrypt customer data and master keys from KMS encrypt the data keys. This service provides centralized control over key management, including user access and key rotation.

AWS KMS is a multitenancy key management module. Other cloud vendors provide a similar key management system, such as Cloud Key Management from GCP and Azure Key Vault from Microsoft.

Sometimes, due to regulatory requirements, customers prefer a dedicated key management store that incorporates physical hardware security. They can choose to store their keys in a **hardware security module (HSM)** in this case. Cloud providers such as AWS also provide stores, such as AWS CloudHSM. You can choose your own HSM vendor as well.

An HSM is a specialized device designed to safeguard cryptographic keys and operations, featuring robust physical and logical security mechanisms. Physically, it's engineered to detect and respond to tampering by erasing keys to prevent breaches. Logically, it employs stringent access controls, allowing only authorized users specific roles and interactions with the device. To prevent data loss, it's crucial to ensure the HSM's high availability, typically by deploying multiple units in different locations.

Data encryption in transit

Data in transit means data being transferred over the network. You may encrypt data at rest in the source and destination but your data transfer pipeline needs to be secure when transferring data. When transferring data over an unencrypted protocol such as HTTP, it can be accessed through an attack such as an **eavesdropping attack** or **man-in-the-middle (MITM)** attack.

In an eavesdropping attack, the attacker captures a small packet from a network and uses it to search for any other type of information. An MITM attack is a tampering-based attack where the attacker secretly alters the communication to start communicating on behalf of the receiver. These kinds of attacks can be prevented by transferring data over SSL using a strong protocol such as the **Transport Security Layer (TSL)**.

You will observe that most websites now use HTTPS for communication, which encrypts data using SSL. By default, HTTP traffic is unprotected. All web servers and browsers support SSL/TSL protection for HTTP traffic (HTTPS). HTTP traffic also applies to service-oriented architectures such as **Representational State Transfer (REST)** and **Simple Object Access Protocol (SOAP)**-based architectures.

SSL/TSL handshakes use certificates to exchange a public key using asymmetric encryption and then use the public key to exchange a private key using symmetric encryption. A security certificate is issued by an acceptable **Certification Authority (CA)** such as Verisign. Procured security certificates need to be secured using a **Public Key Infrastructure (PKI)**. Here's an overview of the standard SSL handshake process using RSA key exchange:

1. **Client Hello:** The client initiates the SSL communication by sending a message to the server. This message includes the SSL version number, preferred cipher settings, and data specific to the user's session.
2. **Server Hello:** The server responds to the client, agreeing to communicate using SSL. It verifies the SSL version number and sends its certificate, which contains the public key.
3. **Authentication and pre-master secret generation:** The client authenticates the server's certificate, checking its common name, validity period, and issuing authority. It then generates a pre-master secret based on the chosen cipher and encrypts it with the public key from the server before sending it.

4. **Decryption and master secret creation:** The server uses its private key to decrypt the pre-master secret. Both parties then use this pre-master secret to create a master secret, following steps defined by the selected cipher.
5. **Session key encryption:** Both the server and client send messages indicating that subsequent communications will be encrypted using the session key, also known as the shared secret. They confirm that message encryption and decryption are successful, ensuring that the rest of the session's communication is securely encrypted.

Non-web transmission of data over the network should also be encrypted, including **Secure Shell (SSH)** and **Internet Protocol Security (IPsec)** encryption. SSH is most prevalent while connecting to servers, and IPsec applies to secure corporate traffic transferred over a **virtual private network (VPN)**. File transfer should be secured using **SSH File Transfer Protocol (SFTP)** or **FTP Secure (FTPS)**, and email server communication needs to be secured by **Simple Mail Transfer Protocol Secure (SMTPS)** or **Internet Message Access Protocol (IMAP)**.

In this section, you learned about various ways to secure data at rest and in motion with different cryptographic techniques.

Data backup and recovery is an essential aspect of protecting your data in the case of any unforeseen incidents. You will learn more about data backup in *Chapter 8, Architectural Reliability Considerations*, in the *Disaster recovery planning* section.

Securing APIs

Application Programming Interfaces (APIs) are the connective tissue between different software systems. They facilitate seamless interactions and data transfers. Think of an API as a server in a restaurant; you (the software application) give the waiter (the API) your order (the request), and the waiter brings back the dish (the data/response) from the kitchen (another software system or database). Due to the critical role APIs play in modern software infrastructure, especially in cloud services and microservice architectures, they've become enticing targets for cyber attackers. As such, securing them has never been more crucial. APIs, by their very nature, expose a gateway to potentially sensitive application functions and data. When improperly secured, APIs can lead to various threats, such as unauthorized access to sensitive data, data corruption, denial of service, and sometimes full system compromises. Furthermore, given the interconnected nature of today's software ecosystems, a vulnerability in one API could potentially compromise an entire suite of applications and services. As businesses increasingly rely on APIs to integrate third-party services and enable functionalities like payment gateways, the implications of an API breach can be severe, affecting revenue, brand reputation, and legal standing.

Here are some best practices to ensure API security:

- **Authentication and authorization:** Utilize strong authentication methods such as OAuth or JWTs to confirm the identities of entities attempting to access the API. Additionally, implement effective authorization protocols to manage access rights. This means that even authenticated users can only access the data and functions they are explicitly permitted to. A secure API knows who is making a request and what that entity is allowed to access.

A banking app uses an API to allow users to check account balances. The app uses OAuth to ensure only authenticated and authorized users can view their specific account details.

- **Rate limiting:** Implement rate limiting to prevent any form of abuse, including brute-force attacks. By limiting the number of requests a user or IP can make within a set timeframe, you can thwart potential attackers from overwhelming your system or trying numerous combinations in quick succession. An online store's API could prevent users from making more than ten requests per minute to prevent system overloads and potential misuse.
- **Input validation:** Always validate and sanitize data sent to your API. This can prevent various forms of attacks, including SQLi, where attackers send malicious data in the hopes of manipulating your systems. An online feedback form uses an API to submit user comments. The system checks to ensure that entries don't contain malicious scripts that could compromise the website.
- **Encryption:** Data in transit to and from your API should be encrypted using protocols such as TLS. This ensures that even if data packets are intercepted, they remain unintelligible to unauthorized parties. A messaging app ensures that the messages sent between users are encrypted. If someone intercepts the messages, they'll only see jumbled characters rather than the actual content. Think of it as speaking in a coded language. Even if someone overhears your conversation, they will only understand it if they know the code.
- **Regularly monitor and audit:** Continuously monitor API activity. Any unusual patterns, such as unexpected spikes in requests or anomalous data access patterns, can be early signs of an attack or vulnerability exploitation. Regular audits can also help identify any lingering security misconfigurations. A cloud storage provider monitors its APIs for unusual data transfer patterns, ensuring that large amounts of data aren't being unexpectedly downloaded or uploaded, which could indicate a security breach. Consider security cameras in stores. They monitor activity and can catch and deter theft.
- **Implement API gateways:** Using API gateways can act as an added layer of protection. They handle request routing, API composition, and other functions, ensuring only legitimate requests reach your back-end systems. An e-commerce site uses an API gateway to manage requests, ensuring that only legitimate and well-structured requests reach its database, filtering out potentially harmful ones. Consider a hotel concierge who checks and confirms your reservation before allowing you to proceed to your room.
- **Error handling:** Avoid exposing sensitive information through error messages. A generic error message should be returned to the user, while detailed error logs should be maintained securely on the server side for diagnostic purposes. When a user tries to reset a password and their email isn't recognized, the system doesn't specify if the email is wrong or doesn't exist. It simply prompts them to try again, preventing potential data phishing.
- **Use web application firewalls:** WAFs can detect and block malicious requests to your API, providing another layer of defense against common web-based threats. For example, if you're running an e-commerce platform, the WAF can scrutinize incoming traffic to your API endpoints, identifying and thwarting potentially harmful requests such as SQLi and XSS attacks. This ensures that only legitimate requests are processed, safeguarding your application from cyber threats.

- **Versioning:** Implement versioning in your APIs. If a security issue is detected in one API version, it can be addressed without affecting other versions, ensuring continuity of service for applications using the unaffected versions. For example, suppose you have a mobile application that relies on an API for fetching user data. By implementing versioning (e.g., v1, v2, v3), if a security vulnerability is discovered in v2, you can quickly address the issue in that version while older (v1) and newer (v3) versions continue to operate securely and without disruption. This approach allows your development team to patch or upgrade specific versions of the API, minimizing the impact on end users.
- **Regular security testing:** Periodically subject your APIs to penetration testing and vulnerability assessments. This proactive approach can help identify and rectify potential weaknesses before they are exploited. A music streaming platform periodically tests its API to ensure that unauthorized users can't access premium features without a valid subscription.

As APIs continue to be the linchpin of modern digital infrastructure, the necessity to secure them escalates. By adhering to industry best practices and maintaining a proactive security posture, businesses can safeguard their operations, customers, and reputation from any threats posed.

There are many governing bodies that publish compliance about data security, which can take the form of a set of checklists to adhere to. Compliance also makes sure that organizations comply with industry and local government rules. Let's learn more about various compliance measures in the next section.

Security and compliance certifications

Many compliance certifications to protect customer privacy and secure data depend on your industry and your geographical location. For any solution design, compliance requirements are among the critical criteria that need to be evaluated. The following are some of the most widely known geographical and industry-standard compliances:

- Global compliance includes certifications that all organizations must adhere to, regardless of their region. These include ISO 9001, ISO 27001, ISO 27017, ISO 27018, SOC 1, SOC 2, SOC 3, and CSA STAR for cloud security.
- The US government requires various kinds of compliance to handle the public sector workload. These include FedRAMP, DoD SRG Level-2, 4, and 5, FIPS 140, NIST SP 800, IRS 1075, ITAR, VPAT, and CJIS.
- Industry-level compliance of an application applies to particular industries. These include PCI DSS, CDSA, MPAA, FERPA, CMS MARS-E, NHS IG Toolkit (in the UK), HIPAA, FDA, FISC (in Japan), FACT (in the UK), Shared Assessment, and GLBA.
- Regional compliance certification applies to a particular country or region. These include the EU GDPR, EU Model Clauses, UK G-Cloud, China DJCP, Singapore MTCS, Argentina PDPA, Australia IRAP, India MeitY, New Zealand GCIO, Japan CS Mark Gold, Spain ENS and DPA, Canada Privacy Law, and US Privacy Shield.

As you can see, there are many compliance certifications available from different regulatory bodies that relate to industry, region, and government policy. We are not going to go into compliance details but you must evaluate your application with compliance requirements before starting your solution design as compliance requirements influence the overall solution design heavily. You need to decide what kind of encryption is required, as well as the logging, auditing, and location of your workload based on your compliance needs.



Logging and monitoring help to ensure robust security and compliance and are essential. If an incident occurs, your team should be notified immediately and should be ready to respond. You will learn more about monitoring and alert methods in *Chapter 9, Operational Excellence Considerations*.

Several compliance industries depend on your application geolocation, industry, and government rules. You have learned about the various categories of compliance and some common compliance standards appropriate for each group. Many organizations are moving to the cloud, so it's vital to understand security in the cloud.

The cloud's shared security responsibility model

As the cloud is becoming the norm and many organizations are moving their workload to a public cloud such as AWS, GCP, and Azure, the customer needs to understand the cloud security model.

Security in the cloud is a joint effort between the customer and the cloud provider.

Customers are responsible for what they implement using cloud services and the applications connected to the cloud. In the cloud, customer responsibility for application security depends upon the cloud provider they are using and the complexity of their system.

The following diagram illustrates a cloud security model from one of the largest public cloud providers (AWS), and it's pretty much applicable to any public cloud provider, such as Azure, GCP, Oracle, IBM, and Alibaba:

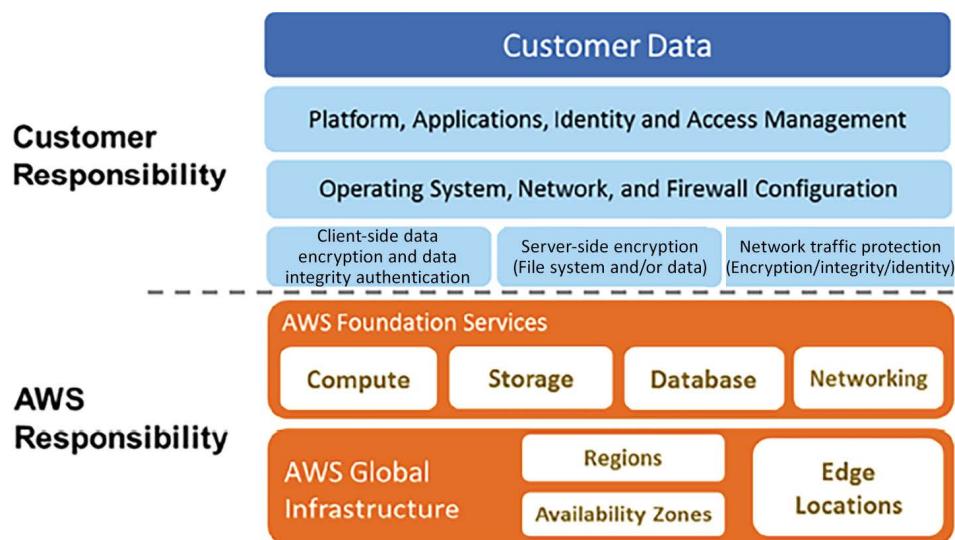


Figure 7.11: AWS Cloud shared security responsibility model

The customer handles the **security in the cloud**, which includes the following:

- **Server's operating system:** The operating system installed on the server could be vulnerable to attacks. Patching and maintenance of the operating system is the customer's responsibility, as software applications depend heavily upon it.
- **Application:** Every application and its environments, such as dev, test, and prod, are maintained by the customer. So, handling password policies and access management is the customer's responsibility.
- **Operating system/host-based firewalls:** Customers must protect their entire system from external attacks. The cloud provides security in that area, but customers should consider an IDS or an IPS to add an extra layer of security.
- **Network configuration and security group:** The cloud provides tools to create a network firewall, but which traffic needs to be stopped or allowed to go through depends on the application requirements. Customers are responsible for setting up firewall rules to secure their systems from external and internal network traffic.
- **Customer data and encryption:** Data handling is the customer's responsibility, as they are more aware of the level of data protection needed. The cloud provides tools for data protection by using various encryption mechanisms but it's the customer's responsibility to apply those tools and secure their data.

As shown in *Figure 7.11*, AWS and other public cloud providers are responsible for **securing the cloud**, specifically the physical infrastructure that hosts your resources. This security encompasses several key areas:

- **Data Centers:** AWS data centers are nondescript facilities that employ round-the-clock security guards. They implement strict access controls, including two-factor authentication, comprehensive access logging and regular review, and video surveillance. Additionally, AWS ensures the secure disposal of data storage devices through methods like disk degaussing and destruction.
- **Hardware Infrastructure:** This includes servers, storage devices, and various other appliances that underpin AWS services. AWS ensures the security and integrity of this hardware.
- **Software Infrastructure:** This refers to the host operating systems, service applications, and virtualization software used in AWS services. AWS maintains the security of this software layer, ensuring it is resilient to threats.
- **Network Infrastructure:** AWS secures its network infrastructure, which comprises routers, switches, load balancers, firewalls, cabling, and more. Part of this security measure involves continuous monitoring of the network's external boundaries. AWS also maintains secure access points and redundant network infrastructure to prevent disruptions and enhance security.

To make your application with industry regulations like PCI-DSS for financial data security and GDPR for data protection in Europe, you need to handle and complete audits for application-level complaints. The public cloud provides various compliance certifications that apply to the hardware portions they manage. As a customer, you get an additional advantage by inheriting security and compliance provided by the cloud provider.

The cloud provides all kinds of tools and services to secure your application in the cloud, along with built-in security at the IT infrastructure level. However, it's up to the customer how they want to utilize those services and make their application secure in the cloud. The cloud offers enhanced visibility and centralized control over your IT inventory, aiding in the effective management and security of your systems.

Security is the priority for any solution, and a solutions architect must ensure their application is secure and protected from any attack. Aim to incorporate automated security best practices as much as possible. Utilizing software-based security mechanisms can significantly enhance your scalability, cost-effectiveness, and overall security. Start by creating a custom baseline image of a virtual server, which encapsulates your security standards. This image can then be used consistently for every new server you deploy, ensuring uniform security across your infrastructure. Additionally, design your entire infrastructure within a template that defines and manages it. This approach allows you to replicate your established security best practices in every new environment you create.

Security is a continuous effort. Each security incident should be treated as an improvement opportunity for the application. A robust security mechanism should have authentication and authorization controls. Organizations and applications should automate responses to security events and protect infrastructure at multiple levels.

Securing threat modeling

Threat modeling is a structured way to identify, assess, and prioritize potential threats to your application. By understanding the potential threats, you can design and implement appropriate countermeasures to prevent, detect, or mitigate the impact of these threats. Threat modeling is often used in software development but can also be applied to other areas, such as infrastructure and operations.

The following are the components of threat modeling:

- **System representation:** Before analyzing threats, you need a clear understanding of the system. This often involves creating diagrams or models of the system's architecture, components, data flows, and potential entry points. For a simple online e-commerce website, you might have a front-end for users, a back-end server handling requests, a database storing user credentials, and an external payment gateway for transactions. Before launching a new feature that allows users to save multiple delivery addresses, the development team wants to ensure there are no security loopholes. They create a data flow diagram showing how this new feature interacts with the existing system.
- **Threat identification:** Enumerate potential threats based on the system's representation. This can involve using techniques like **Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege (STRIDE)** and attack trees. For our e-commerce site, threats could include SQL injection (to access the database), phishing attacks (to steal user credentials), or DoS attacks (to bring the site down). The team realizes that allowing users to save addresses might expose those addresses if there's a data breach. They list this threat alongside others.
- **Threat analysis:** Evaluate the potential impact and likelihood of each identified threat. This can help you to prioritize threats. While SQLi could expose a lot of sensitive user data, a phishing attack might target fewer users but is more likely to occur. The development team assesses that a data breach exposing saved addresses could lead to a loss of user trust and potential misuse. They give this threat a high severity rating.
- **Mitigation strategy:** For each identified threat, determine the best course of action to reduce its risk. This can involve adding security controls, changing system architecture, or even accepting the risk if its potential impact is acceptable. To prevent SQL injection, the team could use parameterized queries. To combat phishing, they might introduce two-factor authentication. To safeguard user addresses, the team decides to encrypt them in the database. They also add alerts for any suspicious activity linked to address changes.
- **Documentation:** Document the findings, including potential threats, their severity, and the chosen mitigation strategies. A document is created detailing that user addresses are encrypted and the encryption method used. Six months later, there's a requirement to shift to a different database. The documentation helps the new database team understand the security measures in place.

- **Review and update:** Threat modeling is not a one-time task. As systems evolve, new threats may emerge, while others may become less relevant. Regularly reviewing and updating the threat model ensures that it remains relevant. The e-commerce site decided to introduce a new chatbot feature. Before deploying the chatbot, the team refers to their threat model to see if introducing this new feature brings any new vulnerabilities or if existing threats have evolved.

In essence, threat modeling helps teams be proactive in their security efforts, addressing potential vulnerabilities before they become issues.

Summary

In this chapter, you learned about various design principles to apply security best practices for your solution design. These principles include key considerations to protect your application by using appropriate access control, data protection, and monitoring.

You need to apply security at every layer. Starting with user authentication and authorization, you learned about applying security at the web layer, application layer, infrastructure layer, and database layer. Each layer is vulnerable to different attacks, and you learned various methods to protect your application with the available technology choices.

For user management, you learned how to use FIM and SSO to handle corporate users and various methods of implementing user authentication and authorization. These include enterprise management services like Microsoft AD and AWS Directory Service. You can also handle millions of users using OAuth 2.0.

At the web layer, you learned about various attack types, such as **DDoS**, **SQLi**, and **XSS**. You learned how to protect against those attacks using different prevention techniques and network firewalls. You learned various techniques to protect code at the application layer and ensure the security of your infrastructure. You dove deep into different network components and methods to build trusted boundaries to limit the attack radius.

You learned about data protection by putting proper data classification in place and tagging your data as confidential, private, or public. You learned about symmetric and asymmetric algorithms and how they differ. You looked at using key management to protect the public/private encryption key. Data can be in motion or sitting in storage, you learned about how to protect data in both modes. You explored API security and best practices to ensure all APIs exposing your applications are secure. You discovered various compliance and shared security responsibility models that are applicable to a cloud workload. In the end, you learned about building threat models.

While this chapter was about applying security best practices, reliability is another essential aspect of any solution design. To make your business successful, you need to create a reliable solution that is always available and can handle workload fluctuation. In the next chapter, you will learn the best practices to make your application reliable with the available technologies. You will learn about various disaster recovery and data replication strategies to make your application more reliable.

Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below to get a free eBook of your choice.



8

Architectural Reliability Considerations

Application reliability is an essential aspect of architecture design and vital for any business to be successful.

Reliability means the ability of the system to recover from failure. It's about making your application fault-tolerant and able to recover from any infrastructure or server failure without impacting the customer experience. Your system should be prepared to handle any situation that could cause disruption.

As all kinds of businesses are now online, high availability has also become a mandatory criterion for online applications. Users want to browse your application anytime and complete tasks such as shopping and banking at their convenience. In this chapter, you will learn various design principles to make your solution reliable. When assessing reliability, you need to consider every component of the architecture. You will understand how to choose the right technology to ensure your architecture's reliability at every layer.

You will learn the following best practices for reliability in this chapter:

- Design principles for architectural reliability
- Technology selection for architectural reliability
- Improving reliability with the cloud

By the end of this chapter, you will have learned about various disaster recovery techniques and data replication methods to ensure the high availability of your application and the continuation of business processes.

Design principles for architectural reliability

Reliability and **high availability (HA)** are foundational pillars for ensuring that applications and infrastructure can meet user demands without interruption. Reliability focuses on the system's ability to operate correctly under specific conditions and for a particular period.

It involves designing systems to contain and manage failures in the smallest scope possible, minimizing the impact on overall operations. This approach requires a comprehensive understanding of potential failure modes and implementing targeted mitigation strategies to either prevent these failures or recover gracefully from them.

HA, discussed in detail in *Chapter 2*, is closely related to reliability but with an emphasis on ensuring that services remain accessible at all times. HA strategies involve creating redundant systems and components to eliminate single points of failure, thereby allowing for seamless failover in case of an outage. The goal is to maintain service continuity even in the face of hardware failures, network disruptions, or software bugs. By integrating reliability and HA into the system design, organizations can ensure that their applications are robust against failures and capable of maintaining consistent service levels.

In this discussion are the standard design principles that help you to strengthen your system's reliability. You will find that all reliability design principles are closely related and complement each other.

Making systems self-healing by applying automation

Integrating self-healing capabilities and automation into your system's design enhances its reliability by allowing it to predict and recover from failures autonomously. A self-healing system proactively detects and rectifies failures across various system layers—be it hardware, network, or software—minimizing the impact on end users. This approach requires identifying critical key performance indicators (KPIs) relevant to your application and business operations, such as request handling capacity per second or web page load times. Infrastructure-level KPIs might include thresholds for CPU and memory utilization, ensuring they do not exceed predefined limits.

To achieve a self-healing architecture, implement a robust monitoring system that tracks these KPIs and alerts you as they approach critical thresholds. This system should be backed by automation strategies that can, for example, spin up additional servers to manage increased load when CPU utilization nears its maximum allowable percentage. This level of proactive monitoring and automated response not only prevents potential failures but also supports the system in maintaining optimal performance levels without manual intervention.

Moreover, embracing automation across your application's life cycle—from deployment and configuration to scaling the infrastructure—fosters a more agile and resilient environment. It enables your team to deploy new features rapidly, experiment more freely, and ensure consistent system performance regardless of fluctuating workloads. Automating the scaling of resources based on scheduled demands or unexpected spikes in traffic ensures that the application remains responsive and available. By leveraging automation for deploying independent jobs and aggregating their outcomes, you can achieve not only higher reliability and efficiency but also enhance the system's ability to self-recover from incidents, making your infrastructure truly resilient and self-sustaining.

Quality assurance

Frequently, you need to apply the same configuration that you have on your development environment to **quality assurance (QA)** environments. There may be multiple QA environments for each testing stage, which includes functional testing, **user acceptance testing (UAT)**, and stress testing environments.

Often, a QA tester discovers a defect caused by wrongly configured resources, which could introduce a further delay in the test schedule. Most importantly, you cannot afford a configuration error in production servers, which can cause widespread outages, so better to test them in advance.

To reproduce precisely the same configuration as your development environment to your QA environment, you may need to document step-by-step configuration instructions. Repeating the same steps to configure each environment manually can be error-prone. There is always a chance of human error, such as a typo, for example, in a database name. The solution to this challenge is to automate these steps by creating a script. The automation script itself can be the documentation.

As long as the script is correct, it is more reliable than manual configuration. It is undoubtedly reproducible. Detecting unhealthy resources and launching replacement resources can be automated, and you can notify the IT operations team when resources are changed. Automation is a fundamental design principle that needs to be applied everywhere in your system.

Creating a distributed system

Monolithic applications have low reliability regarding system uptime, as a tiny issue in a particular module can bring down the entire system. Dividing your application into multiple small services reduces the *impact area*. One part of the application shouldn't impact the whole system, and the application can continue to serve critical functionality. For example, in an e-commerce website, an issue with the payment service should not affect the customer's ability to place orders, as payment can be processed later.

At the service level, scale your application horizontally to increase system availability. Design a system to use multiple smaller components working together rather than a single monolithic system to reduce the impact area. In a distributed design, requests are handled by different system components, and the failure of one component doesn't impact the functioning of other parts of the system. For example, on an e-commerce website, the failure of warehouse management components will not impact the customer placing the order.

However, the communication mechanism can be complicated in a distributed system. This complexity arises from the need to ensure reliable data exchange across various networked computers, each possibly running different operating systems and located in different geographical areas. Challenges include dealing with network latency, handling message delivery guarantees, synchronizing data across nodes to ensure consistency, and implementing fault tolerance to withstand partial system failures. Additionally, the development and maintenance of a communication protocol that efficiently supports the diverse requirements of a distributed architecture add to the intricacy.

The circuit breaker pattern can help here to handle system dependencies. As you learned regarding the circuit breaker pattern in *Chapter 4, Solution Architecture Design Patterns*, the basic idea is simple. You wrap a protected function call in a circuit breaker object, which monitors for failures and takes automated action to mitigate them.

Monitoring and adding capacity

Resource saturation is the most common reason for application failure. Often, you will encounter the issue where your applications start rejecting requests due to CPU, memory, or hard disk overload.

In a traditional on-premises environment, you must calculate server capacity based on the assumption in advance. Online traffic is very unpredictable and fluctuates heavily, driven by global trends. Usually, procuring hardware can take anywhere between 3 to 6 months, and it's tough to guess capacity in advance. Ordering excess hardware will incur an extra cost as a resource is sitting idle, and a lack of resources will cause the loss of business due to application unreliability.

You need an environment where you don't need to guess capacity, and your application can scale on demand.

A public cloud provider such as **Amazon Web Services (AWS)** provides **infrastructure as a service (IaaS)**, facilitating the on-demand availability of resources.

In the cloud, you can monitor system supply and demand. You can automate the addition or removal of resources as needed. It allows you to maintain the level of resources that will satisfy demand without over-provisioning or under-provisioning.

Performing recovery validation

When it comes to infrastructure validation, most of the time, organizations focus on validating a happy path where everything is working. Instead, you should validate how your system fails and how well your recovery procedures work. Validate your application, assuming everything fails all the time. Don't just expect that your recovery and failover strategies will work. Make sure to test them regularly so you're not surprised if something goes wrong.

A simulation-based validation helps you to uncover any potential risks. You can automate a possible scenario that could cause your system to fail and prepare an incident response accordingly. Your validation should improve application reliability so that nothing will fail in production.

Recoverability is sometimes overlooked as a component of availability. To improve the system's **recovery point objective (RPO)** and **recovery time objective (RTO)**, you should back up data and applications and their configuration as a machine image. You will learn more about RTO and RPO in the next section. Suppose a natural disaster makes one or more components unavailable or destroys your primary data source. In that case, you should be able to restore the service quickly and without losing data. Let's discuss specific disaster recovery strategies to improve application reliability and associated technology choices next.

Technology selection for architectural reliability

Application reliability often looks at the availability of the application to serve users. Several factors go into making your application highly available. However, **fault tolerance** refers to the built-in redundancy of an application's components. Your application may be highly available but not be 100% fault-tolerant. For example, if your application needs four servers to handle the user request, you divide them between two data centers for HA.

If one site goes down, your system is still highly available at 50% capacity, but it may impact user performance expectations. However, if you create equal redundancy in both sites with four servers each, your application will not only be highly available but will also be 100% fault-tolerant.

Suppose your application is not 100% fault-tolerant. In that case, you want to add automated scalability, defining how your application's infrastructure will respond to increased capacity needs to ensure your application is available and performing within your required standards. To make your application reliable, you should be able to restore services quickly and without losing data. Going forward, we will address this recovery process as **disaster recovery (DR)**. Before going into various DR scenarios, let's learn more about the RTO/RPO and data replication as it is a key factor to measure when planning for DR.

Planning the RPO and RTO

Business applications need to define service availability in the form of a **service-level agreement (SLA)**. Organizations define SLAs to ensure application availability and reliability for their users. For example, you might say in an SLA that your application should be 99.9% available in a given year, or that the organization can tolerate downtime of 43 minutes per month, and so on. The defined SLA primarily drives the RPO and RTO for an application.

The RPO is the amount of data loss an organization can tolerate in a given period. For example, my application is acceptable if it loses 15 minutes' worth of data. In this case, if you are processing customer orders every 15 minutes for fulfillment, you can tolerate reprocessing that data in case of any system failure at the order fulfillment application. The RPO helps to define a data backup strategy.

The RTO is about application downtime and how long the application should take to recover and function normally after an incidence of failure. The following diagram illustrates the difference between the RTO and RPO:

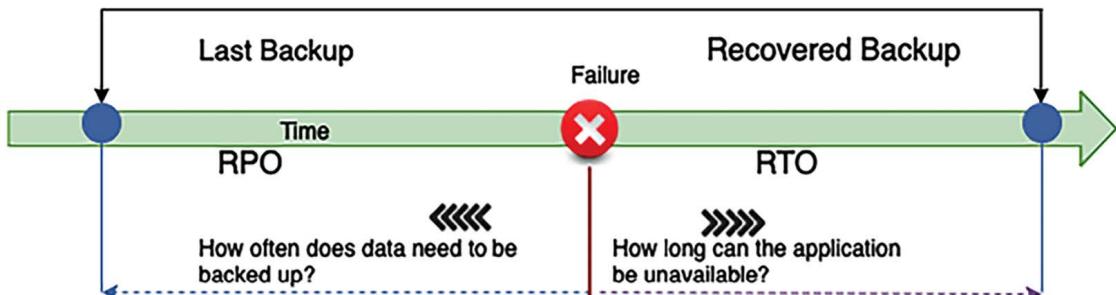


Figure 8.1: RTO and RPO

In the preceding diagram, suppose the failure occurs at 10 A.M. and you took the last backup at 9 A.M.; in the event of a system crash, you would lose 1 hour of data. When you restore your system, there is an hour's worth of data loss, as you have been taking data backups every hour.

In this case, your system RPO is 1 hour, as it can tolerate living with an hour's worth of data loss. In this case, the RPO indicates that the maximum data loss that can be tolerated is 1 hour.

If your system takes 30 minutes to restore to the backup and bring up the system, it defines your RTO as half an hour. This means the maximum downtime that can be tolerated is 30 minutes. The RTO is the time it takes to restore the entire system after a failure that causes downtime, which is 30 minutes in this case.

An organization typically decides on an acceptable RPO and RTO based on the user experience and financial or reputational impact on the business in the case of system unavailability. Organizations plan solutions for effective system recovery per the defined RTO and RPO. Over time, you should aim to reduce the RTO/RPO, which will directly result in business benefits as the application will have more uptime. You can see now how data is the key to system recovery, so let's learn some methods to minimize data loss.

Replicating data

Data replication and snapshots are the keys to DR and making your system reliable. Replication creates a copy of the primary data site on the secondary site. In the event of primary system failure, the system can fail over to the secondary system and keep working reliably. This could be your file data stored in a **NAS drive**, **database snapshot**, or **machine image snapshot**. Sites could be two geo-separated, on-premises systems, two separate devices on the same premises, or a physically separated public cloud.

Data replication is not only helpful for disaster recovery but it can also speed up an organization's agility by quickly creating a new environment for testing and development. Data replication can be synchronous or asynchronous.

Synchronous versus asynchronous replication

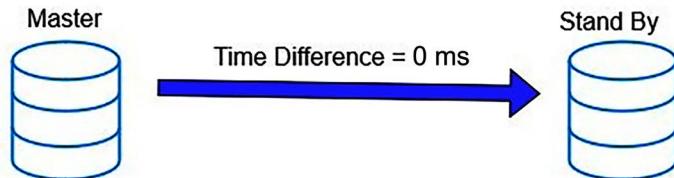
Synchronous replication creates a data copy in real time. Real-time data replication helps to reduce the RPO and increase reliability in the event of a disaster. However, it is expensive as it requires additional resources in the primary system for continuous data replication.

Asynchronous replication creates copies of data with some lag or as per the defined schedule. However, asynchronous replication is less expensive as it uses fewer resources than synchronous replication. You may choose asynchronous replication if your system can work with a longer RPO.

Regarding database technology such as Amazon RDS, synchronous replication is applied if we create an RDS with multiple **Availability Zone (AZ)** failover. This setup ensures that your primary database and its replica in another AZ are always in sync, providing HA and data durability. If the primary database encounters issues, the service can automatically fail over to the replica with minimal disruption. For read replicas, there is asynchronous replication, and you can use that to serve reports and read requests.

As illustrated in the following architecture diagram, in synchronous replication, there is no lag of data replication between the master and standby instance of the database, while, in the case of asynchronous replication, there could be some lag while replicating the data between the primary and replication instance:

Synchronous Data Replication



Asynchronous Data Replication

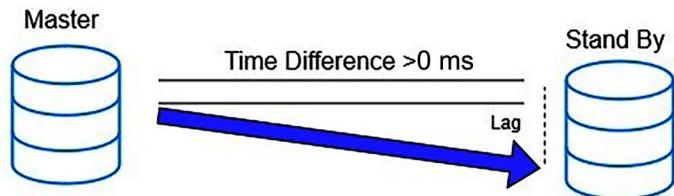


Figure 8.2: Synchronous and asynchronous data replication

Let's explore some data replication methods for the synchronous and asynchronous approaches.

Replication methods

The replication method is an approach to extract data from the source system and create a copy for data recovery purposes. Different replication methods are available to store a copy of data per the storage type for business process continuation. Replications can be implemented in the following ways:

- **Array-based replication:** In this, built-in software automatically replicates data. However, the source and destination storage arrays should be compatible and homogeneous to replicate data. A storage array contains multiple storage disks in a rack.
Large enterprises use array-based replication due to the ease of deployment and reduced compute power of the host system. You can choose array-based replication products such as HP Storage, EMC SAN Copy, and NetApp SnapMirror.
- **Network-based replication:** This can copy data between different kinds of heterogeneous storage arrays. It uses an additional switch or appliance between incompatible storage arrays to replicate data. In network-based replication, the cost of replication could be higher as multiple players come into the picture. You can choose from networked-based replication products such as NetApp Replication X and EMC RecoverPoint.
- **Host-based replication:** In this, you install a software agent on your host that can replicate data to any storage system, such as NAS, SAN, or DAS. You can use a host-based software vendor, for example, Symantec, Commvault, CA, or Vision Solution.

This is prevalent in **small and medium-sized businesses (SMBs)** due to lower upfront costs and heterogeneous device compatibility. However, it consumes more computing power as the agent needs to be installed on the host operating system.

- **Hypervisor-based replication:** This is VM-aware, which means copying the entire virtual machine from one host to another. As organizations mostly use virtual machines, it provides a very efficient DR approach to reduce the RTO. Hypervisor-based replication is highly scalable and consumes fewer resources than host-based replication. It can be carried out by native systems built into VMware and Microsoft Windows. You can choose a product such as Zerto to perform hypervisor-based replication or another product from various vendors.

Previously, in *Chapter 2, Principles of Solution Architecture Design*, you learned about scalability and fault tolerance. In *Chapter 4, Solution Architecture Design Patterns*, you learned about various design patterns to make your architecture highly available. Now, you will discover multiple ways to recover your system from failure and make it highly reliable.

Planning disaster recovery

DR is about maintaining business continuation during system failure. It's about preparing the organization for any possible system downtime and the ability to recover from it. DR planning covers multiple dimensions, including hardware and software failure.



While planning for DR, always ensure you consider operational losses such as power outages, network outages, heating and cooling system failures, physical security breaches, and other incidents, such as fires, floods, or human error.

Organizations invest effort and money in DR planning as per system criticality and impact. A revenue-generating application needs to be up all the time as it significantly impacts company image and profitability. For such an application, the organization will invest lots of effort in creating their infrastructure and training their employees for a DR situation. DR is like an insurance policy that you have to invest in and maintain even when you don't utilize it as, in the case of unforeseen events, a DR plan will be a lifesaver for your business.

Bases of business criticality, such as software applications, can be placed on a *spectrum of complexity*. There are four DR scenarios, sorted from highest to lowest RTO/RPO as follows:

- Backup and restore
- Pilot light
- Warm standby
- Multi-site

As shown in the following diagram, in DR planning, as you progress with each option, your RTO and RPO will reduce while the cost of implementation increases. You need to make a suitable trade-off between RTO/RPO requirements and cost per your application reliability requirements:

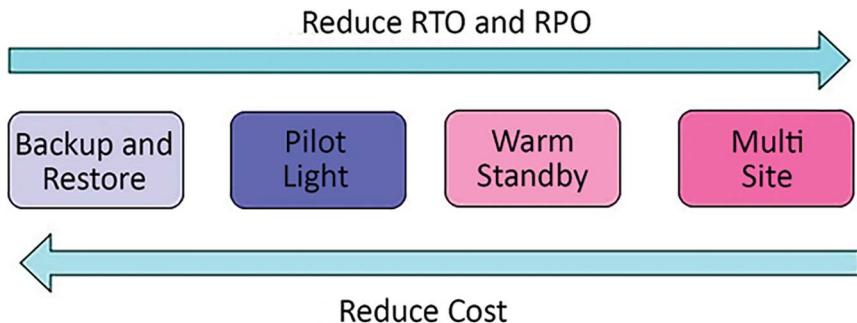


Figure 8.3: The spectrum of DR options

DR strategies are highly customized to each organization's unique needs, with the decision to execute a complete site failover depending on various critical factors. The triggering point for such drastic action varies, ranging from minor disruptions to major catastrophes like data center destruction. For instance, in the event of a significant disaster, an organization may need to quickly assess and prioritize critical services—often those accounting for a substantial portion of its revenue. These priority services may have a predefined RTO, such as a 24-hour window to resume operations before the financial impact becomes too severe, considering potential losses like fines, SLA breaches, and decreased sales. On the other hand, for less catastrophic but still critical service disruptions, a company might set automatic failover protocols for a much shorter downtime tolerance, such as 15 minutes. In both scenarios, the decision criteria for DR involve assessing the business impact analysis, understanding the RTO and RPO for essential services, evaluating the cost of downtime against the recovery process, and ensuring compliance with any regulatory requirements. Additionally, technical feasibility, including the availability and readiness of a secondary site, plays a crucial role in determining the appropriate response to ensure continuity and minimize operational disruptions.

Let's explore each option mentioned above in detail with the technology choices involved. Note that public clouds such as AWS enable you to operate each of the preceding DR strategies cost-effectively and efficiently.

Backup and restore

Backup and restore is the lowest cost option resulting in a higher RPO and RTO. This method is simple to start and highly cost-effective as you only need backup storage space. This backup storage could be a tape drive, hard disk drive, or network access drive. As your storage needs increase, adding and maintaining more hardware across regions could be a daunting task. One of the most cost-effective and straightforward options is to use the cloud as backup storage. Amazon S3 is one example; it provides unlimited storage capacity at a low cost and with a pay-as-you-go model.

The following diagram shows a basic DR system. In this diagram, the data is in a traditional data center, with backups stored in AWS. AWS Import/Export or Snowball kinds of physical hard drives in the size range of 8 TB to 100 TB are used to get the data into AWS, and the information is later stored in Amazon S3:

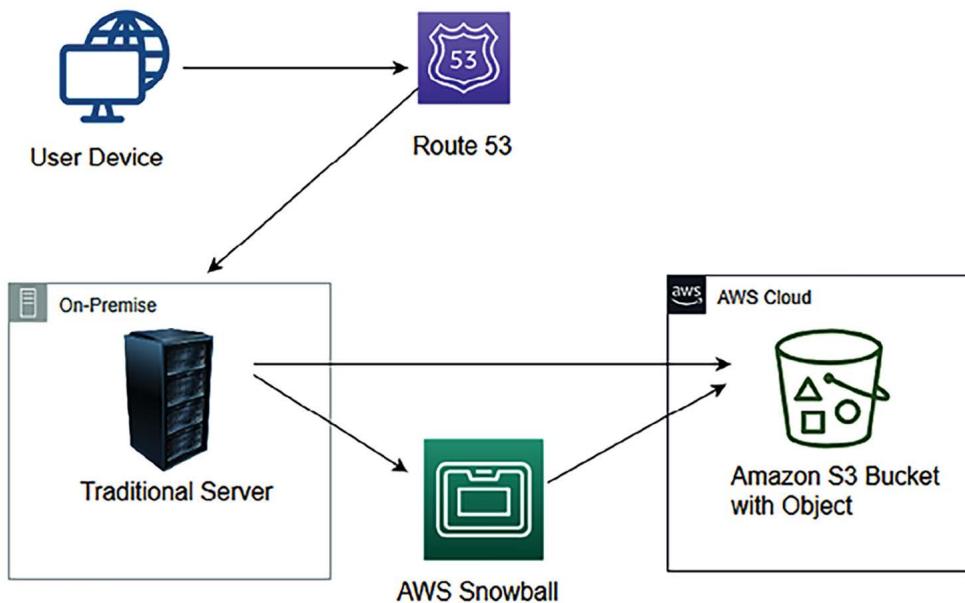


Figure 8.4: Data backup to Amazon S3 from on-premise infrastructure

You can use other third-party solutions available for backup and recovery. Some of the most popular choices are NetApp, VMware, Tivoli, and Commvault.

When planning for DR in a cloud environment, it's crucial to incorporate strategies that leverage the strengths of various cloud providers, such as AWS, Google Cloud Platform (GCP), and Microsoft Azure. This approach ensures flexibility and resilience across different platforms. Here's a generalized procedure applicable across these cloud services:

- **Backup and storage solutions:** Utilize cloud storage services to keep backups of your systems. For AWS, Amazon S3 can serve as a reliable backup storage solution. In GCP, Google Cloud Storage offers durable and highly available object storage. Azure's equivalent, Azure Blob Storage, provides a similar service for storing large amounts of unstructured data.
- **Machine images and configuration:** Prepare machine images that include your operating system, applications, and configurations. AWS uses **Amazon Machine Images (AMIs)**, GCP utilizes Compute Engine images, and Azure offers Azure Virtual Machine images. These images can be customized and pre-configured with necessary software and security patches, ready to be deployed in case of a disaster.

- **System restoration documentation:** Clearly document the steps required to restore your system from the backups across different cloud platforms. This documentation should include how to deploy the stored machine images and how to restore databases and applications from backups.
- **Traffic routing and failover procedures:** Outline the process to reroute traffic from the primary site to the backup site in the cloud. AWS offers Route 53 for DNS management and traffic routing, GCP has Cloud DNS and Traffic Director, and Azure provides Azure Traffic Manager and DNS Zone. Understanding how to utilize these services for failover scenarios is essential.
- **Deployment runbook:** Develop a comprehensive runbook that details deployment configurations and potential issues along with their resolutions. This runbook should be cloud-agnostic and adaptable to the specifics of AWS, GCP, and Azure, ensuring teams are prepared regardless of the cloud platform used.

In the preparation phase, create and store custom machine images and backups in the respective cloud storage solutions—S3 for AWS, Cloud Storage for GCP, and Blob Storage for Azure. Also, prepare snapshots of databases, storage volumes, and any essential files. This proactive approach ensures that, regardless of the cloud provider, your organization can quickly recover from a disaster with minimal downtime and data loss.

This DR pattern is easy to set up and relatively inexpensive. However, in this scenario, both the RPO and RTO will be high; the RTO will be the downtime until the system gets restored from the backup and starts functioning, while the RPO that is lost depends on the system's backup frequency. Let's explore the next approach, pilot light, which improves your RTOs and RPOs further.

Pilot light

The pilot light approach is the next lowest-cost DR method after backup and restore. Like the pilot light in your gas heater, a tiny flame that is always on that can quickly light the entire furnace to heat the house, with this DR method, you need to keep the minimum number of core services up and running in different regions. You can spin up additional resources quickly in the event of a disaster.

You could actively replicate the database tier, then spin up instances from a virtual machine image or build out infrastructure using infrastructure as code, such as CloudFormation, Terraform, Ansible, and so on.

The following diagram shows a pilot light DR pattern. In this case, the database is replicated into AWS, with Amazon EC2 instances of the web servers and application servers ready to go but not currently running:

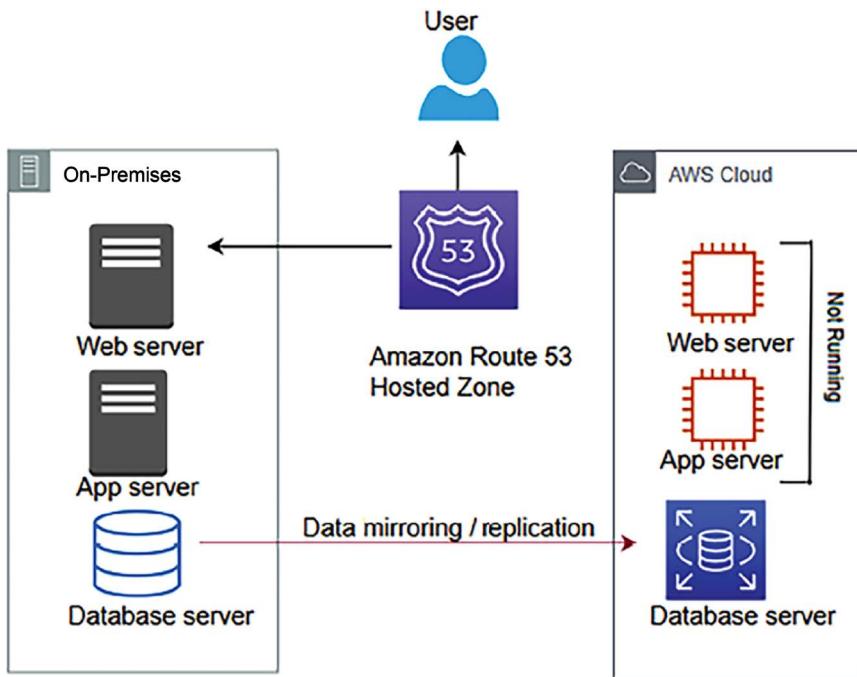


Figure 8.5: The pilot light data replication to DR site scenario

A pilot light scenario is similar to backup and restore, where you take a backup of most of the components and store them passively. However, you maintain active instances with a lower capacity for critical components such as a database or authentication server, which can take a significant time to spin up. You need to automatically start all required resources, including network settings, load balancers, and virtual machine images, as needed. As the core pieces are already running, recovery time is faster than the backup and restore method. The pilot light method is very cost-effective as you are running only some of the resources at full capacity.

You need to enable the replication of all critical data to the DR site—in this case, the AWS cloud. You can use the AWS Data Migration Service to replicate data between on-premises and cloud databases. For file-based data, you can use Amazon File Gateway.



Many other third-party-managed tools provide data replication solutions efficiently, such as Attunity, Quest, Syncsort, Alooma, and JumpMind.

If the primary system fails, as shown in the following diagram, you can start up the Amazon EC2 instances with the latest copy of the data. Then, you redirect Amazon Route 53 to point to the new web server:

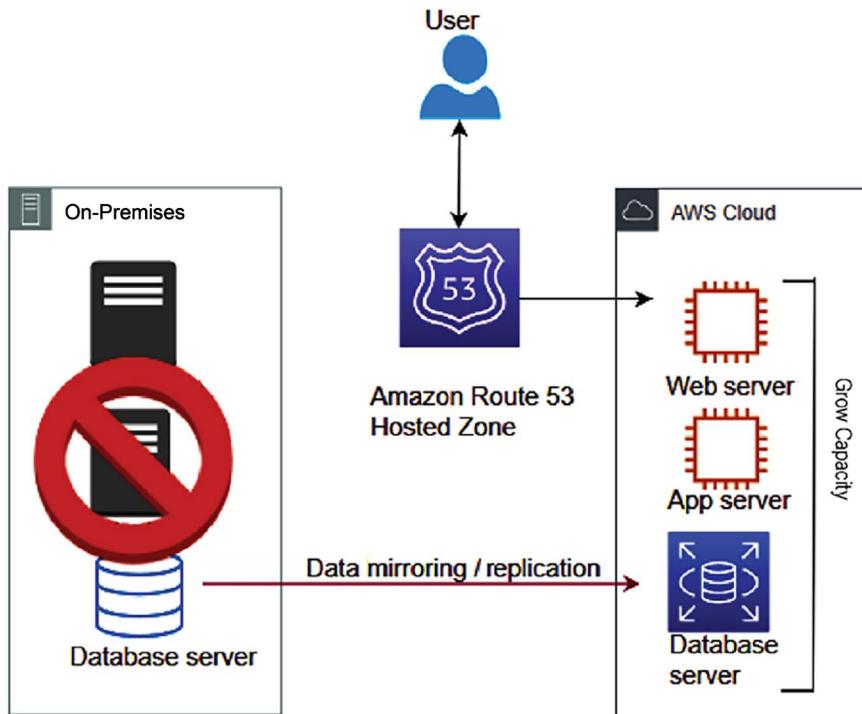


Figure 8.6: Recovery in the pilot light method

For the pilot light method, in the case of a disaster, you need to perform the following steps:

1. Start the application and web servers that were in standby mode. Furthermore, scale out the application servers with horizontal scaling using a load balancer.
2. Vertically scale up the database instance that was running at low capacity.
3. Update the DNS record in your router to point to the new site.

In the pilot light method, you bring up the resources around the replicated core dataset automatically and scale the system as required to handle the current traffic. A pilot light DR pattern is relatively easy to set up and inexpensive. However, in this scenario, the RTO takes longer to automatically bring up a replacement system, while the RPO largely depends on the replication type. Let's explore the next approach, warm standby, which further improves your RTOs and RPOs.

Warm standby

Warm standby, or fully working low-capacity standby, represents an advanced approach to DR by leveraging the cloud's flexibility to deliver a cost-effective DR solution. This method enhances the basic pilot light strategy by maintaining a subset of your environment's services in a continuously running state, albeit at a lower capacity than your full production environment.

The key advantage of a warm standby approach is its balance between cost savings and readiness for recovery. By having essential services already up and running, albeit on a smaller scale, the recovery time in the event of a disaster is significantly reduced compared to cold or pilot light strategies where resources need to be provisioned or scaled up during the recovery process.

Organizations can tailor their warm standby environment to handle a specific percentage of their production traffic, such as 20%, 30%, or 50%, depending on their recovery objectives and budget considerations. This flexibility allows for a customized DR solution that aligns with business needs and risk tolerance levels.

Furthermore, the warm standby environment isn't just limited to DR scenarios; it can also serve dual purposes by providing a platform for non-production uses such as testing, staging, or development work. This dual use of the warm standby environment maximizes the value of the DR investment by utilizing the infrastructure for additional purposes beyond mere standby readiness, thereby optimizing resource utilization and cost efficiency.

The following diagram depicts two systems running in the warm standby method—the central system and a low-capacity system—on an AWS cloud. You can use a router such as Amazon Route 53 to distribute requests between the central system and the cloud system:

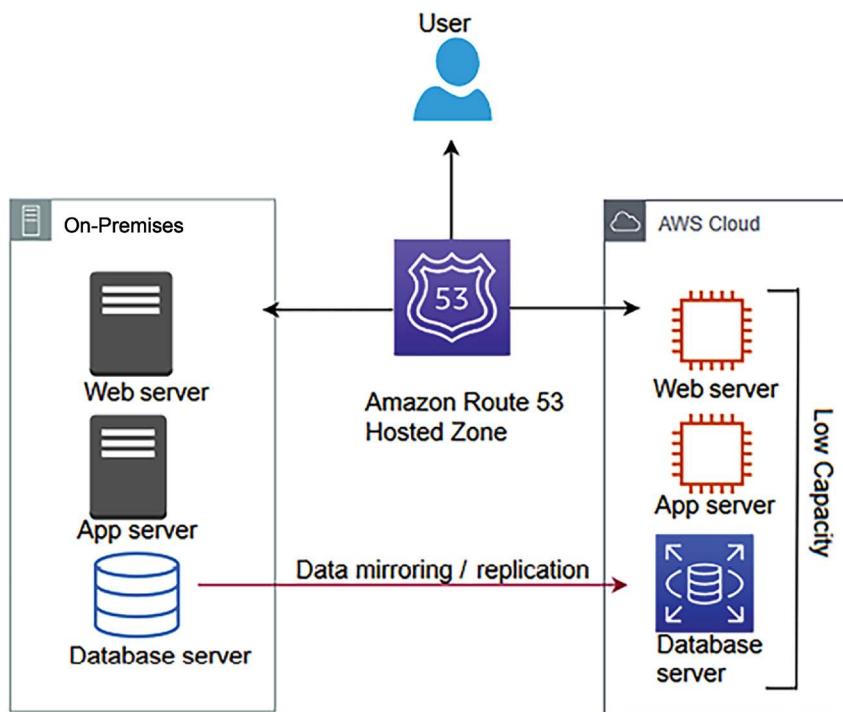


Figure 8.7: Warm standby scenario running an active-active workload with a low capacity

When it comes to databases, warm standby takes a similar approach to pilot light, where data is continuously replicating from the main site to the DR site. However, in warm standby, you run all necessary components 24/7, but they do not scale up for production traffic.

Often, the organization chooses a warm standby strategy for more critical workloads, so you need to ensure there are no issues in the DR site using continuous testing. The best approach to take is A/B testing, where the leading site will handle more significant traffic. A small amount of traffic, approximately 1% to 5%, is routed to the DR site. This will make sure that the DR site is able to serve traffic when the primary site is down. Also, make sure to patch and update the software on the DR site regularly to keep in sync with the production environment.

As shown in the following diagram, during the unavailability of the primary environment, your router switches over to the secondary system, which is designed to automatically scale its capacity up in the event of a failover from the primary system:

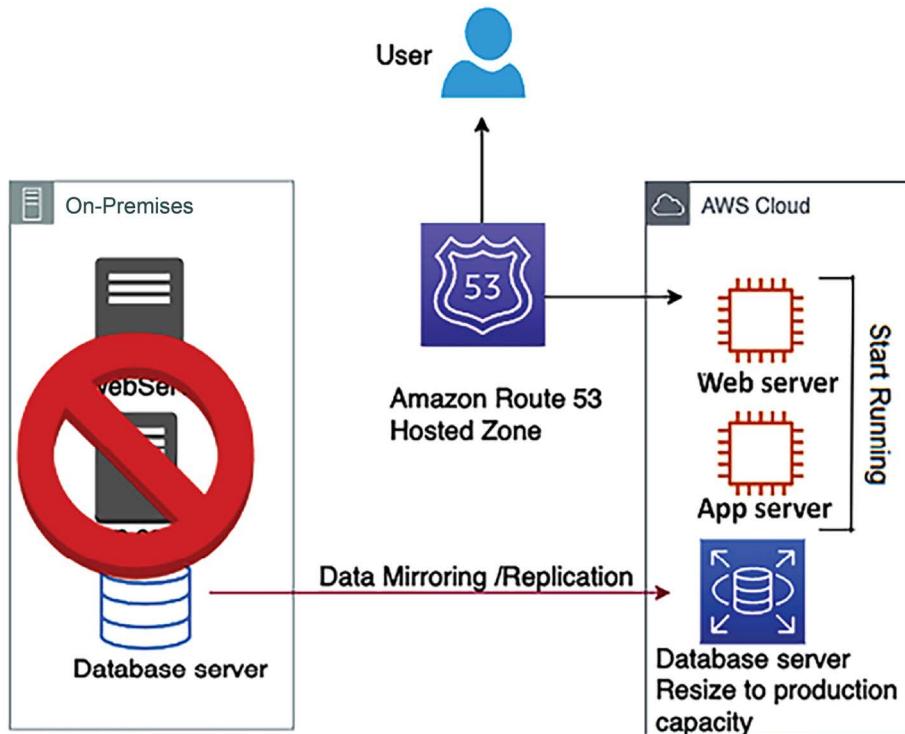


Figure 8.8: Recovery phase in the warm standby scenario

Suppose a failure occurs in the primary site. In that case, you can take the following approach:

1. Perform an immediate transfer of the critical production workload traffic to the DR site by increasing traffic routing from 5% to 100% in the secondary site. For example, in a banking business, you must first bring up your customer-facing website to keep it functioning.
2. Scale up the environment that was running on low capacity. You can apply vertical scaling for databases and horizontal scaling for servers.
3. As you scale up the environment, other non-critical workloads working in the background can now be transferred, such as warehouse management and shipping.

Tools you can use for warm standby such as Terraform, an open-source tool developed by HashiCorp, is renowned for its ability to build, change, and version infrastructure in a safe and efficient manner across various cloud providers. Alongside, Veeam stands out by offering comprehensive backup and replication solutions that cater to cloud, virtual, and physical environments, ensuring robust support for multi-cloud strategies. Zerto further complements these capabilities by providing disaster recovery, backup, and workload mobility software tailored for virtualized infrastructures and cloud settings.

A warm standby DR pattern is relatively complex to set up and expensive. The RTO is much quicker than the pilot light for the critical workload. However, for non-critical workloads, it depends on how quickly you can scale up the system, while the RPO largely depends on the replication type. Let's explore the next approach, multi-site, which provides near-zero RTOs and RPOs.

Multi-site

Lastly, the multi-site strategy, also known as a hot standby, helps you achieve a near-zero RTO and RPO. With this method, your DR site is a replica of the primary site with continuous data replication and traffic flow between sites. It is known as multi-site architecture due to the automated load balancing of traffic across regions or between on-premises and the cloud.

As shown in the following diagram, multi-site is the next level of DR, having a fully functional system running in the cloud at the same time as on-premises systems:

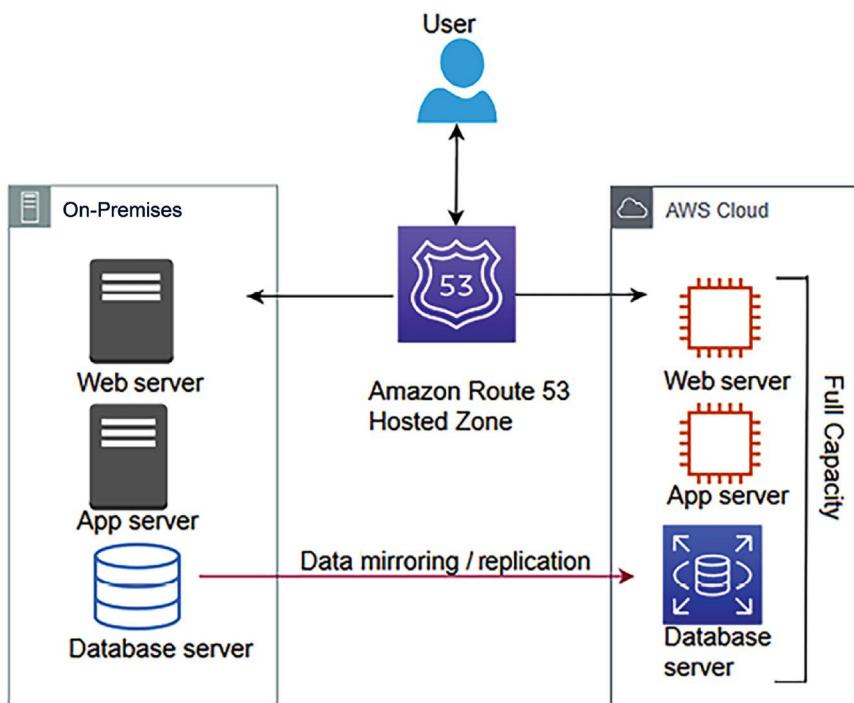


Figure 8.9: Multi-site scenario running an active-active workload with full capacity in AWS

The advantage of the multi-site approach is that it is ready to take a full production load at any moment. It's similar to warm standby but runs at full capacity on the DR site. If the primary site goes down, all traffic can immediately fail over to the DR site, which is an improvement over the loss in performance and time when switching over and scaling up the DR site in the case of a warm standby.

Implementing a multi-site strategy for DR requires a selection of advanced tools and technologies designed to automate and manage the seamless failover process, ensuring operational continuity with minimal performance loss. Cloud management platforms like VMware's vRealize Automation and Microsoft Azure Site Recovery play a crucial role in orchestrating the replication of virtual machines and data, facilitating an immediate switch to the DR site when necessary. Load balancers and global traffic managers, including solutions such as F5 BIG-IP and AWS Route 53, dynamically direct traffic based on site availability and load, ensuring the DR site can handle incoming requests instantly.

Infrastructure-as-Code (IaC) tools like Terraform and AWS CloudFormation enable rapid provisioning and scaling of necessary infrastructure, allowing the DR site to mirror the production environment's capabilities swiftly. Additionally, network performance monitoring tools like SolarWinds and Nagios offer real-time insights into network health, aiding in the prompt detection of issues that could necessitate a failover.

A multi-site DR pattern is the most expensive as it requires redundancy to be built for all components; however, for businesses that require HA and cannot afford any downtime, such as financial institutions, healthcare services, and e-commerce platforms, the investment in a multi-site setup can be justified by the high cost of potential downtime.

The RTO is much quicker for all workloads in this scenario, while the RPO largely depends upon the replication type.

Let's explore some best practices around DR to make sure your system is running reliably.

Applying best practices for DR

As you start thinking about DR, here are some important considerations:

- **Start small and build as needed:** Make sure you first bring up the critical workloads with the most business impact and build upon this to bring up less critical loads.
- **Apply the data backup life cycle:** Take backups of everything, whether it is your file server, machine image, or databases. Keeping lots of active backups could increase costs, however, so make sure to apply a life cycle policy to archive and delete data as per your business needs. For example, you can choose to keep a 90-day active backup and, after that period, store it in low-cost archive storage such as a tape drive or Amazon Glacier. After 1 or 2 years, you may want to set a life cycle policy to delete the data. Compliance with standards such as PCI-DSS may require you to store data for seven years, and in that case, you must opt for archival data storage to reduce costs.

- **Check your software licenses:** Managing software licenses can be a daunting task, especially in the current microservice architecture environment, where you have several services running independently on their virtual machines and databases. Software licenses could be associated with several installations, a number of CPUs, and several users. It becomes tricky when you go for scaling. It's important to monitor and check these; you need to have enough licenses to support your scaling needs. Also ensure you don't buy excessive licenses, which you may not utilize and will cost you more money. Overall, make sure to manage your license inventory, like your infrastructure or software.
- **Plan your scaling:** For horizontal scaling, add more instances with software installed, and in vertical scaling, add more CPU or memory. You need to understand your software licensing agreement and ensure you have enough licenses to fulfill system scaling.
- **Test your solutions often:** DR sites are created for rare DR events and are often overlooked. You need to ensure your DR solution is working as expected to achieve high reliability in case of an incident. Compromising a defined SLA can violate contractual obligations and cause a loss of money and customer trust.
- **Play gameday:** One way to test your solution often is by playing *gameday*. To play gameday, you choose a day when the production workload is small and gather all the team responsible for maintaining the production environment. You can simulate a disaster event by bringing down a portion of the production environment and letting the team handle the situation to keep the environment up and running. These events ensure you have working backups, snapshots, and machine images to handle disaster events.
- **Continuously monitor resources:** Put a monitoring system in place to ensure automated failover to the DR site happens if an event occurs. Monitoring helps you to take a proactive approach and monitoring capacity saves you from resource saturation issues, which can impact your application's reliability.

Creating a DR plan and performing regular recovery validation helps to achieve the desired application reliability. Let's learn more about improving reliability through the use of the public cloud next.

Improving reliability with the cloud

In previous sections, you have seen examples of a cloud workload for the DR site. Many organizations have started to choose the cloud for DR sites to improve application reliability. Also, cloud marketplaces from major providers like AWS, Azure, and GCP offer a wide range of third-party solutions that can be integrated into DR planning and execution. These offerings typically include tools for backup and replication, orchestration, monitoring, and security.

The cloud provides data centers that are available across geographic locations at your fingertips. You can create a reliable site on another continent without any hassle. With the cloud, you can easily create and track the availability of your infrastructure, such as backups and machine images.

In the cloud, easy monitoring and tracking help make sure your application is highly available as per the business-defined SLA. The cloud gives you fine control over IT resources, cost, and handling trade-offs for RPO/RTO requirements.

The cloud provides easy and effective testing of your DR plan. You inherit features available in the cloud, such as the logs and metrics for various cloud services. Built-in metrics are a powerful tool for gaining insight into the health of your system.

With all available monitoring capabilities, you can notify the team of any threshold breach or trigger automation for system self-healing. For example, AWS provides CloudWatch, which collects logs and generates metrics while monitoring different applications and infrastructure components. It can trigger various automation to scale your application.

The cloud provides a built-in change management mechanism that helps to track provisioned resources. Cloud providers extend out-of-the-box capabilities to ensure applications and operating environments are running known software and can be patched or replaced in a controlled manner. For example, AWS provides AWS Systems Manager, which has the capability of patching and updating cloud servers in bulk.

With the cloud, you can design a scalable system, providing flexibility to add and remove resources automatically to match the current demand. Data is one of the essential aspects of any application's reliability. The cloud offers out-of-the-box data backup and replication tools, including machine images, databases, and files. In a disaster, all your data is backed up and appropriately saved in the cloud, which helps the system recover quickly.

Regular interaction across the application development and operation team will help address and prevent known issues and design gaps, thereby reducing the risk of failures and outages. Continually architect your applications to achieve resiliency and distribute them to handle any outages.

Summary

In this chapter, you learned about various principles to make your system reliable. These principles include making your system self-healing by applying automation rules and reducing the impact in the event of failure by designing a distributed system where the workload spans multiple resources.

Overall system reliability heavily depends on your system's availability and ability to recover from disaster events. You learned about synchronous and asynchronous data replication types and how they affect your system reliability. You learned about various data replication methods, including array-based, network-based, host-based, and hypervisor-based methods. Each replication method has its pros and cons. There are multiple vendors' products available to achieve the desired data replication.

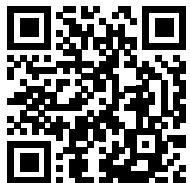
You learned about various disaster planning methods depending on the organization's needs and the RTO and RPO. You learned about the backup and restore method, which has high RTO and RPO and is easy to implement. The pilot light method improves your RTO/RPO by keeping critical resources, such as databases, active in the DR site. The warm standby and multi-site methods maintain an active copy of a DR site's workload and increase application reliability by lowering the system's RTO/RPO and the system's complexity and costs.

You learned about utilizing the cloud's built-in capability to ensure application reliability.

Solution design and launch may only happen sometimes, but operational maintenance is an everyday task. In the next chapter, you will learn about the alerting and monitoring aspects of solution architecture, including various design principles and technology choices to make your application operationally efficient and achieve operational excellence.

Join our book's Discord space

Join the book's Discord workspace to ask questions and interact with the authors and other solution architecture professionals: <https://packt.link/SAHandbook>



9

Operational Excellence Considerations

Application maintainability is one of the main aspects that a solutions architect should consider during architectural design. Every new project starts with lots of planning and resources at the beginning and teams spend the initial months creating and launching your application. After the production launch, the application needs several things to be taken care of to keep operating. You need to continually monitor your application to find and resolve any issues on a day-to-day basis.

The operations team needs to handle application infrastructure, security, and any software issues to make sure your application is running reliably without any problems or issues. Often, enterprise applications are complex, with defined **Service-Level Agreements (SLA)** regarding application availability. Your operations team needs to understand business requirements and prepare themselves accordingly to respond to any event.

Operational excellence is all about ensuring that every component and layer of your system architecture operates in an efficient way. This involves continuous monitoring, optimization, and improvement of processes, systems, and services.

Operational excellence should be implemented across every component and layer of architecture. In modern microservice applications, there are so many moving parts involved that make system operations and maintenance a complicated task.

Your operations team needs to put proper monitoring and alert mechanisms in place to tackle any issues that could hamper the business flow. Operational issues involve coordination from several teams for preparation and resolution.

In this chapter, you will learn various design principles applicable to achieving operational excellence for your solution. You will get an understanding of the right selection of technologies to ensure operational maintainability at every layer of your software application. You will learn the following best practices of operational excellence:

- Design principles for operational excellence

- Selecting technologies for operational excellence
- Achieving operational excellence in the public cloud
- Driving efficiency with CloudOps

By the end of this chapter, you will know various processes and methods to achieve operational excellence. You will have learned about best practices you can apply throughout application design, implementation, and post-production to improve application operability.

Design principles for operational excellence

Operational excellence is about running your application with minimal possible interruption to gain maximum business value. It is about applying continuous improvements to make the system efficient.

The following sections talk about the standard design principles that can help you strengthen your system's maintainability. You will find that all operational excellence design principles are closely related to and complement each other.

Automating manual tasks

Technology moves fast, and IT operations need to keep up with that, particularly where hardware and software inventories are procured from multiple vendors. Enterprises are building hybrid cloud and multi-cloud systems, so you must learn how to handle both on-premises and cloud operations. Modern systems have extensive user bases, with various microservices working together and millions of devices connected in a network. There are many moving parts in an IT operation, making it difficult to run things manually.

Organizations maintain agility, and operations have to be fast to make use of the required infrastructure for new service development and deployment. The operations team has a more significant responsibility to keep services up and running and recover quickly in case of an unexpected event. Now, it is required to take a proactive approach in IT operations rather than waiting for an incident to happen and then reacting.

Your operations team can work very efficiently by applying automation. Manual jobs need to be automated so that the team can focus on more strategic initiatives rather than getting overworked with tactical work. Automating active discovery and response for any security threat is most important to free up the team. Spinning up a new server or starting and stopping services should be automated using an **infrastructure as code** (IaC) approach. Automation allows the team to devote more time to innovation.

For your web-facing application, you can detect anomalies in advance using machine learning prediction, before they impact your system. You can raise an automated security ticket if someone exposes your server to the world with HTTP port 80. You can pretty much automate the entire infrastructure and redeploy it multiple times as a *one-click solution*. Automation also helps to prevent human error, which can occur even if a person is doing the same job repetitively. Automation is now a must-have for IT operations.

Making incremental and reversible changes

Operational optimization is an ongoing process whereby continuous effort is required to identify gaps and improve on these. These gaps could be focused on reliability, availability, performance, and cost-effectiveness, ensuring that the architecture supports the business goals and adapts to changing needs. Achieving operational excellence is a journey. Changes are always required in all parts of your workload to maintain it. For example, often, the operating systems of your server need to be updated with a security patch provided by your vendor. Various software that your application uses needs version upgrades. You might need to make changes in the system to adhere to new compliance requirements.

You should design your workload in such a way that it allows all system components to get updated regularly so the system will benefit from the latest and most significant updates available. Automate your flow to apply small, incremental changes to avoid any significant impact. Any changes should be reversible to restore the system's working conditions in case of any issues. Incremental changes help with thorough testing and improve overall system reliability. Automate any change management to avoid human error and achieve efficiency.

Predicting failures and responding

Preventing failures is vital to achieving operational excellence. Failures are bound to happen, and it's critical to identify them as far in advance as possible. During architecture design, anticipate failure and ensure you design for failure to prevent it from happening. Assume that everything will fail all the time and have a backup plan ready. Perform regular exercises to identify any potential source of failure. Try to remove or mitigate any resource that could cause a failure during system operation.

Create a test scenario based on your SLA that includes a system **recovery time objective (RTO)** and **recovery point objective (RPO)**. Test your scenario, and make sure you understand their impact. Ensure your team is ready to respond to any incident by simulating a production-like scenario. Test your response procedure to make sure it is resolving issues effectively and create a confident team that is familiar with response execution.

Learning from mistakes and refining

As operational failures occur in your system, you should learn from the mistakes and identify the gaps involved. Make sure those same events do not occur again, and you should have a solution ready in case a failure is repeated.

One way to improve is by running a **root cause analysis (RCA)**. During RCA, you gather the team and ask five *Whys*. With each *why*, you peel off one layer of the problem, and after asking the last *why*, you get to the bottom of the issue. After identifying the actual cause of an issue, you can prepare a solution and update the operational runbook with the ready-to-use solution.

As your workload evolves with time, you must ensure the operation procedure gets updated accordingly. Make sure to validate and test all methods regularly and that the team is familiar with the latest updates to execute them.

Keeping the operational runbook updated

Often, a team overlooks documentation, which results in an outdated runbook. A runbook provides a guide to executing a set of actions to resolve issues arising due to external or internal events. A lack of documentation can make your operation people-dependent, which can be risky due to team attrition. Always establish processes to keep your system operations people-independent, and document all aspects.

In the runbook, you want to keep track of all previous events and actions taken by team members to resolve them so that any new team members can quickly resolve similar incidents during operation support.

The system admin should maintain the runbook with steps to start, stop, patch, and update the system. The operations team should include the system testing and validation result, along with the procedure to respond to the event. Your runbook should also include the defined SLA concerning RTO/RPO, latency, scalability, performance, and so on.

Automate processes to annotate documents as a team applies changes to the system and also after every build. You can use annotation to automate your operation, and it is easily readable by code to accommodate business priorities and customer needs continuously.

Selecting technologies for operational excellence

The operations team needs to create procedures and steps to handle any operational incidents and validate the effectiveness of their actions. They need to understand the business need to provide efficient support and collect systems and business metrics to measure the achievement of business outcomes.

The operational procedure can be categorized into three phases—planning, functioning, and improving. Let's explore technologies that can help in each phase.

Planning for operational excellence

The first step in the operational excellence process is to define operational priorities to focus on the high business impact areas. Those areas could be, for example, applying automation, streamlining monitoring, developing team skills as the workload evolves, and focusing on improving overall workload performance.

There are tools and services available that crawl through your system by scanning logs and system activity. These tools offer a fundamental set of assessments that suggest enhancements for the system's environment. They assist in forming priorities by providing key insights and recommendations for optimization.

After identifying and understanding the priorities, you need to design the operation, which includes the workloads to design and build the procedures to support them. The design of a workload should encompass its implementation, deployment, updating process, and operational strategy. An entire workload can be viewed as various application components, infrastructure components, security, data governance, and operations automation. After operation design, create a checklist for operational readiness. These checklists should be comprehensive to ensure the system is ready for operation support when going live in production. This includes logging and monitoring, a communication plan, an alert mechanism, the team skillset, a team support charter, a vendor support mechanism, and so on.

For operational excellence planning, the following are the areas where you need appropriate tools for preparation:

- IT asset management
- Configuration management

Let's explore each area in more detail to understand the available tools and processes.

IT asset management

Operational excellence planning requires a list of IT inventories and tracks their use. These inventories include infrastructure hardware such as physical servers, network devices, storage, end-user devices, and so on. You also need to keep track of software licenses, operational data, legal contracts, compliance, and so on. IT assets include any system, hardware, or information that a company is using to perform a business activity.

Keeping track of IT assets helps an organization to make strategic and tactical decisions regarding operational support and planning. However, managing IT assets in a large organization can be daunting. Various **IT asset management (ITAM)** tools are available for the operations team to help in the asset management process. Some of the most popular ITAM tools are SolarWinds, Freshservice, ServiceDesk Plus, Asset Panda, PagerDuty, and Jira Service Desk.

IT management is more than tracking IT assets. It also involves monitoring and collecting asset data continuously to optimize usage and operation costs. ITAM makes the organization more agile by providing end-to-end visibility and the ability to apply patches and upgrades quickly. The following diagram illustrates ITAM:

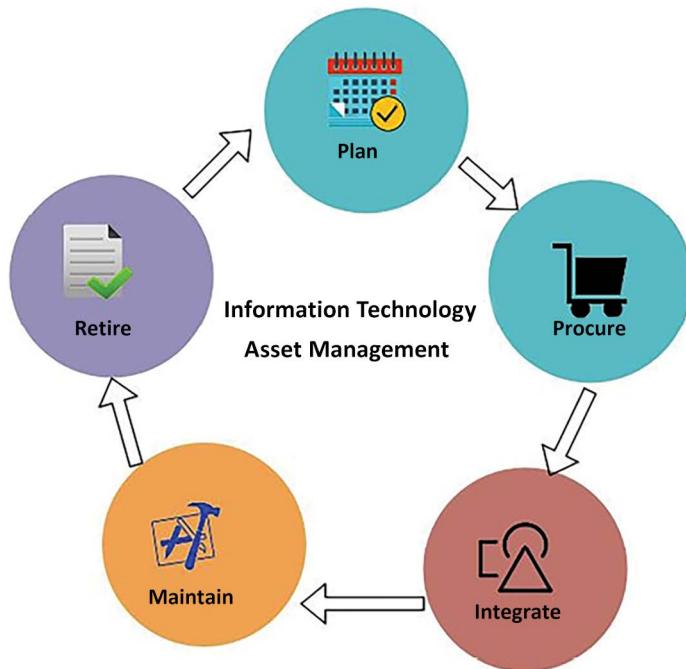


Figure 9.1: ITAM process

As shown in the preceding diagram, the ITAM process includes the following phases:

- **Plan:** An asset life cycle starts with planning, a more strategic focus to determine the need for overall IT assets and procurement methods. It includes a cost-benefit analysis and total cost of ownership.
- **Procure:** In the procurement phase, organizations acquire the asset based on the planning outcome. They may also decide to develop some holdings as required—for example, in-house software for logging and monitoring.
- **Integrate:** In this phase, an asset is installed in the IT ecosystem. The phase includes the operation and support of the asset, including defining user access—for example, installing a log agent to collect logs from all the servers in a centralized dashboard and restricting monitoring dashboard metrics to the IT operations team.
- **Maintain:** In the maintenance phase, the IT operations team keeps track of assets and acts to upgrade or migrate them based on the asset life cycle—for example, applying a security patch provided by the software vendor. This involves keeping track of the end of life for licensed software, such as planning to migrate from Windows Server 2008 to Windows 2022, as the old operating system is getting to the end of its life.

- **Retire:** In the retirement phase, the operations team disposes of the end-of-life asset. For example, if an old database server is getting to the end of its life, then the team takes action to upgrade it and migrate the required users and support to the new server.

ITAM helps organizations adhere to ISO 19770 compliance requirements. It includes software procurement, deployment, upgrade, and support. ITAM provides better data security and helps to improve software compliance. It also provides better communication between business units such as operations, finance, marketing teams, and frontline staff. Configuration management is another aspect of planning for operational excellence that helps to maintain IT inventory data along with details such as owner and current state. Let's learn more about it.

Configuration management

Configuration management maintains **configuration items (CIs)** to manage and deliver an IT service. CIs are tracked in the **configuration management database (CMDB)**. The CMDB keeps track of whether the server is physical or virtual, the operating system and its version (for example, Windows 2022 or Red Hat Enterprise Linux (RHEL) 8.0), the owner of the server (that is, support, marketing, or HR), and whether it has a dependency on other servers such as order management, and so on.

Configuration management is different from asset management. Asset management handles the entire life cycle of an asset, from planning to retirement, while a CMDB is a component of asset management that stores configuration records of an individual asset. As shown in the following diagram, configuration management implements the integration and maintenance part of asset management:

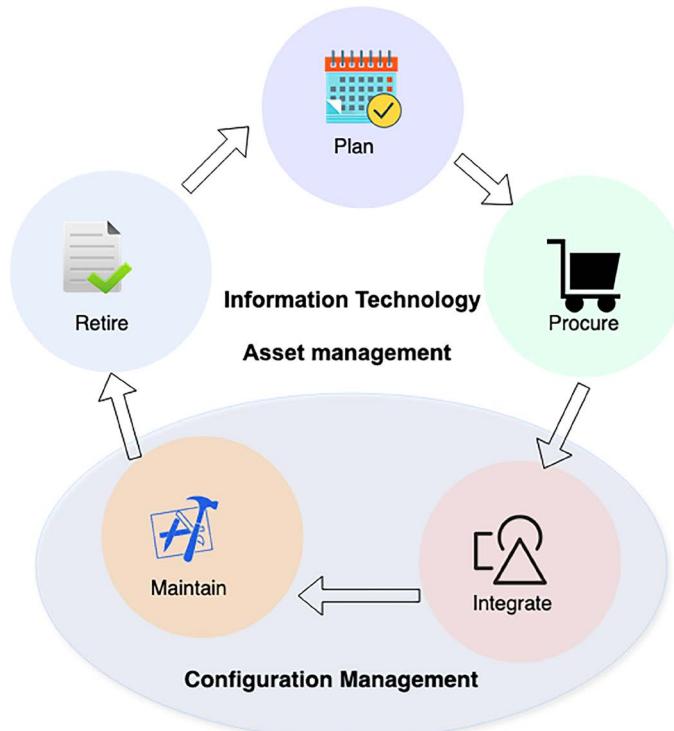


Figure 9.2: IT asset life cycle versus configuration management

Configuration management, as shown in the preceding diagram, implements the *Integrate* and *Maintain* part of asset management.

Configuration management and change management are complementary processes within IT operations. Configuration management focuses on maintaining an accurate and up-to-date record of all the components within an organization's IT environment, including their versions, configurations, and interrelationships. This ensures that systems are consistently and efficiently deployed and operated. On the other hand, change management oversees and controls modifications to the IT infrastructure, ensuring that changes are implemented in a coordinated, systematic manner to prevent unintended consequences. Together, they help maintain the integrity and stability of IT assets, with configuration management providing the detailed information necessary to assess the impact of changes, and change management ensuring that alterations to configurations are properly planned, executed, and documented.

A configuration management tool can help the operations team reduce downtime by providing readily available information on asset configuration. The most popular configuration management tools are Chef, Puppet, Ansible, and Bamboo. You will learn more details about them in *Chapter 11, DevOps and Solution Architecture Framework*.

IT management becomes easier if your workload is in a public cloud such as AWS, Microsoft Azure, or GCP. Cloud vendors provide inbuilt tools to track and manage IT inventories and configurations in one place. For example, AWS provides services such as AWS Config, which tracks all IT inventories that spin up as a part of your AWS cloud workload, and services such as AWS Trusted Advisor, which recommends cost, performance, and security improvements, which you can use to decide how to manage your workload. You can see an example of AWS Trusted Advisor in the following screenshot:

The screenshot shows the AWS Trusted Advisor dashboard with the following sections:

- Trusted Advisor Recommendations:** A summary of checks categorized by status: Action recommended (16), Investigation recommended (49), and Checks with excluded items (0). It includes links to refresh or download all checks.
- Potential monthly savings:** Displays a green box showing \$40.55 in potential monthly savings from cost optimization checks. It includes a link to view all cost optimization checks.
- Checks summary:** A table showing the count of checks for various categories: Security (12), Fault tolerance (2), Cost optimization (1), and Service limits (1).
- Recent changes (11):** A table listing recent changes with their check names and dates. The changes include issues like "Amazon VPC Without Flow Logs", "Amazon S3 Bucket Replication Not Enabled", "Amazon EFS not in AWS Backup Plan", and "Amazon DynamoDB Auto Scaling Not Enabled".

Figure 9.3: AWS Trusted Advisor dashboard

As shown in the preceding screenshot, the AWS Trusted Advisor Dashboard shows 12 security issues that can be further explored to find out more details, among other things.

Configuration management plays a crucial role in continuously monitoring and documenting the configurations of IT resources, enabling the automation of configuration evaluations against predefined standards. The benefits of configuration management include:

- **Continuous monitoring:** It allows for the ongoing observation and documentation of changes in your IT resources' configurations.
- **Change management:** Helps in tracking the interconnections between resources and reviewing dependencies before implementing any changes.
- **Continuous assessment:** Facilitates regular audits and assessments to ensure that your IT resources comply with your organization's policies and guidelines.
- **Enterprise-wide compliance monitoring:** Offers a comprehensive view of compliance status across your enterprise, pinpointing any non-compliant accounts, and allows in-depth examination at the regional account level.
- **Management of third-party resources:** Enables the documentation of configurations for third-party resources, such as GitHub repositories, Microsoft Active Directory resources, and servers, whether on-premises or cloud-based.
- **Operational troubleshooting:** Captures a detailed history of configuration changes, aiding in the simplification of operational problem-solving.

Through configuration management, you gain the ability to conduct security analyses, maintain continuous oversight of resource configurations, and assess these configurations for potential security vulnerabilities. It is instrumental in ensuring compliance with internal policies and regulatory standards in your IT and third-party resources' configurations, and in continuously reviewing resource configuration changes against your desired standards.

In this section, you learned about asset management and configuration management. These are part of the **Information Technology Infrastructure Library (ITIL)** framework, which implements **Information Technology Service Management (ITSM)**, relevant to operational excellence. ITSM helps organizations to run their IT operations daily. You can learn more about ITIL from its governing body, AXELOS, by visiting its website (<https://www.axelos.com/best-practice-solutions/itil>). AXELOS offers ITIL certification to develop skills in the IT service management process.

As you have learned about planning, let's now explore the functioning of IT operations in the next section.

