

2024 EDITION

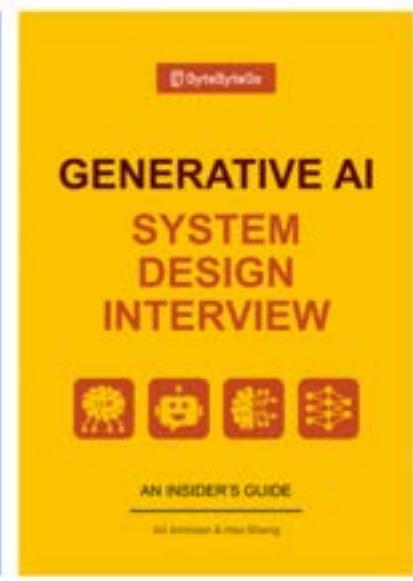
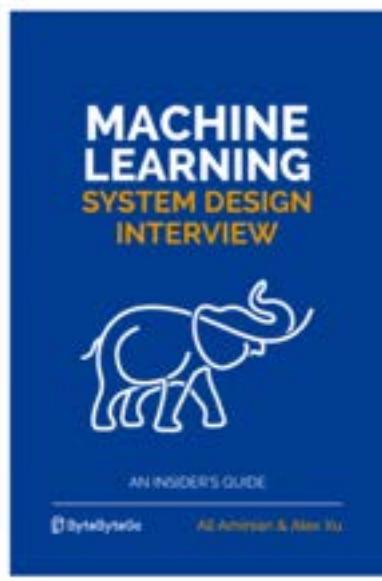
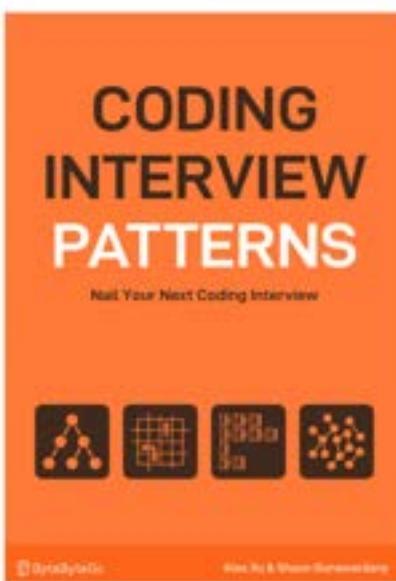
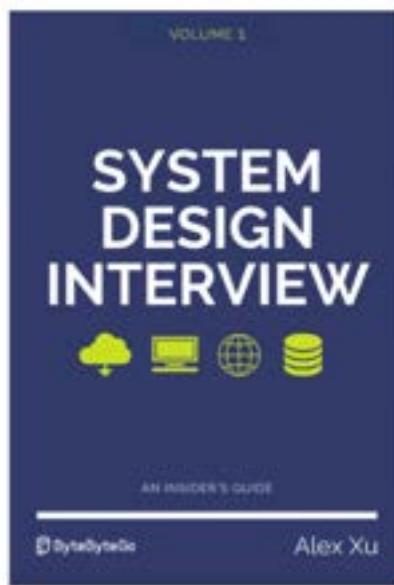
FREE

SYSTEM DESIGN

THE BIG ARCHIVE



The following books are available in paperback on [Amazon](#):



Big Endian vs Little Endian	7
How do we incorporate Event Sourcing into the systems?	9
How can Cache Systems go wrong	11
Linux file system explained	13
My recommended materials for cracking your next technical interview	14
How Git Commands work	16
Top 4 Most Popular Use Cases for UDP	17
How Does a Typical Push Notification System Work?	18
How can Cache Systems go wrong?	20
REST API Cheatsheet	22
Top 8 Programming Paradigms - Part 1	23
Data Pipelines Overview	25
API Vs SDK	27
A handy cheat sheet for the most popular cloud services	29
A nice cheat sheet of different monitoring infrastructure in cloud services	30
REST API Vs. GraphQL	32
Key Use Cases for Load Balancers	34
Top 6 Firewall Use Cases	36
Types of memory. Which ones do you know?	38
How Do C++, Java, Python Work?	40
Top 6 Load Balancing Algorithms	41
How does Git work?	43
HTTP Cookies Explained With a Simple Diagram	44
How does a ChatGPT-like system work?	45
A cheat sheet for system designs	47
Cloud Disaster Recovery Strategies	49
Visualizing a SQL query	51
How does REST API work?	52
Explaining 9 types of API testing	53
Git Merge vs. Rebase vs.Squash Commit!	55
What is a cookie?	57
How does a VPN work?	59
Top Software Architectural Styles	61
Understanding Database Types	63
Cloud Security Cheat Sheet	64
GitOps Workflow - Simplified Visual Guide	66
How does “scan to pay” work?	68
How do Search Engines Work?	70
The Payments Ecosystem	72

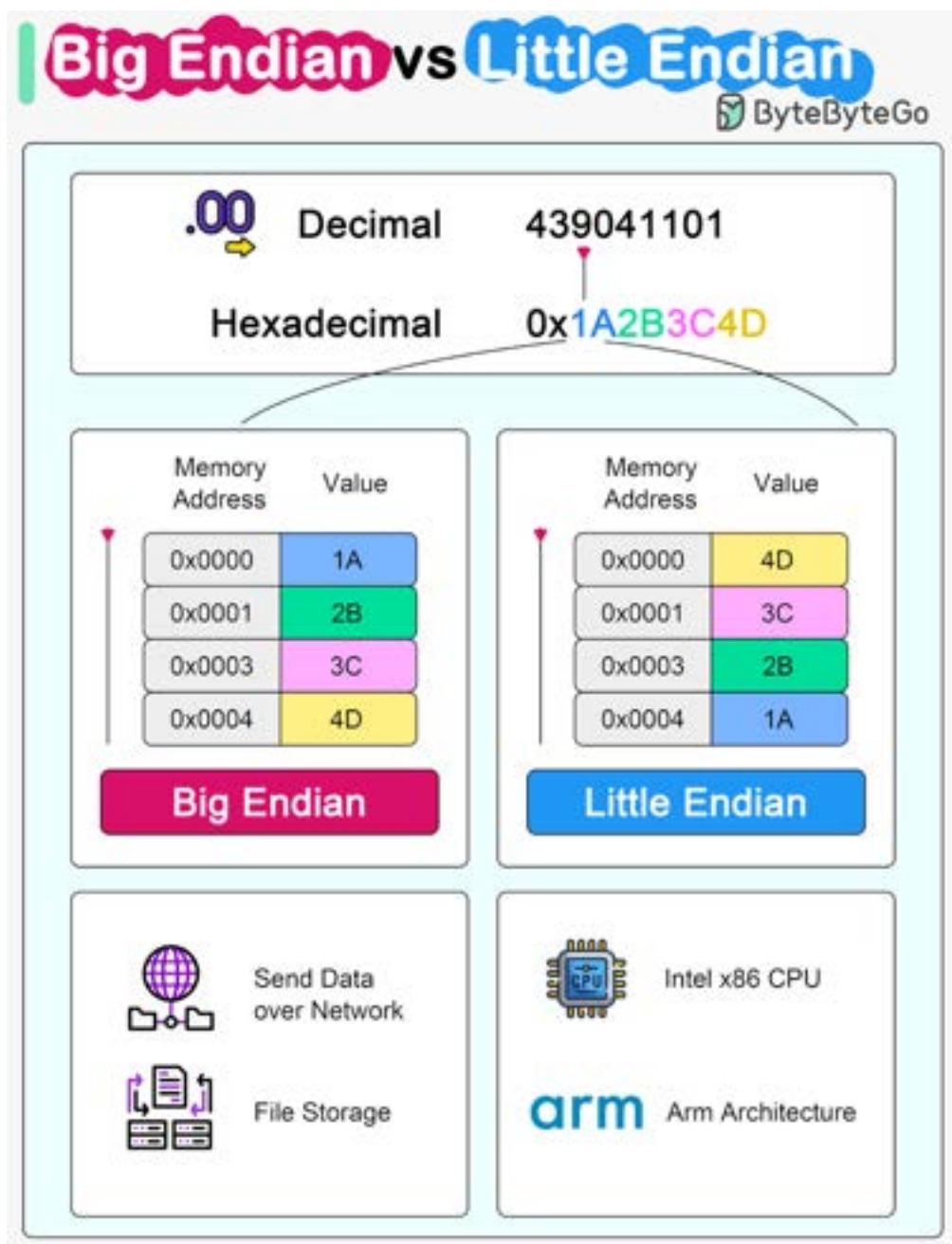
Object-oriented Programming: A Primer	74
Where do we cache data?	75
Flowchart of how slack decides to send a notification	77
What is the best way to learn SQL?	78
What is gRPC?	79
How do live streaming platforms like YouTube Live, TikTok Live, or Twitch work?	80
Linux Boot Process Illustrated	83
How does Visa make money?	85
Session, Cookie, JWT, Token, SSO, and OAuth 2.0 Explained in One Diagram	87
How do we manage configurations in a system?	89
What is CSS (Cascading Style Sheets)?	91
What is GraphQL? Is it a replacement for the REST API?	93
System Design Blueprint: The Ultimate Guide	95
Polling Vs Webhooks	97
How are notifications pushed to our phones or PCs?	99
9 best practices for developing microservices	101
Oauth 2.0 Explained With Simple Terms	102
How do companies ship code to production?	104
How do we manage sensitive data in a system?	106
Cloud Load Balancer Cheat Sheet	108
What does ACID mean?	110
CAP, BASE, SOLID, KISS, What do these acronyms mean?	112
System Design cheat sheet	114
How will you design the Stack Overflow website?	116
A nice cheat sheet of different cloud services	118
The one-line change that reduced clone times by a whopping 99%, says Pinterest	120
Best ways to test system functionality	122
Encoding vs Encryption vs Tokenization	
Encoding, encryption, and tokenization are three distinct processes that handle data in different ways for various purposes, including data transmission, security, and compliance.	124
Kubernetes Tools Stack Wheel	126
How does Docker work?	128
Top 6 Database Models	130
How do we detect node failures in distributed systems?	132
10 Good Coding Principles to improve code quality	134
15 Open-Source Projects That Changed the World	136
Reverse proxy vs. API gateway vs. load balancer	138
Linux Performance Observability Tools	140
Top 9 website performance metrics you cannot ignore	141
How do we manage data?	143
Postman vs. Insomnia vs. ReadyAPI vs. Thunder Client vs. Hoppscotch	145