The functioning of operational excellence

Operational excellence is determined by proactive monitoring and quick response and recovery in the case of an unexpected event. By understanding the operational health of a workload, it is possible to identify when events and responses impact it. Use tools that help you understand the operational health of the system using **metrics** and **dashboards**. You should send log data to centralized storage and define metrics to establish a benchmark. These tools also enable the automation of responses to operational events, triggering their execution in response to specific alerts.



Design your workload components to be replaceable. This approach means that instead of spending time fixing issues, you can reduce recovery time by substituting failed components with known, reliable versions. Then, analyze the failed resources without impacting a production environment.

For the functioning of operational excellence, the following are the areas where appropriate tools are needed:

- Monitoring system health
- Handling alerts and incident response

Let's explore each area with information on the available tools and processes.

Monitoring system health

Keeping track of system health is essential to understanding *workload behavior*. The operations team uses system health monitoring to record any anomalies in system components and acts accordingly. Traditionally, monitoring is limited to the infrastructure layer, keeping track of the server's CPU and memory utilization. However, monitoring needs to be applied to every layer of the architecture. Significant components where monitoring is applied are:

- Infrastructure monitoring
- Application monitoring
- Platform monitoring
- Log monitoring
- Security monitoring

We discuss these in the following subsections.

Infrastructure monitoring

Infrastructure monitoring is essential and is the most popular form of monitoring. Infrastructure includes components required for hosting applications. These are core services such as storage, servers, network traffic, load balancers, and so on.

Infrastructure monitoring may consist of metrics such as the following:

- **CPU usage:** Percentage of CPU utilized by the server in a given period
- **Memory usage:** Percentage of **random-access memory (RAM)** utilized by the server in a given period
- **Network utilization:** Network packets *in and out* over the given period
- **Disk utilization:** Disk read/write throughput and **input/output operations per second (IOPS)**
- **Load balancer:** Number of request counts in a given period

There are many more metrics available, and organizations need to customize those monitoring metrics as per their application monitoring requirements. The following screenshot shows a sample monitoring dashboard for network traffic:

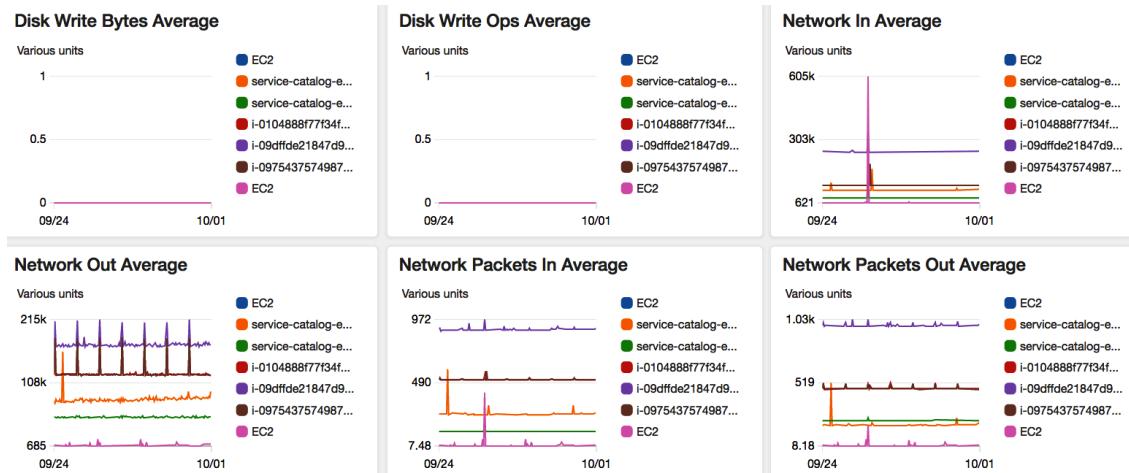


Figure 9.4: Infrastructure monitoring dashboard

You can see on the preceding system dashboard that there is a spike on one day in the Network In Average pane, with color coding applied for different servers. The operations team can dive deep into this and the other graphs and resources to get a more granular view to determine the overall health of the infrastructure.

Application monitoring

Sometimes, your infrastructure is all healthy except for applications having an issue due to some bug in your code or any third-party software issues. You may have applied some vendor-provided operating system security patch that messed up your application. Application monitoring helps with this.

Application monitoring may include metrics such as the following:

- **Endpoint invocation:** Number of requests in a given period
- **Response time:** Average response time to fulfill the request
- **Throttle:** The number of valid requests spilled out as the system runs out of capacity to handle the additional requests
- **Error:** The application throws an error while responding to a request

The following screenshot shows a sample application endpoint-monitoring dashboard:

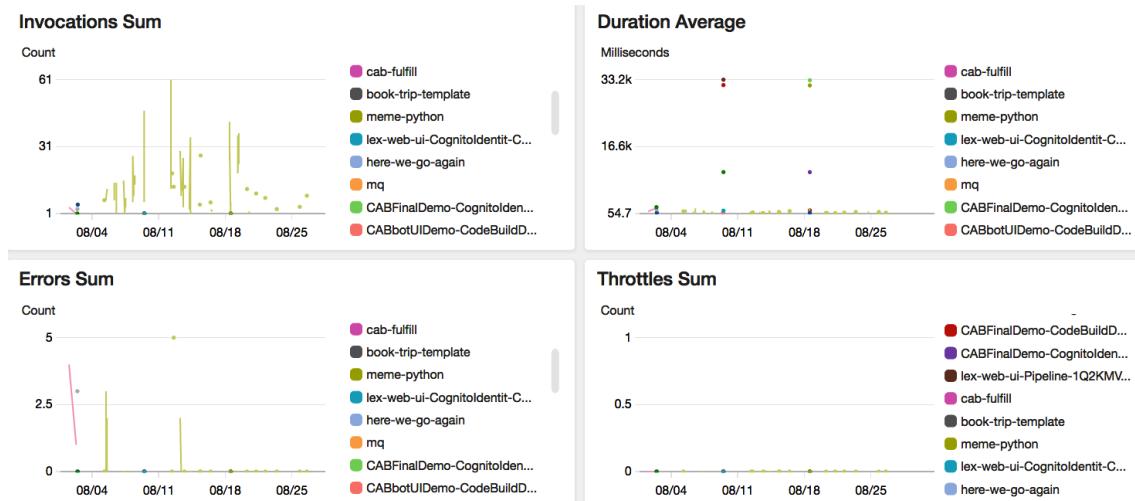


Figure 9.5: Application monitoring dashboard

There could be many more metrics based on the application and technology. For example, a memory garbage collection amount for a Java application, several HTTP POST and GET requests for a RESTful service, a count of 4XX client errors, a count of 5XX server errors for a web application, and what they might be looking for that would indicate poor application health.

Platform monitoring

Your application may be utilizing several third-party platforms and tools that must be monitored. These may include the following:

- **Memory caching:** Redis and Memcached
- **Relational database:** Oracle Database, Microsoft SQL Server, Amazon Relational Database Service (RDS), PostgreSQL
- **NoSQL database:** Amazon DynamoDB, Apache Cassandra, MongoDB
- **Big data platform:** Apache Hadoop, Apache Spark, Apache Hive, Apache Impala, Amazon Elastic MapReduce (EMR)
- **Containers:** Docker, Kubernetes, OpenShift
- **Business intelligence tools:** Tableau, MicroStrategy, Kibana, Amazon QuickSight
- **Messaging system:** MQSeries, Java Message Service (JMS), RabbitMQ, Simple Queue Service (SQS)
- **Search:** Open search, Solr search-based application

Each of the tools mentioned above has its own set of metrics that you need to monitor to make sure your application is healthy as a whole. The following screenshot shows the monitoring dashboard of a relational database platform:

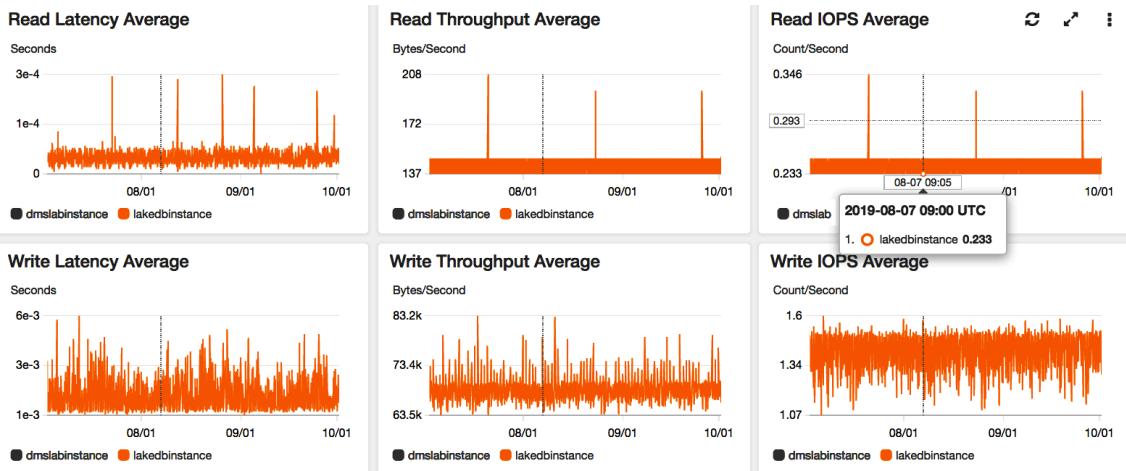


Figure 9.6: Platform monitoring dashboard for a Relational Database Management System (RDBMS)

In the preceding dashboard, you can see the database has lots of writing activity, showing that the application is continuously writing data. On the other hand, read events are relatively consistent except for some spikes.

Log monitoring

Traditionally, log monitoring was a manual process, and organizations took a reactive approach to analyze logs when issues were encountered. However, with more competition and increasing user expectations, it has become essential to take quick action before users notice any issues. For a proactive approach, you should have the ability to stream logs in a centralized place and run queries to monitor and identify issues. For example, if some product page throws an error, you need to know the error immediately and fix the problem before the user complains; otherwise, you will suffer a revenue loss.

In the case of any network attack, you need to analyze your network log and block suspicious IP addresses. Those IPs may send an erroneous number of data packets to bring down your application. Monitoring systems such as AWS CloudWatch, Logstash, Splunk, Google Stackdriver, and so on provide an agent to install on your application server. The agent will stream logs to a centralized storage location. You can directly query central log storage and set up alerts for any anomalies.

The following screenshot shows a sample network log collected in a centralized place:

```
2 789211807855 eni-0c7812c55522bd887 172.31.0.23 172.31.0.252 49232 1433 6 40 1860 1549397294 1549397893 ACCEPT OK
2 789211807855 eni-0c6918ddd57f2978f 104.248.247.78 172.31.0.202 33794 8088 6 1 40 1549397503 1549397563 REJECT OK
2 789211807855 eni-0c6918ddd57f2978f 78.128.112.98 172.31.0.202 58594 3393 6 1 40 1549397503 1549397563 REJECT OK
2 789211807855 eni-0c6918ddd57f2978f 172.104.121.206 172.31.0.202 38620 465 6 1 40 1549397503 1549397563 REJECT OK
2 789211807855 eni-0c6918ddd57f2978f 193.32.160.35 172.31.0.202 48479 40004 6 1 40 1549397503 1549397563 REJECT OK
2 789211807855 cni-0c6918ddd57f2978f 172.31.0.202 172.31.0.23 46346 1433 6 20 1280 1549397503 1549398103 ACCEPT OK
2 789211807855 eni-0c6918ddd57f2978f 172.31.0.23 172.31.0.202 1433 46346 6 20 820 1549397503 1549398103 ACCEPT OK
2 789211807855 eni-0c6918ddd57f2978f 172.31.0.202 172.31.0.23 44622 1433 6 20 1280 1549397503 1549398103 ACCEPT OK
```

Figure 9.7: Raw network log streamed in a centralized data store

You can run a query in these logs and find out the top 10 source IP addresses with the highest number of reject requests, as shown in the following screenshot:

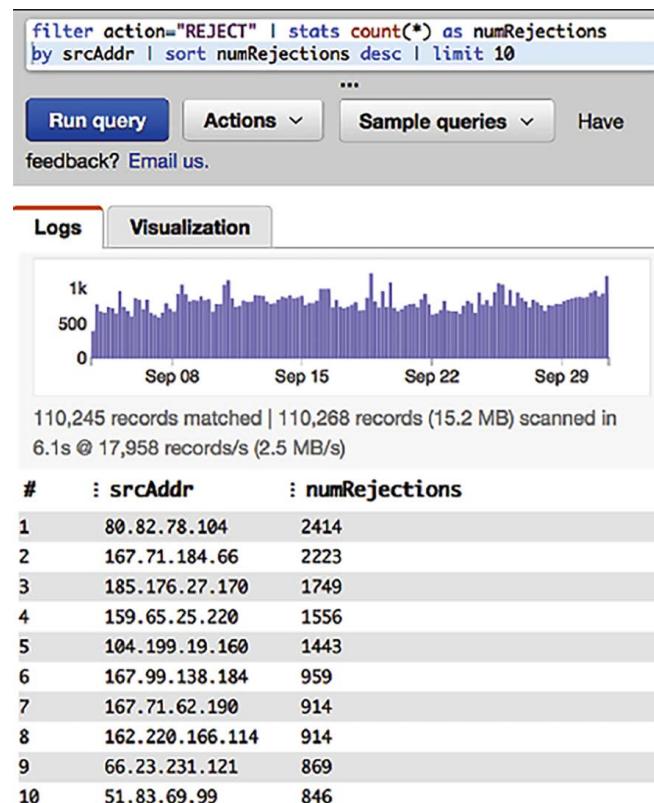


Figure 9.8: Insight from raw network log by running query

As shown in the preceding query editor, you can create a graph and set an alarm if the number of rejections detected crosses a certain threshold, such as more than 5,000.

Security monitoring

Security is a critical aspect of any application. Security monitoring should be considered during solution design. As you learned when we looked at security in the various architectural components in *Chapter 7, Security Considerations*, security needs to be applied at all layers. It would be advisable to implement security monitoring to act and respond to any adverse event.

The following list shows where security monitoring needs to be applied:

- **Network security:** Monitor any unauthorized port opening, suspicious IP address, and activity
- **User access:** Monitor any unauthorized user access and suspicious user activity
- **Application security:** Monitor any malware or virus attack
- **Web security:** Monitor for **Distributed Denial of Service (DDoS)** attacks, SQL injection, or **cross-site scripting (XSS)**
- **Server security:** Monitor any gap in security patches
- **Compliance:** Monitor any compliance lapses such as violations of **payment card industry (PCI)** compliance checks for payment applications or the **Health Insurance Portability and Accountability Act (HIPAA)** for healthcare applications
- **Data security:** Monitor unauthorized data access, data masking, and data encryption at rest and in transit

One example of security monitoring using **Amazon GuardDuty** for the AWS cloud is shown below:

The screenshot shows the AWS GuardDuty interface. The left sidebar has links for 'Findings', 'Settings', 'Lists', 'Accounts', 'What's New', and 'Usage'. The main area is titled 'Findings' and shows a table of findings. The table has columns: Actions, Severity: Medium, High (with a dropdown menu), Saved filters / Auto-archive, Apply saved filters, Filter criteria (Severity: Medium, High selected), Indent type, Resource, Last seen, Account ID, and Count. There are 306 findings out of 432 total. The findings listed include:

Severity: Medium, High	Indention type	Resource	Last seen	Account ID	Count
<input type="checkbox"/> Recon:IAMUser/ResourcePermissions	Admin: ASIA2ZZIW6DOEWGSAYNB	19 hours ago	7425778...	1	
<input type="checkbox"/> UnauthorizedAccess:IAMUser/ConsoleLogin	Admin:	19 hours ago	7425778...	1	
<input checked="" type="checkbox"/> Trojan:EC2/BlackholeTraffic	Instance: I-061c1b8673fab0f671	a day ago	9569546...	3	
<input type="checkbox"/> Trojan:EC2/BlackholeTraffic	Instance: I-061c1b8673fab0f671	2 days ago	5225365...	2	
<input type="checkbox"/> Recon:IAMUser/TorIPCaller	Jsmith: ASIA76IQ3TN3XD2DX4D	3 days ago	5496394...	50	
<input type="checkbox"/> UnauthorizedAccess:IAMUser/TorIPCaller	Jsmith: ASIA76IQ3TN3XD2DX4D	3 days ago	5496394...	6	
<input type="checkbox"/> Recon:IAMUser/TorIPCaller	Jsmith: ASIA76IQ3TNVJWW2WBN	3 days ago	5496394...	10	

Figure 9.9: Security monitoring using Amazon GuardDuty

Other tools that can be used for security monitoring include Imperva, McAfee, Qualys, Palo Alto Networks, Sophos, and Symantec.

While you are putting application monitoring tools in place to monitor all components of your system, it is essential to monitor the monitoring system. Make sure to monitor the host of your monitoring system. For example, if you're hosting your monitoring tool in Amazon **Elastic Compute Cloud (EC2)**, then AWS CloudWatch can monitor the health of EC2.

Handling alerts and incident response

Monitoring is one part of operational excellence functioning; the other part involves handling alerts and acting upon them. Using alerts, you can define the system threshold and when you want to work. For example, if the server CPU utilization reaches 70% for 5 minutes, then the monitoring tool records high server utilization and sends an alert to the operations team to take action to bring down CPU utilization before a system crashes. Responding to this incident, the operations team can add the server manually. When automation is in place, autoscaling triggers the alert to add more servers as per demand. It also sends a notification to the operations team, which can be addressed later.

Incident response is essential for dealing with these alerts and resolving issues. Actions taken can be automated or managed by an operations team to address system outages or failures. This process ensures that any disruptions are promptly and efficiently handled to reduce their impact on the organization's operations, maintaining the system's reliability and availability for users and stakeholders.

Often, it would help if you define the alert category and the operations team prepared for the response as per the alert severity. The following levels of severity provide an example of how to categorize alert priority:

- **Severity 1:** This is a critical priority issue. A Sev1 issue should only be raised when there is a significant customer impact for which immediate human intervention is needed. A Sev1 alert could be that the entire application is down. The typical team needs to respond to these alerts within 15 minutes and requires 24/7 support to fix the issue.
- **Severity 2:** This is a high-priority alert that should be addressed during business hours. For example, the application is up, but the rating and review system is not working for a specific product category. The typical team needs to respond to these alerts within 24 hours and requires regular office hours support to fix the issue.
- **Severity 3:** This is a medium-priority alert that can be addressed during business hours over days—for example, the server disk is going to fill up in 2 days. The typical team needs to respond to these kinds of alerts within 72 hours and requires regular office hours support to fix the issue.
- **Severity 4:** This is a low-priority alert that can be addressed during business hours over the week—for example, **Secure Sockets Layer (SSL)** certification will expire in 2 weeks. The typical team needs to respond to these kinds of alerts within the week and requires regular office hours support to fix the issue.
- **Severity 5:** This falls into the notification category, where no escalation is needed, and it can be simple information—for example, sending a notification that deployment is complete. Here, no response is required in return since it is only for information purposes.

Each organization can have different alert severity levels as per their application needs. Some organizations may want to set four severity levels, while others may go for six. Also, alert response times may differ. Some organizations may want to address Sev2 alerts within 6 hours on a 24/7 basis rather than waiting for them to be addressed during office hours.

While setting up an alert, make sure the title and summary are *descriptive* and *concise*. Often, an alert is sent to a mobile (as an SMS) or a pager (as a message) and needs to be short and informative enough to enable the receiver to take immediate action. Make sure to include proper metrics data in the message body. For example, in the message body, include specific information such as *The disk is 90% full in production-web-1 server* rather than just saying *The disk is full*. The following screenshot from CloudWatch shows an example alarm dashboard:

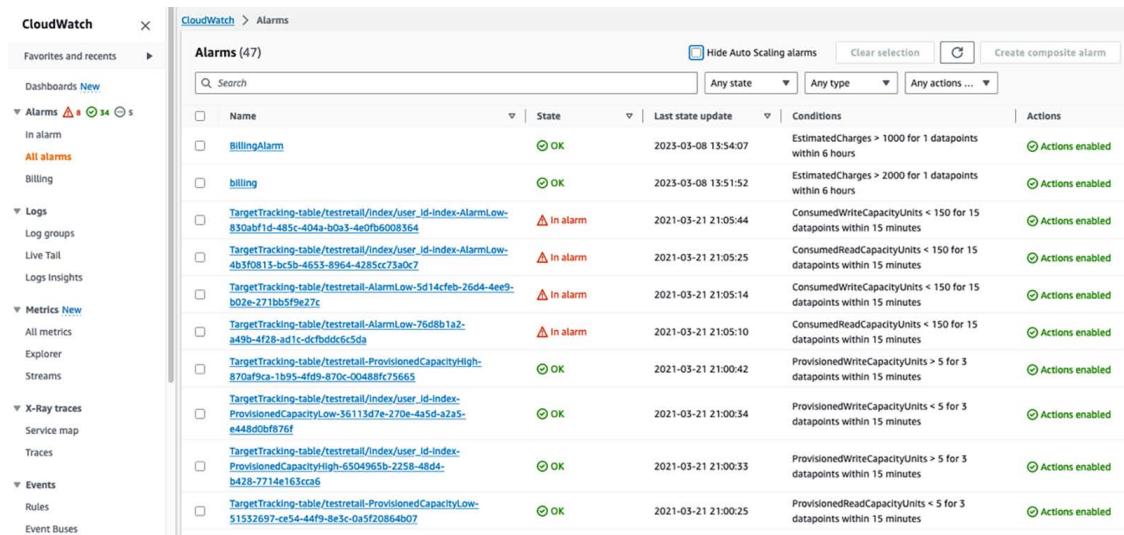


Figure 9.10: Alarm dashboard

As shown in the preceding alarm dashboard, one alarm is in progress when a NoSQL Amazon DynamoDB database table called testretail uses a low write capacity unit and causes unnecessary additional costs.

The bottom alarms and top two alarms have an OK status as data is collected during monitoring that is well within the threshold.



There may sometimes be alarms showing **insufficient data**, which means there needs to be more data points to determine the state of resources you monitor. Consider this alarm valid if data can be collected to move it into the OK state.

Testing incident response in the case of critical alerts is important to make sure you are ready to respond as per the defined SLA. Make sure your threshold is set up correctly so that you have enough room to address the issue and don't send too many alerts. Ensure that as soon as the issue is resolved, your alert gets reset to the original setting and is ready to capture event data again.

An incident is any unplanned disruption that impacts the system and customer negatively. The first response during an incident is to recover the system and restore the customer experience. Fixing the underlying issue can be addressed later as the system gets restored and starts functioning. The automated alert helps to actively discover the incident and minimizes user impact. This can act as a failover to a disaster recovery site if the entire system is down. The primary system can be fixed and restored later.

For example, Netflix uses the *Simian Army* (<https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116>), a set of tools to test system resilience, including Chaos Monkey. Chaos Monkey orchestrates the random termination of a production server to test if the system can respond to disaster events without any impact on end users. Similarly, Netflix has other monkeys to test various dimensions of system architecture, such as Security Monkey, Latency Monkey, and even Chaos Gorilla, which can simulate outages of the entire availability zone.

Monitoring, alerts, and incident response are critical components of achieving operational excellence. All monitoring systems typically have an alert feature integrated with them. A fully automated alert and monitoring system improves the operations team's ability to maintain the health of the system and provide expertise, enabling them to take quick action and enhance the user experience.

As you monitor your application environment, it's crucial to apply continuous improvement and continuously strive to achieve excellence. Let's learn more about improving operational excellence.

Improving operational excellence

Continuous improvement is required for any process, product, or application to excel. Operational excellence needs continuous improvement to attain maturity over time.

It would be advisable to implement small incremental changes as you perform RCA to learn lessons from various operations activities. Learning from failure will help you anticipate any operational event that may be planned (such as deployments) or unplanned (such as utilization surge). You should record all lessons learned and update remedies in your operation runbook. For operational improvement, the following are the areas where you need appropriate tools:

- IT operations analytics (ITOA)
- RCA
- Auditing and reporting

IT operations analytics

ITOA is the practice of gathering data from various resources to make a decision and predict any potential issue that you may encounter. It's essential to analyze all events and operational activities to improve. Analyzing failures will help to predict any future event and keep the team prepared to provide the appropriate response.

A large organization could have hundreds of systems generating a massive amount of data. Implement a mechanism to collect the logs of operations events, various activities across workloads, and infrastructure changes, storing this data for a defined length of time, such as 90 or 180 days. ITOA uses big data architecture to store and analyze multiple terabytes of data from all over the place.

It helps you discover issues you could not find by looking at individual tools and helps you determine dependencies between various systems, providing a holistic view.

As shown in the following diagram, each system has its own monitoring tool that helps to get insights and maintains individual system components. For operations analytics, you need to ingest this data in a centralized place. Having all operation data collection in one place gives you a single source of truth, where you can query required data and run analytics to get meaningful insights:

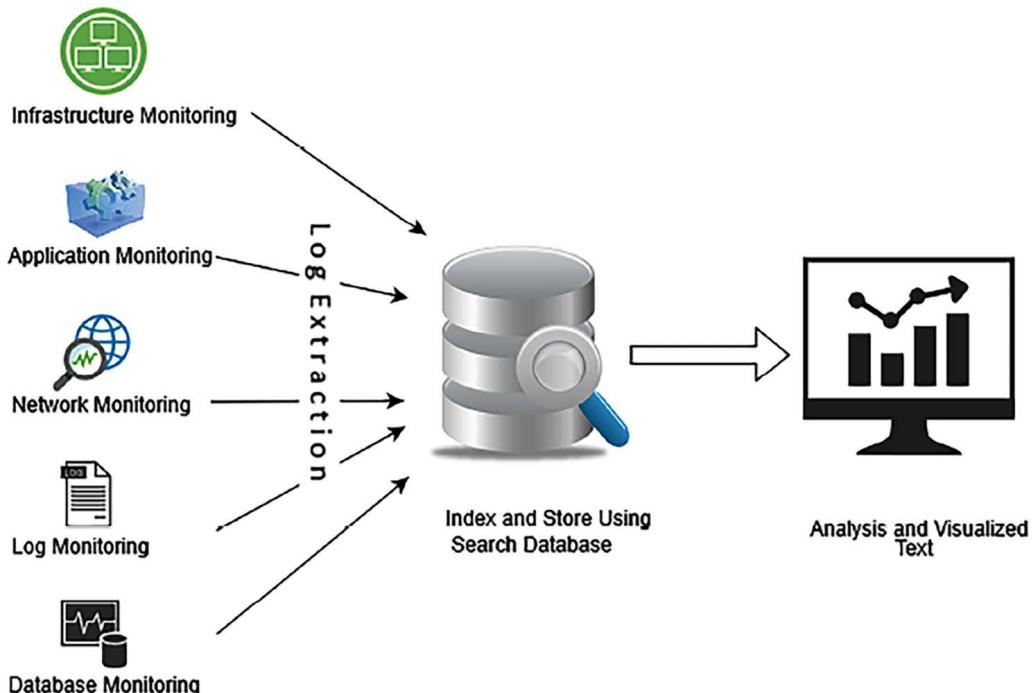


Figure 9.11: Big data approach for ITOA

To create an operations analytics system, you can use scalable big data storage such as Amazon Simple Storage Service (S3). You can also store data in an on-premises Hadoop cluster. For data extraction, an agent, such as the Amazon CloudWatch agent, can be installed on each server, which can send all monitoring data to a centralized storage system. Third-party tools such as ExtraHop and Splunk can help to extract data from various systems.

Once data is collected in centralized storage, you can perform data transformation to convert the data, making it ready for search and analysis. Data transformation and cleaning can be achieved using big data applications like Spark, MapReduce, AWS Glue, etc.

To visualize the data, you can use any business intelligence tool such as Tableau, MicroStrategy, Amazon QuickSight, and so on. Here, we are discussing building an Extract, Transform, and Load (ETL) pipeline. You will learn more details in *Chapter 12, Data Engineering for Solution Architecture*.

You can further perform machine learning to do predictive analysis on a future event. You will learn more about machine learning in *Chapter 13, Machine Learning Architecture*.

Root Cause Analysis

For continuous improvement, you want to prevent any errors from happening again. If you can identify problems correctly, an efficient solution can be developed and applied. It's essential to get to the root cause of the problem to fix the problem.

Five whys is a simple yet effective technique to identify the root cause of a problem. In the *five whys technique*, you gather the team for a retrospective look at an event and ask five consecutive questions to identify actual issues. Take an example of where data must appear in your application monitoring dashboard, but isn't currently. You will ask five whys to get to the root cause.

Problem: The application dashboard is not showing any data.

1. Why: Because the application is unable to connect with the database
2. Why: Because the application is getting a database connectivity error
3. Why: Because the network firewall is not configured to the database port
4. Why: Because the configuring port is a manual check, and the infrastructure team missed it
5. Why: Because the team doesn't have the tools for automation

Root Cause: Manual configuration error during infrastructure creation.

Solution: Implement a tool for automated infrastructure creation.

In the preceding example, at first glance, the issue looks like it is related to the application. After the *five whys* analysis, it turns out to be a bigger problem, and there is a need to introduce automation to prevent similar incidents.

RCA helps the team to document lessons learned and continuously build upon them for operational excellence. Ensure you update and maintain your runbook – as you will code and share best practices across the team.

Auditing and reporting

Auditing is an essential activity to identify any malicious activity in the system by internal or external interference and create recommendations to help resolve this. An audit becomes especially important if your application must comply with regulatory body requirements—for example, PCI, HIPAA, **Federal Risk and Authorization Management Program (FedRAMP)**, and the **International Organization for Standardization (ISO)**. Most regulatory bodies need to conduct regular audits and verify each activity going on in the system to prepare a compliance report and grant a certificate.

An audit is essential to prevent and detect security events. A hacker may silently get into your system and systematically steal information without anyone noticing. Regular security audits can uncover a hidden threat.

Consider conducting a regular audit for cost optimization to identify if resources are running idle when not required. Also, determine resource demand and available capacity so that you can plan.

An IT audit makes sure you safeguard IT assets and license protection and that you ensure data integrity and operations adequately to achieve your organizational goal.

The screenshot below illustrates a data audit that has been stored in an Amazon S3 bucket, utilizing Amazon Macie. Amazon Macie is a data security and privacy service powered by machine learning and pattern-matching technologies. It is specifically designed to detect and safeguard sensitive data within AWS environments.



Figure 9.12: Data audit report summary from Amazon Macie

The data audit report in the preceding screenshot shows data accessibility, encryption, and data sharing reports along with data storage and size details.

Auditing steps include planning, preparing, evaluating, and reporting. Any risk items must be highlighted in the report, and follow-ups should be conducted to address open issues.

Achieving operational excellence in the public cloud

A public cloud provider such as AWS, GCP, or Azure provides many inbuilt capabilities and guidance to achieve operational excellence in the cloud, for example, cloud providers advocate automation, one of the most essential factors for operational excellence.

Taking the example of the AWS cloud, the following services can help to achieve operational excellence:

- The following AWS services help you in the **planning** phase:
 - **AWS Trusted Advisor:** AWS Trusted Advisor checks your workload based on prebuilt best practices and provides recommendations to implement them.
 - **AWS CloudFormation:** With AWS CloudFormation, the entire workload can be viewed as code, including applications, infrastructure, policy, governance, and operations.
 - **AWS Systems Manager:** AWS Systems Manager provides the ability to manage cloud servers in bulk for patching, updates, and overall maintenance.
- The following AWS services help you with the **functioning** phase:
 - **Amazon CloudWatch:** CloudWatch provides hundreds of inbuilt metrics to monitor workload operation and trigger alerts as per the defined threshold. It provides a central log management system and triggers an automated incident response.

- **AWS Lambda:** This AWS service can be used to automate responses to operational events.
- The following AWS services will help you with the **improving** phase:
 - **Amazon OpenSearch:** OpenSearch can be used to analyze log data to gain insight and use analytics to learn from experience.
 - **AWS CodeCommit:** You can share learning with libraries, scripts, and documentation by maintaining them in the central repository as code.

AWS provides various capabilities to provision your applications and infrastructure as code. These capabilities help you to automate operations and incident response. With AWS, you can easily replace failed components with a good version and analyze the failed resources without impacting the production environment.

On AWS, you can collect and combine logs from system operations, workload activities, and infrastructure to create a comprehensive history of activities, a task effectively accomplished using services like AWS CloudTrail. Utilizing AWS tools, you can then query and analyze these operational logs over time. This analysis can help you identify areas for improvement and enhance your system's efficiency and security. In the cloud, resource discovery is easy, as all assets are located under the API- and web-based interfaces within the same hierarchy. You can also monitor your on-premises workload from the cloud. For security auditing in the AWS cloud, Amazon GuardDuty and Amazon Detective provide excellent insight and details across multiple accounts.

Operational excellence requires ongoing commitment. Each operational failure should be thoroughly analyzed to enhance the performance and reliability of your application. This process involves understanding the specific demands and characteristics of your application's load and adapting your operational strategies accordingly. Further, by documenting regular activities as a runbook, following steps to guide issue handling, using automation, and creating awareness, your operations will be ready to deal with any failure event.

Driving efficiency with CloudOps

CloudOps refers to processes, tools, and best practices for efficiently operating and managing cloud environments. Benefits of CloudOps include improved efficiency, reduced costs, better security and compliance, faster recovery from failures, and the ability to scale rapidly.

The key pillars of CloudOps, applicable across cloud providers, are:

- **Set up governance:** Implement a secure, well-architected environment. Utilize tools like AWS Organizations, Azure Management Groups, or Google Cloud Resource Manager for account organization and governance. Enforce policies with tools such as AWS Control Tower, Azure Blueprints, or Google Cloud's Policy Intelligence.
- **Enable compliance:** Continuously monitor configurations with tools like AWS Config, Azure Policy, or Google Cloud Security Command Center. Automate compliance checks and remediation to align with industry standards.

- **Provision and orchestrate:** Accelerate environment setup using infrastructure-as-code with tools such as AWS CloudFormation, Azure Resource Manager templates, or Google Cloud Deployment Manager. Employ tools like AWS Service Catalog, Azure Service Catalog, or Google Cloud Service Catalog to manage standardized IT service portfolios.
- **Monitor and observe:** Ensure observability using tools such as AWS CloudWatch, Azure Monitor, or Google Cloud Operations Suite. Quickly identify and troubleshoot issues to maintain system performance and reliability.
- **Centralize operations:** Manage your infrastructure at scale using tools like AWS Systems Manager, Azure Automation, or Google Cloud Operations for automation and centralized management, enhancing operational efficiency.
- **Manage costs:** Control and optimize expenses with tools such as AWS Cost Explorer, Azure Cost Management, or Google Cloud Cost Management. Set budgets, monitor spending, and detect anomalies to keep costs in check.

By aligning CloudOps practices, you can maintain a consistent and efficient operational framework, regardless of the cloud environment.

Automation is the backbone of CloudOps. It helps organizations manage complex cloud environments more efficiently and with fewer errors. For example, infrastructure changes, which can be error-prone if done manually, are automated through AWS CloudFormation or a similar tool, ensuring consistency and speed. When performance issues are detected by monitoring tools like AWS CloudWatch, automated actions can be triggered to address these issues without manual intervention.

Adopting CloudOps is a journey that starts with foundational governance and compliance. For example, a digital marketing agency may start by securing their cloud environments in alignment with best practices before moving towards full automation of their deployment pipelines. As digital marketing agencies grow, cross-team collaboration becomes crucial for sharing best practices and tools. By starting with governance and compliance and gradually adding automation, teams can manage costs effectively and scale operations efficiently.



Here, we took the AWS example, but the same concept is applicable to any public cloud like GCP and Azure.

With CloudOps, the entire life cycle of building, deploying, monitoring, and operating cloud environments is streamlined, paving the way for agile development and operational excellence.

To learn more details about CloudOps, you can refer to our other book, *AWS for Solutions Architects*.

Summary

Operational excellence can be achieved by continuously improving according to operational needs and lessons learned from past events. You can achieve business success by increasing the excellence of your operations. Focus on developing and managing applications in a way that boosts efficiency and ensures highly responsive deployments. Implementing best practices in your workloads is key to achieving operational excellence.

In this chapter, you learned about the design principles to achieve operational excellence. These principles advocate operation automation, continuous improvement, incremental approaches, predicting failure, and being ready to respond.

You learned about various phases of operational excellence and corresponding technology choices. In the planning phase, you learned about ITAM to track the inventory of IT resources and identify dependencies between them using configuration management.

You learned about alerts and monitoring in the functioning phase of operational excellence and you considered various kinds of monitoring, including infrastructure, application, log, security, and platform monitoring. You learned about the importance of alerts and how to define alert severity and respond to it.

During the improvement phase of operational excellence, you learned about analytics in IT operations by building a big data pipeline for analytics, methods to perform RCA using the *five whys*, and the importance of auditing to save the system from any malicious behaviors and unnoticed threats.

You learned about operational excellence in the cloud and different inbuilt tools that can be utilized for operational excellence in the AWS cloud. Finally, you learned about CloudOps and how it can help you to streamline cloud operations.

As of now, you have learned best practices in the areas of performance, security, reliability, and operational excellence. In the next chapter, you will learn about best practices for cost optimization. You will also learn about various tools and techniques to optimize overall system costs and how to leverage multiple tools in the cloud to manage IT expenditure.

Join our book's Discord space

Join the book's Discord workspace to ask questions and interact with the authors and other solution architecture professionals: <https://packt.link/SAHandbook>



10

Cost Considerations

In the previous chapter, you learned about operational excellence and the use of automation to optimize post-production operations, resulting in reduced human error, increased efficiency, and, ultimately, cost savings. Optimizing the cost of your architecture is a vital aspect of maintaining an efficient and sustainable IT environment. This involves understanding and managing the resources that your applications consume and ensuring that you're only paying for what you need. In this chapter, we'll explore various strategies for cost optimization, including right-sizing resources, selecting appropriate pricing models, and utilizing budgeting and cost management tools.

One of the primary goals of any business is to increase profitability while serving customers. Cost is a crucial topic of discussion when a project is initiated. Application upgrades and the addition of new product features heavily depend on the funding available. The product's costs are everyone's responsibility and need to be considered in every phase of the product life cycle (from planning to post-production). This chapter will help you understand the best practices for optimizing costs for your IT solutions and operations.

Cost optimization is a continuous process and needs to be managed carefully without sacrificing customer experience. Cost optimization doesn't mean cost reduction but it reduces business risk by maximizing **return on investment (ROI)**. You will need to understand your customers' needs before planning any cost optimization strategies and act accordingly. Often, if customers are looking for quality, they are willing to pay a higher price.

In this chapter, you will learn about various design principles for the cost optimization of your solution. The cost aspect needs to be considered for every phase and component of the architecture. You will understand the right technology selection to optimize costs at every layer. You will learn about the following best practices of cost optimization in this chapter:

- Design principles for cost optimization
- Understanding techniques for cost optimization
- Driving cost optimization in the public cloud
- Green IT and its influence on cost considerations

By the end of the chapter, you will have learned about various techniques to optimize costs without risking business agility and outcomes. You will have learned different methods to monitor costs and apply governance for cost control. First things first, let's start with the design principles for cost optimization, which will lay the foundation for building a cost-aware architecture.

Design principles for cost optimization

Cost optimization includes increasing business value and minimizing risk while reducing business costs. It would be best if you planned your application costs by estimating the budget and forecasting expenditure. To realize cost savings, you need to implement a cost optimization plan and closely monitor your expenditures.

There are several principles that can help you achieve cost optimization; common design principles for this are covered in the following sections. You will find that all cost optimization design principles are closely related and complement each other. Let's have a look at these.

Calculating the total cost of ownership

Often, organizations tend to overlook the **total cost of ownership (TCO)** and decide based on the upfront cost to acquire software and services, known as **capital expenditure (CapEx)**. While the upfront cost determination is essential, the TCO matters the most in the long run. The TCO includes both CapEx and **operational expenditure (OpEx)**, covering all the dimensions of the application life cycle. The CapEx cost consists of the price that organizations pay upfront to acquire services and software, while OpEx includes the cost of the operation, maintenance, training, and retirement of software applications. Consider both CapEx and OpEx costs to help make more strategic decisions while calculating your ROI in the long run.

For example, when you buy a refrigerator that will run 24/7, you look for an energy-saving rating to keep your electricity bill low. You are ready to pay a higher price upfront as you know the total cost over time will be lower due to the savings on your energy bill. Now, let's take the example of a data center. There is an upfront hardware acquisition cost involved (CapEx). However, the data center setup requires additional ongoing costs (OpEx), which include heating, cooling, rack maintenance, infrastructure administration, security, and so on.

For a typical use case, when you are purchasing and implementing software, consider the following costs to calculate the TCO:

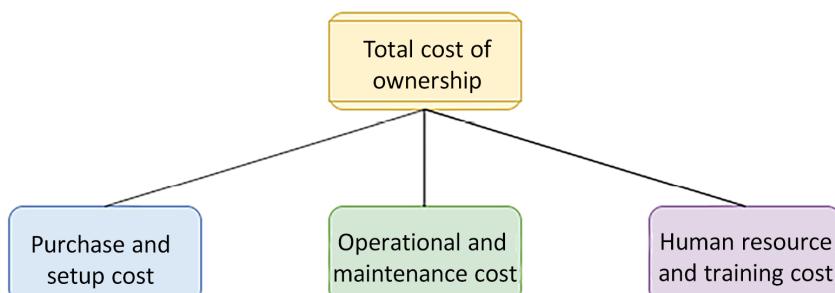


Figure 10.1: TCO for an IT application

Let's look at this at a more granular level. Each TCO component has the following common costs involved for *off-the-shelf* software, such as Oracle or an MS SQL database:



Off-the-shelf software refers to pre-built, mass-produced applications that are designed to cater to a wide audience with similar needs, as opposed to custom software that is tailored to meet the specific requirements of a particular business or user.

- **Purchase and setup costs:** These are the upfront costs to acquire the software and the services to deploy it. This includes the following:
 - Software cost includes the software purchase price with user licenses
 - Hardware costs include purchasing a server and storage to deploy the software
 - Implementation costs consist of the time and effort to get the software ready for production
 - Migration costs include moving data to the new system
- **Operational and maintenance costs:** These are the continuous costs to keep the software running for the business use case, including the following:
 - Software maintenance and support
 - Patching and updates, which software vendors often release to fix any potential bugs
 - Custom enhancements to fit software to your organization's needs
 - Data center cost to maintain the hardware server
 - Security
 - License renewals
- **Human resources and training costs:** These are the overhead costs to train staff to use the software to address business activities. These costs include the following:
 - Application admin staff
 - IT support staff
 - Functional and technical consultants
 - Training and training tools

To optimize costs, you have numerous options, including taking out a subscription for a **Software as a Service (SaaS)** product, such as Salesforce's **customer relationship management (CRM)** platform. The SaaS model is per-user subscription-based primarily, so you need to determine whether you are getting the desired cost savings. For a more significant number of users, you can take a hybrid approach and use the cloud to handle your hardware by choosing an **Infrastructure as a Service (IaaS)** option and installing *off-the-shelf* software. Overall, if the software doesn't meet your requirements, you can build it yourself. In any scenario, calculate the TCO to decide where to make the maximum ROI. Let's look at budget and forecast planning, which can help to control total costs and achieve ROI.

Planning the budget and forecast

Every business needs to plan its expenditures and calculate ROI. Budget planning gives guidance to organizations and teams on cost control. Organizations plan a long-term budget for 1 to 5 years, which helps them run the business based on the funding required. This budget then comes down to the individual projects and applications. During solution design and development, the team needs to consider the available budget and plan accordingly. The budget helps to quantify what the business wants to achieve. The forecast provides an estimate of what the company is doing.

Budget planning is important strategic planning in the long run, and the forecast provides an estimate at a more tactical level to decide the business direction. In application development and operation, without a budget and a forecast, you can quickly lose track and overrun estimated costs. These two terms may be confusing, so let's understand the clear difference between a budget and a forecast:

Aspect	Budget	Forecast
Definition	A detailed financial plan outlining expected revenues, expenses, and resource allocations for a specific period.	An updated projection of a company's financial performance based on current trends and near-term expectations.
Time frame	Typically set for a longer term, such as annually.	More dynamic; updated regularly (monthly or quarterly).
Frequency of adjustment	Adjusted infrequently, maybe once a year or during significant changes.	Updated regularly based on actual business progress and near-term outlook.
Purpose	Used to guide business decisions, strategic planning, and resource allocation.	Used to make informed operational decisions and adjustments based on recent trends and predictions.
Performance evaluation	Used to determine performance by comparing planned versus actual costs and revenues.	Not typically used for performance evaluation against goals but for understanding potential future financial positions.
Examples	Deciding on organization restructuring, planning annual marketing spend, and setting annual sales targets.	Adjusting staffing levels, modifying marketing strategies based on recent performance, and updating revenue expectations.

Table 10.1: Budget versus forecast

Forecast information helps you take immediate action, while the budget may become unachievable due to changes in the market. As shown in the following diagram, while you are working on day-to-day solutions, developments based on historic expenditure forecasts can prompt you to adjust the next month's costs:

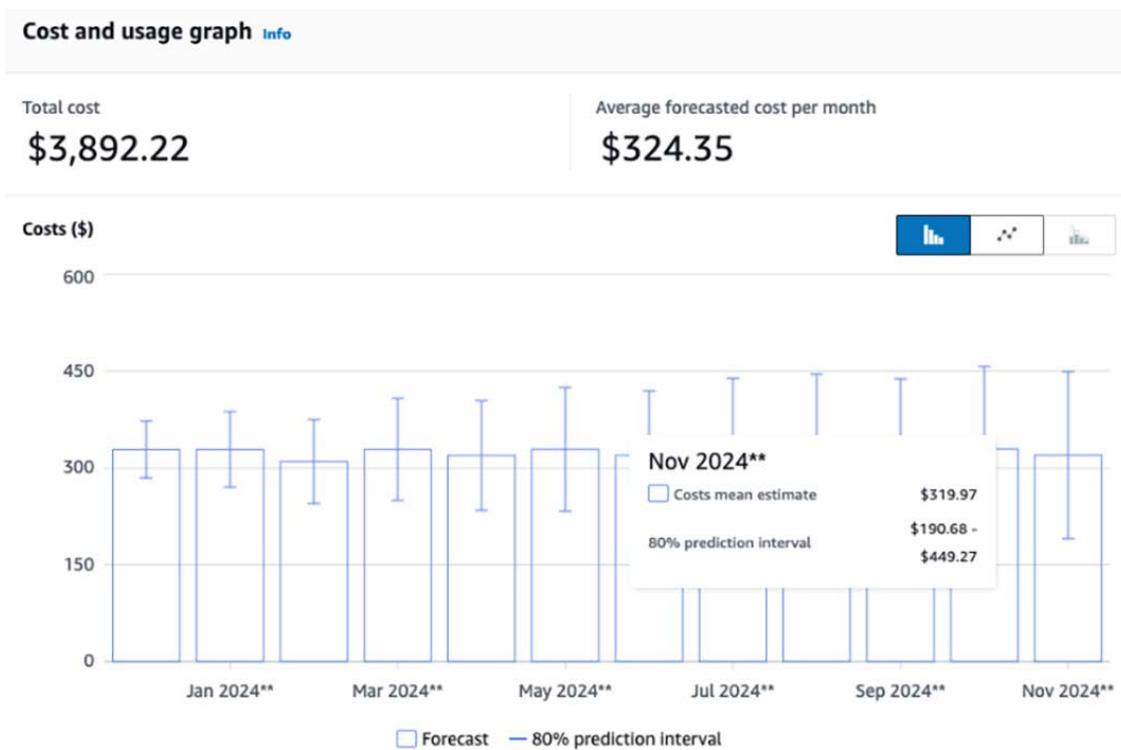


Figure 10.2: Forecast report

As shown in the preceding *Cost and usage* forecast report, if your monthly budget is \$450, you will reach your budget by the end of November 2024. Here, the forecast helps you to act and control costs to stay within budget.

In the next section, let's look at the mechanism to improve cost efficiency by managing demand and services.

Managing demand and service catalogs

Almost every organization has a centralized IT team that works with internal business partners such as application development and support teams of various business units. The IT team manages the demand for IT infrastructure, including the cost of all software and hardware and support to manage application hosting. Business partners often need a greater understanding of the cost drivers for their IT services. For example, application development teams tend to overprovision their development or test environment, resulting in an additional cost.

You can get demand forecasts from various organizational units in advance, which can help you to better align supply of IT Infrastructure across organization. By consolidating all requirements in one place, an organization can benefit from economies of scale. You might achieve a lower variable cost because a large contract can achieve higher economies of scale. The right demand from all organization units is aggregated, translating into lower prices.

For instance, when leveraging public cloud services such as AWS, GCP, or Azure, businesses have the opportunity to secure more favorable pricing through **private pricing agreements (PPAs)** or **enterprise discount programs (EDPs)**. These agreements are especially beneficial for organizations that commit to larger workloads by consolidating their resources within a single cloud provider. By committing to a certain level of usage or spending, companies can negotiate lower prices, translating into significant cost savings.

Organizations can take one of the following two approaches to manage demand and services:

- **Demand management:** To save costs in your existing IT environments (where you may observe that overspending is prevalent), you can take the *demand-led approach*. It will help in improving cost efficiency in the short term as you introduce a few new services. You can analyze historical data to understand factors that are driving demand and capture cases of overprovisioning. Establish a process between the IT team and business partners to streamline operational costs for IT services.
- **Service catalog management:** If there is a demand for new services and you don't have much historical data, you can take the *service-led approach*. In this approach, you need to understand the demand for the most frequently used services and create a catalog. For example, suppose the development team asks for a Linux server with a MySQL database to create a dev environment. In that case, the IT team can create a service catalog that helps the dev team acquire a small Linux server and a database server. Similarly, the IT team can identify the most common set of services and attach a granular cost.

Each approach can have significant cost savings in the short and long term. However, these transformations present substantial challenges as you need to change the project planning and approval process. The business and finance teams need to align and understand the clear relationship between business growth and increased IT capacity. The cost model needs to be built around the most efficient approach by combining cloud, on-premises, and off-the-shelf offerings.

Keeping track of expenditure

You can find the costs of individual systems by tracking expenditures and linking them to the system or business owner. Transparent expenditure data helps identify ROI and reward owners, optimizing resources and reducing cost. It can help you determine the monthly costs for a department or project.

Cost-saving is a shared responsibility, and you need to have a mechanism to hold everyone accountable. Often, organizations introduce a **show-back** or **charge-back** mechanism to share cost responsibility between organizational units.

The centralized billing account informs each organizational unit regarding their expenditure in the **show-back approach** but doesn't charge the actual amount. Each business unit within an organization manages its budget under a master payee account in the **charge-back approach**. The master account charges back the amount to the business units per their monthly IT resource consumption. These approaches make each organizational unit more cost-aware and responsible in their expenditures.

When implementing cost control for your organization, it is better to start with show-back as a stepping stone and move to charge-back as the organizational model matures.

For each business unit, you should create expenditure awareness by configuring notifications so that teams get an alert as they approach the forecasted or budgeted amount of consumption. Create a mechanism to monitor and control your costs by appropriately assigning them to the right business initiative. Provide visibility to create accountability for cost expenditures for each team. Cost tracking will help you understand team operations.

Each workload is different; you should use the pricing model that suits your workload to minimize costs. Establish mechanisms that ensure business objectives are achieved by applying cost optimization best practices. You can avoid overspending by defining a tagging strategy to link business units with specific expenditures and using the **check-and-balance** approach.

Continuous cost optimization

If you follow cost optimization best practices, you should have a good cost comparison with existing activity. It's always possible to reduce the cost of your migrated and matured applications over time. Cost optimizations should not end until the cost of identifying money-saving opportunities exceeds the amount of money you will save. Until that point is reached, you should continually monitor your expenditure and look for new ways to save on costs. You should keep finding an area to save costs by removing idle resources. For instance, consider a scenario where a company uses cloud services for its development, testing, and production environments. By continuously monitoring its cloud expenditure, the company identifies that several instances in the development environment remain underutilized or idle, especially during off-peak hours. To save costs, the company implements a schedule to shut down these instances automatically when they are not in use, such as during nights and weekends. This approach significantly reduces their cloud expenditure as they only pay for the resources when they are actively being used. Over time, this practice of identifying and eliminating idle resources can lead to substantial cost savings, allowing the company to allocate its budget more effectively.

For a balanced architecture in terms of its cost and performance, ensure that the cost paid for resources is well utilized and avoids significantly underutilized IT resources such as server instances.

A biased utilization metric showing exceptionally high or low costs will harm your organization's business. When organizations evaluate utilization metrics without considering the context, it can lead to skewed interpretations and decisions. For instance, basing infrastructure provisioning on data from peak periods like Black Friday for an e-commerce site can result in significant overprovisioning. During less busy times, this leads to substantial underutilization of resources, which is not cost-effective. It's crucial to analyze these metrics within the proper context to avoid such pitfalls and ensure that resources are allocated efficiently, aligning with actual demand and not just peak or atypical usage periods. This approach prevents unnecessary expenditure and supports more strategic, effective cost management.

Application-level metrics for cost optimization need to be considered carefully. For example, introduce archival policies to control data storage capacity. To optimize the database, you should check for appropriate database deployment needs, such as whether multi-location deployments for the database are essential or whether provisioned **input/output operations per second (IOPS)** are applicable as per your database utilization needs. You can use the SaaS model to help your employees focus on applications and business activities to reduce your administrative and operational overhead.

To identify a gap and apply necessary changes for cost-saving, you should implement resource management and change control processes during the project life cycle. The aim is to help your organization design the architecture as optimally and cost-effectively as possible. Keep looking for new services and features that directly reduce your costs.

In this section, you explored various design principles aimed at cost optimization, starting from effective budget planning to proactive cost monitoring and continuous cost-saving strategies. These principles are essential for ensuring that your architecture not only meets performance and operational requirements but also aligns with financial goals, enabling your organization to maximize the value of its investments and minimize unnecessary expenditures. Let's learn some techniques that can help you optimize costs and increase ROI.

Understanding techniques for cost optimization

Enterprises are investing more in technology to gain a competitive edge and keep up with rapid growth. With economic instability, cost optimization becomes an essential but challenging task. Companies spend a lot of time researching and reducing costs in procurement, operations, and vendors. Many companies even share data centers, call centers, and workspaces as a cost-saving method. Sometimes, organizations need more time to upgrade to avoid buying new, expensive hardware.

An organization can save more by looking into its IT architecture across the organization. Improving existing architecture can open doors to bring more opportunities and business to the company, even if it requires a bit of adjustment to the budget. Let's identify focus areas where companies can save money and gain more revenue with techniques such as moving to the cloud, simplified architecture, virtualization, and shared resources.

Reducing architectural complexity

Organizations often need a centralized IT architecture, resulting in each business unit trying to build its own set of tools. Lack of overall control causes a lot of duplicate systems and data inconsistency. IT initiatives in individual business units are driven by a short-term goal.

In such cases, business units need to be better aligned with long-term organizational vision, such as the digital transformation of the entire organization. Furthermore, it adds complexity to maintaining and upgrading those systems. A simple step to define standards and avoid duplication can help save costs.

The following diagram shows a complex architecture on the left-hand side, where business units work in their application without standardization, causing duplicate applications with many dependencies—this kind of architecture results in high costs and risks. Any new experiment takes a long time to market, which results in losing the competitive edge. Process standardization, the development of reusable architecture patterns, and the establishment of a suite of shared services collectively provide a comprehensive and flexible framework that supports an agile environment. By applying automation within this framework, organizations can streamline operations, avoid redundant efforts, and significantly reduce costs. This holistic approach not only enhances operational efficiency but also boosts ROI, demonstrating a strategic alignment of architectural practices with business objectives for optimal financial performance.

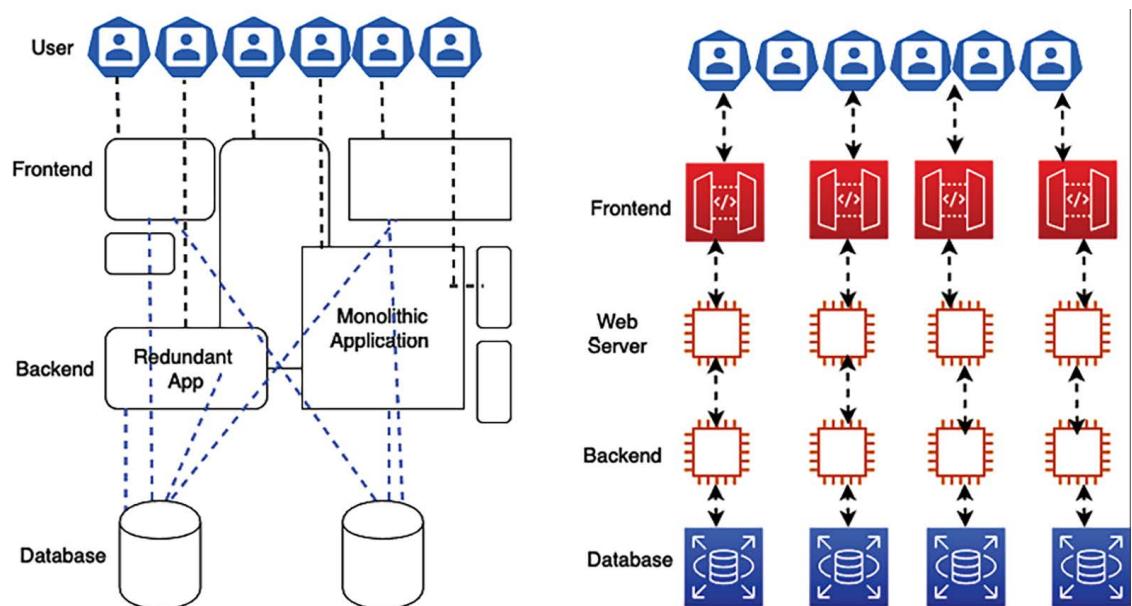


Figure 10.3: Architectural standardization

The first thing is to eliminate duplication and identify function reuse across the business unit to reduce the architectural complexity. Incorporating all these reusable components across the organization into a service catalog can significantly enhance accessibility and efficiency. A service catalog acts as a centralized repository where teams can find and leverage predefined architecture patterns, services, and resources. During the *gap analysis* of existing architecture, you will find there is so much code, so many existing components, and a project that can be reused across the organization to support your business requirements. To reduce the complexity of IT architecture, think of an out-of-the-box solution that fits your business needs and provides ROI. Customization should be your last approach if no other option is available.

Any new application needs to have an accessible integration mechanism to interact with the existing system using **RESTful architecture**. Harmonizing the UI design across the application provides a set of standard UI packages that can be reused for any new application.

Similarly, other modules can be reutilized with service-oriented design. You learned about RESTful patterns in *Chapter 4, Solution Architecture Design Patterns*; these help you keep all the different pieces of software working separately and still communicating with each other to build an entire system.

In the modular approach, each team is responsible for developing a service that every team across the organization can use to avoid duplication. As an architect, you should help the team create a service-oriented design, where each team handles individual architecture components as a service that can be developed independently. With the help of microservices architecture, you can deploy an entire application modularly. If one component is not working, you can rework it without impacting the whole application. For example, a payment service developed to collect payment from a customer visiting an e-commerce website can also be used to pay vendors in the vendor management system.

Once you set up a centralized IT architecture, taking a modular approach helps you keep the cost down. Empowering your IT architecture team can help align organizational units with the company's vision and support other parallel projects to follow the overall strategy. It also helps to provide consistency in other critical services that are often overlooked, such as legal, accounting, and human resources.

With the help of the IT architecture team, you can get excellent feedback and ensure that projects are aligned with business needs and requirements. By overseeing the overall architecture across teams, an architect can advise whether any duplicate effort, project, process, or system needs to be aligned with the business need. The centralized architecture will reduce complexity and tech debt, bring more stability, and increase quality. The overall idea of centralized architecture is to increase IT efficiency, so let's learn more about that.

Increasing IT efficiency

Nowadays, every company uses and consumes IT resources. Having too many servers, laptops, and software licenses and a high storage capacity consumes a lot of funding. Licenses are one of the resources that are sometimes underused, undiscovered, idle, or installed incorrectly and consume a lot of funding. A centralized IT team can lead the effort for license optimization by keeping track of used software licenses and retiring additional licenses. They can save costs by negotiating a bulk discount with the vendor.

To increase IT efficiency, cancel non-compliant projects that take additional funding and resources. Also, you should help teams revisit the strategy to support or continuously terminate unused and non-aligned projects.

The following methods can be considered for cost optimization:

- Re-evaluate projects with high costs, as they may need to be better aligned with the business vision. Reshape projects with high value but no direct impact on the IT strategy.
- De-prioritize projects with little to no business value, even if aligned with the IT strategy.
- Cancel non-compliant projects with low business value.
- Decommission or retire unused applications.
- Replace old legacy systems by modernizing them to reduce maintenance costs.
- Avoid duplicate projects by reutilizing existing applications.
- Wherever possible, consolidate data and develop an integrated data model. You will learn about maintaining a centralized data lake in *Chapter 12, Data Engineering for Solution Architecture*.
- Consolidate vendor procurement across the organization to save IT support and maintenance expenditure costs.
- Consolidate any system that does the same thing as payment and access management.
- Eliminate costly, wasteful, overprovisioned projects and expenditures.

Moving to the cloud can be an excellent consideration for increasing IT resources efficiently and reducing costs. Public cloud providers, such as **Amazon Web Services (AWS)**, offer a *pay-as-you-go* model, which means you only pay for what you are using. For example, the developer desktop can shut down during non-working hours and weekends, reducing workspace costs by up to 70%.

The batch processing system needs to be spin-up only to process jobs and can be shut down immediately afterward. It works just like any electrical appliance that you switch off when not required to save electricity costs.

Applying automation is a great mechanism to increase overall IT efficiency. Automation helps eliminate costly human labor and reduces the time spent performing daily routine jobs without error. Automate things wherever possible to provision servers, run monitoring jobs, and process data.

Make sure to make a suitable trade-off to improve results while deciding to optimize costs. Let's take an example. If you go to a theme park where you want to go on lots of good rides, you are willing to pay a higher price to see the value of your spending. To attract more customers, if the vendor decides to reduce the price and make adjustments by lowering the number of enjoyable rides, you may go to another theme park as you are looking for a good time. Here, competitors will gain an edge and attract existing customers, while the current vendor will lose business. In this case, cost reduction adds business risk, which is not the right cost-optimization approach.

Your goal should be measurable, and these measures should focus on both business output and the cost of your system. Quantitative measures help you understand the impact of increasing output and reducing costs. The organizational and team-level goals must align with the application's end users. At the organizational level, the goals will be across organizational business units. At the team level, they will be more aligned with individual systems. For example, you can set up a goal at the business unit level to reduce the cost per transaction by 10% every quarter or 15% every six months. Defining goals ensures that systems improve over their lifetime. Let's look at how we can apply standardization and governance.

Applying standardization and governance

Organizations need a strategy to analyze misalignment and overconsumption, reduce complexity, define guidelines to use appropriate and efficient systems, and implement a process wherever it is required. Creating and implementing these guidelines will help companies develop a standard infrastructure and reduce duplicate projects and complexity.

To implement governance, you must set up resource limits across the organization. Putting the service catalog in place with **Infrastructure as Code (IaC)** helps ensure teams are not overprovisioned with resources beyond their allocated capacity. You should have a mechanism to quickly understand and take action on business requirements. Consider resource creation and decommissioning when applying resource limits and defining the process to change them.



With IaC, the entire infrastructure setup from networks and servers to databases and application services can be defined in code files using languages like YAML or JSON. These files can then be version-controlled, allowing teams to track changes over time, roll back to previous configurations, and apply the same configurations across different environments, ensuring consistency and reducing configuration drift. Popular tools like Terraform, AWS CloudFormation, and Ansible enable IaC, allowing teams to define IaC and apply these definitions to create or change infrastructure.

Businesses operate multiple applications by various teams. These teams can belong to different business units within their revenue stream. Determining resource costs to the application, business unit, or team drives efficient usage behavior and helps reduce costs. You can define resource capacity based on cost attribution and the group, organization unit, or department requirements. You can use resource tagging and account structuring to organize the cost structure.

As shown in the following screenshot, you can organize your accounts in different **organization units (OUs)**, such as HR and Finance, and each department under the OU can have its own accounts. For example, here, HR has separate accounts for **Payroll** and **Marketing**, while Finance has individual accounts for **Sales** and **Marketing**:

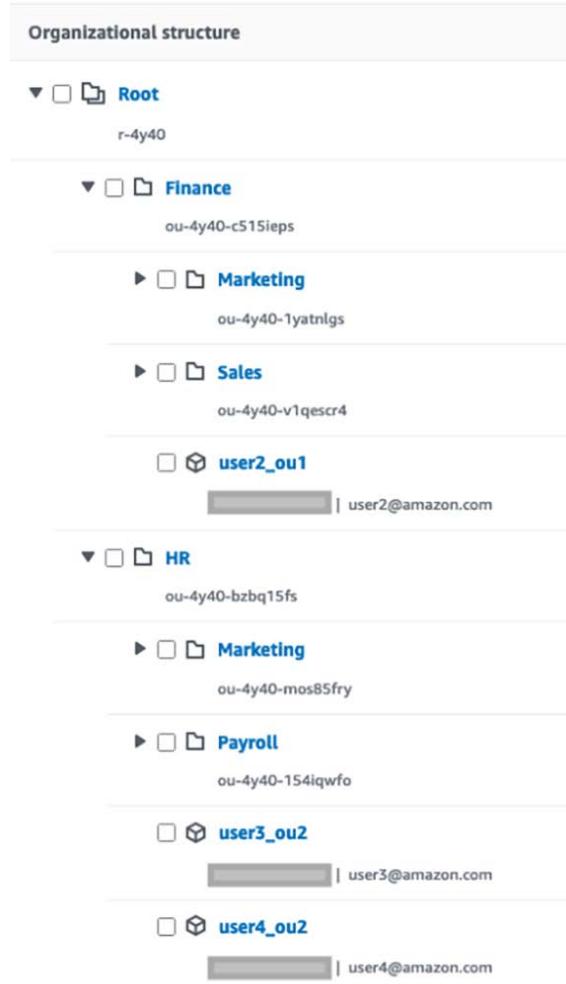


Figure 10.4: Enterprise account structure for OUs

You can control costs at each business unit and department level in the preceding account structuring strategy. Adopting a charge-back mechanism for each department increases accountability for costs at a more granular level, which helps to optimize costs.

Account structuring helps you to apply high security and compliance standards across the organization. As each account is linked to a parent account, you can significantly deal with the mass utilization of vendor resources by consolidating expenditures across the organization.

Resource cost tagging

Almost every public cloud provider provides tagging capabilities out of the box. You can embed server metadata such as the DNS name or hostname for on-premises. Tagging helps you organize costs and define a capacity limit, security, and compliance. It can be an excellent tool for inventory management and keeping an eye on the growing need for resources at every level of the organization.

Resource tagging is a powerful strategy for managing and optimizing costs in cloud computing environments. It involves assigning metadata tags to cloud resources, such as virtual machines, storage instances, or databases, to categorize and track them based on various criteria like project, environment, department, or cost center.

As shown in the following screenshot, to get complete cost visibility and consolidation across resources, you can tag each resource provisioned at the team level, which provides more granular control:

Key (128 characters maximum)	Value (256 characters maximum)
Type	AppServer
Environment	Dev
Department	Marketing
Business Unit	Finance
Add another tag (Up to 50 tags maximum)	

Figure 10.5: Resource tagging for cost visibility

In the preceding diagram, you can see the tagging strategy, which indicates that the given server is for application deployment and is utilized by the development team. The Marketing department of the Finance business unit owns this server. In this way, the organization can get a granular level of cost expenditure visibility, and the team will be more frugal in their spending. However, you can adopt the show-back mechanism at the team level and charge-back mechanisms at the department and business unit levels.

You can define your mechanism for tagging, where you can attach a name and value such as resource name and owner name to any resource. The following screenshot shows the costs sorted by the **aws:createdBy** tag, which helps to determine the cost of each resource that is auto-created by AWS:

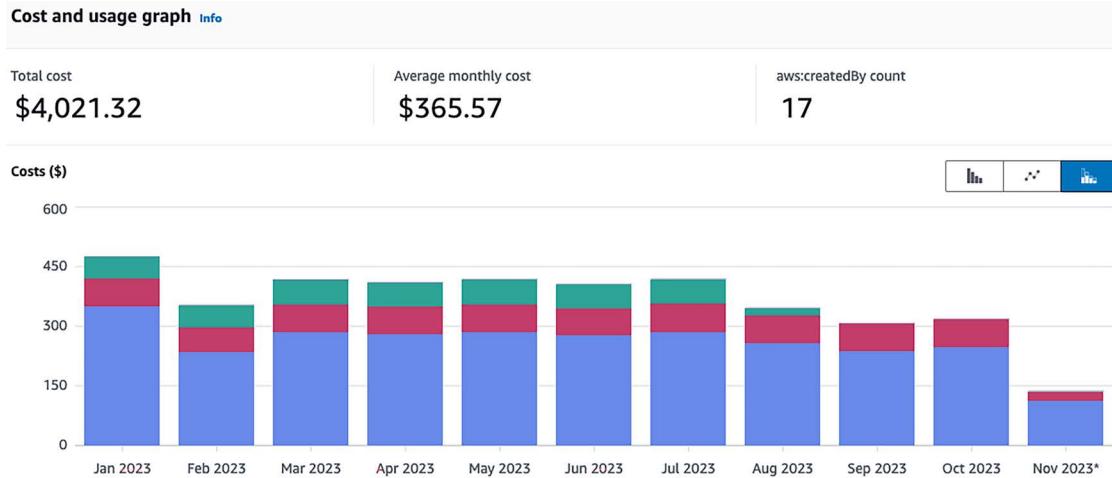


Figure 10.6: Resources expenditures dashboard for a cost tag

Business leaders should evaluate the overall requirements to create efficient IT architectures. Collaboration is required to develop a robust IT architecture and define governance across functional teams to set up accountability. Also, set up a standard to review the architecture, create the baseline for any new project initiative, and explain the process to ensure that the system complies with the correct architecture, and identify the route to improvement.

Engage all impacted stakeholders within your business in usage and cost discussions. The CFO and application owners must understand resource consumption and purchasing options. Department owners must understand the overall business model and the monthly billing process. This will help to set the direction for the business units and the whole company.

Make sure third-party vendors are aligned with your financial goals and can adjust their engagement models. Vendors should provide a cost analysis of any application they own and develop. Each team within the organization should be able to translate business, cost, and usage factors from management into system adjustments, which helps the application implement and achieve the company's desired goals.

Monitoring cost usage and reports

Accurate cost factors help you to determine the profitability of business units and products. Cost tracking helps you allocate resources in the right places to increase ROI. Understanding cost drivers helps you control your business expenditure.

To optimize costs, you must know your expenditure pattern across the organization. You need to have insights into IT expenditure over time to determine cost-saving opportunities. You can take the required steps for cost optimization and understand the impact by creating a visualization of cost trends, which shows historical costs and forecasts by resources and departments across the organization.

Your team needs to gather data by logging all data points, analyze them with monitoring, and create a visualization report.

You need detailed insights into your workload resource utilization to identify cost-saving opportunities. Cost optimization depends on your ability to forecast future spending and implement methods to align cost and usage as per your forecast. The following are the primary areas where you should have data visualizations for cost-saving:

- Determine the most significant investments in resources
- Analyze and understand your expenditure and usage data
- Budget and forecast
- Receive an alert when you exceed your budgeted or forecasted thresholds

The following report shows resource expenditures in AWS over six months. The visualization shows that the cloud computing server EC2, represented by the fifth bar in each month in orange, consumed the highest costs singularly from May 2023 to July 2023. As the business unit can visualize the consistently high cost, it prompts the system admin to look in depth at cost optimization and find idle resources. The admin performed a cleanup by stopping those EC2 servers, which brought down the cost in August and eliminated it completely from September onward:

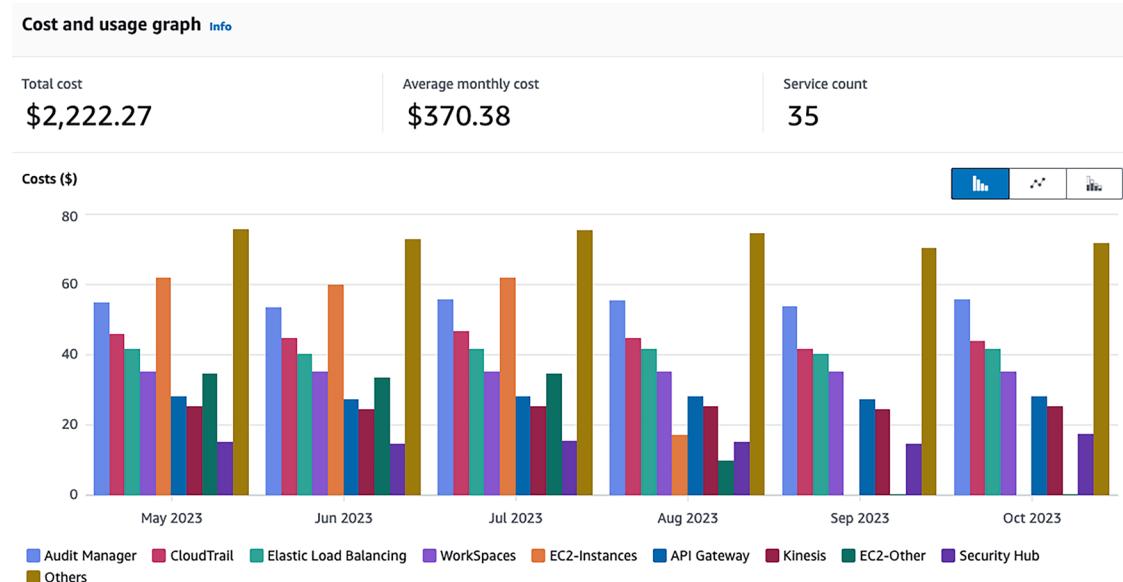


Figure 10.7: Resource cost and usage report by service

The preceding report helped business owners understand cost patterns and take a reactive approach to cost control. The reactive approach caused hidden costs, which went undetected for a specified period. With the proactive approach, the forecast helped business owners to make a decision ahead of time.

The following report shows monthly cost expenditure in the filled bars and forecast spending in the empty bars with estimated ranges. Looking at the report, you can see that it is likely that costs may increase in the next couple of months, and you can take action to understand cost attributes and control costs:



Figure 10.8: Cost trend and cost forecast report

Monitoring your costs against the budget can give you another proactive measure to control costs. Setting up an alert when expenditure reaches a certain proportion of the budget (50% or 80%) helps you review and adjust your ongoing costs.

In the following report, you can visually determine the current cost against the budgeted cost, which was high a year ago. Based on the following report, IT admins could take action to optimize the cost and bring it down to within the monthly budget.



Figure 10.9: Cost and budget report

Cost and budget reports help you to control costs by taking proactive action. Combining your actual running costs with budgets and forecasts provides a great deal of cost control daily.

You can also set up an alert when your actual cost reaches a certain threshold in the budget or forecast, which alerts you proactively via email or SMS and tells you to take proactive action to control costs.

The following screenshot shows that an alert has been set up for when the estimated cost goes above \$500. You can set up multiple alerts to get information when the cost reaches, for example, \$300 and \$400 of the estimated charge:

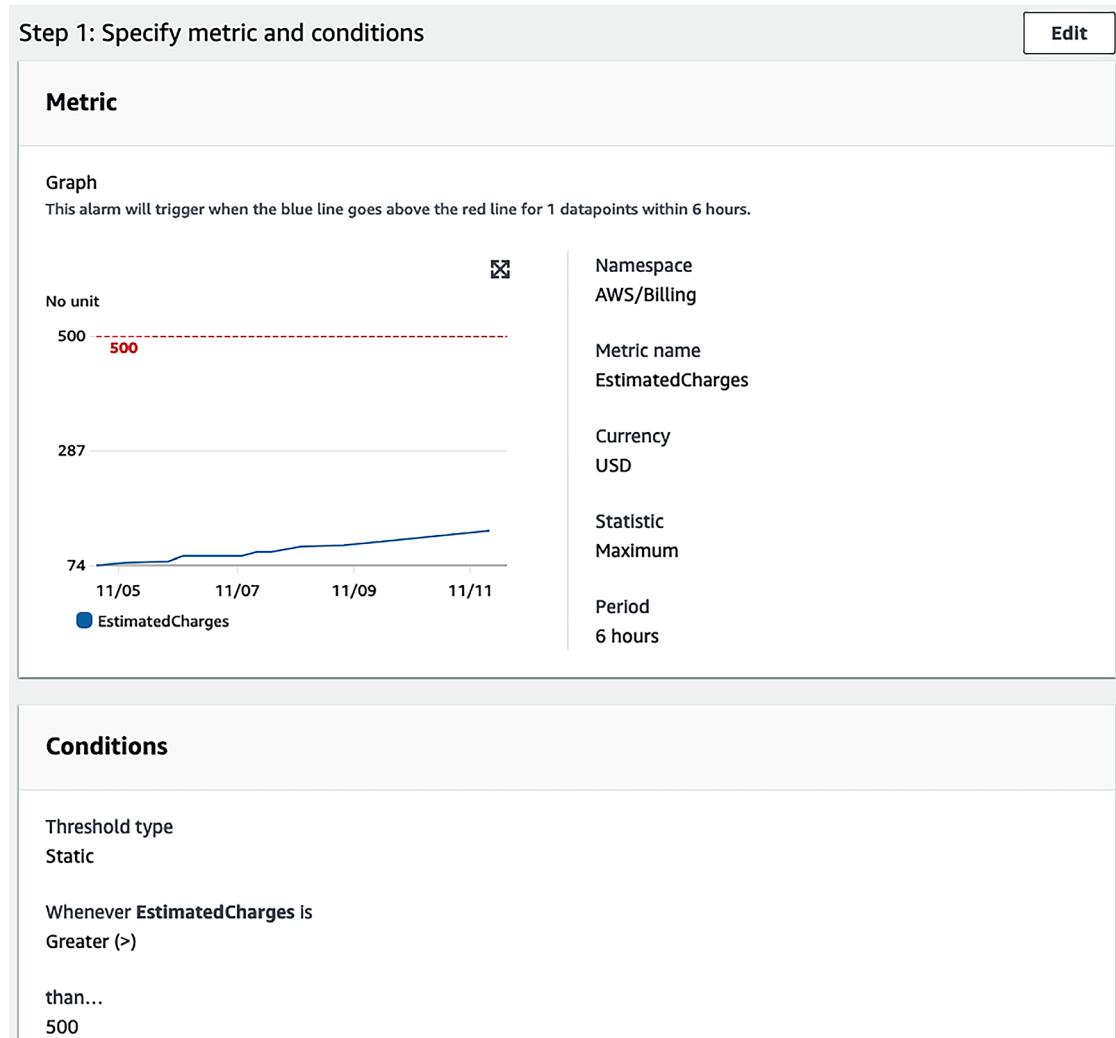


Figure 10.10: Alert based on actual cost

One way to control cost is by right-sizing your environment with resource monitoring and triggering alarms for over- or underutilization. Analysis of resources can be performed using monitoring tools such as **Splunk** or **CloudWatch** and custom logs, where customized metrics such as application memory utilization of your system can be monitored to perform right-sizing. Low resource utilization could be a criterion for identifying opportunities for cost optimization. For example, CPU utilization, RAM utilization, network bandwidth, and the number of connections to the application can be analyzed and monitored.

You must be careful when resizing your environment to ensure you are not affecting the customer experience. The following are best practices to apply when you perform right-sizing:

- Make sure monitoring reflects the end-user experience. Select the correct period. For example, performance metrics should cover 99% of the user's request-response time rather than taking an average response time.
- Select the correct monitoring cycle, such as every hour, day, or week. For example, if you conduct daily analyses, you might miss a weekly or monthly cycle of high utilization and underprovision your system.
- Assess the cost of change against the cost saving. For example, you may have to perform additional testing or engage resources to perform resizing. This cost-benefit analysis will help you assign resources.
- Identify application utilization against your business requirement. For example, look at how many user requests are expected to come by the end of the month or during the peak season. Identifying and optimizing the utilization gap allows you to save costs. To do this, use the right tool that covers all dimensions, from cost-saving to system utilization and the impact on customer experience due to changes, and then utilize reports to understand the business ROI impact due to cost changes. The public cloud follows a different cost model, often an on-demand pay-as-you-go cost structure.

You have to be very diligent when using cloud resources, as every second counts toward your cost, and it can be costly if you overlook cost optimization and monitoring. Let's learn more about cost optimization in the public cloud.

Driving cost optimization in the public cloud

Public clouds, such as AWS, Microsoft Azure, and GCP, provide excellent cost optimization with a *pay-as-you-go* model. This model allows customers to trade CapEx for variable expenses, paying for IT resources as they consume them. OpEx is usually lower due to economies of scale. It can be cost-effective to be in the cloud and benefit from continued price reductions that occur over time. The other advantage is that you get additional tools and functionality out of the box with a cloud provider such as AWS, which helps you achieve better agility.

You need a different mindset when defining the cloud cost structure model, as it differs from traditional cost models, which most enterprises have been following for decades. You have all the infrastructure available at your fingertips in the cloud, which requires greater control and regulation. Clouds provide several tools for cost governance and regularization. For example, in AWS, you can set up service limits for each account so the development team cannot utilize more than 10 servers, and production can have the required number of servers and databases with a buffer.

All resources are associated with accounts in the cloud, so it's easy to keep track of IT resource inventories in a single place and monitor their utilization. In addition to that, you get tools to collect data across various IT resources and provide suggestions. As shown in the following screenshot, AWS Trusted Advisor crawls through all resources in the account and offers cost-saving recommendations based on resource utilization:

The screenshot shows the AWS Trusted Advisor interface under the 'Cost optimization' tab. At the top, it displays 'Potential monthly savings' of \$48.9. Below this, four categories are listed: 'Action recommended' (1 item), 'Investigation recommended' (4 items), 'No problems detected' (17 items), and 'Checks with excluded items' (0 items). The 'Cost optimization checks' section contains two entries: 'AWS Lambda Functions with High Error Rates' and 'Idle Load Balancers'. Both entries include a brief description, a 'Last updated' timestamp, and a refresh button.

Figure 10.11: Cost-saving suggestions from AWS Trusted Advisor

In the preceding screenshot, AWS Trusted Advisor has detected an idle load balancer and advises shutting it down to save monthly costs of up to \$40. Further checks have identified a Lambda function with a high error rate, which needs to be fixed to reduce costs.

The cloud can provide an excellent value proposition for cost-saving. To begin with, you can create a hybrid cloud, where you establish connectivity between your on-premises data center and the cloud. You can move development and test servers to the cloud to determine the cost structure and potential savings. Once you have set up cost governance in the cloud, move more workloads as per the cost-benefit analysis. However, you need to assess your workload and whether it can be moved to the cloud and, if so, define a strategy. You learned about cloud migration in *Chapter 3, Cloud Migration and Hybrid Cloud Architecture Design*.

Increasingly, public cloud providers offer managed services, eliminating any infrastructure maintenance costs and overheads for alerting and monitoring configurations. A managed service reduces the total cost of ownership by reducing cost as service adoption increases.

The public cloud offers savings plans or reserved instances, which allow organizations to commit to a certain amount of usage in exchange for a significantly lower price compared to on-demand rates. By analyzing resource usage data, organizations can make informed decisions about purchasing these plans, aligning their commitments with actual usage patterns to maximize cost savings.

Incorporating savings plans into a cloud cost optimization strategy enables organizations to strike a balance between flexibility and cost efficiency. They can retain the pay-as-you-go model's agility for variable workloads while capitalizing on savings plans for predictable, steady usage, ensuring they optimize their cloud expenditure effectively.

Green IT refers to the environmentally responsible and eco-friendly use of computers and their resources. It encompasses practices like reducing energy consumption, minimizing e-waste, and designing data centers that are less energy-intensive. Let's learn more about it.

Green IT and its influence on cost considerations

Green IT, also known as green computing, refers to environmentally sustainable computing or IT. It is the study and practice of using computers and IT resources in a more efficient and environmentally sustainable way. Green IT practices can significantly influence cost considerations in several ways:

- **Energy efficiency:** Utilizing energy-efficient hardware and practices can reduce the power consumption of data centers and IT infrastructure, leading to substantial savings in electricity costs. For example, using Energy Star-rated equipment or optimizing data center layouts for cooling can lower energy bills.
- **Virtualization:** Virtualizing servers and storage can lead to a reduction in physical hardware needs. This reduces energy consumption and costs associated with cooling and minimizes the space required for data centers.
- **Cloud computing:** Moving to cloud services can be more energy-efficient than maintaining on-premises data centers. Cloud providers often have economies of scale and more modern, efficient, and green data centers. This can translate into lower costs for power and cooling and reduced carbon footprints.
- **Hardware recycling and reuse:** Proper IT equipment recycling and reuse can lead to cost savings. Companies can recoup some of their initial investment by recycling or selling old equipment. Additionally, buying refurbished hardware can be significantly cheaper than new purchases.
- **Telecommuting:** Promoting remote work reduces the need for office space, energy consumption, and employee commuting costs. This can result in direct and indirect cost savings for the organization and employees while also benefitting the environment.
- **Electronic document management:** Reducing paper use through electronic document management systems can lead to cost savings on paper, printing, and storage, as well as the environmental benefits of less paper waste.
- **Sustainable IT procurement:** Choosing suppliers and products that prioritize sustainability can result in long-term cost savings. Products designed to be more durable or with longer extended warranties may have higher upfront costs but a lower total cost of ownership over time.
- **Maintenance optimization:** Regular maintenance and updates can extend the life of IT equipment, delaying the need for costly replacements and reducing electronic waste.
- **IT asset disposal:** Efficiently managing IT asset disposal can reduce waste management costs and ensure compliance with environmental regulations, avoiding potential fines.

- Carbon credits:** By reducing carbon emissions through green IT practices, organizations can earn carbon credits that can be sold or traded, offering a potential revenue stream or savings on carbon taxes.

Green IT can lead to significant cost savings for organizations by reducing energy consumption, minimizing waste, optimizing equipment usage, and improving overall sustainability. These savings often offset the initial investment required to implement green IT initiatives. Let's look at how cloud providers like AWS can help improve your green IT posture.

Cost-effective and green application hosting on AWS

A use case for green IT involving AWS might focus on leveraging AWS's cloud infrastructure to achieve environmental and cost benefits. AWS provides services that support green IT initiatives, such as serverless architectures, energy-efficient data centers, and resource optimization tools. AWS has been proactive in supporting green IT and sustainability goals through various initiatives and services that enable customers to reduce their carbon footprint and optimize energy use.

AWS has launched a customer carbon footprint tool, as shown in the following diagram. This tool provides data visualizations of historical carbon emissions, trends in emissions as the usage of AWS evolves, estimated emissions avoided by using AWS instead of on-premises data centers, and forecasted emissions based on current usage.

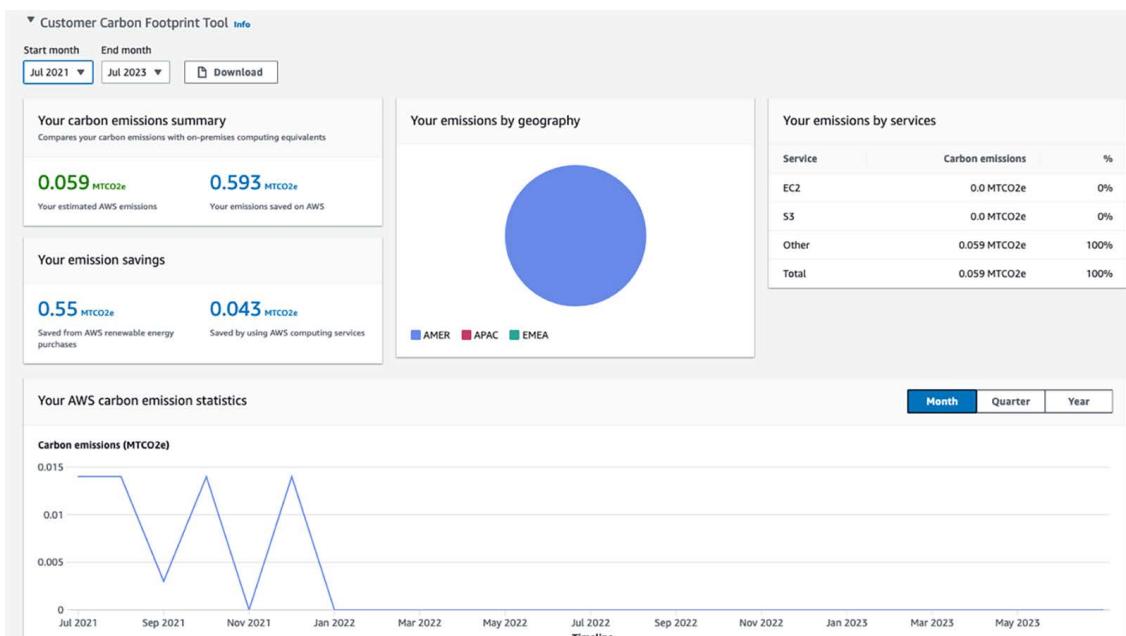


Figure 10.12: Carbon footprint report

The preceding report shows the carbon footprint for the last two years, which shows carbon emission consumption and savings. Carbon footprints will change as AWS moves toward 100% renewable energy by 2025 and net-zero carbon by 2040.



AWS has built comprehensive guidance on building sustainable architecture. You can find more details by visiting the sustainability pillar of the AWS Well-Architected Framework here: <https://docs.aws.amazon.com/wellarchitected/latest/sustainability-pillar/sustainability-pillar.html>.

Suppose you plan to deploy a new web application that expects variable traffic. You can aim to optimize costs and adhere to green IT principles using the following steps:

1. Planning and design:

- The company decided to use AWS due to its commitment to sustainability and the energy efficiency of its infrastructure.
- They opt for a serverless architecture using AWS Lambda for computing, which allows them to run code without provisioning or managing servers. This choice is cost-effective and green since it eliminates the need for underutilized servers.

2. Implementation:

- The web application's frontend is hosted using Amazon S3 and delivered globally via Amazon CloudFront, reducing latency and minimizing the energy required to serve end users by using energy-efficient data centers near the users' locations to serve content like video and images.
- The application's backend is built on AWS Lambda and Amazon DynamoDB, which provides a serverless database solution that automatically scales to match the workload.

3. Optimization:

- For continuous integration and deployment, AWS CodePipeline and AWS CodeBuild are used, which helps to avoid the overhead of maintaining dedicated build servers.

4. Monitoring and management:

- AWS CloudWatch monitors the application's performance and resource utilization, ensuring efficient operation.
- AWS Trusted Advisor recommendations are applied to optimize costs further and ensure the company uses the most efficient resources possible.

5. Cost management:

- The company uses AWS Budgets to set cost thresholds and receives alerts if their costs are likely to exceed the budget, allowing them to adjust usage proactively.
- They use AWS Cost Explorer to analyze and understand their AWS spending and usage over time, ensuring continuous cost optimization.

The company successfully deploys its web application with a scalable and cost-effective infrastructure by leveraging AWS's serverless services and resource optimization tools. They reduce their carbon footprint by utilizing AWS's energy-efficient global infrastructure and only consuming computing power when needed, adhering to green IT principles. Additionally, the pay-as-you-go pricing model of AWS services ensures they only pay for the resources they consume, optimizing both costs and energy usage.

AWS can facilitate green IT practices, allowing you to deploy a scalable, high-performance web application while being mindful of environmental impact and operational costs. AWS's serverless offerings and efficient resource management align with the company's goal of cost optimization and sustainability.

Summary

Cost optimization is a continuous effort from application inception (from proof of concept to implementation and post-production). You need to review architecture and cost-saving efforts continuously.

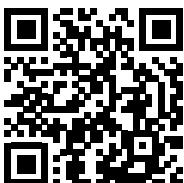
In this chapter, you learned about the various design principles to optimize costs. Before making any purchase decision, you should consider the total cost of ownership for the entire software or hardware life cycle. Budget planning and tracking forecasts help you to stay on the cost optimization path. Always keep track of your expenditures and look for possible opportunities for continuous cost optimization by managing demand without affecting the user experience or business value.

You learned about various cost optimization techniques, including reducing architectural complexity by simplifying enterprise architecture and setting a standard that everyone can follow. It's recommended to avoid duplication by identifying and consolidating idle and repeated resources to negotiate the bulk purchase cost. Apply standardization across the organization to limit resource provision and develop a standard architecture. Tracking data for your actual costs against budgets and forecasts can help you to take proactive action. You learned about various reports and alerts that can help control costs. You also learned about cost optimization in the cloud, which can help you further optimize value. Sustainability is an essential aspect of today's IT workload and you learned about how green IT is influencing cost considerations. You also learned about building and tracking green IT consumption using AWS.

Automation and agility are major factors that increase resource efficiency, and DevOps can provide a great deal of automation. In the next chapter, you will learn about various DevOps components and strategies to efficiently deploy your workload in the most automated way.

Join our book's Discord space

Join the book's Discord workspace to ask questions and interact with the authors and other solution architecture professionals: <https://packt.link/SAHandbook>



11

DevOps and Solution Architecture Framework

In the previous chapter, you explored creating a cost-aware architecture and methods to continually optimize costs without compromising performance. Automation and collaboration across teams are crucial for developing robust applications and saving costs. This chapter will delve into DevOps, a methodology that fosters collaboration between development and operations teams while automating the processes of application deployment and monitoring.

In traditional environments, the development team and the IT operations team work in silos. The development team gathers requirements from business owners and develops the applications. System administrators are solely responsible for operations and for meeting uptime requirements. These teams generally have little direct communication during the development life cycle, and each team rarely understands the processes and needs of the other team.

Each team has its own tools, processes, and redundant approaches, sometimes resulting in conflict. For example, the development and **quality assurance (QA)** teams could test the build on a specific **operating system (OS)** patch. However, the operations team deploys the same build on a different OS version in the production environment, causing issues and delays in the delivery.

DevOps is a methodology that promotes collaboration and coordination between developers and operational teams to deliver products or services continuously. This approach is constructive in organizations where the teams rely on multiple applications, tools, technologies, platforms, databases, devices, and so on in the process of developing or delivering a product or service. Although there are different approaches to the DevOps culture, all are about achieving a common goal. DevOps is about delivering a product or service in the shortest amount of time by increasing operational efficiency through shared responsibilities.

Security is the top priority for any application, and security incidents can have a severe business impact. Despite this, security is often an afterthought in the deployment process, treated as a separate concern handled reactively by specialized security teams within an organization.

Embedding security in the DevOps process as a crucial aspect can be achieved by implementing DevSecOps. DevSecOps is about integrating security early and throughout the software development life cycle, thus breaking down silos and fostering collaboration between development, operations, and security teams. DevSecOps helps deliver without compromising quality, reliability, stability, resilience, or security.

In this chapter, you will learn about the following DevOps topics:

- Introducing DevOps
- Understanding the components of DevOps
- Continuous integration/Continuous deployment (CI/CD)
- Introducing DevSecOps in security
- Combining DevSecOps and CI/CD
- Implementing a CD strategy
- Implementing continuous testing in the CI/CD pipeline
- Using DevOps tools for CI/CD
- Implementing DevOps best practices
- Building DevOps and DevSecOps in the cloud

By the end of this chapter, you will have learned about the importance of DevOps in application deployment, testing, and security. You will also learn about the best practices of DevOps and DevSecOps and their different implementation tools and techniques.

Introducing DevOps

In a **DevOps** (short for **d**evelopment and **o**perations) approach, the development and operations teams work collaboratively during the build and deployment phases of the software development life cycle, sharing responsibilities and providing continuous feedback. The software builds are tested frequently throughout the build phase in production-like environments, allowing the early detection of defects.

DevOps is a combination of culture and practices. It requires organizations to change their culture by breaking down the barriers between all teams in the product development and delivery life cycle. DevOps is not just about development and operations; it involves the entire organization, including management, business/application owners, developers, QA engineers, release managers, the operations team, and system administrators.

Speed enables organizations to stay ahead of the competition and address customer requirements quickly. Good DevOps practices encourage software development engineers and operations professionals to work better together. This results in closer collaboration and communication, leading to a shorter **time to market**, reliable release, improved code quality, and better maintenance.

Sometimes with DevOps, you will find that a software application's development and operations are handled by a single team, where engineers work across the entire application life cycle. Such a team needs to develop a range of skills that are not limited to a single function. Application testing and security teams may also work more closely with the operations and development teams from the inception to the production launch of an application.

Developers benefit from feedback provided by the operations teams and create strategies for testing and deployment.

System administrators don't have to implement defective or untested software in production environments because they participate in the *build phase*. As all software development and delivery life cycle stakeholders collaborate, they can also evaluate the tools they intend to use at each step of the process, verify compatibility between the devices, and determine whether any tools can be shared across the teams.

DevOps is gaining popularity as the preferred operating culture, especially for organizations that leverage cloud or distributed computing technologies. Let's learn about some of the various benefits of DevOps and why it is essential for your application workload.

Understanding the benefits of DevOps

The goal of DevOps is a CI/CD model that can be utilized to make the software development life cycle repeatable, reliable, stable, resilient, and secure. These characteristics of the model help in improving operational efficiency. To achieve this goal, teams must collaborate and participate in the development and delivery process. All technical team members should have experience with the processes and tools involved in the development pipeline.

A mature DevOps process provides a number of benefits, as shown in the following diagram:



Figure 11.1: Benefits of DevOps

The benefits provided by DevOps are elaborated on here:

- **Speed:** Releasing product features faster helps to accommodate your customers' changing business needs and expand your market. A DevOps model enables an organization to achieve results faster.
- **Fast delivery:** DevOps processes facilitate efficiency by automating end-to-end pipelines, from code build to code deployment and production launch. Rapid delivery helps you to innovate faster. The faster release of bug fixes and features allows you to gain a competitive edge.

- **Reliability:** DevOps processes provide checks to ensure delivery quality and the safety of rapid application updates. DevOps practices like CI and CD embed automation testing and security checks for a positive end-user experience.
- **Scalability:** DevOps helps to scale your infrastructure and application on an on-demand basis by including automation everywhere.
- **Collaboration:** The DevOps model builds a culture of ownership whereby the teams consider their actions. The operations and dev teams work together in a shared responsibility model. Collaboration simplifies the process and increases efficiency.
- **Security:** In an agile environment, frequent changes require stringent security checks. The DevOps model automates security and compliance best practices, monitors them, and takes corrective action in an automated way.

Teams take full ownership of the services they deliver, often beyond the traditional scope of their roles, and develop thinking from a customer point of view to solve any issue. Let's look at understanding the different components of DevOps processes.

Understanding the components of DevOps

DevOps tools and automation bring together development and system operations. The following are the critical components of a DevOps practice:

- CI/CD
- Continuous monitoring and improvement
- Infrastructure as code
- Configuration management

A common best practice across all the elements is **automation**. Automation can involve scripts, templates, and other tools. In a thriving DevOps environment, infrastructure is managed as code. Automation enables DevOps teams to set up and tune test and production environments rapidly. Let's explore more details about each component.

Continuous integration/Continuous deployment

In **CI**, developers commit code frequently to a code repository. The code is built frequently. Each build is tested using automated unit tests and integration tests. In **CD**, you go further and frequently deploy your code in production. Builds are deployed to test environments and are tested using automated and possibly manual tests. Successful builds pass tests and are deployed to staging or production environments.

The following diagram illustrates the impact of CI versus CD in the software development life cycle:

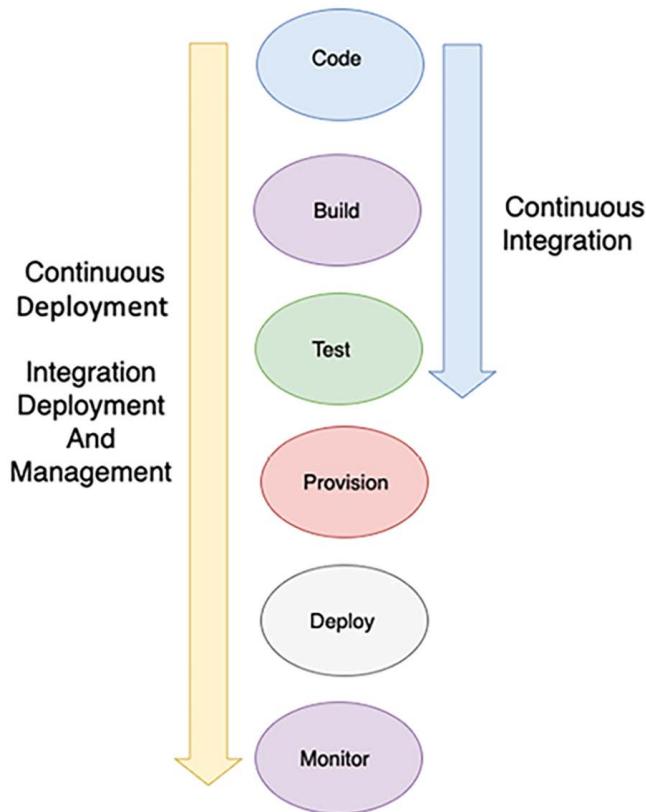


Figure 11.2: CI/CD

As shown in the preceding diagram, CI refers to the software development life cycle's building and unit testing stages. Every update that is committed in the code repository creates an automated build and test. The CD is an essential aspect of CI that extends the CI process further to deploy the build in production. In CI/CD practices, several people work on the code. They all must use the latest working build for their efforts. Code repositories maintain different versions of the code and also make the code accessible to the team. You check out the code from the repository, make changes, write new code in your local copy, compile and test your code, and frequently commit your code back to the main repository. In CI/CD, software development life cycle phases such as code, build, deploy, and test are automated using the DevOps pipeline.

CI automates most of the software release process. It creates an automated flow that builds, tests, and then stages the update. However, a developer must trigger the final deployment to a live production environment that is not automated. It expands upon CD by deploying all code changes to testing and/or production environments after the build stage. Developers will always have a tested and deployment-ready build if CD is implemented correctly.

The following diagram illustrates everything related to the automation of an application, ranging from code commits into a code repo to the deployment pipeline. It shows an end-to-end flow from the build to the production environment, where the developer checks the code change into the code repository, which the CI server pulls. The CI server triggers the build to create a deployment package with new application binaries and corresponding dependencies. These new binaries are deployed in a targeted development or testing environment. Binaries also get checked into the artifact repository for safe version-controlled storage:

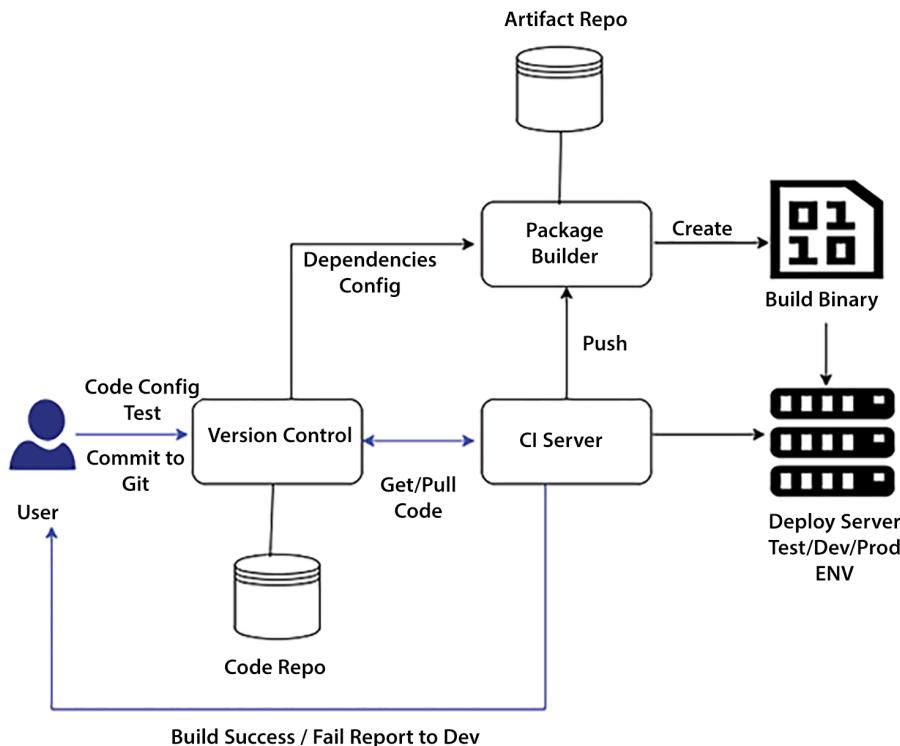


Figure 11.3: CI/CD for DevOps

A robust CD pipeline also automates the provisioning of infrastructure for testing and production environments and enables the monitoring and management of test and production environments. CD does not mean that every change committed by the developer goes into production. Instead, it means that every change is *ready* to go into production. When the changes are staged and tested in the stage environment, a manual approval process initiates and gives a green signal to deploy to production. Thus, in CD, deploying to production becomes a business decision and is still automated with tools.

Continuous monitoring and improvement

Continuous monitoring helps us to understand the impact of application and infrastructure performance on the customer. You can learn how code changes impact users by analyzing data and logs. Active monitoring is essential in the era of 24/7 services and constant updates to applications and infrastructure.

You can proactively monitor services by creating alerts and performing real-time analysis. You can track various metrics to monitor and improve your DevOps practice.

Examples of DevOps-related metrics are as follows:

- **Change volume:** This is the number of user stories developed, the number of lines of new code, and the number of bugs fixed.
- **Deployment frequency:** This indicates how often a team is deploying an application. This metric should generally remain stable or show an upward trend.
- **Lead time from development to deployment:** The time between the beginning of a development cycle and the end of deployment can be used to identify inefficiencies in the intermediate steps of the release cycle.
- **Percentage of failed deployments:** The percentage of failed deployments, including the number of deployments that resulted in outages, should be low.
- This metric should be reviewed in conjunction with the change volume. Analyze potential points of failure if the change volume is low but the number of failed deployments is high.
- **Availability:** Track how many releases caused failures that possibly resulted in violations of service-level agreements (SLAs). What is the average downtime for the application?
- **Customer complaint volume:** The number of complaint tickets filed by customers indicates the quality of your application.
- **Percentage change in user volume:** The number of new users signing up to use your application and the resulting increase in traffic can help you scale your infrastructure to match the workload.

After your deploy builds to the production environment, monitoring your application's performance is essential. As we discussed automating environments, let's explore **Infrastructure as code (IaC)** in detail.

Infrastructure as code

Provisioning, managing, and even deprecating infrastructure is a costly activity in terms of human effort. Furthermore, errors can be made by repeated attempts to build and modify environments manually. Whether working from prior experience or a well-documented runbook, the tendency for a human to make a mistake is a statistical probability.

We can automate the task of creating a complete environment. Task automation can help to complete repetitive tasks and provide significant value effortlessly.

With IaC, we can define our infrastructure in the form of **templates**. A single template may consist of a part or the entirety of an environment. More importantly, this template can be used repeatedly to create the same environment again.

In IaC, infrastructure is spun up and managed using code. An IaC model helps you interact with infrastructure programmatically at scale and avoid human errors by automating resource configuration. That way, you can work with infrastructure the same way you would with code by using code-based tools. As the infrastructure is managed through code, the application can be deployed using a standardized method, and any patches and versions can be updated repeatedly without errors.

Some of the most popular IaC scripting tools are Ansible, Terraform, Azure Resource Manager, Google Cloud Deployment Manager, Chef, Puppet, AWS Cloud Development Kit (CDK) and AWS CloudFormation.

The following is a code sample from AWS CloudFormation, which provides IaC capability to automate infrastructures on the AWS cloud platform.

```
AWSTemplateFormatVersion: '2010-09-09'
Description: 'Create an S3 Storage with a parameter to choose own bucket name.'

Parameters:
  S3NameParam:
    Type: String
    Default: 'architect-book-storage'
    Description: 'Enter the S3 Bucket Name'
    MinLength: '5'
    MaxLength: '30'

Resources:
  Bucket:
    Type: 'AWS::S3::Bucket'
    DeletionPolicy: Retain
    Properties:
      AccessControl: Private
      BucketName:
        Ref: S3NameParam
      Tags:
        - Key: 'Name'
          Value: 'MyBucket'

Outputs:
  BucketName:
    Description: 'BucketName'
    Value:
      Ref: S3NameParam
```

The preceding code creates Amazon S3 object storage with an option for the user to provide their choice of storage name, as shown below:

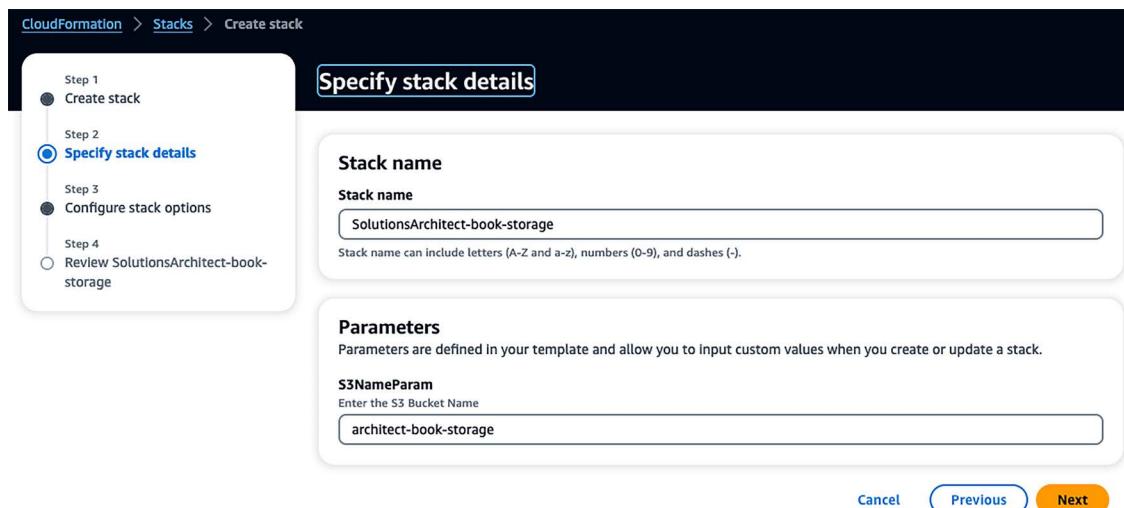


Figure 11.4: IaC using AWS CloudFormation

After execution of the code, the Amazon S3 bucket gets created, as you can see in Resources:

The screenshot shows the 'Resources' tab for the stack 'SolutionsArchitect-book-storage'. The top navigation bar includes 'Delete', 'Update', 'Stack actions ▾', and 'Create stack ▾'. Below the navigation bar, there are tabs for 'Stack info', 'Events', 'Resources' (which is selected and highlighted in blue), 'Outputs', 'Parameters', 'Template', and 'Change sets'. The main area is titled 'Resources (1)' and contains a search bar with the placeholder 'Search resources'. A table displays the single resource: 'Bucket' with logical ID 'architect-book-storage' (Physical ID 'architect-book-storage'), type 'AWS::S3::Bucket', and status 'CREATE_COMPLETE'. There are also buttons for refresh, back, forward, and zoom.

Figure 11.5: Automated AWS S3 object storage creation using AWS CloudFormation

Multiple teams can use the provided code to create any amount of Amazon S3 storage. As data is so essential, the admin chose to add bucket "DeletionPolicy": "Retain," which ensures storage doesn't get deleted when infrastructure comes down and that data is safe.

You can see how you can implement standardization, consistency, and compliance across organizations using IaC. Configuration management is another vital aspect of the DevOps process. Let's learn more about it.

Configuration management

Configuration management (CM) is the process of using automation to standardize resource configurations across your entire infrastructure and applications. CM tools such as Chef, Puppet, and Ansible can help you manage IaC and automate most system administration tasks, including provisioning, configuring, and managing IT resources.

By automating and standardizing resource configurations across the development, build, test, and deployment phases, you can ensure consistency and eliminate failures caused by misconfiguration. CM can also increase the productivity of your operations by allowing you to deploy the same configuration to hundreds of servers automatically at the push of a button. CM can also be leveraged to deploy changes to configurations.

Although you can use registry settings or databases to store system configuration settings, a CM application allows you to maintain version control as well, in addition to storage. CM is also a way to track and audit configuration changes. If necessary, you can even maintain multiple configuration settings versions for various software versions.

CM tools include a controller machine that manages server nodes. For example, Chef requires a client agent application installed on each server to manage, and a master Chef application is installed on the controller machine. Puppet also works the same way with a centralized server. However, Ansible has a decentralized approach that doesn't require installing agent software on the server nodes.

The following table shows a high-level comparison between the various popular configuration management tools:

	Ansible	Puppet	Chef
Mechanism	Controller machine applies changes to servers using Secure Shell (SSH)	Master synchronizes changes to Puppet node	The Chef workstation looks for changes in Chef servers and pushes them to the Chef node.
Architecture	Any server can be the controller	Centralized control by the Puppet master	Centralized control by Chef server
Script Language	YAML	Domain-specific on Ruby	Ruby
Scripting Terminology	Playbook and roles	Manifests and modules	Recipes and cookbooks
Test Execution	Sequential order	Non-sequential order	Sequential order

Table 11.1 – Popular CM tools comparison chart

CM tools provide a domain-specific language and set of features for automation. Some of these tools have a steep learning curve whereby the team has to learn the tool. AWS provides a managed platform called OpsWorks to manage Chef and Puppet in the cloud. It provides various attributes to manage IT infrastructure through automation, as shown below:

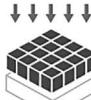
Layers	Instances
 <p>A layer is a blueprint for a set of instances. It specifies the instance's resources, installed packages, profiles and security groups.</p> <p>Add a layer</p>	 <p>An instance represents a server. It can belong to one or more layers, that determine the instance's resources and configuration.</p> <p>Add an instance or register a server</p>
Apps	Deployments and Commands
 <p>An app represents code stored in a repository that you want to run on application server instances.</p> <p>Add an app</p>	 <p>You can deploy the code from your repository to the appropriate server or run commands on some or all instances in your stack.</p> <p>Deploy an app or run a command</p>
Resources	Monitoring
 <p>The Resources page enables you to use any of your account's Elastic IP addresses, volumes, or RDS instances in your stack.</p> <p>Register resources</p>	 <p>AWS OpsWorks uses Amazon CloudWatch to provide thirteen custom metrics with detailed monitoring for each instance in the stack.</p> <p>Show monitoring</p>
Permissions	Tags <small>NEW</small>
 <p>Permissions specify how imported IAM users can access this stack. To import users, go to the Users page.</p> <p>Manage permissions</p>	 <p>You can specify tags to apply to resources in the stack. Tags can help you identify resources in cost allocation reports.</p> <p>Manage stack tags</p>

Figure 11.6: AWS OpsWorks service capabilities for managed Chef and Puppet

Security has become a priority for any organization, so complete security automation is the need of the hour. Organizations are moving to tight security implementations and monitoring to avoid human error, using the DevOps process popularly known as **DevSecOps**. In the next section, let's explore DevSecOps (short for development, security, and operations).

Introducing DevSecOps for Security

We are now more focused on security than ever. In many situations, security is the only way to win customer trust. DevSecOps is about the automation of security and the implementation of security at scale. The development team constantly makes changes, and the DevOps team publishes them in production (changes are often customer-facing). DevSecOps ensures application security in the overall process.

DevSecOps is not there to audit code or CI/CD artifacts. Organizations should implement DevSecOps to enable speed and agility, but not at the expense of validating security which slows down the development and deployment process. The power of automation is in increasing product feature launch agility while implementing the required security measures. A DevSecOps approach results in built-in security; security is not applied only as an afterthought. DevOps is about adding efficiency to speed up the product launch life cycle, while DevSecOps validates all building blocks without slowing the life cycle.

To institute a DevSecOps approach in your organization, start with a solid DevOps foundation across the development environment, as security is everyone's responsibility. It would be best to embed security in the architecture design from inception to create collaboration between development and security teams. Automate continuous security testing and build it into the CI/CD pipeline to avoid any security gaps. To keep track of any security breach, extend monitoring to include security and compliance by monitoring for drift from the design state in real time. Monitoring should enable alerting, automated remediation, and removal of non-compliant resources.

Codifying everything is an essential requirement that opens up infinite possibilities. DevSecOps aims to keep the pace of innovation, which should meet the pace of security automation. A scalable infrastructure needs scalable security, requiring automatic incident response remediation to implement continuous compliance and validation.

Combining DevSecOps and CI/CD

A DevSecOps practice must be embedded with every CI/CD pipeline step. DevSecOps ensures the security of the CI/CD pipeline by managing the proper access and roles assigned to each server and making sure the build servers, such as Jenkins, are hardened to be protected from any security glitch. In addition to that, we need to ensure that all artifacts are validated, and code analysis is in place.

It's advisable to be ready for incident response by automating continuous compliance validation and incident response remediation. For instance, if an organization needs to comply with the Payment Card Industry Data Security Standard (PCI-DSS), continuous compliance validation would involve setting up automated tools and processes to constantly check that the handling, processing, and storage of credit card information meet PCI-DSS requirements.

The following diagram provides us with multiple stages to test security boundaries and catch security issues and ensure compliance with policies as early as possible:

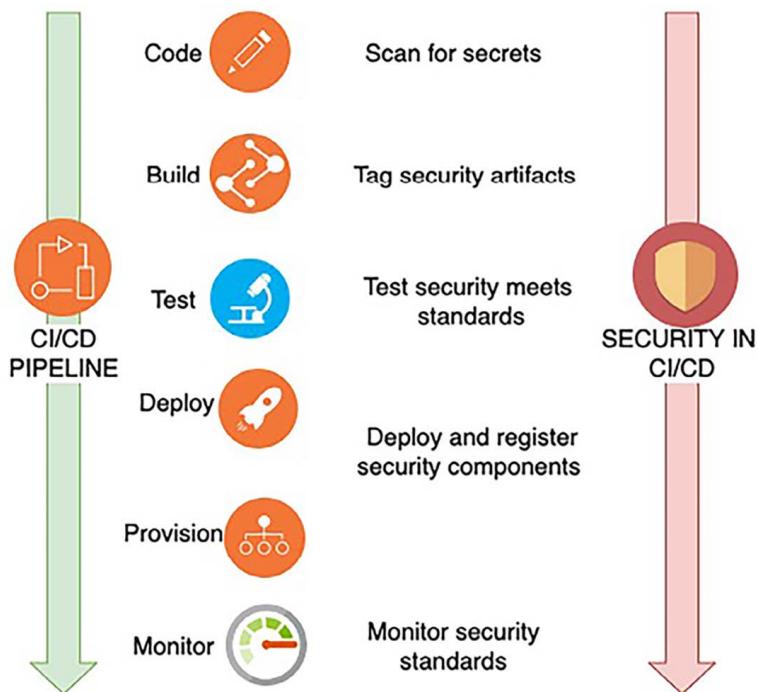


Figure 11.7: DevSecOps and CI/CD

At each integration point, you can identify different issues, as illustrated in the preceding diagram:

- In the **Code** phase, scan all code to ensure no secret or access key is hardcoded between code lines.
- During the **Build** phase, include all security artifacts, such as the encryption key and access token management, and tag them for easy identification.
- During the **Test** phase, scan the configuration to make sure all security standards are met by test security.
- In the **Deploy** and **Provision** phases, ensure all security components are registered. Perform a checksum to ensure no changes in the build files. A checksum is a technique used to determine the authenticity of received files. Operating systems provide a `checksum` command to validate the file and ensure no changes are made during file transfer.
- Monitor all security standards during the **Monitor** phase. Perform continuous audits and validation in an automated way.

You can integrate multiple tools into DevSecOps pipelines to identify security vulnerabilities at various stages and aggregate the vulnerability findings.

Application security testing (AST), which involves using tools to automate the testing, analysis, and reporting of security vulnerabilities, is a critical component of application development. AST can be broken down into the following four categories to scan security vulnerabilities in software applications:

- **Software composite analysis (SCA)**: SCA evaluates the open-source software's security, license compliance, and code quality in a codebase. SCA attempts to detect publicly disclosed vulnerabilities contained within a project's dependencies. Popular SCA tools are OWASP Dependency-Check, Synopsys' Black Duck, WhiteSource, Synk, and GitLab.
- **Static application security testing (SAST)**: SAST involves scanning an application's code prior to compilation. These tools provide developers with immediate feedback during the coding process, allowing for the early correction of issues before the code build phase. As a white-box testing method, SAST analyzes the source code to identify vulnerabilities that could make applications prone to attacks. Its key advantage is its integration early in the DevOps cycle, during the coding stage, as it doesn't require a functioning application or code execution. Popular SAST tools include SonarQube, PHPStan, Coverity, Synk, Appknox, Klocwork, CodeScan, and Checkmarx.
- **Dynamic application security testing (DAST)**: DAST identifies security vulnerabilities by mimicking external attacks on an application while it is running. It assesses the application from the outside, probing exposed interfaces for vulnerabilities. Known as black-box security testing or a web application vulnerability scanner, DAST tools include OWASP ZAP, Netsparker, Detectify Deep Scan, StackHawk, Appknox, HCL AppScan, GitLab, and Checkmarx.
- **Interactive application security testing (IAST)**: IAST examines code for security vulnerabilities while the application is actively being tested or used, thus reporting issues in real time without causing delays in the CI/CD pipeline. IAST tools are typically implemented in QA environments alongside automated functional tests. Notable IAST tools are GitLab, Veracode, CxSAST, Burp Suite, Acunetix, Netsparker, InsightAppSec, and HCL AppScan.

You will learn about integrating some of the above tools in the DevOps pipeline later in the chapter, under the *Building DevOps and DevSecOps in the cloud* section. DevSecOps CI/CD confirms that the code is validated against the corporate security policy.

It helps to avoid any infrastructure and application failure in later deployment due to different security configurations. DevSecOps maintains agility and ensures security at scale without affecting DevOps' pace of innovation. Let's learn about the CD strategy in the DevOps pipeline.

Implementing a CD strategy

CD provides seamless migration of the existing version of an application to the new version. Some of the most popular techniques to implement through CD are as follows:

- **In-place deployment**: Update application on a current server
- **Rolling deployment**: Gradually roll out the new version in the existing fleet of servers
- **Blue-green deployment**: Gradually replace the existing server with the new server
- **Red-black deployment**: Instant cutover to the new server from the existing server
- **Immutable deployment**: Stand up a new set of servers altogether

Let's explore each option in more detail.

In-place deployment

In-place deployment is a method of rolling out a new application version on an existing fleet of servers. The update is done in one deployment action, requiring some downtime. Hardly any infrastructure changes are needed for this update. There is also no need to update existing **Domain Name System (DNS)** records. The deployment process itself is relatively quick. If the deployment fails, redeployment is the only option for restoration.

As a simple explanation, you are replacing the existing application version (v1) on the application infrastructure with the new version (v2). In-place updates are low-cost and fast to deploy.

Rolling deployment

With a rolling deployment, the server fleet is divided into groups, so it doesn't need to be updated simultaneously. The deployment process runs old and new software versions on the same server fleet but with different subgroups. A rolling deployment approach helps to achieve zero downtime. If a new version deployment fails, then only a subset of servers is impacted from the entire fleet, and the risk is minimal because half of the fleet will still be up and running. A rolling deployment helps to achieve zero downtime; however, deployment time is little more than in-place deployment.

A rolling deployment not only helps achieve zero downtime, enhancing the user experience, but it is also cost-neutral in terms of additional resource allocation. Unlike blue-green deployments that require doubling the environment for a period, rolling deployments update existing resources one by one, avoiding the need for extra infrastructure. While the deployment time might be slightly longer compared to in-place deployments, this method doesn't incur extra costs from provisioning additional resources, making it an efficient strategy for continuous delivery without impacting your budget. Let's learn about blue-green deployment now.

Blue-green deployment

The idea behind blue-green deployment is that your blue environment is your existing production environment carrying live traffic. In parallel, you provide a green environment, which is identical to the blue environment, other than having the new version of your code. When it's time to deploy, you route production traffic from the blue to the green environment. If you encounter any issues with the green environment, you can roll it back by reverting traffic to the original blue environment. DNS cutover and swapping auto-scaling groups are the two most common methods to re-route traffic in blue-green deployment.

Using auto-scaling policies, you can gradually replace existing instances with instances hosting the new version of your application as your application scales out. This option is best used for minor releases and small code changes. Another option is to leverage DNS routing to perform sophisticated load balancing between different versions of our application.

As illustrated in the following diagram, after creating a production environment that hosts the new version of our application, you can use the DNS route to shift a small portion of traffic to the new environment:

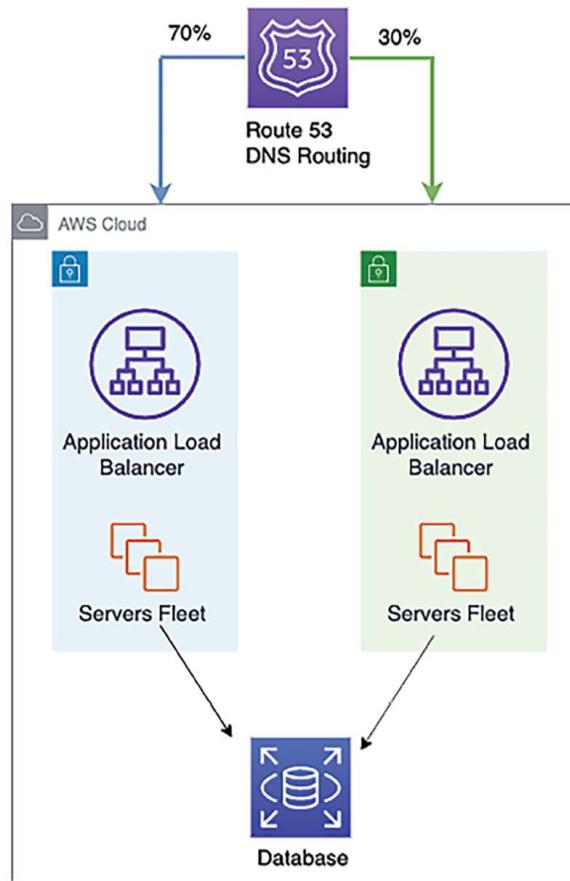


Figure 11.8: Blue-green deployment DNS gradual cutover

Test the green environment with a fraction of production traffic, called **canary analysis**. If the environment has functional issues, you can immediately tell and switch traffic back before impacting your users significantly. Continue to gradually shift traffic, testing the ability of the green environment to handle the load. Monitor the green environment to detect issues, providing an opportunity to change traffic back, thus limiting the blast radius. Finally, decommission the blue environment and release the resources when all the metrics are correct.

Blue-green deployment helps to achieve zero downtime and provides easy rollback. You can customize the time to deploy as per your needs. However, these zero downtimes do come with a cost consideration as this approach requires maintaining two identical production environments, one active (blue) and one idle (green). The need to duplicate the environment means higher operational costs due to the additional resources required. However, this cost is often justified by the value it adds in terms of risk mitigation and uninterrupted user experience.

Red-black deployment

Before standing up a new version of a system in red-black deployment, perform canary testing. The canary replaces around 1% of its existing production system with the latest version of the application and monitors the newest version for errors. The system is deemed ready for deployment if the canary clears this initial test. A new system version stands up alongside the old version in preparation for the switchover. The initial capacity of the new system is set manually by examining how many instances are currently running in production and setting this number as the desired capacity for the new auto-scaling group. Once the new system is up and running, both systems are red. The current version is the only version accepting traffic.

The system is then cut over from the existing version to the new version using the DNS service. At this point, the old version is regarded as black; it is still running but is not receiving any traffic. If any issues are detected with the new version, reverting becomes as simple as pointing the DNS server back to the old version's load balancer.

Red-black deployment is also known as **dark launch** and is slightly different from blue-green deployment. In red-black deployment, you do a sudden DNS cutover from the old version to the new version, while in blue-green deployment, the DNS gradually increases traffic to the new version. Blue-green deployments and dark launches can be combined to deploy both versions of software side by side. Two separate code paths are used, but only one is activated. A feature flag activates the other code path. This deployment can be used as a beta test where you can explicitly enable the new features.

Red-black deployment, similar to blue-green deployment, involves running two identical environments to ensure zero downtime and facilitate easy rollbacks. The cost implications are primarily associated with the need to double resources during the deployment phase. You'll have a "red" environment (the current live version) and a "black" environment (the new version). Both environments must be fully operational, which effectively doubles the resource requirements—including compute, storage, and network resources—during the transition period. While this approach significantly reduces deployment risks and provides a seamless user experience, the duplicated environment leads to increased costs. However, since the additional resources are only needed for the duration of the deployment window, this cost is temporary and can be considered an investment in stability and reliability.

Immutable deployment

An immutable or disposable upgrade is more straightforward if your application has unknown dependencies. An older application infrastructure that has been patched and re-patched over time becomes increasingly difficult to upgrade. This type of upgrade technique is more common in an immutable infrastructure.

During the new release, a new server instance set is rolled out by terminating older instances. For disposable upgrades, you can set up a cloned environment with deployment services such as Chef, Puppet, Ansible, and Terraform or use them combined with an auto-scaling configuration to manage the updates.

In addition to downtime, you must consider the cost while designing your deployment strategy. Consider the number of instances you need to replace and your deployment frequency to determine the cost. Use the approach that best fits, taking your budget and downtime into consideration.

In this section, you learned about various CD strategies that help make your application release more efficient and hassle-free. Let's see the best practices for choosing the right deployment type.

Best practices for choosing the right deployment strategy

Choosing the right deployment strategy is crucial for successful application updates and a seamless user experience. Here are the best practices for choosing among the various deployment strategies:

- **In-place deployment:** In-place deployment is ideal for scenarios where simplicity is key and the application is relatively small or has a limited user base. For instance, updating a company's internal tool with a small team fits this approach well. It involves updating the application on the current server, but it's important to note that it can cause downtime. This strategy is not the best fit for large-scale or high-availability applications. A notable example would be updating a small-scale web service overnight with low user traffic. It's crucial to have a rollback strategy in case the update fails to restore the previous version and minimize disruption quickly.
- **Rolling deployment:** Rolling deployment is suitable for applications that need minimal downtime but don't require additional resources. This approach updates the application gradually across the existing fleet of servers. An example would be deploying an update to an e-commerce website's servers in stages, ensuring that only a portion of users experience any potential issues at a time. However, this method is unsuitable for applications that cannot simultaneously handle different versions. Continuous monitoring of application performance during the deployment is key to addressing issues as they arise.
- **Blue-green deployment:** Blue-green deployment is best for critical applications where zero downtime is essential. A financial services company might use this strategy to update its customer-facing application. Once the green environment is thoroughly tested and ready, traffic is switched from blue to green. This method requires double the resources but offers a seamless user experience and quick rollback capability. It's crucial to ensure that load balancing and DNS switching mechanisms are robust and reliable.
- **Red-black deployment:** Red-black deployment is similar to blue-green but focuses on a faster cutover to the new version. It is particularly effective for quickly releasing new versions and is often used in containerized environments. For example, a media streaming service might deploy a new version of its platform using this strategy, ensuring immediate availability of new features to all users. While it offers rapid release and immediate switching, thorough testing of the new version is crucial as rollback involves reverting to the old environment.
- **Immutable deployment:** Immutable deployment ensures consistency and reliability, especially in cloud environments. Each deployment involves setting up new servers, guaranteeing a predictable and stable state. This approach could benefit an application with complex dependencies, as it avoids the "configuration drift" seen in long-lived environments. This strategy requires efficient management of infrastructure resources, as it involves provisioning new servers and decommissioning old ones with each release.

In each of these strategies, it's important to assess factors like the application's complexity, scale, user base, and the impact of potential downtime.

Additionally, resource availability, infrastructure costs, and the application's criticality should guide the deployment strategy choice. Regularly updating and refining the deployment approach in line with technological and organizational changes is also key to maintaining an effective deployment process.

You need to perform application testing at every step for high-quality delivery, which often requires significant effort. A DevOps pipeline can help you automate the testing process and increase the quality and frequency of feature releases. Let's learn more about continuous testing in the CI/CD pipeline.

Implementing continuous testing in the CI/CD pipeline

DevOps is key for continually changing business scenarios based on customer feedback, demand for new features, or shifts in market trends. A robust CI/CD pipeline ensures further features/feedback are incorporated in less time, and customers can use the new features faster.

With frequent code check-ins, having a good testing strategy baked into your CI/CD pipeline ensures you close that feedback loop with quality. Continuous testing is essential in balancing the CI/CD pipeline. While adding software features rapidly is good, ensuring that the features adhere to good quality standards is achieved by continuous testing.

Unit tests form the most significant amount of your testing strategy. They typically run on the developer's machine and are the fastest and cheapest. A general rule of thumb is to incorporate 70% of your testing efforts in unit testing. Bugs caught at this stage can quickly be fixed with fewer complexities.

The developer often performs unit tests, and once the code is ready, it is deployed for integration and system testing. These tests require their own environments and sometimes separate testing teams, which makes the testing process costlier. Once the team ensures that all intended features work as expected, the operations team needs to run performance and compliance tests. These tests need production-like environments and are costly. Also, **user acceptance testing (UAT)** needs a replica of production-like environments, causing more expense.

As illustrated in the following diagram, developers perform unit tests to test code changes/new features in the development phase. Testing is usually done on a developer's machine after coding is complete. It is also recommended to run static code analysis on the code changes and do code coverage, adherence to coding guidelines, and so on. Smaller unit tests with no dependencies run faster. Therefore, the developer can find out quickly if the test has failed:

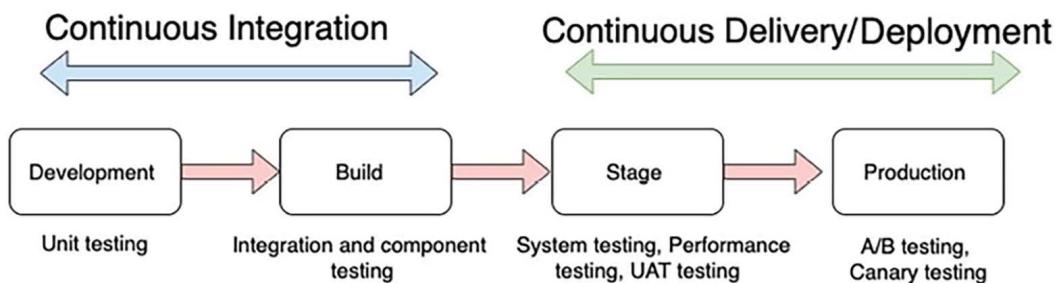


Figure 11.9: Continuous testing in CI/CD

The **build phase** is the first test for integration between different components and individual components. The build phase is also an excellent time to test if the code committed by a developer breaks any existing features and to perform regression testing.

A **staging environment** is a mirror of the production environment. An end-to-end system test is performed at this stage (the UI, backend logic, and API are tested extensively). Performance testing tests the application performance under a particular workload. Performance tests include load tests and stress tests. UAT is also performed at this stage in readiness for production deployment. Compliance testing is done to test for industry-specific regulatory compliance.

For example, let's assume you are integrating continuous testing into the CI/CD pipeline for a video personalization feature in an online streaming platform. When your development team commits their code changes, the CI tool, such as Jenkins, automatically initiates the build process and conducts a series of automated tests. These include unit tests to validate individual components of the personalization feature, integration tests to ensure compatibility with existing system components, and UI tests to confirm user interaction flows smoothly. Performance tests are particularly critical in this scenario to verify that the new feature does not degrade the streaming experience. If any issues arise during these tests, the pipeline is halted, allowing developers to make necessary corrections, thus ensuring that only thoroughly vetted code progresses to the next stages. After passing automated testing, the feature advances to a staging environment, mimicking production, for further testing and validation. This additional layer of scrutiny ensures the feature performs well under various scenarios and user behaviors through user acceptance testing before deployment.

A/B testing

Often, in software development, which feature's implementation will be most successful in the real world needs to be clarified. An entire computer science discipline—**human/computer interaction (HCI)**—is devoted to answering this question. While UI experts have several guidelines to help design suitable interfaces, the best choice of design often can only be determined by giving it to users and seeing whether they can use the design to complete a given task.

A strategy such as A/B testing or canary analysis tests the new application version in the production phase. In A/B testing, the new application version is deployed to a small percentage of production servers and tested for user feedback. Gradually, depending on how well the users receive the new application, the deployment is increased to span all production servers.

As shown in the following diagram, A/B testing is a methodology in which two or more versions of features are given to different sets of users. Detailed metrics on the usage of each implementation are gathered, and UI engineers examine this data to determine which implementation should be adopted going forward:

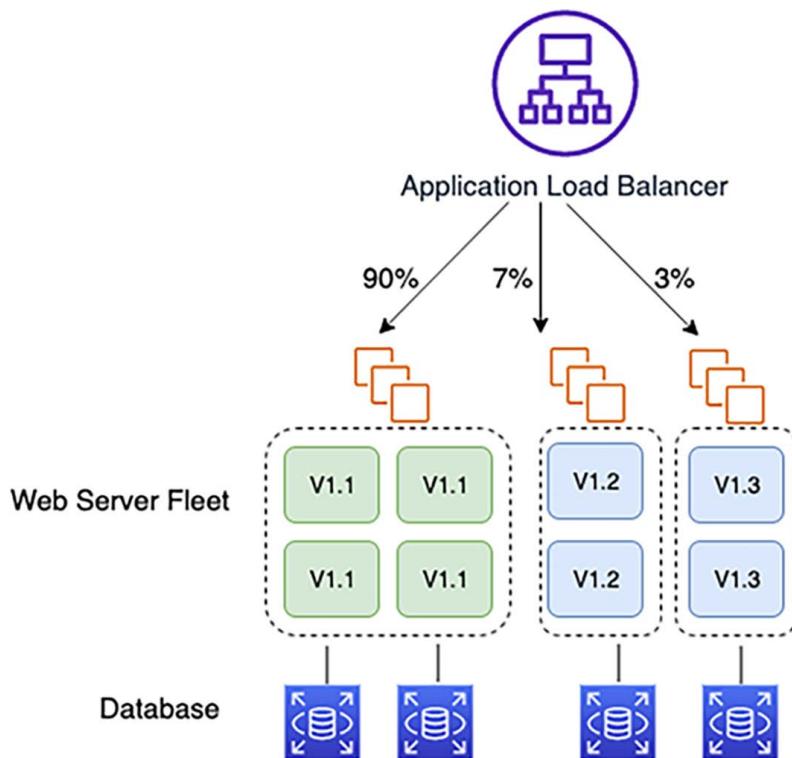


Figure 11.10: Split users by feature experiment using A/B testing

The preceding diagram represents an A/B testing scenario where different versions of a web application are being tested simultaneously to evaluate their performance, user engagement, or other defined metrics. Here's how the A/B testing process is depicted in this architecture:

1. **Traffic distribution:** The application load balancer plays a crucial role by directing incoming user traffic to different versions of the web application. In this scenario, the majority of the traffic (90%) is routed to the stable production version (V1.1), while the new versions under test, V1.2 and V1.3, receive a smaller fraction of the traffic, 7% and 3% respectively.
2. **Web server fleet:** Each version of the application runs on a separate set of web servers or instances, ensuring that changes in one version do not affect the others. This isolation is essential for obtaining accurate test results. The version receiving the most traffic serves as the control group, while the other versions with changes or new features are the test groups.
3. **Database:** All versions of the application interact with the same backend database. This is common in A/B tests where the underlying data remains consistent across different user experiences. However, care must be taken to ensure database schemas and interactions are compatible across all versions to prevent errors or inconsistencies in data handling.

You need to make sure that throughout the A/B test, performance metrics are continuously monitored to assess how each version of the application performs in real-world conditions. This includes evaluating response times, error rates, and resource utilization, among other factors. After a significant amount of data has been collected, the results are analyzed to determine which version of the application performs best based on the testing criteria. Decisions can then be made on whether to fully roll out a new version, make further modifications, or roll back changes based on the insights gained from the A/B test.

Using DevOps tools for CI/CD

To build a CI/CD pipeline, a developer requires various tools. These include a code editor, a source repository, a build server, a deployment tool, and orchestrating an overall CI pipeline. Let's explore some popular technology choices of developer tools for DevOps, both in the cloud and on-premises.

Code editor

DevOps is a hands-on coding role where you often need to write a script to automate the environment. You can use the Ace editor or the **cloud-based AWS Cloud9 integrated development environment (IDE)**. Ace provides syntax highlighting and other features that are valuable for developers. Cloud9 is integrated with the AWS platform, making it easy for developers to create serverless applications, and work with AWS services. It also supports collaborative coding and is equipped with essential tools for popular programming languages.

You can use a web-based code editor on your local computer or install a code editor on your local server that connects to the application environments—such as dev, test, and prod—to interact. In an environment, you store your project's files and run the tools to develop your apps. You can save these files locally on the instance or server or clone a remote code repository into your environment. The AWS Cloud9 IDE is a cloud-native IDE provided as a managed service.

The Ace editor lets you write code quickly and easily. It's a web-based code editor but provides performance similar to popular desktop-based code editors such as Eclipse, Vim, **Visual Studio Code (VS Code)**, and so on. It has standard IDE features such as live syntax and matching parentheses highlighting, auto-indentation and completion, toggling between tabs, integration with version control tools, and multiple cursor selections. It works with large files, having hundreds of thousands of lines without typing lag. It has built-in support for all popular coding languages and debugging tools, and you can also install your tools. For a desktop-based IDE, VS Code and Eclipse are other popular code editor options that DevOps engineers can choose.

Source code management

There are multiple choices available for your source code repository. You can set up, run, and manage your Git server, where you will be responsible for everything.

You can choose to use a hosting service such as GitHub or Bitbucket. If you are looking for a cloud solution, **AWS CodeCommit** offers a secure, highly scalable, and managed source control system to host private Git repositories.

You need to set up authentication and authorization for your code repository to provide access to authorized team members for code to read or write. You can apply data encryption in transit and at rest. When you push into the code repository (`git push`), it encrypts and stores the data. When you pull from the code repository (`git pull`), it decrypts the data and then sends it back to the caller. The user must be authenticated with the proper access level to the code repository. Data can be encrypted in transit by transmitting through encrypted network connections using HTTPS or SSH protocols.

CI server

A CI server is also known as a **build server**. With teams working on multiple branches, merging back into the main branch is complicated. CI, in this scenario, plays a key role. CI server hooks provide a way to trigger the build based on the event when code is committed to the repository. Hooks, incorporated in almost every version control system, refer to custom scripts triggered by specified necessary actions in a repository. Hooks can run either on the client side or the server side.

Pull requests are common for developers to notify and review each other's work before it is merged into common code branches. A CI server provides a web interface to review changes before adding them to the final project. If there are any problems with the proposed changes, the source code can be sent back to the developer to tweak as per the organization's coding requirements.

As shown in the following diagram, server-side hooks in combination with the CI server are used to increase the velocity of integration:

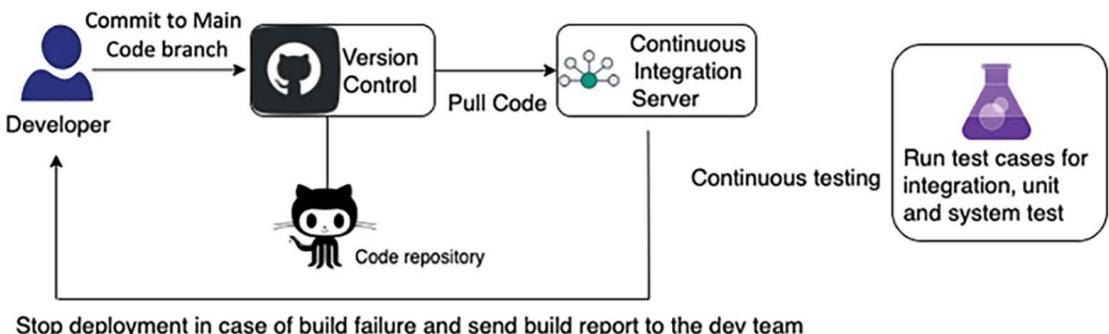


Figure 11.11: Automation of CI

As illustrated in the preceding diagram, using `post-receive`, you can direct new branches to trigger tests on a CI server to verify that the new build integrates correctly and that all units function correctly. The developer is notified of test failures and then knows to merge their branch with the mainline only after fixing the problems. The developer can build from their branch, test the changes there, and get feedback on how well their changes work before deciding whether to merge their branch into the mainline.

Running integration and unit tests significantly reduces resistance when merging that branch into the mainline. Hooks can also be customized to test merges into the mainline and block any merges that don't pass. Integration is best accomplished with a CI server.

Jenkins is the most popular choice for building the CI server. As illustrated in the following diagram, you can host the Jenkins cluster in the AWS Elastic Compute Cloud (EC2) server's fleet and auto-scale as per build load:

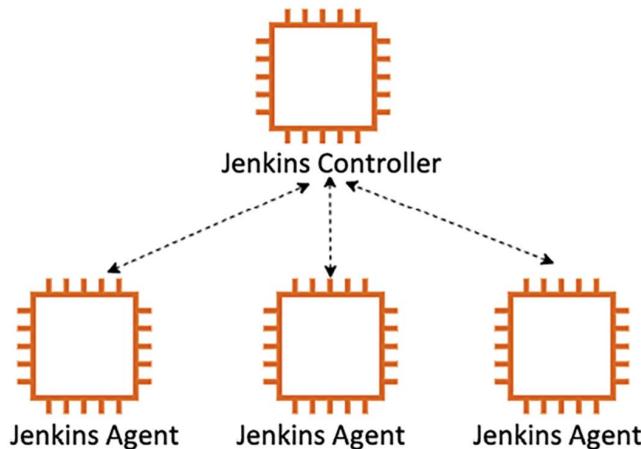


Figure 11.12: Auto-scaling of Jenkins CI servers

The **Jenkins Controller** offloads builds to the agent node instance in the case of overload. When the load goes down, the **Jenkins Controller** automatically terminates agent instances.

However, you have to maintain security and patch the server by yourself. For native cloud options and managed services, you can use managed code-build services such as AWS CodeBuild, eliminating the need for server administration and significantly reducing costs with a **pay-as-you-go** model—the service scales as per your demand. Your team is empowered to focus on pushing code and lets a service build all the artifacts.

While a CI server helps you build the correct version of code from a source code repository by collaborating across team members of the development team, code deployment helps the team get code ready for testing and release for end-user consumption. Let's learn about code deployment in more detail.

Code deployment

Once your build is ready, you can deploy the Jenkins server or choose AWS CodeDeploy as a cloud-native managed service. You can use other popular tools, such as Chef or Puppet, to create a deployment script. The options for specifying a deployment configuration are as follows:

- **OneAtATime:** Only a single instance in a deployment group at a time installs a new deployment. Suppose a deployment on a given instance fails. In that case, the deployment script will halt the deployment and return an error response detailing the number of successful versus the number of failed installations.
- **HalfAtATime:** Half of the instances in the deployment group install a new deployment. The deployment succeeds if half of the instances successfully install the revision. **HalfAtATime** is a good option for production/test environments where half of the instances are updated to a new revision, and the other half remain available in production at an older revision.

- **AllAtOnce:** Each instance installs the latest revision available whenever it next polls the deployment service. This option is best used for development and test deployments as it has the potential to install a non-functioning deployment on every instance in a deployment group.
- **Custom:** You can use this command to create a custom deployment configuration specifying a fixed number of healthy hosts that must exist in a deployment group at any given time. This option is a more flexible implementation of the OneAtATime option. It allows for the possibility that a deployment may fail on one or two instances that have become corrupt or are improperly configured.

The following diagram illustrates life cycle events during deployment:

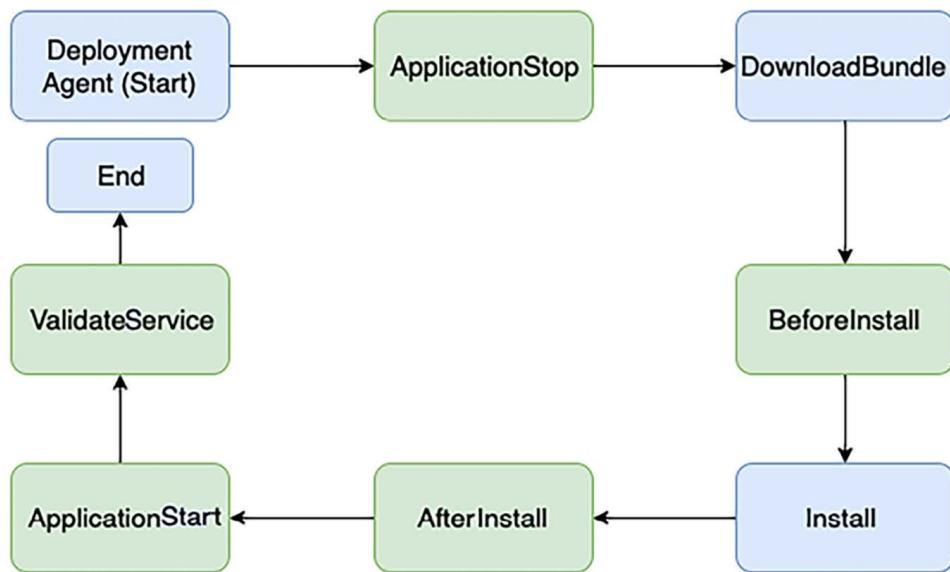


Figure 11.13: Deployment life cycle events

The deployment agent runs through a series of steps to execute a deployment. These steps are called life cycle events. In the preceding diagram, the steps shown in lighter boxes can be controlled by human intervention; however, the steps shown in darker boxes are automated and controlled by a deployment agent. Let's go into some detail for each of these steps:

1. **ApplicationStop:** To trigger a deployment, the first requirement is to stop the application server so that traffic stops serving while files are copied. Examples of software application servers are Tomcat, JBoss, or WebSphere servers.
2. **DownloadBundle:** After stopping the application server, the deployment agent starts downloading a pre-built deployment bundle from an artifactory such as JFrog Artifactory. The artifactory stores the application binary, which can be deployed and tested for application before the new version launch.
3. **BeforeInstall:** The deployment agent triggers pre-install steps such as creating a backup of the current version and any required configuration update via a script.

4. **Install:** In this step, deployment agents start the installation—for example, running an Ant or Maven script to install a Java application.
5. **AfterInstall:** The deployment agent triggers this step after completing your application installation. It may include updating post-installation configuration, such as local memory settings and log parameters.
6. **ApplicationStart:** In this step, the agent starts the application and notifies the success or failure operations team.
7. **ValidateService:** The validation step fires after everything else is done, allowing you to check the app quickly. It includes steps such as performing automated sanity tests and integration tests to verify if the new version of the application has been installed correctly. The agent also sends a notification to the team when testing is successful.

You have learned about various code deployment strategies and steps as independent components. However, you must stitch all the DevOps steps together to set up an automated CI/CD pipeline. Let's learn more about the code pipeline, which can help you build an end-to-end CI/CD pipeline.

Code pipeline

The code pipeline is about orchestrating everything together to achieve CD. The entire software release process is fully automated in CD, including building and deploying to the production release. Over some time, with experiments, you can set up a mature CI/CD pipeline. The path to the production launch is automated, thus enabling the rapid deployment of features and immediate customer feedback. You can use cloud-native managed services such as AWS CodePipeline to orchestrate the overall code pipeline or use the Jenkins server.

The code pipeline enables you to add actions to stages in your CI/CD pipeline. Each action can be associated with a provider that executes the action. The code pipeline action categories and examples of providers are as follows:

- **Source:** Your application code needs to be stored in a central repository with version control called **source code repositories**. Some popular code repositories are AWS CodeCommit, Bitbucket, GitHub, **Concurrent Versions System (CVS)**, **Subversion (SVN)**, and so on.
- **Build:** The build tool pulls code from the source code repository and creates an application binary package. Some of the popular build tools are AWS CodeBuild, Jenkins, Solano CI, and so on. Once the build is completed, you can store binaries in an artifactory such as JFrog.
- **Deploy:** The deployment tool helps you to deploy application binaries on the server. Some popular deployment tools are AWS Elastic Beanstalk, AWS CodeDeploy, Chef, Puppet, Jenkins, and so on.
- **Test:** Automated testing tools help you to complete and perform post-deployment validation. Some popular test-validating tools are Jenkins, BlazeMeter, Ghost Inspector, etc.
- **Invoke:** You can use an events-based script to invoke activities such as backup and alert. Any scripting language, such as a shell script, PowerShell, and Python, can be used to invoke various customized activities.

- Approval:** Approval is an essential step in CD. You can either ask for manual approval by an automated email trigger, or approval can be automated from tools.

In this section, you learned about various DevOps tools to manage the **software development life cycle (SDLC)**, such as a code editor, a repository, and build, test, and deployment tools. The other tools you need to integrate into DevOps pipelines are continuous logging, continuous monitoring, and operation handling, which you learned in *Chapter 9, Operational Excellence Considerations*. As of now, you have learned about various DevOps techniques for each SDLC phase. Let's learn more about best practices and anti-patterns.

Implementing DevOps best practices

While building a CI/CD pipeline, consider your need to create a project and add team members to it. The project dashboard provides visibility to the code flow through the deployment pipeline, monitors the build, triggers alerts, and tracks application activities. The following diagram illustrates a well-defined DevOps pipeline:

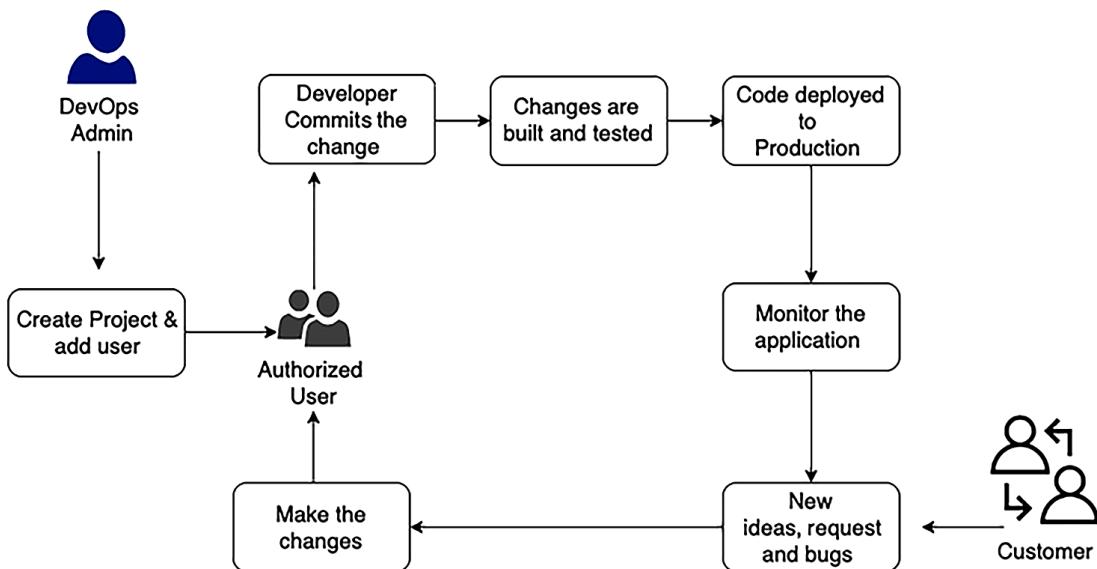


Figure 11.14: CI/CD workflow best practice

Consider the following points while designing the pipeline:

- The number of stages:** Stages could be development, integration, system, user acceptance, and production. Some organizations also include dev, alpha, beta, and release stages.
- Types of tests in each stage:** Each stage can have multiple types of tests, such as unit tests, integration tests, system tests, UATs, smoke tests, load tests, and A/B tests at the production stage.
- The sequence of a test:** Test cases can be run in parallel or need to be in sequence.
- Monitoring and reporting:** Monitor system defects and failures and send notifications as failures occur.

- **Infrastructure provisioning:** Methods to provision infrastructure for each stage.
- **Rollback:** Define the rollback strategy to fall back to the previous version if required.

Having a system that requires manual intervention where it's avoidable slows down your process. So, automating your process using CD will accelerate it.

Another common anti-pattern is keeping configuration values for a build inside the code or even having developers use different tools in their build processes, leading to inconsistent builds between developers. It takes lots of time and effort to troubleshoot why particular builds work in one environment rather than in others. To overcome this, it is better to store build configurations outside of code. Externalizing these configurations to tools that keep them consistent between builds enables better automation and allows your process to scale much more quickly. Not using a CD process can lead to last-minute, middle-of-the-night rushes to get a build to work. Design your CD process to *fail fast* to reduce the likelihood of any last-minute surprises.

Externalizing environment-specific configurations is crucial for maintaining consistency and scalability across builds. Some tools and services that facilitate this abstraction include:

- **AWS Systems Manager Parameter Store:** Provides secure, hierarchical storage for configuration data management and secrets management. You can store data such as passwords, database strings, and license codes as parameter values.
- **ConfigMaps and Secrets in Kubernetes:** Kubernetes objects that allow you to separate configuration artifacts from image content to keep containerized applications portable.
- **Docker Swarm secrets:** Used to manage sensitive data with Docker containers, offering a way to securely transmit and store secrets within the Swarm cluster.
- **Consul by HashiCorp:** A service networking solution to automate network configurations with distributed key-value stores.

By using these tools, you can manage your configurations and secrets outside of your application code and templates, making it easier to manage and rotate them securely without redeploying or changing the application.

To effectively gauge the impact of CI/CD within your DevOps framework, monitoring **key performance indicators (KPIs)** is essential. Key CI/CD KPIs include:

- Deployment frequency, which signals how often updates reach production, reflecting the agility of your release process
- Lead time for changes, which shows the duration from code commitment to live deployment, with shorter spans indicating a more efficient development cycle

- Change failure rate, which identifies the proportion of deployments causing failures, with a lower rate denoting better deployment stability
- **Mean time to recovery (MTTR)**, which measures the average recovery duration from a failure, with quicker recoveries exemplifying a team's effective incident management
- Automated test pass rate, which highlights the reliability of your code through the success rate of automated tests in each CI/CD cycle

The twelve-factor methodology can be used to apply architecture best practices at each step of application development, as recommended by The Twelve-Factor App (<https://12factor.net/>), which enterprises adopt for the end-to-end development and delivery of web applications. This applies to all coding platforms regardless of programming languages. Nowadays, most applications are built as web apps and utilize a cloud platform. Let's learn how to build end-to-end DevOps and security automation in the cloud.

Building DevOps and DevSecOps in the cloud

As you have learned in previous sections, building a CI/CD pipeline requires multiple tools, and adding security automation on top of that increases the complexity. Integrating a range of tools and consolidating the results of vulnerability assessments from the ground up can be a complex task. A public cloud provider like AWS offers the adaptability required to construct DevSecOps pipelines. This includes the straightforward integration of both cloud-native and third-party tools, as well as the capability to aggregate security findings effectively.

DevSecOps pipeline architecture covers CI/CD practices, including SCA, SAST, and DAST tools:

- **Software composition analysis (SCA)** tools analyze open-source components in your application to identify known vulnerabilities, licensing issues, and outdated libraries. They can automate the process of checking for updates and security patches, making it easier to manage your application's dependencies.
- **SAST** tools are designed to analyze source code or compiled versions of code to detect security vulnerabilities. They can identify issues such as input validation errors, insecure dependencies, and potential backdoors without executing the code.
- **DAST** tools assess running applications for vulnerabilities. Unlike SAST tools that analyze static code, DAST tools interact with the application from the outside, performing black-box testing to detect issues such as SQL injection, cross-site scripting, and authentication problems.

Integrating these tools into a CI/CD pipeline allows for continuous and automated security testing, enabling teams to detect and resolve security issues promptly and improve the overall security posture of their applications. The following diagram visualizes the concepts of security automation in the pipeline:

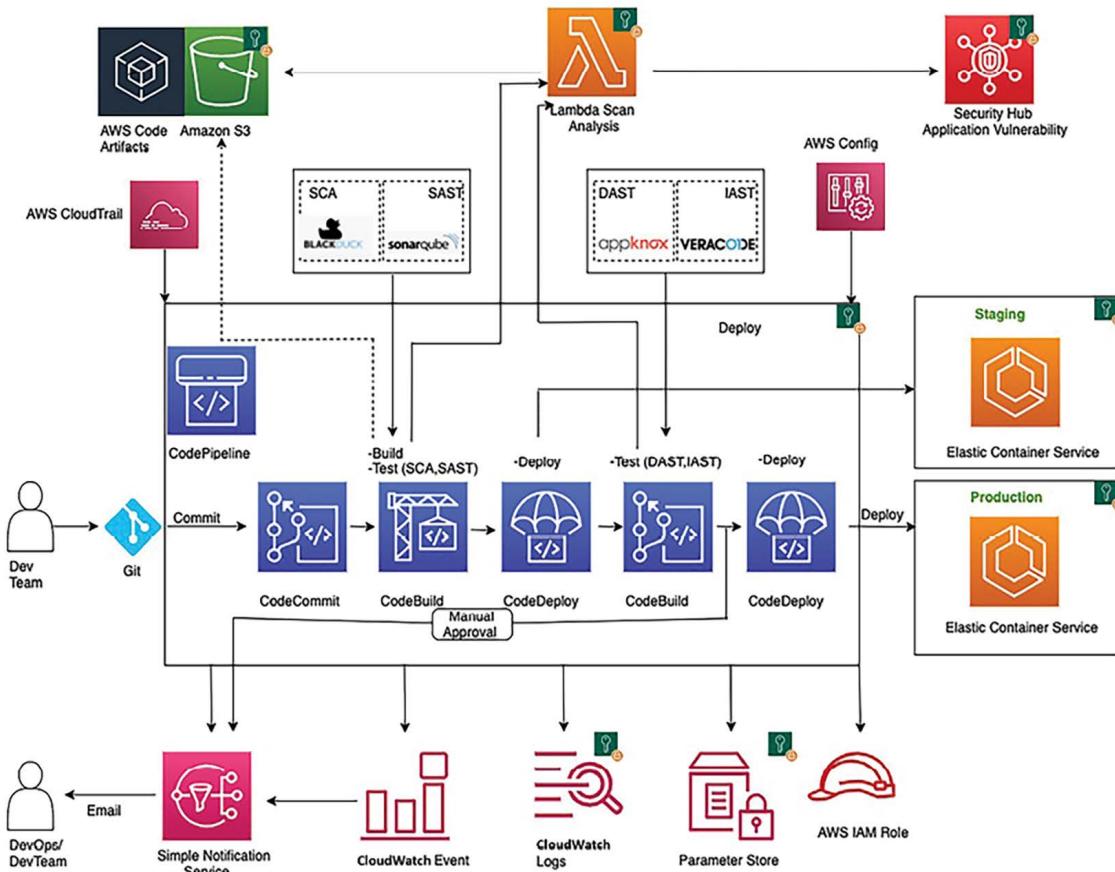


Figure 11.15: DevSecOps CI/CD pipeline architecture in the AWS cloud

The preceding diagram shows that the CI/CD pipeline gets triggered when a developer commits the code in GitHub. An event gets generated to start the AWS CodePipeline using AWS CloudWatch. AWS CodePipeline orchestrates the CI/CD pipeline, including code commit, build, and deployment. AWS CodeBuild compiles the build and then uploads the resulting artifacts to AWS CodeArtifact. For initiating the scanning process, AWS CodeBuild fetches authentication details, including tokens for scanning tools, from the AWS Parameter Store.

Once the deployment is successfully completed, CodeBuild initiates DAST. If this process uncovers any vulnerabilities, CodeBuild triggers a Lambda function. This function then proceeds to log the security findings in AWS Security Hub. Suppose DAST finds no security issues. In that case, the build can advance for approval, and the pipeline notifies the approver for action to push the build into the production AWS ECS environment. During the CI/CD pipeline run, AWS CloudWatch monitors all the changes and sends email notifications to DevOps and the dev team through SNS notifications.

AWS CloudTrail tracks any critical changes, such as pipeline updates, deletions, and creation, and sends notifications to the DevOps team for audit purposes. Further, AWS Config tracks all the configuration changes.

In DevSecOps, securing the CI/CD pipeline is achieved through AWS **Identity and Access Management (IAM)** roles, which restricts access strictly to the necessary resources. Encryption and **Secure Sockets Layer (SSL)** are employed to protect pipeline data both at rest and in transit. Sensitive details like API tokens and passwords are securely stored in the AWS Parameter Store.

Centralizing security findings in AWS Security Hub facilitates the automation of remediation processes. Depending on the nature of the security issue, a Lambda function can be triggered to perform the required remedial actions. For instance, if an SSH port is inadvertently exposed, the system can automatically restrict server access from the internet. This automation relieves the burden on DevOps and security teams, allowing them to address vulnerabilities from a single tool, rather than managing multiple dashboards.

Addressing security threats early in the application development life cycle can significantly reduce the cost of making changes to the application. Automating this process can further expedite the delivery of these changes, making a DevSecOps pipeline an essential component for successful application development.

DevOps integrates culture, practices, and tools to merge application development with operations, enabling rapid delivery of new application features. DevSecOps extends this by weaving security into the DevOps process, ensuring that secure and compliant application changes are delivered swiftly and operations are consistently automated. This integration is key to maintaining a secure, efficient, and resilient application development environment.

Summary

In this chapter, you have learned about the key components of a strong DevOps practice along with its benefits, CI/CD, and continuous monitoring and improvement. The agility of CI/CD can be achieved only by applying automation everywhere. To automate, you learned about IaC and configuration management. You also looked at various automation tools such as Chef, Puppet, and Ansible to automate configuration management.

As security is the priority, you learned about DevSecOps, which is DevOps in security. CD is one of the critical aspects of DevOps. You learned about various deployment strategies: rolling, blue-green, and red-black. Testing is another aspect of ensuring the quality of your product. You learned about the concept of continuous testing in DevOps and how A/B testing can help improve the product by taking direct feedback from a customer in the live environment.

You have learned about the stages in a CI/CD pipeline. You have learned about the tools and services you can use and the best practices you can follow for a robust CI/CD pipeline. You have learned how individual services work and discussed how to integrate services to build a sophisticated solution.

Until this point, you have learned about various aspects of solution architecture. As every organization has lots of data, they put great effort into getting insight into their data. In the next chapter, you will learn about collecting, processing, and consuming data to get a more in-depth insight.

Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below to get a free eBook of your choice.



12

Data Engineering for Solution Architecture

In the previous chapter, you learned about the DevOps process, which automates the application deployment pipeline and fosters a culture of collaboration among development, operations, and security teams. This chapter will introduce you to data engineering, including the various tools and techniques used to collect data from different parts of your application to gain insights that can drive your business.

Data is being generated everywhere with high velocity and volume in the internet and digitization era. Getting insights from these enormous amounts of data at a fast pace is challenging. We must continuously innovate to ingest, store, and process this data to derive business outcomes.

With the convergence of cloud, mobile, and social technologies, advancements in many fields, such as genomics and life sciences, are growing ever-increasingly. Tremendous value is found in mining this data for more insight. Modern stream processing systems must produce continual results based on data with high input rates at low latency.

The concept of *big data* refers to more than just the collection and analysis of data. The actual value for organizations in their data can be used to gain insight and create competitive advantages. Not all big data solutions must end in visualization. Many solutions, such as **machine learning (ML)** and other predictive analytics, feed these answers programmatically into other software or applications, extracting the information and responding as designed.

As with most things, getting faster results costs more, and big data is no exception. Some answers might not be needed immediately, so the solution's latency and throughput can be flexible enough to take hours to complete. Other responses, such as in predictive analytics, may be needed as soon as the data is available.

In this chapter, you will learn about the following topics to handle and manage your big data needs:

- What is big data architecture?
- Designing big data processing pipelines

- Data ingestion, storage, processing, and analytics
- Visualizing data
- Designing big data architectures
- Big data architecture best practices

By the end of this chapter, you will know how to design big data and analytics architecture. You will learn about the big data pipeline steps, including data ingestion, storage, processing, visualization, and architecture patterns.

What is big data architecture?

The sheer volume of collected data can cause problems. With the accumulation of more and more data, managing and moving data along with its underlying big data infrastructure becomes increasingly difficult. The rise of cloud providers has facilitated the ability to move applications to the cloud. Multiple sources of data result in increased volumes, velocity, and variety. The following are some common computer-generated data sources:

- **Application server logs:** Application logs and games
- **Clickstream logs:** From website clicks and browsing
- **Sensor data:** Weather, water, wind energy, and smart grids
- **Images and videos:** Traffic and security cameras

Computer-generated data can vary from semi-structured logs to unstructured binaries. Computer-generated data sources can produce pattern matching or correlations in data that generate recommendations for social networking and online gaming. You can also use computer-generated data, such as blogs, reviews, emails, pictures, and brand perceptions, to track applications or service behavior.

Human-generated data includes email searches, natural language queries, sentiment analysis on products or companies, and product recommendations. Social graph analysis can produce product recommendations based on your circle of friends, jobs you may find interesting, or even reminders based on your circle of friends' birthdays, anniversaries, and so on.

Typical barriers you hear from analytics teams that prevent them from delivering the most value to their organizations are:

- **Limited insight into customer experiences and operations:** To create new customer experiences, organizations need better visibility into their business. Complex and costly data collection, processing systems, and added scale costs require organizations to limit the types and amount of data they collect and analyze.
- **Need to make quicker decisions:** This is a two-part problem:
 - Traditional data systems are overwhelmed, resulting in workloads taking a long time to complete.
 - More decisions need to be made in seconds or minutes, requiring systems to collect and process data in real time.

- **Enabling innovation with ML:** Organizations are adding and growing their data science teams to help optimize and grow their business. These users need more access to data with their choice of tools without the traditional red tape and processes that will slow them down.
- **Technical staff and cost to scale self-managed infrastructures:** Customers who manage infrastructure on-premises need help to quickly scale to meet business demand. Managing infrastructure, high availability, scaling, and operational monitoring takes time, especially at scale.

In **big data architecture**, the general flow of a significant data pipeline starts with data and ends with insight. How you get from start to finish depends on a lot of factors. The following diagram illustrates a data workflow pipeline that will help you design your data architecture:

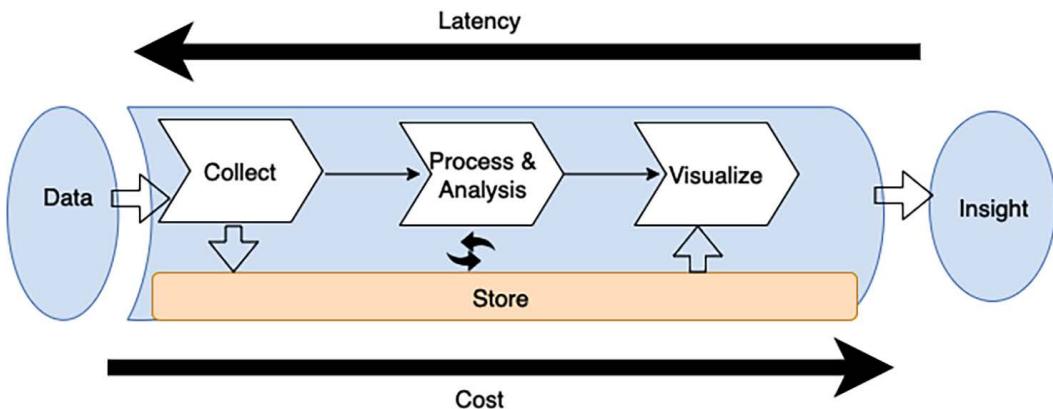


Figure 12.1: Big data pipeline for data architecture design

As shown in the preceding diagram, the standard workflow of the big data pipeline includes the following steps:

1. Data is collected (ingested) by an appropriate tool.
2. The data is stored persistently.
3. The data is processed or analyzed. The data processing/analysis solution takes the data from storage, performs operations, and then stores the processed data again.
4. The data is then used by other processing/analysis tools or by the same tool again to get further answers from the data.
5. To make answers useful to business users, they are visualized using a **business intelligence (BI)** tool or fed into an ML algorithm to make future predictions. Once the appropriate answers have been presented to the user, this gives them insight into the data they can use to make further business decisions.