How does gRPC work?	
RPC (Remote Procedure Call) is called “remote” because it enables communications between remote services when services are deployed to different servers under microservice architecture. From the user’s point of view, it acts like a local function call.	147
Have you heard of the 12-Factor App?	151
How does Redis architecture evolve?	153
Cloud Cost Reduction Techniques	155
Linux file permission illustrated	157
My Top 9 Favorite Engineering Blogs	158
9 Best Practices for Building Microservices	160
Roadmap for Learning Cyber Security	162
How does Javascript Work?	163
Can Kafka Lose Messages?	165
You're Decent at Linux if You Know What Those Directories Mean :)	167
Netflix's Tech Stack	169
Top 5 Kafka use cases	171
Top 6 Cloud Messaging Patterns.	172
How Netflix Really Uses Java?	175
Top 9 Architectural Patterns for Data and Communication Flow	177
What Are the Most Important AWS Services To Learn?	179
8 Key Data Structures That Power Modern Databases	181
How do we design effective and safe APIs?	182
Who are the Fantastic Four of System Design?	183
How do we design a secure system?	185
Things Every Developer Should Know: Concurrency is NOT parallelism.	187
HTTPS, SSL Handshake, and Data Encryption Explained to Kids.	189
Top 5 Software Architectural Patterns	191
Top 6 Tools to Turn Code into Beautiful Diagrams	193
Everything is a trade-off.	194
What is DevSecOps?	196
Top 8 Cache Eviction Strategies.	198
Linux Boot Process Explained	200
Unusual Evolution of the Netflix API Architecture	202
GET, POST, PUT... Common HTTP “verbs” in one figure	204
Top 8 C++ Use Cases	206
Top 4 data sharding algorithms explained.	208
10 years ago, Amazon found that every 100ms of latency cost them 1% in sales.	210
Load Balancer Realistic Use Cases You May Not Know	212
25 Papers That Completely Transformed the Computer World.	214
IPv4 vs. IPv6, what are the differences?	216
My Favorite 10 Books for Software Developers	218

Change Data Capture: Key to Leverage Real-Time Data	220
Netflix's Overall Architecture	222
Top 5 common ways to improve API performance.	224
How to diagnose a mysterious process that's taking too much CPU, memory, IO, etc?	226
What is a deadlock?	227
What's the difference between Session-based authentication and JWTs?	229
Top 9 Cases Behind 100% CPU Usage.	231
Top 6 ElasticSearch Use Cases.	233
AWS Services Cheat Sheet	235
How do computer programs run?	236
A cheat sheet for API designs.	238
Azure Services Cheat Sheet	240
Why is Kafka fast?	241
How do we retry on failures?	243
7 must-know strategies to scale your database.	245
Reddit's Core Architecture that helps it serve over 1 billion users every month.	247
Everything You Need to Know About Cross-Site Scripting (XSS).	249
15 Open-Source Projects That Changed the World	251
Types of Memory and Storage	253
How to load your websites at lightning speed?	254
25 Papers That Completely Transformed the Computer World.	256
10 Essential Components of a Production Web Application.	258
Top 8 Standards Every Developer Should Know.	259
Explaining JSON Web Token (JWT) with simple terms.	261
11 steps to go from Junior to Senior Developer.	262
Top 8 must-know Docker concepts	264
Top 10 Most Popular Open-Source Databases	266
What does a typical microservice architecture look like?	267
What is SSO (Single Sign-On)?	269
What makes HTTP2 faster than HTTP1?	271
Log Parsing Cheat Sheet	273
4 Ways Netflix Uses Caching to Hold User Attention	275
Top 6 Cases to Apply Idempotency.	277
MVC, MVP, MVVM, MVVM-C, and VIPER architecture patterns	279
What are the differences among database locks?	280
How do we Perform Pagination in API Design?	282
What happens when you type a URL into your browser?	284
How do you pay from your digital wallet by scanning the QR code?	286
What do Amazon, Netflix, and Uber have in common?	288
100X Postgres Scaling at Figma.	290
How to store passwords safely in the database and how to validate a password?	292

Cybersecurity 101 in one picture.	294
What do version numbers mean?	295
What is k8s (Kubernetes)?	297
HTTP Status Code You Should Know	299
18 Most-used Linux Commands You Should Know	300
Iterative, Agile, Waterfall, Spiral Model, RAD Model... What are the differences?	302
Design Patterns Cheat Sheet - Part 1 and Part 2	304
9 Essential Components of a Production Microservice Application	305
Which latency numbers you should know?	307
API Gateway 101	309
A Roadmap for Full-Stack Development.	310
OAuth 2.0 Flows	312
10 Key Data Structures We Use Every Day	313
Top 10 k8s Design Patterns	315
What is a Load Balancer?	317
8 Common System Design Problems and Solutions	319
How does SSH work?	321
How to load your websites at lightning speed?	322
Why is Nginx so popular?	324
How Discord Stores Trillions of Messages	325
How does Garbage Collection work?	327
A Cheat Sheet for Designing Fault-Tolerant Systems.	329
If you don't know trade-offs, you DON'T KNOW system design.	331
8 Tips for Efficient API Design.	333
The Ultimate Kafka 101 You Cannot Miss	335
A Cheatsheet for UML Class Diagrams	336
20 Popular Open Source Projects Started or Supported By Big Companies	339
A Crash Course on Database Sharding	341
Is PostgreSQL eating the database world?	343
The Ultimate Software Architect Knowledge Map	344
A Crash Course on Scaling the Data Layer	347
How can Cache Systems go wrong?	348
4 Popular GraphQL Adoption Patterns	350
Top 8 Popular Network Protocols	352
11 Things I learned about API Development from POST/CON 2024 by Postman.	353
How do Search Engines really Work?	355
The Ultimate Walkthrough of the Generative AI Landscape	357
Cheatsheet on Relational Database Design	358
My Favorite 10 Soft Skill Books that Can Help You Become a Better Developer	360
REST API Authentication Methods	362
The Evolving Landscape of API Protocols	366

Big Endian vs Little Endian



Microprocessor architectures commonly use two different methods to store the individual bytes in memory. This difference is referred to as “byte ordering” or “ endian nature”.