The tools you deploy in your pipeline determine your *time to answer*, which is the latency between when your data was created and when you can get insight from it. The best way to architect data solutions while considering latency is to determine how to balance throughput with cost because a higher performance and reduced latency usually result in a higher price. For example, a financial trading platform requires real-time analytics to provide its users with immediate insights for quick decision making.

To achieve this, the platform might employ an expensive data processing pipeline that includes in-memory databases, real-time stream processing, and high-speed data ingestion services. This setup ensures low latency, allowing traders to respond to market changes instantaneously. Here, the business necessity for real-time analytics justifies the high costs associated with the low-latency architecture.

Designing big data processing pipelines

One of the critical mistakes many big data architectures make is handling multiple data pipeline stages with one tool. A fleet of servers managing the end-to-end data pipeline, from data storage and transformation to visualization, may be the most straightforward architecture, but it is also the most vulnerable to breakdowns in the pipeline. Such tightly coupled big data architecture typically does not provide the best possible balance of throughput and cost for your needs. When you are designing a data architecture, use FLAIR data principles as explained in the following:

- **F – Findability:** This refers to the capability to easily locate available data assets and access their metadata, which includes information like ownership and data classification, along with other crucial attributes necessary for data governance and compliance.
- **L – Lineage:** The ability to trace the origin of data, track its movement and history, and understand as well as visualize how data flows from its sources to its points of consumption.
- **A – Accessibility:** This involves the facility to request and obtain security credentials that grant the right to access a specific data asset. It also implies the need for a networking infrastructure that supports efficient data access.
- **I – Interoperability:** Ensuring that data is stored in formats that are accessible and usable by most, if not all, internal processing systems within the organization.
- **R – Reusability:** Data should be documented with a known schema, and the source of the data should be clearly attributed. This aspect often includes principles of **master data management (MDM)**, which focuses on the management of critical data from different domains to provide, with accuracy and consistency, a single point of reference.

Big data architects recommend decoupling the pipeline between ingestion, storage, processing, and getting insight. There are several advantages to decoupling storage and processing in multiple stages, including increased *fault tolerance*. For example, if something goes wrong in the second round of processing and the hardware dedicated to that task fails, you won't have to start again from the beginning of the pipeline; your system can resume from the second storage stage. Decoupling your storage from various processing tiers allows you to read and write to multiple data stores.

The following diagram illustrates various tools and processes to consider when designing a big data architecture pipeline:

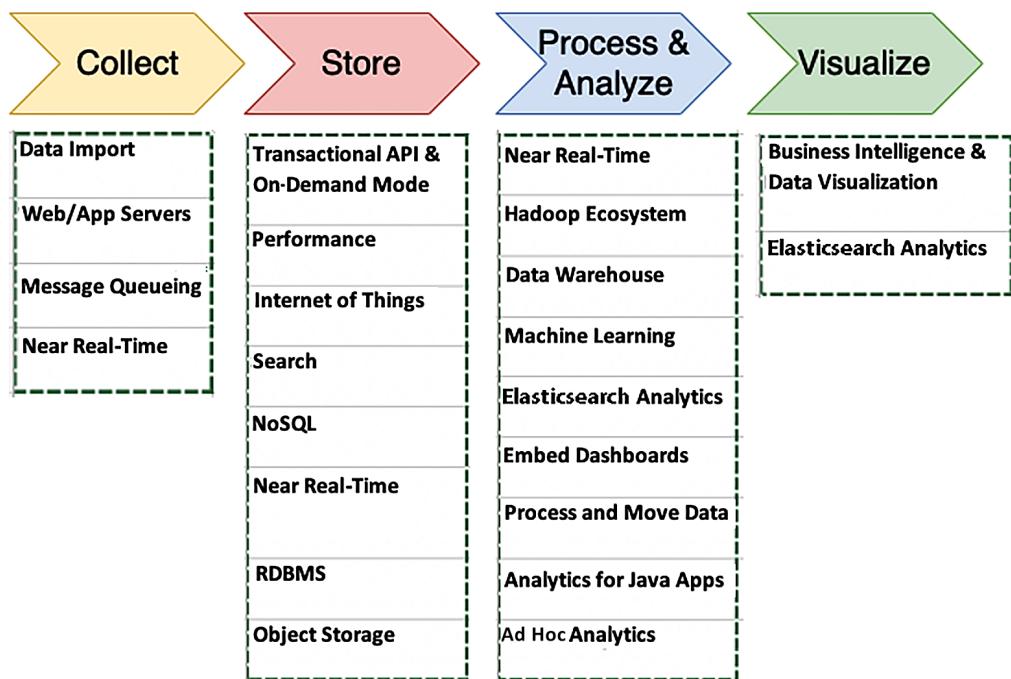


Figure 12.2: Tools and processes for big data architecture design

The things you should consider when determining the right tools for your big data architectures include the following:

- The structure of your data
- Maximum acceptable latency
- Minimum acceptable throughput
- Typical access patterns of your system's end users

Your data structure impacts both the tools you use to process it and where you store it. The ordering of your data and the size of each object you're storing and retrieving are also essential considerations. How your solution weighs latency/throughput and cost determines the time to answer.

User access patterns are another essential component to consider. Some jobs require regularly joining many related tables, and others require daily or less frequent data storage. Some jobs require comparing data from a wide range of data sources, and other jobs pull data from only one unstructured table. Knowing how your end users will most often use the data will help you determine the breadth and depth of your big data architecture. Let's dive deep into each process and the tools involved in big data architecture.

Data ingestion, storage, processing, and analytics

To turn raw data into actionable intelligence that can inform decision making and strategic planning for businesses, data needs to be managed through several key stages, beginning with **data ingestion**—the collection of data from various sources. This can include everything from user-generated data to machine logs, or real-time streaming data. Once collected, the data needs to be stored in **data storage**, which can be done in databases, data lakes, or cloud storage solutions, depending on the data type and intended use.

Following storage, **data processing and analytics** come into play, which involves sorting, aggregating, or transforming the data into a more usable form, where analytics can be performed on the processed data to extract meaningful insights. Analytics can range from simple queries and reporting to complex ML algorithms and predictive modeling. Let's learn about these stages in detail.

Data ingestion

Data ingestion is the act of collecting data for transfer and storage. There are lots of places from where data can be onboarded. Predominantly, data ingestion falls into one of the categories of databases, streams, logs, and files. Among these, databases are the most popular. These typically consist of your main upstream transactional systems that are the primary data storage for your applications. They take on both relational and non-relational flavors, and several techniques for extracting data from them exist.

Streams are open-ended sequences of time-series data, such as clickstream data from websites or **Internet of Things (IoT)** devices, usually published in an API we host. Applications, services, and operating systems generate logs. As shown in the following diagram, use the type of data your environment collects, and how it is collected, to determine what kind of ingestion solution is ideal for your needs:

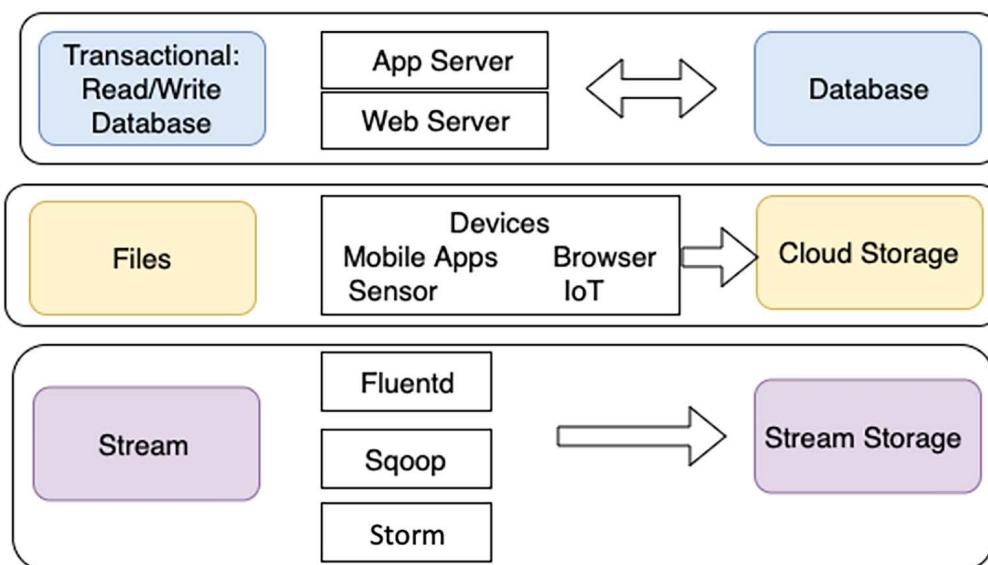


Figure 12.3: Type of data ingestion

As shown, transactional data storage must be able to store and retrieve data quickly. End users need quick and straightforward data access, making app and web servers the ideal ingestion methods. For the same reasons, NoSQL and **relational database management system (RDBMS)** databases are usually the best solutions for these kinds of processes.

Data transmitted through individual files is typically ingested from connected devices. A large amount of file data does not require fast storage and retrieval compared to transactional data. For file data, often a transfer is one way, where data is produced by multiple resources and ingested into a single object or file storage for later use.

Stream data such as clickstream logs should be ingested through an appropriate solution such as **Apache Kafka** or **Fluentd**. Apache Kafka is a popular choice for this purpose, offering robust publish-subscribe capabilities that can handle massive amounts of data efficiently. Fluentd is another tool that can be used for data ingestion, particularly known for its log aggregation capabilities.

Initially, these logs are stored in stream storage solutions such as Kafka, so they're available for real-time processing and analysis. Long-term storage of these logs is best in a low-cost solution such as object storage.

Streaming storage decouples your collection system (producers) from the processing system (consumers). It provides a persistent buffer for your incoming data. The data can be processed, and you can pump the data at a rate dependent on your needs. Let's learn about some popular data ingestion technologies.

Technology choices for data ingestion

Let's look at some popular open source tools for data ingestion and transfer:

- **Apache DistCp:** DistCp stands for *distributed copy* and is part of the Hadoop ecosystem. The DistCp tool is used to copy large data within a cluster or between clusters. DistCp achieves efficient and fast data copying by utilizing MapReduce's parallel processing distribution capability. It distributes directories and files into map tasks to copy file partitions from source to target. DistCp also does error handling, recovery, and reporting across clusters.
- **Apache Sqoop:** Sqoop is also part of the Hadoop ecosystem project and helps to transfer data between Hadoop and relational data stores such as RDBMS. Sqoop allows you to import data from a structured data store into the **Hadoop Distributed File System (HDFS)** and to export data from the HDFS into a structured data store. Sqoop uses plugin connectors to connect to relational databases. You can use the Sqoop extension API to build a new connector or use one of the included connectors that support data exchange between Hadoop and standard relational database systems.
- **Apache Flume:** Flume is open source software mainly used to ingest a large amount of log data. Apache Flume collects and aggregates data to Hadoop reliably and distributes it. Flume facilitates streaming data ingestion and allows analytics.

More open source projects, such as Apache Storm and Apache Samza, are available for streaming to process unbounded data streams reliably.

Ingesting data to the cloud

Ingesting data into the cloud is critical to managing and leveraging big data. The three major cloud providers—AWS, Google Cloud Platform (GCP), and Azure—offer various data ingestion services. Each has unique features and capabilities tailored to different needs and data volumes. Let's look at some of the unique features of these three cloud providers:

- AWS data ingestion services:
 - **AWS Direct Connect:** This offers a high-speed, private network connection to AWS, reducing latency and increasing bandwidth. It's ideal for transferring large volumes of data and provides a more consistent network speed than internet-based transfers.
 - **AWS Snowball and Snowmobile:** These services provide physical devices for transferring vast volumes of data (in terabytes and petabytes (PBs)) to AWS. Snowball is suitable for hundreds of terabytes, while Snowmobile can handle up to 100 PBs in a single transfer, ideal for huge datasets.
 - **AWS Database Migration Service (DMS):** This facilitates the migration of databases to AWS. It supports both homogeneous and heterogeneous migrations and can handle ongoing data replication through **change data capture (CDC)**.
- GCP data ingestion services:
 - **Google Cloud Storage Transfer Service:** This allows for the transfer of large volumes of data to Google Cloud Storage from online data sources like Amazon S3 and HTTP/HTTPS locations, as well as from on-premises data storage.
 - **Pub/Sub:** This offers real-time messaging and streaming data ingestion. It's a scalable and flexible service that enables the ingestion of streaming data like logs and event data for real-time analytics.
 - **Dataflow:** An integrated service for both data ingestion and processing. It's handy for **extract, transform, and load (ETL)** tasks and real-time event stream processing.
- Azure data ingestion services:
 - **Azure Data Factory:** This data integration service supports both on-premises and cloud data movements and transformations. It enables the ingestion of data from a variety of sources, processing it using computing services like Azure HDInsight and Azure Batch, and subsequently publishing the processed data to storage solutions such as Azure SQL Data Warehouse.
 - **Azure Event Hubs:** A robust and scalable data streaming platform and event ingestion service, Azure Event Hubs is capable of handling millions of events per second. This makes it an ideal solution for real-time analytics on data originating from various sources like applications, websites, or IoT devices.
 - **Azure Import/Export service:** This service is designed for the bulk transfer of large data volumes to and from Azure Blob Storage and Azure Files. It leverages physical disks for data transfer, making it a viable option for situations where transferring large amounts of data over a network might be too slow or expensive.

Each cloud provider offers a unique set of tools to meet various data ingestion needs, from real-time streaming to large-scale data migration, ensuring flexibility and scalability in big data management.

Streaming data is also becoming very important to ingest and analyze. You will learn more about streaming data in the *Streaming data stores* section. Let's learn more about the techniques you can use to choose the right storage and the available storage choices.

Storing data

One of the most common mistakes when setting up storage for a big data environment is using one solution, frequently an RDBMS, to handle all of your data storage requirements.

You will have many tools available, but they need to be optimized for the task they need to complete. One solution is not necessarily the best for all of your needs; the best solution for your environment might be a combination of storage solutions that carefully balance latency with cost. An ideal storage solution uses the right tool for the right job. The following diagram combines multiple factors related to your data and the storage choice associated with it:

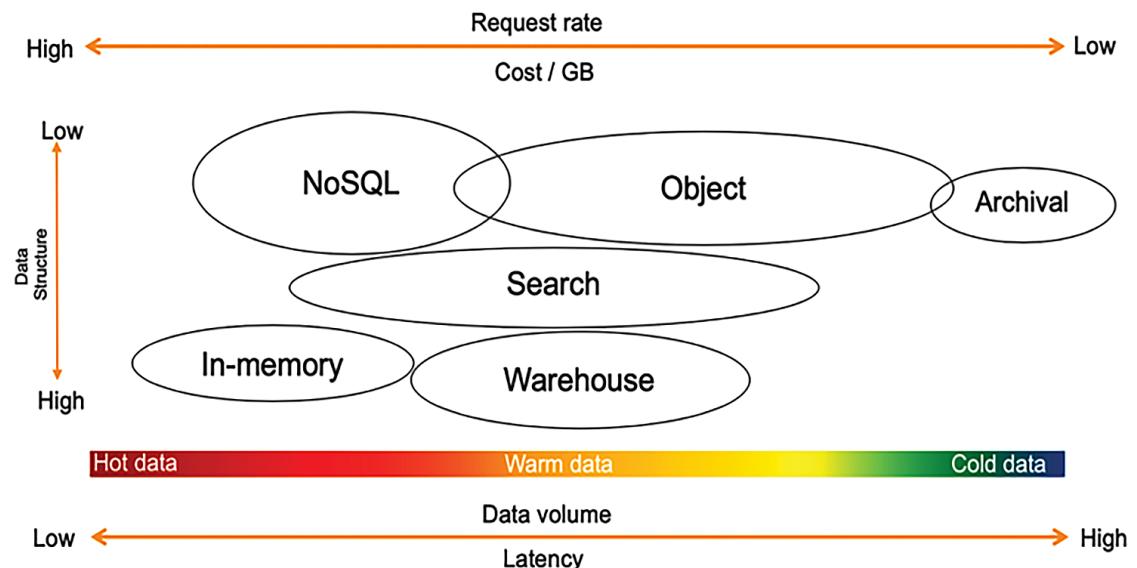


Figure 12.4: Understanding data storage

As shown in the proceeding diagram, choosing a data store depends upon the following factors:

- **How structured is your data?** Does it adhere to a specific, well-formed schema, as with Apache weblogs (logs are generally poorly structured and unsuitable for relational databases), standardized data protocols, and contractual interfaces? Is it completely arbitrary binary data, as in images, audio, video, and PDF documents? Or is it semi-structured with a general structure but potentially high variability across the records, as in JSON or CSV?

- **How quickly does new data need to be available for querying?** Is it a real-time scenario where decisions are made as new records stream in, such as campaign managers adjusting based on conversion rates or a website making product recommendations based on user behavior similarity? Is it a daily, weekly, or monthly batch scenario, such as model training, financial statement preparation, or product performance reporting? Or is it somewhere in between, such as with user engagement emails, where it doesn't require real-time action, and you can have a buffer of a few minutes or even a few hours between the user action and the touchpoint?
- **What is the size of the data ingest?** Is the data ingested recorded by the record as data comes in, such as with JSON payloads from REST APIs that measure at least a few KBs at best? Is it a large batch of records arriving simultaneously, such as system integrations and third-party data feeds? Or is it somewhere in between, such as with a few micro-batches of clickstream data aggregated together for more efficient processing?
- **What is the total volume of data and its growth rate?** Are you in GBs and TBs, or do you intend to store data in PBs or exabytes (EBs)? How much of this data is required for your specific analytics use cases? Do most of your queries only require a specific rolling window of time? Or, do you need a mechanism to query the entirety of your historical dataset?
- **What will the cost be to store and query the data in any particular location?** When it comes to any computing environment, we generally see a *triangle of constraints* between performance, resilience, and low cost. The better the performance and the higher the resilience you want your storage to have, the more expensive it will be. You can have quick queries over PBs of data but settle on querying TBs of data in a compressed format to meet your cost requirements.

Finally, what type of analytic queries will run against the data? Will it power a dashboard with a fixed set of metrics and drill down? Will it participate in large numerical aggregations rolled up by various business dimensions? Or will it be used for diagnostics, leveraging string tokenization for full-text searching and pattern analysis?

When you determine your data's characteristics and understand the data structure, you can assess which solution you need to use for your data storage. Let's learn about the various solutions for storing data.

Technology choices for data storage

As we discussed, a single tool can only do a few things. It would be best if you used the right tool for the right job, and a data lake enables you to build a highly configurable big data architecture to meet your specific needs. Business problems need to be narrower, deeper, and more complex for one tool to solve everything, especially big data and analytics.

For example, hot data will need to be stored and processed in memory, so caches or in-memory databases like Redis or SAP HANA are appropriate. AWS offers the ElastiCache service, providing a managed Redis or memcached environment. NoSQL databases are ideal when facing high-velocity but small-sized records, for example, user-session information or IoT data. NoSQL databases are also useful for content management to store data catalogs. Let's learn about the most popular and commonly used storage for structured data.

Structured data stores

Structured data stores have been around for decades and are the most familiar technology choice for storing data. Most transactional databases such as Oracle, MySQL, SQL Server, and PostgreSQL are row-based due to dealing with frequent data writes from software applications. Organizations often repurpose transactional databases for reporting purposes, requiring frequent data reads but much fewer data writes. Looking at high data-read requirements, more innovation is coming into querying on structured data stores, such as the columnar file format, which helps to enhance data-read performance for analytics requirements.

Row-based formats store the data in rows in a file. Row-based writing is the fastest way to write the data to the disk, but it is not necessarily the quickest read option because you need to skip over a lot of irrelevant data. Column-based formats store all the column values together in the file. This leads to better compression because the same data types are grouped. It also typically provides better read performance because you can skip columns that are not required.

Let's look at common choices for the structured data store. For example, you need to query the total number of sales in a given month from the order table, which has fifty columns. The query will scan the entire table with all fifty columns in a row-based architecture. In columnar architecture, the query will scan the order sales column, thus improving data query performance. Let's look into more details about relational databases, focusing on transaction data and data warehousing to handle data analytics needs.

Relational databases

A RDBMS is more suitable for **online transaction processing (OLTP)** applications. Some popular relational databases are Oracle, MSSQL, MariaDB, PostgreSQL, and so on. Some of these traditional databases have been around for decades. Many applications, including e-commerce, banking, and hotel booking, are backed by relational databases. Relational databases are very good at handling transaction data where complex joint queries between tables are required. Looking at transaction data needs, the relational database should adhere to the **atomicity, consistency, isolation, and durability (ACID)** principles as follows:

- **Atomicity:** Atomicity means the transaction will be executed fully from end to end, and, in the case of any error, the entire transaction will roll back.
- **Consistency:** Consistency means that all data should be committed to the database when transactions are completed.
- **Isolation:** Isolation requires multiple transactions to run concurrently in isolation without interfering with each other.
- **Durability:** In case of any interruption, such as a network or power failure, the transaction should be able to resume to the last known state.

Data from relational databases is often offloaded to data warehousing solutions for reporting and aggregation purposes. Let's learn more about data warehousing.

Data warehousing

Data warehouses are central repositories that store accumulations of data from one or multiple sources. They store current and historical data to help create analytical reports for business data analytics. However, data warehouses store data centrally from various systems, but they cannot be treated as data lakes. Data warehouses handle only structured relational data, while data lakes work with structured and unstructured data, such as JSON logs and CSV data.

Data warehouse databases are more suitable for **online analytical processing (OLAP)** applications. These databases are optimized for operations that involve reading large amounts of data, allowing for the aggregation and summarization of data to extract valuable business insights.

Take, for example, a banking scenario where a bank maintains a data warehouse that stores comprehensive information about customer accounts, transactions, loan details, and branch information. Additionally, the bank collects and stores data on customer interactions, service usage, and online banking activities in a related system.

Through OLAP, the bank can perform complex analyses of this combined data. It can query the data warehouse to uncover trends, such as identifying the most popular types of accounts or loans, analyzing transaction volumes over time, or assessing the usage patterns of online versus in-branch banking services. This analytical capability enables the bank to make informed decisions on product offerings, customer service improvements, and operational strategies, ultimately enhancing customer satisfaction and driving business growth.

Data warehouses provide fast aggregation capabilities over vast volumes of structured data. While these technologies, such as Amazon Redshift, Netezza, and Teradata, are designed to execute complex aggregate queries quickly, they must be optimized for high volumes of concurrent writes. So, data needs to be loaded in batches, preventing warehouses from serving real-time insights over hot data.

Modern data warehouses use a columnar base to enhance query performance. Examples of this include Amazon Redshift, Snowflake, and Google BigQuery. These data warehouses provide fast query performance due to columnar storage and improved I/O efficiency. In addition, data warehouse systems such as Amazon Redshift increase query performance by parallelizing queries across multiple nodes and taking advantage of **massively parallel processing (MPP)**.

Columnar storage enhances query performance by storing data in columns rather than rows, enabling improved data compression, selective data reading, and faster operations. This approach allows for more effective compression as similar data is stored sequentially, facilitating faster data retrieval as only necessary columns are accessed during queries. It also optimizes CPU cache utilization by loading relevant data into memory, enhancing the processing speed. Additionally, columnar storage supports massive parallel processing, where multiple processors can work on different data segments simultaneously, significantly boosting performance for analytical tasks that involve large datasets and require quick aggregation and filtering.

Data warehouse solutions such as Amazon Redshift can process PBs of data and provide decoupled compute and storage capabilities to save costs. In addition to columnar storage, Redshift uses data encoding, distribution, and zone maps to increase query performance. More traditional row-based data warehousing solutions include Netezza, Teradata, and Greenplum.

Data warehouses lead to the physical separation of data from different applications, necessitating data architects to construct new infrastructures around these warehouses. The constraints of traditional data warehouses have become more pronounced with the growing diversity of enterprise data, including text, IoT data, images, audio, and video. Moreover, the advent of ML and **artificial intelligence (AI)** has brought forth iterative algorithms that demand direct data access and do not rely on SQL, thus highlighting the limitations of conventional data warehouse models. You will learn more about overcoming these challenges later in this chapter, in the *Designing big data architectures* section.

NoSQL databases

NoSQL databases such as DynamoDB, Cassandra, and MongoDB address the scaling and performance challenges you often experience with a relational database. As the name suggests, NoSQL is a non-relational database. NoSQL databases store data without an explicit and structured mechanism to link data from different tables (no joins, foreign keys, or normalization enforced).

NoSQL utilizes several data models, including columnar, key-value, search, document, and graph. NoSQL databases provide scalable performance, high availability, and resilience. NoSQL typically does not enforce a strict schema, and every item can have an arbitrary number of columns (attributes), meaning one row can have four columns. In contrast, another can have ten columns in the same table. The partition key is used to retrieve values or documents containing related attributes. NoSQL databases are highly distributed and can be replicated. They are durable and don't experience performance issues when highly available.

SQL versus NoSQL databases

SQL databases have existed for decades, and most are already familiar with relational databases. Let's learn about some significant differences between SQL and NoSQL databases:

Properties	SQL Databases	NoSQL Databases
Data model	The relational model normalizes data in SQL databases into tables containing rows and columns. A schema includes tables, columns, relationships between tables, indexes, and other database elements.	NoSQL databases operate without enforcing a fixed schema, offering flexibility in data storage and retrieval. They often utilize a partition key to access values from sets of columns. This type of database is well-suited for storing semi-structured data, including formats like JSON, XML, and various other document types, such as data catalogs and file indexes.
Transaction	SQL-based traditional RDBMSs support and comply with ACID transactional data properties.	NoSQL databases sometimes trade certain ACID properties, which are characteristic of traditional RDBMSs, in order to facilitate horizontal scaling and enhance flexibility in their data models.

Performance	SQL-based RDBMSs were used to optimize storage when storage was expensive and minimize the disk footprint. For traditional RDBMSs, performance has mostly relied on the disk. Index creation and table structure modifications are required to achieve performance query optimizations.	In NoSQL systems, the performance is significantly influenced by factors such as the size of the underlying hardware cluster, network latency, and the manner in which the application interacts with the database.
Scale	SQL-based RDBMS databases are most straightforward to scale vertically with high-configuration hardware. The additional effort requires relational tables to span distributed systems, such as performing data sharding.	NoSQL databases are engineered to scale out horizontally, utilizing distributed clusters composed of cost-effective hardware. This approach is aimed at boosting throughput while minimizing any impact on latency.

Table 12.1 – SQL versus NoSQL database comparison

Depending on your data, various categories of NoSQL data stores exist to solve a specific problem. Let's learn about the types of NoSQL databases.

Types of NoSQL databases

The following are the major NoSQL database types:

- **Columnar databases:** Apache Cassandra and Apache HBase are the popular columnar databases. A columnar data store helps you scan a particular column when querying the data rather than scanning the entire row. Suppose an item table has ten columns with one million rows, and you want to query the number of items available in inventory. In that case, the columnar database will apply the query to the item quantity column rather than scanning the entire table.
- **Document databases:** Some of the most popular document databases are **MongoDB**, **Couchbase**, **MarkLogic**, **DynamoDB**, **DocumentDB**, and **Cassandra**. You can use a document database to store semi-structured data in JSON and XML formats.
- **Graph databases:** Popular graph database choices include **Amazon Neptune**, **JanusGraph**, **TinkerPop**, **Neo4j**, **OrientDB**, **GraphDB**, and **GraphX** in Spark. A graph database stores vertices and links between vertices called **edges**. Graphs can be built on both relational and non-relational databases.
- **In-memory key-value stores:** Some of the most popular in-memory key-value stores are **Redis** and **Memcached**. They store data in memory for heavy reading applications. Any query from an application first goes to an in-memory database, and if the data is available in the cache, it doesn't hit the master database. The in-memory database is suitable for storing user-session information, which results in complex queries and frequently requests data such as user profiles.

NoSQL has many use cases, but you must index all your data to build a search. Let's learn more about search data stores.

Search data stores

The Elasticsearch service is one of the most popular search engines for big data use cases like click-stream and log analysis. Search engines work well for warm data that can be queried ad hoc across any number of attributes, including string tokens.

Amazon OpenSearch Service provides data search capabilities and the support of open source Elasticsearch clusters, including API access. It also provides Kibana as a visualization mechanism to search for indexed data stores. AWS manages capacity, scaling, and patching of clusters, removing any operational overhead. Log search and analysis is a popular big data use case where OpenSearch helps you analyze log data from websites, server fleets, IoT sensors, and so on. Various applications in industries such as banking, gaming, marketing, application monitoring, advertisement technology, fraud detection, recommendations, and IoT utilize OpenSearch and Elasticsearch. ML-based search services, such as Amazon Kendra, are also available, providing more advanced search capabilities using natural language processing (NLP).

Unstructured data stores

When you look at the requirements for an unstructured data store, Hadoop is a perfect choice because it is scalable, extensible, and very flexible. It can run on consumer hardware, has a vast ecosystem of tools, and appears cost-effective. Hadoop uses a *master-and-child-node* model, where data is distributed between multiple child nodes, and the primary node coordinates jobs for running queries on data. The Hadoop system is based on MPP, making it fast to perform queries on all data types, whether structured or unstructured.

When a Hadoop cluster is created, each child node created from the server comes with a block of the attached disk storage called a local HDFS disk store. You can run the query against stored data using common processing frameworks like Hive, Pig, and Spark. However, data on the local disk persists only for the life of the associated instance.

If you use Hadoop's storage layer (HDFS) to store your data, you are coupling storage with compute. Increasing storage space means adding more machines, which also increases compute capacity. For maximum flexibility and cost-effectiveness, you need to separate compute and storage and scale them independently. Overall, object storage is more suited to data lakes to store all kinds of data cost-effectively and efficiently. Cloud-based data lakes backed by object storage provide flexibility to decouple compute and storage. Let's learn more about object storage.

Object storage

Object storage refers to data stored and accessed with units often referred to as objects stored in buckets. In object storage, files or objects are not split into data blocks, but data and metadata are kept together. There is no limit on the number of objects stored in a bucket, and they are accessed using API calls (usually through HTTP, GET, or PUT) to read and write to and from buckets. Typically, object storage is not mounted as a filesystem on operating systems because the latency of API-based file requests and lack of file-level locking provide poor performance as a filesystem.

Object storage offers scale and has a flat namespace, reducing management overhead and metadata management. Object storage has become more popular with the public cloud and is the go-to storage to build a scalable data lake in the cloud. Amazon S3, Azure Blob Storage, and Google Cloud Storage in GCP are the most popular object storage options.

Vector Database (**VectorDB**)

VectorDB has become very popular recently due to an increased focus on generative AI and ML. Vector data typically refers to high-dimensional data points, often used in the context of ML models. For example, an image, text, or audio file can be converted into a vector representation (a list of numbers) that captures its essential features. These vectors are used in ML tasks such as similarity search (finding the most similar items), clustering, or classification. For example, if you want to build customer segmentation, vector embeddings can be used to cluster customers into different groups based on their purchasing behavior or preferences. By analyzing the vector representations of customers' purchase histories or interactions with a website, businesses can identify distinct clusters of similar customers. This enables them to tailor marketing strategies, personalize offers, or develop targeted products for each specific customer group, enhancing customer satisfaction and loyalty.

VectorDB, or vector databases, represent an emerging category in the database technology landscape, primarily focused on efficiently handling vector data. This data type is often associated with ML, particularly in areas like image recognition, NLP, and recommendation systems.

A vector database's core functionality is storing and managing vector data efficiently. This involves storing the high-dimensional data points and optimizing the database architecture to support quick and efficient querying, often in the form of nearest neighbor search.

Advanced vector databases may incorporate ML models directly into the database, enabling on-the-fly transformation of raw data (like images or text) into vectors, which can then be stored or queried.

A common use case is finding items similar to a given query item. For instance, the database can quickly retrieve images most similar to the query image in an image search. Vector databases can power recommendation engines by matching user profiles with product vectors to suggest relevant items. They can efficiently handle and query large-scale text data transformed into vector space for various NLP applications. The following are the pros of **VectorDB**:

- **Speed and efficiency:** Tailored to handle high-dimensional data, vector databases can perform similarity searches much faster than traditional databases.
- **Scalability:** They are designed to scale with the size of the data, which is crucial in ML applications where datasets are often large.
- **Integration with ML/AI pipelines:** Seamless integration with ML workflows, allowing direct querying and manipulation of vector data.

Let's look at some of the cons of **VectorDB** as well:

- **Complexity:** The management and indexing of high-dimensional vector data can be complex.
- **Resource intensive:** These databases might require significant computational resources, especially for large-scale datasets.

- **Emerging technology:** Being relatively new, the ecosystem around vector databases might not be as mature as traditional databases, which can be a consideration for enterprise adoption.

Vector databases are part of a broader trend towards specialized databases tailored for specific types of data and workloads, particularly in the context of ML and AI. They represent a significant step in evolving database technology to keep pace with data science and analytics advancements. As this technology matures, it's likely to become an integral part of the data infrastructure in organizations heavily invested in ML and AI.

Blockchain data stores

Blockchain technology, commonly associated with cryptocurrencies, offers a revolutionary approach to data management and transaction processing in various sectors beyond finance. Blockchain data stores offer a robust mechanism for decentralized verification, fundamentally altering how transactions are recorded and validated across various sectors. In a blockchain-based land registry system, for instance, every transaction involving property sales or purchases is recorded on a shared ledger, instantly accessible and verifiable by all network participants. This transparency contrasts with traditional centralized systems, where data is managed by a single authority, reducing the risk of fraud and enhancing trust among participants.

The immutability and security features of blockchain further increase its application across industries. In healthcare, for example, blockchain ensures that once patient records are entered into the system, they remain unchanged and secure. This immutability is vital for medical professionals who depend on accurate historical data for treatment decisions. Additionally, the cryptographic security of blockchain protects sensitive health information, allowing access only to authorized users and ensuring patient privacy. These attributes make blockchain an invaluable tool in sectors where data integrity and security are paramount.

To achieve immutability, blockchain networks play a key role, which is a decentralized digital ledger that records transactions across multiple computers in a way that ensures the integrity and security of the data. In a blockchain network, transactions are grouped into blocks, and each block is linked to the previous one, forming a chain. This structure makes it extremely difficult to alter information retroactively without the consensus of the network participants. The following are the types of blockchain networks:

- **Public blockchain:** Ethereum is often used for **decentralized applications (DApps)** and smart contracts. Ethereum is open, and anyone can join and participate in the network. For instance, a developer might create a DApp for **decentralized finance (DeFi)** on the Ethereum network, allowing users to engage in financial transactions without traditional banks.
- **Private blockchain:** This type of blockchain is restricted and controlled by an organization, offering more privacy and control. A pharmaceutical company might use a private blockchain to manage its drug development process. Access to the blockchain is restricted to company researchers and regulators, ensuring sensitive data is kept confidential.
- **Consortium blockchain:** This involves multiple organizations managing a blockchain network, balancing decentralization with control. An example would be a group of shipping companies forming a consortium to manage a shared blockchain. This blockchain could be used to track cargo shipments across the globe, with each company maintaining a node on the network.

Cloud providers like **Amazon Web Services (AWS)** offer blockchain as a service, simplifying the setup and management of blockchain networks. Amazon **Quantum Ledger Database (QLDB)** is an example of a centralized ledger database that provides an immutable and cryptographically verifiable record of transactions. Popular managed blockchain services include **Amazon Managed Blockchain (AMB)**, **R3 Corda**, **Ethereum**, and **Hyperledger**, catering to various needs from financial transactions to supply chain management.

Streaming data processing used to be a niche technology, but now it's becoming common as every organization wants to get fast insight from real-time data processing. Let's learn more about streaming data stores.

Streaming data stores

Streaming data has a continuous data flow with no start and end. Lots of data from various real-time resources, such as stock trading, autonomous cars, smart spaces, social media, e-commerce, gaming, ride apps, and so on, needs to be stored and processed quickly. Netflix provides real-time recommendations based on the content you are watching, and Lyft uses streaming to connect passengers to a driver in real time.

Storing and processing streaming data is challenging as there is a continuous stream of data coming in, and you cannot predict the storage capacity. Along with high volume, streaming data comes with very high velocity, which requires a scalable storage system that can store the data and provide the ability to replay it. Data streams can become very expensive to maintain and complex to manage over time. Popular streaming data storage services are Apache Kafka, Apache Flink, Apache Spark Structured Streaming, Apache Samza, and Amazon Kinesis. AWS provides managed Kafka, known as Amazon Managed Streaming for Kafka. Let's learn more details about streaming data ingestion and storage technology:

- **Amazon Kinesis:** Amazon Kinesis offers three capabilities. The first, **Kinesis Data Streams (KDS)**, is a place to store a raw data stream to perform any downstream processing of the desired records. The second is **Amazon Kinesis Data Firehose (KDF)**, which facilitates transferring these records into common analytic environments like Amazon S3, Elasticsearch, Redshift, and Splunk. Firehose will automatically buffer up all the records in the stream and flush out to the target as a single file or set of records based on either a time or data-size threshold that you can configure or whichever is reached first.
- The third is **Kinesis Data Analytics (KDA)**, which performs analytics on stream records using Apache Flink. The output can subsequently flow into further streams you create to build an entire serverless streaming pipeline.
- **Amazon Managed Streaming for Kafka (MSK):** MSK is a fully managed, highly available, and secure service. Amazon MSK runs applications on Apache Kafka in the AWS cloud without needing Apache Kafka infrastructure management expertise. Amazon MSK provides a managed Apache Kafka cluster with a ZooKeeper cluster to maintain configuration and build a producer/consumer for data ingestion and processing.

- **Apache Flink:** Flink is another open source platform for streaming data and batch data processing. Flink consists of a streaming dataflow engine that can process bounded and unbounded data streams. A bounded data stream has a defined start and end, while an unbounded data stream has a start but no end. Flink can perform batch processing on its streaming engine and supports batch optimizations.
- **Apache Spark Streaming:** Spark Streaming helps ingest live data streams with high throughput and a fault-tolerant, scalable manner. Spark Streaming divides the incoming data streams into batches before sending them to the Spark engine for processing. Spark Streaming uses DStreams, which are sequences of **resilient distributed datasets (RDDs)**.
- **Apache Kafka:** Kafka is one of the most popular open source streaming platforms that helps you publish and subscribe to a data stream. A Kafka cluster stores a recorded stream in a Kafka topic. A producer can publish data in a Kafka topic, and consumers can take the output data stream by subscribing to the Kafka topic.
- Streaming storage needs to persist a continuous stream of data and provide the ability to maintain the order if required. You will learn more about streaming architecture in the upcoming section, *Streaming data architecture*.

Data storage in the cloud

Cloud data storage is a crucial aspect of modern IT infrastructure, offering scalability, flexibility, and cost-effectiveness. The leading cloud service providers – AWS, GCP, and Azure – provide various data storage options to cater to different needs, from simple file storage to complex databases and data warehousing solutions. The following lists the key characteristics of cloud data storage across these platforms.

- **AWS:**
 - **Amazon Simple Storage Service (S3):** This is a highly scalable object storage service known for its high data availability, security, and performance. Amazon S3 is versatile, perfect for storing any volume of data applicable in various scenarios like websites, mobile apps, backup and restoration, archival needs, enterprise applications, IoT devices, and big data analytics.
 - **Amazon Elastic Block Store (EBS):** EBS offers block-level storage volumes for use with EC2 instances. It's particularly suitable for data that demands consistent and low-latency performance, such as databases or ERP (Enterprise Resource Planning) systems.
 - **Amazon Relational Database Service (RDS):** RDS streamlines the setup, operation, and scaling of a relational database in the cloud. It offers a cost-effective solution with resizable capacity while automating many of the time-consuming tasks associated with database administration.
 - **Amazon S3 Glacier:** This service provides secure, durable, and low-cost cloud storage for archiving and long-term backup. Amazon S3 Glacier is ideal for storing data that is accessed infrequently, offering a solution for long-term data retention.

- GCP:
 - **Google Cloud Storage:** This offers object storage for companies of all sizes. It's highly scalable and flexible, providing secure and durable storage for high-demand applications and workloads.
 - **Persistent Disk:** This provides block storage for Google Compute Engine instances. It offers high-performance SSD and HDD storage that can be attached to instances running in Compute Engine or **Google Kubernetes Engine (GKE)**.
 - **Cloud SQL:** A fully managed database service that makes it easy to set up, maintain, manage, and administer relational databases on Google Cloud.
 - **Google Cloud Bigtable:** A scalable, fully managed NoSQL database service for large analytical and operational workloads.
- Microsoft Azure:
 - **Azure Blob Storage:** This is Azure's object storage solution designed for the cloud. It excels at storing large amounts of unstructured data, such as text or binary data. This includes various types of content like documents, media files, backups, and logs, making it highly versatile for a wide range of uses.
 - **Azure File Storage:** Offers cloud-based, fully managed file shares that are accessible using the standard SMB protocol. This service is particularly useful for businesses looking to migrate their existing on-premises file shares to the cloud environment.
 - **Azure SQL Database:** A comprehensive, fully managed relational database service in the cloud. It provides the capabilities of SQL Server, but without the need for extensive infrastructure and database administration tasks, simplifying database management.
 - **Azure Disk Storage:** This delivers high-performance, reliable block storage for Azure Virtual Machines. Azure Disk Storage includes both SSD and HDD options, catering to a range of requirements from high-speed performance to cost efficiency.

Cloud data storage services across these platforms are designed to provide secure, scalable, and accessible storage solutions, accommodating various applications and use cases. Each service has its specific strengths, making them suitable for different performance, scalability, data access, and cost requirements.

Once you ingest and store data, processing the data in the desired structure is essential to visualize and analyze it for business insights. Let's learn more about data processing and transformation.

Processing data and performing analytics

Data analytics is the process of ingesting, transforming, and visualizing data to discover valuable insights for business decision making. Over the previous decade, more data has been collected than ever before, and customers are looking for greater insights into their data.

These customers also want these insights in the least amount of time, sometimes even in real time. They want more ad hoc queries to answer more business questions. To answer these questions, customers need more powerful and efficient systems.

Batch processing typically involves querying large amounts of cold data. In batch processing, it may take hours to get answers to business questions. For example, you may use batch processing to generate a billing report at the end of the month. Stream processing in real time typically involves querying small amounts of hot data, and it takes only a short amount of time to get answers. MapReduce-based systems such as Hadoop are examples of platforms that support the batch jobs category, while data warehouses are examples of platforms that support the query engine category.

Streaming data processing activities ingest a data sequence and incrementally update functions in response to each data record. Typically, they ingest continuously produced streams of data records, such as metering data, monitoring data, audit logs, debugging logs, website clickstreams, and location-tracking events for devices, people, and physical goods.

The following diagram illustrates a data lake pipeline for processing, transforming, and visualizing data using the AWS cloud tech stack:

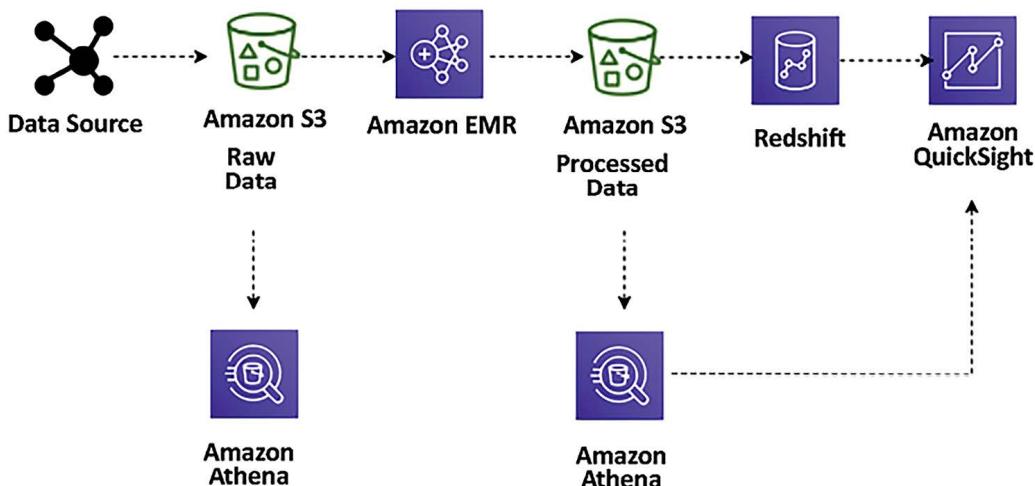


Figure 12.5: Data lake ETL pipeline for big data processing

Here, the ETL pipeline uses Amazon Athena for ad hoc querying of data stored in Amazon S3. The data ingested from various data sources (for example, web application servers) generates log files that persist into S3. These files are then transformed and cleansed into a set form required for meaningful insights using Amazon Elastic MapReduce (EMR) and loaded into Amazon S3. Amazon EMR provides a managed Hadoop server in the cloud to perform data processing using various open source technologies such as Hive, Pig, and Spark.

These transformed files are loaded into Amazon Redshift using the COPY command and visualized using Amazon QuickSight. Using Amazon Athena, you can query the data directly from Amazon S3 when the data is stored and after transformation (with aggregated datasets). You can visualize the data from Athena in Amazon QuickSight. You can easily query these files without changing your existing dataflow.

Let's look at some popular tools for data processing.

Technology choices for data processing and analysis

The following are some of the most popular data processing technologies that help you to perform transformation and processing for a large amount of data:

- **Apache Hadoop** uses a distributed processing architecture in which a task is mapped to a cluster of commodity servers for processing. Each piece of work distributed to the cluster servers can be run or re-run on any server. The cluster servers frequently use the HDFS to store data locally for processing. The Hadoop framework takes a big job, splits it into discrete tasks, and processes them in parallel. It allows for massive scalability across an enormous number of Hadoop clusters. It's also designed for fault tolerance, where each worker node periodically reports its status to a primary node, and the primary node can redistribute work from a cluster that doesn't respond positively. Some of the most popular frameworks used with Hadoop are **Hive**, **Presto**, **Pig**, and **Spark**.
- **Apache Spark** is an in-memory processing framework. Apache Spark is a massively parallel processing system with different executors that can take apart a Spark job and run tasks in parallel. To increase the parallelism of a job, add nodes to the cluster. Spark supports batch, interactive, and streaming data sources. Spark uses **directed acyclic graphs (DAGs)** for all the stages during the execution of a job. The DAGs can keep track of your data or lineage transformations during the jobs and efficiently minimize the I/O by storing the DataFrames in memory. Spark is also partition-aware to avoid network-intensive shuffles.
- **Hadoop User Experience (HUE)** enables you to run queries and scripts on your cluster through a browser-based **user interface (UI)** instead of the command line. HUE provides the most common Hadoop components in a UI. It enables browser-based viewing and tracking of Hadoop operations. Multiple users can access the cluster via HUE's login portal, and administrators can manage access manually or with **Lightweight Directory Access Protocol (LDAP)**, **Pluggable Authentication Modules (PAM)**, **Simple and Protected GSSAPI Negotiation Mechanism (SPNEGO)**, OpenID, OAuth, and **Security Assertion Markup Language 2.0 (SAML2)** authentication. HUE allows you to view logs in real time and provides a metastore manager to manipulate Hive metastore contents.
- **Pig** is typically used to process large amounts of raw data before storing it in a structured format (SQL tables). Pig is well suited to ETL operations such as data validation, loading, transformation, and combining data from multiple sources in multiple formats. In addition to ETL, Pig supports relational operations such as nested data, joins, and grouping. Pig scripts can input unstructured and semi-structured data (such as web server logs or clickstream logs). In contrast, Hive consistently enforces a schema on input data. Pig Latin scripts contain instructions on filtering, grouping, and joining data, but Pig is not intended to be a query language. Hive is better suited to querying data. The Pig script compiles and runs to transform the data based on the instructions in the Pig Latin script.
- **Hive** is an open source data warehouse and query package that runs on top of a Hadoop cluster. Being able to use SQL is a skill that helps the team easily transition into the big data world. Hive uses a SQL-like language called **Hive Query Language (HQL)**, making it easy to query and process data in a Hadoop system. Hive abstracts the complexity of writing programs in a coding language like Java to perform analytics jobs.

- **Presto** is a Hive-like query engine, but it is much faster. It supports the **American National Standards Institute (ANSI)** SQL standard, which is easy to learn and the most popular skill set. Presto supports complex queries, joins, and aggregation functions. Unlike Hive or MapReduce, Presto executes queries in memory, which reduces latency and improves query performance. You need to be careful while selecting the server capacity for Presto, as it needs to have high memory. A Presto job will restart in the event of memory spillover.
- **HBase** is a NoSQL database developed as a part of the open source Hadoop project. HBase runs on the HDFS to provide non-relational database capabilities for the Hadoop ecosystem. HBase helps to store large quantities of data in a columnar format with compression. Also, it provides a fast lookup because large portions of the data cache are kept in memory while cluster instance storage is still used.
- **Apache Zeppelin** is a web-based editor for data analytics built on top of the Hadoop system, also known as a Zeppelin notebook. It uses the concept of an interpreter for its backend language and allows any language to be plugged into Zeppelin. Apache Zeppelin includes some basic charts and pivot charts. It's very flexible in terms of any output from any language backend that can be recognized and visualized.
- **Ganglia** is a Hadoop cluster monitoring tool. However, you need to install Ganglia on the cluster during launch. The Ganglia UI runs on the primary node, which you can see using an SSH tunnel. Ganglia is an open source project designed to monitor clusters without impact on their performance. Ganglia can help to inspect the performance of the individual servers in your cluster and the performance of clusters as a whole.
- **JupyterHub** is a multi-user Jupyter notebook. Jupyter Notebook is one of the most popular tools among data scientists to perform data engineering and ML. The JupyterHub notebook server provides each user with a Jupyter Notebook web-based IDE. Multiple users can use their Jupyter notebooks simultaneously to write and execute code for exploratory data analytics.

Data processing in the cloud

Data processing in the cloud is a fundamental aspect of modern big data and analytics strategies. Three major cloud service providers—AWS, GCP, and Azure—offer various data processing services, each with unique features and capabilities. The following are some unique features of each of them:

- AWS Data Processing Services:
 - **Amazon EMR**: This provides a cloud-native Hadoop environment, supporting a wide range of big data frameworks like Apache Spark, Hadoop, HBase, and Presto. EMR is ideal for processing large datasets, and it offers flexibility by separating compute and storage, allowing for cost-efficient scaling.
 - **AWS Glue**: This is a fully managed ETL service that simplifies data preparation for analytics. It automates the cumbersome data preparation work, generates ETL scripts, and facilitates data movement between various AWS services. Glue is particularly effective for data cataloging and job scheduling.
 - **Amazon Athena**: A serverless, interactive query service that allows SQL queries directly on data stored in Amazon S3. It is highly useful for ad hoc data analysis and BI querying, with no infrastructure management required.

- GCP Data Processing Services:
 - **Google BigQuery:** This is a fully managed, serverless data warehouse solution designed for rapid, cost-efficient SQL querying across extensive datasets. BigQuery is particularly geared towards real-time analytics and is capable of handling streaming data effectively.
 - **Cloud Dataflow:** A fully managed service dedicated to processing data in both stream and batch modes. Built on Apache Beam, Cloud Dataflow offers a unified programming model, simplifying the development of parallel data processing pipelines. It's adept at handling a range of tasks from complex ETL processes to batch and real-time streaming workloads.
 - **Cloud Dataprep:** An advanced data service that allows users to visually explore, clean, and prepare both structured and unstructured data for analysis. Seamlessly integrated with BigQuery and Cloud Dataflow, Cloud Dataprep enhances the capabilities of data exploration and transformation.
- Azure Data Processing Services:
 - **Azure HDInsight:** A fully managed cloud service that makes it easy to process massive amounts of data with popular open source frameworks such as Apache, Hadoop, Spark, Kafka, and HBase. It suits various scenarios like ETL, data warehousing, ML, and IoT.
 - **Azure Databricks:** A fast, easy, and collaborative Apache Spark-based analytics platform. It integrates deeply with other Azure services and provides a unified platform for ETL processes, streaming analytics, ML, and data warehousing.
 - **Azure Synapse Analytics:** This is a comprehensive analytics service that merges the capabilities of big data and data warehousing. It provides a cohesive experience for ingesting, preparing, managing, and delivering data for instant BI and ML applications. Azure Synapse Analytics enables the simultaneous querying of both data lakes and databases, streamlining data analysis processes.

Each cloud provider's data processing services are designed to meet specific needs in the data life cycle, from processing and transforming large datasets to interactive querying and real-time analytics. This diversity ensures businesses can choose the most suitable tools and platforms according to their specific data processing requirements and objectives.

Data analysis and processing are huge topics that warrant a book on their own. This section gave a high-level overview of popular and common tools used for data processing. There are many more proprietary and open source tools available. As a solutions architect, you must be aware of various available tools to make the right choice for your organization's use case.

Business analysts need to create reports and dashboards and perform ad hoc queries and analyses to identify data insights. Let's learn about data visualization in the next section.

Visualizing data

Data insights are used to answer important business questions such as revenue by customer, profit by region, or advertising referrals by site, among many others. In the big data pipeline, enormous amounts of data are collected from various sources. However, it is difficult for companies to find information about inventory per region, profitability, and increases in fraudulent account expenses. Some of the data you continuously collect for compliance purposes can also be leveraged for generating business.

The two significant challenges of BI tools are the cost of implementation and the time it takes to implement a solution. Let's look at some technology choices for data visualization.

Technology choices for data visualization

The following are some of the most popular data visualization platforms, which help you prepare reports with data visualization as per your business requirements:

- **Amazon QuickSight** is a cloud-based BI tool for enterprise-grade data visualizations. It comes with a variety of visualization graph presets such as line graphs, pie charts, treemaps, heat maps, and histograms. Amazon QuickSight has a data-caching engine known as a **Super-fast, Parallel, In-memory Calculation Engine (SPICE)**, which helps render visualizations quickly. You can also perform data preparation tasks such as renaming and removing fields, changing data types, and creating new calculated fields. QuickSight also provides ML-based visualization insights and other ML-based features, such as auto forecast predictions.
- **Kibana** is an open source data visualization tool for stream data visualization and log exploration. Kibana offers close integration with Elasticsearch and uses it as a default option to search for data on top of the Elasticsearch service. Like other BI tools, Kibana also provides popular visualization charts such as histograms, pie charts, and heat maps, and offers built-in geospatial support.
- **Tableau** is one of the most popular BI tools for data visualization. It uses a visual query engine, which is a purpose-built engine, to analyze big data faster than traditional queries. Tableau offers a drag-and-drop interface and the ability to blend data from multiple resources.
- **Spotfire** uses in-memory processing for faster response times, enabling extensive datasets from various resources. It allows you to plot your data on a geographical map and share it on Twitter. With Spotfire recommendations, it inspects your data automatically and suggests how to visualize it best.
- **Jaspersoft** enables self-service reporting and analysis. It also offers drag-and-drop designer capabilities.
- **Power BI** is a popular BI tool provided by Microsoft. It provides self-service analytics with a variety of visualization choices.

Data visualization is an essential and massive topic for solutions architects. As a solutions architect, you need to be aware of the available tools and make the right choice per your business requirements for data visualization.

You have learned about various data pipeline components, from ingestion, storage, and processing, to visualization. In the next section, let's put them together and learn how to orchestrate a big data architecture.

Designing big data architectures

Big data solutions are comprised of data ingestion, storage transformation, data processing, and visualization in a repeated manner to run daily business operations. You can build these workflows using the open source or cloud technologies you learned about in previous sections.

First, you need to learn which architectural style is right for you by working backward from the business use case. You need to understand the end user of your big data architecture and create a user persona to understand the requirements better. To identify the key personas you are targeting with big data architecture, you need to understand some of the following points:

- Which teams, units, or departments inside your organization are they a part of?
- What is their level of data analysis and data engineering proficiency?
- What tools do they typically use?
- Do you need to cater to the organization's employees, customers, or partners?

For your reference, taking an example of a retail store chain analysis, you may identify the following personas:

- The **product manager** persona, who owns a product line/code but only sees turnover for their product.
- The **store manager** persona, who wants to know the sales turnover and product mix for a single store (only able to see their store).
- The **admin** persona, who wants to have access to all data.
- The **data analyst**, who wants to access all data with PII data redacted.
- The **customer retention managers**, who want to understand repeated customer traffic.
- **Data scientists** need access to raw and processed data to build recommendations and forecasts.

Once you have a clear understanding of your user persona, the next step is to identify the business use cases these personas aim to address. Some examples include:

- **Customer spending trends:** Analyze how many customers are increasing or decreasing their spending over time. Characterize these customers based on their spending patterns.
- **Growth categories among higher spenders:** Identify which product or service categories are witnessing faster growth among customers who are spending more over time.
- **Decline categories among lower spenders:** Determine the categories where there is a noticeable decline in engagement among customers who are spending less over time.
- **Impact of demographics on spending:** Investigate which demographic factors, such as household size, presence of children, or income level, influence customer spending habits. Also, assess which demographic factors impact engagement with specific product or service categories.

- **Effectiveness of direct marketing:** Explore whether there is evidence to suggest that direct marketing campaigns lead to improved overall customer engagement.
- **Cross-category impact of direct marketing:** Assess whether direct marketing efforts in one category have a positive effect on customer engagement in other categories.

While you get details on the use case, the essential aspect of building your data architecture is to understand access patterns and data retention, which can be analyzed by using the following queries:

- How often do key users and personas run their reports, queries, or models?
- What is their expectation for data freshness?
- What is their expectation of data granularity?
- What portion of data is most frequently accessed for analysis?
- How long do you intend to retain data for analysis?
- At what point can data age out of the data lake environment?

There is always some sensitivity attached when you deal with data. Each country and area has its local regulatory compliance requirements, which solutions architects need to understand, such as:

- What compliance requirements does your business have?
- Are you subject to data locality, privacy, or redaction requirements?
- Who is authorized to see which records and which attributes in the dataset?
- How will you enforce the deletion of records on request?
- Where can you store data, for example, local to geolocation, county, or global?

As a data architect, you must also consider the return on investment and how it will help overall business decisions. To understand, you may want to go through the following points:

- What primary business processes and decisions does your data lake support?
- What level of granularity is required for these decisions?
- What is the impact of data latency on business decisions?
- How do you plan to measure success?
- What is the expected return on the time and material invested?

Ultimately, you want to build a data architecture where you can provide flexibility to make technology choices, for example, by using the best of cloud-based managed services and open source technologies to capitalize on existing skills and investments. You want to build big data solutions to use parallelism to achieve high performance and scalability. It is best to make sure any components of your big data pipeline can scale in or scale out independently so that you can adjust it according to different business workloads.

To utilize the full potential of your solution, you want to provide interoperability with existing applications so that components of the big data architecture are also used for ML processing and enterprise BI solutions. It will enable you to create an integrated solution across data workloads. Let's learn about some big data architecture patterns.

Data lake architecture

A data lake serves as a centralized repository that accommodates both structured and unstructured data, encompassing the diverse data types present in a corporation. It has emerged as a solution for transferring all enterprise data into a cost-effective storage system, like Amazon S3. In a data lake, data can be accessed through generic APIs and open file formats, including Apache Parquet and Optimized Row Columnar (ORC). This storage method preserves data in its original form, utilizing open source file formats, thereby facilitating direct analytics and ML applications.

The data lake is becoming a popular way to store and analyze large volumes of data in a centralized repository. Data can be stored as is in its current format, and you don't need to convert data into a predefined schema, which increases the data ingestion speed. As illustrated in the following diagram, the data lake is a single source of truth for all data in your organization:

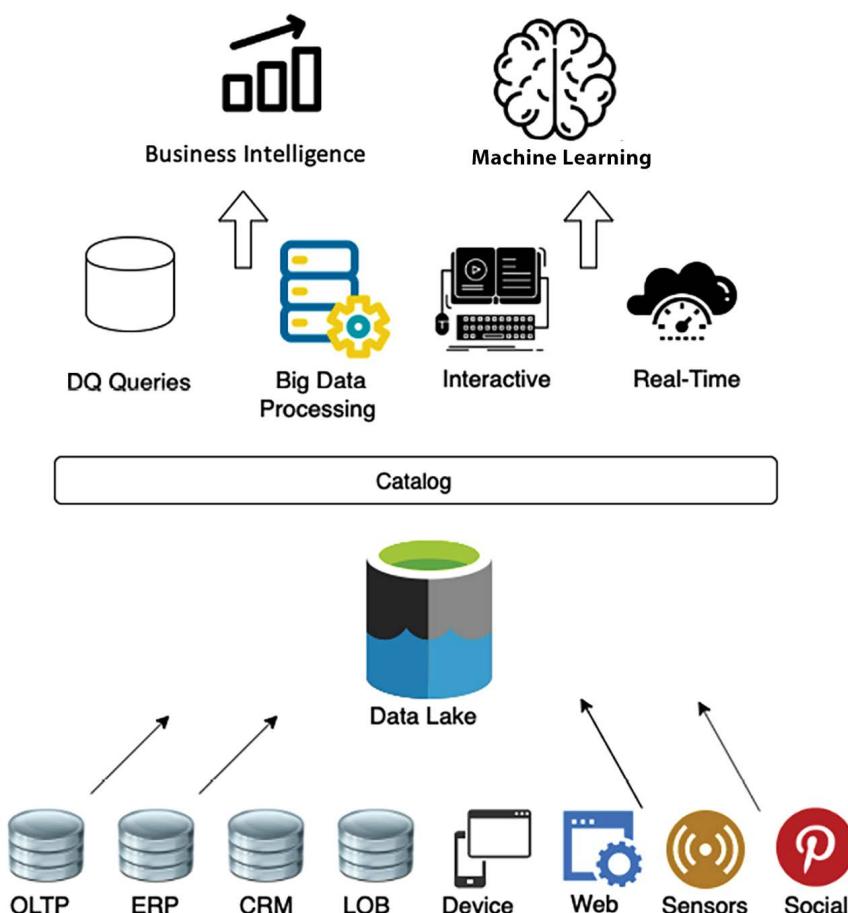


Figure 12.6: Object store for data lake

The following are the benefits of a data lake:

- **Data ingestion from various sources:** Data lakes let you store and analyze data from multiple sources, such as relational and non-relational databases and streams, in one centralized location for a single source of truth. This answers questions such as *Why is the data distributed in many places?* and *Where is the single source of truth?*
- **Collecting and efficiently storing data:** A data lake can ingest any data structure, including semi-structured and unstructured data, without needing schema. This answers questions such as: *How can I ingest data quickly from various sources and in multiple formats and store it efficiently at scale?*
- **Scale up with the volume of generated data:** Data lakes allow you to separate the storage and compute layers to scale each component separately. This answers questions such as: *How can I scale up with the volume of data generated?*
- **Applying analytics to data from different sources:** With a data lake, you can determine the schema on reading and create a centralized data catalog on data collected from various resources. This enables you to perform quick ad hoc analysis. This answers questions such as: *Can I apply multiple analytics and processing frameworks to the same data?*

It would help if you had an unlimited scalable data storage solution for your data lake. Decoupling your processing and storage provides many benefits, including the ability to process and analyze the same data with various tools. Although this may require an additional step to load your data into the right tool, Amazon S3, as your central data store, provides even more benefits than traditional storage options. The following diagram provides a view of the data lake using AWS services:

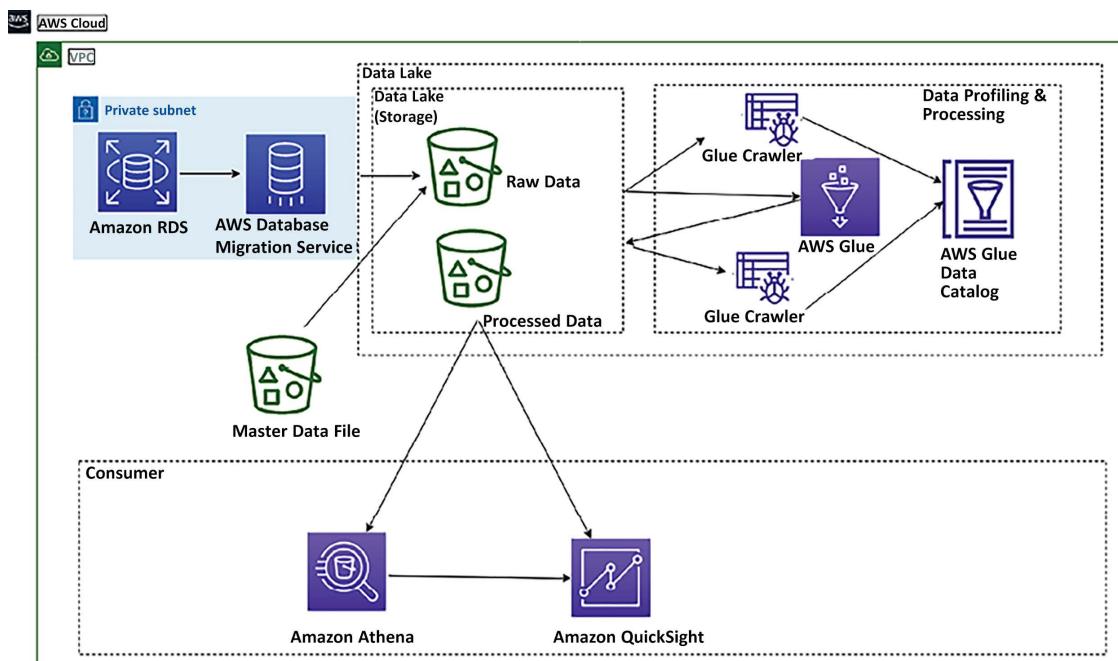


Figure 12.7: Data lake architecture in the AWS platform

The preceding diagram depicts a data lake using Amazon S3 storage. Data is ingested into centralized storage from various resources such as relational databases and master data files. In the data lake's raw layer, all data is kept in its original format. This data then undergoes cataloging and transformation via the AWS Glue service. AWS Glue is a serverless solution for data cataloging and ETL processes, built on the Spark framework within the AWS cloud platform. Here, the AWS Glue crawler helps in cataloging data stores. It automatically scans your data sources, identifies data formats, and infers schemas, creating and populating a data catalog with metadata information. The crawler classifies the data to understand its format and structure and creates table definitions in the data catalog, which makes it easy to build queries for data analytics. Once transformed, this data is stored in the data lake's processed layer, making it available for various consumption purposes.

Data engineers can run ad hoc queries using Amazon Athena, a serverless query service built on top of managed Presto instances, and use SQL to query the data directly from Amazon S3. Business analysts can use Amazon QuickSight, Tableau, or Power BI to build visualizations for business users or load selective data in Amazon Redshift to create a data warehouse mart. Finally, data scientists can consume this data using Amazon SageMaker to perform ML.

One tool cannot do everything. You need to use the right tool for the right job, and data lakes enable you to build a highly configurable big data architecture to meet your specific needs. Business problems must be narrower, deeper, and more complex for one tool to solve everything, especially big data and analytics.

However, with time, organizations realized that data lakes have their limitations. As data lakes use cheap storage, organizations store as much of their data as they can in data lakes, providing the flexibility of open, direct access to files. Quickly, data lakes started becoming **data swamps** due to data quality issues and granular data security. However, to address the data lake's performance and quality issues, organizations process a small subset of data in the data lake to a downstream data warehouse to use in BI applications for important decisions.

The dual system architecture between a data lake and a data warehouse requires continuous data engineering to maintain and process data between these two systems. Each step in data processing carries the risk of failures that can compromise data quality. Additionally, maintaining consistency between the data lake and the data warehouse can be both challenging and expensive. Users face the burden of paying double for storage—once for the data stored in the lake and again for data replicated in the warehouse. This is in addition to the ongoing costs associated with continuous data processing.

To address the dual-system problem, a new type of architecture called the data lakehouse has been discovered. Let's learn more about lakehouse architecture.

Lakehouse architecture

The lakehouse architecture has emerged as a solution to bridge the gaps between traditional data lakes and data warehouses, integrating the strengths of both. This architecture is designed to harness the expansive storage capacity of data lakes for ingesting and keeping vast quantities of data in open formats, which are essential for analytics. Simultaneously, it aims to provide the ease of SQL-based querying and the reliability associated with data warehouses. Key characteristics of lakehouse architecture include:

- **Data storage in open-data formats:** Lakehouse architecture stores data in open formats, facilitating interoperability and flexibility in data processing and analytics.
- **Decoupled storage and compute:** It separates storage and computing resources, allowing independent scaling and optimization of each, leading to cost efficiency and performance improvement.
- **Transactional guarantees:** Ensuring data integrity, lakehouse architecture provides transactional guarantees, akin to those in traditional database systems, supporting reliable concurrent access and modifications.
- **Support for diverse consumption needs:** Designed to cater to a wide range of data consumption requirements, lakehouse architecture accommodates different data analytics and processing approaches, from batch to real-time streaming.
- **Secure and governed:** The architecture emphasizes security and governance, ensuring that data access is controlled, and compliance with data privacy regulations is maintained.
- **Unified platform:** Lakehouse architecture provides a unified platform for various data operations, from ETL processes and ML to BI and reporting, eliminating the need for disparate systems.
- **Enhanced query performance:** By leveraging techniques like indexing, caching, and data clustering, lakehouse architecture improves query performance, making it suitable for complex analytical workloads.
- **Cost-effective scalability:** The architecture offers cost-effective scalability options, balancing the need for performance with budgetary constraints, especially beneficial for growing data volumes.
- **Flexible data management:** Lakehouse architecture supports flexible data management practices, accommodating evolving data schemas and structures, making it ideal for agile and evolving business environments.

The lakehouse architecture represents a significant evolution in data management, offering a comprehensive, scalable, and efficient approach to handling vast and diverse datasets while ensuring data integrity, security, and easy accessibility.

The following diagram shows a sample lakehouse architecture using Redshift Spectrum for data sharing. Amazon Redshift Spectrum provides the ability to query data from the data lake without storing data in the data warehouse. Suppose you were already using Amazon Redshift for data warehousing. In that case, you don't need to load all the data into the Amazon Redshift cluster. Still, you can use Spectrum to query data directly from the Amazon S3 data lake and combine it with data warehouse data.

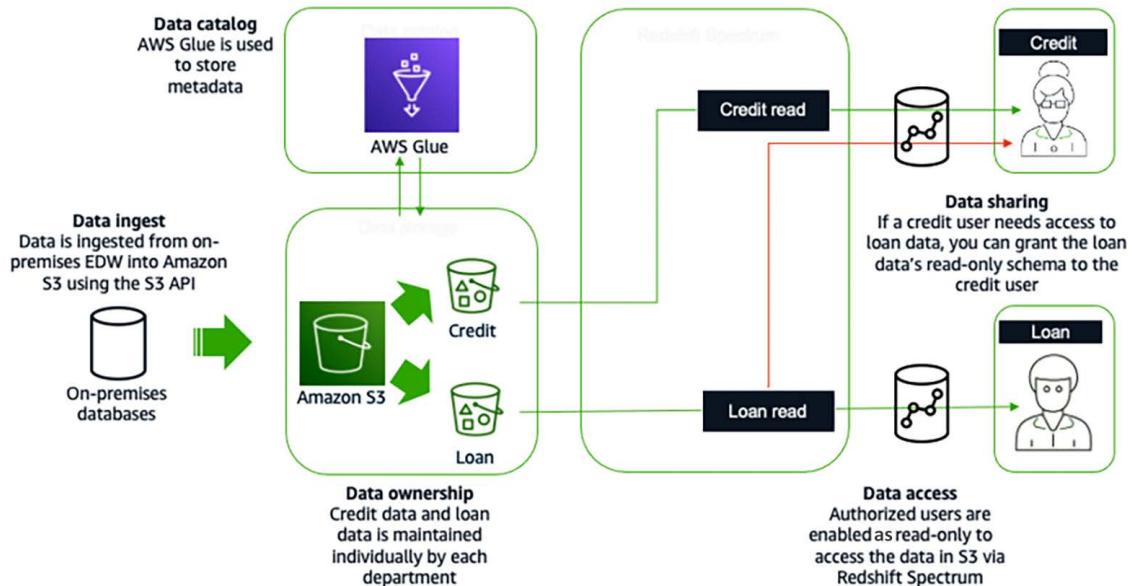


Figure 12.8: Lakehouse architecture in the AWS cloud platform using Redshift Spectrum

Data is ingested from an on-premises enterprise data warehouse (EDW) into S3 using the S3 API in the preceding diagram. AWS Glue stores the metadata and the credit and loan data individually. Data analysts in the loan department would be granted read-only access to the loan data for data access. Similarly, credit analysts would be granted read-only access to the credit data. For data sharing, if a credit analyst needs access to the loan data, the credit analyst can be given the loan data's read-only schema.

Lakehouse architecture has benefits; however, more is needed for large organizations with a complex application landscape driven by geographically separated business units. These business units have built data lakes and warehouses as their analytical sources. Each business unit may merge multiple internal application data lakes to support their business. Centralized enterprise data lakes or data lakehouses are challenging to achieve as the pace of change is generally low, and it isn't easy to meet all requirements across different business units. To handle this problem, you need domain-oriented decentralized data ownership and architecture. That's where data mesh comes into the picture. Let's learn more about data mesh architecture.

Data mesh architecture

The major difference between data mesh and data lake architecture is that data is intentionally left distributed rather than trying to combine multiple domains into a centrally managed data lake. Data mesh provides a pattern that allows a large organization to connect multiple data lakes/lakehouses within large enterprises and facilitate sharing with partners, academia, and even competitors.

Data mesh represents a significant shift in both architecture and organizational approaches toward managing extensive analytical datasets. It is built upon four fundamental principles:

- **Domain-oriented decentralization of ownership and architecture:** This principle emphasizes decentralizing data ownership and architecture decisions to specific business domains. It encourages individual domains to take responsibility for their data, leading to more tailored and effective data solutions.
- **Data served as a product:** Treating data as a product means it is maintained, improved, and presented with the end user in mind. It shifts the focus from data as a mere resource to a valuable asset that provides utility and solves user problems.
- **Federated data governance with centralized audit controls:** This principle strikes a balance between decentralized data management and the need for overarching governance. It allows for domain-specific data governance while maintaining centralized controls for auditing and compliance.
- **Common access that makes data consumable:** Ensuring data is accessible and usable across the organization, this principle focuses on creating a common framework that enables easy and efficient data consumption.

It encourages data-driven agility and supports domain-local governance through a lightweight centralized policy. Data mesh provides better ownership by isolating data resources with clear accountability. The core concept of data mesh is to feature data domains as nodes in data lake accounts.

A data producer contributes one or more data products to a central catalog in a data mesh account where federated data governance is applied to sharing data products, delivering discoverable metadata and audibility. A data consumer searches for a catalog and gains access to a data product by accepting a resource share via the data mesh pattern. The following is a data mesh architecture in the AWS cloud:

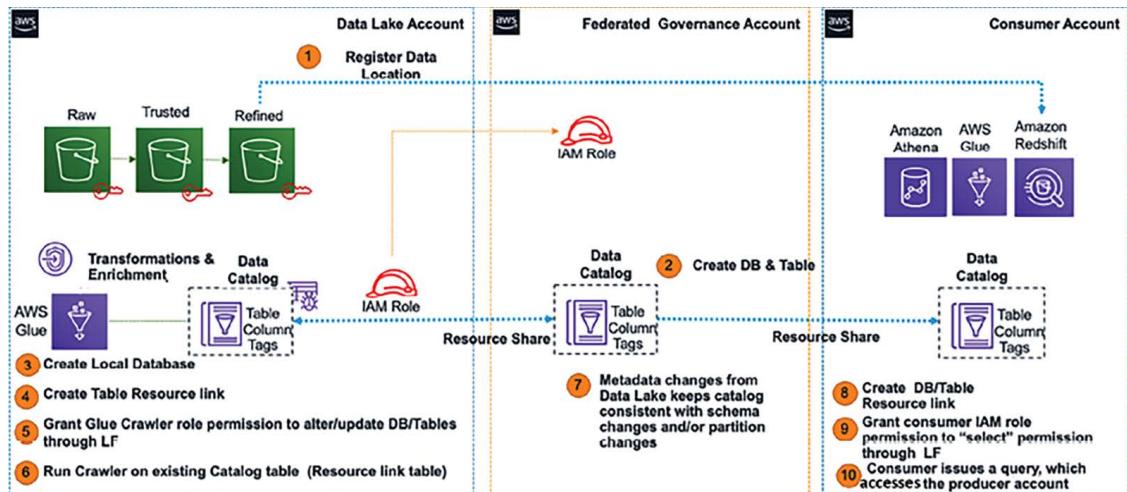


Figure 12.9: Data mesh architecture on the AWS cloud platform

The following are the components implemented to build a data mesh, as shown in the preceding diagram:

- The central AWS account is where data products are registered, comprising databases, tables, columns, and rows.
- Access control tags and tag access policies are managed centrally.
- It stores data permissions that implement sharing with a consumer. Permissions can be direct or based on tags.
- Applies security and governance policies to producer and consumer accounts and their published data products.

With a data mesh architecture, you can accelerate the independent delivery of the business domain lakehouses. Data mesh increases data security and compliance within domains and enables self-service data product creation, discovery, and subscription, allowing consumers to access data products transparently. There is a growing need to provide fast insight and act quickly based on customer needs, which makes streaming data analytics an essential aspect of any business. Let's learn more details about streaming data analytics architecture.

Streaming data architecture

Streaming data, a rapidly expanding segment of data, requires the capability to ingest and swiftly process real-time data from a variety of sources. These sources include video, audio, application logs, website clickstreams, and IoT telemetry data, all aimed at delivering prompt business insights. The typical use cases for streaming data follow a consistent pattern:

1. **Data generation:** Sources continuously produce data.
2. **Ingestion:** This data is then delivered through an ingestion stage to a streaming storage layer.
3. **Stream storage:** In this layer, the incoming data is durably captured and made accessible for real-time processing.
4. **Stream processing:** Here, the data residing in the storage layer is processed. This processing might involve filtering, aggregating, or analyzing the data as it streams.
5. **Data output:** The processed data is then dispatched to a designated destination, which could be a database, a data lake, or another storage solution, for further use or long-term storage.

This flow ensures that data is not only captured as it is generated but also processed in a timely manner, leading to quicker decision making and more immediate business insights.

Streaming data architecture is different as it needs to process a continuous massive data stream with very high velocity. Often, this data is semi-structured and needs a lot of processing to get actionable insights. While designing streaming data architecture, you need to quickly scale data storage while getting real-time pattern identification from time-series data.

It would be best to think about the producer who generated a stream of data, such as IoT sensors, how to store and process the data using a real-time data processing tool, and finally, how to query the data in real time. The following diagram shows a streaming data analytics pipeline using a managed service on the AWS platform:

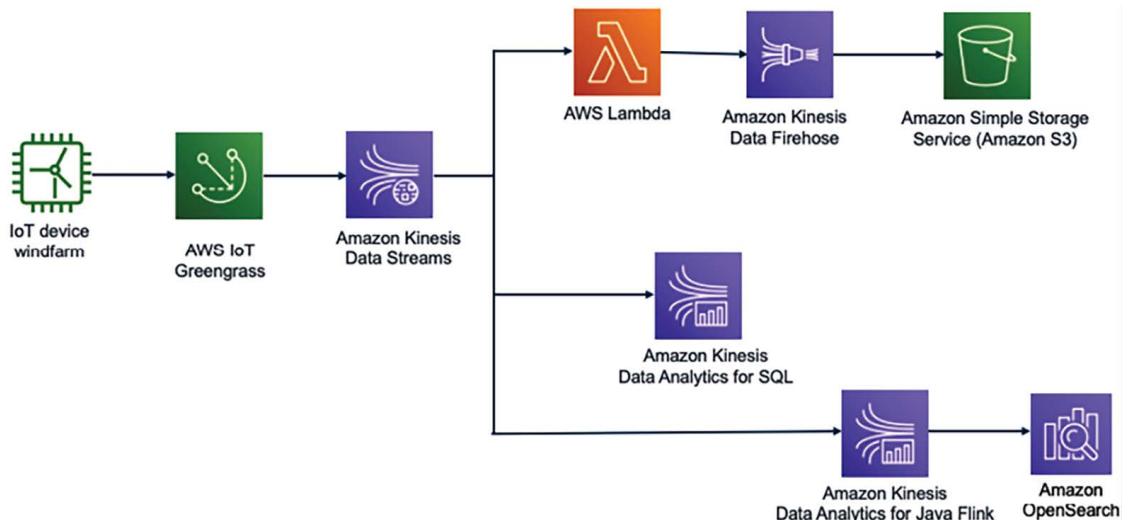


Figure 12.10: Streaming data analytics for IoT data

In the preceding diagram, data is ingested from the wind farm to understand the health and speed of a wind turbine. It's important to control wind turbines in real time to avoid costly repairs in the case of high wind speeds beyond the wind turbine's limit.

The wind turbine data is ingested into Kinesis Data Streams using AWS IoT Greengrass. Kinesis Data Streams can retain the streaming data for up to a year and provide replay capability. These are subjected to the fan-out technique to deliver the data to multiple resources, where you can message data using Lambda and store it in Amazon S3 for further analytics using Amazon Kinesis Firehose.

You can perform real-time queries on streaming data using simple SQL queries with Kinesis Data Analytics for SQL and you can automate a data pipeline to transform streaming data in real time using Kinesis Data Analytics for Java Flink and store the processed data in Amazon OpenSearch to get data insights. You can also add Kibana to OpenSearch to visualize the wind turbine data in real time.

The preceding solution is data agnostic and easily customizable, enabling customers to quickly modify pre-configured defaults and start writing code to include their specific business logic.

Choosing the right big data architecture

Choosing between data lake, lakehouse, and data mesh architectures depends on your specific business requirements, data strategy, and technical capabilities. Each architecture offers unique benefits and is suited for different data management and analytics scenarios. To aid in making the right choice, the following list highlights the benefits, important considerations, and ideal use cases for each type of architecture:

- **Data lake architecture:** A data lake is primarily intended for the storage of large volumes of raw data in its original format.
 - **Benefits:** It provides high scalability and flexibility in handling various data types. It's cost-effective for storing large amounts of data and can be used as a central repository for all organizational data.
 - **Considerations:** Without proper governance, data lakes can become unmanageable ("data swamps"). They require careful management to ensure data quality and accessibility.
 - **Use cases:** It is suitable for big data analytics, ML, and situations where you need to store and analyze large volumes of diverse data at a low cost. It is particularly suitable for situations where there is a requirement to store diverse types of data – including structured, semi-structured, and unstructured – without having a predetermined schema at the point of data entry.
- **Lakehouse architecture:** This combines elements of both data lakes and data warehouses.
 - **Benefits:** It aims to provide the low-cost scalability of data lakes with the robust schema and performance optimization of data warehouses. It offers a unified platform for all types of data processing and analytics, reducing data silos. It also supports ACID transactions and schema enforcement, improving data reliability and quality.
 - **Considerations:** Implementing a lakehouse architecture can be complex, requiring integrating various components and ensuring consistency and reliability across different workloads.
 - **Use cases:** It is best for organizations requiring big data processing and traditional BI from a single platform. It's ideal for use cases that need real-time analytics and reporting on large and diverse datasets.
- **Data mesh architecture:** It focuses on decentralizing the data architecture and ownership. It treats data as a product, with domain-oriented teams owning and providing their data as products to the rest of the organization.
 - **Benefits:** It encourages a more agile and flexible data management and analytics approach. It also promotes data democratization, allowing for faster decision making and innovation within domains.
 - **Considerations:** It requires a cultural shift in how data is managed and shared. It demands strong governance and standardization across domains to ensure data interoperability and quality.
 - **Use cases:** It is suitable for large organizations with multiple independent teams or departments, where different domains produce and consume data.

The following are some key decision factors:

- **Organizational structure:** Consider whether your organization is centralized or decentralized. Data mesh is more suitable for the latter.

- **Data volume and variety:** Data lakes are ideal for massive, diverse datasets, while lakehouses provide a more structured environment for such data.
- **Analytical needs:** A lakehouse might be the best fit if you need real-time analytics combined with big data processing.
- **Governance and compliance:** Assess your data governance, quality, and compliance needs. A lakehouse architecture tends to offer more robust governance mechanisms.
- **Technical expertise:** Implementing and managing a data mesh or lakehouse architecture requires specific technical expertise and resources.

Ultimately, the choice depends on aligning the architecture with your business goals, technical capabilities, and data strategy. Each architecture has its strengths, and the best choice may even be a hybrid approach, depending on your specific requirements.

Big data architecture best practices

You learned about various big data technology and architecture patterns in previous sections. Let's look at the following reference architecture diagram with different layers of a data lake architecture to learn best practices.