- **Little Endian**

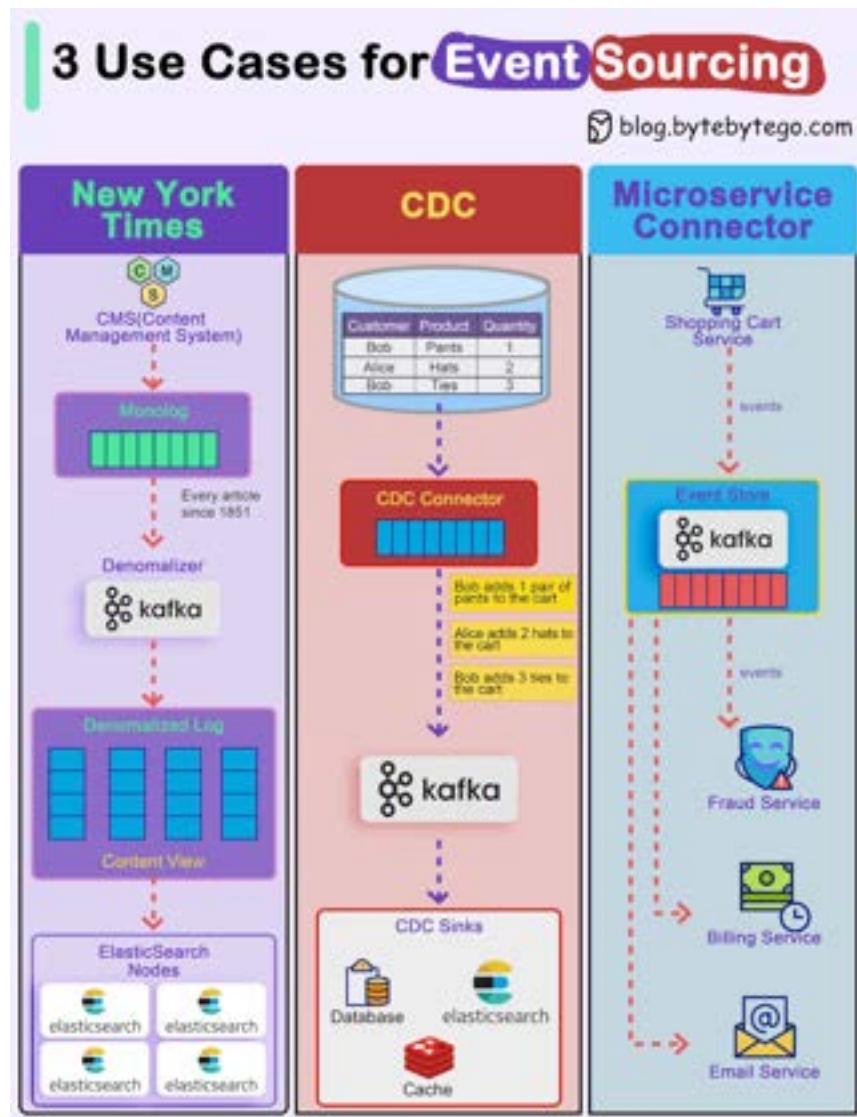
Intel x86 processors store a two-byte integer with the least significant byte first, followed by the most significant byte. This is called little-endian byte ordering.

- **Big Endian**

In big endian byte order, the most significant byte is stored at the lowest memory address, and the least significant byte is stored at the highest memory address. Older PowerPC and Motorola 68k architectures often use big endian. In network communications and file storage, we also use big endian.

The byte ordering becomes significant when data is transferred between systems or processed by systems with different endianness. It's important to handle byte order correctly to interpret data consistently across diverse systems.

How do we incorporate Event Sourcing into the systems?



Event sourcing changes the programming paradigm from persisting states to persisting events. The event store is the source of truth. Let's look at three examples.

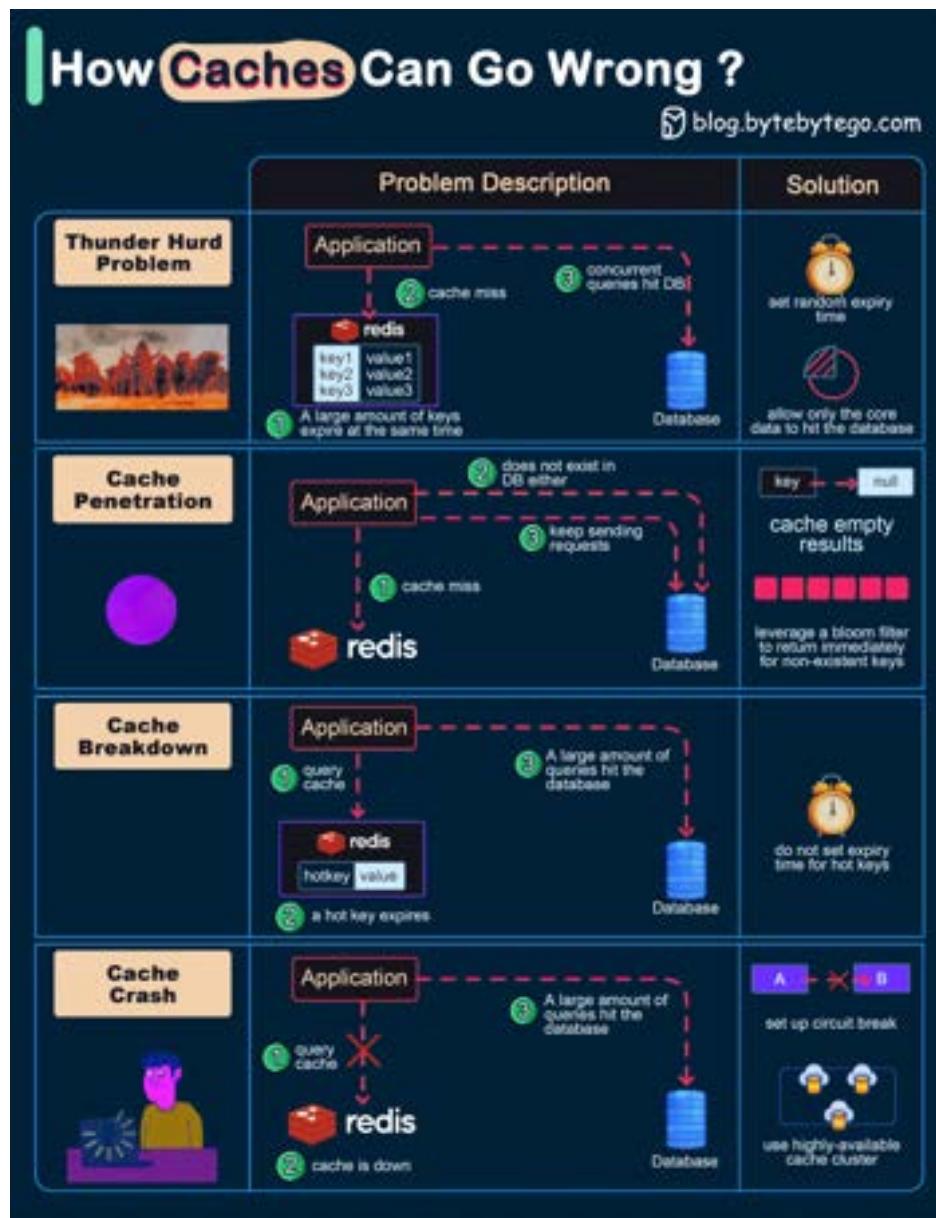
1. **New York Times**
The newspaper website stores every article, image, and byline since 1851 in an event store. The raw data is then denormalized into different views and fed into different ElasticSearch nodes for website searches.
2. **CDC (Change Data Capture)**
A CDC connector pulls data from the tables and transforms it into events. These events are pushed to Kafka and other sinks consume events from Kafka.
3. **Microservice Connector**
We can also use event sourcing paradigm for transmitting events among microservices. For example, the shopping cart service generates various events for adding

or removing items from the cart. Kafka broker acts as the event store, and other services including the fraud service, billing service, and email service consume events from the event store. Since events are the source of truth, each service can determine the domain model on its own.

Over to you: Have you used event sourcing in production?

How can Cache Systems go wrong

The diagram below shows 4 typical cases where caches can go wrong and their solutions.



1. Thunder herd problem

This happens when a large number of keys in the cache expire at the same time. Then the query requests directly hit the database, which overloads the database.

There are two ways to mitigate this issue: one is to avoid setting the same expiry time for the keys, adding a random number in the configuration; the other is to allow only the core business data to hit the database and prevent non-core data to access the database until the cache is back up.

2. Cache penetration

This happens when the key doesn't exist in the cache or the database. The application cannot retrieve relevant data from the database to update the cache. This problem creates a lot of pressure on both the cache and the database.

To solve this, there are two suggestions. One is to cache a null value for non-existent keys, avoiding hitting the database. The other is to use a bloom filter to check the key existence first, and if the key doesn't exist, we can avoid hitting the database.

3. Cache breakdown

This is similar to the thunder herd problem. It happens when a hot key expires. A large number of requests hit the database.

Since the hot keys take up 80% of the queries, we do not set an expiration time for them.

4. Cache crash

This happens when the cache is down and all the requests go to the database.

There are two ways to solve this problem. One is to set up a circuit breaker, and when the cache is down, the application services cannot visit the cache or the database. The other is to set up a cluster for the cache to improve cache availability.

Over to you: Have you met any of these issues in production?

Linux file system explained



The Linux file system used to resemble an unorganized town where individuals constructed their houses wherever they pleased. However, in 1994, the Filesystem Hierarchy Standard (FHS) was introduced to bring order to the Linux file system.

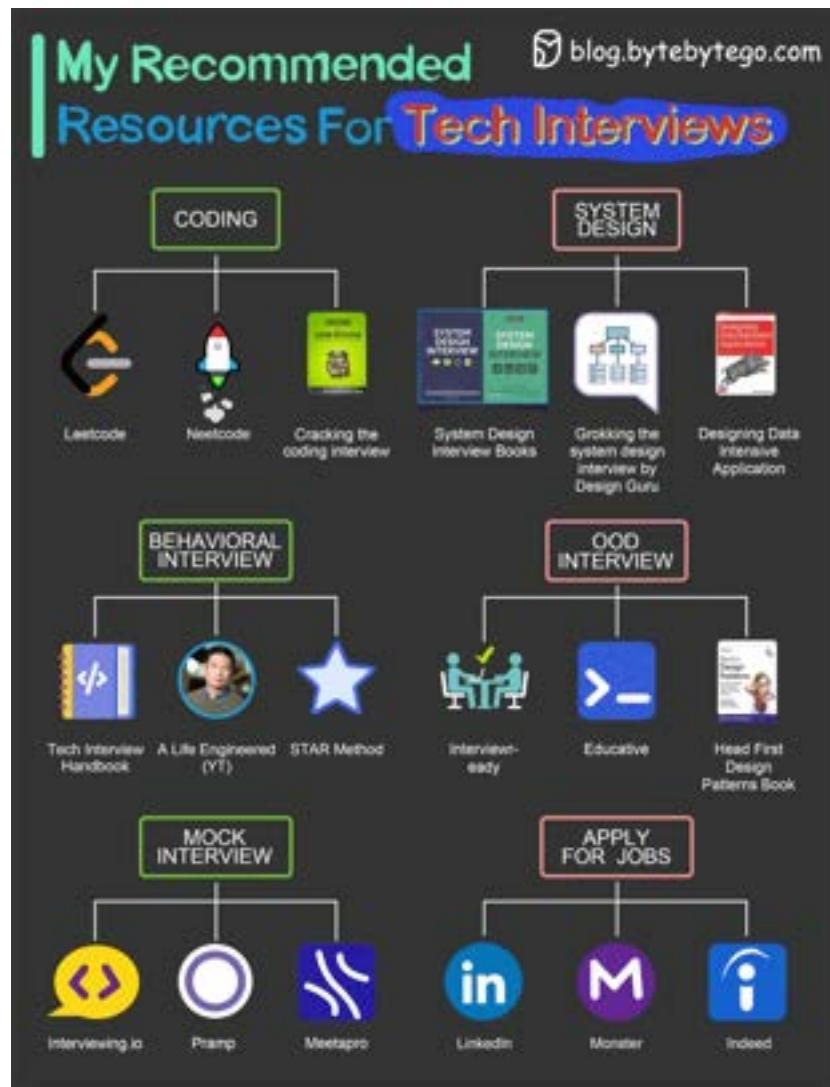
By implementing a standard like the FHS, software can ensure a consistent layout across various Linux distributions. Nonetheless, not all Linux distributions strictly adhere to this standard. They often incorporate their own unique elements or cater to specific requirements.

To become proficient in this standard, you can begin by exploring. Utilize commands such as "cd" for navigation and "ls" for listing directory contents. Imagine the file system as a tree, starting from the root ('/'). With time, it will become second nature to you, transforming you into a skilled Linux administrator.

Have fun exploring!

Over to you: which directory did you use most frequently?

My recommended materials for cracking your next technical interview



Coding

- Leetcode
- Cracking the coding interview book
- Neetcode

System Design Interview

- System Design Interview Book 1, 2 by Alex Xu, Sahn Lam
- Grokking the system design by Design Guru
- Design Data-intensive Application book

Behavioral interview

- Tech Interview Handbook (Github repo)
- A Life Engineered (YT)
- STAR method (general method)

OOD Interview

- Interviewready
- OOD by educative
- Head First Design Patterns Book

Mock interviews

- Interviewingio
- Pramp
- Meetapro

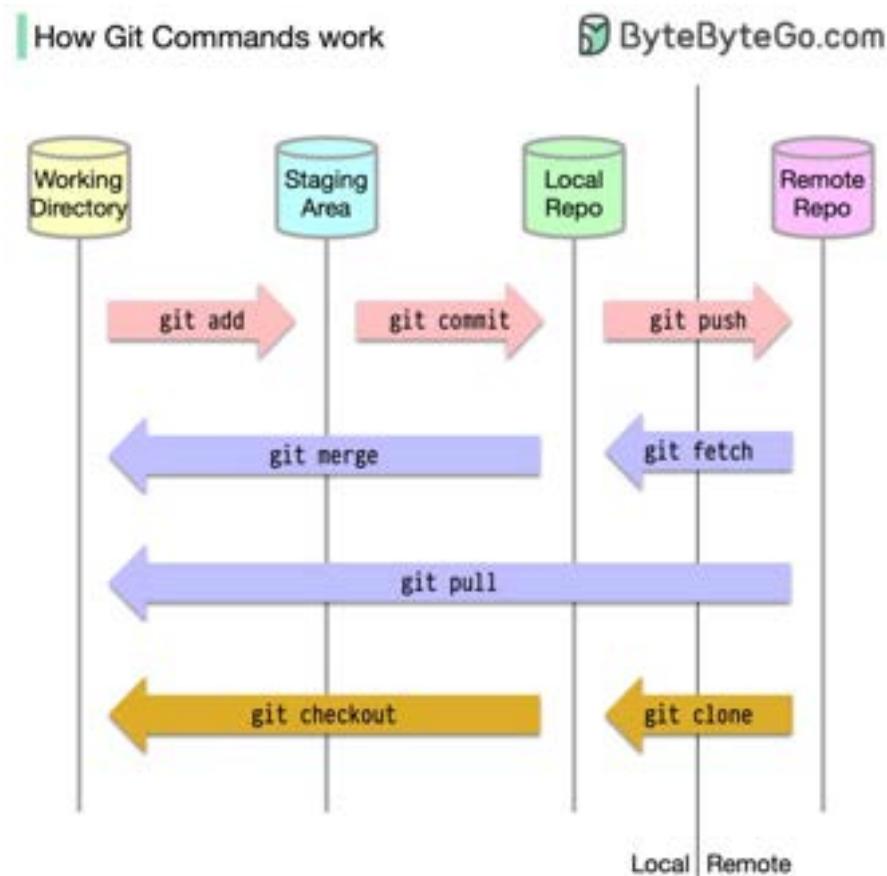
Apply for Jobs

- Linkedin
- Monster
- Indeed

Over to you: What is your favorite interview prep material?

How Git Commands work

Almost every software engineer has used Git before, but only a handful know how it works.



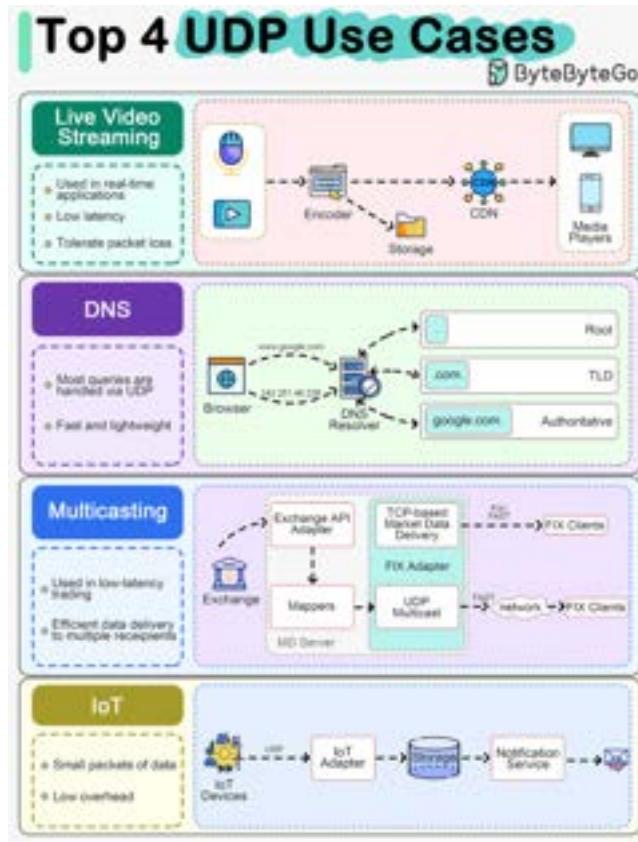
To begin with, it's essential to identify where our code is stored. The common assumption is that there are only two locations - one on a remote server like Github and the other on our local machine. However, this isn't entirely accurate. Git maintains three local storages on our machine, which means that our code can be found in four places:

- Working directory: where we edit files
- Staging area: a temporary location where files are kept for the next commit
- Local repository: contains the code that has been committed
- Remote repository: the remote server that stores the code

Most Git commands primarily move files between these four locations.