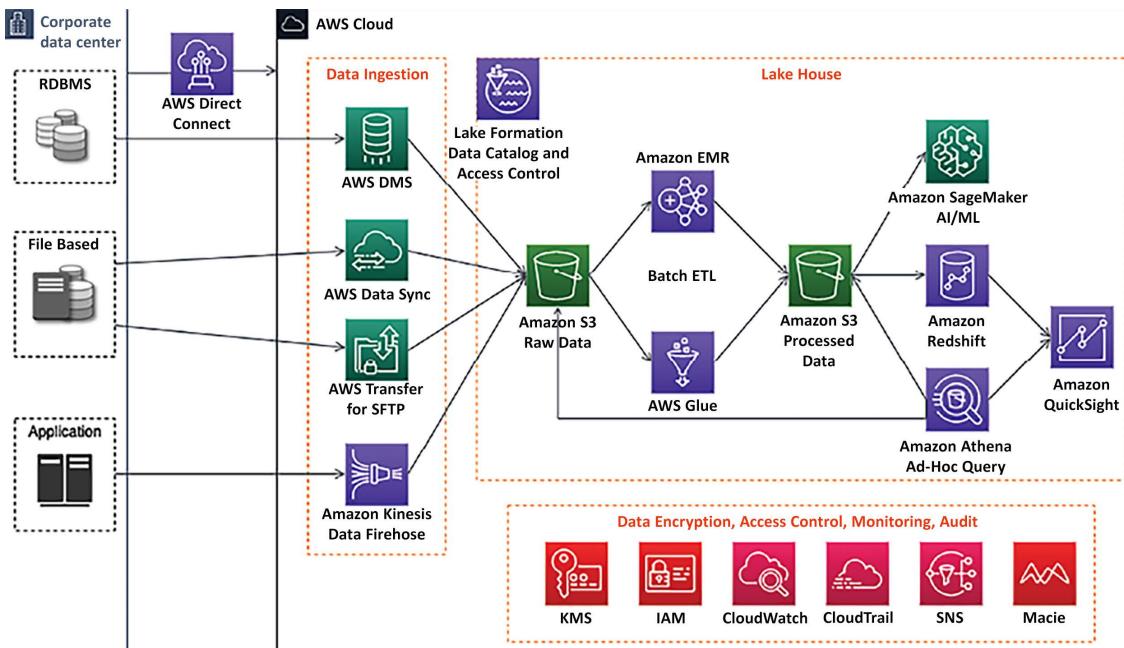


Figure 12.11: Data lake reference architecture

The preceding diagram depicts an end-to-end data pipeline in a data lake architecture using the AWS cloud platform with the following components:

- AWS Direct Connect will set up a high-speed network connection between the on-premises data center and AWS to migrate data. If you have large volumes of archive data, using the AWS Snow family to move it offline is better.

- A data ingestion layer with various components to ingest streaming data using Amazon Kinesis, relational data using AWS Data Migration Service (DMS), secure file transfer using AWS Transfer for Secure Shell File Transfer Protocol (SFTP), and AWS DataSync to update data files between cloud and on-premises systems.
- Centralized data storage for all data using Amazon S3, where data storage has multiple layers to store raw data, processed data, and archive data.
- Amazon Redshift is a cloud-native data warehouse solution with Redshift Spectrum to support lakehouse architecture.
- An ad hoc query functionality using Amazon Athena.
- A quick ETL pipeline based on Spark using AWS Glue.
- Amazon EMR will re-utilize existing Hadoop scripts and other Apache Hadoop frameworks.
- Amazon Lake Formation to build comprehensive data cataloging and granular access control at the data lake level.
- The AI/ML extension with Amazon SageMaker.

Other components include Amazon Key Management Service (KMS) for data encryption, Amazon Identity and Access Management (IAM) for access control, Amazon Macie for PII data detection to adhere to data compliance such as Payment Card Industry Data Security Standard (PCI DSS), CloudWatch to monitor the operation, and CloudTrail to audit the data lake activities.

You need to validate your big data architecture using the following criteria:

- Security:
 - Classify data and define corresponding data protection policies using resource-based access control.
 - Implement a strong identity foundation using user permission and single sign-on (SSO).
 - Enable environment and data traceability for audit purposes.
 - Apply security at all layers and protect data in transit and at rest using SSL and encryption at all layers.
 - Keep people away from data, such as locking down write access to production datasets.
- Reliability:
 - Enforce data hygiene using automated data profiling using data cataloging.
 - Manage the life cycle of data assets, transitioning, and expiration using data tiering between the data warehouse and data lake.
 - Preserve data lineage by maintaining the history of data movement through the data catalog.
 - Design resiliency for analytics pipelines and monitor system SLAs with automated recovery of ETL job failures.

- Performance efficiency:
 - Use data profiling to improve performance with data validation and to build a sanitization layer.
 - Continuously optimize data storage, such as using data compression with a Parquet format, data partition, file size optimization, and so on.
- Cost optimization:
 - Adopt a consumption model and determine whether you need an ad hoc or fast query pattern.
 - Delete out-of-use data; define data retention rules and delete or archive data out of the retention period.
 - Decouple compute and storage with a data lake-based solution.
 - Implement migration efficiency using different migration strategies for various data sources and volumes.
 - Use managed and application-level services to reduce the cost of ownership.
- Operational excellence:
 - Perform operations as code using tools such as CloudFormation, Terraform, and Ansible.
 - Automate operations such as building an orchestration layer with Step Functions or Apache Airflow.
 - Anticipate failure in advance by continuously monitoring and automating the recovery of ETL job failures.
 - Measure the health of your workload.

You can use the preceding checklist as a guide to validate your big data architecture. Data engineering is a vast topic that warrants multiple books to cover each topic in depth.

In this chapter, you learned about various components of data engineering with a popular architecture pattern, which will help you get started and explore the topic in more depth.

Summary

In this chapter, you learned about the big data architecture and components for a big data pipeline design. You learned about data ingestion and various technology choices available to collect batch and stream data for processing. As the cloud is central to storing the vast amounts of data produced today, you learned about the various services available to ingest data in the AWS cloud ecosystem.

Data storage is one of the central points of handling big data. You learned about various kinds of data stores, including structured and unstructured data, NoSQL, and data warehousing, with the appropriate technology choices associated with each. You learned about cloud data storage from popular public cloud providers.

Once you collect and store data, you need to transform it to get insights into that data and visualize your business requirements. You learned about data processing architecture and technology choices to choose open source and cloud-based data processing tools per your data requirements. These tools help you get data insights and visualizations per the nature of your data and organizational needs.

You learned about various big data architecture patterns, including data lake, lakehouse, data mesh, streaming data architecture, reference architecture, and how to choose the right architecture for your data needs. Finally, you learned big data architecture best practices by combining all your learning in the reference architecture.

As you collect more data, it's always beneficial to get future insights, which can be exceptionally beneficial for business. You often need ML to predict future outcomes based on historical data. In the next chapter, let's learn more about ML and how to make your data architecture future-proof.

Join our book's Discord space

Join the book's Discord workspace to ask questions and interact with the authors and other solution architecture professionals: <https://packt.link/SAHandbook>



13

Machine Learning Architecture

In the previous chapter, you learned about ingesting and processing big data and getting insights to understand your business. In the traditional way of running a business, the organization's decision maker looks at past data and uses their experience to plot the future course of the company's direction. It's not just about setting up the business vision but also improving the end user experience by predicting and fulfilling their needs or automating day-to-day decision-making activities such as loan approval.

However, with the sheer amount of data available now, it's become difficult for the human brain to process all data and predict the future. That's where **artificial intelligence (AI)** and **machine learning (ML)** come in. AI is the broader concept of machines carrying out tasks in smart ways like Siri and Alexa to understand your questions and give answers, and ML is a specific subset of AI that involves teaching computers to learn and make decisions based on data. They help us to predict future courses of action by looking at a large amount of historical data. Most enterprises are investing in ML today, primarily because of the acceleration brought by generative AI (GenAI). ML is fast becoming the technology that helps companies differentiate themselves—through the creation of new products, services, and business models, allowing them to innovate and gain a competitive advantage.

AI and ML are excellent for solving business problems because they present countless use cases in different lines of business across a company, and the high degree of impact these use cases can make. For example, with ML you can build a new level of customer service with call center intelligence or help marketing teams deliver on their personalization objectives by using an ML-based personalized marketing campaign.

Within the scope of this chapter, we'll cover the following topics to handle and manage your ML needs:

- What is machine learning?
- Working with data science and machine learning
- Machine learning in the cloud
- Building machine learning architecture
- Design principles for machine learning architecture
- MLOps

- Deep learning
- Natural language processing (NLP)

By the end of this chapter, you will have an understanding of ML architecture. You will learn about the various ML models and the ML workflow. You will understand the process of creating an ML model pipeline through feature engineering, model training, inference, and model evaluation.

What is machine learning?

Machine learning drives better customer experiences, more efficient business operations, and faster, more accurate decision making. With the rise in compute power and the proliferation of data, ML has moved from the periphery to become a core differentiator for businesses and organizations across industries. ML use cases can apply to most businesses, like personalized product and content recommendations, contact center intelligence, virtual identity verification, and intelligent document processing. There are also customized use cases built for specific industries—like clinical trials in pharma or assembly line quality control in manufacturing.

ML uses technology in order to find new trends and inculcate mathematical predictive models based on past factual data. ML can help to solve complex problems such as the following:

- You may need to learn how to create complex code rules to make a decision; for example, if you want to recognize people's emotions in images and speech, there are no easy ways to code the logic to achieve that.
- When you need to analyze a large amount of data for decision making and the volume of data is too large for a human to do this efficiently. For example, while a human can do it with spam detection, the amount of data makes it impractical to do this quickly.
- Relevant information may only become available dynamically when you need to adapt and personalize user behavior based on individual data; for example, with individualized product recommendations or website personalization.
- When many tasks with a lot of data are available, you cannot track the information fast enough to make a rule-based decision—for example, fraud detection and NLP.

Humans handle data prediction based on the results of their analyses and their experiences. Using ML, you can train a computer to provide expertise based on available data and get a prediction based on new data. Here are some prevalent use cases for ML across various industries:

- **Predictive maintenance:** Predict whether a component will fail in advance based on sensor data. This is commonly applied in estimating the **remaining useful life (RUL)** of automotive fleets, manufacturing equipment, and IoT sensors. Its primary benefits are increased vehicle and equipment uptime and significant cost savings. This application is widespread in the automotive and manufacturing industries.
- **Demand forecasting:** Use historical data to project key demand metrics quicker and more accurately, aiding in making more accurate business decisions about production, pricing, inventory management, and purchasing/restocking. The main advantages include meeting customer demand, minimizing inventory carrying costs by reducing surplus inventory, and decreasing waste. Industries like financial services, manufacturing, retail, and, **consumer packaged goods (CPG)** frequently employ this use case.

- **Fraud detection:** Automate the identification of potentially fraudulent activities and flag them for further review. The primary benefit of this is the reduction of costs related to fraud and the maintenance of customer trust. This use case is implemented in the financial services and online retail sectors.
- **Credit risk prediction:** Explain individual predictions from credit applications to predict the likelihood of whether credit will be paid back or not (often called a *credit default*). The benefit lies in identifying bias and complying with regulatory requirements. This use case is predominantly used in the financial services and online retail industries.
- **Data extraction and analysis from documents:** Understand text in handwritten and digital documents, extracting information for classification, and decision-making purposes. This use case is widespread in sectors like healthcare, financial services, legal, mechanical, electrical, and education industries.
- **Personalized recommendations:** Make customized recommendations based on historical data. This approach is common in the retail and education sectors.
- **Churn prediction:** Estimate the probability of customers discontinuing their services. This is often utilized in industries such as retail, education, and software as a service (SaaS) providers.

The main idea behind ML is to make available a training dataset to an ML algorithm and have it predict something from a new dataset, for example, feeding some historical stock market trend data to an ML model and having it predict how the market will fluctuate in the next six months to one year.

When developing ML systems, it's important to carefully combine data and code. Both must come together in an organized manner and should evolve in a controlled way to build toward the common goal of a robust and scalable ML system.

The data you use for training, testing, and making decisions with the ML system inference will change over time as data comes from different places. Your code also needs to change with data to accommodate data from various sources. Without a systematic approach, there can be divergence in how code and data change. This mismatch can cause problems when you try to use your ML system for real tasks. It can also get in the way of smooth deployment, and lead to results that are hard to understand, trace, or repeat later. There are various types of ML; let's explore them.

Types of machine learning

ML helps computers learn things without us needing to program every detail. It is like teaching computers to learn from experience. Imagine teaching your dog a trick: you show it what to do, and then it learns and does it! With ML, computers can learn from data and then use this learning to make decisions. Let's look at the different ways computers can learn.

Supervised learning

In supervised learning, the algorithm is given a set of training examples where the data and target decisions are known. It can then predict the target value for new datasets containing the same attributes. With this type of learning, the algorithm is taught using a dataset where input data comes with the correct answer or target. The algorithm learns to connect inputs with their correct outputs using these examples.

This type of learning is often used for tasks where you need to classify things into categories or predict numbers, like in classification and regression tasks. For example, it can be used to classify emails as spam or not spam or to predict the price of a house based on its features.

Unsupervised learning

In unsupervised learning, the algorithm is provided with a massive volume of data and should discover patterns and relationships among the data. It can then draw inferences from datasets.

Human intervention is not required in unsupervised learning, for example, auto-classification of documents based on context. It addresses the problem where correct output is unavailable for training examples, and the algorithm must find patterns in data using clustering.

In unsupervised learning, a model is trained using an unlabeled dataset. The algorithm works on its own to discover patterns, structures, or relationships in the data without any specific guidance or labeled examples to follow. This type of learning is often applied to clustering, dimensionality reduction, and density estimation tasks. News agencies or legal firms often deal with massive data. Using unsupervised learning, they can automate document categorization, efficiently manage their digital repositories, and improve information retrieval processes, such as recommending similar articles or cases to readers or researchers.

Semi-supervised learning

This approach mixes elements of both supervised and unsupervised learning. It involves using a small amount of labeled data along with a more significant amount of unlabeled data to improve model performance. Semi-supervised learning is particularly useful when obtaining labeled data is either costly or takes a lot of time. It's often used in scenarios where there's a limited amount of labeled data, but plenty of unlabeled data. Within the biomedical field, for example, semi-supervised learning can be very advantageous. For instance, annotating medical images requires a lot of time and resources, making semi-supervised learning a practical solution. Models can be initially trained on a small set of labeled images and then fine-tuned using a more extensive set of unlabeled images, maximizing utility while minimizing costs and resources.

Reinforcement learning

This type of learning involves training agents (or computer programs) to make a series of sequential decisions in a certain setting. The goal is for the agent to learn the best actions to take in order to maximize a cumulative reward over time. Agents learn by taking action, receiving feedback (rewards or punishments), and adjusting their strategies. Reinforcement learning is used in autonomous robotics, game playing (e.g., AlphaGo), and recommendation systems. Autonomous vehicles employ reinforcement learning by navigating through traffic and adjusting actions based on the environment, thus ensuring optimal decision making in diverse scenarios. The vehicle makes sequences of decisions (like changing lanes, adjusting speed, etc.), receiving positive reinforcement for safe, efficient actions and negative reinforcement for undesirable actions.

Self-supervised learning

This is a type of unsupervised learning where the algorithm generates labels or targets from the data. It often involves tasks like predicting missing parts of data. Self-supervised learning has gained popularity in NLP and computer vision for pre-training models on large datasets before fine-tuning specific tasks. In image processing or computer vision, self-supervised learning can be employed to predict the next frame in a video sequence, thus facilitating models that understand movement and development within visual data. Pre-training models in this manner and then fine-tuning them for specific tasks, like object detection, can yield impressive results.

Multi-instance learning

In multi-instance learning, each data point is a bag containing multiple instances (sub-data points). The goal is to learn from bags of data while only having access to bag-level labels. Multi-instance learning has applications in drug discovery, image classification, and content-based image retrieval. Considering e-commerce platforms, multi-instance learning could be employed to predict whether a user will purchase a product within a session (a bag), using various instances like page views, clicked products, and time spent on pages. The bag-level label might indicate whether a purchase was made during that session, providing a robust basis for predictions and personalized content delivery.

These diverse learning paradigms, each with its own specialty and application area, make ML a versatile field, adaptable to various scenarios and challenges across industries and domains. By choosing a paradigm tailored to the specificities and available data of a given problem, ML practitioners can derive insightful models and facilitate intelligent, automated decision-making across applications. The key is to select a learning type that aligns best with the available data and the problem at hand, ensuring that models are both robust and applicable.

In the next section, let's learn how data science goes hand in hand with ML.

Working with data science and machine learning

ML is all about working with data. The quality of the training data is crucial to the success of an ML model. High-quality data leads to a more accurate ML model and the right prediction.

Data often has multiple issues, such as missing values, noise, bias, outliers, and so on. Exploring the data makes us aware of this, providing us with necessary information on data quality and cleanliness, interesting patterns in the data, and likely paths forward once you start modeling. Data science includes data collection, data preparation, analysis, preprocessing, and feature engineering.

Data preparation is the first step in building an ML model. It is time consuming and constitutes up to 80% of the time spent on ML development. Data preparation has always been considered tedious and resource intensive due to the inherent nature of data being “dirty” and not ready for ML in its raw form. “Dirty” data could include missing or erroneous values and outliers. Feature engineering is often needed to transform the inputs to deliver more accurate and efficient ML models.

The first step in data preparation is to understand the business problem. Data scientists are often eager to jump into the data directly, start coding, and produce insights. However, without a clear understanding of the business problem, any insights you develop have a high chance of becoming a solution that cannot address the problem at hand. It makes much more sense to start with a straightforward user story and business objectives before getting lost in the data.



After building a solid understanding of the business problem, you can narrow down the ML problem categories and determine whether ML will be suitable to solve your business problem.

Data preparation often involves multiple steps such as data cleaning, dealing with missing values, data normalization/standardization, and data labeling. You will learn about these steps in detail in the upcoming *Building machine learning architecture* section later in this chapter. Most standalone data preparation tools are equipped with functionalities for data transformation, feature engineering, and visualization. Data transformation might include tasks like converting currencies (for example, from dollars to euros) or changing measurement units (such as from kilograms to pounds). Feature engineering involves creating new data columns (features) from existing ones to enhance the dataset's utility for ML models; for instance, extracting the day of the week or the month from a date column can help the model discern time-related patterns. While these tools excel in preparing data, they often lack built-in capabilities for model validation, a critical step in assessing an ML model's performance. What's needed is a framework that provides all these capabilities in one place and is tightly integrated with the rest of the ML pipeline. Data preparation modules therefore need curation and integration before they are deployed in production.

As shown in the following diagram, data preprocessing and learning to create an ML model are interconnected—your data preparation will heavily influence your model, while the model you choose heavily influences the type of data preparation you will do. Finding the correct balance is highly iterative and is very much an art (or trial and error):

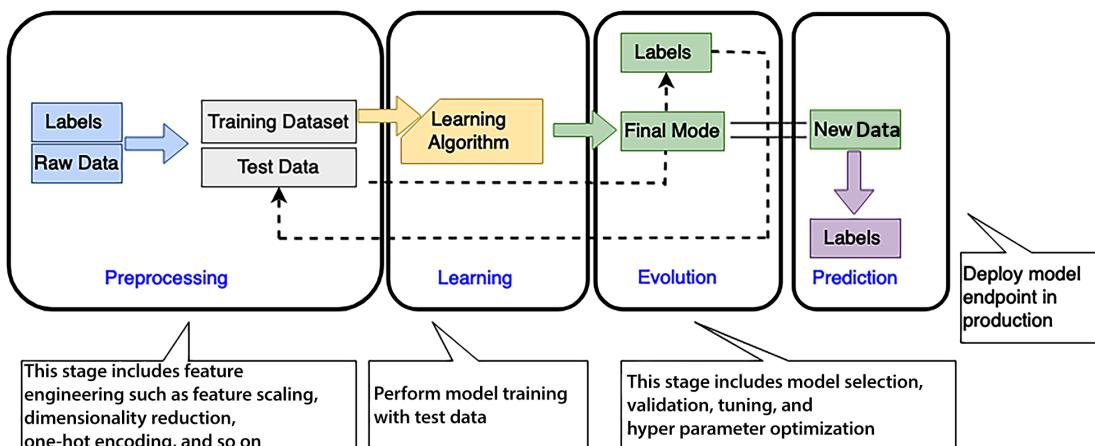


Figure 13.1: ML workflow

As shown in the preceding diagram, the ML workflow includes the following phases:

- **Preprocessing:** In this phase, the data scientist preprocesses the data and divides it into training (70% of the data), validation (10% of the data), and testing (20% of the data) datasets. Your ML model is trained using the training dataset, which helps it learn and give the right prediction. Once training is complete, the model is then evaluated with a separate validation dataset to assess its performance and generalization capabilities. Once the model is ready, you can test it using a testing dataset. Features are independent attributes of your dataset that may or may not influence the outcome. Feature engineering involves finding the right feature, which can help to achieve model accuracy. The label is your target outcome, which is dependent on feature selection. You can apply dimensionality reduction to choose the right feature, which filters and extracts the most compelling feature for your data.
- **Learning:** You select the appropriate ML algorithm per the business use case and data in this phase. This is the core of the ML workflow, where you train your ML model on your training dataset. To achieve model accuracy, you need to experiment with various hyperparameters and perform model selection. Hyperparameters are the configuration settings used to control the learning process in ML algorithms.
- **Evaluation:** Once your ML model has been trained in the learning phase, you want to evaluate its accuracy with a known dataset. To assess your model, you use the validation dataset kept aside during the preprocessing step. Required model tuning needs to be performed per the evaluation result if your model prediction accuracy needs to be revised to the exceptions determined by validation data.
- **Prediction:** Prediction is also known as inference. In this phase, you deploy your model and start making a prediction. These predictions can be made in real time or in batches.

GenAI has led a paradigm shift in the landscape of ML and AI. At its core are **foundational models (FMs)** like GPT-4, which have been trained on vast, internet-scale datasets, redefining the conventional norms of data labeling and model customization. This groundbreaking technology empowers organizations to fine-tune FMs with limited data tokens, thereby significantly reducing the manual effort and time traditionally associated with data preparation.

However, it is vital to recognize that GenAI is not a silver bullet because it is not designed to address all AI and ML problems. Also, the development of FMs is a resource-intensive endeavor, demanding substantial computational power and access to extensive datasets. As such, many enterprises opt to leverage FMs provided by renowned third-party companies, such as OpenAI, Google, Meta, and Anthropic, who have pioneered the development of these models.

Nonetheless, the story does not end there. Custom model training remains a compelling option, especially when specific, tailored solutions are required. While GenAI provides an innovative approach to problem solving, the strategic decision to adopt it should align with an organization's unique goals, resources, and constraints. You will learn more about GenAI in *Chapter 14, Generative AI Architecture*.

As per your data input, the ML model often has overfitting or underfitting issues, which you must consider to get the right outcome. Let's learn more about this.

Evaluating ML models—overfitting versus underfitting

In overfitting, your model needs to generalize, which means it should perform well not just on the data it was trained on (the training set) but also on new, unseen data (the test set or validation set). If a model is overfitting, it has essentially memorized the training data, capturing noise along with the underlying pattern, which leads to poor performance on any new data. If a model shows high performance metrics on the training data but significantly lower metrics on the test data, it's a sign of overfitting.

This typically indicates that the model is too flexible for the amount of training data, allowing it to *memorize* the data, including noise. Overfitting corresponds to high variance, where small changes in the training data result in significant changes to the results.

In underfitting, your model fails to capture essential patterns in the training dataset. Typically, underfitting indicates that the model is too simple or has too few explanatory variables. An underfitting model needs to be more flexible to model real patterns and corresponds to high bias, indicating that the results show a systematic lack of fit in a certain region.

The following graphs illustrate the clear difference between overfitting and underfitting, as they correspond to a model with a good fit:

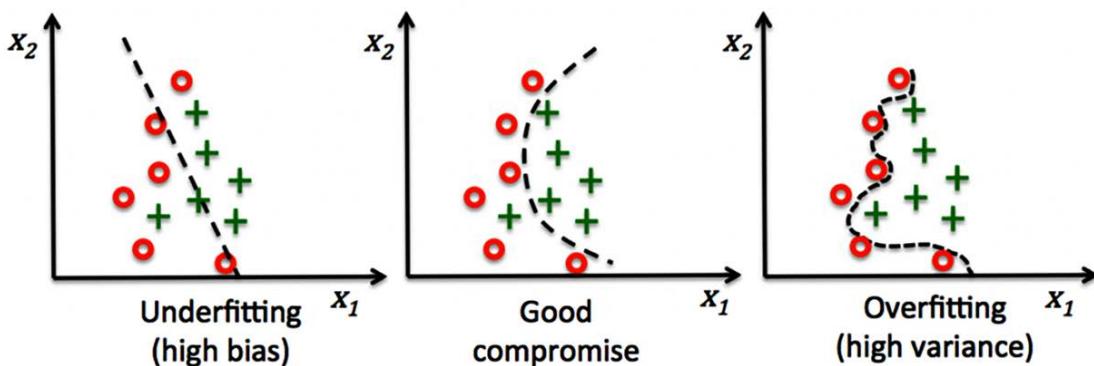


Figure 13.2: ML model overfitting versus underfitting

The ML model categorizes two data point categories, illustrated by the preceding graphs' rings and crosses. The ML model tries to determine whether a customer will buy a given product or not. The chart shows predictions from three different ML models. You can see an overfitted model (on the right) traversing through all ringed data points in training and failing to generalize the algorithm for real-world data outside of the training dataset. On the other hand, the underfitted model (on the left) leaves out several data points and needs to be more accurate. A good model (shown in the middle) usually provides clear data point predictions. Creating a good ML model is like creating art; you can find the right fit with model tuning.

Popular machine learning algorithms

The popularity of an algorithm often depends on its applicability and performance in diverse use cases, ease of understanding, and implementation, as well as its ability to scale and adapt to different types of data. Let's look at some popular ML algorithms.

Linear regression

Linear regression tries to understand how one thing (let's say, X) can help predict another thing (Y) by finding a linear relationship between them. Imagine you're at a farmer's market. When you observe the prices of pumpkins, you notice that as the size of the pumpkin increases, so does its price. Linear regression acts like drawing a straight line through all the price points of pumpkins, ensuring the line is as close to all points as possible.

Real estate is a sector where linear regression plays a vital role. For instance, if a company wants to predict the selling price of a house, it will look at features like the number of rooms, location, and age of the property. If houses with more rooms have typically been sold for higher prices in the past, the model will predict a higher price for houses with more rooms in the future. It's like predicting the price of our pumpkin based on its size.

Logistic regression

Logistic regression tells you the probability or chance of something happening, with "yes" or "no" answers. Imagine you are trying to predict whether a book will be a bestseller or not. Logistic regression will look at features like the number of pages, the popularity of the author, and the genre to predict the likelihood (between 0 and 1) of it becoming a bestseller.

In healthcare, logistic regression can predict the likelihood of a patient having a disease based on various symptoms and test results. For example, by considering factors like age, blood pressure, and cholesterol levels, it can predict the probability of a person having heart disease. Doctors might conduct further tests if the probability is high, ensuring early and proactive management of potential health risks.

Decision trees

Decision trees help you make a series of decisions by asking questions. Imagine you want to decide what to wear. The decision tree might ask: "Is it raining?" If "yes," it might suggest a raincoat. If "no," it might ask another question, like "Is it hot?", and suggest clothes accordingly, helping you navigate through various options until you find the best answer.

Decision trees can help predict whether a customer will buy a product in the retail sector. For instance, it might ask: "Has the customer bought something in the last month?" If "yes," they might likely buy again soon. If "no," it might consider other factors like recent website visits or clicked promotions to predict their purchasing behavior, helping retailers target customers with relevant ads and offers.

Random forests

As its name implies, a random forest is like creating a forest where each tree is a decision tree that casts a vote to decide on the outcome. Each tree is given a random subset of the data and makes its best decision. Then, all trees “vote” to provide a final answer. This approach often results in better, more stable predictions than a single decision tree.

In finance, random forests can be used for predicting whether to approve or deny loan applications. For example, trees might consider different aspects like credit score, income, and debt to make individual decisions. The final decision, made through a majority vote of all trees, is more accurate and robust than relying on a single model, thereby reducing the risk of lousy loan approval.

K-Nearest Neighbours (k-NNs)

Using k-NNs is like looking at a new thing and trying to understand it by comparing it to similar things we already know. If you find a fruit you’ve never seen before, you might decide whether it’s likely to taste sweet or sour by comparing it to similar-looking fruits whose taste you’re familiar with. If it looks like other sweet fruits, you might guess it’s sweet.

K-NN is widely utilized in recommendation systems, like those in e-commerce websites. If a user has bought a particular product, k-NN finds similar products that other similar users have bought and recommends them to the user. For example, suppose a user buys a detective novel. In that case, k-NN looks for other users who bought the same novel and then recommends other books that those users have bought, enhancing the user’s shopping experience by showing relevant products.

Support vector machines (SVMs)

SVMs are decisive algorithms that aim to keep things clear and separate. Imagine you have a big table and put apples on one side and bananas on the other. SVMs try to find the broadest possible line (or gap) to separate these two fruits so that all apples are on one side and all bananas are on the other, causing no mix-ups.

In the field of handwriting recognition, SVMs are helpful. For instance, if you write a number, say “4,” SVMs help the computer decide whether it’s indeed a “4” or maybe a “9” by looking at many examples of how people write these numbers and finding the best boundary that separates “4” from “9,” thus helping in accurately reading handwritten numbers.

Neural networks

Think of neural networks like a mini-brain inside the computer that learns from lots of examples to make decisions. When you learn to ride a bike, you might initially fall, but gradually, you learn how to balance and pedal by understanding what went wrong in the previous attempts. Neural networks learn similarly, adjusting from errors to make better decisions next time.

For example, social media platforms use neural networks to identify and tag people in photos in image recognition. The network learns by looking at many pictures of a person and noticing features like nose shape and eye color. When a new photo is uploaded, it compares these features with its learned knowledge, making the best guess on who’s in it.

K-means clustering

K-means clustering is a means of grouping similar data points together. It's like organizing a big party where you want to create groups (or clusters) of friends who share similar interests so that they enjoy each other's company. You repeatedly try different ways to group individuals, trying to ensure that everyone in a group is as similar to each other as possible, ensuring a fun time for all.

A popular application of k-means is customer segmentation for marketing strategies. Businesses can use k-means to group customers into clusters based on their purchasing behavior. For instance, one cluster might be customers who buy frequently but spend little each time, while another might be customers who buy infrequently but make big purchases. Each group can be targeted with different marketing strategies to maximize sales.

XGBoost

XGBoost learns from past mistakes and becomes wiser and wiser with each decision. If you were solving math problems and you solved one incorrectly, XGBoost would look at it, understand where you went wrong, and remember this mistake so that the next time it faces a similar problem, it doesn't repeat the same error.

In the credit industry, XGBoost is widely used to predict whether a customer will default on a loan. Looking at many factors like income, age, and previous loan history predicts the probability of a customer defaulting. If an applicant is predicted to have a high risk of default, the loan might be denied, thereby minimizing the risk for the bank.

These algorithms are the foundation of many ML projects, chosen based on the specific problem and data types (e.g., text, images, numerical data) with which one works. Some of them, like neural networks, require more computational resources and data, while others, like decision trees or k-NN, might be applicable even with smaller datasets.

We continue our exploration of ML by next looking at popular ML tools and frameworks.

Popular machine learning tools and frameworks

ML is accomplished using various tools and frameworks, each designed to aid different aspects of developing ML models – from data processing and algorithm design to model training and deployment. Here are some of the popular ones.

Popular tools and frameworks for data preparation and exploration include:

- **NumPy:** The core Python library for scientific computing. Numerical Python, or NumPy, is a library of multi-dimensional array objects and a set of operations for manipulating such arrays.



An array is a collection of data items of the same type, stored in contiguous memory locations.

NumPy facilitates easier and more efficient numerical and logical operations on large datasets. Imagine a retail company that wants to calculate the monthly average sales to analyze the performance and decide the future strategy. They have daily sales data stored in a numerical format. Using NumPy, they can easily calculate monthly average sales by organizing the daily sales data into an array, summing it, and dividing it by the number of days.

- **Pandas:** A library offering Python users simple, high-performance data structures and data analysis capabilities, allowing users to analyze and manipulate data. It presents Series and DataFrames, two essential data structures for Python that are constructed on top of NumPy.



A Series is a column, and a DataFrame is a multi-dimensional table comprising a series collection.

Pandas functionalities make it easy to clean, analyze, and visualize data. For example, imagine a grocery store wanting to analyze its sales data to understand which products are the best selling and which ones are not doing well. They have a large dataset with information about every transaction, including the product name, quantity sold, and price. Using pandas, they can easily manipulate this data, finding the total sales for each product, sorting them, and identifying the top-selling items.

- **Scikit-learn:** A straightforward and effective predictive data analysis tool that works with pandas and NumPy. Numerous supervised and unsupervised learning techniques are supported by scikit-learn. It is extensively utilized in ML, data mining, and data analysis. Scikit-learn has many built-in tools for model selection, evaluation, data import, and improvement. Imagine a bank wants to predict whether a customer will default on their loan. They have historical data on previous customers, including age, salary, marital status, and whether they defaulted. Using scikit-learn, they can build a model (like a decision tree, logistic regression, or another appropriate algorithm) that learns from this data, and then use this model to predict the likelihood of new customers defaulting on their loans.

Popular tools and frameworks for data visualization include:

- **Matplotlib:** A popular and feature-rich Python library for making static, interactive, and animated visualizations. In addition to line, scatter, error bar, histogram, bar, pie, box, and 3D plots, Matplotlib offers an incredibly versatile foundation for creating a vast array of visualizations. This tool allows developers and data scientists to visualize their data in various forms of plots, which can be very useful for understanding the data distribution and patterns, debugging issues, or visualizing the relationships among the data. Let's say a teacher wants to visually present the scores of students in a class to highlight overall performance and outliers quickly. Using Matplotlib, the teacher can create a variety of charts, like bar charts, scatter plots, or histograms, to represent the distribution of scores in an easily interpretable visual format.

- **Seaborn:** A Matplotlib-based statistical data visualization library that offers a high-level interface for designing appealing graphs. Seaborn has several built-in themes and color palettes to make creating aesthetically pleasing and visually informative charts easy. It is particularly well suited for visualizing complex datasets with multiple variables, thanks to its support for creating multi-plot layouts and functionalities to visualize the relationship between multiple variables. Imagine a retail business that wants to understand its customer purchasing behavior across different product categories over a period of time. With Seaborn, analysts can create a heatmap to visually represent purchasing frequency across various product categories in different months, allowing quick insights into trends and customer preferences.
- **Business intelligence (BI) tools:** BI tools such as Tableau, Microsoft Power BI, Amazon QuickSight, and MicroStrategy are used for converting raw data into an understandable format. These tools help people visualize, understand, and make decisions with their data. Unlike other mentioned tools, these tools come with a graphical user interface that allows users to drag and drop items to analyze data, making it particularly accessible to individuals without a coding background. BI tools can connect to numerous data sources, providing real-time data insights. You can create and share dashboards, which provide interactive visualizations with embedded analytics. Consider a restaurant chain that wants to optimize its supply chain and menu based on customer purchasing behavior and seasonal trends. Using a BI tool, the company can visualize sales data across various dimensions, such as time, customer demographics, and product categories, to identify patterns and inform decision-making processes.

Popular tools and frameworks for model development and training include:

- **TensorFlow:** A comprehensive open-source platform designed to manage a range of ML tasks. TensorFlow supports a range of APIs for building, training, and deploying AI models. A key feature of TensorFlow is its ability to create dataflow graphs. These graphs show how data moves through a series of processing steps or nodes. In these graphs, each node stands for a mathematical operation, and the connections between nodes, known as edges, represent tensors, which are multi-dimensional data arrays. TensorFlow provides tools for developers to use large-scale ML and nurtures an extensive library that makes it convenient to learn and develop AI models, ranging from beginners to experts. Imagine a healthcare start-up wanting to leverage ML to predict the onset of diseases based on various patient metrics like age, genetics, weight, and lifestyle habits. They could utilize TensorFlow to build a neural network model that considers all these factors to predict the likelihood of disease occurrence.
- **PyTorch:** A popular ML library due to its flexibility, ease of use, and dynamic computational graph, which is particularly useful for deep learning. Developers, researchers, and data scientists favor it for both research and production due to its flexibility and extensive functionality. The dynamic computation graph enables users to change network behavior on the go, and the library provides a rich API for application in various ML tasks like classification, regression, and reinforcement learning, to name a few. Imagine an e-commerce company that wants to develop a chatbot to enhance customer experience. Using PyTorch, they could develop a deep learning model that understands the customer's language and provides useful and accurate responses to customer queries in real time.

- **Keras:** An open-source software library that serves as an easy-to-use API for building and training deep learning models. It can run atop other popular ML libraries like TensorFlow, making it highly versatile. Keras is particularly favored for its simplicity and ease of use in experimentation. With Keras, data scientists and developers can turn their ideas into results with minimal delay, which is vital in innovative projects. Let's consider a retail company trying to recommend products to customers based on their past purchase history. The company could use Keras to create a recommendation system that analyzes customer buying patterns and suggests products they are likely to buy.
- **Apache Spark's MLLib:** An ML library that's part of Apache Spark, designed to scale up to meet the demands of big data. MLLib provides various ML algorithms including classification, regression, clustering, and collaborative filtering, as well as tools for model selection and evaluation. It also provides APIs for saving models for later use. MLLib is designed to handle large-scale ML tasks efficiently. Given its distributed computing capability, MLLib can quickly handle vast datasets, making it particularly valuable for scenarios where large-scale data analysis and model training are essential. Moreover, MLLib can be utilized with different data sources and formats, offering flexibility in dealing with various data types. Imagine a financial institution that wants to identify fraudulent credit card transactions as they occur. Using MLLib, data scientists can utilize vast amounts of transaction data to train models that identify unusual purchase patterns or anomalous transactions indicative of fraud, allowing for real-time detection and mitigation of fraudulent activities.

Popular tools and frameworks for model deployment include:

- **Docker:** A platform designed to make creating, deploying, and running applications using containers easier. Docker is not an ML tool per se, but it plays a crucial role in deploying ML models and applications efficiently and consistently. Docker allows developers and data scientists to package an application along with all of its dependencies (libraries, tools, and scripts) into a "container." This container can be transferred and run consistently across various computing environments, which means the application will work the same way regardless of where it's being run. Imagine a software development team that is creating an ML application to predict stock prices. They have data scientists who use various tools and libraries for model creation and software engineers who build the application using different technologies. Using Docker, they can create a cohesive workflow where everyone can work in a consistent environment, ensuring that the model and application behave the same way during development, testing, and deployment despite being developed with different tools.
- **Flask:** A micro web framework written in Python. It's simple to learn and simple to utilize, making it great for beginners, but it doesn't include additional features (like form validation or database abstraction layers) that a full-stack framework might offer. However, its simplicity and ease of use make it popular for deploying lightweight web applications and APIs, especially in the data science and ML community. Imagine a scenario where a data scientist has developed an ML model to predict whether an email message is spam or not. This model could be utilized by a web application where users submit their emails, and in return, the application tells them whether the email is spam. Using Flask, the data scientist can create a simple web server that accepts email text, uses the ML model to predict if it's spam, and then returns the result to the user, all through a web interface.

Popular integrated development environments (IDEs) include:

- **Jupyter Notebook:** An open-source web application that enables the creation and sharing of interactive documents. These documents can contain live code, equations, visualizations, and explanatory text, making them versatile tools for data analysis, scientific research, and educational purposes. It supports various languages, like Python, R, and Julia, and is extensively used in data cleaning, statistical modeling, ML, and much more due to its interactive computational environment. Jupyter is crucial in data science, academic research, and scientific computing for enabling users to create reproducible analyses and compellingly communicate their results through visualization and narrative text. Let's consider a scenario where a biologist wants to analyze data on bird species and their migrations. The biologist could use a Jupyter notebook to write Python code that loads the data, visualizes migration patterns, and perhaps even uses ML to predict future migration timings or paths based on historical data.
- **RStudio:** An open-source IDE for R, a statistical computing and graphics programming language, that works with the standard version of R and can also work with the version of R available in the cloud. RStudio provides a robust set of capabilities for script development, data visualization, and statistical analysis, supporting the comprehensive utilization of the R language. Imagine a retail company wanting to understand the purchasing behaviors of its customers. Using RStudio, a data analyst could input sales data, apply statistical analysis, and create visualizations (like scatter plots, histograms, or bar charts) to identify buying trends, popular items, and peak shopping periods, potentially employing ML to forecast future sales trends.
- **Apache Zeppelin:** An open-source notebook-based environment similar to Jupyter, which allows data engineers, data analysts, and data scientists to develop, organize, execute, and share data workflows and collaboratively execute code. Zeppelin supports various data processing backends like Apache Spark, Python, and JDBC. Users can create data-driven, interactive, and collaborative documents with Scala, Python, SQL, and so on. A particular strength of Zeppelin lies in its built-in data visualization and some integrations that aren't as out-of-the-box for Jupyter users. Consider a scenario in the healthcare sector where analysts want to explore patient data to understand patterns in disease outbreaks. Using Zeppelin, they can interactively explore datasets, integrate various data processing backends, and create visualizations like heatmaps or line charts to represent outbreaks over geographic regions or timelines visually.

Zeppelin, RStudio, and Jupyter notebooks are the most common environments for data engineers doing data discovery, cleansing, enrichment, labeling, and preparation for ML model training.

As the cloud is becoming a go-to platform for ML model training, let's learn about some available ML cloud platforms.

Machine learning in the cloud

ML development is a complex and costly process. There are barriers to adoption at each step of the ML workflow, from collecting and preparing data, which is time consuming and undifferentiated, to choosing the right ML algorithm, which is often done by trial and error, and lengthy training times, which leads to higher costs. Then there is model tuning, which can be a very long cycle and requires adjusting thousands of different combinations. Once you've deployed a model, you must monitor it and then scale and manage its production.

To solve these challenges, all major public cloud vendors provide an ML platform that facilitates ease of training, tuning, and deploying ML models anywhere at a low cost. For example, Amazon SageMaker is one of the most popular platforms that provides end-to-end ML services. SageMaker provides users with an integrated workbench of tools brought together in one place through SageMaker Studio. Users can launch Jupyter Notebook and JupyterLab environments instantly through SageMaker Studio. SageMaker also provides complete experiment management, data preparation, and pipeline automation and orchestration to help make data scientists more productive. SageMaker also provides the fully managed RStudio platform, which is one of the most popular IDEs among R developers for ML and data science projects. SageMaker provides fully managed servers in the cloud. Beyond notebooks, SageMaker provides other managed infrastructure capabilities as well. From distributed training jobs, data processing jobs, and even model hosting, SageMaker takes care of all of the scaling, patching, high availability, and so on associated with building, training, and hosting models.

Similarly, GCP provides the Google Cloud AI platform, with different services to perform ML experiments, and Microsoft Azure offers Azure ML Studio.

In addition to managed ML platforms, cloud vendors also provide ready-to-use AI services. AI services allow developers to easily add intelligence to any application without needing ML skills. The pre-trained models provide ready-made intelligence for your applications and workflows to help you personalize the customer experience, forecast business metrics, translate conversations, extract meaning from documents, and more. For example, AWS provides the Amazon Comprehend AI service, which has pre-trained models that support keyphrase detection and sentiment analysis natively in multiple languages.

The cloud is increasingly serving as a primary platform for accessing and utilizing GenAI FMs, offering a cost-effective and scalable environment for testing and deploying these advanced AI systems. These FMs are trained on vast datasets and can be fine-tuned for specific tasks, making them versatile tools for a wide range of applications. The cloud's scalability and resource availability make it an ideal environment for working with these large, resource-intensive models. Both open-source and commercial GenAI FMs are available, offering options for different needs and budgets.

This accessibility of GenAI FMs via the cloud democratizes advanced AI capabilities, enabling businesses and developers to leverage cutting-edge AI technology without the need for significant upfront investment in computational infrastructure. For example, using an API, Amazon Bedrock allows you to access multiple third-party foundation models from companies like stability.ai, Meta, Mistral, Anthropic, Amazon, and AI21. Similarly, Azure provides API access to OpenAI's GPT-4, and GCP provides access to their FM model Gemini. You will learn more about them in Chapter 14, *Generative AI Architecture*.

Data scientists can leverage managed cloud environments to accelerate data preparation and model training. When complete, they can one-click deploy the model and begin serving inferences over HTTP.

Let's learn more about some of the essential things to consider when you are designing ML architecture.

Building machine learning architecture

Building a robust and scalable workflow from a loose collection of code is a complex and time-consuming process, and many data scientists need to gain experience building workflows. An ML workflow can be defined as an orchestrated sequence that involves multiple steps. Data scientists and ML developers first need to package numerous code recipes and then specify the order in which they should execute, keeping track of code, data, and model parameter dependencies between each step.

Added complexity in ML workflows warrants monitoring changes in data used for training and predictions because changes in the data could introduce bias, leading to inaccurate predictions. In addition to monitoring the data, data scientists and ML developers also need to monitor model predictions to ensure they are accurate and don't become skewed toward particular results over time. As a result, it can take several months of custom coding to get the individual code recipes to execute in the correct sequence and as expected.

ML architectures need to protect model artifacts and provide self-service capabilities for model development and training. It's essential for your ML architecture to facilitate automated, comprehensive documentation of the entire model development life cycle, encompassing stages from across development, training, and deployment.

ML applications should also employ a continuous integration and continuous deployment (CI/CD) pipeline that is seamlessly integrated with change control systems. This integration is crucial for model management and deployment. Additionally, the environments require pre-defined security configurations.

The following are the ML architecture components from the AWS ML platform to help you understand ML architecture.



Other ML platforms are Azure ML Studio, H2O.ai, SAS, Databricks, and the Google AI platform.

Prepare and label

To prepare data for ML, you need to run your data processing workloads, such as feature engineering, data validation, model evaluation, and model interpretation. Feature engineering also preprocesses datasets to convert the input datasets into a format expected by your ML algorithm. For example, if you're working with a dataset that includes dates, you might extract the day of the week, the month, and the time of the year as separate features, as these could have predictive value for your model. You can use the various tools and techniques mentioned in the previous section to wrangle data as per your ML needs. A managed ML platform like Amazon SageMaker also provides a data wrangler and feature store capability to simplify the data processing job. SageMaker Data Wrangler allows you to easily prepare your data for ML by providing a visual interface to access, combine, clean, and transform data. This tool helps you perform common data preparation tasks without writing code, speeding up the process and reducing the chance of errors.

Additionally, SageMaker Feature Store is a centralized repository to store, share, and manage curated features for ML models. This ensures consistency across different models and reduces redundancy in feature engineering efforts. Feature Store helps in maintaining a consistent set of features for training and inference, facilitating better model performance and easier model maintenance.

During the data processing phase, labeling your data is a crucial step. This process helps in organizing and constructing accurate datasets for ML. To facilitate this, you can engage third-party services specializing in data labeling, like Labelbox, CrowdAI, Docugami, and Scale, which offer expertise in image labeling and other types of data annotation. Additionally, platforms like Amazon SageMaker Ground Truth provide an automated solution for image data labeling.

Once your data is ready, the next step is to select a suitable algorithm and build the model.

Select and build

Before creating an ML model, you first want to understand business problems clearly, which will help you select the suitable algorithm. As explained in the previous section, you can choose from a list of algorithms and ML frameworks, broadly both *supervised* and *unsupervised* ML algorithms. This may be dictated by the data available. Once you select the suitable algorithm for your use case to build an ML model, you need a platform to train and develop your model.

Jupyter Notebook and RStudio are the most popular platforms among data scientists to build ML models. You can use cloud platforms such as Amazon SageMaker to spin up Jupyter Notebook or RStudio Workbench. AWS provides SageMaker Studio and RStudio, a web-based visual interface, in a place where all the developmental steps of ML will be performed by you.

To select your model, you can choose several built-in ML algorithms that you can use for various problem types, or get a number of models and algorithms available in the cloud market, making it easy to get started quickly.

The next step is to train and tune the model. Let's learn more about it.

Train and tune

To accelerate the training process, it's advisable to utilize a distributed compute cluster. This setup allows you to distribute the training workload across multiple computing resources, significantly speeding up the training phase. By employing such a cluster, you can parallelize the computation, which means different parts of the training data can be processed simultaneously. As a result, the model training completes faster, and the output, which your applications can utilize, becomes available sooner. This approach not only enhances efficiency but also enables the handling of larger datasets, contributing to the development of more accurate and robust ML models. Model tuning is also known as hyperparameter tuning, which is critical to achieving result accuracy.

You need to find the most effective version of a model by running multiple training jobs on your dataset using the chosen algorithm and varying ranges of hyperparameters. Following this, choose the correct hyperparameter values that yield a model that performs the best, as determined by a metric of your choice.

While you are tuning the model, it's crucial to have debugging capabilities that help capture real-time metrics during the training phase, such as training and validation accuracy, confusion matrices, and learning gradients. These metrics are essential for enhancing the accuracy of your model. Additionally, it's important to produce documentation to help improve model accuracy. You need to capture the input parameters, configurations, and results and categorize them as different experiments. This organization allows you to efficiently search for previous experiments by their characteristics, review previous experiments with their outcomes, and visually compare the results of various experiments to inform further adjustments and improvements. Most managed ML platforms, such as Amazon SageMaker, provide these features for you.

Amazon SageMaker also provides Autopilot, a feature that automates several aspects of model development. Autopilot examines raw data and applies feature processing techniques. It then picks the most suitable algorithms, conducts training, tunes multiple models, and monitors their performance. The models are ranked based on their performance metrics.

After finalizing your model, you need to deploy it and manage it in a production environment to gain valuable insights and achieve your desired outcomes.

Deploy and manage

You must deploy your trained ML model into a production environment to enable real-time or batch data predictions. Implement auto-scaling for your ML instances across various locations to ensure high redundancy and establish a RESTful HTTPS endpoint for your application. The application should be configured to have an API call to an ML endpoint to achieve low latency and high data processing speeds. This architectural approach facilitates rapid integration of your new models into your application, streamlining the process because changes to the model do not necessitate modifications in the application's code.

Data is subject to rapid changes due to factors like seasonality or unforeseen events, making it essential to continuously monitor your model for both accuracy and ongoing relevance to your business. A significant factor that can affect the accuracy of deployed models is if the data that is used for generating predictions differs from the training data used to train the model. For instance, changing economic conditions could drive new interest rates, which in turn could impact home purchasing predictions. This phenomenon is known as concept drift, whereby the patterns and correlations a model was trained on no longer hold true in the current data environment. To address this, you need to automatically detect concept drift in deployed models, with detailed alerts that assist in pinpointing the exact source of the problem.

Model compatibility is another crucial factor during deployment. Once a model has been built and trained using MXNet, TensorFlow, PyTorch, or XGBoost, for example, you can choose your target hardware platform from Intel, NVIDIA, or ARM. You need to compile your trained ML models to run optimally and efficiently in deploying compiled models to edge devices. This step ensures that your models not only deliver and provide high-performance and low-cost inference but also maintain cost effectiveness.

You should be able to run large-scale ML inference applications, which can include tasks like image recognition, speech recognition, NLP, personalization, and fraud detection. As you progress through the different stages of building and deploying ML models, understanding how to fine-tune and adapt them for efficient deployment and operation becomes key, particularly for applications that require real-time processing and responsiveness. Let's look at a reference architecture to connect all components.

ML reference architecture

The following example architecture depicts a bank loan approval workflow based on customer data built on the AWS cloud platform.

Customer data is ingested in the cloud and the ML framework decides on the customer loan application.

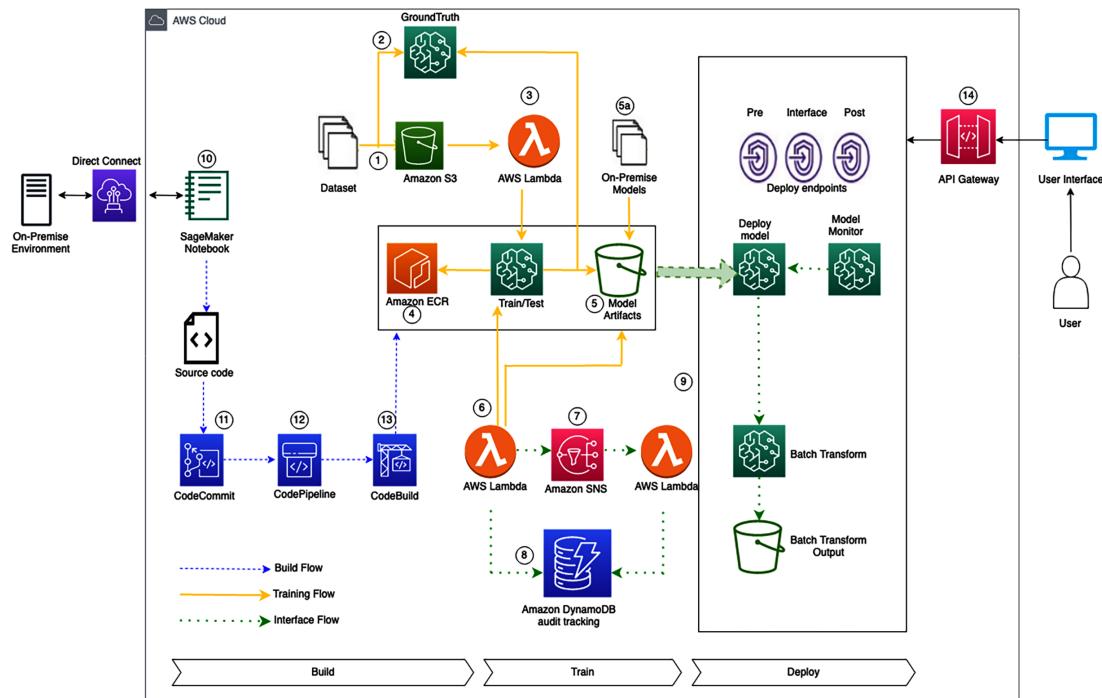


Figure 13.3: ML architecture in the AWS cloud

In designing the preceding architecture, some fundamental design principles to consider as a guide are:

- **Training workflow:**

1. Datasets enter the process flow using S3. This data may be raw input data or preprocessed from on-premises datasets.
2. Ground Truth is used to build a high-quality, labeled training dataset for ML models. If required, the Ground Truth service can label the data.

3. AWS Lambda can be used for data integration, preparation, and cleaning before datasets are passed to SageMaker.
 4. Data scientists will interface with SageMaker to train and test their models. The Docker images used by SageMaker are stored in ECR and can be custom images with custom toolsets created through the build flow steps or use one of the pre-built Amazon images.
 5. Model artifacts to be used as part of the deployment phase are output to S3. The output from the SageMaker model can also be used to label data using Ground Truth. Models that have been pre-built and trained on-premises or other platforms can be deposited into the model artifacts S3 bucket and deployed using SageMaker.
 6. AWS Lambda can trigger an approval workflow based on a new model artifact being deposited into the S3 bucket.
 7. Amazon Simple Notification Service (SNS) can be used to provide an automatic or manual approval workflow based on human intervention to deploy the final model. The supporting Lambda function takes the output from SNS to deploy the model.
 8. DynamoDB stores all model metadata, actions, and other associated data for audit tracking.
 9. To host the final model, we deploy the endpoint with the associated configuration as part of the final step in the workflow.
- **Build workflow:**
 10. SageMaker notebook instances are used to prepare and process data and to train and deploy ML models. These notebooks can be accessed via a VPC endpoint for the SageMaker service.
 11. CodeCommit provides the repository for the source code to trigger the build jobs required for any custom Docker images used by SageMaker.
 12. The CodePipeline service manages the end-to-end build pipeline for the custom Docker images and uses the CodeBuild service for the build/test phase.
 13. CodeBuild will build and perform unit testing of the custom Docker image and push it to Amazon ECR (this process can be managed centrally or by business functions requiring the tools).
 - **Deploy workflow:**
 14. As SageMaker endpoints are private, Amazon API Gateway exposes the model endpoint to end users for inference.

Batch transform jobs are typically utilized to obtain inferences for an entire dataset. By employing a trained model and a dataset, the output from the batch job is stored in S3. Additionally, you can utilize SageMaker Model Monitor to oversee production models and send alerts in case of any quality issues.

This section taught you about ML architecture with a CI/CD pipeline. We look at ML architecture design principles next.

Design principles for machine learning architecture

Designing an effective ML architecture requires a strategic approach, prioritizing scalability, maintainability, efficiency, and robustness. Here are some design principles that professionals typically adhere to when developing ML architectures.

Organizing the machine learning system into modules

Modularity breaks down the ML system into separate, interchangeable components or modules, each responsible for a distinct function. In an ML model, for instance, you could have one module for data ingestion, another for preprocessing, one for model training, and yet another for prediction serving. Consider a retail recommendation system: the data ingestion module might be responsible for collecting user interaction and purchase history, while another module uses that data to train a model that recommends products. The advantage is that if a better recommendation algorithm is developed, the training module can be replaced or updated without disrupting the data ingestion module.

In a **financial fraud detection system**, as another example, modularity allows the development team to isolate and update the prediction model whenever new fraud patterns are identified without altering the data collection or transaction monitoring modules. This compartmentalized approach promotes streamlined troubleshooting, targeted upgrades, and generally enhanced system manageability.

Ensuring scalability

Scalability refers to the ML architecture's capacity to gracefully handle increases in workload or demand, ensuring consistent performance. This is pivotal when managing larger datasets or when user requests grow significantly. For instance, in a streaming service like Netflix, the recommendation system must scale to accommodate the growing number of users and their viewing histories without compromising on the speed or accuracy of the recommendations. Scalability ensures that the service remains uninterrupted and consistently high-performing even as data and demand grow.

Another real-world example would be an **e-commerce platform** during a Black Friday sale. The system must scale to process and analyze exponentially higher transaction and user data.

Ensuring reproducibility

It is vital to ensure that ML models can reliably reproduce outcomes. This means that if the model is retrained with the same data, code, and parameters, it should produce the same results. An e-learning platform may use an ML model to personalize learning content for each user. If a particular model version yields impressive results, being able to reproduce it ensures a consistent user experience and facilitates future debugging and development.

Consider an ML model used for **diagnosing medical conditions** from imaging data in healthcare. Ensuring reproducibility means that the diagnoses remain consistent and reliable across different instances of the model, reinforcing trust in the automated system among healthcare professionals and patients and ensuring that scientific studies using the model are valid and verifiable.

Implementing data quality assurance

Data quality assurance means implementing mechanisms to validate and ensure the data's accuracy, completeness, and reliability, fed into ML models. For a system like a voice-activated virtual assistant, which is continually learning from user interactions to improve response accuracy, ensuring that the incoming data is accurate and relevant is paramount to train the model effectively. Faulty or low-quality data could lead the model to learn incorrect patterns, diminishing the user experience.

Take an **autonomous vehicle navigation system** as another example. Ensuring data quality is crucial because the decisions made by the ML model based on this data directly impact the safety and efficacy of the vehicle.

Ensuring flexibility

Flexibility in ML architecture refers to the ability to easily modify and adapt the system to accommodate changes or enhancements in data, technology, and requirements. A flexible system can integrate new data sources, manage different data types, and adapt to different algorithms or technologies as needed. Imagine a news aggregator app that uses ML to personalize content for users. Flexibility allows this app to easily adapt its model to new data sources (like a new news website) or to integrate new types of data (like video news or podcasts) without requiring a comprehensive architecture redesign.

With **customer support chatbots**, as another example, having flexibility allows the chatbot to adapt its responses and interaction style based on evolving user expectations and linguistic trends. Suppose the model identifies a shift in user interaction style or a surge in specific inquiries. A flexible architecture enables it to integrate new data or adjust its algorithms to enhance user interactions and satisfaction.

Ensuring robustness and reliability

Ensuring **robustness and reliability** means that the ML architecture should produce consistent, dependable results and be resilient to variations in the input data or system disturbances. For instance, a robust ML model for an email provider should consistently filter spam messages irrespective of the variety of spam techniques or message content. It should reliably protect the user's inbox even if spammers change their strategies or use different languages and terminologies.

In **automated stock trading**, the robustness and reliability of the ML models are vital to ensure that trading decisions are consistent and protected from volatile market conditions or deceptive trading activities. An ML system must recognize and navigate through market noise, erroneous data, or manipulative trading activities to safeguard investments and maintain investors' trust.

Ensuring privacy and security

Privacy and security involve safeguarding the data and the ML model from unauthorized access and ensuring that personal or sensitive data is handled ethically and complies with regulations. In a personal finance app that uses ML to provide financial advice, for example, it's imperative to protect the user's financial data and ensure that the model's predictions are secure from malicious attacks to preserve both user privacy and model integrity.

Considering **personalized marketing** as a use case, handling user data, such as shopping history, preferences, and personal details, with utmost privacy and security is essential. Ensuring the ML model, which curates personalized marketing content, adheres to data protection regulations and is resilient against data breaches protects the end users. It preserves the brand's reputation and legal compliance.

Ensuring efficiency

Efficiency is about maximizing the performance of the ML system while minimizing the resources used. An efficient ML model ensures that the computational, data storage, and other resource usages are optimized without compromising the model's output quality. In the case of mobile apps that use ML for features like image recognition or language translation, an efficient model will provide quick and accurate results without excessively draining the device's battery or utilizing intensive computational resources.

An example of **real-time fraud detection** in online transactions highlights the necessity of efficiency. The ML model must rapidly analyze transaction data and accurately identify fraudulent activities to provide immediate alerts or actions, all while managing computational resources to handle countless transactions happening every second. Efficiency ensures swift, accurate fraud detection without imposing unsustainable computational costs or latency in transaction processing.

Ensuring interpretability

Ensuring **interpretability** in ML architecture means the model outputs are understandable and explainable to humans. For instance, a healthcare platform that employs ML to assist doctors in diagnosing diseases should offer interpretations of its predictions, enabling doctors to understand the reasoning behind the diagnostic suggestions and, thereby, make informed decisions for patient care.

Consider an ML application in **credit scoring**. Interpretability is paramount to both the end users, who might want to understand the factors influencing their credit score, and the regulators, ensuring that the scoring model is not biased and is compliant with legal standards. An interpretable ML model can elucidate which factors (e.g., transaction history, loan repayments, etc.) influence the credit score, providing transparency and facilitating trust among users and regulatory bodies.

Implementing real-time capability

Real-time capability refers to the ML architecture's ability to process data and produce outputs in real time or near real time, which is crucial in scenarios that require instant decision making. An autonomous vehicle, for instance, utilizes ML to make immediate decisions based on real-time inputs from various sensors and cameras, such as identifying obstacles and deciding the optimal path. The architecture must process, evaluate, and act upon real-time data to navigate dynamic environments safely.

In **customer support** provided by virtual assistants and chatbots, real-time capability ensures customer queries are addressed immediately and accurately, enhancing user experience and satisfaction. The ML model must comprehend user inputs, process the relevant data, and generate real-time responses to facilitate smooth and coherent interactions, even as user inquiries and conversations diversify and escalate.

Ensuring fault tolerance

Fault tolerance implies that the ML architecture should maintain its functionality and produce reasonable outputs even when some system components fail or it encounters unexpected input data. For instance, an e-commerce recommendation system should continue to provide product suggestions to users even if specific data sources (like recent browsing history) are temporarily unavailable, ensuring continuous user engagement and potential sales.

In **industrial equipment monitoring** using ML to predict maintenance needs and detect failures, fault tolerance ensures that the system can still provide valuable insights even when some sensors fail or provide erratic data. The ML model should identify and manage such anomalies, providing reliable equipment health assessments and ensuring safe and continuous operations in the industrial setup.

Through adherence to these principles in ML architecture, models can robustly navigate through various real-world applications, from ensuring safe and efficient operations in industries to providing real-time, insightful interactions in customer support.

ML is applicable everywhere; for example, it can be applied in solving customer problems such as predictive maintenance, providing accurate forecasting for businesses, or building personalized recommendations for end users. ML use cases are not only limited to customer problems but also help you to handle your IT applications by optimizing your workload with predictive scaling, identifying log patterns, fixing errors before they cause issues in production, or budget forecasting for IT infrastructure. So, solutions architects must know about ML use cases and associated technology.

Earlier in this book, you learned about DevOps to automate and operationalize your development workload. As ML is becoming mainstream, MLOps has become essential to learning ML at scale in production. Let's explore more details on operationalizing the ML workload with MLOps.

MLOps

An ML workflow is a set of operations developed and executed to produce a mathematical model, which eventually is designed to solve a real-world problem. But these models have no value until they are deployed in production other than proofs of concept. ML models almost always require deployment to a production environment to provide business value.

At its core, MLOps fundamentally focuses on transitioning an experimental ML model into a fully operational production system. MLOps is an emerging practice, different from traditional DevOps due to the unique nature of the ML development life cycle and the specific ML artifacts it produces. The ML life cycle revolves around discerning patterns from training data, making the MLOps workflow particularly sensitive to changes in data, as well as variations in data volumes and quality.

A well-developed MLOps practice should support the monitoring of ML life cycle activities as well as the ongoing supervision of models once they are operational in production environments. This dual focus ensures both the efficiency of the development process and the effectiveness of the deployed models.

MLOps implementation makes it simple for organizations to feel confident in building a mature MLOps framework, eliminating extensive coding. Like any other workload, you want to develop MLOps by applying best practices such as security, reliability, high availability, and performance, and considering the cost for the deployment phase of the ML life cycle. Let's have a look at a few MLOps principles.

MLOps principles

Any changes in the code, training data, or model should trigger the build process in the ML development pipeline to make sure your model is performing well by accommodating changes immediately.

An ML pipeline should follow these MLOps principles while developing ML systems:

- **Automation:** The deployment of ML models in production should be automated. The MLOps team should automate the end-to-end ML workflow from data engineering to model inference in production without any manual intervention. This automation makes sure that there is no lapse in the production model where there are changes in training data and your model stays relevant. The MLOps pipeline can trigger model training and deployment based on events such as calendar scheduling, messaging, monitoring, data changes, model training code changes, and application code changes.
- **Versioning:** Versioning is an essential aspect of MLOps. Every ML model and related script version should be maintained in a version control system such as GitHub to make the models reproducible and auditable.
- **Testing:** ML systems require extensive testing and monitoring. Each ML system should have at least the following three scopes for testing:
 - Feature and data tests, which include validating data quality and selecting the right features for your ML model
 - Model development tests, which include business metric tests, model staleness tests, and model performance validation tests
 - ML infrastructure tests, which include ML API usage tests, complete ML pipeline integration tests, and training and production server availability tests
- **Reproducibility:** Every phase of an ML workflow should be reproducible, which means that ML model training, processing of data, and emplacement of the ML model must deduce similar results for similar input. It will ensure a robust ML system.
- **Deployment:** MLOps integrates the principles of ML with the culture of DevOps, emphasizing CI/CD and **continuous training/continuous monitoring (CT/CM)**. Through automated deployment and testing, MLOps facilitates the early detection of issues, enabling quick rectifications and iterative learning. This approach streamlines the process of deploying ML models into production, ensuring they remain effective and up to date.

- **Monitoring:** Over time, the performance of a model may degrade in production due to factors such as data drift. This scenario necessitates the continuous deployment of new or updated models to counteract any decline; they must be shipped into production constantly to address performance decline or to enhance model fairness. After deploying an ML model, it's critical that a monitoring system is implemented to ensure that the ML model performs according to expectations and maintains its effectiveness in providing accurate and reliable outputs, which is essential for maintaining the overall quality and trustworthiness of the ML application as expected.

Having learned about MLOps design principles in this section, let's consider some best practices to apply MLOps in your ML workload.

MLOps best practices

Due to many moving parts (data, model, or code) and challenges in solving business problems using ML, MLOps can be a challenging task.

Based on the principles outlined in the previous section, the following are the best practices that ML engineers/full-stack data scientists should practice while deploying ML solutions in production, which will help reduce technical debt and maintenance overhead in ML projects and drive the most business value out of them:

- **Design considerations:** To develop a maintainable ML system, the architecture/system design should be modular and, as much as possible, loosely coupled. Implementing a loosely coupled architecture enables different teams within an organization to operate independently. This means they don't have to depend on other teams for support or services. As a result, each team can work more swiftly and efficiently, contributing to the overall value and productivity of the organization.
- **Data validation:** Data validation is crucial for a successful ML system. In a production environment, data can present several challenges. One such issue arises if the statistical properties of the production data are different from those of the training data, indicating potential problems with properties, the training data itself, or the data sampling process. Additionally, data drift can occur, which might cause the statistical properties of data over time to change for successive batches. This drift can impact the model's performance, as it was trained on data with different statistical characteristics.
- **Model validation:** Reusing models is different from reusing software. It's best to tune models to fit each new scenario. Ensuring that models are thoroughly validated before they are moved into production is very important. To confirm that the model performs effectively on live data, it's important to conduct both online and offline data validation. This process helps to establish that the model's predictions are accurate and reliable under actual operating conditions.
- **Model experiment tracking:** It's essential to meticulously document all experiments conducted with your ML models. Experimentation in ML may involve trying out different combinations of code (encompassing preprocessing, training, and evaluation methods), datasets, and hyperparameters. Each unique combination of these elements produces metrics, which you should compare against the outcomes of your other experiments. This comparison is key to understanding which approaches are most effective and optimizing your ML models accordingly.

- **Code quality check:** Every ML model specification (ML training code that creates an ML model) should undergo a code review phase. Checking the quality of the code as the initial path of a pipeline activated by a pull request is a nice practice in general.
- **Naming conventions:** Following a standard naming convention (like *PEP8* for Python programming) in your ML coding practices is an effective strategy to help mitigate the challenges posed by the **Changing Anything Changes Everything (CACE)** principle. Consistent and clear naming conventions facilitate easier understanding and modification of the code. This not only helps in maintaining the integrity of the project when changes are made but also enables new team members to establish familiarity with your project quickly.
- **Model predictive service performance monitoring:** Other than project metrics (such as **root mean square error (RMSE)**) that calculate the behavior of a model with reference to the aim of the business, operational metrics such as latency, scalability, and service updates are also crucial to monitor to avoid business losses.
- **The CT/CM process:** In a production environment, model performance can deteriorate due to factors such as data drift. This necessitates the ongoing deployment of updated models to enhance or maintain the model's fairness and accuracy. To effectively manage this, a CT/CM process is essential. CT ensures that models are regularly updated and trained with the latest data, while CM keeps track of the model's performance in real time, identifying any issues or deviations that might occur due to changing data patterns. Together, CT and CM form a robust framework for ensuring the long-term reliability and effectiveness of ML models in production.
- **Resource utilization:** Understanding the requirements of your system during both the training and deployment phases is crucial for efficient operations. This insight helps your team effectively optimize the resources used for your experiments, which in turn can lead to significant cost savings. Proper resource management ensures that you allocate just the right amount of computational power, memory, and other resources necessary for the tasks at hand, without underutilizing or over-committing resources. This balance is key to maintaining both the performance and the cost effectiveness of your ML projects.

MLOps plays a crucial role in the industrialization of AI. MLOps collaborates ML, data engineering, and DevOps to effectively build, deploy, and maintain ML systems in production.

Deep learning is now the go-to mechanism to solve complex ML problems. Let's learn more about deep learning.

Deep learning

ML is about forecasting and solving complex problems using NLP, enabling computers to understand, interpret, and generate human language in a valuable and meaningful way. NLP is used in numerous applications, including language translation, sentiment analysis, chatbots, and voice assistants, allowing for more intuitive, human-like interaction with machines. While ML needs a pre-defined set of labeled data for supervised learning, deep learning uses a neural network for unsupervised learning to simulate human brain behaviors, using a large amount of data to develop ML capabilities. A neural network is a series of algorithms that recognize underlying relationships in a set of data through a process that mimics the way the human brain operates.

Deep learning involves a neural network of multiple layers where you don't need to do data labeling upfront. However, depending on your use case, you can use both labeled and unlabeled data with deep learning. The following diagram shows a simple deep learning model:

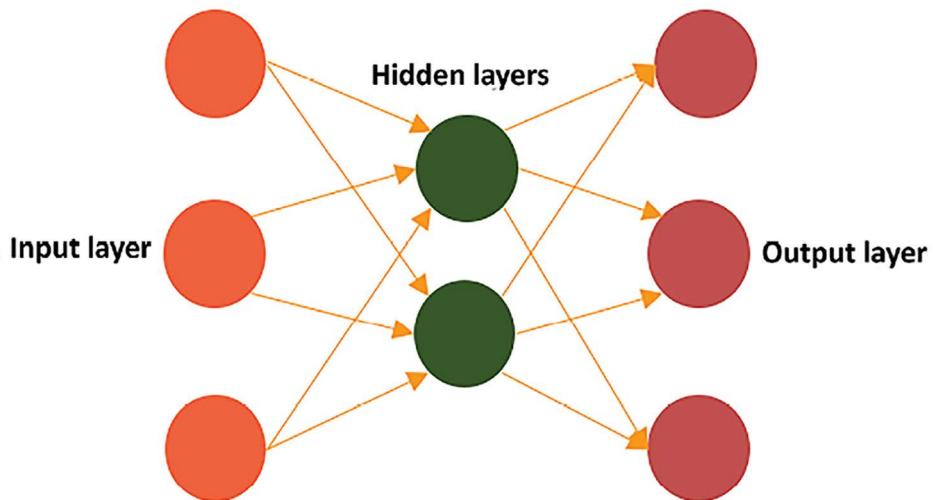


Figure 13.4: An overview of deep learning layers

In the preceding diagram, a deep learning model has interconnected nodes where input layers provide data input through various nodes. This data goes through multiple hidden layers to calculate the output and deliver the final model inference through the output node layer. The input and output layers are visible layers, and learning happens in the middle layer through weights and bias, as shown in the following diagram:

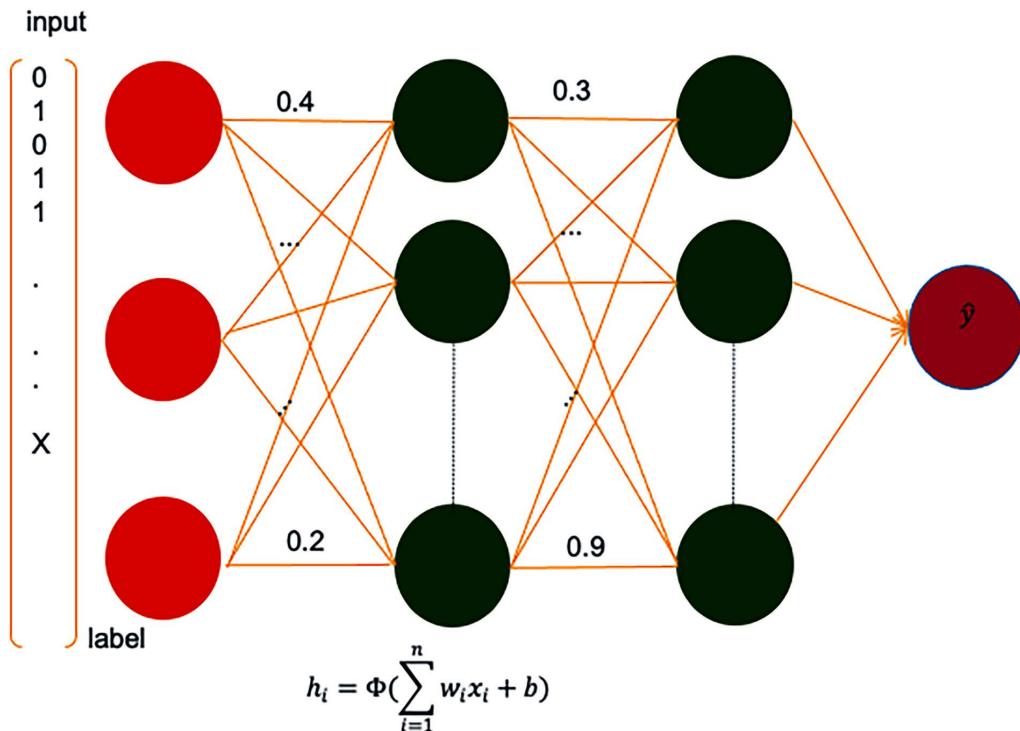


Figure 13.5: Deep learning neural network model

In the preceding diagram, you can see a series of hidden layers between where each layer applies some weight functions to the interconnected nodes to learn the pattern, in the same way as a human brain. You can see **label** data coming in as input and going through neural network nodes with their weight (0.2, 0.4, 0.3, and 0.9) indicated between vertices.

Weight is a neural network parameter that plays a role in transforming input data as it passes through the hidden layers. Essentially, the weight determines the extent to which a given input will influence the output. It can be thought of as representing the strength or intensity of the connection between nodes in the network.

For instance, if the weight from node A to node B is high, it implies that neuron A has a more significant influence over neuron B. Weights that are close to zero indicate that changing this particular input will have minimal or no impact on the output. Conversely, if the weights are negative, it suggests an inverse relationship – this means increasing the input will decrease the output and vice versa. This mechanism of weights is fundamental to how neural networks process and learn from data.

The preceding learning method is called **forward propagation**, where data flows from the input layer all the way to the output layer. Here, the output of one layer is fed as the input to the next, leading up to the final output.

On the other side, there's another technique known as **backpropagation**. This involves calculating the error in the network's predictions (the discrepancy between what the network predicts and the actual outcome). The network uses algorithms to calculate prediction errors and then adjusts its internal parameters—the weights and biases—based on this error. This adjustment happens in reverse, starting from the output layer and moving backward through the layers, which is why it's called “backpropagation.”

Through the combined use of forward and backward propagation, a neural network is able to learn and improve. It processes data (forward propagation), identifies any inaccuracies in its predictions (backpropagation), and then tweaks its parameters to reduce these errors. This cycle is key to training the neural network to gradually become more efficient and accurate at its task with the training algorithm.

Deep learning encompasses different types of neural networks, each suited for different applications. The two most frequently used are:

- **Convolutional neural networks (CNNs):** These are adept at processing data with a grid-like topology, such as images, making them highly effective for tasks involving visual input, such as computer vision and image classification tasks.
- **Recurrent neural networks (RNNs):** RNNs excel in handling sequential data, making them ideal for tasks that involve understanding language and speech, such as NLP and speech recognition.

Some of the most popular frameworks for building these neural network models are TensorFlow, which has built-in support for various neural network architectures, and MXNet, which also supports a range of network architectures and is known for its efficiency and scalability, particularly in the context of high-performance deep learning applications. In addition to this, other popular deep learning frameworks are PyTorch, Chainer, Caffe2, ONNX, Keras, and Gluon.

This section has provided you with a high-level view of deep learning. It is a complex topic and requires an entire book to cover the basics. You will find multiple books available on each of the frameworks. Deep learning model training requires a lot of processing power and could be very costly. However, public cloud providers such as AWS, GCP, and Azure make it easy to use high-powered GPU-based instances to train these models with the pay-as-you-go method.

Deep learning in the real world

Deep learning is widely popular, and there are multiple use cases of deep learning across various industries. Let's take a look at some of the examples.

Healthcare: diagnosis and prognosis

Deep learning models assist professionals in healthcare by providing a second opinion and sometimes even spotting details humans might miss. These models are trained on vast datasets of medical images, learning to identify features associated with diseases and conditions, and they can predict the probability of a patient having a particular disease. For instance, Google's DeepMind has developed a model to spot eye diseases in scans. By analyzing 3D scans of patients' eyes, the deep learning system can recommend how patients should be referred for treatment.

Autonomous vehicles: navigation and safety

Deep learning models assist autonomous vehicles in understanding their surroundings, making decisions, and navigating the world. They are trained on extensive datasets of various driving scenarios and learn to recognize objects (like pedestrians and other vehicles), understand road signs, and make safe driving decisions (like when to brake or steer). Deep learning essentially allows these vehicles to interpret and understand the world around them, making automated driving possible and progressively safer as the technology evolves.

Tesla, Waymo, and other companies utilize deep learning for their autonomous vehicles. These vehicles are equipped with many sensors that feed data into deep learning models, enabling them to make real-time decisions.

Manufacturing: quality control and predictive maintenance

In manufacturing, deep learning models optimize operations and enhance quality control. By analyzing data from manufacturing processes, models can identify manufacturing anomalies or defects in products early in the production chain, ensuring high-quality output. General Electric employs deep learning for predictive equipment maintenance, using models that analyze data from machinery to predict when they might fail or need maintenance, thereby reducing downtime.

With its capacity to derive insights from vast, complex datasets, deep learning finds applications across diverse real-world scenarios, driving innovations and optimizing operations in various industries. From healthcare, where it aids in diagnostics, to manufacturing, where it ensures optimal operations and quality, deep learning is an integral part of technological advancements in numerous sectors.

NLP

NLP aims to understand and read, as well as utilize, the human language well. It combines AI and computational linguistics so that computers can process human language through spoken words or text. Let's look at some of the use cases of NLP.

Chatbots and virtual assistants

One of the common applications of NLP is in creating chatbots and virtual assistants like Siri, Alexa, or customer service chatbots on various websites.

Chatbots engage in interactive conversations with users, often assisting them in finding information, answering queries, or facilitating transactions. They leverage NLP to understand user inputs (questions or commands) and generate relevant responses. By analyzing the text, NLP models discern the intent behind the user's message and respond accordingly. This application is widely utilized across industries like retail, banking, and customer service to enhance user experience and provide immediate responses to customer inquiries.

Sentiment analysis

Companies use sentiment analysis, also known as opinion mining, to gauge sentiment toward their brand, products, or service by analyzing written or spoken words from customers, such as customer reviews and social media comments.

Sentiment analysis models, developed with NLP, examine text data to determine the emotional tone behind them, categorizing sentiments as positive, negative, or neutral. For instance, a company could analyze product reviews to identify whether customers are generally satisfied or dissatisfied. This information is crucial for businesses to understand customer perception and strategically adjust their products or services accordingly. Modern contact center solutions, such as Amazon Connect, have revolutionized customer service by incorporating real-time analysis of customer conversations. These systems can analyze voice interactions to determine the customer's sentiment, which can assist customer support representatives during calls to drive customer engagement.

Text summarization

Automated summarization tools that generate concise summaries of lengthy documents or articles employ NLP to understand and condense the content effectively.

Text summarization involves reducing a text document without losing its critical information and presenting it in a shortened form. NLP models extract essential details and key points from the document, providing a summarized version that retains the core message. This can be particularly useful in sectors like law or research, where professionals must review vast documents and extract pertinent information efficiently.

Machine translation

Google Translate, which can translate text into different languages, heavily relies on NLP to understand and translate the text accurately.

Machine translation models utilize NLP to comprehend the text and its context in the source language and generate equivalent text in the target language. NLP ensures that the translation adheres to grammatical and syntactical rules and maintains the original message's meaning and context. This has global implications, breaking down language barriers and facilitating international communication and information exchange.

By enabling machines to understand and process human language, NLP opens up many applications that enhance communication, provide insights, and streamline processes across numerous domains, including customer service, marketing, and global communication.

Large language models (LLMs), such as **generative pretrained transformer (GPT)** models, have emerged as transformative tools in NLP, showcasing impressive capabilities across various NLP tasks. These models are designed to understand, generate, and work with human-like text, enabling them to engage in activities like text completion, summarization, and translation. You will learn more about various LLMs in the next chapter.

Overall, ML and AI are vast topics and warrant multiple books to understand them more thoroughly. In this chapter, you were just given an overview of ML models, types, and workflows.

Summary

In this comprehensive chapter, you journeyed through the fundamental concepts and practical applications of ML. You began by understanding the core principles of ML and its close relationship with data science, emphasizing the pivotal role of data in training and evaluating ML models. You explored different types of ML, ranging from supervised and unsupervised learning to reinforcement learning and deep learning. Each type was elucidated with real-world examples and common algorithms, providing you with an understanding of when and how to apply them.

Next, you delved into the critical concepts of model overfitting and underfitting, exploring the delicate balance required to achieve model generalization. You examined various strategies and techniques to address these challenges effectively.

Popular AI tools and frameworks were covered and the chapter also ventured into cloud-based ML, demonstrating the advantages and capabilities of harnessing cloud platforms for ML projects. The role of data in ML was expounded, with an emphasis on feature engineering and selection, as well as the meticulous process of building, training, tuning, deploying, and managing ML models. We discussed the design principles for ML architecture, providing best practices for architecting scalable and efficient ML systems.

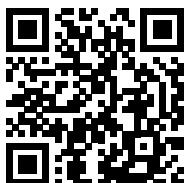
MLOps was explored as an essential component of ML project development, providing a structured framework for designing and implementing robust ML systems.

Deep learning took center stage, revealing its profound impact on various industries, from image recognition to NLP. We dissected the architecture of deep learning models and explored practical applications in the real world.

The chapter culminated in exploring NLP and LLMs, demonstrating how these models transform communication, translation, and content generation. Taking a step further, GenAI is a key technology for the next evolution. In the next chapter, you will learn about GenAI architecture, various FMs, and available offerings that will help you gain expertise in GenAI.

Join our book's Discord space

Join the book's Discord workspace to ask questions and interact with the authors and other solution architecture professionals: <https://packt.link/SAHandbook>



14

Generative AI Architecture

Generative AI technology goes beyond mere industry jargon – that is, it is an advanced instrument used for reshaping business operations by automating essential tasks such as content generation, image creation, and knowledge assistance. Generative AI represents a thrilling leap forward in the tech world, igniting significant enthusiasm among those passionate about technological innovation. Referred to as GenAI, an abbreviation of Generative artificial intelligence, this form of technology stands out for its remarkable ability to independently produce new content, such as text, images, music, videos, coding, and so on, with capabilities that closely mimic human-like creativity.

The use of generative AI is increasing in different business areas. It can greatly reduce the time, resources, and costs needed to operate a business when used well. For instance, ChatGPT can assist in creating marketing campaigns for products or serve as a travel planner, while Midjourney can generate images in just a second.