Over to you: Do you know which storage location the "git tag" command operates on? This command can add annotations to a commit.

Top 4 Most Popular Use Cases for UDP

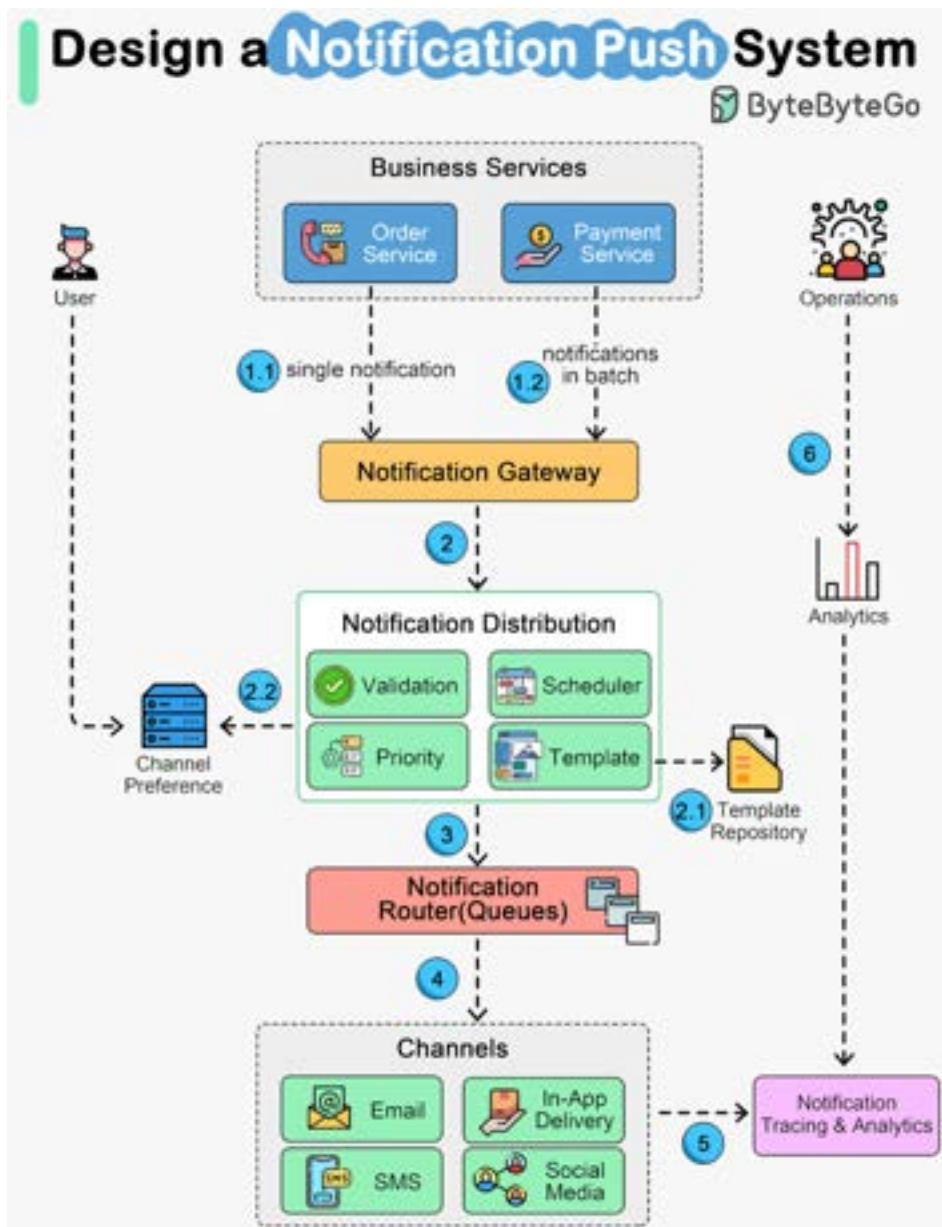


UDP (User Datagram Protocol) is used in various software architectures for its simplicity, speed, and low overhead compared to other protocols like TCP.

- **Live Video Streaming**
Many VoIP and video conferencing applications leverage UDP due to its lower overhead and ability to tolerate packet loss. Real-time communication benefits from UDP's reduced latency compared to TCP.
- **DNS**
DNS (Domain Name Service) queries typically use UDP for their fast and lightweight nature. Although DNS can also use TCP for large responses or zone transfers, most queries are handled via UDP.
- **Market Data Multicast**
In low-latency trading, UDP is utilized for efficient market data delivery to multiple recipients simultaneously.
- **IoT**
UDP is often used in IoT devices for communications, sending small packets of data between devices.

How Does a Typical Push Notification System Work?

The diagram below shows the architecture of a notification system that covers major notification channels:

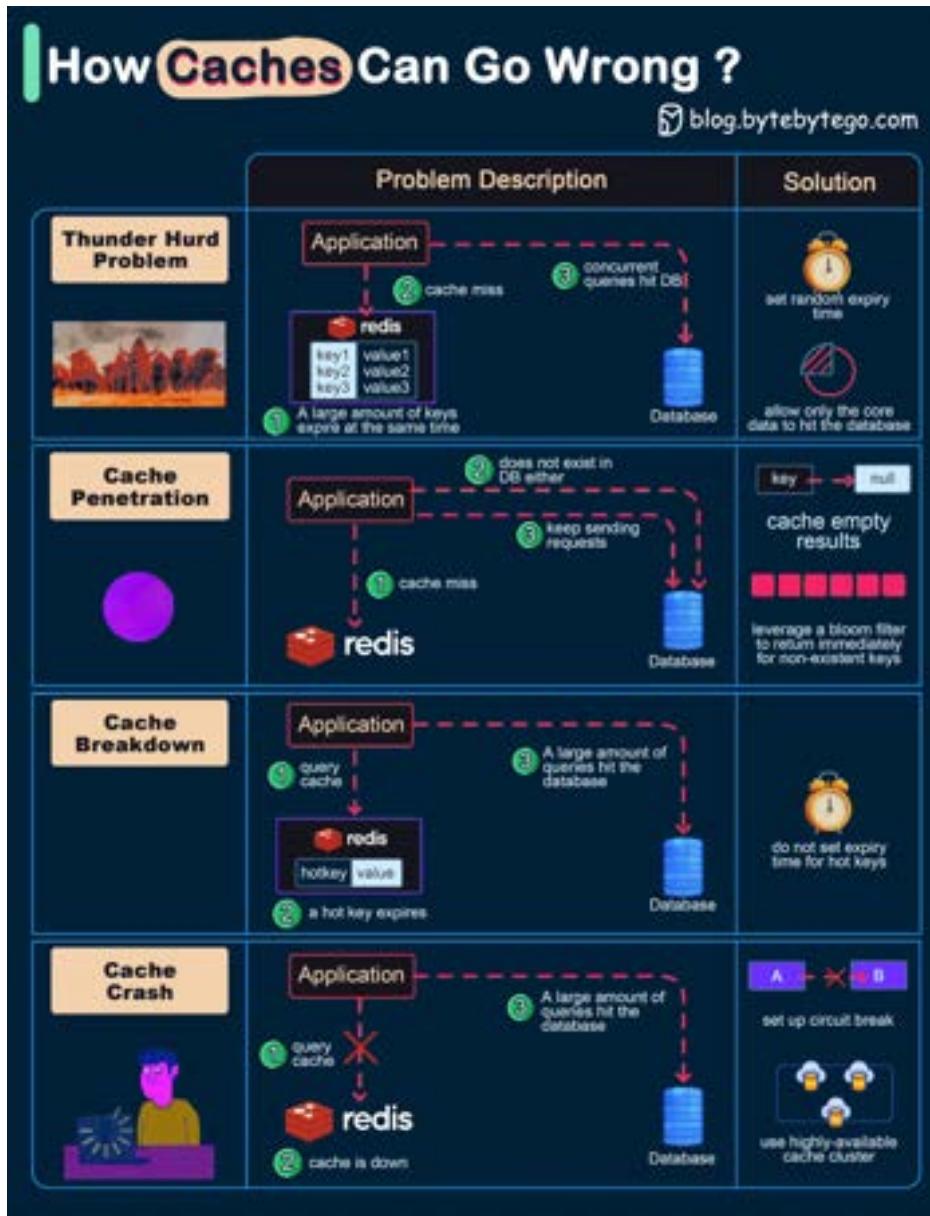


- In-App notifications
- Email notifications
- SMS and OTP notifications
- Social media pushes

Let's walk through the steps.

- Steps 1.1 and 1.2 - The business services send notifications to the notification gateway. The gateway can handle two modes: one mode receives one notification each time, and the other receives notifications in batches.
- Steps 2, 2.1, and 2.2 - The notification gateway forwards the notifications to the distribution service, where the messages are validated, formatted, and scheduled based on settings. The notification template repository allows users to pre-define the message format. The channel preference repository allows users to pre-define the preferred delivery channels.
- Step 3 - The notifications are then sent to the routers, normally message queues.
- Step 4 - The channel services communicate with various internal and external delivery channels, including in-app notifications, email delivery, SMS delivery, and social media apps.
- Steps 5 and 6 - The delivery metrics are captured by the notification tracking and analytics service, where the operations team can view the analytical reports and improve user experiences.

How can Cache Systems go wrong?



The diagram below shows 4 typical cases where caches can go wrong and their solutions.

1. Thunder herd problem

This happens when a large number of keys in the cache expire at the same time. Then the query requests directly hit the database, which overloads the database.

There are two ways to mitigate this issue: one is to avoid setting the same expiry time for the keys, adding a random number in the configuration; the other is to allow only the core business data to hit the database and prevent non-core data to access the database until the cache is back up.

2. Cache penetration

This happens when the key doesn't exist in the cache or the database. The application cannot retrieve relevant data from the database to update the cache. This problem creates a lot of pressure on both the cache and the database.

To solve this, there are two suggestions. One is to cache a null value for non-existent keys, avoiding hitting the database. The other is to use a bloom filter to check the key existence first, and if the key doesn't exist, we can avoid hitting the database.

3. Cache breakdown

This is similar to the thunder herd problem. It happens when a hot key expires. A large number of requests hit the database.

Since the hot keys take up 80% of the queries, we do not set an expiration time for them.

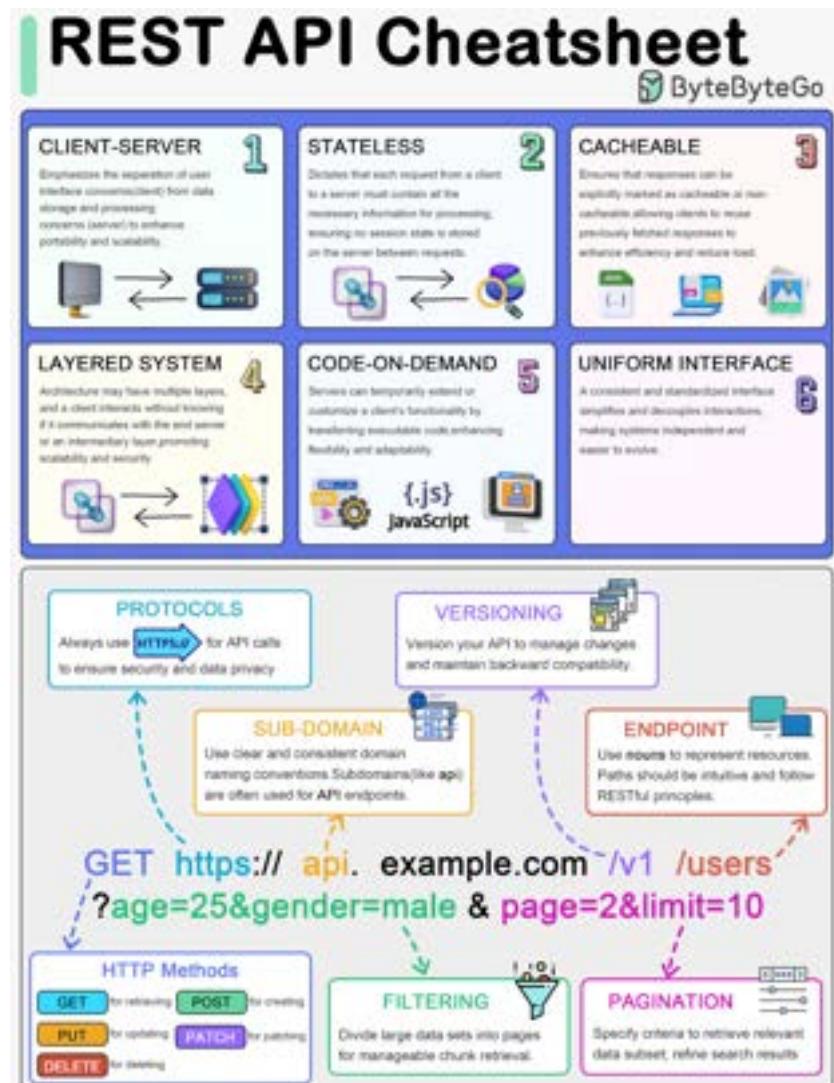
4. Cache crash

This happens when the cache is down and all the requests go to the database.

There are two ways to solve this problem. One is to set up a circuit breaker, and when the cache is down, the application services cannot visit the cache or the database. The other is to set up a cluster for the cache to improve cache availability.

Over to you: Have you met any of these issues in production?

REST API Cheatsheet



This guide is designed to help you understand the world of RESTful APIs in a clear and engaging way.

What's inside:

- An exploration of the six fundamental principles of REST API design.
- Insights into key components such as HTTP methods, protocols, versioning, and more.
- A special focus on practical aspects like pagination, filtering, and endpoint design.

Whether you're beginning your API journey or looking to refresh your knowledge, this blog and cheat sheet combo is the perfect toolkit for success.

Top 8 Programming Paradigms - Part 1



- **Imperative Programming**

Imperative programming describes a sequence of steps that change the program's state. Languages like C, C++, Java, Python (to an extent), and many others support imperative programming styles.

- **Declarative Programming**

Declarative programming emphasizes expressing logic and functionalities without describing the control flow explicitly. Functional programming is a popular form of declarative programming.

- **Object-Oriented Programming (OOP)**

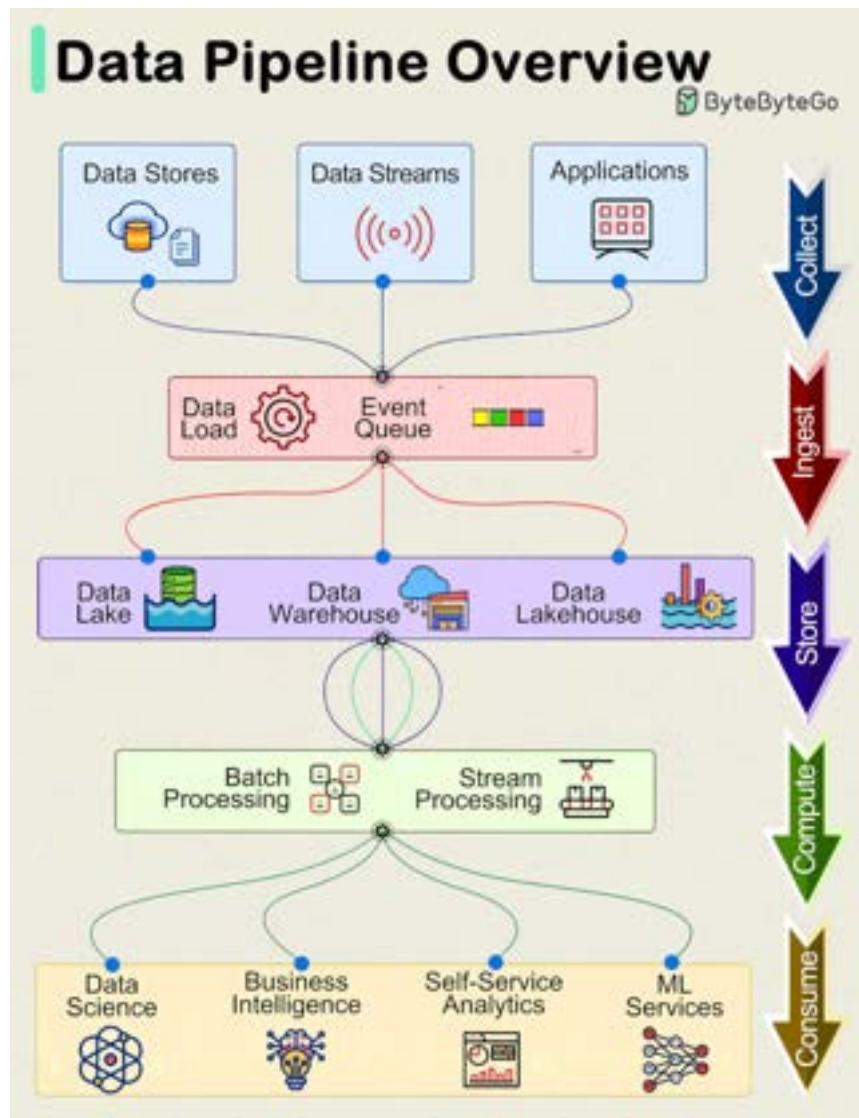
Object-oriented programming (OOP) revolves around the concept of objects, which encapsulate data (attributes) and behavior (methods or functions). Common object-oriented programming languages include Java, C++, Python, Ruby, and C#.