You might have encountered generative AI applications like ChatGPT, Midjourney, Gemini (formerly Bard), Amazon Q, and Claude.ai, among others. This technology learns from a lot of information it gathers, including from the internet, and uses that knowledge to develop new content. It's like having a smart assistant that can generate all sorts of things without needing human input for every detail. However, it's crucial to understand that this isn't magic – it's the outcome of a great deal of smart thinking and advancements in the field of technology.

But the most exciting part is that we can use these foundation models in many ways. For instance, they can be used to generate creative content, automate customer service with chatbots, enhance data analysis, provide personalized recommendations, streamline language translations, and even aid in research by summarizing complex documents. In this chapter, you'll learn about generative AI in more detail, including:

What is generative AI?

- Generative AI use cases
- The basic architecture of generative AI systems
- Popular generative AI foundational models

- How to start with generative AI
- Generative AI reference architecture
- Challenges in implementing generative AI

Get ready for an exciting journey into the world of generative AI. We will uncover the mysteries behind its remarkable abilities to influence the direction of our new world.

What is generative AI?

Generative AI is artificial intelligence with the remarkable ability to develop new content and ideas. This includes things like having conversations, creating stories, producing images and videos, and even making music.

In December 2022, the design team at the **Laboratory for Artificial Intelligence in Design (AiDLab)** located in Hong Kong orchestrated a groundbreaking fashion exhibition titled **Fashion X AI** (<https://www.fashionxai.com/event-highlights-fashionshow>). This showcase was unique because every design featured in the event was created by AI, drawing inspiration from mood boards, color palettes, and concepts provided by human designers.

Like other types of AI, generative AI relies on **machine learning (ML)** models. These models are quite large and are pre-trained using vast amounts of data. We often call these models **foundation models (FMs)**.

The FMs we have today (like OpenAI GPT-4 or Google Gemini for large language tasks, Stable Diffusion from Stability AI for converting text into images, and the OpenAI Sora text-to-video generator) can perform various tasks across many different areas. They can write blog posts, generate images, solve math problems, hold conversations, and even answer questions based on information in a document. These models are incredibly versatile and have the potential to revolutionize how we create and interact with content.

Generative AI has the potential to bring about sweeping changes to the global economy. According to Goldman Sachs, generative AI could drive a 7% (or almost \$7 trillion) increase in global GDP and lift productivity growth by 1.5% over 10 years. You can read more details in the Goldman Sachs **Artificial Intelligence (AI)** report at <https://www.goldmansachs.com/intelligence/artificial-intelligence/>.

FMs stand out due to their size and general-purpose nature, setting them apart from traditional ML models that are designed for specific tasks like sentiment analysis, image classification, and trend forecasting. Unlike these traditional models, which require gathering labeled data, training, and deployment for each task, FMs offer a more versatile approach. A single pre-trained FM can be adapted for various tasks. Moreover, these models can be tailored to perform domain-specific functions that are unique to individual businesses. Importantly, this customization can be achieved using only a fraction of the data, along with the computing resources that are needed to train a model from scratch.

The success of FMs can be attributed to three key reasons:

- **Transformer architecture:** The transformer architecture, a type of neural network, plays a pivotal role. It is efficient, scalable, and parallelizable and can effectively model input and output data relationships.

- **In-context learning:** A groundbreaking training paradigm called in-context learning has emerged. This approach allows pre-trained models to be equipped with instructions for new tasks or just a few examples. This eliminates the need for additional training on labeled data, enabling models to be immediately applied to new tasks.
- **Emergent behaviors at scale:** As model sizes increase and larger datasets are used, models begin to display previously unseen capabilities. This phenomenon is referred to as “emerging capabilities.” For example, larger models can generate more coherent and contextually relevant text, recognize complex patterns in data, and even perform tasks like image recognition and language translation with greater accuracy. They can also handle multi-step reasoning questions, provide detailed explanations, and generate creative content, such as writing music or creating artwork, with nuanced understanding and creativity. Larger models have the potential to perform tasks that are beyond their capabilities before reaching a critical size.

To understand better, let's look at some use cases where generative AI can help.

Generative AI use cases

Let's look at various use cases across different categories such as customer experience, employee production, and business operations efficiency, and learn how generative AI is enhancing existing AI capabilities and bringing forth entirely new possibilities:

Customer experience transformation

Generative AI is changing the game in how customers interact with businesses. Imagine you're shopping online for shoes. A generative AI-based virtual assistant on the website greets you and helps you find the perfect pair based on your style and size preferences. It can even show you images of the shoes and answer any questions you have. Let's look at some more such use cases where generative AI can help to improve customer experience and engagement:

- **Chatbots and virtual assistants:** Imagine you visit a website and a chatbot pops up to help you out. Generative AI powers these chatbots. They can talk to you like humans, understand your questions, and provide helpful answers.
- **Intelligent contact centers:** Generative AI is at work when you call a customer service hotline. It ensures that your interactions are more personalized, efficient, and satisfying. Your issues are addressed promptly and accurately.
- **Personalization:** Have you noticed that recommendations on platforms like Netflix and Amazon understand your preferences? That's generative AI in action. It learns from your behaviors and tailors its suggestions to match your tastes.
- **Content moderation:** Generative AI helps keep things clean and safe on social media and other platforms. It scans user-generated content, like comments and posts, to make sure they follow the rules and guidelines.

Employee productivity enhancement

Generative AI isn't just for customers; it's also boosting employee productivity. Imagine you're working on a project and need to write a report about it. Instead of starting from scratch, you use generative AI to help you write an introduction and critical points. This gives you a head start, and you can focus on adding your insights and expertise. Here are some use cases where generative AI helps to boost employee productivity:

- **Conversational search:** You might use a search system when you need information. Generative AI makes these systems smarter. You can ask questions in everyday language, and the AI will understand and give you the correct answers.
- **Content creation:** Writing reports and articles can take a lot of time. Generative AI helps here, too. It can generate content sections, like summaries or explanations, which you can use to create polished documents.
- **Text summarization:** Imagine you're reading a long research paper. Instead of going through all the pages, generative AI can summarize the main points. This saves time and helps you grasp the essential information faster.
- **Code creation:** For programmers, writing code is a big part of the job. Generative AI can assist by suggesting code snippets based on what you're trying to achieve. This speeds up coding tasks and makes development smoother.

When integrating generative AI into enterprise scenarios, it's crucial to navigate legal considerations around the content it generates. You need to understand the source of the content, and establishing clear ownership rights is essential to prevent intellectual property disputes. There are potential barriers to adoption, such as concerns over copyright infringement and data privacy. To mitigate these risks, your enterprises can consider developing their own generative AI assistants using proprietary data. This approach not only helps avoid legal complications but also ensures that the generated content is aligned with the organization's specific needs and retains its unique value.

Optimizing business operations

Generative AI isn't limited to customer interactions; it also enhances various operational aspects. In a manufacturing plant, machines are monitored by sensors. Generative AI analyzes the data from these sensors and predicts when a machine will likely have issues. This allows maintenance to be scheduled proactively, preventing unexpected breakdowns and production interruptions. The following are some use cases where generative AI helps to improve business operations:

- **Intelligent document processing:** In businesses, there are many documents to handle. Generative AI can read and understand these documents, extracting meaningful information automatically. This saves time and reduces errors. For example, the generative AI model can ingest mortgage lending documents and answer questions about mortgage rates, payment terms, duration, etc.
- **Predictive maintenance:** For companies that use machinery, predicting when equipment needs maintenance is crucial. Generative AI analyzes data from machines and systems to forecast maintenance requirements, preventing breakdowns and minimizing downtime.

- **Quality control and visual inspection:** Ensuring products meet high standards is essential in manufacturing. Generative AI can examine images of products, identifying defects or inconsistencies that human eyes might miss.
- **Data augmentation:** Training AI models requires a lot of data. Generative AI helps here by creating synthetic data that can be used to improve the accuracy and reliability of these models.

In this section, you learned about generative AI use cases. Now, let's learn what goes on behind the scenes by learning about generative AI architecture.

The basic architecture of generative AI systems

At the heart of generative AI systems is a massive FM. FMS are large-scale, pre-trained models that have been trained on vast datasets and can be fine-tuned or adapted for a wide range of tasks and applications. To understand the architecture of generative AI systems, let's break it down into simple components:

- **Generator:** The core element that generates new data, whether it's images, text, music, or other forms of content. The generator learns patterns and relationships from existing data and uses this knowledge to produce new, similar content. For example, the generator takes random noise in image generation and produces images that resemble the training data.
- **Latent space:** A conceptual space where the model represents data in a compressed form. It's like a compact representation of the data that the generator uses to create new content. This is a lower-dimensional vector space from which the generator generates data. This is like the secret recipe book an artist uses. It helps the generator come up with different types of creations. For instance, the latent space could represent different writing styles in text generation. In image synthesis, the latent space might represent different features like color and texture.
- **Loss function:** A measure of how well the generated content matches the desired output. The loss function helps the model learn and improve over time by minimizing the difference between generated and real data. Imagine a coach telling an artist how close their work is to perfection. The artist learns and gets better by following this guidance.
- **Training data:** The existing data that the model learns from. It could be images, text, audio, or any other type of content that is available from which the model learns. Just like a chef learns by tasting different foods, the generator learns what it should create from examples. For instance, if it's creating songs, it learns from listening to existing songs.

Types of generative models

Before learning about the generative models, let's learn how they differ from the typical ML discriminating model. A typical ML discriminating model, also known as a discriminative model, is designed to differentiate between different classes or categories of data. Unlike generative models that aim to generate new data points, discriminative models focus on distinguishing existing data points based on their features. These models predict the probability of a given outcome based on the input data. Common examples of discriminating ML models include logistic regression, support vector machines, and so on. You learned about this concept in detail in *Chapter 13, Machine Learning Architecture*.

Generative models distinguish themselves from discriminating models, which are tailored to categorize or tag text based on predefined groupings. Discriminating models are commonly deployed in applications such as facial recognition, where their training focuses on identifying specific features or attributes within a person's visage.

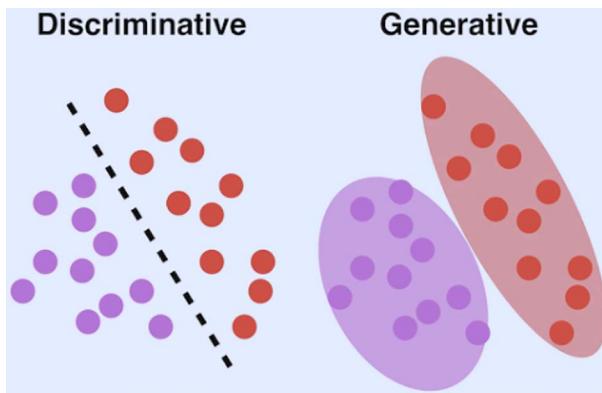


Figure 14.1: Generative models versus discriminating models

As shown in the preceding diagram, generative models try to understand the patterns and structure within the data. It's like they are learning the hidden rules of a game and then using those rules to create something new that looks like the original game. Discriminating models, on the other hand, focus on telling things apart. They are like detectives who are trained to recognize differences between things. Discriminative models are typically chosen for supervised learning tasks where the goal is classification or regression, whereas generative models are selected when the goal is to understand data distribution or generate new data points.

Generative AI encompasses various models that create new content. We'll look at some notable types in the following subsections.

Generative Adversarial Networks (GANs)

GANs are made up of two components: the generator and the discriminator. The generator's role is to produce content, and the discriminator's job is to judge the authenticity of that content, determining whether it's real or counterfeit. They engage in a sort of "competition," where the generator aims to create content convincing enough to deceive the discriminator. As this process continues, the generator progressively improves at crafting content that appears increasingly realistic. The following diagram shows the workings of the GAN model:

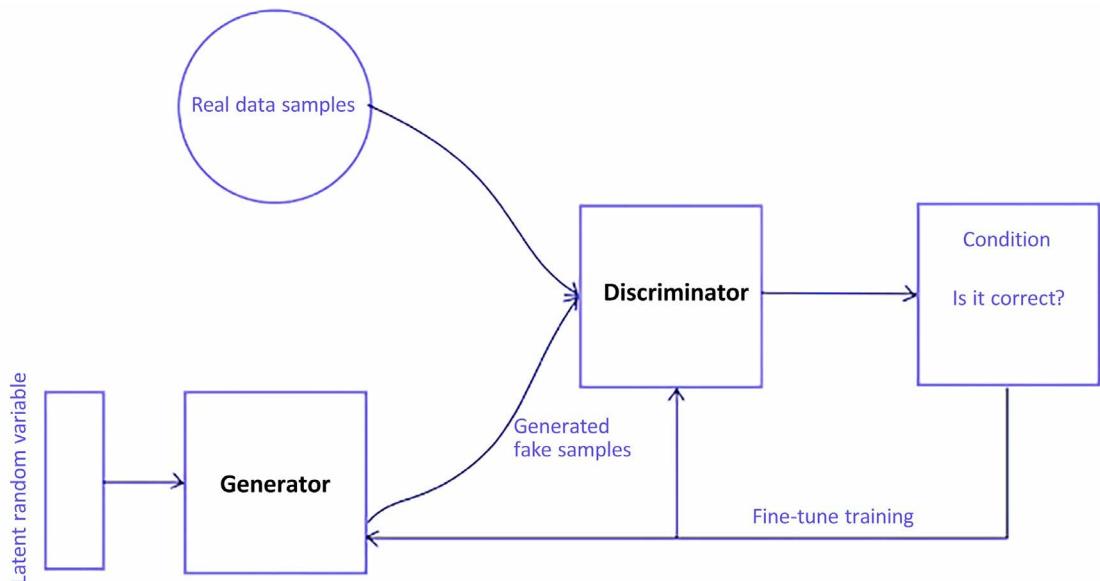


Figure 14.2: Training flow of GANs

The preceding diagram represents the basic structure of a GAN. Let's look at each step with the example of image creation:

- **Generator:** This component of the GAN takes in random noise as its input. This noise is often referred to as a “latent random variable.” The generator’s role is to produce data that is similar to the real data it has been trained on. Imagine this as an artist in training, initially creating random sketches based on some basic patterns of artwork. For example, the generator starts by creating random images that are intended to look like famous paintings.
- **Real data samples:** These are authentic data instances that the GAN is designed to mimic. They serve as the benchmark for the quality of the data that the generator creates. In our example, these are actual famous paintings from history, the masterpieces that the generator is attempting to emulate. For example, authentic paintings by artists like Van Gogh or Picasso are fed into the GAN as examples of “real” artwork.
- **Generated fake samples:** The generator uses the input noise to create new data samples. These samples are intended to be indistinguishable from the real data samples, although they are entirely generated by the model. These are the new images the generator creates, trying to replicate the quality and style of the real artwork samples. For example, the generator produces images that mimic the brushstrokes and color schemes of Van Gogh or Picasso’s works.

- **Discriminator:** This component takes in both real data samples and fake data samples generated by the generator. Its job is to distinguish between the two, effectively deciding whether each sample it receives is real or fake. Think of this as an art critic who examines both the real masterpieces and the generated images to decide whether the new images are genuine artworks or imitations. For example, the discriminator reviews the images, trying to determine which ones are the actual Van Gogh or Picasso paintings and which are the imitations.
- **Condition:** The discriminator makes a decision on whether the data is real or fake and provides this information as feedback to the generator. The art critic (discriminator) assesses the generated images and gives feedback, such as pointing out which aspects make them look fake. For example, the discriminator notes that the color palette in a generated image doesn't quite match the original artist's style and labels it as fake.
- **Fine-tune training:** Based on the discriminator's assessments, the generator adjusts its parameters in an effort to create better fake samples that are more likely to fool the discriminator. This feedback loop continues with the discriminator also improving its ability to discern real from fake. This adversarial process continues until the generator becomes adept at creating realistic data. Based on the feedback, the artist in training (generator) learns from the criticism and improves its technique to create more convincing artwork. For example, taking the feedback into account, the generator adjusts its technique, maybe altering the color mix or brushstroke style to better imitate the masterpieces.

The generator and the discriminator are essentially in a continuous game, with the generator trying to produce increasingly realistic data, and the discriminator striving to get better at telling real data from fake. The “training” is complete when the discriminator can no longer reliably distinguish fake data from real data, meaning the generator’s output is convincingly realistic.

GANs have several practical applications across business domains; many popular tools leverage this model.

Variational Autoencoders (VAEs)

Imagine you have a massive pile of Lego blocks in various shapes and sizes, and your task is to store them neatly in a small box. But there’s a catch: you can only store instructions on how to rebuild the original Lego structures, not the blocks themselves. This is similar to what VAEs do with data.

In this analogy, the “encoder” is like you taking each Lego structure, figuring out the best way to rebuild it using fewer blocks, and then writing down those instructions. The space inside your small box, where you keep these instructions, is like the “latent space” – a compressed version of the original structures.

Later, when you want to rebuild a Lego structure, you look at your instructions in the box. The “decoder” is like you following those instructions to build a new Lego structure that looks very similar to the original one using a new set of blocks.

So, a VAE takes large, complex data (the original Lego structures), compresses it into a simpler, smaller form (instructions in the box), and then uses that compressed form to generate new data that resembles the original data (rebuilding the Lego structures). This process is useful in technology for tasks like creating new images, music, or any digital content that mimics the original data’s style. As shown in the following diagram, a handwritten image is encoded and decoded using VAEs:

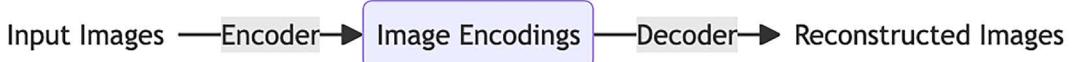


Figure 14.3: Image reconstruction flow using VAEs

The preceding diagram depicts the process flow of image reconstruction using VAEs. Here's an explanation of how VAEs typically work for this task with an example of reconstructing a person's face:

- **Input images:** These are the original images that you feed into the VAE system. The goal is to be able to reconstruct these images after they've been encoded and decoded. Let's say we have a set of face photographs. Each image is a clear, high-resolution photo of a person's face.
- **Encoder:** The encoder part of a VAE takes the input images and compresses them into a smaller, more compact representation known as the latent space or image encodings. This process involves learning the essential features and patterns present in the input images. The encoder analyzes input photographs and compresses each one into a smaller set of numbers that describe the key features of the faces, like the shape of the eyes, nose, and mouth. Imagine it like creating a unique code that could represent a face in much less space than the original picture.
- **Image encodings:** At this stage, the encoder has translated the input images into a set of encodings that represent the key features of the images in a much-reduced dimensionality compared to the original images. In the context of VAEs, these encodings also capture the probability distribution of the input data. These sets of numbers (encodings) are the essence of the photographs, stored in a compact form, which we can think of as the detailed features of the images. In the case of faces, these features might capture variations in facial features among different individuals.
- **Decoder:** The decoder takes these encodings and attempts to reconstruct the original images. It uses the compressed data to generate images that are as close as possible to the original input images. The decoder acts like an artist given the task of drawing a person's face. It takes these numerical codes and uses them to recreate the photographs of the faces. It tries to draw each face as accurately as possible, just from this compact code.
- **Reconstructed images:** The final output of a VAE. These are the images that have been reconstructed by the decoder from the image encodings. The quality of these images is dependent on how well the VAE has learned to compress and reconstruct the data. The result is a series of new face photographs generated by the VAE. These reconstructed images should closely resemble the original input photos. If you were to compare them side by side with the originals, you would find them similar, though they might be slightly blurry or have minor differences due to the loss of detail during the compression process.

In essence, the flow describes a VAE's ability to learn efficient representations of data and generate new data that resembles the original input. This process is used in various applications, including image denoising, inpainting, and as a generative model to create new images that share properties with the training dataset.

Transformer-based generative models

These models, such as GPT-4, are built upon the transformer architecture, which excels in understanding and generating data sequences, such as text. They learn patterns in language and context, allowing them to generate coherent and contextually relevant text. The following diagram shows the working of the transformer in the model:

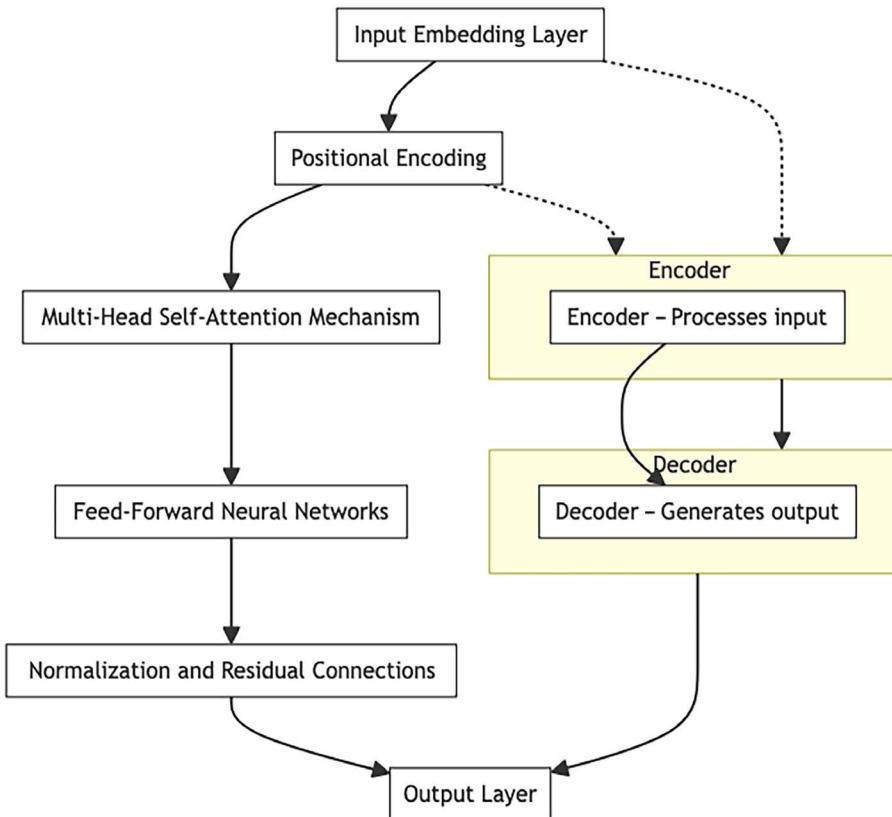


Figure 14.4: Component of transformer-based generative models

The preceding diagram shows the workflow of a transformer model, which is an advanced type of neural network used in **Natural Language Processing (NLP)** tasks such as translation, text generation, and more. Let's look at each step with the example of language translation:

- **Input embedding layer:** The process begins with the input embedding layer, where individual elements (like words in a sentence) are transformed into numerical vectors that the model can process. For example, the sentence “How are you?” enters the model, and each word is turned into a numerical vector. This is like giving each word a unique, identifiable badge number so the model can understand and manipulate them.

- **Positional encoding:** Positional encoding is added to these vectors to give the model information about the position of each word within the sentence since transformers do not inherently understand the order of words. For instance, along with the badge number, each word is given a position tag. “How” is tagged as the first word, “are” as the second, and “you” as the third. This helps the model consider the order of words.
- **Encoder:** The combined embeddings (input embeddings plus positional encoding) are then fed into the encoder. The encoder processes the input data, capturing the context of each word relative to the others in the sequence. It’s like the encoder reads the sentence and understands the meaning of each word in the context of the entire sentence. The encoder reviews the words with their badge numbers and position tags to understand the sentence’s meaning. For example, it notes that “How” in the first position usually starts a question.
- **Multi-head self-attention mechanism:** Within the encoder, the multi-head self-attention mechanism allows the model to weigh the influence of different parts of the input differently. It’s as if the model is considering different aspects of the meaning of a word by looking at the other words around it. The encoder pays special attention to how each word in the sentence relates to every other word. It notices, for example, that “How” is connected to “you” to form a polite inquiry about someone’s well-being.
- **Feed-forward neural networks:** Next, the processed information passes through feed-forward neural networks, which further process the data sequentially in each layer to refine and abstract the representation. These networks refine the information from the attention mechanism, almost like a group of editors polishing a draft to better convey a sentence’s intent.
- **Normalization and residual connections:** Along the way, normalization and residual connections are applied to help maintain data flow and mitigate the risk of data transformation errors in deeper layers of the network. These elements ensure that the information flowing through the model is neither too dampened nor too amplified. To prevent errors from growing through the network layers, these components act like checkpoints that keep the data on the right track.
- **Decoder:** After the encoder has processed the input, the decoder uses this information to generate the output. It receives the processed data from the encoder and starts producing the transformed sequence, such as translating the sentence into another language or generating a response in a dialogue. The decoder takes the processed information from the encoder and begins generating the output. If it’s translating, it will start producing the translated sentence.
- **Output layer:** The final output of the decoder is sent to the output layer, which translates the advanced neural network output back into a readable format, like a sentence in human language. This is where the final output begins to take shape. If the model is translating the sentence, this layer starts building the translation based on all the processed information.

The transformer model reads and understands the input data (sentence, paragraph, etc.), processes it to understand the context, and generates a relevant output based on that understanding. For “How are you?”, the encoder part of the model processes the question, while the decoder generates a response or a translation, one word at a time, considering both the information from the encoder and what it has already generated.

Think of the encoder as the part that looks at the input information and the decoder as the part that creates the output. For example, GPT-4 is based on the transformer model. When you give it a starting point, it can generate text that makes sense and fits the context.

This model uses “self-attention” to determine which words in the starting point are important and how they connect. This way, it can really understand what you’re asking for and give you a good response.

Other important generative models

Apart from the types mentioned, there are other notable generative models:

- **PixelCNN and PixelRNN:** These models generate images pixel by pixel, capturing intricate details and dependencies within the image. Imagine drawing a picture pixel by pixel, ensuring each pixel fits with the ones around it.
- **Flow-based models:** These models learn how to transform one data distribution into another, allowing them to generate samples matching the desired distribution. This is a recipe that turns simple ingredients into a fancy dish following specific instructions.

These generative models have strengths and applications, making them suitable for a large range of tasks, from image generation to text creation. Their diverse capabilities contribute to the rich landscape of generative AI.

Importance of hyperparameter tuning and regularization in architectures

Hyperparameter tuning and regularization are fine-tuning and safety measures for generative AI architectures. For example, in image generation, you might adjust hyperparameters such as the learning rate, which determines how fast the model learns; if it’s too high, the model might learn the wrong patterns, like someone learning to play a song on the piano but pressing the keys too hard or too softly. Regularization might involve techniques like dropout, where you randomly ignore some of the model’s neurons during training to make the model robust, much like training a football team with some players sitting out so the team doesn’t rely too much on any one player. They play a crucial role in making these systems work well and create high-quality content. Let’s understand their importance.

Hyperparameter tuning

Think of hyperparameters as knobs and switches that control how the generative AI system learns and creates. They affect things like the speed of learning, the level of detail in the output, and the balance between creativity and accuracy.

Imagine trying to find the perfect oven temperature for baking a cake. Too hot and it burns; too cold and it stays gooey. Hyperparameter tuning is similar. It helps adjust the parameters so that the AI system learns in the best way, creating content that’s just right.

For example, hyperparameters might control the length of melodies, the tempo, or the instruments used in a music generation system. Tuning them ensures that the music sounds harmonious and matches the desired style.

Regularization

Regularization is like adding safety nets to a tightrope walker. It prevents the AI system from getting too carried away and creating too wild or unrealistic content. It's a way to keep the output in check and ensure it is well-behaved.

In a generative AI system, regularization helps prevent overfitting. Overfitting is when the system becomes too good at mimicking the training data but needs help with new, unseen data. Regularization techniques simulate adding minor penalties to certain parts of the learning process, helping the system generalize better and create more diverse and creative content.

For instance, regularization in an image generation system might ensure that the generated images have consistent colors and shapes, preventing them from looking too noisy or strange.

Hyperparameter tuning and regularization matter as they fine-tune the generative AI system's performance and ensure it produces high-quality, consistent, and realistic content. Without them, the system might either create content that's too boring or content that's too chaotic and nonsensical.

Just like a chef adjusts the cooking time and adds the right spices to make a perfect dish, hyperparameter tuning and regularization fine-tune the generative AI system to create content that's creative and aligned with the desired output. They ensure the system stays on the right track, creating exciting and reliable content.

Popular generative AI FMs

The field of generative AI is rapidly evolving, with various organizations pushing the boundaries and launching powerful foundation models to drive innovation. The launch of models like ChatGPT has undoubtedly contributed to the acceleration of this trend. Both established tech giants and emerging start-ups are actively participating in the generative AI boom, aiming to develop more sophisticated and capable FMs. Here's a list of some of the most popular FMs in generative AI:

- **Amazon:** Amazon Web Services (AWS) is one of the top cloud providers and has a large set of offerings in ML and generative AI. AWS has launched a generative AI service called AWS Bedrock with accessibility to popular FM models using APIs in a serverless manner. Amazon SageMaker JumpStart is another offering that provides access to a wide range of FM models and the ability to tune them as needed. Amazon Titan is AWS's flagship generative AI model. Amazon's Titan suite encompasses a series of FMs that cater to a variety of generative tasks, including:
 - **Titan Text Embeddings:** For contextual text representations
 - **Titan Multimodal Embeddings:** For interpreting data across text and images
 - **Titan Text Lite:** For efficient text processing in resource-constrained environments
 - **Titan Text Express:** For rapid text processing tasks
 - **Titan Image Generator:** For creating or modifying visual content from textual inputs

You can learn about Amazon title models and keep an eye on upcoming developments by visiting the Amazon Bedrock page here: <https://aws.amazon.com/bedrock/titan/>.

- **OpenAI:** OpenAI is a research organization that creates and promotes open and ethical AI. It has created several generative AI models, such as:
 - **DistilGPT2:** Efficient text generation model
 - **GPT-3:** Versatile model for text generation and question-answering
 - **GPT NeoXT:** Advanced model for diverse language tasks
 - **GPT-3.5:** Generates longer, coherent texts with efficiency
 - **GPT-4:** Multimodal model with human-like performance
 - **CLIP:** Learns relationships between text and images
 - **CLIP-Guided Diffusion:** Creates images aligned with text prompts
 - **DALL·E:** Generates images from natural language prompts
 - **MuZero:** Learns to play games through self-play
 - **Text-to-Speech (TTS):** Converts text to natural-sounding speech
 - **Whisper:** Audio-to-text transcription model
 - **Embeddings:** Converts text into numerical data
 - **Moderation:** Assesses text for sensitive content
 - **Sora:** Generates videos from written prompts

OpenAI is working on GPT-5, which is their latest and most advanced model in training. To learn more about OpenAI's models, you can visit their official website here: <https://openai.com/>. OpenAI provides detailed information about their models, research, publications, and API access on their platform.

- **Google:** Google is a pioneer in AI and ML. It has developed several generative AI models, such as:
 - **Google Gemini:** A large language model for language translation, content creation, and query answering
 - **BERT:** A model that improved contextual understanding in language processing
 - **BigGAN:** Generates high-resolution, realistic images for visual content creation
 - **Text-to-Text Transfer Transformer (T5):** Automates content generation for various NLP tasks
 - **Flan T-5 models:** Tailored for specific language processing tasks including text and code
 - **Pathway Language Model (PaLM):** Among the largest language models, excelling in text generation and translation
 - **LaMDA:** Designed for dialogue applications, mimicking human conversation
 - **Falcon-7B and Falcon-40B:** Models designed for language translation, question-answering, and text generation
 - **Chinchilla by DeepMind:** A massive language model focused on text generation and language translation tasks

Google is now focusing on Gemini and building a more advanced version of it extending to a subscription model. You can learn more about Google's AI models and research by visiting the Google AI website (<https://ai.google/>) or the DeepMind website (<https://deepmind.com/>).

- **Anthropic:** Anthropic is a research organization that aims to create general and scalable AI that can align with human values and preferences. It has received significant investments from various large tech companies, including Amazon (which invested \$5 billion) and Google (which invested \$2 billion). It has developed a generative AI model family called Claude, which includes the following models:
 - **Claude:** An FM offering advanced language understanding and generation capabilities
 - **Claude 2:** An enhanced version of Claude with improved language processing abilities and context understanding
 - **Claude 2.1:** A further refined version offering more nuanced language generation and comprehension
 - **Claude Instant:** Designed for speed, delivering fast responses while maintaining effective language understanding
 - **Claude 3:** The latest model family, which sets new industry benchmarks across various cognitive tasks and comes in three variations – Haiku, Sonnet, and Opus.

Anthropic made these models available via different platforms, such as Amazon Bedrock and Google Vertex AI, in addition to their own Claude AI web chat interface. For the most current and comprehensive list of models, please visit Anthropic's website directly at <https://www.anthropic.com/clause>.

- **Meta (Facebook) AI:** Meta AI is a research organization that develops and applies AI for various products and services related to social media, communication, content creation, and more. It has developed generative AI models such as:
 - **RoBERTa:** An enhanced BERT model that achieves better performance through more extensive training and fine-tuning
 - **DETR:** Simplifies object detection in images by combining convolutional neural networks with the transformer architecture
 - **Llama:** A range of language models designed for understanding and generating human-like text, available in various sizes to suit different computational and application needs
 - **BlenderBot:** Conversational AI that can engage in meaningful and coherent interactions, simulating human-like dialogue
 - **Faiss:** A library for efficient similarity search and clustering, ideal for handling large datasets and complex similarity tasks

You can learn about the latest developments in the area of generative AI from Meta by visiting their website at <https://ai.meta.com/>.

- **Microsoft:** Microsoft uses OpenAI offerings extensively and has made a \$10 billion investment and offers the **OpenAI Model as a Service (MaaS)**. However, it has also developed generative AI models such as **Turing-NLG** and **MPT-7B**. Under Microsoft's MaaS model, it offers OpenAI models such as GPT4, GPT3.5, DALL-E, and Whisper. You can learn about the Microsoft Azure model catalog by visiting their generative AI offering page here: <https://azure.microsoft.com/en-us/products/machine-learning/generative-ai>.

- **AI21 Labs:** AI21 Labs is a research organization focusing on natural language understanding and generation. It has created several generative AI models, such as **Deep Extension of Latent Logic (DELL)**, **Jurassic-1**, and **Jurassic-2**. It launched AI21 Studio to democratize access to its models and also partnered with Amazon to make them available through Amazon Bedrock. You can learn about the latest offerings of AI21 by visiting its official blog website here: <https://www.ai21.com/blog>.
- **Nvidia:** Nvidia specializes in **graphics processing units (GPUs)**, gaming, cloud computing, AI, and more. It has created several generative AI models, such as **StyleGAN2** and **GANVerse3D**. You can learn more about Nvidia models here: <https://www.nvidia.com/en-us/ai-data-science/generative-ai/>.
- **Jasper.ai:** Jasper.ai is a technology company that provides generative AI solutions for marketers. It has developed a generative AI model called Jasper. They launched Jasper AI Copilot to extend their offering. You can learn more about Jasper here: <https://www.jasper.ai/>.
- **Hugging Face:** Hugging Face is a technology company that provides open-source tools and platforms for NLP. It has created several generative AI models, such as **Bloom models**, **BloomZ 176B**, **Lyra-Fr 10B**, and **Lyra-Mini**. To learn more about Hugging Face and its range of generative AI models, you can visit its official website and explore its Model Hub, where it provides detailed information and access to its models. Here's the link to get you started: <https://huggingface.co/docs/hub/en/models-the-hub>.

The above list is not complete but it talks about some of the most popular models. Massive developments are going on for FMs in generative AI. As research in this area continues, we can expect to see even more powerful and versatile models being developed in the future.

How to start with generative AI

Starting with generative AI involves selecting the right tools and platforms that suit your needs. Whether you're an end user looking to engage in AI-generated conversations or a developer/ML scientist aiming to create sophisticated applications, numerous resources are available from different providers to help you embark on your generative AI journey. Getting started with generative AI can be exciting! The following subsections provide a breakdown of how different types of users can begin their exploration into generative AI.

For end users

For individuals seeking to harness the capabilities of generative AI in their day-to-day activities such as content creation, marketing materials, email composition, and efficient learning, several accessible tools can be employed:

- **ChatGPT** offers a user-friendly chatbot experience driven by GPT-3.5, an advanced language model. This tool responds with natural language based on the input it receives, enabling engaging conversations on various topics. At the time of writing, ChatGPT can be accessed at <https://chat.openai.com> for free, with the option to upgrade to GPT-4 for more advanced features for a monthly subscription of \$20. You can also explore various purpose-built custom applications available in the GPT Store created by the builder community.

- **Claude** is a generative AI model developed by Anthropic. Claude specializes in generating text for emails, summaries, stories, and more. Its capabilities can be found at <https://claude.ai/chat/>, contributing to content creation while aligning with human values.
- **Google Gemini (formerly Bard)** is a chatbot offered by Google. Like ChatGPT, Gemini can answer your questions comprehensively and informally and generate different creative text formats, like poems, code, scripts, musical pieces, emails, letters, etc. You can explore its capabilities at <https://gemini.google.com/app>. Gemini is the successor of Google's first Q&A app, which was formerly known as Bard.
- **Copilot** provides a generative AI service by Microsoft, utilizing models like GPT-4. It facilitates conversations using natural language, making it easier to interact and communicate with an AI-driven system. This service is accessible at <https://www.bing.com/chat>, allowing users to engage in conversations seamlessly and intuitively.
- **Amazon Q**, a service offered by AWS, is designed to significantly enhance productivity and decision-making within organizations. It serves as an advanced tool that can quickly provide relevant answers to urgent queries, assist in problem-solving, generate content, and execute tasks by tapping into the wealth of knowledge contained within a company's databases, code bases, and enterprise systems. You can learn more about Amazon Q by visiting the AWS page here: <https://aws.amazon.com/q/>.
- **Perplexity AI** represents a breakthrough in search technology, functioning as an advanced AI-driven chat tool that goes beyond traditional search engines. Acting as a conversational search engine, Perplexity AI employs NLP and ML techniques to accurately respond to a wide array of questions. It offers users quick access to information across various subjects, simplifying the search process. Moreover, it invites users to delve deeper into topics of interest by asking follow-up questions or seeking additional details, thereby enriching the user's understanding and learning experience. You can explore it by visiting <https://www.perplexity.ai>.

There are many other AI apps available for different purposes from companies like Jasper, Midjourney, Canva, and Luminar. By leveraging these generative AI tools, individuals can streamline tasks, foster creativity, and enhance productivity in their daily endeavors, from crafting content to having interactive dialogues. Each tool brings its own set of unique features, making them versatile assets for simplifying and enhancing various aspects of daily life.

For builders

Builders such as app developers, data scientists, and ML engineers can use generative AI to increase their productivity multi-fold with the help of generating code, tuning already developed models, and accessing existing models through APIs. Let's learn more:

- **Increasing productivity through code generation:** Generative AI tools provide the ability to generate code so that you can focus on business logic rather than writing repeated code. Some of the most popular code-generation tools are:

- **Amazon CodeWhisperer:** AWS provides this service, which uses NLP and ML to generate code snippets based on natural language queries. For example, you can ask CodeWhisperer to create a Lambda function that sends an email using SES, and it will generate the following code for you:

```
#create a Lambda function that sends an email using SES
def lambda_handler(event, context):
    client = boto3.client('ses')
    response = client.send_email(
        Source='XXXXXXXXXXXXXXXXXXXXXX',
        Destination={
            'ToAddresses': [
                'XXXXXXXXXXXXXXXXXXXXXX',
            ],
        },
        Message={
            'Subject': {
                'Data': 'Hello from SES!',
            },
            'Body': {
                'Text': {
                    'Data': 'Hello from SES!',
                },
            },
        },
    )
    print(response)
    return response
```



CodeWhisperer works with 15+ programming languages, including Python, Java, and JavaScript, and popular **integrated development environments (IDEs)**, including VS Code, IntelliJ IDEA, AWS Cloud9, the AWS Lambda console, JupyterLab, and Amazon SageMaker Studio. You can learn more about Amazon CodeWhisperer here: <https://aws.amazon.com/pm/codewhisperer/>.

- **Azure Copilot:** This tool uses OpenAI Codex, a large language model trained on billions of lines of code, to generate code suggestions within VS Code. You can use Azure Copilot to write code in multiple languages, such as Python, JavaScript, TypeScript, and others. Here is an example of how Azure Copilot works:

```
# Suppose you want to write a function in JavaScript that takes an
array of numbers and returns the average
# You can start by typing the function name and parameters

function average(numbers) {

    # Then you can press Ctrl+Space to trigger Azure Copilot
    # suggestions
    # Azure Copilot will suggest the following code based on the
    context and common patterns

    // initialize the sum to zero
    let sum = 0;
    // loop through the array of numbers
    for (let number of numbers) {
        // add the number to the sum
        sum += number;
    }
    // calculate the average by dividing the sum by the length of
    the array
    let average = sum / numbers.length;
    // return the average
    return average;
}
```

- **ChatGPT interpreter:** This tool uses ChatGPT, a chatbot based on GPT-3, to generate code interactively based on natural language input. You can use the ChatGPT interpreter to write code in Python, Java, and C#. Here is an example of how the ChatGPT interpreter works:

```
User: Write a function in Python that takes a list of numbers and
returns the sum of the squares of the odd numbers.
Chatgpt: def sum_of_squares_of_odd_numbers(numbers):
    # initialize the sum to zero
    sum = 0
    # Loop through the list of numbers
    for number in numbers:
        # check if the number is odd
        if number % 2 == 1:
            # square the number and add it to the sum
            sum += number ** 2
    # return the sum
    return sum
```

- **Google Codey:** Codey is capable of supporting more than 20 programming languages, including popular ones like Python, Java, JavaScript, and Go. Codey's main objective is to significantly speed up the software development life cycle. It achieves this through real-time code completion and generation functionalities. Developers have the flexibility to tailor Codey to fit their specific code bases, enhancing its utility across a wide array of coding projects.

If you're interested in harnessing the power of generative AI FMs for your applications, you're in luck! Many of these models are easily accessible through APIs offered by renowned cloud platforms and organizations. Let's look at them in detail.

Using generative AI FMs in your applications with public cloud providers

Integrating generative AI FMs into your applications is now more accessible than ever, thanks to a range of cloud platforms offering APIs. Here's a closer look at some of these popular platforms and how you can utilize them:

- **AWS:** AWS introduced the general availability of Amazon Bedrock and Agents for Amazon Bedrock as part of their commitment to leading the cloud AI space by offering advanced AI solutions and partnerships with industry-leading FM providers. Amazon Q, a new generative AI-powered assistant tailored for professional use and trained on over 17 years of AWS expertise, exemplifies AWS's innovative approach to integrating AI into the workplace, promising to enhance productivity and creativity within enterprises.
- **Amazon Bedrock:** Amazon Bedrock is a robust cloud-based platform provided by AWS, designed for training, building, and deploying ML models. It offers an extensive suite of APIs for various tasks, including NLP, computer vision, and speech recognition. Bedrock provides access to FMs from Amazon and leading AI organizations such as AI21 Labs, Anthropic, Cohere, Meta, and Stability AI. To start developing a generative AI application using Amazon Bedrock, you'll first select an FM suitable for your needs. This can be done through the Amazon Bedrock Console or API. Once you've chosen an FM, you can seamlessly integrate it into your application using the Amazon Bedrock API. Examples of available FMs include Amazon Titan for text summarization, Jurassic-2 for instruction-following language models, and Claude 3 for thoughtful dialogue and content creation. You can start using Amazon Bedrock by using the link here: <https://aws.amazon.com/bedrock/>.
- **SageMaker JumpStart:** SageMaker JumpStart is another offering from AWS aimed at simplifying the ML development process. It provides pre-built ML models and workflows to accelerate your ML projects. SageMaker JumpStart offers APIs for various tasks, such as NLP, computer vision, and speech recognition. To start with generative AI applications through SageMaker JumpStart, you'll choose a pre-trained model that aligns with your project's requirements. Once selected, you can deploy this model within your application using the SageMaker JumpStart API. Available models include Hugging Face for NLP, ImageNet for image classification, and YOLOv5 for object detection. You can learn how to start with SageMaker JumpStart by using this link: <https://docs.aws.amazon.com/sagemaker/latest/dg/studio-jumpstart.html>.

- **Microsoft Azure:** Microsoft is proactively embedding generative AI technologies across its entire suite of solutions, including Azure, M365, Dynamics 365, Power Platform, Windows, and GitHub, showcasing the transformative power of generative AI on its product lines. For enterprise clients, Microsoft has launched its generative AI initiatives through Azure OpenAI Service, setting its offerings apart from those directly available from OpenAI by focusing on features such as private networking, top-tier security, scalability, and regional service availability.
- **Azure OpenAI:** Azure OpenAI is Microsoft's offering, providing access to various foundation models for NLP, computer vision, and speech recognition. You can utilize the Azure OpenAI API to access models like GPT-3 for NLP, DALL-E for image generation from textual descriptions, and Speech Services for speech recognition and synthesis tasks. Azure AI Studio includes a model catalog, similar to Amazon SageMaker JumpStart. Microsoft introduced MaaS in Azure AI with similar features to Amazon Bedrock with ready-to-use APIs, hosted fine-tuning, and integration tools. You can sign up for Azure OpenAI by following this link: <https://azure.microsoft.com/en-us/products/ai-services/openai-service>.
- **Google Cloud Platform (GCP):** GCP has integrated its generative AI capabilities into Vertex AI, showcasing its commitment to enhancing its suite of solutions. The cornerstone of its AI initiative is the PaLM 2 FM, which now supports over 25 Google products and is available to GCP customers through the PaLM API. Google has developed industry-specific FMs, including Med-PaLM for healthcare and Sec-PaLM for security applications, and has introduced a collection of AI assistants branded as Duet AI within Google Workspace and Google Cloud.

In a significant expansion of their FM portfolio, Google Cloud announced Gemini, its latest first-party FM. Gemini will be available in various configurations, including Ultra, Pro, and Nano, to cater to a broad spectrum of applications. Additionally, Google Cloud offers Model Garden and Generative AI Studio on Vertex AI, which facilitate access to both in-house and third-party models. Despite the wide range of models available, as of now, Google Cloud provides direct access only to its in-house PaLM 2 models via an API for hosted use, highlighting its strategy to blend proprietary technology with open innovation for generative AI solutions. Google Cloud is dividing its Duet AI product strategy into two offerings: Duet AI for Google Workspace and Duet AI for Google Cloud. Duet AI for Google Workspace will compete directly with Microsoft's M365 Copilot. Duet AI for Google Workspace is generally available and, curiously, is priced at exactly the same cost as M365 Copilot at the time of writing.

- **Google Cloud Generative AI:** Google Cloud Generative AI opens the door to Google's powerful generative pre-trained transformer models. You can leverage the Google Cloud Generative AI API to tap into models like DALL-E 2 for image generation, T5 for NLP tasks, and BigGAN for generating high-resolution images from simple natural language prompts. Get started with the Google AI service by referring to the link here: <https://cloud.google.com/ai/generative-ai>.

The following table shows the FMs available from major public cloud providers through APIs at the time of writing:

Public Cloud Providers	Available FM Providers	Available FMs
AWS	Amazon Anthropic AI21 Labs Cohere Meta Stability.ai	Titan Text Embeddings Titan Multimodel Embeddings Titan Text Lite Titan Text Express Titan Image Generator Jurassic-2 Ultra Jurassic-2 Mid Claude 2 Claude 2.1 Claude Instant Cohere Command Cohere Command Light Cohere Embed Llama 2 Llama 2 13B Llama 2 70B Stable Diffusion XL 1.0
Microsoft Azure	OpenAI Models as a Service Meta	GPT-4 GPT-4 Turbo, GPT-4 Vision, GPT-3.5 GTP-3.5 Turbo Embeddings models DALL-E Whisper Llama

GCP	Google	PaLM 2 Imagen Codey Embeddings
-----	--------	---

Table 14.1: FMs accessible through public cloud providers

While these platforms are among the top choices for GenAI application development, several other cloud providers offer similar capabilities, including IBM Cloud, Alibaba Cloud, and Tencent Cloud.

The optimal platform for your project will depend on your specific requirements, whether you need a serverless environment (Amazon Bedrock), a wide range of pre-trained models (SageMaker JumpStart), access to OpenAI's GPT-3 models (Azure OpenAI), or Google's LaMDA models (Google Cloud Generative AI). Each platform brings unique strengths to the table, empowering you to create generative AI applications catering to diverse use cases and industries. While you can access many FMs, choosing the suitable model is essential for your application's success. Let's learn how to choose the best FM for your needs.

Choosing the right FM

Choosing the right FM for your project is a crucial step in ensuring the success of your generative AI application. Here are some key factors to consider when selecting the most suitable FM:

- Problem identification:** Begin by clearly defining the problem you intend to solve with generative AI. Determine whether your project involves NLP, computer vision, speech recognition, or other tasks. This initial step helps narrow down your options to FMs designed for your specific domain. Suppose you are developing an agent assistant for a customer support application. Identify the problem as an NLP task focused on chat and call interactions. Look for FMs specifically designed for NLP tasks.
- Data consideration:** The nature and volume of your available data are critical. Some FMs require extensive datasets to train effectively, while others can work well with smaller or specialized datasets. Ensure that you have access to the appropriate data resources for training and evaluation. For your agent assistant app, get access to a large dataset of customer inquiries and responses. If the dataset is extensive and diverse, you can consider FMs that excel with ample data.
- Performance evaluation:** Once you've identified potential FMs that align with your problem and data, assess their performance on a validation dataset. This evaluation provides insight into how well each FM addresses your challenge and data characteristics. Look for FMs that demonstrate strong performance metrics for your use case. Suppose you shortlist GPT-3, GPT-4, and BERT as potential FMs for your application. Evaluate each FM's performance on a validation dataset by measuring metrics like response coherence, accuracy, and user satisfaction. Choose the FM that achieves the highest scores for your specific chatbot requirements. This ensures that your chatbot provides valuable and contextually relevant responses to customers.

4. **Fine-tuning:** Fine-tuning involves training the FM on your dataset to enhance its performance and align it with your problem domain. This process helps tailor the model to produce more accurate and relevant outputs. Let's say you decide to use GPT-4 as your FM but you notice it needs help understanding customer-specific jargon. Fine-tune GPT-4 on your customer support dataset to enhance its understanding of industry-specific terms and phrases. This adaptation ensures that your chatbot provides more accurate and relevant responses, improving the overall user experience.
5. **Iteration:** ML is inherently iterative. Be prepared to iterate on your model as needed. This may involve refining your dataset, adjusting hyperparameters, or experimenting with different FMs to achieve the desired level of performance. Continuous refinement is often necessary to optimize your generative AI application. Your agent assistance application is deployed but you receive user feedback about occasional irrelevant responses. Continuously iterate on your FM by refining the dataset, adjusting hyperparameters, and addressing specific issues raised by users. This iterative process helps maintain and enhance chatbot performance over time, ensuring ongoing customer satisfaction.
6. **Prompt engineering:** Prompt engineering is a technique where humans skillfully craft prompts to direct generative AI models, like chatbots or text generators, towards generating specific, desired outcomes. This **Human-in-the-loop (HITL)** approach is crucial because the phrasing of prompts can greatly affect the AI's responses, ensuring they are pertinent, precise, and context-sensitive. It plays an essential role in refining interactions with AI, customizing its responses to suit particular tasks or objectives effectively.

By following these expanded steps, you can effectively navigate the process of choosing, adapting, and optimizing the right FM for your generative AI project while considering real-world examples and use cases. As the FM trains with a very large dataset, it can get confused and provide inaccurate responses. Let's learn about how to prevent it in more detail.

Preventing model hallucinations

Model hallucinations, in the context of generative AI models, refers to situations where the model generates responses or outputs that are incorrect, irrelevant, or imaginary. These responses are often produced when the model encounters input that it cannot adequately comprehend or when it extrapolates beyond the scope of its training data. Model hallucinations can result in misleading, nonsensical, or potentially harmful information being presented to users. For instance, consider a medical diagnosis AI tool based on a generative model. If the model starts hallucinating, it might provide incorrect or irrelevant medical advice or diagnosis based on a patient's symptoms, leading to potentially dangerous health outcomes. Similarly, in the financial sector, an AI model used for predicting market trends might hallucinate and produce inaccurate forecasts, causing significant financial losses for users relying on its predictions. Addressing model hallucinations is crucial to ensure the trustworthiness and reliability of AI systems, especially in critical domains where decisions can have significant consequences.

To prevent model hallucinations and enhance the accuracy and reliability of generative AI models, several techniques and strategies can be employed, such as:

- **Data quality and quantity:** Ensure that the training data used for the model is of high quality, diverse, and representative of the intended application domain. Having a large and varied dataset helps the model understand a wide range of contexts and reduces the likelihood of hallucinations. A diverse dataset may include text from various domains, languages, and styles in NLP. When training a chatbot, a diverse dataset can help the model understand different user queries and provide contextually relevant responses.
- **Fine-tuning:** After the initial training on a large dataset, fine-tune the model on domain-specific data. Fine-tuning adapts the model to the specific task or knowledge domain, reducing the chances of generating hallucinatory content. An example is fine-tuning a pre-trained language model like Amazon Titan or GPT-4 on medical literature to create a medical chatbot that can answer questions, provide information, and assist healthcare professionals in understanding complex medical texts.
- **Prompt engineering:** Craft clear and contextually relevant prompts or input queries when interacting with the model. Well-structured prompts can guide the model to produce more accurate responses aligned with user expectations. Instead of a vague prompt like “Tell me about history,” a structured prompt could be “Summarize key events of World War II.” Content generation for educational materials benefits from clear and specific prompts to ensure accurate information delivery.
- **Retrieval-Augmented Generation (RAG):** Implement techniques like RAG to retrieve relevant information from a knowledge base or documents and use this retrieved context to generate responses. This approach helps in grounding the model’s output in factual and domain-specific information, reducing hallucinations, retrieving relevant product details from a database, and using them to generate accurate product descriptions. E-commerce platforms can implement RAG to create detailed and factual product descriptions.
- **Threshold filtering:** Set a threshold for response quality or relevance. Only accept responses from the model that meet a certain level of confidence or relevance, and discard or re-prompt for responses that fall below this threshold—rejecting responses with a confidence score below 0.8 ensures high-quality answers. Customer support chatbots can use threshold filtering to provide reliable responses to user inquiries.
- **Human review:** Incorporate human review and moderation to filter out potentially hallucinatory responses. Human reviewers can assess and correct model outputs to ensure accuracy and safety. Content generation platforms can employ human review to maintain content quality and prevent misinformation.
- **Continual monitoring and feedback:** Monitor the model’s performance and gather user feedback. Use this feedback to identify and address instances of model hallucinations and improve the model over time. Gather user feedback on chatbot interactions and analyze it for improvements. Chatbots and virtual assistants can continually evolve based on user feedback to provide better assistance.
- **Safety measures:** Implement safety measures and constraints within the model to prevent it from generating harmful, biased, or inappropriate content and implement profanity filters in chatbots to block offensive language. Online communities and social platforms employ safety measures to maintain a respectful and safe environment.

- **Domain limitations:** Clearly define and communicate the scope and limitations of the model to users. This helps manage user expectations and reduces the chances of the model providing hallucinatory responses when faced with out-of-scope queries. It informs users that a weather chatbot can't provide medical advice. Specialized chatbots, such as weather or travel assistants, benefit from setting clear boundaries to avoid misleading users.
- **Regular updates and maintenance:** Keep the model up to date with the latest data and advancements in the field of AI. Regular updates and maintenance can improve its overall performance and reduce the occurrence of hallucinations. Update a language model with the latest vocabulary and language trends. News agencies use updated language models to generate real-time news summaries.

By combining these strategies, developers and organizations can minimize the risk of model hallucinations and create more reliable and trustworthy generative AI systems.

Enterprises are strategically deploying generative AI applications internally as a prudent initial step before wider external deployment. This approach serves as a buffer, reducing the risk of delivering incorrect or inappropriate content to customers. This internal deployment allows organizations to refine the AI models in a controlled environment, where the consequences of errors are less severe and can be quickly addressed. Employees, who are familiar with the business context and nuances, can provide valuable feedback on the model's outputs, identifying any irrelevant or incorrect responses generated by the AI.

For instance, a company could deploy an AI-powered chatbot internally to assist with IT support or HR inquiries. As employees interact with the chatbot, they can report any anomalies or instances where the bot provides bizarre or inaccurate answers. This feedback loop enables the company to fine-tune the AI model, improving its accuracy and relevance before rolling it out to customer-facing scenarios.

Such a phased approach not only mitigates the risks associated with AI hallucinations but also builds confidence in the technology, both within the organization and among its customers. By the time the generative AI application is introduced to external users, it has undergone extensive validation and refinement, ensuring a more reliable and effective user experience.

Let's learn about a reference architecture that will help you to build a generative AI -based application.

Generative AI reference architecture for building a mortgage assistant app

The homebuying process can often appear daunting to prospective buyers, primarily due to the overwhelming amount of paperwork involved. Frequently, buyers find themselves needing more time and an in-depth understanding of the intricate details within these documents. Consequently, they experience feelings of being overwhelmed, confused, and occasionally frustrated as they grapple with grasping the significance of what they are signing, particularly in the context of mortgage-related paperwork.

Addressing these challenges becomes paramount in enhancing the overall customer experience and building trust between buyers and lenders throughout the loan application and closing process. To alleviate this burden and empower homebuyers, a generative AI solution can assist them in comprehending their loan terms and conditions better without relying on mortgage experts or attorneys.

This section delves into the architecture of constructing a virtual home lending agent application. At its core, this application revolves around the text summarization of critical mortgage documents. The architecture, as presented, adopts a serverless approach using AWS and harnesses Amazon Bedrock to access the generative AI FM. It's worth noting that you can implement this architecture with the technology you choose.

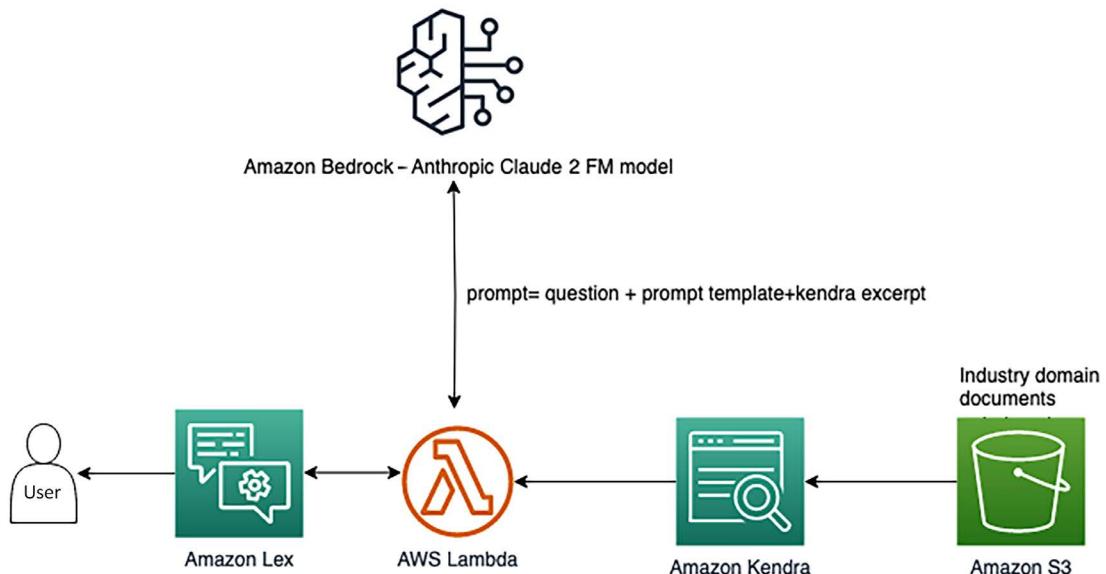


Figure 14.5: Generative AI-based home lending virtual assistant app

We devised an Amazon Lex-based **Virtual Assistant (VA)** application within the preceding architecture diagram. Amazon Lex is a service provided by AWS to build a serverless chatbot. This VA is an intuitive interface where users can interact and seek specific answers regarding their mortgage documents. Leveraging natural language understanding and processing capabilities, the VA interprets users' questions and prompts, subsequently gaining access to domain-specific document excerpts indexed by Amazon Kendra.

Amazon Kendra's intelligent search capabilities are pivotal in efficiently retrieving relevant document excerpts. These carefully curated excerpts are then transmitted to the Amazon Bedrock FM Claude 2, generating informative and coherent responses to the users' queries. In this manner, the generative AI solution simplifies the comprehension of complex mortgage documents. It elevates the overall homebuying experience by providing buyers with a reliable and knowledgeable resource throughout the process. This, ultimately, culminates in heightened customer satisfaction and greater trust in the lending institution.

Implementing the generative AI solution places significant emphasis on ensuring that users exclusively receive responses that strictly adhere to the scope of the documents utilized. To preempt any instances of model hallucinations or inaccurate responses, the architecture utilizes RAG. An instrumental facet of the RAG-based approach is integrating the company's distinctive knowledge base or content as an integral context component. This knowledge base is seamlessly integrated with the user's request, thereby forming a comprehensive prompt. This consolidated prompt is subsequently relayed to the Amazon Bedrock FM, generating highly accurate and concise summaries as responses.

Harnessing the capabilities of RAG while seamlessly incorporating domain-specific knowledge ensures that users consistently receive responses that are not only contextually relevant but also exceptionally precise. This precision serves as a linchpin in delivering a superior user experience and bolstering the trustworthiness of the generative AI application throughout the mortgage document review process.

This solution streamlines and simplifies the content within these documents, ensuring that home buyers can rapidly access the pertinent information they require. This not only saves time but also aids in significantly reducing confusion.

Challenges in implementing generative AI

Implementing generative AI, while highly promising, comes with its set of challenges and considerations. In the following subsections, we delve into some of the primary challenges associated with generative AI.

Training stability issues

One of the significant challenges encountered in generative AI is training stability issues. These issues can manifest as convergence problems, slow training, or even divergence, making it difficult to obtain high-quality generative models.

One prevalent application of generative AI involves using a GAN to create high-definition images. Training stability issues may arise during the training of a GAN for image generation. For instance, the generator may produce nonsensical or highly distorted images. These issues can hinder the GAN from converging to a satisfactory solution, leading to poor image generation quality.

Addressing and preventing training stability issues in generative AI involves a multifaceted approach. Techniques such as spectral normalization and progressive growing can stabilize training processes. Proper weight initialization and careful monitoring of loss curves help prevent issues like mode collapse and slow convergence. Additionally, the use of batch normalization and regularization techniques can contribute to training stability. By combining these strategies and fine-tuning hyperparameters, developers can enhance the stability of generative AI models, ensuring more reliable and robust performance across various applications.

Mode collapse

Generative AI has made remarkable strides in creating human-like content across various domains. It does, however, face difficulties such as mode collapse, which has an impact on the variety and caliber of outputs produced. When a generative model generates few or repetitive outputs due to its inability to fully capture the diversity of the target distribution, mode collapse occurs. The model becomes fixated on a subset of possible data, neglecting the broader spectrum of variations within the dataset.

Imagine a scenario where you're training a text generation model to produce diverse content, such as generating news articles. In this context, mode collapse could manifest as the model repeatedly generating the same headline or content, failing to explore the vast array of possible articles.

Suppose you're using a text generator to create news articles. Despite having access to a diverse dataset of news topics, the model consistently generates headlines with only a few variations, presenting the same news story repeatedly, greatly diminishing the quality and utility of the generated content.

By incorporating diversity-promoting objectives, fine-tuning hyperparameters, and introducing randomness, developers can effectively address and prevent mode collapse, enhancing generative AI's utility and richness across numerous applications.

Latent space interpolation challenges

Latent space interpolation is a fascinating aspect of generative AI that allows models to generate intermediate outputs between two points in the latent space. However, it comes with challenges related to the meaningfulness and quality of the generated outputs. The core challenge with latent space interpolation is that generating outputs between points in the latent space may only sometimes result in semantically meaningful or coherent content. In essence, the generated transitions may need more continuity and relevance.

Consider a scenario where you aim to create a generative model that produces images with smooth transitions between different artistic styles—for example, transitioning an image from the style of Van Gogh to the style of Picasso while maintaining visual coherence. When interpolating between two images of distinct styles, the resulting transitions might appear blurred, distorted, or semantically incoherent. This diminishes the artistic quality and intended smoothness of the style transitions.

To make generative models work better, especially when creating intermediate results, developers use specific techniques to improve how these models learn and create outputs. Here's a simple breakdown:

- **Disentangled representation learning:** This method helps the model learn features in a way that separates them clearly. For example, if the model is learning about faces, it learns to distinguish features like age, hairstyle, or glasses independently. This clarity helps the model generate more accurate and realistic transitions or changes.
- **Fine-tuning interpolation methods:** Interpolation is like filling in the gaps between two known points. In the context of generative models, fine-tuning these methods means making the steps or transitions between one output and another smoother and more logical. This helps avoid sudden, unrealistic changes when the model generates a series of images or sounds.

- **Leveraging semi-supervised learning:** This technique uses a mix of labeled (known) and unlabeled (unknown) data during training. It helps the model make better guesses about the unlabeled data by learning from the patterns it saw in the labeled data. This approach can improve how the model fills in gaps or transitions, making the output more coherent and realistic.

By using these techniques, developers ensure that when a generative model produces a series of outputs (like frames in a video or steps in a transformation), the results change smoothly and make sense, enhancing the overall quality and usefulness of the model in creative tasks.

Ethical concerns and misuse

Ethical concerns and the potential misuse of generative AI technologies have emerged as critical challenges in today's digital landscape. Generative AI's remarkable content creation and manipulation capabilities can be exploited for malicious purposes, such as creating deepfake videos, spreading misinformation, or generating offensive content. These activities raise serious ethical concerns and pose threats to trust and integrity.

Consider a scenario where generative AI creates deepfake videos that impersonate individuals, including public figures, celebrities, or politicians. Malicious actors may create deepfake videos featuring politicians making false statements or engaging in inappropriate behavior. These videos can weaponize disinformation, manipulate public opinion, or damage reputations.

Addressing ethical concerns and preventing misuse in generative AI is critical to maintaining trust and integrity in the digital age. By combining strict content moderation, authentication measures, responsible AI guidelines, regulatory frameworks, counter-forensic techniques, and public education, we can safeguard against the malicious misuse of generative AI technologies. This collective effort ensures that generative AI benefits society responsibly and ethically.

These challenges highlight the complexities of working with generative AI. Addressing them involves a combination of technical enhancements, ethical considerations, and regulatory measures. Additionally, continuous research and development in AI are essential to mitigate these challenges and ensure the responsible and beneficial use of generative models.

Summary

In this chapter, we delved into the fascinating world of generative AI, starting with a comprehensive exploration of what it is. We explored the diverse use cases that generative AI enables, from transforming customer experiences to enhancing employee productivity and optimizing various aspects of business operations.

To understand the basic architecture of generative AI systems, we broke down the different types of generative models, including GANs, VAEs, and transformer-based models. We highlighted the importance of hyperparameter tuning and regularization in constructing effective generative AI architectures.

In the context of FMs, we provided insights into some of the popular generative AI FMs offered by key players in the field, such as Amazon, OpenAI, Google, Nvidia, and several others. These models serve as the backbone of generative AI applications.

For those eager to start their journey with generative AI, we offered guidance tailored to different user groups. End users can explore generative AI through chatbots, builders can leverage it to generate code, and developers can integrate generative AI FMs into their applications.

We also discussed the critical aspect of selecting the right FM for your project, ensuring it aligns with the specific requirements and data characteristics. To maintain model accuracy and prevent model hallucinations, we explored strategies to guide your generative AI endeavors.

Furthermore, we introduced a reference architecture for building a mortgage assistant app using generative AI. This architecture streamlines the understanding of complex mortgage documents, enhancing the overall homebuying experience.

Lastly, we examined the challenges of implementing generative AI, including training stability issues, mode collapse, latent space interpolation challenges, and ethical concerns related to potential misuse.

Throughout this chapter, we have laid the groundwork for a comprehensive understanding of generative AI, its applications, models, and challenges, setting the stage for further exploration and practical implementation in the world of artificial intelligence.

Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below to get a free eBook of your choice.



15

Rearchitecting Legacy Systems

Today's organizations are operating in a challenging environment. The pace of change is unprecedented. Regulators and institutions are imposing new reporting and security requirements, new technologies are disrupting consumers' expectations and perceptions, and the ecosystem is constantly evolving as new players enter the market. As a result, organizations are redefining their business models to provide the customer focus, agility, and technology they need to attract talent, be competitive, and grow.

Application modernization has become a critical component of these new business models to set up dev/test environments rapidly, experiment with new ideas, and develop new products and services. In addition to eliminating the need to invest in expensive and cumbersome infrastructure, a new system enables innovation through a broad set of available technologies.

Legacy systems are applications that have been deployed in your data center for decades without undergoing many changes. These systems are outdated and challenging to maintain in a fast-changing technological environment. Legacy systems are defined by age and the inability to meet growing business needs, due to the underlying architecture and technology.

Often, large enterprises deal with legacy applications to run crucial day-to-day business tasks. These legacy systems are spread across many industries, such as healthcare, finance, transportation, manufacturing, and supply chain. Often, companies have to spend heavily on the maintenance and support of these systems, which warrants the need to architect legacy systems. Rearchitecting and modernizing legacy applications helps organizations be more agile and innovative and optimizes cost and performance.

In this chapter, you will learn about challenges and issues with legacy applications and techniques to rearchitect them. Rewriting complex legacy applications may pose an additional risk of business disruption, so you will learn about refactoring applications or considering the option to migrate into a more flexible infrastructure. The following topics will be covered in this chapter:

- Learning the challenges of legacy systems
- Defining a strategy for system modernization
- Looking at legacy system modernization techniques

- Defining a cloud migration strategy for legacy systems
- Mainframe migration to the public cloud
- Modernizing legacy code with generative AI

By the end of the chapter, you will have learned about the various challenges and modernization drivers for legacy systems. You will have learned about the various strategies and techniques for the modernization of legacy systems. As the public cloud is becoming a go-to strategy for many organizations, you will also have learned about the cloud migration of legacy systems.

Learning the challenges of legacy systems

A legacy application presents significant challenges for an organization. On the one hand, there are critical applications that an organization has used for decades. On the other hand, legacy applications hold back the organization's pace of innovation.

End users are looking for the most modern, technologically advanced applications in a hugely competitive environment. All new features usually come with the latest software, and legacy applications limit your ability to add those features that benefit end users.

The following diagram shows some significant challenges that organizations face with legacy systems:

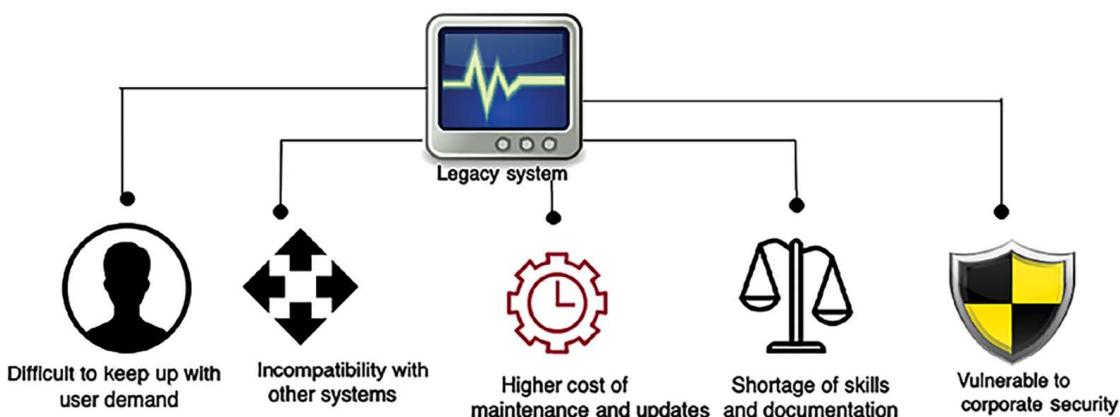


Figure 15.1: Challenges with a legacy system

Before we dive into the solution, it's essential to understand the issues clearly. Let's explore the challenges of legacy systems in more depth to understand them better.

Difficulty in keeping up with user demand

Customer focus is the key to business success, and being unable to keep up with the latest technology trends can harm a business significantly. You can take the example of Nokia, which used to lead the global mobile phone market. As smartphones came into play nearly a decade ago, Nokia still stuck with a legacy system, resulting in near bankruptcy. It was a similar story with Kodak—one of the largest businesses in the camera industry. Kodak could not move with digital innovation and adopt it into its systems, which resulted in Kodak becoming bankrupt in 2012. There are many such examples of large enterprises being unable to survive due to a lack of legacy modernization and innovation.

Users are very demanding in the current climate of fast-changing technology and fierce competition. Now, organizations have to change as per the user's terms, as they have multiple choices. As technology moves, the user moves with it and starts using the most recent and popular applications. Your competitors can jump ahead if they provide new features that the user needs. A recent example is Google, a pioneer in AI/ML that may have developed **Generative AI (GenAI)** technology much earlier. However, OpenAI moved swiftly to launch ChatGPT, putting Google on the back foot and forcing it into a catch-up game, consequently losing ground to competitors in the GenAI market. These examples underline the importance of adopting emerging technologies to maintain a competitive advantage.

A legacy system also challenges enterprise applications with an internal user base. An old system built on mainframes primarily uses the command line, which could be more user-friendly in the digital age. In contrast, new-generation workers demand a more user-friendly approach to perform their routine tasks. However, you may need more support from management, who may have been working with legacy systems for decades and are used to them.

The technology at the core of large enterprises needs to be updated and comprises systems dating back decades. Organizations running their core systems on legacy, on-premises technology face severe challenges when enabling modern customer experiences. Many systems are the product of multiple mergers and acquisitions, resulting in fractured data siloes, excessive infrastructure costs, and slow development time. This creates inefficient processing and decision-making, a lack of business agility, poor customer responsiveness, and high maintenance costs. Under these conditions, it's challenging for IT to meet the modern needs of internal stakeholders and customers.

Higher cost of maintenance and updates

As legacy systems have been set up and working for decades, they may look less expensive. But over time, the total cost of ownership turns out to be higher, as support and updates for old systems are usually more costly.

Those updates are often unavailable out of the box, and lots of manual workarounds are required to maintain the systems. Most legacy systems are not automation-friendly, resulting in more human effort.

Legacy systems mostly have a large chunk of proprietary software, which results in significantly higher license fees. In addition to that, old software no longer receives support from providers, and buying additional support out of the life cycle could be very costly. On the other hand, modern systems mainly adopt open source technologies that drive the cost down. The operational outage from a legacy system can take more time and drive up operating expenses, due to years of technical debt and hard-to-debug code. People with the skill set to maintain legacy systems (such as DB2, COBOL, Fortran, Delphi, and Perl) are hard to find, significantly increasing hiring costs and system risk.

Legacy applications have been in operation for decades, and, over time, many changes will have been accommodated without unused code being removed, which amounts to lots of technical debt. Any initiative to reduce technical debt could be risky due to unknown impacts and dependencies. As a result, organizations are forced to invest in unnecessary code and system maintenance for fear of breaking a system by making any significant changes.

However, modernizing legacy systems may be costly due to unknown dependencies and outages. A careful **cost-benefit analysis (CBA)** needs to be considered, and the **return on investment (ROI)** needs to be determined when deciding to proceed with modernization. As stakeholders need to see the immediate benefit of modernization, procuring finances for legacy modernization can be challenging.

Shortage of skills and documentation

Legacy technologies (such as mainframes) have multiple complex components that depend on each other. They are extensive proprietary and costly servers that are not readily available if someone wants to develop skills for them independently. It is challenging to retain application development resources and even more challenging to hire people with hands-on experience in old technology and operating systems.

Often, legacy systems are decades old, and most of the workforce with the relevant skills to manage them has retired. Also, these systems may need documentation to record the years of work that went into them. Significant knowledge loss is possible as an old workforce rotates with a new workforce. A lack of knowledge makes it risky to change the system due to unknown dependencies. Any small feature requests are challenging to accommodate due to system complexity and skills shortages.

New cutting-edge technologies such as advanced analytics, ML, GenAI, and the **Internet of Things (IoT)** are built on new technology platforms. As new technologies are not well integrated with legacy systems, an organization may lose out to a competitor if it cannot use the full capabilities of emerging technologies. A modern system helps build an organization's brand as an innovative company where most of the new generation of the workforce wants to work. Development and training are an even more significant expense for legacy technologies.

Often, automation helps to reduce costs by reducing human effort. Many tools are available in modern systems to build automation—such as DevOps pipelines, code reviews, and automation testing—that a legacy system may not utilize, resulting in additional costs.

Vulnerability to corporate security issues

Security is a top priority for any organization and system. A legacy application on an old operating system (such as Windows XP or Windows 2008) is vulnerable to security issues due to a lack of vendor support. Software vendors continuously determine new security threats and release patches to accommodate them in the latest software versions. Any legacy software announced as **End of Life (EOL)** from a vendor doesn't get a new security patch, which leaves your application running in the old software version exposed to several security threats.

System health checks are often ignored for legacy applications, which makes them more vulnerable to security attacks. The skills gap makes providing continuous support and help difficult, which means systems need to be more secure. A single vulnerability can pose a high risk of exposing your application, database, and critical information to attackers.

In addition to a security vulnerability, legacy applications are complex to maintain due to compliance. As regulations keep changing over time to enforce tight security around data handling and usage, legacy systems require changes to adhere to local governance and compliance needs.

For example, compliance to the European Union's General Data Protection Regulation (GDPR) requires each system to enable users to request to delete their data. While modern systems may provide these features out of the box in an automated and self-service manner, this may need to be performed manually and become more complex in legacy systems.

Adhering to compliance needs can lead to more operation costs and time-consuming maintenance.

Incompatibility with other systems

In addition to end users, often systems need to integrate with each other. Those systems may be associated with different departments, clients, partners, or suppliers. The various systems need to exchange data in a standard format that evolves. Almost every few years, files and data format standards are changed to increase data exchange efficiency, and most systems need to change to adopt them. Hard-to-change legacy systems that stick to using an old format could result in system incompatibility and a system that your suppliers and partners may not want to use. The inability to accommodate standard needs adds significant business risk due to complex workarounds and lost productivity.

Adding a workaround for simple business needs may make a system more complex. Modern systems are built on a service-oriented architecture, making it easier to accommodate any new requirement by adding a new service independently. Old systems are often built with a monolithic architecture, and adding any new feature means you need to rebuild and test the entire system.

Modern architectures are API-oriented and can be easily integrated with other systems to offload heavy lifting. For example, a taxi-booking app uses Google Maps for Global Positioning System (GPS) navigation and Facebook or X for user authentication. A lack of APIs makes these integrations harder in a legacy system, resulting in complex custom code.

A legacy application can face a scalability issue as load increases from another dependent upstream system. Often, legacy applications are built with a monolithic architecture and are hardware-dependent. Scalability is a big challenge with a monolithic system, as it cannot scale horizontally due to hardware dependency and vertical scaling being limited to the maximum system capacity. Also, increased demand in one portion of the monolith requires the entire system to be scaled. Breaking monolithic applications into microservices can help with scaling challenges and keeping up with the load.

In addition to software maintenance, legacy applications are costly for hardware infrastructure as they run on a particular version. They spread across multiple databases with duplicate data and similar functionality. Due to their monolithic nature, it's hard to consolidate and use the flexibility of cloud-based infrastructures to save costs. Also, decentralized systems allow software teams to select software stacks based on the needs of the microservice, rather than having to conform to a singular technology stack for all system functionality, enabling the use of diverse technologies based on the needs of each microservice and/or the team supporting the microservice.

Let's look at some key advantages of system modernization.

Defining a strategy for system modernization

Often, a legacy system gets left out of an enterprise's overall digital strategy, and issues get addressed as needed. A reactive approach prevents organizations from executing overall system modernization and benefits.

If your legacy system has serious business challenges, such as security and compliance issues, or cannot address a business need, you can take a **big-bang approach**. In the big-bang method, you build a new system from scratch and shut down the old system. This approach is risky but addresses a business need that can be mitigated from the existing legacy system.

The other approach is a **phased approach**, where you upgrade one module at a time and keep running both the old and the new systems. A phased approach is less risky but takes a long time and may be more expensive, as you need to maintain both environments with increased network and infrastructure bandwidth.

Approaching your application portfolio, prioritizing specific applications, and having an overall plan are the first steps. As you use the cloud, you design a new operating model and end up with a combination of tools. You can use third-party tools to frame your needs and tool preferences. And finally, you can use a consulting partner to complete migration and modernization projects more successfully and quickly.

Taking any of these approaches can provide various benefits once the modernization of an application is completed.

Assessment of a legacy application

An organization may have multiple legacy systems, with tens of thousands to millions of code lines. In a modernization situation, a legacy system must align with the business strategy and the investment cost. Also, it is possible to reutilize some parts or completely write them from scratch, but the first step is to conduct the assessment and better understand the overall system.

In the assessment phase, the solutions architect needs to make the system easy to assess more quickly and make informed decisions. Assessments can be done in days and weeks, depending upon the size of your workload and its complexity. The following points are the primary areas that solutions architects need to focus on when conducting an assessment:

- **Technology assessment:** You need to understand the technology stack used by the existing system. If the current technology is outdated and lacks vendor support, you might need to replace it. Consider upgrading if a better version of the technology becomes available. Often, newer versions are backward-compatible, with minimal changes required.
- **Architecture assessment:** You need to understand the overall architecture to make it future-proof. There may be a case where you need to make a minor upgrade to the technology but the overall architecture needs to be more monolithic and scalable. It would be best to audit the architecture in relation to cost, scalability, availability, performance, and security. Significant architecture changes may be required to align the application with business needs.
- **Code and dependency assessment:** Legacy systems often have hundreds of thousands of lines of code in a monolithic setting. Various modules tied to each other make the system very complicated. Code that appears not to be used in one module might impact other modules if it is removed without due diligence. These code lines may have been written decades back and needed regular refactoring and review. Even if the technology and architecture look fine, you must determine if the code is upgradable and maintainable. You also need to understand whether UI-related upgrades are required to improve the user experience.

As a solutions architect, you want to determine the dependencies across various modules and code files. Modules may be tightly coupled, and you must define an approach to perform simultaneous upgrades when modernizing the overall architecture. During your assessment, you may find the following patterns:

- First, many customers realize they have many old apps that do not relate well to the future business model; they can be retired. For example, around 10–20% of the app portfolio can be retired.
- Second, thousands of SaaS vendors didn't exist 5–7 years ago; these can replace many on-premises apps. For example, most customers have landed on Salesforce as a CRM platform. This shift to SaaS shrinks down the operational portfolio managed by IT operations—it still presents security and identity work but has lower operation costs.

Then, decisions can be made to lift and shift and, during a move, replatform the operating system, database, or language to reduce costs, such as customers choosing to re-platform from Windows Server to Linux, and from Oracle to Postgres, to reduce database licensing costs. You learned about these techniques in *Chapter 3, Cloud Migration and Hybrid Cloud Architecture Design*.

If you choose to modernize, you should focus on modernizing apps that differentiate your business. Let's look into the modernization approach.

Defining the modernization approach

For stakeholders, there may be no immediate incentive for application modernization. You need to choose the most cost-effective method and deliver results faster.

The following diagram shows the modernization approach:

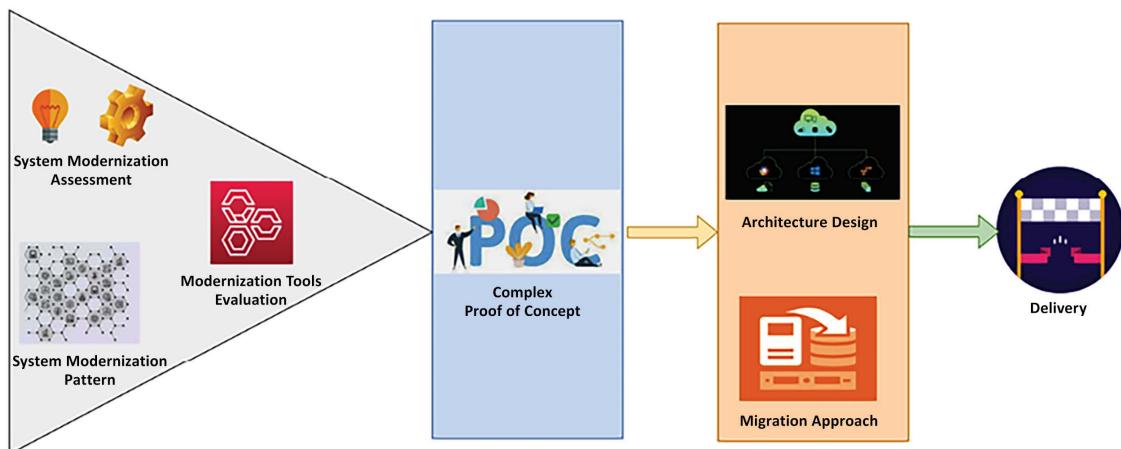


Figure 15.2: The legacy system modernization approach

After your system assessment, you must understand the existing architecture pattern and its limitations. As per your tech stack, you need to evaluate migration tools. For example, you may use an emulator for mainframe migration or vCenter if you rehost your application to VMware. Select various modernization approaches and create a **proof of concept (POC)** to identify gaps. Some approaches are listed here:

- **Architecture-driven modernization:** The architecture-driven approach is required to achieve the most agility. Often, an architectural approach is language-independent and platform-independent by applying service-oriented patterns, which gives the development team the flexibility to be more innovative. You may choose this approach if your assessment shows that you need significant architectural changes. Start implementing the most critical feature first and then build a POC highlighting the gaps and the required effort. Take the microservices approach to achieve scalability and ensure better integration with other systems, depending on your legacy application.
- **System re-engineering:** In the re-engineering approach, you must deeply understand the legacy system and perform reverse engineering to build a new modernized application. You need to be sure to make technological choices that help you create a future-proof system. You should take this approach if the legacy system is over-complicated and requires long-term projects. Start with application modernization first and upgrade the database as a final cutover in a phased approach. You need to build a mechanism where the legacy and upgraded modules co-exist, with the ability to communicate in a hybrid manner.
- **Migration and enhancements:** You can use migration and minor enhancement approaches if your existing system technology works relatively well but is restricted due to hardware limitations and cost. For example, you can lift and shift the entire workload to the cloud for better infrastructure availability and cost optimization. In addition, a cloud provider extends several out-of-the-box tools that help you make changes more frequently and apply better automation. A migration approach enables you to modernize your application with less effort and makes it future-proof, keeping it relevant for the long term. However, lift and shift are limited and may only be suitable for some workloads.

As you aim to migrate and modernize, consider specific IT domains that require substantial redesign and modernization. This modernization includes developer operating system environments as they affect patch management. Security, network, and identity are next and offer an excellent opportunity for scalability, resilience, and cost reduction. After that comes storage, backup, and database tools as more apps move to the cloud. Also, you need to modernize your monitoring and management tools, which require training and re-skilling. Let's look into various strategies to modernize legacy systems.

Benefits of system modernization

Creating a future digital strategy by addressing the growing need for legacy system modernization can have many advantages, as shown in the following diagram:

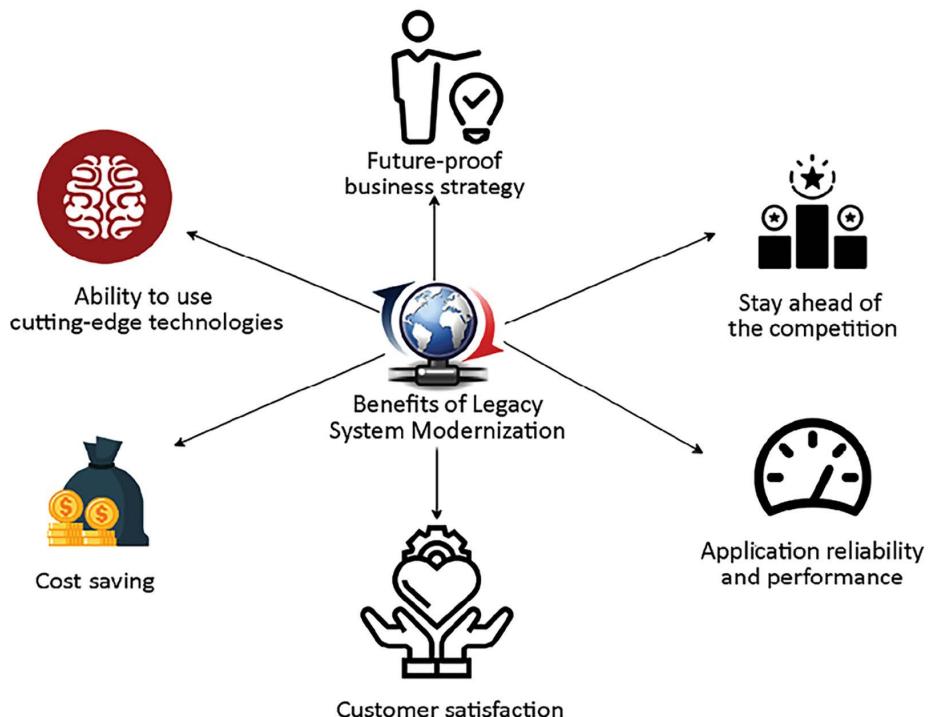


Figure 15.3: Benefits of legacy system modernization

The following are the significant benefits of application modernization:

- **Customer satisfaction:** Using the latest technology gives a better **user interface (UI)**, **user experience (UX)**, and an omnichannel experience. Consumers have grown accustomed to real-time access to information from any device, location, or time through personal experiences. You don't need to build different variations of the UI; it can be built once and deployed across devices such as laptops, tablets, and smartphones. A fast and slick UI leads to better customer experience and business growth.
- **Future-proof business strategy:** Modernizing your application allows you to be more agile and innovative. A team can accommodate the changing needs of the business comfortably and evolve with new technology.
- **Stay ahead of the competition:** Users always look for the latest stuff and move to new applications that give a better experience. The modernization of your application helps you stay ahead of the competition by following the latest trends. For example, voice integration is widely provided in apps, and you can enhance security with face detection. This is only possible when your application adopts the latest technology.
- **Application reliability and performance:** Every new software API and operating system version tries to address and improve performance issues. Using the latest software and hardware helps you achieve better performance, scalability, and high availability. Application modernization helps you reduce operational outages and enhance security.

- **Ability to use cutting-edge technologies:** Legacy systems prevent you from getting insights from data that could help you grow your business. By modernizing your database and creating a data lake, you can use big data and ML to get all kinds of insights. This also helps you retain employees when people get the opportunity to work with new technologies.
- **Cost savings:** Overall, any modernization leads to cost savings by reducing operational maintenance and providing a more natural upgrade. Utilization of open-source software reduces licensing costs, hardware flexibility helps to adopt a cloud pay-as-you-go model, and automation reduces the human resources needed for routine jobs and improves overall efficiency.

By migrating legacy core systems, organizations can modernize their core systems to reduce the cost of ownership, automate manual back-office processes, eliminate data silos, improve customer experience, and launch new market-facing applications faster.

There are several benefits of legacy system modernization but they can be very complex and require lots of effort. A careful assessment needs to be conducted to take the right approach. Let's explore the assessment techniques of a legacy application.

Looking at legacy system modernization techniques

Per your existing application analysis, you can take various approaches to upgrade your legacy system. The most straightforward approach is migration and rehosting, where you don't need to change the existing system. However, a simple migration may not solve the long-term problem or provide any benefits.

You can take a more complex approach, such as rearchitecting or redesigning the entire application if the system no longer meets business needs. The following diagram illustrates the impact of the various methods:

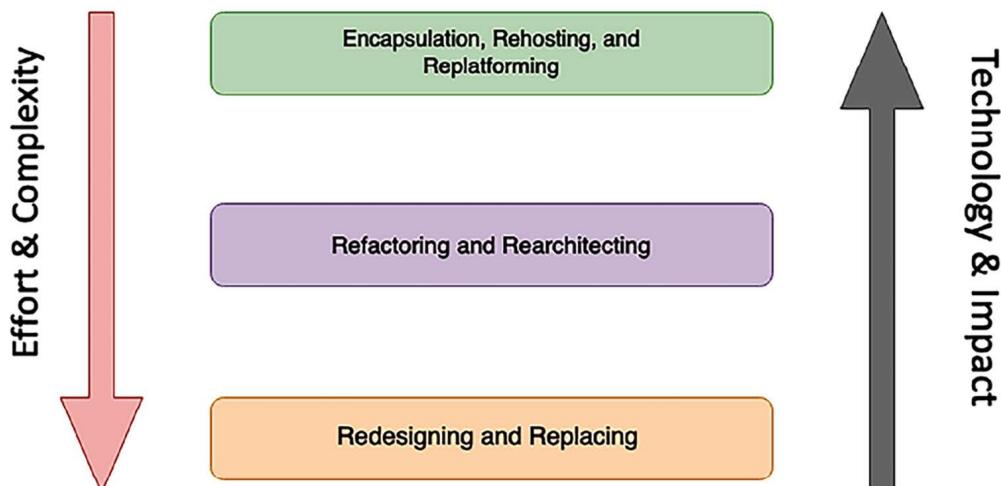


Figure 15.4: Legacy system modernization techniques

Let's examine the various modernization techniques shown in the preceding diagram.

Encapsulation, rehosting, and replatforming

Encapsulation is the most straightforward approach. If the system is business-critical and needs to communicate with other applications running on the latest technology, you should use this approach. With encapsulation, you need to build an API wrapper around your legacy system, allowing other business applications to communicate with a legacy application. An API wrapper is a common approach whereby you start migrating your applications to the cloud but keep the legacy application in the on-premises data center for modernization in the later phase. You can choose the encapsulation option if your legacy code is well-written and maintained but, again, you cannot benefit from technology advancements and hardware flexibility.

The **rehosting** approach is among the most straightforward strategies, whereby you migrate your application to another hardware provider, such as the AWS cloud, without any code changes. Again, as with encapsulation, the rehosting option can reduce costs due to vendor contracts but you may not benefit from technology advancements and hardware flexibility.

An organization often takes this approach when it needs to quickly move out of an existing contract. For example, you can take the first step to the cloud in phase one and apply modernization in phase two.

The **replatforming** approach may get more complex than the rehosting approach but will provide immediate benefits. Organizations often choose this approach if the server reaches EOL, where no support is available, and an upgrade is necessary to handle security issues. For example, if Windows Server 2012 is reaching EOL, consider upgrading the operating system to the Windows Server 2022 version. You need to rebuild your binaries with the new operating system and carry out testing to make sure everything works properly but there are no significant code changes. Again, as with rehosting, with replatforming, you may not benefit from technology advancements. However, it will allow you to have continuous support from the vendor.

While the preceding three approaches are the simplest ones, they cannot provide the full benefit of an application upgrade. Let's look at approaches that help you to take full advantage of application modernization.

Refactoring and rearranchitecting

You can refactor your code to accommodate the new system in the refactoring approach. The overall architecture will be the same in refactoring, but you upgrade your code to suit the latest programming language and operating system version. You can refactor the portion of code to apply automation and perform feature enhancement. If your technology is still relevant and can accommodate business needs with code changes, you should take this approach.

In the **rearchitecting** approach, you change the system architecture by reutilizing the existing code as much as possible. For example, you may create a microservices architecture from your monolithic architecture. You can take one module at a time and convert it into a service-oriented architecture by giving each module a RESTful endpoint. The rearchitecting option helps you achieve the desired scalability and reliability; however, overall performance results may be average due to the utilization of existing code.

Redesigning and replacing

The **redesigning** approach is the most complex but provides the maximum benefit. You can choose this approach if the legacy system needs to be updated and cannot accommodate business needs. With redesigning, you must build the entire system from scratch while keeping the overall scope intact.

The following diagram shows the legacy mainframe system migration to the AWS cloud:

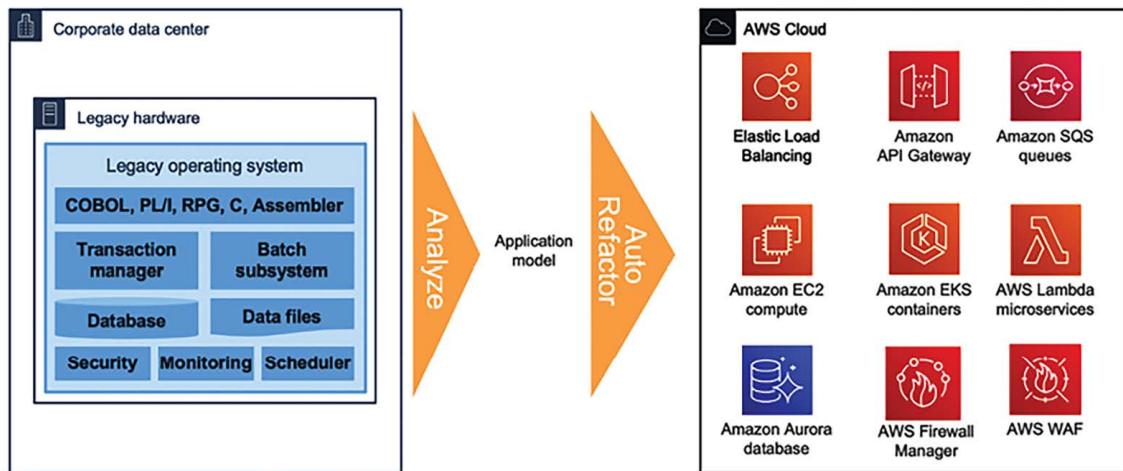


Figure 15.5: Legacy mainframe system modernization to the cloud

A legacy mainframe system is rearchitected and refactored to similar cloud services as a *modernization approach*. Building a cloud-native application helps you utilize and fully benefit from cloud services in aspects of scalability, performance, reliability, and cost. It helps your team to be more agile and innovative by accommodating rapidly changing technology in your system. You learned about cloud migration strategy and benefits in *Chapter 3, Cloud Migration and Hybrid Cloud Architecture Design*.

Redesigning a legacy system requires a long-term project with lots of effort and increased cost. Before kicking off modernization, as a solutions architect, you should carefully analyze if any SaaS product or **commercially available off-the-shelf (COTS)** products can handle your business needs at a lower cost. A **cost-benefit analysis (CBA)** between *redesign* and *purchase* is essential before proceeding with the redesigning option.

Sometimes, it's more beneficial to replace the legacy system with new third-party software. For example, your organization may have a decade-old **Customer Relationship Management (CRM)** system that cannot scale and provide the desired feature. You can subscribe to SaaS products such as Salesforce CRM to replace the legacy system. SaaS products are subscription-based and offer per-user licenses, so they may be the right choice if you have few users. Building its application may be more cost-effective for a vast enterprise with tens of thousands of users. You should conduct a CBA to understand the ROI when investing in SaaS products. Let's look at a cloud migration strategy in a nutshell here.

Defining a cloud migration strategy for legacy systems

As the cloud becomes increasingly popular, more organizations are looking to migrate to the cloud for their legacy application modernization needs. You learned about various cloud migration techniques in *Chapter 3, Cloud Migration and Hybrid Cloud Architecture Design*. The cloud allows you to scale your application while keeping costs low and helps you achieve desirable performance, high availability, and reliability while maintaining application security.

Cloud providers such as AWS, Microsoft Azure, and GCP provide many options that can help you modernize your system. For example, you can take a serverless approach to build a microservice using AWS Lambda and Amazon API Gateway, with Amazon DynamoDB as the backend.

In the previous section, we discussed various legacy system modernization techniques and their application in moving to the cloud. The flow illustrated in the following diagram will help you decide whether to use cloud migration to modernize your legacy application:

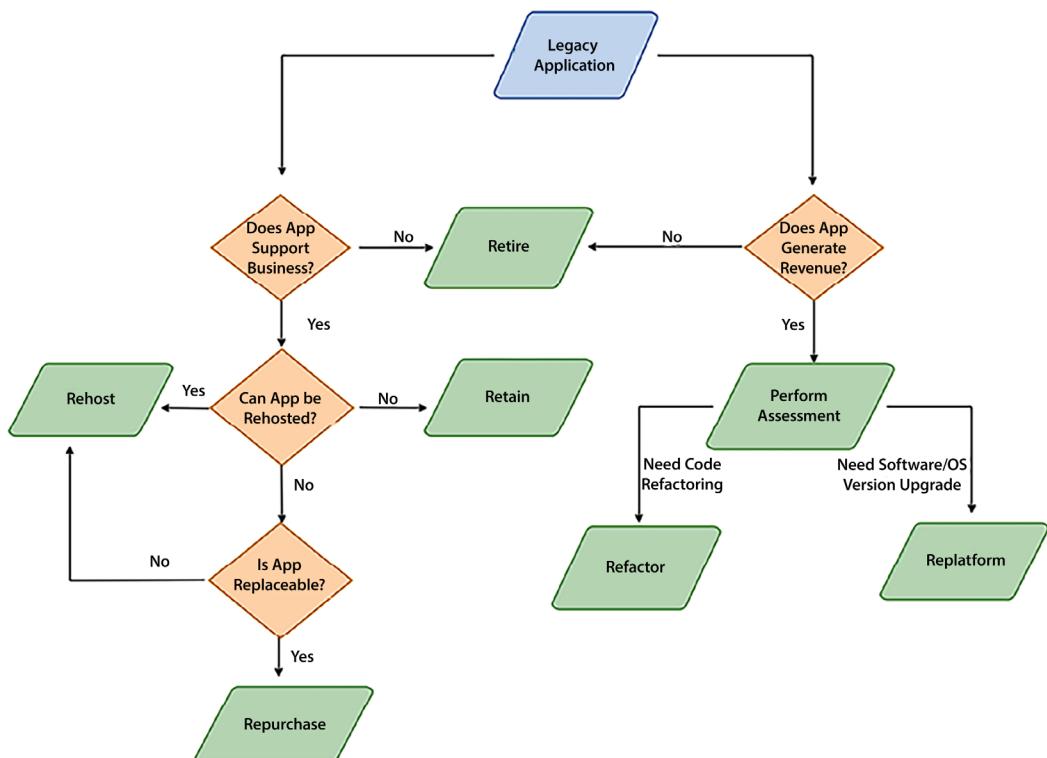


Figure 15.6: Cloud migration path for legacy system modernization

As shown in the preceding diagram, if your application is still heavily used by businesses and generates revenue, you should continue with minimal changes. In that situation, you can refactor your application into the cloud or re-platform it in the cloud if the server is reaching EOL.

If you want to keep existing applications the same to sustain business and still want to move to the cloud entirely to save and optimize costs, then take the *lift-and-shift approach* to rehost the legacy application in the cloud. If your legacy application is replaceable, you can buy a cloud-native SaaS version of the product and retire your legacy application. You may want to retain your legacy system in the on-premises data center if there are too many business dependencies and it cannot move into the cloud due to incompatibility.

You should perform a **total cost of ownership (TCO)** analysis to understand the advantages of moving to the cloud. This includes:

- **Infrastructure costs:** Compare the costs of on-premises infrastructure, including servers, storage, networking, and data center facilities, against the cloud services expenses.
- **Maintenance and administrative costs:** Factor in the costs associated with maintaining and managing on-premises infrastructure, such as IT staff salaries, compared to the managed services in the cloud that reduce the need for in-house management.
- **Scalability and flexibility:** Evaluate the cost implications of the cloud's ability to scale resources up or down based on demand, which can lead to cost savings compared to the fixed costs of on-premises infrastructure.
- **Licensing and subscription costs:** Include the costs of software licenses and cloud service subscriptions.
- **Migration costs:** Consider the one-time costs of migrating workloads to the cloud, including data transfer costs, tooling, and potential downtime.
- **Security and compliance:** Assess the costs related to achieving and maintaining security and compliance standards, both in on-premises environments and the cloud.

It is recommended to take the most complex module of the legacy application and build a POC to ensure your entire system will be cloud-compatible before starting the full project. A detailed POC covering the critical business cases will help you identify gaps and reduce the migration risk significantly.

When evaluating cloud compatibility, several key factors should be considered. Firstly, assess the architectural fit, ensuring the application's design adheres to cloud principles such as scalability, resilience, and decoupling. Next, identify any dependencies on specific hardware or local resources that the cloud may not optimally support. It's also crucial to confirm that the cloud environment can meet the application's performance requirements, taking into account factors like network latency and resource availability. Additionally, ensure that the application's security and compliance demands can be fully met in the cloud. Finally, evaluate the cost-effectiveness of migrating to the cloud, ensuring it aligns with the organization's financial objectives. This holistic approach helps determine whether an application is well-suited for a cloud environment, facilitating informed decision-making regarding migration.

Documentation and support are a critical part of any application maintenance. Let's learn more about them.

Documentation and support

Prepare proper documentation and support for the long-term sustainability of a new system and graceful migration to it. Provide documentation for your coding standards that everyone can follow, which helps keep the new system current. Keep your architecture documents as working artifacts, keeping them updated as technology trends change. Keeping your system updated will ensure you avoid facing the legacy system modernization situation again.

Prepare a comprehensive runbook to support new and old systems. You can keep the old system for some time until the new system can accommodate all business requirements and run satisfactorily. Update the support runbook, and ensure that you don't lose knowledge due to employee attrition and that the overall knowledge base is not processed in a people-dependent manner.

Keeping track of system dependencies helps you to determine the impact of any changes in the future. You will learn more about documentation in *Chapter 16, Solution Architecture Document*. Prepare training content to train staff on the new system, and ensure they can support it if there's an operational outage.

Mainframes are one of the workloads that are legacy and still running on-premises in many organizations. Let's learn about how to migrate them to the cloud.

Mainframe migration with the public cloud

Many enterprises are moving their mainframe workloads to the cloud to take advantage of cost reduction, increased agility, technical debt reduction, digital strategy support, the legacy mainframe skills gap, and data analytics. Mainframe workloads are more challenging to migrate than x86-based workloads because legacy mainframe applications are often developed and deployed in a tightly coupled manner. For example, a mainframe application might include programs used by several subsystems or directly called by other applications. In these cases, changes made to the underlying programs also affect the associated subsystems and applications.

Transitioning from mainframe systems to the cloud presents a unique opportunity, even though cloud providers may not support the exact mainframe hardware architecture. For organizations, this transition involves strategic choices: they can emulate mainframe environments on x86 platforms or refactor their applications for x86 compatibility. While refactoring demands more upfront investment, it paves the way for a scalable application landscape in the cloud, ultimately aligning with digital transformation goals and driving innovation.

For legacy applications, you must take an incremental approach, where the migration is planned in waves, as a best practice. This approach helps to reduce risks because you select and prioritize closely related applications to be migrated together. However, this approach could sometimes be more complex for mainframe migrations because the mainframe application code can use temporal coupling (invoked synchronously) or deployment coupling (linked modules).

Let's see some unique challenges that you may face with mainframe migration.

Challenges of mainframe modernization

Modernizing mainframes presents unique challenges due to their scale, complexity, and the critical nature of the applications they often run. These systems frequently use outdated programming languages, possess intricate and undocumented dependencies, and require specialized knowledge that is increasingly scarce. The following are some of the key challenges:

- **Outdated programming languages:** Mainframes often run on legacy code bases that use older programming languages, which modern IT professionals may need to become more familiar with.
- **Complex system dependencies:** Many mainframe applications have evolved over decades with intricate, undocumented dependencies, making it difficult to disentangle and modernize them without breaking functionality.
- **Specialized knowledge requirements:** The expertise required to operate and maintain mainframe systems is becoming rare as the workforce skilled in these older technologies retires.
- **Data integrity and security:** During modernization, there is a critical need to maintain the integrity and security of data, which can be challenging when transitioning from a closed, secure mainframe environment to a more open system.
- **Business continuity risks:** Mainframes typically manage essential business operations. Modernization efforts must be carefully planned to avoid disrupting these business-critical processes.
- **Integration with modern technologies:** Integrating mainframe applications with newer cloud-based services and technologies can be complex due to differing architectures and communication protocols.
- **Scaling challenges:** Adapt mainframe applications, often not designed for horizontal scaling, to modern cloud environments where elastic scaling is the norm.
- **Cost implications:** Assess and justify the financial investment required for modernization, which can be substantial, especially in the short term.
- **Performance considerations:** Ensure that the modernized systems deliver equivalent or improved performance compared to the highly optimized mainframe systems.
- **Cultural and organizational resistance:** You may need to overcome resistance within the organization, as mainframe systems are deeply embedded in the company's operational fabric, and change can be met with apprehension from stakeholders accustomed to the existing systems.

Ensuring data integrity and security during the transition is also a significant concern, as is the risk of disrupting business operations, given that mainframes commonly handle core business processes.

Migrating the coupled application code affects dependent applications and carries some risks. To reduce these risks, you can decouple the mainframe application code without impacting dependent applications. From a code migration perspective, the two main types of legacy mainframe applications are standalone applications and applications with shared code. Let's look at the details of each migration pattern.

Migrating standalone applications

Let's assume that there are two applications, A and B, that are standalone mainframe applications. Each application consists of programs and subprograms that it uses exclusively.

Because the applications are self-contained, you can group the COBOL programs and subprograms by application for code refactoring, as shown in the following diagram.

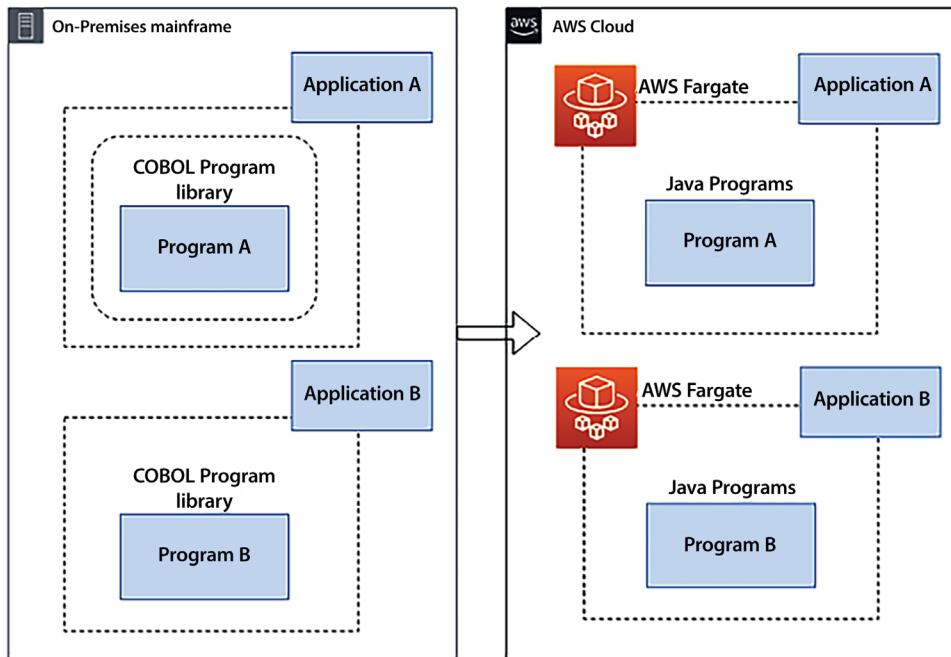


Figure 15.7: Mainframe modernization for a standalone application

In the preceding diagram, the mainframe programs and subprograms are written in COBOL, and the code is migrated to Java on AWS. However, you can use these decoupling patterns with your programming languages of choice. The migration pattern is legacy automated refactoring, where code, data, and dependencies are automatically converted to a modern language, data store, and framework while guaranteeing functional equivalence with the same business functions. Refactoring involves using automated tools to convert the mainframe programming language (such as COBOL) into modern programming languages (such as Java or .NET).

Refactored applications are deployed on containers provisioned and managed by AWS Fargate. Fargate is a serverless compute engine for containers that works with both Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS). Here, mainframe database tables and mainframe files are migrated with the application.

After grouping, you can migrate applications A and B in the same or different waves. In either case, package the refactored modern components for each application and deploy them together in a runtime environment. After migration, retire the on-premises mainframe applications and their components. Let's look at more complicated scenarios where multiple applications share code.

Migrating applications with shared code

Let's assume mainframe applications A and B run shared code called program AB. You need to perform an impact analysis of the shared program AB to migrate applications A and B and program AB together. Based on the impact analysis, identify the number of dependent applications that use shared programs, such as program AB. You need to complete a business domain analysis to determine whether the shared program can be aggregated into a domain with applications and exposed as an API as one of the domain services. Let's look at some approaches you can take to decouple the applications in preparation for migration.

Application decoupling using a standalone API

Using this approach, you instantiate a standalone API by converting the shared COBOL program AB into a Java program. You can use automated refactoring tools provided to generate network APIs for the program to minimize refactoring efforts. You can take this approach when the shared program can be instantiated as a standalone service. The remaining components of applications A and B are refactored into Java and migrated to the cloud. You can migrate the applications in the same wave, as shown in the following diagram:

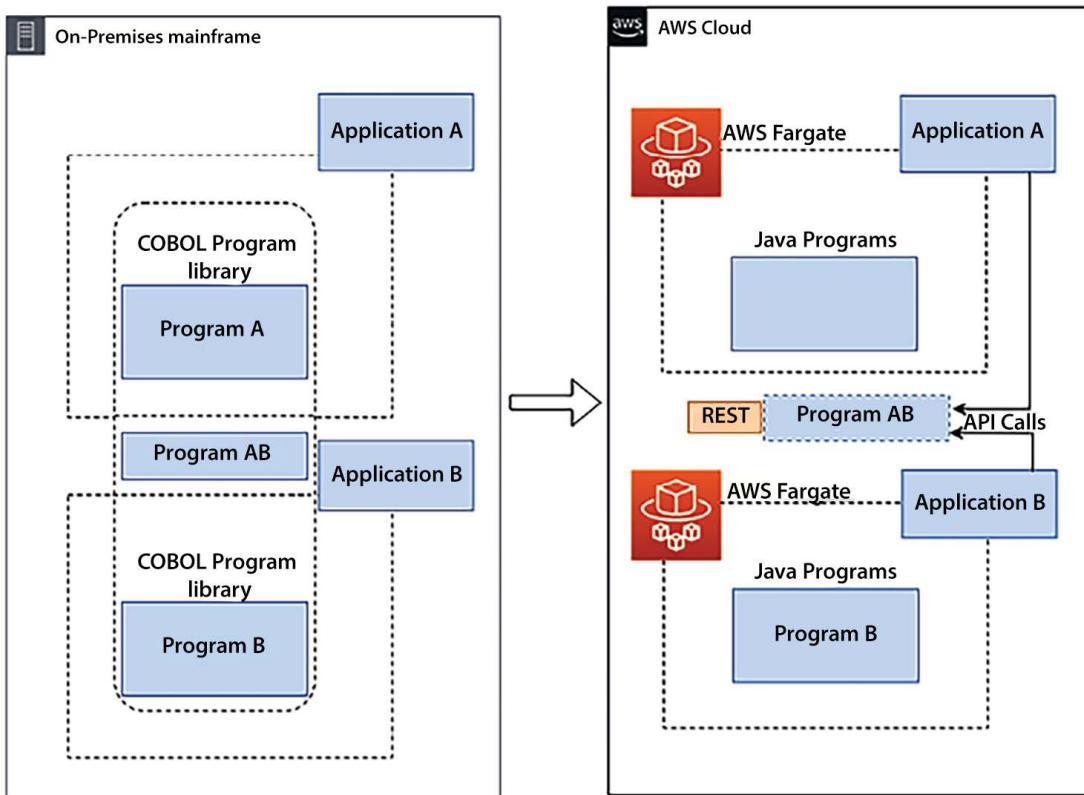


Figure 15.8: Migration of shared program applications using a standalone API

In this approach, you must refactor both applications with their respective programs and migrate them to the cloud. You need to use the impact analysis report from the analysis phase to help developers and teams identify the refactored applications that call the shared program AB. Replace the inner program call with network API calls to the shared program AB. After the migration, retire the on-premises mainframe applications and their components.

Application decoupling using a shared library

The shared program AB is converted into a Java standard library and packaged with the migration applications in this approach. You should take this approach when the shared program is a supporting library instead of a standalone service. The remaining components of applications A and B are refactored into Java programs and migrated to the cloud.

This approach refactors applications A and B with their associated programs into Java and migrates them to the cloud. You should maintain the source code of the applications in a fully managed source control service, such as AWS CodeCommit. The teams that use the shared program can collaborate on code changes by using pull requests, branching, and merging and can control the changes made to the shared program code. After the migration, retire the on-premises mainframe applications and their components.

When applications are too big to be grouped into the same migration wave, you can migrate them in multiple waves and maintain service continuity during the migration. With this approach, you can modernize your applications in phases without bundling them. Migrating your applications in separate waves decouples them without requiring significant code changes in the mainframe.

Application decoupling using message queues

In this approach, the shared program AB is converted into a Java program and migrated to the cloud as part of application A. A message queue is used as an interface between the refactored application in the cloud and the legacy application on-premises. Using this approach, you can break up tightly coupled mainframe applications into producers and consumers and make them more modular to function independently. The additional advantage is that you can migrate the applications in different waves.

You can take this approach when applications on the mainframe can communicate with the migrated applications in the cloud through a message queue. It would be best to ensure that the queuing architecture pattern meets the business requirements for the applications on the mainframe because it involves rearchitecting the existing applications.

You should take the message queue approach if applications not part of the first wave require longer (six months or more) to be migrated to the cloud. When applications are too big to be grouped into the same migration wave, you can migrate them in multiple waves, as shown in the following diagram, and maintain service continuity during migration.

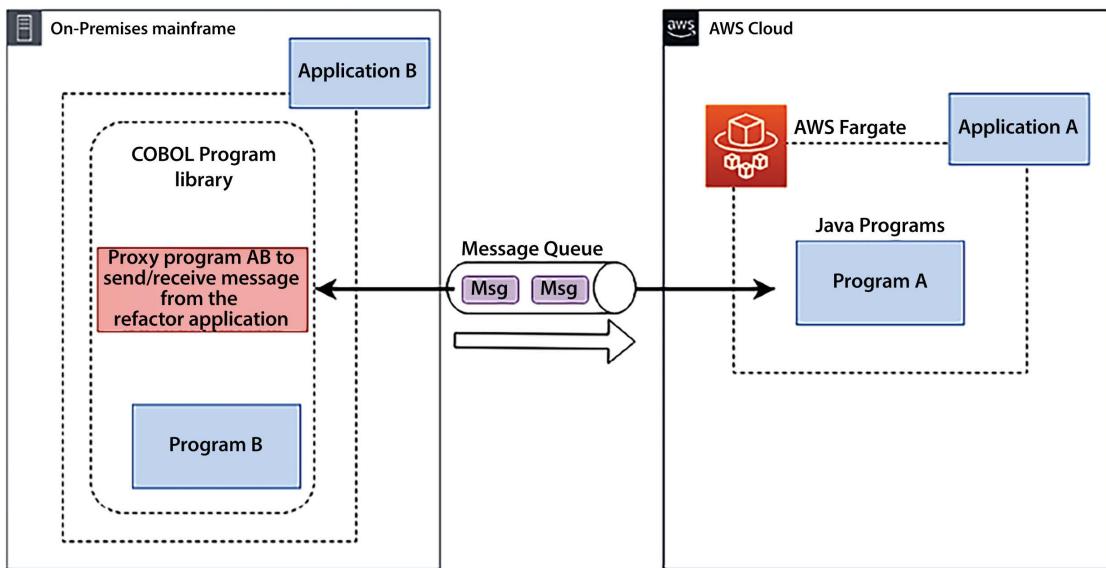


Figure 15.9: Migration of shared program applications using a message queue

As shown in the preceding diagram, you need to follow these steps for migration:

1. Migrate (refactor) application A with its associated programs to the cloud while application B continues to reside on-premises.
2. Refactor application A (in the cloud) to communicate with application B (on-premises) through a message queue.
3. Refactor application B on-premises to replace the shared program with a proxy program that sends and receives messages from application A through the message queue.
4. After successfully migrating application A, retire the on-premises application A and its components (including the shared program). Application B and its components will continue to reside on-premises.
5. In the next set of migration waves, migrate application B and its components. The loosely coupled queuing architecture continues to interface between applications A and B in the cloud. This reduces the refactoring effort for application B without impacting application A.

As a best practice, you should perform code analysis to produce a dependency map for the mainframe applications and identify the list of programs shared by applications. After that, group applications that share the same programs for the same migration wave to reduce program calls between the on-premises environment and the cloud. At the planning stage, run an impact analysis to identify applications that share programs with the application you plan to migrate, and select the correct decoupling patterns for application migration. In this section, you noticed that we used the AWS example for mainframe modernization. Let's take a detailed look at the benefits of the public cloud for mainframe modernization.

Benefits of using the public cloud for mainframe modernization

Leveraging the public cloud for mainframe modernization brings numerous benefits, such as enhanced scalability, flexibility, and cost savings. The cloud's pay-as-you-go model reduces capital expenditure, while its advanced services facilitate innovation, especially in areas like AI, ML, and analytics. Cloud environments also provide improved disaster recovery capabilities and the opportunity to redesign applications to be more resilient and adaptable to changing business needs. Let's look at some of the key benefits of the cloud migration of a mainframe workload:

- **Enhanced scalability:** Cloud platforms can automatically adjust resources to handle workload spikes, unlike mainframes that require manual scaling. For instance, an e-commerce website can handle holiday shopping traffic spikes without downtime.
- **Cost-effectiveness:** With the cloud's pay-as-you-go model, companies save on the high upfront hardware and maintenance costs. For example, startups can launch new applications without investing in expensive mainframe infrastructure.
- **Flexibility and agility:** Cloud services allow businesses to quickly experiment and deploy new applications. A company could, for instance, swiftly test a new customer service application in various markets without a lengthy setup process.
- **Innovation acceleration:** Cloud providers offer cutting-edge AI, ML, and analytics tools. A retail business could use these tools to analyze consumer data and personalize marketing strategies, which could be improved on a traditional mainframe.
- **Improved disaster recovery:** Cloud platforms have built-in redundancy and backup solutions. For example, a financial institution can ensure continuous operation and data integrity, even in a local disaster.
- **Resource optimization:** The cloud enables more efficient use of computing resources. A company might use cloud services to run applications only when needed, reducing idle computing resources common in mainframe environments.
- **Faster time-to-market:** The agility of the cloud shortens the development cycle. A mobile app developer can quickly deploy and update apps, staying ahead in a competitive market.
- **Global reach:** With data centers worldwide, cloud services allow businesses to deploy applications close to their users. An online streaming service, for example, can provide low-latency content to users globally.
- **Better security features:** Major cloud providers invest heavily in cybersecurity. This means a small business can benefit from security measures on par with large enterprises, something challenging to achieve with on-premises mainframes.
- **Easier integration with modern technologies:** The cloud simplifies integration with modern apps and services. A healthcare provider, for example, could integrate cloud-based AI diagnostics tools with its patient management system, a task that would be complex and resource-intensive in a mainframe environment.

AWS provides the **Mainframe Modernization (M2)** platform, designed to migrate and modernize on-premises mainframe applications to a cloud-native, fully managed runtime environment on AWS. The following are the key features of the AWS M2 platform:

- **Automated refactoring:** Transforms legacy language applications into agile Java-based services using AWS Blu Age, adhering to modern web frameworks and cloud DevOps best practices.
- **Replatforming:** Migrates COBOL applications using the integrated Micro Focus toolchain, modernizing infrastructure while preserving the programming language for agility with DevOps cloud-native operations.
- **Data replication and file transfer:** Enhances mainframe functions through near real-time data replication with Precisely and file transfer capabilities with BMC Software.
- **Application testing:** Automates the validation of modernized mainframe applications, reducing costs and accelerating testing timelines with a scalable cloud-native service.

This service aligns with the growing need for legacy system modernization, providing a comprehensive solution for enterprises transitioning from traditional mainframe infrastructure to more agile, cost-effective cloud environments. You can learn more by referring to this AWS page: <https://aws.amazon.com/mainframe-modernization/>.

When possible, perform mainframe migration incrementally to reduce complexity and risk. By doing incremental migration, migration teams can provide faster feedback regarding the migration progress, and businesses can use the feedback to optimize internal processes to accelerate the pace of migration. As GenAI becomes more popular with many out-of-the-box offerings, it can help to accelerate your modernization journey. Let's learn more about it.

Modernizing legacy code with generative AI

Modernizing legacy code with GenAI represents a cutting-edge approach to software development. GenAI tools can analyze and understand legacy code, often written in outdated programming languages, and assist in rewriting or translating it into modern, more efficient languages or frameworks. This process accelerates code modernization and helps maintain the functionality of legacy systems while leveraging the advantages of current technologies.

By automating part of the code conversion process, genAI reduces the manual effort and expertise required, making the modernization process more accessible and less error-prone. This approach mainly benefits businesses looking to update their legacy systems without disrupting operational efficiency. Modernizing legacy code with GenAI involves tools like Codex, part of OpenAI's offerings (which powers GitHub Copilot), and, potentially, foundation models. These tools leverage AI to understand and refactor legacy code into more modern, efficient programming languages or frameworks. For example, Codex can interpret older, less common programming languages and provide suggestions or direct translations into newer languages, like Python or JavaScript. This facilitates updating legacy systems to be more maintainable and compatible with current development practices.

Similarly, Amazon CodeWhisperer is AWS's AI-powered coding assistant. Like GitHub Copilot, it helps developers by providing code suggestions and automating some coding tasks. CodeWhisperer leverages ML models to understand the code's context and offers relevant recommendations. This tool can enhance developer productivity, aid in coding best practices, and potentially assist in modernizing legacy code by suggesting contemporary coding techniques and solutions. CodeWhisperer, integrated into a developer's workflow, can significantly streamline the process of maintaining, updating, and optimizing codebases, including the translation or refactoring of legacy systems. CodeWhisperer also provides contextual recommendations to make sure that code library reuse and coding pattern consistency are maintained.

Additionally, foundation models, trained on diverse data across many domains, can assist in understanding complex legacy codebases, identifying redundant or inefficient code segments, and suggesting optimizations or modern coding patterns.

Integrating these tools into the modernization process speeds up the translation of legacy code. It contributes to more maintainable, scalable, and secure software systems, crucial for businesses aiming to stay competitive in a rapidly evolving digital landscape.

Summary

In this chapter, you learned about the various challenges with legacy applications and why it is essential to modernize them. You learned about the benefits an organization can get by upgrading its application to the latest technology. Application modernization can be complicated and risky, but it is often worthwhile.

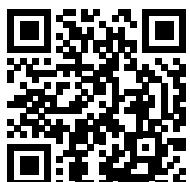
The outcome you get from the upgrade is a trade-off against the amount of investment and energy you put into it. Before defining the modernization approach, it's essential to understand your legacy system thoroughly, and you learned about various assessment attributes of an application in technology, architecture, and code.

After the assessment, the next step is to define the modernization approach, and you learned about various modernization approaches, including architecture-driven, system re-engineering, and migration approaches. You also learned about multiple techniques for system modernization, including straightforward approaches (encapsulation and rehosting) and complex approaches (rearchitecting and redesigning). The cloud can provide a significant value proposition, and you learned about the decision-making approach you need to take with modernization in the cloud. You learned about mainframe modernization challenges and the cloud's benefits to ease your mainframe modernization journey. Finally, you learned how GenAI can help improve developer efficiency by updating legacy code as a coding assistant.

You focused on the various technical aspects of solution architecture; however, documentation is one of the critical elements of architectural design that keeps your system maintainable in the long run. The next chapter will discuss the documentation required for a solutions architect to prepare, contribute to, and maintain maximum business value.

Join our book's Discord space

Join the book's Discord workspace to ask questions and interact with the authors and other solution architecture professionals: <https://packt.link/SAHandbook>



16

Solution Architecture Document

In previous chapters, you learned about the various aspects of solution architecture design and optimization. As the solutions architect works on the design, it is essential to have consistent communication with other stakeholders for successful application delivery. The solutions architect must communicate a design to all technical and non-technical stakeholders.

The **solution architecture document (SAD)** provides an end-to-end view of the application and helps everyone to be on the same page. In this chapter, you will learn about various aspects of the SAD, which addresses the needs of all stakeholders associated with the development of the application.

You will learn about the structure of the SAD and other types of documents of which the solutions architect needs to be aware, such as the request for proposal, where the solutions architect needs to provide input to make strategic decisions. We will cover the following topics to gain a deeper understanding of the documentation involved in a solution architecture:

- Purpose of the SAD
- Views of the SAD
- Structure of the SAD
- Life cycle of the SAD
- SAD best practices and common pitfalls
- IT procurement documentation for a solution architecture

By the end of this chapter, you will know about the SAD, its structure, and the various details that need to be accommodated in the documentation. You will learn about various IT procurement documentation, such as the request for proposal, the request for information, and the request for quotation, in which a solutions architect participates to provide feedback. Let's start with the basics by learning the purpose of the SAD.

Purpose of the SAD

Often, architecture documentation needs are not addressed, and teams start working on implementation without understanding the overall architecture. The SAD provides a broad view of the overall solution design to keep all stakeholders informed.

The SAD is vital for various stakeholders, including project managers, who rely on it to oversee project coordination and progress. Business analysts use it to align the project with business requirements. Technical teams, including developers and IT professionals, refer to it for implementing and maintaining the proposed solutions. Senior management utilizes the document to make informed strategic decisions. Finally, clients or end users, who are the ultimate beneficiaries, depend on this document to ensure the project outcome meets their needs and expectations.

The SAD helps to achieve the following goals:

- Communicate the end-to-end application solution to all stakeholders.
- Provide a high-level overview of the architecture and different views of the application design to address the application's service-quality requirements, such as reliability, security, performance, and scalability.
- Provide traceability of the solution back to the business requirements and look at how the application will meet all functional and **non-functional requirements (NFRs)**.
- Provide all views of the solution required for design, building, testing, and implementation.
- Define the impacts of the solution for estimation, planning, and delivery purposes.
- Define the business process, continuation, and operations needed for a solution to work uninterrupted after the production launch.

SADs define the purpose and goal of the solution and address critical components such as solution constraints, assumptions, and risks often overlooked by the implementation team. The solutions architect must ensure they create the document in an easy language that business users can understand and relate the business context with the technical design. Documentation helps retain knowledge in case of resource attrition and makes the overall design process people independent.

For existing applications where modernization efforts are needed, the SAD presents an abstract view of the current and future architectures and a transition plan. The solutions architect understands the existing system dependencies and documents them to uncover any potential risk in advance. The migration plan helps businesses understand the tools and technology required to handle the new system and plan resources accordingly.

The solutions architect conducts various assessments during solution design by building a **proof of concept (POC)** or through market research. A SAD should list all architecture assessments and their impact, along with the choice of technology. A SAD presents a conceptual view of the solution design's current and target state and maintains a change record. Let's understand the various views of a SAD in the next section.

Views of the SAD

The solutions architect needs to create a SAD that both business and technical users understand. A SAD bridges the communication gap between the business user and the development team to understand the function of the overall application. The best way to capture all stakeholders' input is by putting yourself in their situation and looking at problems from the stakeholders' perspectives. The solutions architect evaluates the business and technical aspects of architecture design to take cognizance of all technical and non-technical user requirements.

As illustrated in the following diagram, a holistic overview of the SAD comprises various views to cover different aspects derived from the business requirements:

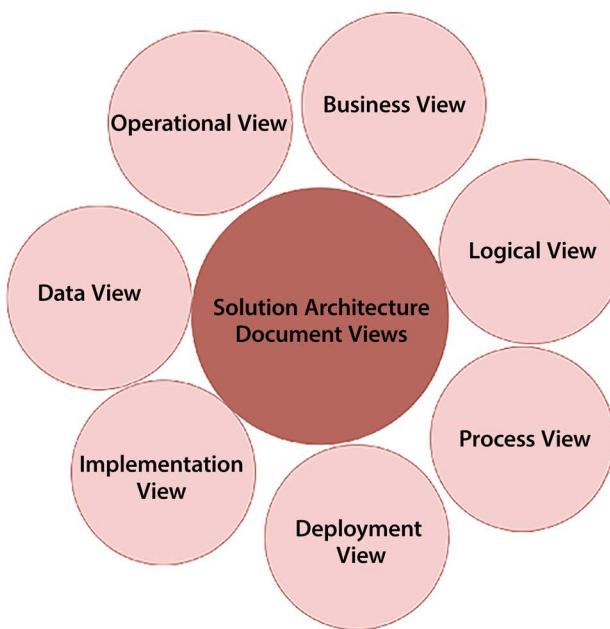


Figure 16.1: SAD views

Solutions architects can choose standard diagrams such as a **Unified Modeling Language (UML)** diagram or a block diagram from **Microsoft Visio** to represent various views. These tools are widely recognized and help convey complex architectural concepts in an easily digestible format. Overall, the diagram should be easy to read and understandable to all business and technical stakeholders. A SAD should include the following views, wherever possible, to address everyone's needs:

- **Business View:** Architecture design addresses business concerns and solves business purposes. The business view shows the value proposition of the overall solution and product. To simplify, the solutions architect may choose to detect high-level scenarios related to business and present these as a use case diagram. The business view also describes stakeholders and the required resources to execute the project. You can define the business view as a use case view as well.
- **Logical View:** This presents various packages on the system so that business users and designers can understand the various logical components of the system. The logical view offers a chronicled order of the system in which it should be built. It shows how the multiple packages of the system are connected and how the user can interact with them. For example, in a banking application, the user first needs to authenticate and authorize using a security package, log in to the account using the account package, apply for a loan using a loan package, and so on. Each package represents a different module and can be built as a microservice.
- **Process View:** This presents more details, showing how the critical processes of the system work together. It can be reflected using a state diagram. The solutions architect can create a sequence diagram to show more details. In a banking application, a process view can present the approval of a loan or account.

- **Deployment View:** This presents how the application will work in production. It shows how connected system components (such as the network firewall, load balancer, application servers, and database) are. The solutions architect should create a simple block diagram that business users can understand. You can add more details to the UML deployment diagram to show various node components and their dependencies for technical users, such as the development and DevOps teams. The deployment view represents the physical layout of the system.
- **Implementation View:** This is the core of the SAD and represents architectural and technology choices. The solutions architect needs to put the architecture diagram here—for example, if it is 3-tier, N-tier, or event-driven architecture—along with their reasoning.
- It would help if you also detailed technology choices—for example, using Java versus Node.js, along with their pros and cons. In the implementation view, you want to justify the resources and skills required to execute the project. The development team uses an implementation view to create a detailed design, such as a class diagram, but that doesn't need to be part of the SAD.
- **Data View:** Most applications are data-driven, so the data view is essential. The data view represents how data will flow between the different components and how it will be stored. It can also be used to explain data security and data integrity. The solutions architect can use the entity-relationship diagram to show the relationship between different tables and schemas in the database. You will learn more about the entity-relationship diagram in the *Data architecture* section. The data view also explains the reports and analytics needed.
- **Operational View:** This explains how the system will be maintained post-launch. Often, you define **service-level agreements (SLAs)**, alert and monitoring functionality, a disaster recovery plan, and a support plan for the system. The operational view also details how system maintenance will be carried out, such as by deploying a bug fix, patching, backing up and recovering, and handling security incidents.

All the views listed ensure the SAD covers all aspects of the system and stakeholders. You may include additional views—such as a physical architecture view, a network architecture view, or a security (controls) architecture view—per the stakeholders' requirements. As a solutions architect, you must view and understand system functioning in a comprehensive manner. Let's explore the structure of the SAD in more detail in the next section.

Structure of the SAD

The structure of the SAD can differ from project to project per the stakeholder requirements and the nature of the given project. Your project could be creating a new product from the ground up, modernizing a legacy application, or moving the entire system to the cloud.

For each project, the SAD document may differ, but overall, it should consider various stakeholders' views and include an array of sections as shown in the following screenshot:

Contents	4.2 Application Architecture
	4.2.1 Application components
1. Solution Overview	4.3 Data Architecture
1.1 Solution Purpose	4.3.1 Data Flow and Context
1.2 Solution Scope	4.4 Integration Architecture
1.2.1 In Scope	4.4.1 Interface Component
1.2.2 Out of Scope	4.5 Infrastructure Architecture
1.3 Solution Assumptions	4.5.1 Infrastructure Component
1.4 Solution Constraints	4.6 Security Architecture
1.5 Solution Dependencies	4.6.1 Identity and Access Management
1.6 Key Architecture Decisions	4.6.2 Application Threat Model
2. Business Context	5. Solution Implementation
2.1 Business Capabilities	5.1 Development
2.2 Key Business Requirements	5.2 Deployment
2.2.1 Key Business Processes	5.3 Data Migration
2.2.2 Business Stakeholders	5.4 Application Decommissioning
2.3 Non-Functional Requirements	6. Solution Management
2.3.1 Scalability	6.1 Operational Management
2.3.2 Availability and Reliability	6.1.1 Monitoring and Alert
2.3.3 Performance	6.1.2 Support and Incident Management
2.3.4 Portability	6.1.3 Disaster Recovery
2.3.5 Security	6.2 User On-boarding
3. Conceptual Solution Overview	6.2.1 User system requirement
3.1 Conceptual and Logical Architecture	7. Appendix
4. Solution Architecture	7.1 Open Items
4.1 Information Architecture	7.2 Proof of Concept findings
4.1.1 Information components	

Figure 16.2: Structure of a SAD

In the preceding SAD structure, you can see different sections covering multiple solution architecture and design aspects. The solutions architect may choose to add additional subsections or remove some sections as per the project requirements. For example, you can add another introduction section to discuss the document's purpose with a summary. For a transition project, you may add a subsection to present the existing architecture, compare it with the target architecture, and so on. Let's look into the details of each section.

Solution overview

In the solution overview section, we briefly introduce the solution in a couple of paragraphs, describing the functioning of the solution and its different components at a very high level. Adding a high-level block diagram showing various components in one place is nice. The following diagram illustrates the solution overview of an e-commerce platform:

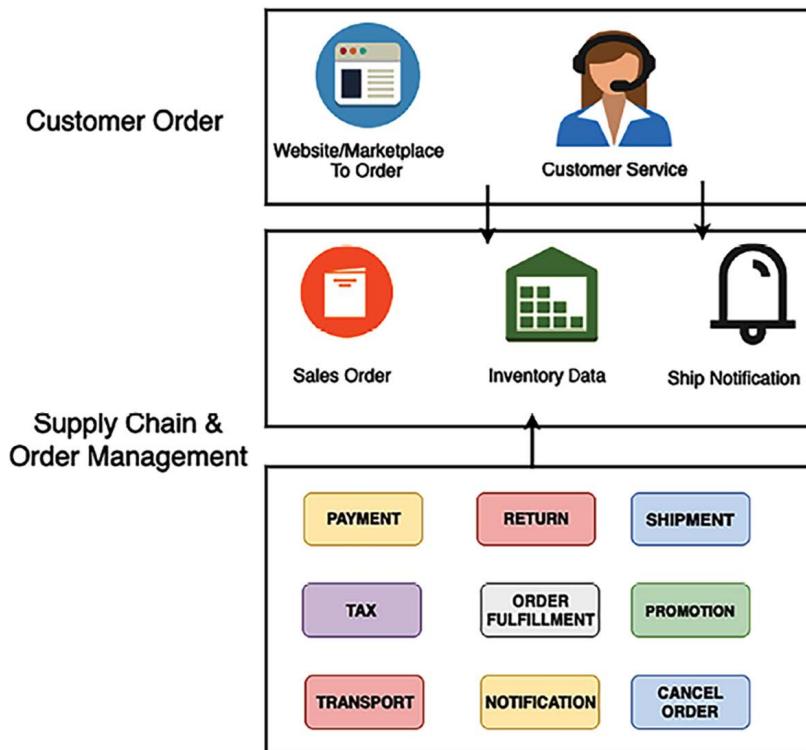


Figure 16.3: Solution overview of an e-commerce platform

You need to provide a brief about each component in simplified language so that the business user can understand the overall workings of the solution. For example, the above diagram outlines a typical e-commerce order and supply chain management workflow in a simple way:

- **Customer Order:**
 - The process begins when a customer places an order through a website or an online marketplace.
 - Customer service plays a role in assisting customers with their orders, handling inquiries, and resolving issues that may arise during the order process.
- **Supply Chain & Order Management:**
 - **Sales Order:** Once an order is placed, it is recorded as a sales order, which triggers the order fulfillment process.

- **Inventory Data:** The system checks inventory data to ensure the ordered items are in stock.
- **Ship Notification:** After the order is processed and ready for shipment, a notification is sent out, possibly including tracking information for the customer.
- **Order processing:**
 - **Payment:** Processing the customer's payment, including the application of any taxes and processing of promotional codes.
 - **Tax:** Calculating and applying the correct sales tax based on the customer's location and the purchased items.
 - **Transport:** Arranging the logistics for delivering the order to the customer's location.
 - **Order fulfillment:** The actual picking, packing, and preparation for shipping of the ordered items.
 - **Shipment:** The process of sending the order out for delivery.
 - **Promotion:** Applying discounts or special offers that might be part of the sales order.
 - **Notification:** Keeping the customer informed about the status of their order.
 - **Return:** Handling any returns, if the customer is not satisfied with their purchase or if there are issues with the order.
 - **Cancellation:** Processing any cancellations if the customer decides to cancel the order before it's shipped.

These elements allow customer orders to be handled efficiently from the moment they are placed until they are delivered.

Next, we outline the solution overview with the following subsections:

- **Solution purpose:** This provides a brief outline of the business concern that the solution will solve and the justification for building this solution. In our scenario outlined earlier, the purpose of this solution is to streamline and automate the process of order management within a supply chain. It aims to address business concerns around efficiency, accuracy, and customer satisfaction in order fulfillment.
- **Solution scope:** This is where the proposed solution's business scope is stated, clearly describing out-of-scope items that the solution will not accommodate. The scope includes the end-to-end automation of the order management system, from customer order placement to shipment notification. It does not cover post-delivery customer engagement like feedback collection or returns processing.
- **Solution assumptions:** List all the assumptions made by the solutions architect in coming up with the solution. For example, in our scenario, the solution assumes a minimum network bandwidth availability to ensure real-time data processing. It also assumes the integration capability with various marketplaces and shipping carriers and customers have access to digital payment methods.

- **Solution constraints:** List all technical, business, and resource constraints. Often, constraints come from industry and government compliances, which need to be listed in this section. You can also highlight the risk and mitigation plan. The solution must comply with data protection regulations like GDPR for customer data privacy in the EU and PCI-DSS for storing customer's payment information in the USA. Resource constraints include a fixed budget and timeline for deployment. There may be technical constraints related to the integration of legacy systems.
- **Solution dependencies:** List all upstream and downstream dependencies. For example, an e-commerce website must communicate with a shipping system such as UPS or FedEx to ship a package to customers. The solution depends on real-time inventory data for accurate order processing from the inventory management system. It requires integration with payment gateways for financial transactions.
- **Key architecture decisions:** List significant problem statements and proposed solutions. Describe the pros and cons of each option, why a particular decision was made, and the rationale behind it.

For example, let's take the decision to use a cloud-based platform for scalability. It was chosen for its ability to handle varying order volumes and the need for less upfront capital investment. The trade-off is a continuous operational expense.

Another decision could be the adoption of an API-first approach for integration. This was selected to ensure flexibility and the ease of integrating with a variety of partners and services. However, this increases the complexity of API management.

After giving a solution overview, you want to relate it to the business context. In the next section, let's look at the business context view in more detail.

Business context

In the business context section, the solutions architect must provide a high-level overview of the business capabilities and requirements the solution will address. This section only contains an abstract view of requirements. Detailed requirements need to be mentioned and noted down in a separate document. However, the external link to the requirements document can be provided here. You should include the following primary subsections:

- **Business capabilities:** Briefly describe the business capabilities for which the solution is being designed. Make sure to include the benefits of capabilities and how they will address customer needs.
- **Key business requirements:** List all key business concerns the solution will address. Provide a high-level view of key requirements and add a reference to the detailed requirements document.
- **Key business processes:** Solutions architects should show key processes with a business process document. The following diagram illustrates a simplified view of an e-commerce application business process model:

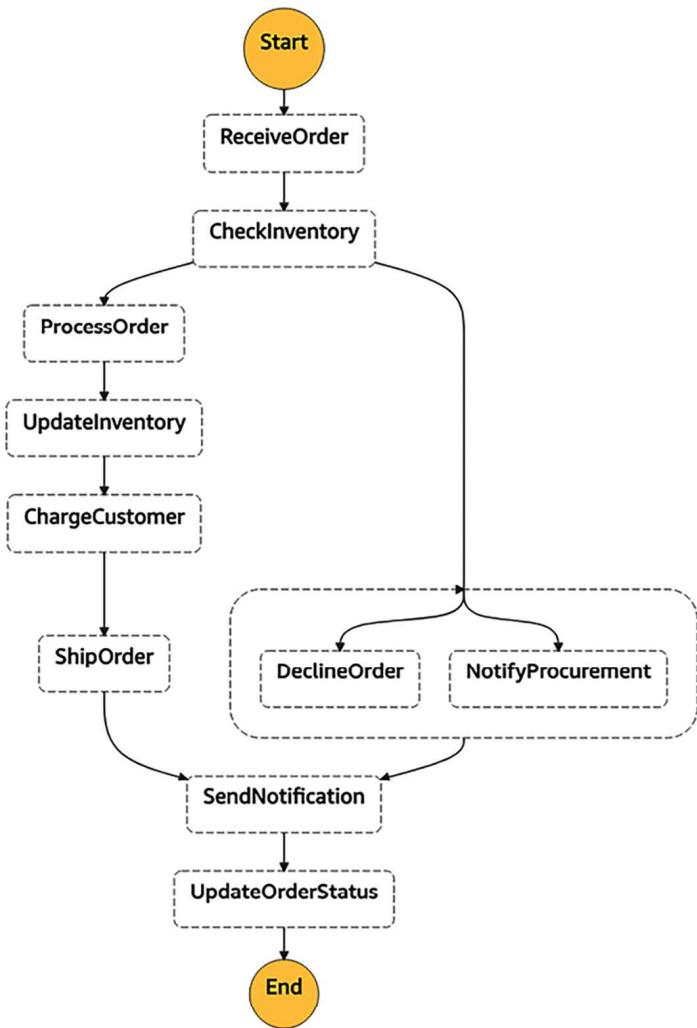


Figure 16.4: Business process diagram of an e-commerce platform

- **Business stakeholders:** List stakeholders directly or indirectly impacted by the project. This includes sponsors, developers, end users, vendors, and partners.
- **NFRs:** Solutions architects must focus more on NFRs as these often get missed by the business user and development team. At a high level, an NFR should include:
 - **Scalability:** How can the application scale as workloads fluctuate? (For example, scale from 1,000 to 10,000 transactions per second per day or month.)
 - **Availability and reliability:** What is the acceptable downtime for system availability? (For example, 99.99% availability or 45 minutes' downtime per month.)
 - **Performance:** What is the performance requirement? Where can the system handle the load increase without impacting the end-user experience? (For example, the catalog page needs to load within 3 seconds.)

- **Portability:** Can the application run on multiple platforms without any additional work? (For example, the mobile app must run on the iOS and Android operating systems.)
- **Capacity:** What is the maximum workload that an application can handle? (For example, the maximum number of users, the number of requests, the expected response time, and the expected application load.)

The conceptual view of architecture is a sweet spot that provides a good system overview for both business and technical stakeholders. Let's learn more about the conceptual view in more detail.

Conceptual solution overview

The conceptual solution overview section provides an abstract-level diagram that captures a big-picture view of the whole solution, including both its business and technical aspects. It provides a basis for analyses and trade-off studies to help refine and optimize the solution architecture in sufficient detail to support solution design and implementation. The following diagram illustrates a conceptual architecture diagram of an e-commerce platform:

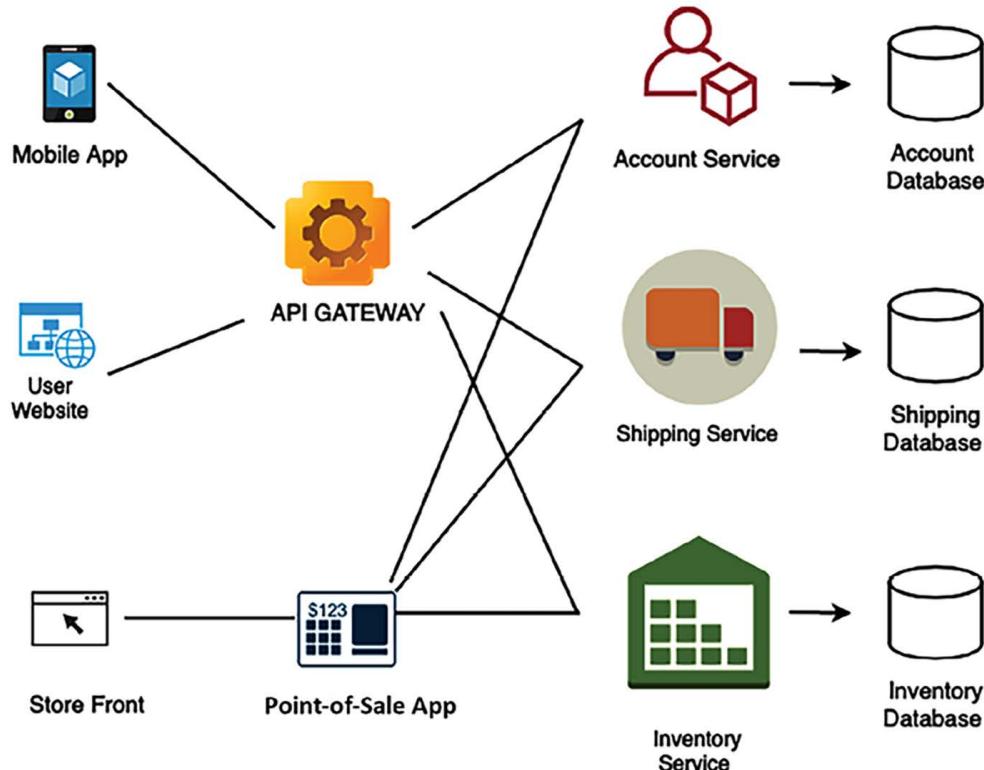


Figure 16.5: Conceptual architecture diagram of an e-commerce platform

The preceding diagram shows an abstract view of significant modules and information flowing between them. The conceptual architecture provides a good understanding of the overall architecture for both business and technical users. However, technical users need further architectural depth. Let's dive deeper into the solution architecture in the next section.

Solution architecture

The solution architecture section dives deep into each part of the architecture. It provides different views that the technical team can use to create a detailed design and work on implementation. These views could target other user groups, such as developers, infrastructure engineers, DevOps engineers, security engineers, and **user experience (UX)** designers.

Let's get into the following major subsections to learn more details.

Information architecture

This section provides a user navigation flow to the application. The solutions architect needs to put an application navigation structure at a high level. As shown in the following diagram, for an e-commerce website, it takes three clicks for the user to navigate to the desired page:

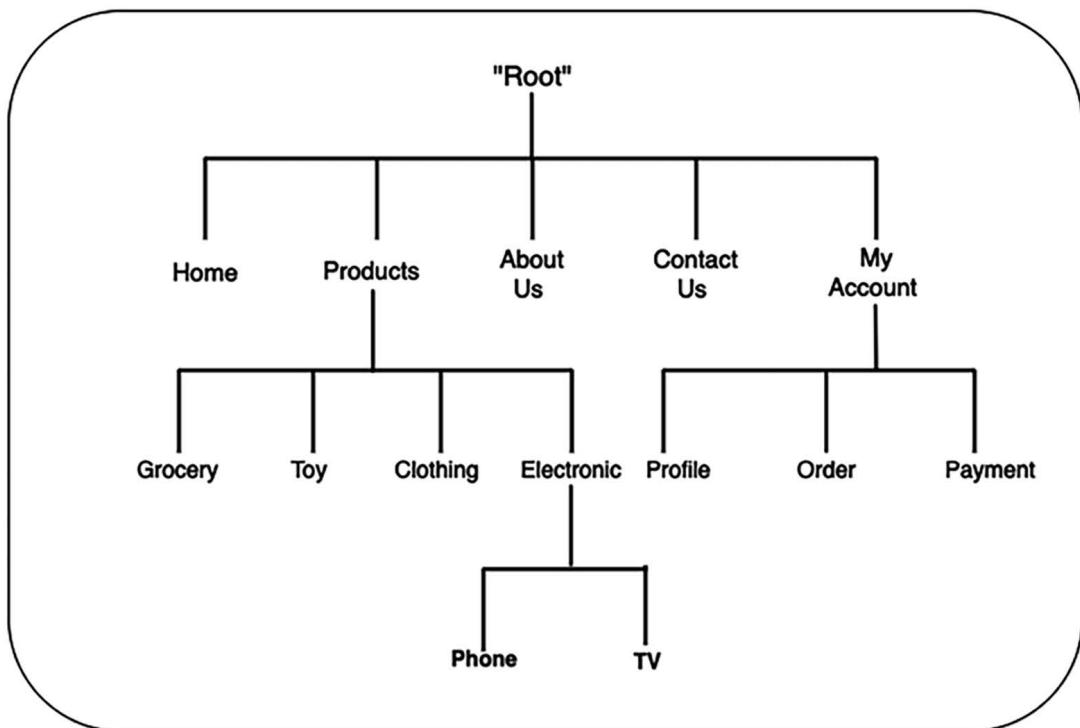


Figure 16.6: Informational architecture diagram of an e-commerce platform

Solutions architects can add more details, such as website navigation, taxonomy, or a high-level wireframe that UX designers can use to generate a detailed wireframe.

Application architecture

This section targets the development team. It provides more implementation details for a software architect or development team to build a detailed design. The following diagram shows the application architecture for an e-commerce website, with technology building blocks such as caching, networking, content distribution, and data storage:

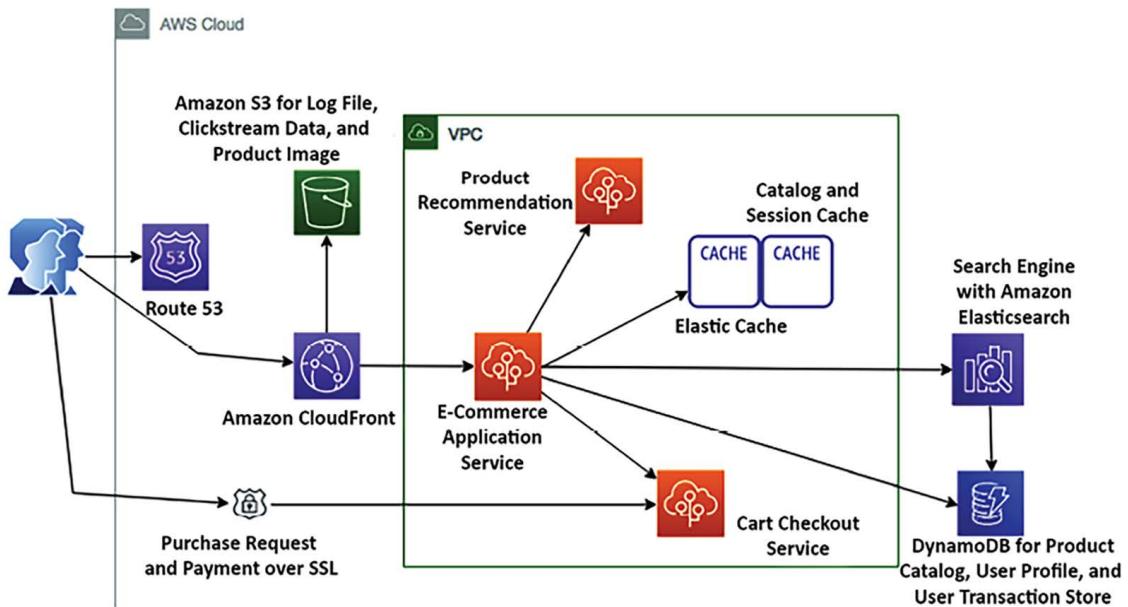


Figure 16.7: Application architecture diagram of an e-commerce platform

As shown in the preceding diagram, you need to list the components of the application architecture required for a cloud-based e-commerce platform to deliver the desired online shopping experience. These components include:

- **User interaction:** Customers interact with the e-commerce platform through a web interface, starting with a secure purchase request made over the **Secure Sockets Layer (SSL)** for encrypted communications.
- **Content delivery:** Amazon CloudFront, a **content delivery network (CDN)**, serves static content such as images, stylesheets, and client-side scripts efficiently to the user. It reduces latency by caching content closer to the user's location.
- **Domain Name System (DNS):** Amazon Route 53 is utilized for DNS management, directing user requests to the most appropriate endpoints, like the CloudFront distribution or the application load balancer.
- **Application processing:** Inside the **Virtual Private Cloud (VPC)**, the e-commerce application service processes dynamic requests, such as page rendering based on user profiles and shopping history. A product recommendation service within the VPC offers personalized product suggestions to users based on their behavior and preferences.

- Caching mechanism:** Amazon ElastiCache is employed to speed up data retrieval by caching frequently accessed data, like session state and frequently viewed product information. This reduces the load on the backend databases and improves the application's response time.
- Data storage and processing:** A cart checkout service manages user cart interactions and transactions. Catalog and session cache data is stored for quick access. A search engine built with Amazon Elasticsearch provides powerful search capabilities across the product catalog.
- User profile and transaction data:** User profile information, along with transaction data, is stored in Amazon DynamoDB, offering fast and scalable NoSQL database capabilities.
- Data logging:** Amazon S3 is used for logging data such as clickstream data, product interactions, and system logs, enabling in-depth analysis and insights into user behavior and system performance.

This section lists all application modules that need to be retired, retained, replatformed, and transformed for an application modernization architecture.

Data architecture

This section is primarily utilized by the database admin and development team to understand database schemas and how tables are related. This section often includes an **entity-relationship diagram (ERD)** showing the relationships of entity sets stored in a database, such as the following:

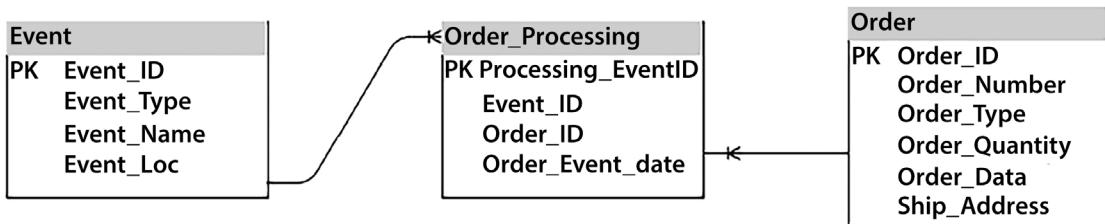


Figure 16.8: ERD of an e-commerce platform

An ERD is a visual representation of entities (which typically correspond to database tables) and the relationships between them. It's a graphical tool used in database design to illustrate the logical structure of databases. The preceding diagram is an example of an ERD for an order processing system. The ERD illustrates the relationship between events that occur and the orders that are processed within the system. Here are the components and their relationships:

- Event entity:** Represents an occurrence or action within the system, with attributes like Event_ID (primary key), Event_Type, Event_Name, and Event_Loc. These attributes might detail what the event is, where it took place, and other characteristics.
- Order entity:** Represents a customer order, with attributes including Order_ID (primary key), Order_Number, Order_Type, Order_Quantity, Order_Date, and Ship_Address. These attributes store information specific to each order, such as the amount ordered, shipping details, and when the order was placed.

- Order_Processing entity:** This serves as an associative entity (or junction table) that connects Events to Orders, indicating that an Event will lead to Order processing. It has its own primary key (Processing_EventID) and includes the foreign keys Event_ID and Order_ID, which link to the Event and Order entities, respectively. The presence of the Order_Event_date attribute suggests that it also records when an Event resulted in an Order being processed.

The lines between the entities represent the relationships. In this ERD, the “crow’s feet” notation at the ends of the relationship lines indicates “many,” while the single line indicates “one,” denoting the cardinality of the relationships.

The data architecture section lists all data objects that need to be considered during application development.

Integration architecture

Integration architecture refers to the expanded framework that allows different software applications, systems, and services to communicate and work together effectively. It involves designing and implementing methods and middleware that facilitate the exchange of data and processes across diverse systems within an organization or between an organization and external parties. This section mainly targets vendors, partners, and other teams. For example, the following diagram shows all the integration points with other systems in an e-commerce application:

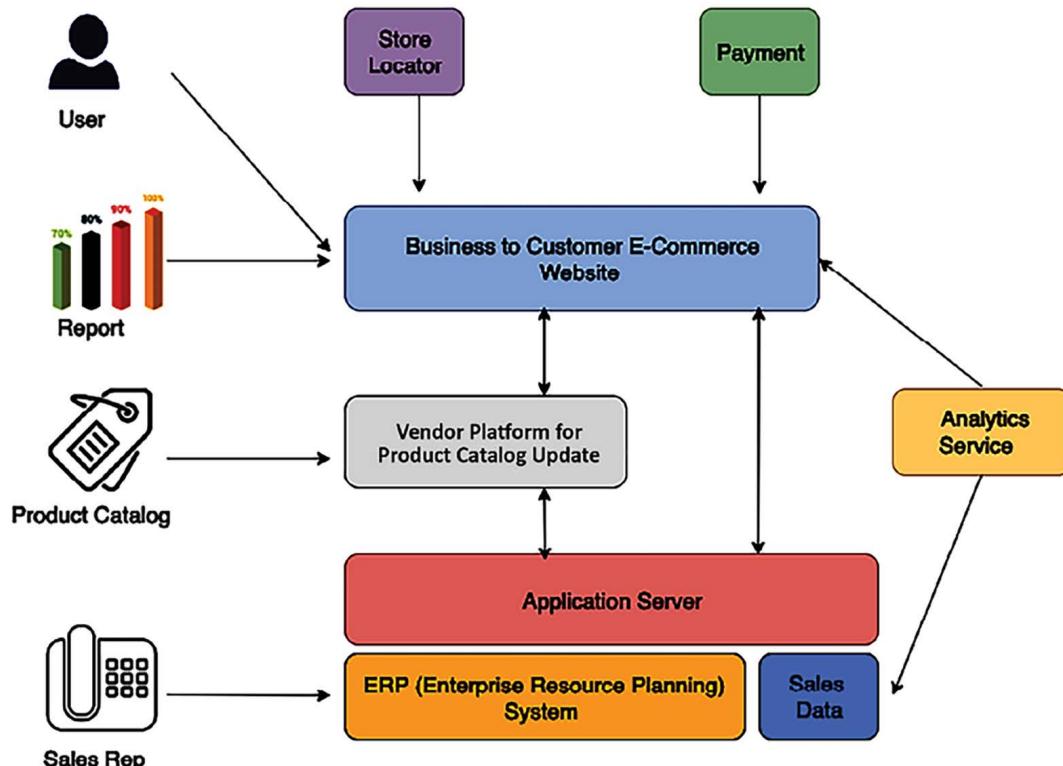


Figure 16.9: Integration architecture diagram of an e-commerce platform

The integration architecture section lists all upstream systems that provide data to your application. Any platforms, services, or databases that your application receives data from are listed here, along with the nature of the data flows and details of the downstream systems to which your application sends data. This could include other applications, databases, or services that rely on data produced or processed by your application. You need to list all system dependencies regarding your application.

Infrastructure architecture

This section is primarily targeted at the infrastructure team and system engineers. The solutions architect must include a deployment diagram outlining the logical server location and dependencies.

For example, the following figure illustrates the production deployment diagram for an e-commerce application. You can produce a separate diagram for other environments, such as the dev, quality assurance (QA), and User Acceptance Testing (UAT) environments:

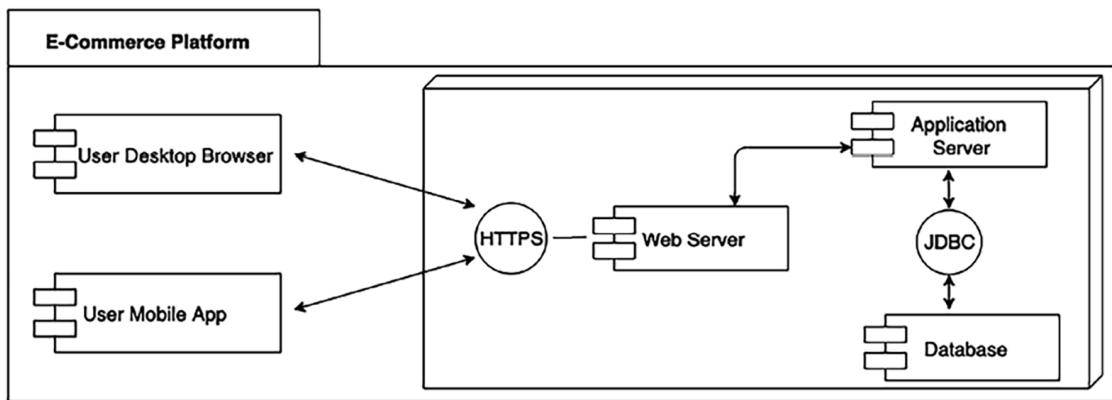


Figure 16.10: Deployment diagram of an e-commerce platform

This section lists all server configurations, databases, networks, and switches to deploy the application.

Security architecture

This section includes all the security and compliance aspects of the application, including:

- **Identity and access management (IAM)** such as Active Directory (AD), user authentication, and authorization management
- Infrastructure security such as the firewall configuration, **intrusion prevention system (IPS)/ intrusion detection system (IDS)**, and antivirus software
- Application security such as **web application firewalls (WAFs)** and **distributed denial of service (DDoS)** protection
- **Data security** at rest and in transit using SSL, encryption algorithms, key management, and so on

Overall, the solutions architect can include an application security threat model to identify potential vulnerabilities, such as **cross-site scripting (XSS)** and **SQL injection (SQLi)**, and plan to protect the application from any security threat.

Solution implementation

The solution delivery section includes essential considerations to develop and deploy a solution. It can consist of the following major subsections:

- **Development:** This section is essential for the development team. It discusses development tools, programming language, code repository, code versioning, and branching, along with the rationale behind the choices.
- **Deployment:** This section mainly focuses on DevOps engineers and talks about the deployment approach, deployment tools, various deployment components, and deployment checklist, along with the rationale behind the choices.
- **Data migration:** This section helps the team to understand the data migration and ingestion approach, the scope of data migration, various data objects, data ingestion tools used, sources of data and data formats, and so on.
- **Application decommissioning:** This section lists existing systems that need to be decommissioned and an exit strategy for the current system if the **return on investment (ROI)** is not realized. The solutions architect must provide an approach and timeline for decommissioning the old system and assess the overall impact.

The SAD includes a development approach and tools. However, it does not have a detailed application-level design, such as a class diagram or pseudocode. Such details need to be handled by the software architect or senior developer under the corresponding software application details design document. As a solution gets deployed, it needs to be managed in production. Let's learn about the details that go into the solution management section.

Solution management

The solution management section is focused on production support and ongoing system maintenance across other non-product environments. This section covers the operational aspects of the solution, including monitoring, incident management, user onboarding, and support and recovery processes. The solution management section is primarily targeted at the operations management team. This section addresses the following areas:

- Operational management, such as system patching and upgrades of dev, test, staging, and prod environments
- Tools to manage application upgrades and new releases
- Tools to manage system infrastructure
- System monitoring and alerts; operations dashboard
- Production support, SLA, and incident management
- Disaster recovery and **business process continuation (BPC)**

A solutions architect must research and collect data to validate the right solution during design. Such additional details can be put in the Appendix section. Let's learn more about the Appendix section of a SAD.

Appendix

Like every business proposal document, the SAD also has an open Appendix section containing any data supporting your overall architecture and solution choices. In this section, the solutions architect can include open issues and any research data, such as the outcome of the POC, tool comparison data, and vendors' and partners' data.

This section on the structure of the SAD gave you a good overview of the SAD structure along with different sections. A SAD should include the major sections mentioned previously; however, the solutions architect may exclude some sections or include others to meet specific requirements of the organization or project. As with other documents, it's essential to continue to iterate upon SADs and look for opportunities to improve. More robust SADs lead to well-defined implementation guidelines and reduce any risk of failure.

A SAD is a running document created during the initial stages and updated over the years based on various changes throughout the application life cycle. Let's now look into the life cycle of the SAD.

Life cycle of the SAD

The life cycle of a SAD aligns with different phases of a project's life cycle. Previously in this chapter, we explored the various sections of the SAD, and these are each created at distinct stages. The life cycle of a SAD typically includes the following phases:

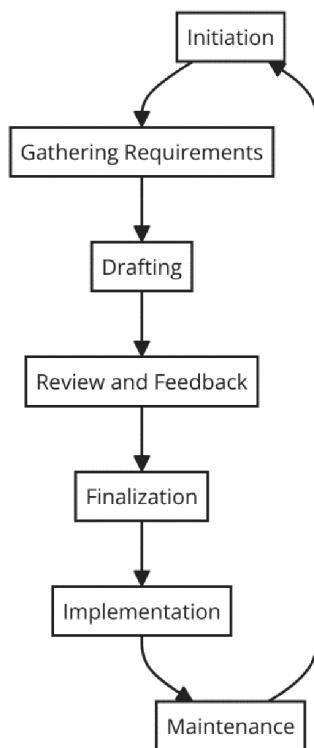


Figure 16.11: SAD life cycle

Let's look at the phases in the SAD life cycle, shown in the preceding diagram, in more detail:

1. **Initiation:** The initiation phase is where the need for a SAD is recognized, usually at the project's conception. This phase defines objectives, such as outlining the software architecture for a new enterprise application. This step sets the direction and scope for the document, ensuring it aligns with the project's goals and stakeholder expectations.
2. **Gathering requirements:** In this crucial phase, detailed requirements are collected from key stakeholders. For a retail e-commerce platform, this might involve gathering insights on user experience, payment processing, and inventory management. This phase ensures that the SAD addresses all critical aspects of the project.
3. **Drafting:** The drafting phase involves creating the initial version of the SAD. This document might detail the architecture of a cloud-based system, specifying the use of certain AWS services, database schemas, and security protocols. This draft serves as the blueprint for the project's technical implementation.
4. **Review and feedback:** Once the draft is prepared, it's shared with stakeholders for review. In the case of a healthcare management system, clinicians, IT staff, and administrative personnel might provide feedback on various aspects like patient data flow and compliance with health regulations.
5. **Finalization:** After incorporating feedback, the SAD is finalized. This might involve adjusting a network architecture to improve data flow efficiency based on stakeholder input.
6. **Implementation:** The finalized document guides the project's implementation. For instance, developers and project managers use the SAD to ensure alignment with the planned architecture and technology stack in a software development project.
7. **Maintenance:** Over time, the SAD is revisited and updated to reflect technological changes, business objectives, or external factors. For example, introducing new data privacy laws might necessitate updates to the SAD to ensure continued compliance.

Let's now look at some of the best practices and pitfalls to avoid when producing SADs.

SAD best practices and common pitfalls

Effective management of SADs involves certain best practices. Keeping the document clear and concise is crucial, making it easily understandable while avoiding technical jargon. Regular involvement of stakeholders in the process ensures that the document aligns with business and technical requirements. Keeping the SAD up to date with the latest project changes and developments is vital for its relevance and usefulness. Additionally, aligning the architecture outlined in the SAD with the organization's broader business goals is essential, ensuring that the proposed technical solutions support and enhance the business objectives.

There are also common pitfalls to avoid when working with a SAD. Overcomplicating the architecture can lead to challenges in implementation and maintenance. A lack of flexibility in the document can hinder its ability to adapt to changes in project scope or objectives. Insufficient stakeholder engagement can result in a misalignment with actual business needs and requirements.

Moreover, poor documentation practices can lead to misunderstandings, implementation errors, and challenges in project execution. It's essential to address these pitfalls proactively to ensure the success of the solution architecture.