- **Aspect-Oriented Programming (AOP)**

Aspect-oriented programming (AOP) aims to modularize concerns that cut across multiple parts of a software system. AspectJ is one of the most well-known AOP frameworks that extends Java with AOP capabilities.

- Functional Programming
Functional Programming (FP) treats computation as the evaluation of mathematical functions and emphasizes the use of immutable data and declarative expressions. Languages like Haskell, Lisp, Erlang, and some features in languages like JavaScript, Python, and Scala support functional programming paradigms.
- Reactive Programming
Reactive Programming deals with asynchronous data streams and the propagation of changes. Event-driven applications, and streaming data processing applications benefit from reactive programming.
- Generic Programming
Generic Programming aims at creating reusable, flexible, and type-independent code by allowing algorithms and data structures to be written without specifying the types they will operate on. Generic programming is extensively used in libraries and frameworks to create data structures like lists, stacks, queues, and algorithms like sorting, searching.
- Concurrent Programming
Concurrent Programming deals with the execution of multiple tasks or processes simultaneously, improving performance and resource utilization. Concurrent programming is utilized in various applications, including multi-threaded servers, parallel processing, concurrent web servers, and high-performance computing.

Data Pipelines Overview



Data pipelines are a fundamental component of managing and processing data efficiently within modern systems. These pipelines typically encompass 5 predominant phases: Collect, Ingest, Store, Compute, and Consume.

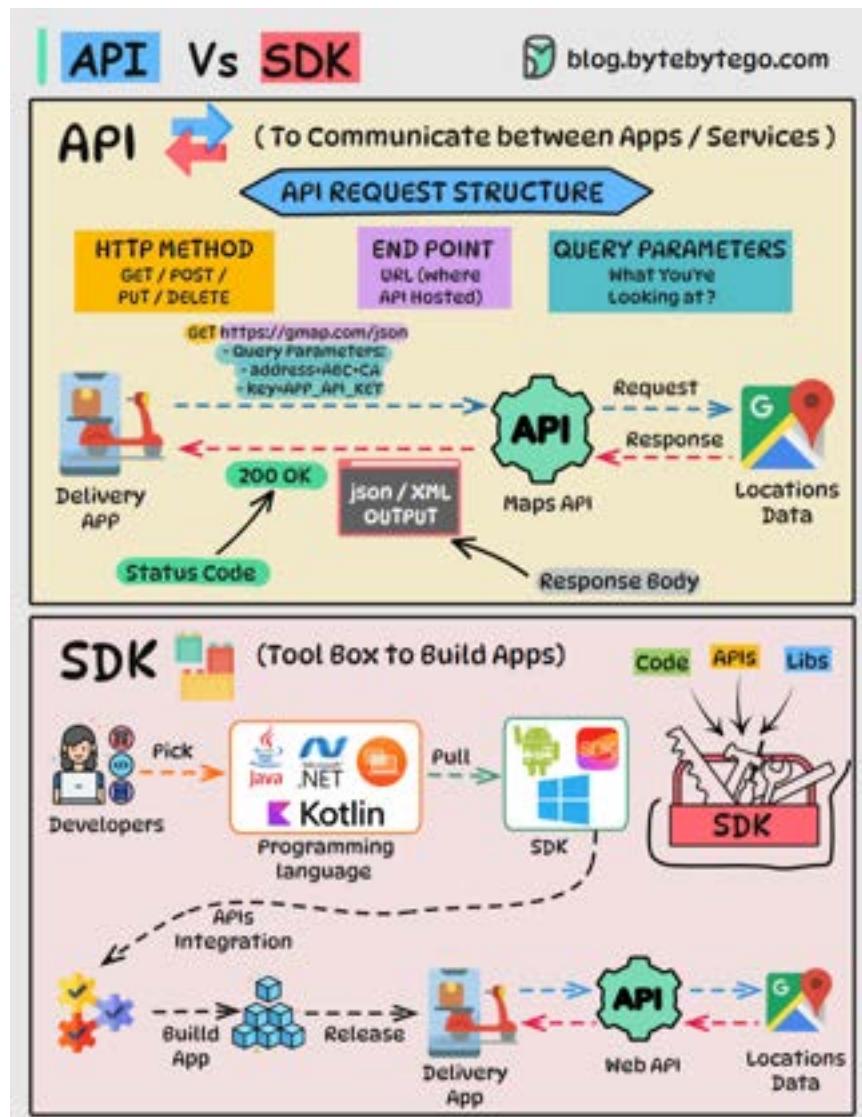
1. Collect:
Data is acquired from data stores, data streams, and applications, sourced remotely from devices, applications, or business systems.
 2. Ingest:
During the ingestion process, data is loaded into systems and organized within event queues.
 3. Store:
Post ingestion, organized data is stored in data warehouses, data lakes, and data lakehouses, along with various systems like databases, ensuring post-ingestion storage.

4. Compute:
Data undergoes aggregation, cleansing, and manipulation to conform to company standards, including tasks such as format conversion, data compression, and partitioning. This phase employs both batch and stream processing techniques.
5. Consume:
Processed data is made available for consumption through analytics and visualization tools, operational data stores, decision engines, user-facing applications, dashboards, data science, machine learning services, business intelligence, and self-service analytics.

The efficiency and effectiveness of each phase contribute to the overall success of data-driven operations within an organization.

Over to you: What's your story with data-driven pipelines? How have they influenced your data management game?

API Vs SDK



API (Application Programming Interface) and SDK (Software Development Kit) are essential tools in the software development world, but they serve distinct purposes:

API: An API is a set of rules and protocols that allows different software applications and services to communicate with each other.

1. It defines how software components should interact.
2. Facilitates data exchange and functionality access between software components.
3. Typically consists of endpoints, requests, and responses.

SDK: An SDK is a comprehensive package of tools, libraries, sample code, and documentation that assists developers in building applications for a particular platform, framework, or hardware.

1. Offers higher-level abstractions, simplifying development for a specific platform.
2. Tailored to specific platforms or frameworks, ensuring compatibility and optimal performance on that platform.
3. Offer access to advanced features and capabilities specific to the platform, which might be otherwise challenging to implement from scratch.

The choice between APIs and SDKs depends on the development goals and requirements of the project.

Over to you: Which do you find yourself gravitating towards – APIs or SDKs – Every implementation has a unique story to tell. What's yours?

A handy cheat sheet for the most popular cloud services



What's included?

- AWS, Azure, Google Cloud, Oracle Cloud, Alibaba Cloud
- Cloud servers
- Databases
- Message queues and streaming platforms
- Load balancing, DNS routing software
- Security
- Monitoring

Over to you - which company is the best at naming things?

A nice cheat sheet of different monitoring infrastructure in cloud services

MONITORING CHEAT SHEET <small>blog.bytebybyte.com</small>				
Element	AWS	Google Cloud	Azure	Open Source / 3rd Party
Data Collection	Cloud Watch Cloud Watch Logs Cloud Trail Config Custom agents / Scripts	Cloud Monitoring Cloud Logging Cloud Audit Logs Custom agents / Scripts	Azure Monitor Azure Activity Log Azure Policy Security Center Custom agents / Scripts	ZABBIX Prometheus CloudWatch Metrics Insights Cloud Operations
Data Storage	S3	Cloud Storage	Blob Storage	MINIO GLUSTER ceph
Data Analysis	CloudWatch Metrics Insights	Cloud Operations	Azure Monitor Metrics Explorer	Grafana kibana
Alerting	SNS	Cloud Monitoring Alerts	Azure Monitor Alerts	PagerDuty slack
Visualization	CloudWatch Dashboard QuickSight	Cloud Monitoring Dashboard Data Studio	Azure Monitor Dashboard Power BI	Grafana Superset Metabase redash
Reporting and Compliance	Config Rules Trusted Advisor	Security Command Center	Policy Compliance Security Center Compliance	OpenSCAP CISQy
Automation	Lambda Step Functions	Cloud Functions	Azure Functions Azure Automation	Jenkins ANSIBLE
Integration	CloudFormation CodePipeline	Cloud Deployment Manager Cloud Build	Azure Automation Azure DevOps	Pulumi Terraform GitLab Jenkins Travis CI
Feedback Loop	Well-Architected Tool	Well-Architected Framework	Well-Architected Framework	Scout APM Cloud Custodian

This cheat sheet offers a concise yet comprehensive comparison of key monitoring elements across the three major cloud providers and open-source / 3rd party tools.