In addition to the SAD, solution architecture often involves a significant procurement proposal with a specific requirement known as a **request for x (RFx)** document. Let's become familiar with RFx documents.

IT procurement documentation for a solution architecture

IT procurement documents are popularly known as **RFx documents**. This is a term that includes different stages of the procurement process. RFx refers to a formal requesting process. RFx documents are categorized as **request for proposal (RFP)**, **request for information (RFI)**, and **request for quotation (RFQ)** documents.

Solutions architects are often involved in the procurement process, either leading the process or providing their input. These procurements may be related to outsourcing, contracting, procuring software such as a database or development tools, or buying SaaS solutions.

As these documents can be highly technical and will likely have a broad, long-term impact, the solutions architect needs to provide input by responding to any procurement requirements.

Let's understand the difference between the different RFx documents:

- **RFI:** RFI comes early in the procurement process, where buyers invite information from different vendors to make an informed decision regarding their choice of procurement for a later stage. An RFI document collects information about the capabilities of the various suppliers, where the buyer can compare all suppliers in similar parameters and proceed to the next proposal steps with shortlisted suppliers. For example, let's say a company is exploring the market to identify **learning management systems (LMSs)** that meet its training needs. The company issues an RFI to collect detailed information about the functionalities, integration capabilities, and user experiences of the LMS platforms available.
- **RFP:** In this process, suppliers shortlisted from the RFI process get more information about the project's outcome. An RFP document is more open-ended than an RFI one, where suppliers can provide the best way to acquire solutions for the buyer. The supplier can include multiple choices, with the pros and cons of each approach. For example, say a government agency wants to upgrade its IT infrastructure. It issues an RFP outlining its current system, the improvements required, and the performance criteria. Vendors respond with proposals that include technical solutions, project timelines, and cost estimates.
- **RFQ:** In this process, buyers narrow down the requirements compared to the RFP and list the exact requirements of work, equipment, and supplies. Suppliers must provide a cost for the listed requirements, and the buyer can choose the best quote to award the contract to. For example, say a technology startup is preparing to scale its IT infrastructure to accommodate a growing user base.

The startup needs to procure additional cloud computing resources to handle the increased load. It has already determined its requirements in terms of compute instances, memory, storage, and bandwidth. The RFQ requests that IT infrastructure providers submit their best quotes for monthly and yearly pricing options, along with any discounts for long-term commitments. The startup also asks for information on additional services such as support, monitoring, and security features that could be included in the quoted price.

The RFP is the most popular choice, as often, to speed up the process, the buyer organization often chooses to only ask for RFP documents from potential vendors. In such a situation, the RFP document needs to use a clear structure so that the buyer can compare preferred vendors in terms of capabilities, solution approaches, and costs to make a quick decision.

Due to the technicalities of procurement in IT organizations, solutions architects play an essential role in evaluating vendors' capabilities and approaches from the buyer's perspective and responding to RFP documents from the supplier side. During the RFP process for an IT workload, a solutions architect's expertise is critical. Here's how they might contribute:

1. **Understanding requirements:** The solutions architect begins by thoroughly understanding the project's technical and business requirements, which may include enhancing existing systems, migrating to the cloud, or integrating new technologies.
2. **Designing the solution:** They draft a preliminary design of the IT architecture that addresses the identified requirements. This includes selecting appropriate technologies, designing the infrastructure layout, and considering integration with existing systems.
3. **Collaborative contribution:** The architect often collaborates with cross-functional teams, including business analysts, project managers, and technical leads, to ensure that the proposal aligns with both business objectives and technical feasibility.
4. **Estimating resources:** They estimate the resources required for the project, which may include hardware, software, cloud services, and personnel hours, ensuring the proposal is both competitive and realistic.
5. **Risk assessment:** Solutions architects identify potential risks in the project and propose mitigation strategies to include in the RFP response.
6. **Documentation:** They help create detailed technical documentation that explains how the proposed solution meets the RFP requirements. This documentation often includes system diagrams, data flow diagrams, and detailed descriptions of the proposed environment.
7. **Pricing strategy:** Solutions architects may work with the finance team to develop a pricing strategy for the proposal, ensuring the costs are in line with the value delivered.
8. **Presentation:** Solutions architects may be part of the team that presents the proposal to the potential client, explaining the technical aspects and answering any technical queries they may have.

The role of a solutions architect is crucial for tailoring a solution that not only meets the client's requirements but also ensures technological viability and cost-effectiveness. For instance, if a company is soliciting proposals for a new cloud-based CRM system, the solutions architect would first analyze the existing IT infrastructure and assess the necessary features such as scalability and data security.

They'd design a cloud solution that seamlessly integrates with current systems like ERP and aligns with the marketing strategies. By collaborating with potential vendors, the architect ensures the chosen CRM platforms align with the company's specific needs and evaluates their compatibility. Additionally, they would work closely with legal teams to ensure the solution meets stringent compliance and data management requirements.

A key task is creating a migration strategy to transition data from old systems to the new CRM, aiming for minimal business disruption. This also involves estimating the total cost of ownership and considering factors like subscription costs, customization, data transfer, and training needs. The solutions architect has a hand in drafting the technical segments of the RFP response, detailing the proposed architecture, data strategy, and security measures. They would outline the implementation roadmap, addressing how the system will scale over time and what support structure will be in place post-deployment.

Moreover, the solutions architect presents and defends the technical strategy to the client's decision-makers, illustrating the benefits and practicality of the proposed solution. This ensures that the client invests in a reliable, secure, and scalable system that supports the business's growth and adapts to evolving market demands.

Summary

A SAD aims to keep all stakeholders on the same page and get formal agreement on solution design and requirements. As stakeholders comprise both business and technical users, you learned about various SAD views that the solutions architect needs to consider. You must include views for non-technical users, such as business, process, and logical views. For technical users, include applications, development, deployment, and operational views.

In this chapter, you learned about the detailed structure of the SAD, with major sections and subsections.

Various sections of the SAD include details such as an overview of the solution, business, and conceptual architecture. You also learned about various architecture views in the architecture diagram, such as application, data, infrastructure, integration, and security. You also learned about other sections of the SAD, covering solution delivery considerations and operations management.

It has been a long journey of learning. You are almost at the end of the book, but before closing, we will provide some tips for becoming a solutions architect and improving your knowledge.

In the next and final chapter, you will learn various soft skills such as communication styles, ownership, critical thinking, and continuous learning techniques to help you become a better solutions architect.

Join our book's Discord space

Join the book's Discord workspace to ask questions and interact with the authors and other solution architecture professionals: <https://packt.link/SAHandbook>



17

Learning Soft Skills to Become a Better Solutions Architect

In the previous chapters, you learned how a solutions architect needs to accommodate all stakeholders' needs. Even if the solutions architect's role is technical, they need to work across the organization, from senior management to the development team. Soft skills are essential and critical factors to become a successful solutions architect.

Solutions architects should keep updated with current technology trends, keep evolving their knowledge, and always be curious to learn new things. You can become a better solutions architect by applying continuous learning. You will discover methods to learn new technologies in this chapter and how to share and contribute back to the technical community.

Solutions architects need to define and present an overall technical strategy to address business concerns. They need to work across business and technical teams to negotiate the best solution, which requires excellent communication skills. In this chapter, you will learn the soft skills a solutions architect must have, including communication:

- Importance of soft skills in solution architecture
- Acquiring pre-sales skills
- Taking ownership and accountability
- Being flexible and adaptable
- Design thinking
- Being a builder by engaging in coding hands-on
- Becoming better with continuous learning
- Being a mentor to others
- Becoming a technology evangelist and thought leader

By the end of this chapter, you will know about the soft skills required for a solutions architect to succeed. You will learn about methods to acquire strategic skills (such as pre-sales and executive communication) and develop design thinking and personal leadership skills (such as thinking big and ownership). You will learn techniques to establish yourself as a leader and continue improving your skill set.

Importance of soft skills in solution architecture

The importance of soft skills in solution architecture cannot be overstated, as these skills are integral to the effectiveness and success of a solutions architect.

Firstly, effective communication is paramount. Solutions architects must simplify complex technical details for non-technical stakeholders, ensuring clarity and alignment between technical and business objectives. This skill is vital in bridging the gap between technical teams and business units, facilitating mutual understanding and collaborative goal setting.

Collaboration and teamwork also play a critical role. Given that solutions architects often work in multidisciplinary teams, the ability to work harmoniously with individuals from diverse backgrounds and expertise is crucial. This collaboration extends to conflict resolution, consensus building, and fostering an environment of shared responsibility and collective success.

Furthermore, problem-solving and critical thinking are essential soft skills for solutions architects. The nature of their work often involves navigating complex technical challenges and finding innovative solutions that align with business strategies. This requires technical understanding, creativity, analytical thinking, and a solutions-oriented mindset.

Lastly, leadership and adaptability are essential. Solutions architects frequently take on leadership roles in project teams, guiding technical direction and making pivotal decisions. This requires technical leadership and the ability to inspire, motivate, and guide teams toward a common vision. Additionally, the tech landscape is constantly evolving, and adaptability and a learning mindset enable solutions architects to stay current and responsive to new trends and technological shifts. These soft skills and technical expertise make solutions architects invaluable assets to any organization.

In the following sections, we look at some key soft skills a solutions architect should possess, within and related to the overarching skills mentioned here.

Acquiring pre-sales skills

Pre-sales is a critical phase for complex technology procurement, whereby the customer collects detailed information to make a buying decision. In the customer organization, a solutions architect is involved in the pre-sales cycle to validate technology and infrastructure resources from various vendors. In the vendor organization, the solutions architect needs to respond to customers' requests for proposals (RFPs) and present a potential solution to acquire new business for an organization. Achieving this requires a specific skill set.

Key skills

Pre-sales requires a unique skill set that combines strong technical knowledge with soft skills, including the following:

- **Communication and negotiation skills:** Solutions architects need excellent communication skills to engage customers with the correct and latest details. Presenting precise details of the solution and industry relevance helps customers understand how your solution can address their business concerns. Solutions architects work as a bridge between the sales and technical teams, which makes communication and coordination a critical skill. Solutions architects must also create agreements by collaborating with customers and internal teams, which requires excellent negotiation skills. In particular, strategic-level decisions have a significant impact across multiple groups. Solutions architects need to negotiate between the teams, work on trade-offs, and develop an optimized solution.
- **Listening and problem-solving skills:** Solutions architects need strong analytical skills to identify the right solution per customer need. It's important solutions architects listen to and understand customer use cases by asking the right questions to create a good solution. Solutions architects need to understand gaps and develop a solution for immediate business impact with long-term **return on investment (ROI)**. Performance is more important for some customers, while others may be more focused on cost based on their application's user base. The solutions architect must provide the right solution per their customer's primary **key performance indicator (KPI)** goal.
- **Customer-facing skills:** The solutions architect often needs to work with both the internal and external customer teams. They influence stakeholders at all levels, from C-level executives to development engineers. They present solutions and demos to senior management, who look at your proposal from a business perspective. C-level executive support and commitment to initiatives always result in the success of the adopted solution, which makes customer-facing skills very important. The C-level executive needs details of the solution in a defined time-bound meeting, and the solutions architect needs to utilize the allotted time to their best advantage. You will learn more about the executive conversation in the next section of this chapter—*Presenting to C-level executives*.
- **Working with teams:** The solutions architect establishes a relationship with the business and product teams. To prepare an optimal application, the solutions architect must work with the business and technical teams at all levels. The solutions architect needs to be a good team player, share ideas, and find a way to work across teams.

The skills mentioned above are required for pre-sales and applicable to the solutions architect's day-to-day job functions. Solutions architects come from a technical background, and, being in such a role, they need to acquire critical skills to communicate at an executive level. Let's learn more about executive conversations in the next section.

Presenting to C-level executives

A solutions architect needs to handle various challenges from a technical and business perspective. One of the most challenging tasks can be getting executive buy-in.



Senior executives such as the **Chief Executive Officer (CEO)**, **Chief Technology Officer (CTO)**, **Chief Financial Officer (CFO)**, **Heads of Line of Business (LoBs)**, and **Chief Information Officer (CIO)** are regarded as C-level as they have a tight schedule and need to make lots of high-impact decisions.

As a solutions architect, you may have many details to present, but your C-level meetings are time-bound. A solutions architect needs to get the maximum value from their meeting in the allotted time slot. The primary question is: *How do we get senior executives' attention and support in a limited time?*

Start by explaining your agenda and planned meeting structure. Executives ask lots of questions to utilize their time properly, and your agenda should convey that they will get the chance to ask clarification questions. Following that, the key to an executive presentation is summarizing the primary points in the first 5 minutes. You should prepare so that if your 30-minute slot is reduced to 5 minutes, you should still be able to convey your points and get buy-in for the next step.

Support your summary with facts and data that align with their industry and organization. Keep the details with you in case they want to dive deep into a particular area; you should be able to pull up and show all the relevant data.

Don't present everything in detail by stating information that may seem relevant from your perspective but doesn't make much sense for an executive audience. For example, as a solutions architect, you may focus more on the benefits of the technical implementation. However, senior management focuses more on ROI by reducing operational overhead and increasing productivity.

Tailor your demo or presentation to resonate with the customer's field and domain terminologies. Incorporating terms and concepts familiar to them not only fosters trust but also affirms that you have tailored your technical solution to address their unique industry challenges effectively.

You should be ready to answer the following questions that concern executives:

- **How will the proposed solution benefit our customers?** Business revolves around the customer. Executives look at their company's growth, which is only possible if their customers are satisfied. Make sure to do your research on their customer base and their needs. Be ready to present benefits backed by reliable data.
- **What assumption did you make to baseline the solution?** Often, these meetings are at the initial phase when you need more details. Solutions architects always need to make some assumptions to baseline the solution. List down your hypothesis in bullet points, and have a mitigation plan if things don't work as per your assumptions.
- **What will be my ROI?** Executives are always looking for ROI by determining the **total cost of ownership (TCO)**. Be ready with data to provide an estimated cost of ownership, solution maintenance costs, training costs, overall cost savings, and so on.

- **What happens if we continue as it is today and do nothing?** Senior management may go into extreme vetting mode to identify ROI. They want to understand if the investment is worth it. It would be best if you are ready with your market research—for example, technology trends, customer trends, and the competitive situation.
- **What will our competitor's reaction be regarding your solution?** Competition is everywhere, and often, the executive worries about it. They want to understand if your solution is innovative to beat the competition and give their organization the edge. It's better to do some upfront research and add competitive data relating to their industry and customer base.
- **What is your suggestion, and how can I help?** You should always have a crisp list of action items as the next step while providing your suggestions. You need to get buy-in from executives and make them feel involved by asking for help. For example, you can ask the CIO to connect you with the engineering or product teams to take an overall solution to the next step.

Let's next look at the leadership skills a solutions architect should have as a technical leader for the organization.

Taking ownership and accountability

Taking ownership and positioning yourself as a leader helps you to win trust with accountability. Ownership doesn't mean you need to execute things alone; it is more about taking new initiatives and holding on to them for your organization. You can have ideas that can benefit your organization regarding productivity, agility, cost savings, and increasing the customer base. Sometimes, you may need more time or resources to execute your idea, but you should bring it forward as a new initiative and engage others for execution.

Accountability is about taking responsibility to drive the outcome. Ownership and accountability go hand in hand, where you are creating initiative and working on getting the result. People can trust you to execute any job and drive results. Accountability helps you build trust with your customers and team, resulting in a better work environment and goal achievement.

As a solutions architect, taking ownership helps you see things from the customers' and sponsors' perspectives. You feel motivated and a part of something meaningful that you enjoy doing. Make sure to define and create key successes and the objective key result. The goal/objective should be measurable using specific key results, and they must be time-bound. Let's learn more about **objectives and key results (OKRs)**.

Defining strategy execution with OKRs

Strategy execution is complex and challenging. Excelling in strategy execution is essential for realizing the organizational vision, mission, and goals. The idea needs to be converted into actionable elements to keep teams aligned and everyone moving in the same direction. Goal setting and managing goals are some of the best-established ways to get things done.

OKRs are principles and practices (vision and execution) of goal setting. An OKR is a strategy management system that focuses on strategy execution, a simple framework that lets you define the organization's primary strategy and priorities. Objectives are the principles, and key results are the practice—a *what and how* of organizational vision. OKRs are based on four superpowers, as illustrated in the following diagram:

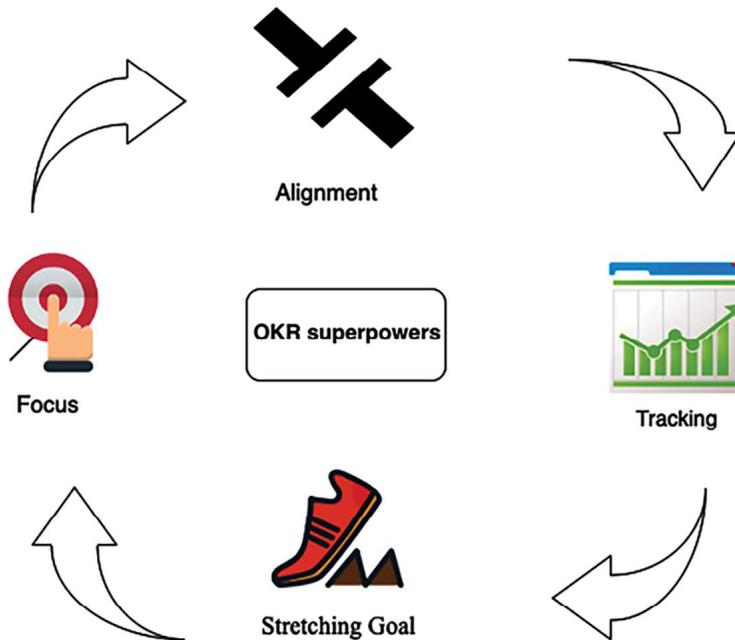


Figure 17.1: Superpowers of OKRs

OKRs' superpowers include the following:

- **Focus:** Start with the question: *What are our main priorities, and where should people concentrate their efforts?* Commit to what truly matters and provide clarity on what is essential.
- **Alignment:** Make goals public and transparent. Connect with the team and get cross-team, bottom-up, and sideways alignment.
- **Tracking:** Visually track the key results of each objective down to the percentage point.
- **Stretching Goal:** Create ambitious goals to achieve something remarkable. Stretching goals allows people to reimagine and rethink.

OKRs set clear, measurable goals and align them with the organization's strategic mission. In solution architecture, OKRs can guide the design and implementation of systems to ensure they contribute to the business's overarching goals.

For a solutions architect, OKRs could involve something like this:

Objective: Improve the system's resilience and fault tolerance.

Key Results:

- Decrease system downtime by 30% within the next quarter.
- Implement a multi-zone deployment strategy for critical services by the next release cycle.
- Achieve 99.99% availability for the user-facing application layer within six months.

Here, the architect uses OKRs to set clear targets for system performance and reliability, which are crucial aspects of their responsibilities. They also help prioritize tasks, measure progress, and communicate the impact of architectural decisions to stakeholders.

OKRs provide visibility and a meaningful outcome to all stakeholders at various levels, from executive sponsors to teams. OKRs make the vision and mission of the organization clear. Team members working on day-to-day activities need visibility and clarity of the mission. They need to see how their everyday work impacts that. The OKR framework allows you to define this link and provide visibility and meaning for everyone on the team.

Thinking big

Solutions architects should be able to see the big picture and think ahead. A solutions architect creates a foundation upon which the team puts building blocks and launches the product. Thinking big is one of the critical skills that solutions architects should possess to consider an application's long-term sustainability.

Thinking big doesn't mean you need to set a very unrealistic goal. Your goal should be big enough to challenge you and bring you out of your comfort zone.

Thinking big is about predicting the organization's needs and staying ahead of technological advancements, ensuring that today's designs can adapt and remain effective in the future. Thinking big is critical for success at both a personal and an organizational level.

You should always be confident in your capability while thinking big. It may seem challenging to achieve initially, but you will find the way as you start working toward the goal. Believe in yourself, and you will notice that others start supporting and believing in you. Thinking big helps to inspire people around you to become a part of your success. Set long-term goals, such as *where you want to see yourself and your organization in the next decade?*. Take one step at a time to gear a short-term goal to a long-term goal.

Once you set up the stretching goal by thinking big, it will help you take the initiative and explore new challenges. However, it's best if you have support from your peers and team to deliver the result, who can provide you with the right feedback and extend help as needed. Become someone people want to help; this is a two-way door. To get help, you need to be open to helping others. Adaptability is another critical skill for solutions architects to work with others. Let's learn more about it.

Being flexible and adaptable

Adaptability and flexibility go hand in hand, and as a solutions architect, you must be flexible to adapt to new environments, working cultures, and technologies. Adaptability means you are always open to new ideas and to working with appropriate teams.

Teams may adopt a process and technology that is best suited for them. As a solutions architect, you must be flexible in accommodating team requirements during solution design. For example, in a microservices architecture, each service communicates via a standard RESTful API over the HTTP protocol. Teams may write code in a different language or tool, such as Python, Java, Node.js, or C#. The only requirement is that teams expose their APIs securely so that the entire system can build upon utilizing them.

You need different mindsets and perspectives to look into a problem and get a more innovative solution. Encouraging teams to fail fast and innovate helps an organization to be competitive.

Personal traits of flexibility are demonstrated by the following:

- Thinking about various solutions to solve a problem with the team and taking the best approach
- Helping team members to offload their work
- Volunteering to fill up a place if a team member needs to take time off for weeks due to personal reasons
- Being able to collaborate effectively with teams across different locations and time zones

You need to be open-minded and adaptable to changes in technology and processes. You may face resistance when bringing change to your team or organization. You need to encourage others to be flexible and convey the importance of change. For example, when an organization wants to move its workload from on-premises to the cloud, it often needs more support, as people have to learn a new platform. You need to explain the value proposition of the cloud and how it will help them be more agile and innovate faster.

As a solutions architect, you must be adaptable to multiple assignments and set the proper execution priority. You should be able to adjust to the situation and work under pressure.

A solutions architect needs critical design thinking to create an innovative solution. Let's learn more about design thinking in the next section.

Design thinking

A solutions architect has the primary role of system design, which makes design thinking an essential skill. Design thinking is one of the most successful approaches adopted across industries to solve challenging and unclear problems. Design thinking helps you to look at problems and solutions from a different perspective, which you might have yet to consider in the first instance. Design thinking focuses more on delivering results by providing a solution-based approach to solving a problem. It helps to question the problem, solution, and associated risk to develop the most optimized strategy.

Design thinking helps you redefine problems in a more human-centric way by putting yourself in the place of end users and customers. The following diagram illustrates the primary principles of design thinking:

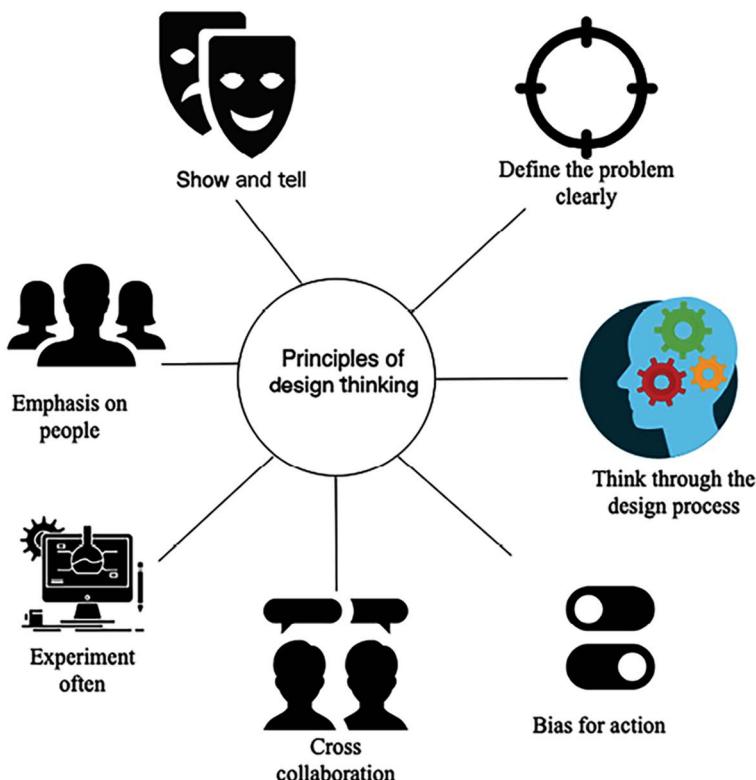


Figure 17.2: Principles of design thinking

The following points are some design-thinking principles:

- **Experiment often:** Create a prototype to understand the idea's implementation in real-life situations. Adopt a fail-fast strategy and experiment more often.
- **Cross-collaboration:** Bring in people from different backgrounds to look for problems in a diversified way and make sure solutions accommodate everyone's needs.
- **Emphasis on people:** Collect feedback from various users and put yourself in their place to understand the problem from a different perspective.
- **Show and tell:** Present your thoughts in visuals so they are easier to grasp for everyone in the room.
- **Define the problem clearly:** Create a well-defined and clear vision for a given challenge, which can help others understand clearly and encourage them to contribute.
- **Think through the design process:** Understand the overall design process with clear goals and methods.
- **Bias for action:** The ultimate design to deliver a solution rather than just thinking. Be proactive in pushing forward and creating activities that can result in a workable solution.

Design thinking provides a solid foundation for applying empathy and creating a holistic view of the problem. To adopt design thinking, there is a five-phase model proposed by d.school (<https://dschool.stanford.edu/resources/getting-started-with-design-thinking>). They are pioneers in teaching and applying design thinking. The following diagram illustrates the five phases of design thinking:

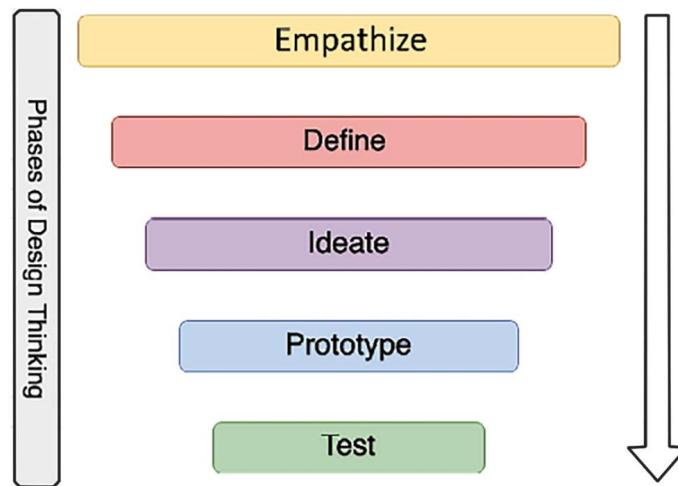


Figure 17.3: Five phases of design thinking

Design thinking is an iterative approach that needs to evolve continuously. The output from one phase can recursively be input to other phases until the solution gets solidified. A brief overview of the phases follows:

- **Empathize:** Empathy is the building block and foundation of design in the human context. To empathize, you should observe your users' behaviors and engage with them to understand the actual issue. Try to immerse yourself in—and experience—the problem by putting yourself in the situation.
- **Define:** Empathizing helps define the problem as you experience the user's needs and the problem they face. In the define phase, you apply your insight and define the problem clearly, which can fuel brainstorming to find an innovative yet simple solution.
- **Ideate:** The ideation phase is about moving from problem to solution. You work with the team to find various alternative solutions by challenging assumptions. You need to get the obvious solution out of your head and work collaboratively to find all possible solutions, which allows for innovation.
- **Prototype:** The prototype phase helps to convert ideas into concrete solutions. Prototyping can provide lots of learning and help resolve disagreements by showing a **proof of concept (POC)**. It helps you to find gaps and risks. You should build a quick prototype without lots of investment, which allows you to handle failure and increase learning.
- **Test:** The test phase is about getting feedback on your solution and reiterating it accordingly. The test phase helps you to redefine the solution and learn more about your users.

Design thinking accommodates all the phases to develop a logical and practical solution. When designing application architecture, you can relate the phases and principles of design thinking to your real life. There is particular stress on prototyping, as that is the only way to solidify your proposal and existing solutions with data and facts. A solutions architect's primary job is to understand the business concern and create a technical solution design with a prototype the team can implement. To build a prototype, the solutions architect must get their hands dirty and engage in hands-on coding. Let's learn more about it.

Being a builder by engaging in coding hands-on

A solutions architect is a builder who learns by doing. Building a prototype is worth a thousand pictures. It helps to reduce miscommunication and ideate solutions. Presenting a POC and prototyping are integral parts of the solutions architect's role.

Prototyping is the pre-solution phase, which helps to deepen your understanding of the application design and user. It helps you to think about and build multiple solution paths. By testing a prototype, you can refine your solution and inspire others, such as teams, customers, and investors, by demoing your vision.

A solutions architect is a technical leader who works closely with the development team. In an empowered agile team of developers, a solutions architect needs to show a piece of code as a POC in addition to a PowerPoint presentation. A solutions architect doesn't need to be part of the development team but works collaboratively to convey the solution to the dev team in their language. Successful delivery is only possible if the solutions architect can understand the deep technical aspect of a solution that comes with continuous coding and hands-on experience.

A solutions architect is often seen as a mentor and player-coach; having some hands-on coding helps them establish credibility. A solutions architect needs to decide which programming languages and tools the team should use. A hands-on approach helps identify gaps that may not fit your team or solution requirements—always learning new technologies enables you to make a better decision on behalf of the organization. Let's learn more about the techniques of continuous learning.

Becoming better with continuous learning

Solutions architects need to continually absorb new knowledge and enhance their skill sets to help the organization make better decisions. Continuous learning keeps your skill set relevant and builds confidence. It opens up your mind and changes prospects.

Learning can be challenging with a full-time job and a busy family life. Continuous learning is about developing the habit of always learning something new, whereby you have to be motivated and disciplined. You first need to set up learning goals and apply effective time management to achieve them. This often slips through the net when you get busy with regular daily work.

Everyone has their own style of learning. Some people may like formal education; some may read books; others may want to listen to and watch tutorials. You need to find the learning style that is most effective for you and suited to your lifestyle.

For example, you can listen to audiobooks and tutorials when commuting to work. You can read books during a business flight or watch video tutorials during exercise hours in the gym. Overall, you need to make some adjustments to put time aside from your busy work life for continuous learning. Here are some of the ways to engage yourself in constant learning:

- **Learning new technologies, frameworks, and languages by trying them out:** Solutions architects are builders ready to experiment hands-on. As a successful solutions architect, you must keep learning new technologies by building a small POC. Understanding modern programming languages and frameworks will help you provide the best advice on technology adoption for an organization and team.
- **Learning new skills by reading books and tutorials:** The traditional method of reading physical books offers significant benefits, particularly in the context of focused learning. It allows readers to disconnect from the distractions of online activity, fostering deeper concentration. This approach is especially beneficial for those who spend much of their day in front of a computer screen, as it provides a restful break for the eyes. Engaging with physical books can enhance the learning experience by reducing digital fatigue and promoting undivided attention to the material.

Similarly, there are millions of books available on Kindle to read anytime and anywhere. Audiobook platforms such as Audible and Google Play's audiobooks can help you listen to books during your commute. So many convenient resources are available that there is no excuse not to apply continuous learning.

Online learning has been revolutionary and made it easy to understand and dive deep into any area. You now have massive knowledge bases at your fingertips to learn anything. An online platform such as Udemy or Coursera provides thousands of video tutorial courses in all areas you can watch online or download to your device for offline learning.

- **Keeping up with technology news and developments by reading articles on websites and blogs:** The best way to keep yourself updated with technology trends is by subscribing to technical news and blogs. TechCrunch.com, Wired.com, and Cnet.com are popular websites where you can find the latest technology trends. Major newspapers such as CNBC, The New York Times, and the BBC News and CNN channels also have technology articles that provide good insight into industry trends. You can subscribe to blogs for new learning in respective technology areas. For example, for cloud platform learning, you can subscribe to Amazon Web Services (AWS) blogs, which have thousands of articles and use cases in the area of the AWS cloud, and similar blogs are available from other public clouds such as Azure and Google Cloud Platform (GCP).
- **Writing a blog, whitepaper, or book:** Sharing knowledge is the best way to learn as you think through use cases when trying to present to others. Publishing blogs and articles on popular blog-publishing platforms such as Medium, Blogger, and LinkedIn helps you share your learning and also learn from others. Active participation in question-and-answer platforms enables you to find an alternative solution for any given problem. Some popular question/answer platforms are Quora, Reddit, Stack Overflow, and Stack Exchange.

- **Solidify your knowledge by teaching others:** Teaching others helps you collaborate and get a different perspective on your knowledge. Use cases proposed by participants often give you different ways of finding a solution. Running a full-day workshop with a hands-on lab and concept building helps you solidify your learning and learn with others.
- **Taking online classes:** Sometimes, you want to go for formal learning to be more disciplined, and you want to be flexible. Online courses provide flexibility, help you adjust to other priorities, and save time. Online courses offer an organized way to learn new technologies and enhance knowledge.
- **Learning from teammates:** Teammates share the same working environment, and you spend most of the day with them. Learning with teammates can speed up your learning. The team can adopt a divide-and-conquer strategy whereby members can share their topics and present deep-dive lunch-and-learn sessions. These sessions are a standard method used by many organizations to conduct regular learning sessions among team members. Each team member shares their new learning in a weekly learning session, and everyone quickly learns about new topics.
- **Attending and participating in user groups and conferences:** All large vertical industry and technology organizations conduct conferences and hands-on sessions to provide insight into new technology trends. Participating in industry conferences and user group meetings helps to develop networking and understand technology trends. Some of the large technology conferences from industry leaders include AWS re:Invent, Google Cloud Next, Microsoft Ignite, SAP SAPPHIRE, and Strata Data Conference. You could also create a local user group and conduct a meetup in your local area, which will help you to collaborate with professionals across industries and organizations.

Continual professional development is essential for a solutions architect, with technology's rapid evolution necessitating routine updates to credentials and certifications to stay current with new advancements and industry standards.

A solutions architect plays a technical leadership role, and good leadership warrants preparing more leaders like you, which is possible through mentorship. Solutions architects should play a player-coach role and mentor others. Let's look at this in more detail.

Being a mentor to others

Mentoring is about helping others and setting them up for success based on your learning and experience. It is an effective way to develop leaders, through one-to-one mentor/mentee relationships. To be a good mentor, you must establish an informal communication style where the mentee can develop a comfort zone. The mentee can seek advice in multiple areas, such as career development or personal aspects, such as work-life balance. You should do an informal needs assessment and set mutual goals and expectations.

Mentorship is more about listening than talking. Sometimes, people need someone to listen to them and advise as required. You should listen carefully first and understand their point of view.

Help the mentee make their own decisions, making them feel more accomplished. As a good mentor, when advising on someone's career, you need to be open to advice on the best fit for the mentee, even if it may not necessarily be the best fit for the company. Always provide honest, constructive feedback to help them identify and overcome gaps.

The critical trait of a mentor is the ability to inspire people. Often, people may choose you as a mentor if they see a role model in you. Help your mentee realize their full potential without putting your view forward, and help them achieve what they had never before thought of. There are always mutual benefits to being a mentor; you also learn from mentees about people's behavior and growth. Being a mentor to others will ultimately help you to become a better leader and person.

To take your expertise to the next level, you can become a technology evangelist and thought leader. Let's explore this next.

Becoming a technology evangelist and thought leader

Technology evangelism is about being an expert in advocating technology and your product. Some organizations with an extensive product base roll out a separate technology evangelist role. Still, a solutions architect often needs to assume the role of an evangelist as part of their job. As a technology evangelist, you must be aware of current technology trends to understand real-world problems and advocate for your technology to solve business concerns.

Technology evangelism involves participating in an industry conference as a public speaker and promoting your platform. It allows you to become a thought leader and an influencer, which can help the organization increase its platform and product adoption. Public speaking is one of the critical skills required for a solutions architect to interact on various public platforms and present in front of large audiences.

An evangelist also creates and publishes blog posts, whitepapers, and microblogs to advocate their product. They socialize the content to increase adoption and interact with users to understand their feedback. An evangelist works backward from the customer and communicates feedback to the internal team to help make the product better. With time, as an evangelist, you will refine the message that works in the organization's best interests.

A solutions architect is a role with multiple responsibilities, and taking more ownership will help you succeed in your career.

Summary

In this chapter, you learned about the soft skills required for a solutions architect to succeed, and the importance of these. A solutions architect needs pre-sales skills, which help them support the organization's pre-sales cycle, such as with the RFP.

You learned about the presentation skills required for executive conversations and buy-in and the strategic understanding that a solutions architect should have to define key objectives and results for an organization. To execute at various levels, solutions architects should be able to think big and be flexible and adaptable. You learned details about solutions architects taking ownership and being accountable for their actions.

A solutions architect's role has the primary responsibility of architecture design. You learned about design thinking, with its principles and phases. You also learned about the importance of continuous learning and different techniques to carry on learning and keep up with market trends. You also explored additional responsibilities of the solutions architect—to work as a mentor and evangelist.

It has been a long journey through this book, learning all about solutions architects, from their roles and responsibilities to different aspects of solution design and architecture optimization. I hope you have learned a lot and that it will help you develop your career as a solutions architect or help you succeed in your current role.

Happy learning!

Leave a review!

Enjoyed this book? Help readers like you by leaving an Amazon review. Scan the QR code below to get a free eBook of your choice.





packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

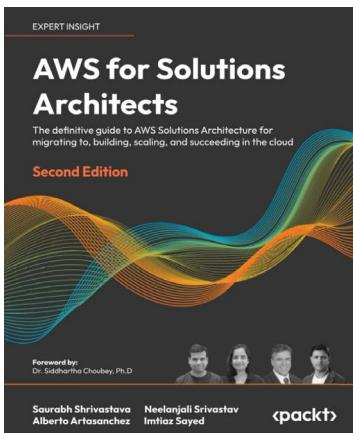
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



AWS for Solutions Architects – Second Edition

Saurabh Srivastava

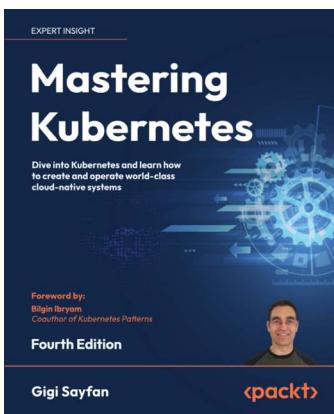
Neelanjali Srivastav

Alberto Artasanchez

Imtiaz Sayed

ISBN: 978-1-80323-895-1

- Optimize your Cloud Workload using the AWS Well-Architected Framework
- Learn methods to migrate your workload using the AWS Cloud Adoption Framework
- Apply cloud automation at various layers of application workload to increase efficiency
- Build a landing zone in AWS and hybrid cloud setups with deep networking techniques
- Select reference architectures for business scenarios, like data lakes, containers, and serverless apps
- Apply emerging technologies in your architecture, including AI/ML, IoT and blockchain



Mastering Kubernetes – Fourth Edition

Gigi Sayfan

ISBN: 978-1-80461-139-5

- Learn how to govern Kubernetes using policy engines
- Learn what it takes to run Kubernetes in production and at scale
- Build and run stateful applications and complex microservices
- Master Kubernetes networking with services, Ingress objects, load balancers, and service meshes
- Achieve high availability for your Kubernetes clusters
- Improve Kubernetes observability with tools such as Prometheus, Grafana, and Jaeger
- Extend Kubernetes with the Kubernetes API, plugins, and webhooks



Workflow Automation with Microsoft Power Automate – Second Edition

Aaron Guilmette

ISBN: 978-1-80323-767-1

- Learn the basic building blocks of Power Automate capabilities
- Explore connectors in Power Automate to automate email workflows
- Discover how to make a flow for copying files between cloud services
- Configure Power Automate Desktop flows for your business needs
- Build on examples to create complex database and approval flows
- Connect common business applications like Outlook, Forms, and Teams
- Learn the introductory concepts for robotic process automation
- Discover how to use AI sentiment analysis

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Solutions Architect's Handbook - Third Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

A/B testing 332-334
accelerated processing unit (APU) 186
ace editor 334
Active Directory (AD) 208, 215, 491
Active Directory Authentication Library (ADAL) 215
Active Directory Certificate Services (AD CS) 215
Active Directory Domain Services (AD DS) 215
Active Directory Federation Services (AD FS) 215
Active Directory Lightweight Directory Services (AD LDS) 215
active monitoring 204
Active Server Pages (ASP) 105
adaptability and flexibility 505, 506
Advanced Encryption Standard (AES) 233
agent-based solutions 81
agentless solutions 81
Agile organization
 Solutions Architect 26
AI21 Labs
 reference link 436
alerting mechanisms 66
alerts and incident response
 handling 280-282
Amazon 433
Amazon Bedrock
 reference link 433

Amazon CodeWhisperer
 reference link 438
Amazon DynamoDB 195
Amazon ECS 187
Amazon Elastic Compute Cloud (EC2) instances 37
Amazon Kafka 363
Amazon Kinesis 362
Amazon Kinesis Data Firehose (KDF) 362
Amazon Machine Images (AMIs) 254
Amazon Managed Blockchain (AMB) 362
Amazon Managed Streaming 362
Amazon Managed Streaming for Kafka (MSK) 362
Amazon OpenSearch Service 196
Amazon QuickSight 369
Amazon Relational Database Service (Amazon RDS) 37, 86
Amazon Route 53 198
Amazon S3 Glacier 363
Amazon Web Services (AWS) 8, 29, 61, 158, 198, 216, 248, 298, 362, 433, 510
 cost-effective, hosting on 310-312
 Elastic Block Store (EBS) 363
 green application, hosting on 310-312
 Relational Database Service (RDS) 363
 S3 Glacier 363
 Simple Storage Service (S3) 363
Amazon Web Services Directory Service 216
American National Standards Institute (ANSI) 367
Anthropic 435
 reference link 435

- Apache** 118
- Apache DistCp** 351
- Apache Flink** 363
- Apache Flume** 351
- Apache Hadoop** 366
- Apache Kafka** 351
- Apache Lucene library** 196
- Apache Spark** 366
- Apache Spark's MLlib** 398
- Apache Spark Streaming** 363
- Apache Sqoop** 351
- Apache Zeppelin** 367, 399
- app caching pattern** 119
- application and network layer**
 - network security 227, 228
 - operating system hardening 226
 - secure code 227
 - securing 226
 - software vulnerability mitigation 227
- Application Architect** 7, 8
- application layer** 105
- application monitoring** 275, 276
 - metrics 275
- Application Programming Interfaces (APIs)** 235
 - best practices 235-237
 - securing 235
- applications**
 - accessibility, achieving 50, 51
 - architectural constraints, overcoming 57, 58
 - architectural interoperability, ensuring 53
 - portability, ensuring 53
 - usability, achieving 50
- application security testing (AST)** 326
 - dynamic application security testing (DAST) 326
 - interactive application security testing (IAST) 326
 - software composite analysis (SCA) 326
- static application security testing (SAST) 326
- application server scaling**
 - session management 35
- Application Specific Integrated Circuits (ASICs)** 184
- Architect Evangelist** 9
- architectural performance design principles** 177
 - caching, applying 182, 183
 - concurrency, handling 181, 182
 - latency, reducing 178, 179
 - network throughput 179-181
- architectural reliability design principles** 246
 - capacity, adding 248
 - capacity, monitoring 248
 - disaster recovery, planning 252, 253
 - distributed system, creating 247
 - recovery validation, performing 248
 - self-healing capabilities and automation, integrating 246
- architectural reliability technology** 248
 - data replication 250
 - recovery point objective (RPO), planning 249, 250
 - recovery time objective (RTO), planning 249, 250
- architectural security, design principles** 207
 - auditing 209
 - authentication control, implementing 208
 - authentication, implementing 208
 - authorization control, implementing 208
 - automation 210
 - blast radius, reducing 209
 - data protection 210, 211
 - monitoring 209
 - security, applying 208
 - security incidents 211
- architectural security technology** 211
 - application and network layer, securing 226
 - cyberattacks 221

- data security 232
user identity and access management 211- 213
web security, handling 221
- architecture-driven modernization** 460
- array-based replication** 251
- artificial intelligence (AI) 357, 385, 422
- asymmetric-key encryption 233
- Asynchronous Pull method** 167
- asynchronous replication**
versus synchronous replication 250, 251
- atomicity, consistency, isolation, and durability (ACID)** 355
- auditing** 284, 285
- authentication** 208
- Authentication Server (AS)** 214
- authorization** 208
- automated stock trading** 407
- automation** 54, 316
alerting 55
application testing 54
applying 54, 55
deployment automation 55
IT infrastructure 54
logging 55
monitoring 55
security automation 55
- autonomous vehicle navigation system** 407
- auto-scaling** 164, 144
applying 199, 201
group 34
- auto-updates** 227
- AWS App Mesh** 159
- AWS CloudHSM** 234
- AWS CodeCommit** 334
- AWS data ingestion services** 352
- AWS Direct Connect location** 96
- AWS M2 platform**
- key features 474
- AWS Trusted Advisor** 308
- AXELOS**
reference link 273
- Azure Container Registry** 187
- Azure data ingestion services** 352
- Azure Disk Storage** 364
- Azure File Storage** 364
- Azure Kubernetes Service (AKS)** 188
- Azure SQL Database** 364
- B**
- Backend for Frontend (BFF) pattern** 171, 172
key aspects 171
- backpropagation** 415
- bastion host** 230
- big-bang approach** 458
- big data architecture** 346, 347
best practices 381-383
data lake architecture 372-374
data mesh architecture 376, 377
designing 370, 371
factors 380
lakehouse architecture 375, 376
selecting 379
streaming data architecture 378, 379
- big data processing pipelines**
designing 348, 349
- blockchain data store** 361
- blockchain networks, types**
consortium blockchain 361
private blockchain 361
public blockchain 361
- block storage**
working with 191
- Bloom models** 436
- BloomZ 176B** 436

blue-green deployment 327, 328
bounded contexts 150
 order processing context 150
 product catalog context 150
 user account context 150
budget
 versus forecast 292
buffer 224
buffer overflow attack 224
build phase 332
build server 335
bulkheads pattern
 implementing 125, 126
business continuity plan 55, 56
business intelligence (BI) 347, 397

C

cache aside pattern 119
cache-based architecture
 app caching pattern 119
 building 112, 113
 cache distribution pattern 114, 115
 cache proxy pattern 117
 Memcached 120
 Redis 120
 rename distribution pattern 116
 rewrite proxy pattern 118, 119
cache distribution pattern 114, 115
cache hit 120
cache miss 120
cache proxy pattern 117
 benefits 117
canary analysis 328
capital expenditure (CapEx) 143, 290
Carbonite 89
cascade failure 40

CD strategy
 blue-green deployment 327, 328
 immutable deployment 329
 implementing 326, 327
 in-place deployment 327
 red-black deployment 329
 rolling deployment 327
Center of Excellence (CoE) 78
Certification Authority (CA) 234
change data capture (CDC) 88, 352
charge-back approach 294
Chief Executive Officer (CEO) 502
Chief Financial Officer (CFO) 502
Chief Information Officer (CIO) 502
Chief Technology Officer (CTO) 502
CI/CD pipeline
 continuous testing, implementing 331, 332
 versus development, security, and operations
 (DevSecOps) 324, 325
circuit breaker pattern 125
circuit breakers 40
CI server 335, 336
Classless Inter-Domain Routing (CIDR) 228
Claude 435
Clean Architecture 136, 137
 components 136
C-level executives 501-503
client-server architecture 109
Cloud Architect 8
cloud automation 98
 advantages 98
cloud computing models
 Infrastructure as a Service (IaaS) 64
 Platform as a Service (PaaS) 64
 Software as a Service (SaaS) 64
cloud data storage 363, 364
 Amazon Web Services (AWS) 363

- data analytics 364, 365
- Google Cloud Platform (GCP) 364
- Microsoft Azure 364
- working with 191, 192
- Cloud Development Kit (CDK)** 320
- CloudEndure** 89
- cloud infrastructure architect** 10
- cloud migration** 77-79
 - application, designing 83-86
 - application, executing 87
 - application optimization 93, 94
 - cutover process 90
 - information, analyzing 81, 82
 - integrating 90
 - operation phase 92, 93
 - plan, creating 82, 83
 - portfolio discovery 79-81
 - validating 90
 - workload discovery 79-81
- cloud migration, application**
 - data migration 87, 88
 - server migration 89
- cloud migration strategy**
 - creating 69-71
 - defining, for legacy systems 465, 466
 - documentation and support 467
 - lift and shift migration 71
 - retain strategy 74
 - retire strategy 75
 - risks 76
 - selecting 75, 77
- cloud-native approach** 72
 - refactor method 73
 - repurchase method 74
- cloud-native architecture** 65, 66, 142, 143
 - benefits 68
 - designing 66-68
- cloud-native architecture, anti-patterns** 172
 - costs consideration 174
 - lack of testing 173
 - manual scaling 172
 - monitoring and logging 173
 - network latency, ignoring 173
 - over-optimization 173
 - security best practices, ignoring 173
 - single point of failure 172
 - tightly coupled services 173
- cloud-native architecture, characteristics**
 - API-first 142
 - automation 142
 - continuous monitoring and improvement 142
 - DevOps practices 142
 - elasticity and stability 142
 - microservices 142
 - resilience and fault tolerance 142
 - serverless computing 142
 - statelessness 142
- cloud-native monitoring** 66
- CloudOps model pillars** 99, 286, 287
 - benefits 98
 - implementing 97
- cloud providers**
 - AWS data ingestion services 352
 - Azure data ingestion services 352
 - GCP data ingestion services 352
- cloud reliability**
 - improving 262, 263
- Cloud SQL** 364
- cloud's shared security responsibility model** 238, 240
- CloudWatch** 306
- code builder** 509
- code deployment** 336-338
 - AllAtOnce 337
 - custom 337
 - HalfAtATime 336
 - OneAtATime 336

code pipeline 338
examples 338

cold start 146

commercially available
off-the-shelf (COTS) 464

compliance certifications 237, 238

computational choice
containers, working with 186
making 184, 186
serverless computing 189

computer-generated data sources
application server logs 346
clickstream logs 346
images and videos 346
sensor data 346

Concurrent Versions System (CVS) 338

configuration items (CIs) 271

configuration management
(CM) 271-273, 322, 323
benefits 273

configuration management
database (CMDB) 271

consortium blockchain 361

Consul Connect 159

consumer packaged goods (CPG) 386

container application
benefits 129, 130
deploying 128, 129

container-based architecture
building 131-133

container deployment 130, 131

containerization method 89

containers
working with 186

content delivery network (CDN) 182, 197, 488

content distribution network
(CDN) 35, 114, 178

continuous integration and continuous
deployment (CI/CD) pipelines 14, 55, 227

continuous integration (CI) 65

continuous learning 509-511

continuous testing
implementing, in CI/CD pipeline 331, 332

continuous training/continuous monitoring
(CT/CM) 410

Controller 121
responsibilities 121

convolution neural networks (CNNs) 415

cost-benefit analysis (CBA) 456, 464

cost-effective
hosting, on AWS 310-312

cost optimization 289
driving, in public cloud 307-309

cost optimization, design principles 290
budget and forecast, planning 292, 293
continuous cost optimization 295, 296
demand and service catalogst, managing 293
demand and service catalogst, managing 294
expenditure, tracking 294, 295
total cost of ownership (TCO),
calculating 290, 291

cost optimization, methods
considering 298

cost optimization, techniques 296
architectural complexity, reducing 296-298
cost usage and reports, monitoring 302-307
governance, applying 299-301
IT efficiency, increasing 298, 299
resource cost tagging 301, 302
standardization, applying 299-301

Create, Read, Update, Delete (CRUD) 109

credit scoring 408

cross-platform compatibility 202

Cross-Site Request Forgery (CSRF) attack 223

cross-site scripting (XSS) 223, 491

- customer relationship management (CRM)** 70, 213, 291, 464
- cutover process** 90
 - live migration cutover 91, 92
- cyberattacks** 221
 - buffer overflow attack 224
 - Cross-Site Request Forgery (CSRF) attack 223
 - Denial of Service (DoS) attack 221, 222
 - Distributed Denial of Service (DDoS) attack 222
 - memory corruption attack 224
 - SQL injection (SQLi) attack 223
 - XSS attack 223
- D**
 - DAEMON tools** 227
 - dark launch** 329
 - data access layer**
 - database-level isolation 107
 - row-level isolation 107
 - table-level isolation 107
 - data affinity** 190
 - data analytics** 350, 364, 365
 - Data Architect** 11
 - database application architecture**
 - handling 133, 134
 - high-availability database pattern 135
 - database choice**
 - data search functionality, building 196
 - making 194
 - nonrelational databases 195
 - Online Analytical Processing (OLAP) 195, 196
 - Online Transaction Processing (OLTP) 194
 - database layer** 106
 - database scaling** 36
 - database storage** 193
 - data classification** 232
 - private data 232
 - public data 232
 - restricted data** 232
 - data-driven design** 48
 - data encryption at rest** 232, 233
 - data encryption in transit** 234, 235
 - Data Encryption Standard (DES)** 233
 - data ingestion** 350, 351
 - in cloud 352
 - open source tools 351
 - data lake**
 - benefits 373
 - data lake architecture** 372-374
 - data mesh architecture** 376, 377
 - fundamental principles 377
 - data migration** 87, 88
 - Data Migration Service (DMS)** 382
 - data processing** 350, 366
 - Apache Hadoop 366
 - Apache Spark 366
 - Apache Zeppelin 367
 - Ganglia 367
 - Hadoop User Experience (HUE) 366
 - HBase 367
 - Hive 366
 - in cloud 367, 368
 - JupyterHub 367
 - Pig 366
 - Presto 367
 - data protection**
 - data at rest 210
 - data at transit 210
 - data in use 210
 - data replication** 250
 - asynchronous replication, versus synchronous replication 250, 251
 - replication method 251
 - data science**
 - working with 389-391

- data search functionality**
 - building 196
- data security** 232, 491
 - data classification 232
 - data encryption at rest 232, 233
 - data encryption in transit 234, 235
 - encryption key management 233
- data storage** 350, 353, 354
 - blockchain data store 361
 - NoSQL databases 357
 - object storage 359
 - search data stores 359
 - streaming data stores 362
 - structured data stores 355
 - technology choices 354
 - unstructured data stores 359
 - VectorDB 360, 361
- data swamps** 374
- data tier** 106
- data visualization** 369
 - Amazon QuickSight 369
 - Jaspersoft 369
 - Kibana 369
 - Power BI 369
 - Spotfire 369
 - Tableau 369
- data warehousing** 356, 357
- DDoS mitigation** 224, 225
- decentralized applications (DApps)** 361
- decentralized finance (DeFi)** 361
- decision trees** 393
- Deep Extension of Latent Logic (DELL)** 436
- deep learning** 413-415
 - autonomous vehicles 416
 - healthcare 416
 - manufacturing 416
- DeepMind**
 - reference link 434
- defense-in-depth (DiD) approach** 209
- DELETE method** 110
- demand management** 294
- Denial of Service (DoS) attack** 221, 222
- deployment strategy**
 - best practices 330, 331
- design principles, for ML architecture** 406
 - data quality assurance, implementing 407
 - efficiency, ensuring 408
 - fault tolerance, ensuring 409
 - flexibility, ensuring 407
 - interpretability, ensuring 408
 - machine learning system, organizing into modules 406
 - privacy and security, ensuring 407
 - real-time capability, implementing 408
 - reliability, ensuring 407
 - reproducibility, ensuring 406
 - robustness, ensuring 407
 - scalability, ensuring 406
- design thinking** 506
 - phases 508
 - principles 507
 - reference link 508
- development and operations (DevOps)** 314
 - benefits 315, 316
 - building 341-343
 - CI/CD 316-318
 - components 316
 - configuration management (CM) 322, 323
 - continuous improvement 318, 319
 - continuous monitoring 318, 319
 - Infrastructure as Code (IaC) 319-321
- development, security, and operations (DevSecOps)** 323
 - building 341-343
 - security 323
 - versus CI/CD pipeline 324
- DevOps architect** 14-16

- DevOps best practices**
implementing 339-341
- DevOps metrics**
examples 319
- DevOps tools for CI/CD**
CI server 335, 336
code deployment 336-338
code editor 334
code pipeline 338
source code management 334
using 334
- digital signal processor (DSP)** 186
- direct-attached storage (DAS)** 192
- directed acyclic graphs (DAGs)** 366
- disaster recovery (DR)** 249
backup and restore 253-255
best practices, applying 261, 262
multi-site strategy 260, 261
pilot light 255-257
planning 252, 253
scenarios 252
warm standby 257-260
- disaster recovery plans** 56
Backup and Store 56
Multi-Site 56
Pilot Lite 56
Warm Standby 56
- disaster recovery replication** 89
- disk copy method** 89
- Distributed Denial of Service (DDoS) attack** 49, 198, 222, 279
- DNS routing strategy**
defining 198, 199
- Docker** 186-398
- Domain-Driven Design (DDD)**
aggregate 124
anti-corruption layer 124
bounded contexts 123
building 123, 125
domain 123
domain events 124
entities 123
factories 124
repositories 124
services 124
ubiquitous language 123
value objects 124
- Domain Name System (DNS)** 38, 114, 126, 209, 327
- dynamic application security testing (DAST)** 326, 341
- Dynatrace** 89
- E**
- eavesdropping attack** 234
- e-commerce application**
implementing 160
- edge computing**
using 197, 198
- effective data management** 202
- elastic architecture**
building 37
- Elastic Block Store (EBS)** 192, 363
- Elastic Compute Cloud (EC2)** 165, 303, 336
- Elastic Container Registry (ECR)** 187
- Elastic Container Service (ECS)** 131, 160, 186, 469
- Elastic File System (EFS)** 192
- Elastic IP (EIP)** 127
- elasticity** 32
- Elastic Kubernetes Service (EKS)** 131, 133, 160, 188, 469
- Elastic Load Balancing (ELB)** 118, 199
- Elastic MapReduce (EMR)** 276, 365
- Elastic Network Interface (ENI)** 127

- Elasticsearch, Logstash, and Kibana (ELK)** 196
- encapsulation** 463
- encryption key management** 233
- endurance testing** 203
- Enhanced Data rates for Global Evolution (EDGE)** 209
- enterprise data warehouse (EDW)** 376
- enterprise discount programs (EDPs)** 294
- Enterprise Resource System (ERP)** 212
- Enterprise Solutions Architect** 7
- entity-relationship diagram (ERD)** 489
- Ethereum** 362
- event-driven architecture (EDA)**
- creating 168
 - event stream model 170, 171
 - publisher/subscriber (pub/sub) model 169
- event stream model** 170, 171
- challenges 170
- extensible markup language (XML)** 216
- extract, transform, and load (ETL)** 283, 352
- ## F
- fan-out/fan-in pattern** 155, 156
- benefits 156
- Fargate** 469
- fault tolerance** 248
- architecture 41, 42
- Federal Risk and Authorization Management Program (FedRAMP)** 284
- federated identity management (FIM)** 213
- Field Programmable Gate Arrays (FPGAs)** 184
- file storage**
- working with 191
- financial fraud detection system** 406
- Flask** 398
- floating IP pattern**
- creating** 126, 127
- Fluentd** 351
- forecast**
- versus budget 292
- forward propagation** 415
- foundational models (FMs)** 391
- Freshservice** 269
- FTP Secure (FTPS)** 235
- Function as a Service (FaaS)** 65, 144, 189
- future-proof extendable and reusable architecture**
- building 51, 52
- ## G
- Ganglia** 367
- GANVerse3D** 436
- GCP data ingestion services** 352
- GenAI architect** 13
- General Data Protection Regulation (GDPR)** 457
- Generalist Solutions Architect roles** 7
- Application Architect 7, 8
 - Architect Evangelist 9
 - Cloud Architect 8, 9
 - Enterprise Solutions Architect 7
- Generative Adversarial Networks (GANs)** 426-428
- generative AI, challenges** 448
- ethical concerns and misuse 450
 - latent space interpolation 449, 450
 - mode collapse 449
 - stability issues, training 448
- generative AI, FMs** 433
- AI21 Labs 436
 - Amazon 433
 - Anthropic 435
 - factors 443, 444
 - Google 434
 - Hugging Face 436

- Jasper.ai 436
Meta AI 435
Microsoft 435
Nvidia 436
OpenAI 434
with public cloud providers 440-443
- generative AI (genAI) 13, 422, 436**
for builders 437-439
for end users 436, 437
model hallucinations, preventing 444, 446
modernizing legacy code, using 474, 475
reference architecture building, for mortgage
assistant app 446-448
- generative AI, systems architecture 425**
generative model types 425, 426
generator 425
latent space 425
loss function 425
training data 425
- generative AI, use cases 423**
business operations optimization 424
customer experience transformation 423
employee productivity enhancement 424
- generative model types 425, 426**
flow-based models 432
Generative Adversarial Networks
(GANs) 426-428
PixelCNN 432
PixelRNN 432
transformer-based generative models 430, 431
Variational Autoencoders (VAEs) 428, 429
- generative pretrained transformer (GPT) 418**
- GET method 109**
- Global Positioning System (GPS) 457**
- Google AI**
reference link 434
- Google Anthos 186**
- Google Cloud Bigtable 364**
- Google Cloud Platform (GCP) 8, 29, 61, 254, 352, 364, 510**
Cloud SQL 364
Google Cloud Bigtable 364
Google Cloud Storage 364
Persistent Disk 364
- Google Cloud Storage 364**
- Google Kubernetes Engine (GKE) 188, 364**
- graphics processing unit (GPU) 184, 436**
- green application**
hosting, on AWS 310-312
- green computing 309**
Green IT 309
influencing, on cost considerations 309, 310
- H**
- Hadoop Distributed File System (HDFS) 351**
- hard disk drive (HDD) 179**
- hardware security module (HSM) 234**
- HBase 367**
- Heads of Line of Business (LoBs) 502**
- Health Insurance Portability and Accountability
Act (HIPAA) 279**
- Hexagonal Architecture 136**
- high-availability database pattern 135**
- high availability (HA) 38, 245**
- high availability (HA) architecture**
building 38
component failure, addressing 40
redundancy, achieving 39, 40
resilient architecture 38
- Hive 365, 366**
- Hive Query Language (HQL) 366**
- horizontal scaling 32**
- host-based IDS 231**
- host-based replication 251**
- hot standby 260**

- HTTP methods**
- DELETE method 110
 - GET method 109
 - POST method 110
 - PUT method 110
- Hugging Face** 436
- human/computer interaction (HCI)** 332
- Human-in-the-Loop (HITL)** 444
- Human Monitoring and Engagement (HELM)** 444
- hybrid cloud** 62
- hybrid cloud architecture**
- benefits 96
 - creating 94-96
 - risks 96
- Hyperledger** 362
- hyperparameter tuning** 432
- hypervisor-based replication** 252
- I**
- idempotent operations** 40
- Identify and Access Management (IAM)** 213, 343, 382, 491
- Identity Cloud Service (IDCS)** 216
- identity provider (IdP)** 213
- immutable architecture**
- creating 43, 44
- immutable deployment** 329
- industrial equipment monitoring** 409
- Information Technology Infrastructure Library (ITIL)** 93, 273
- Information Technology Service Management (ITSM)** 93, 273
- Infrastructure Architect** 10, 11
- Infrastructure as a Service (IaaS)** 64, 248, 291
- Infrastructure as Code (IaC)** 15, 266, 261, 299, 319-321
- infrastructure monitoring** 274, 275
- metrics 274
- input/output (I/O) operations** 80
- input/output operations per second (IOPS)** 295
- integrated development environment (IDE)** 334, 438
- interactive application security testing (IAST)** 326
- International Organization for Standardization (ISO)** 284
- Internet Control Message Protocol (ICMP)** 229
- internet DNS cache** 114
- internet gateway (IGW)** 229
- Internet Message Access Protocol (IMAP)** 235
- Internet of Things (IoT)** 66, 195, 350
- Internet Protocol (IP)** 126
- Internet Protocol Security (IPsec)** 235
- interoperability** 53
- intrusion detection system (IDS)** 209, 230, 491
- host-based IDS 231
 - network-based IDS 231
- intrusion prevention system (IPS)** 209, 491
- Signature-Based Detection 230
 - Statistical Anomaly-Based Detection 230
- Istio** 159
- IT asset management (ITAM) tool** 269
- configuration management 271-273
 - process 270
- IT operations analytics (ITOA)** 282, 283
- IT procurement documentation**
- for solution architecture 495-497
- J**
- Jasper.ai** 436
- reference link 436
- Jaspersoft** 369
- Java Message Service (JMS)** 277

JavaScript Object Notation (JSON) 109
JavaServer Pages (JSP) 105
Jenkins Controller 336
Jira Service Desk 270
JMeter 204
job observer pattern 166, 167
JSON Web Token (JWT) 220, 221
JupyterHub 367
Jupyter Notebook 399

K

Keras 398
Kerberos 214, 215
key-based encryption
 asymmetric-key encryption 233
 symmetric-key encryption 233
Key Distribution Center (KDC) 214
Key Management Service (KMS) 233, 382
key performance indicator (KPI) 246, 340, 501
Kibana 196, 369
Kinesis Data Analytics (KDA) 362
Kinesis Data Streams (KDS) 362
k-means clustering 395
K-Nearest Neighbours (k-NNs) 394
kubelet 188
Kubernetes 187, 188
Kubernetes cluster 188

L

Laboratory for Artificial Intelligence in Design (AiDLab) 422
lakehouse architecture 375, 376
language translation
 example 430, 431
large language models (LLMs) 418
learning management systems (LMSs) 495

legacy system challenges
 corporate security issues 456
 incompatibility 457
 learning 454
 maintenance cost and update 455, 456
 skills and documentation 456
 user demand 454, 455
legacy system modernization techniques 462
 encapsulation 463
 rearchitecting 463
 redesigning 464
 refactoring 463
 rehosting 463
 replacing 464
 re-platforming 463
lift and shift migration 71
 relocate 72
 replatform 71
 reshost 71
Lightweight Directory Access Protocol (LDAP) 215, 366
linear regression 393
Linkerd 159
live migration cutover 91, 92
load balancer
 application load balancer 199
 applying 199
 Layer 4 load balancing 199
 Layer 7 load balancing 199
 network load balancer 199
LoadRunner 204
load testing 203
logic tier 105
logistic regression 393
log monitoring 277, 279
Logstash 196
loose coupling 44-46
Lyra-Fr 10B 436
Lyra-Mini 436

M

- machine learning algorithms** 393
 - decision trees 393
 - k-means clustering 395
 - K-Nearest Neighbours (k-NNs) 394
 - linear regression 393
 - logistic regression 393
 - neural networks 394
 - random forests 394
 - support vector machines (SVMs) 394
 - XGBoost 395
- machine learning architecture**
 - building 401
 - deploy and manage 403
 - design principles 406
 - prepare and label 401, 402
 - reference architecture 404, 405
 - select and build 402
 - train and tune 402
- machine learning (ML)** 345, 385-387, 422
 - in cloud 399, 400
 - multi-instance learning 389
 - reinforcement learning 388
 - self-supervised learning 389
 - supervised learning 387
 - types 387
 - unsupervised learning 388
 - use cases 386
 - working with 389-391
- machine learning tools**
 - Apache Spark's MLlib 398
 - BI tools 397
 - Docker 398
 - Flask 398
 - Keras 398
 - Matplotlib 396
 - NumPy 395
 - Pandas 396
 - PyTorch 397
- scikit-learn 396
- Seaborn 397
- TensorFlow 397
- man-in-the-middle (MITM) attack** 234
- manual scaling** 172
- Massively Parallel Processing (MPP)** 195, 356
- master data management (MDM)** 348
- Matplotlib** 396
- mean time to recovery (MTTR)** 341
- Memcached** 114
 - versus Redis 120
- memory corruption attack** 224
- mentor** 511
- message queues**
 - used, for decoupling shared code applications
 - migration 471, 472
- Meta AI** 435
 - reference link 435
- Microfocus** 89
- microservice architecture**
 - best practices 151
 - creating 150-153
 - fan-out/fan-in pattern 155, 156
 - Saga pattern 153, 155
 - service mesh pattern 157-160
- microservices** 142
- Microsoft** 435
- Microsoft Azure** 364
 - Azure Blob Storage 364
 - Azure Disk Storage 364
 - Azure File Storage 364
 - Azure SQL Database 364
- Microsoft Azure model**
 - reference link 435
- Microsoft Visio** 479
- migration and enhancements** 460

- minimum viable product (MVP)**
 - approach 58, 59
 - ML Architect** 12
 - ML model evaluation**
 - overfitting, versus underfitting 392
 - MLOps** 409
 - automation 410
 - best practices 411, 412
 - deployment 410
 - monitoring 411
 - principles 410, 411
 - reproducibility 410
 - testing 410
 - versioning 410
 - mobile application performance**
 - battery consumption 202
 - considerations 201
 - cross-platform compatibility 202
 - effective data management 202
 - limited resources 201
 - load times, optimization 201
 - network efficiency 202
 - quality assurance 202
 - testing 202
 - user experience (UX) design 202
 - user interface (UI) 201
 - mobile BFF** 172
 - Model** 121
 - responsibilities 121
 - Model as a Service (MaaS)** 435
 - model hallucinations**
 - preventing 444, 446
 - techniques and strategies 445
 - models Foundation Models (FMs)** 422
 - Model-View-Controller (MVC) architecture** 121
 - advantages 122
 - applying, to design online bookstore 122, 123
 - modernization approach**
 - architecture-driven modernization 460
 - defining 459
 - migration and enhancements 460
 - system re-engineering 460
 - modernizing legacy code**
 - with generative AI (genAI) 474, 475
 - mortgage assistant app**
 - used, for building generative AI reference architecture 446-448
 - MoSCoW** 58
 - multi-cloud approach** 97
 - advantages 97
 - challenges 97
 - multi-factor authentication (MFA)** 208, 209
 - multi-tenant SaaS-based architecture**
 - creating 106-108
 - multitier architecture** 104
- ## N
- NAT gateway** 229
 - natural language processing (NLP)** 359, 416
 - chatbots and virtual assistants 416
 - machine translation 417
 - sentiment analysis 417
 - text summarization 417
 - NetApp** 89
 - Netezza** 356
 - Network Access Control List (NACL)** 229
 - network address translation (NAT)** 229
 - Network Architect** 11
 - network area storage (NAS)** 191
 - working with 191
 - network-attached storage (NAS)** 232
 - network-based IDS** 231
 - network-based replication** 251
 - networking** 197
 - Network Interface Card (NIC)** 127

- network performance** 197
 auto-scaling, applying 199, 201
 DNS routing strategy, defining 198, 199
 edge computing, using 197, 198
 load balancer, applying 199
- network security** 227, 228
 NACL 229
 routing table 229
 security groups 229
 Subnets 229
 Virtual Private Cloud (VPC) 228
- neural networks** 394
- Nginx** 118, 199
- non-functional requirements (NFRs)** 2, 478
- nonrelational databases** 195
- NoSQL databases** 357
 types 358
 versus SQL databases 357, 358
- n-tier layered architecture**
 building 104, 105
- NumPy** 395
- Nvidia** 436
 reference link 436
- ## O
- objectives and key results (OKRs)** 503
 used, for defining strategy execution 503-505
- object-oriented analysis and design (OOAD)**
 concepts 52
- object storage**
 working with 191, 192
- off-the-shelf software** 291
- Online Analytical Processing (OLAP)** 193-196
- online transaction**
 processing (OLTP) 193, 194, 355
- on-premises cloud** 62
- OpenAI** 434
 reference link 434
- Open Authorization (OAuth)** 218, 219
- OpenID** 218
- OpenID Connect (OIDC)** 218
- open source tools, data ingestion**
 Apache DistCp 351
 Apache Flume 351
 Apache Sqoop 351
- Open Web Application Security Project (OWASP)** 227
- operating system (OS)** 313
- operation**
 designing for 57
- operational excellence** 57
- operational excellence, design principles** 266
 failures, predicting 267
 incremental and reversible change 267
 manual tasks, automating 266
 mistakes and refining 267
 operational runbook, updating 268
- operational excellence technologies**
 auditing 284, 285
 functioning 273, 274
 improvement 282
 in public cloud 285, 286
 IT asset management (ITAM) tool 269
 IT operations analytics (ITOA) 282, 283
 planning 268, 269
 reporting 284, 285
 root cause analysis (RCA) 284
 selecting 268
 system health monitoring 274
- operational expenditure (OpEx)** 143, 290
- Optimized Row Columnar (ORC)** 372
- Oracle Cloud Platform (OCP)** 62
- organization units (OUs)** 300
- OS cloning technique** 89
- OS copy method** 89

ownership and accountability 503

solutions architect, building 505

strategy execution, defining with OKRs 503-505

P

page cache 183

PagerDuty 270

Pandas 396

passive monitoring 204

Payment Card Industry Data Security Standard (PCI DSS) 382

payment card industry (PCI) 279

performance

designing for 42

performance monitoring

managing 204, 205

performance optimization, technology selection 183

computational choice, making 184, 186

database choice, making 194

network performance 197

storage choice, making 190

performance testing 203

endurance testing 203

load testing 203

spike testing 203

stress testing 203

volume testing 204

Persistent Disk 364

personalized marketing 408

Personally Identifiable Information (PII) 211

phased approach 458

Pig 365, 366

Pipes-and-Filters architecture 167, 168

primary components 167

Platform as a Service (PaaS) 64, 188

platform monitoring 276, 277

portfolio discovery 80

POST method 110

Power BI 369

predictive scaling 33

pre-sales skills

acquiring 500

C-level executives 501-503

unique skill 500, 501

presentation tier 105

Presto 367

principle of least privilege 208

private blockchain 361

private cloud 62

private pricing agreements (PPAs) 294

private subnet 229

proof of concept (POC) 459, 478, 508

prototyping 509

public cloud 62

cost optimization, driving 307-309

operational excellence technologies 285, 286

public cloud architecture 63, 64

public cloud mainframe migration 467

benefits 473, 474

challenges 468

shared code applications migration 470

standalone applications, migrating 469

public cloud providers 65

public-key encryption 233

Public Key Infrastructure (PKI) 234

public subnet 229

publisher/subscriber (pub/sub) model 169

PUT method 110

PyTorch 397

Q

quality assurance (QA) 202, 246, 247, 313, 491
Quantum Ledger Database (QLDB) 362
queue-based architecture
 building 164
 job observer pattern 166, 167
 queuing chain pattern 164-166
 terminology 164
queuing chain pattern 164-166
 benefits 165

R

R3 Corda 362
random-access memory (RAM) 274
random forests 394
reactive architecture 161-164
 implementing 162
Reactive Manifesto
 reference link 161
reactive scaling 34
real-time fraud detection 408
rearchitecting 463
recovery point objective
 (RPO) 15, 55, 135, 248, 267
 planning 249, 250
recovery time objective
 (RTO) 15, 55, 135, 248, 267
 planning 249, 250
recurrent neural networks (RNNs) 415
red-black deployment 329
redesigning 464
Red Hat Enterprise Linux (RHEL) 271
Redis 114
 versus Memcached 120
Redshift 356
redundancy 39
refactoring 463
regularization 433
rehosting 463
reinforcement learning 388
relational database management system (RDBMS) 351
relational databases 355
Relational Database Service (RDS) 276, 363
reliability 245
remaining useful life (RUL) 386
rename distribution pattern 116
replacing 464
re-platforming 463
replication method 251
 array-based replication 251
 host-based replication 251
 hypervisor-based replication 252
 network-based replication 251
reporting 284, 285
Representational State Transfer (REST) 108, 234
request for information (RFI) 495
request for proposal (RFP) 495, 500
request for quotation (RFQ) 495
request for response (RFR) 17
resilient architecture 38
resilient distributed datasets (RDDs) 363
responsibilities, Solutions Architect 16, 17
 post-launch operability and maintenance,
 ensuring 25
responsibilities, SSA roles
 architecture constraints 21-23
 functional requirements (FRs), analyzing 18
 NFRs, defining 18-20
 POC and prototype, developing 24
 solution design and delivery 24
 solution scaling and technology evangelism 25

- stakeholders, engaging 21
- technology selections, making 23
- RESTful architecture** 297
- RESTful-architecture-based e-commerce website**
 - building 111, 112
- RESTful web service architecture** 109
 - client-server architecture 109
 - code on demand 110
 - layered system 110
 - resource-based 110
 - resource representation 110
 - stateless 109
 - uniform interface 109, 110
- REST web services** 109
- retain strategy** 74
- retire strategy** 75
- return on investment**
 - (ROI) 74, 289, 456, 492, 501
- rewrite proxy pattern** 118, 119
- RFx documents** 495
- Rivest-Shamir-Adleman (RSA)** 233
- role-based authentication (RBA)** 212
- rolling deployment** 327
- root cause analysis (RCA)** 211, 267, 284
- routes** 229
- routing policies**
 - failover routing policy 198
 - geolocation routing policy 198
 - geoproximity routing policy 198
 - latency routing policy 199
 - simple routing policy 198
 - weighted routing policy 199
- RStudio** 399
- S**
- Saga pattern** 153, 155
- process flow 153, 154
- working 153
- scalability** 32
- scalable architecture design**
 - building 32-34
 - database scaling 36
 - elastic architecture, building 36, 37
 - session management, for application server scaling 35
- scaling**
 - horizontal scaling 32
 - predictive scaling 33
 - reactive scaling 34
 - vertical scaling 33
- scikit-learn** 396
- scope creep** 23
- Seaborn** 397
- search data stores** 359
- secure code** 227
- Secure Shell File Transfer Protocol (SFTP)** 382
- Secure Shell (SSH)** 235
- Secure Sockets Layer (SSL)** 222, 343, 488
- security** 48
 - adding 48, 49
 - aspects 48
- Security Architect** 14
- Security Assertion Markup Language 2.0 (SAML2)** 366
- Security Assertion Markup Language (SAML)** 216-218
- security certifications** 237, 238
- security monitoring** 279, 280
- semi-supervised learning** 388
- server auto scaling** 34
- Serverless Application Model (SAM)** 145
- serverless architecture**
 - building 144, 145
 - cold start 146

- compliance and regulation 147
- considerations 145, 146
- cost management 146
- data storage and persistence 146
- dependencies 147
- distributed monitoring 146
- distributed tracing 146
- event-driven design 146
- granular function design 146
- managed services, using 147
- performance consideration 146
- scalability 146
- security 146
- statelessness 146
- testing and debugging 147
- serverless computing** 189
- server migration** 89
- service catalog management** 294
- ServiceDesk Plus** 270
- service-level agreement (SLA)** 97, 249, 265, 480
- service mesh pattern** 157-160
 - AWS App Mesh 159
 - Consul Connect 159
 - Istio 159
 - Linkerd 159
 - primary features 158
- service-oriented architecture (SOA)** 108
 - RESTful-architecture-based e-commerce website, building 111, 112
- service-oriented thinking**
 - developing 46, 47
- shared code applications migration** 470
 - decoupling, with message queues 471, 472
 - decoupling, with shared library 471
 - decoupling, with standalone API 470, 471
- shared library**
 - used, for decoupling shared code applications migration 471
- show-back approach** 294
- Signature-Based Detection** 230
- Simple and Protected GSSAPI Negotiation Mechanism (SPNEGO)** 366
- Simple Mail Transfer Protocol Secure (SMTPS)** 235
- Simple Notification Service (SNS)** 169
- Simple Object Access Protocol (SOAP)** 108, 234
- Simple Queue Service (SQS)** 165, 277
- Simple Storage Service (S3)** 144, 192, 283, 363
- Simple Web Token (SWT)** 220
- single point of failure** 172
- single sign-on (SSO)** 208, 213, 382
- single tenancy** 106
- small and medium-sized businesses (SMBs)** 251
- smart TV BFF** 172
- software architect** 7
- Software as a Service (SaaS)** 64, 106, 291
- software composition analysis (SCA)** 326, 341
- software development life cycle (SDLC)** 339
- software vulnerability mitigation** 227
- SolarWinds** 269
- solid-state drive (SSD)** 183
- solution architecture** 487
 - IT procurement documentation 495-497
 - soft skills 500
- solution architecture design anti-patterns**
 - avoiding 137, 138
- solution architecture document (SAD)** 477
 - best practices 494, 495
 - goals 478
 - life cycle 493, 494
 - purpose 477, 478
 - views 480
- solution architecture document (SAD) structure** 480, 481
 - Appendix 493
 - business context 484, 486

- conceptual solution overview 486
- solution architecture 487
- solution implementation 492
- solution management 492
- solution overview 482-484
- solution architecture document (SAD) views** 478, 479
 - application architecture 488, 489
 - business view 479
 - data architecture 489
 - data view 480
 - deployment view 480
 - implementation view 480
 - information architecture 487
 - infrastructure architecture 491
 - integration architecture 490, 491
 - security architecture 491
 - logical view 479
 - operational view 480
 - process view 479
- solution architecture layers**
 - application caching 114
 - client-side caching 114
 - database caching 114
 - internet DNS cache 114
 - web content caching 114
- solutions architect** 2, 509
 - attributes 3, 4
 - benefits 2
 - building 505
 - career path 28
 - challenges 26, 27
 - cloud knowledge and certifications 29
 - skill development 28, 29
- Solutions Architect's role** 4
 - Generalist Solutions Architect roles 7
 - lifecycle, of solution delivery 5, 6
 - SSA roles 9
- source code management** 334
 - source code repositories 338
 - Spark 365
 - spike testing 203
 - Splunk 306
 - Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege (STRIDE)** 241
 - Spotfire** 369
 - SQL databases**
 - versus NoSQL databases 357, 358
 - SQL injection (SQLi) attack** 223, 491
 - SSA roles** 10
 - Data Architect 11
 - DevOps architect 14, 15
 - GenAI architect 13
 - Industry Architect 15, 16
 - Infrastructure Architect 10
 - ML Architect 12
 - Network Architect 11
 - Security Architect 14
 - SSH File Transfer Protocol (SFTPS)** 235
 - staging environment** 332
 - standalone API**
 - used, for decoupling shared code applications migration 470, 471
 - stateful architectural design** 148
 - building 147, 148
 - stateless** 109
 - stateless architectural design** 148, 149
 - building 147, 148
 - static application security testing (SAST)** 326, 341
 - static content**
 - scaling 34, 35
 - Statistical Anomaly-Based Detection** 230
 - storage area network (SAN)** 88, 232
 - working with 191

storage choice

block storage, working with 191
cloud data storage, working with 191, 192
database storage 193
file storage, working with 191
making 190
network area storage (NAS), working with 191
object storage, working with 191, 192
storage area network (SAN), working with 191

strategy execution

defining, with OKRs 503-505

streaming data architecture 378, 379

use cases 378

streaming data stores 362**stress testing** 203**structured data stores** 355

data warehousing 356, 357
relational databases 355

Structure Query Language (SQL) 223**Subversion (SVN)** 338**Super-fast, Parallel, In-memory Calculation Engine (SPICE)** 369**supervised learning** 387**Supervisord tools** 227**support vector machines (SVMs)** 394**Sustainability Pillar - AWS Well-Architected Framework**

reference link 311

symmetric-key encryption 233**synchronous replication**

versus asynchronous replication 250, 251

SYN floods 223**systemd** 227**system health monitoring** 274

alerts and incident response, handling 280-282
application monitoring 275, 276
infrastructure monitoring 274, 275
log monitoring 277, 279

platform monitoring 276, 277

security monitoring 279, 280

system modernization strategy

benefits 460, 461
defining 457
legacy application assessment 458, 459
modernization approach, defining 459

system re-engineering 460

System V init 227

T

Tableau 369

technology evangelist 9, 512

templates 319

TensorFlow 397

Teradata 356

testing 202

The Open Group Architecture Framework (TOGAF) 28

thought leader 512

threat modeling

components 241, 242
security 241

three-tier architecture 104

application layer 105
database layer 106
web layer 105

Ticket-Granting Server (TGS) 214

tightly coupled services 173

timeouts 40

time to live (TTL) 114

total cost of ownership (TCO) 466
calculating 290, 291

traffic rejection 40

transformer-based generative models 430, 431

Transmission Control Protocol (TCP) 223

Transport Layer Security (TLS) 210

Transport Security Layer (TSL) 234
Trusted Advisor Dashboard 273
trusted execution environments (TEEs) 210
Twelve-Factor App
 reference link 341

U

unattended upgrades 227
Unified Modeling Language (UML) 479
Uniform Resource Identifiers (URIs) 224
unstructured data stores 359
unsupervised learning 388
user acceptance testing (UAT) 90, 247, 331, 491
user data copy method 89
User Datagram Protocol (UDP) reflection 222
User Experience (UX) 105, 202 461, 487
user identity and access management 211-213
 Active Directory (AD) 215
 federated identity management (FIM) 213
 JSON Web Token (JWT) 220, 221
 Kerberos 214, 215
 Open Authorization (OAuth) 218, 219
 Security Assertion Markup Language (SAML) 216, 217
user interface (UI) 366, 461

V

Variational Autoencoders (VAEs) 428, 429
VectorDB 360, 361
 cons 360
 pros 360
vertical scaling 33
View 121
 responsibilities 121
Virtual Assistant (VA) 447

virtual machines (VMs) 62
Virtual Private Cloud (VPC) 133, 488
virtual private network (VPN) 235
Visual Studio Code (VS Code) 334
VM copy method 89
volume testing 204
VPC flow logs 230

W

WAF sandwich architecture 225
web application firewall (WAF) 198, 209, 224
web BFF 172
web layer 105
WebLoad 204
web security
 handling 221
web security mitigation 224
 DDoS mitigation 224, 225
 web application firewalls (WAFs) 224
Wide Area Networks (WANs) 193
workload rationalization 74

X

XGBoost 395
XSS attack 223

Z

Zeppelin notebook 367

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/978-1-83508-423-6>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

mail2divakar@gmail.com