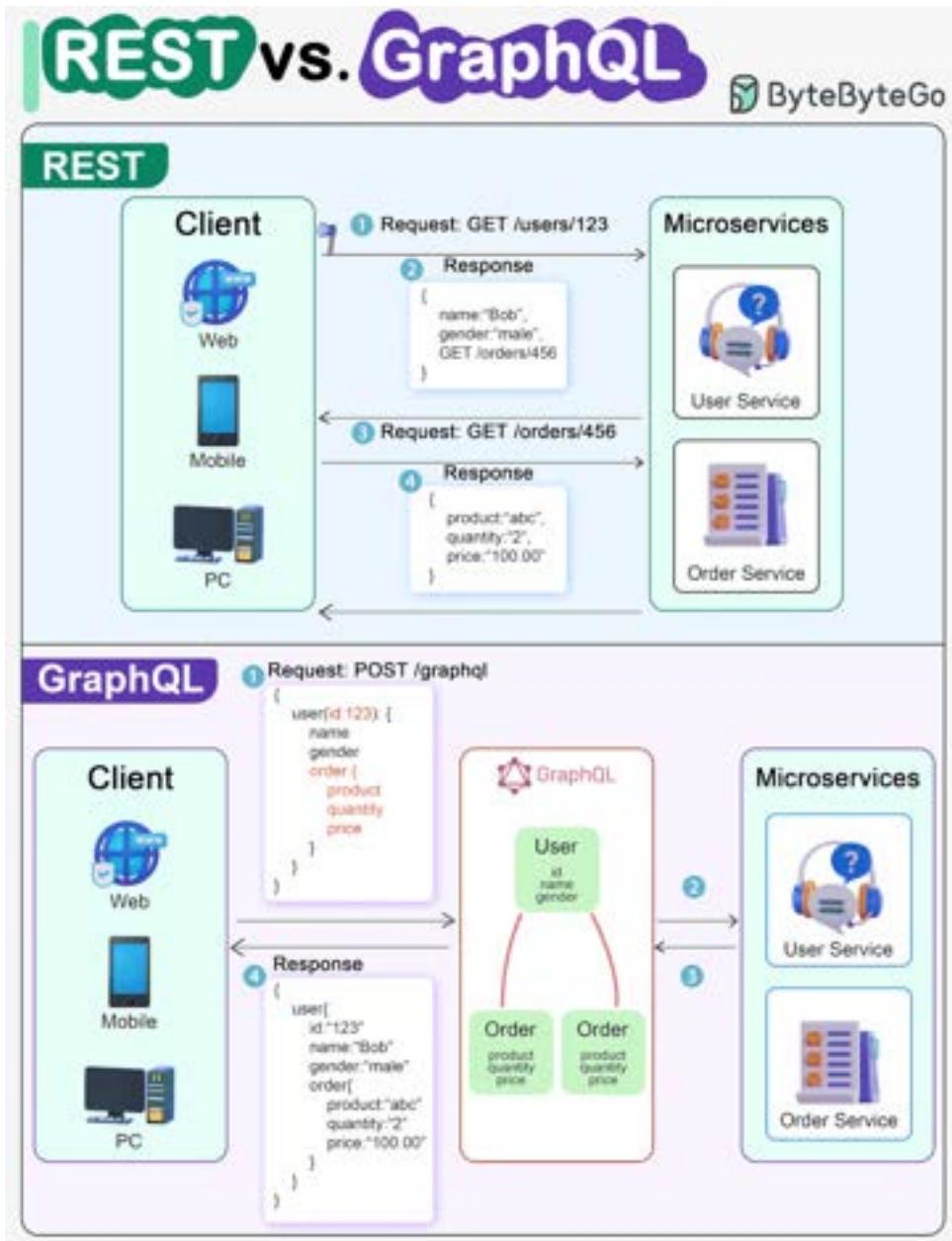
Let's delve into the essential monitoring aspects covered:

- Data Collection: Gather information from diverse sources to enhance decision-making.
- Data Storage: Safely store and manage data for future analysis and reference.
- Data Analysis: Extract valuable insights from data to drive informed actions.
- Alerting: Receive real-time notifications about critical events or anomalies.
- Visualization: Present data in a visually comprehensible format for better understanding.
- Reporting and Compliance: Generate reports and ensure adherence to regulatory standards.
- Automation: Streamline processes and tasks through automated workflows.

- Integration: Seamlessly connect and exchange data between different systems or tools.
- Feedback Loops: Continuously refine strategies based on feedback and performance analysis.

Over to you: How do you prioritize and leverage these essential monitoring aspects in your domain to achieve better outcomes and efficiency?

REST API Vs. GraphQL



When it comes to API design, REST and GraphQL each have their own strengths and weaknesses.

REST

- Uses standard HTTP methods like GET, POST, PUT, DELETE for CRUD operations.
- Works well when you need simple, uniform interfaces between separate services/applications.
- Caching strategies are straightforward to implement.
- The downside is it may require multiple roundtrips to assemble related data from separate endpoints.

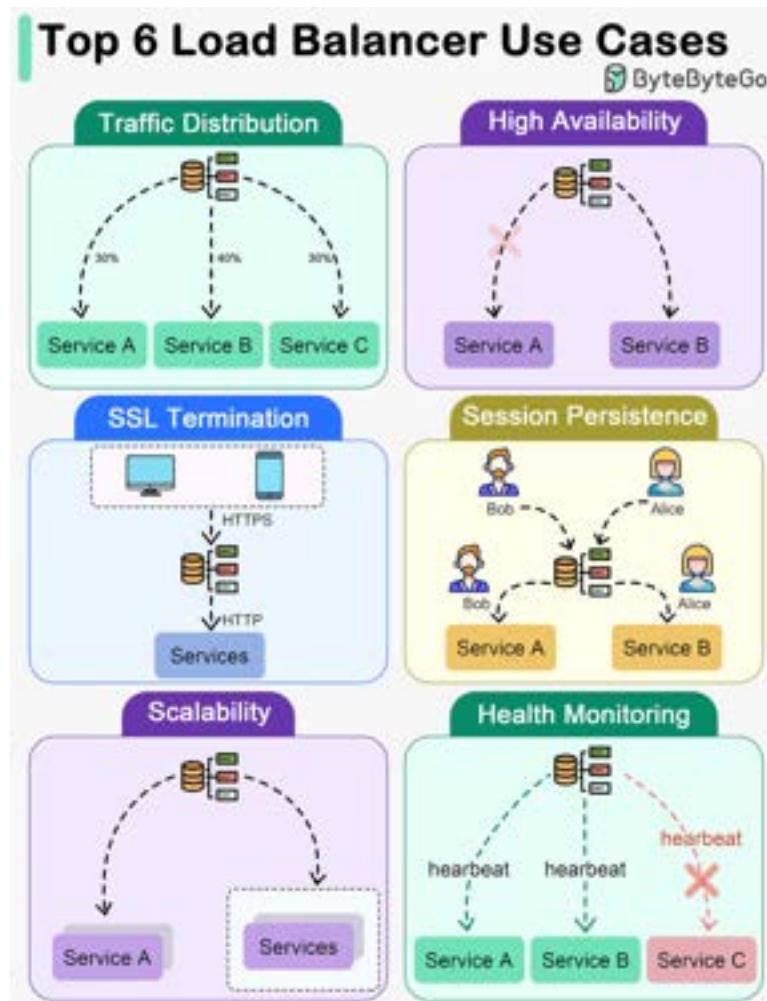
GraphQL

- Provides a single endpoint for clients to query for precisely the data they need.
- Clients specify the exact fields required in nested queries, and the server returns optimized payloads containing just those fields.
- Supports Mutations for modifying data and Subscriptions for real-time notifications.
- Great for aggregating data from multiple sources and works well with rapidly evolving frontend requirements.
- However, it shifts complexity to the client side and can allow abusive queries if not properly safeguarded
- Caching strategies can be more complicated than REST.

The best choice between REST and GraphQL depends on the specific requirements of the application and development team. GraphQL is a good fit for complex or frequently changing frontend needs, while REST suits applications where simple and consistent contracts are preferred.

Key Use Cases for Load Balancers

The diagram below shows top 6 use cases where we use a load balancer.



- **Traffic Distribution**
Load balancers evenly distribute incoming traffic among multiple servers, preventing any single server from becoming overwhelmed. This helps maintain optimal performance, scalability, and reliability of applications or websites.
- **High Availability**
Load balancers enhance system availability by rerouting traffic away from failed or unhealthy servers to healthy ones. This ensures uninterrupted service even if certain servers experience issues.
- **SSL Termination**
Load balancers can offload SSL/TLS encryption and decryption tasks from backend servers, reducing their workload and improving overall performance.

- **Session Persistence**

For applications that require maintaining a user's session on a specific server, load balancers can ensure that subsequent requests from a user are sent to the same server.

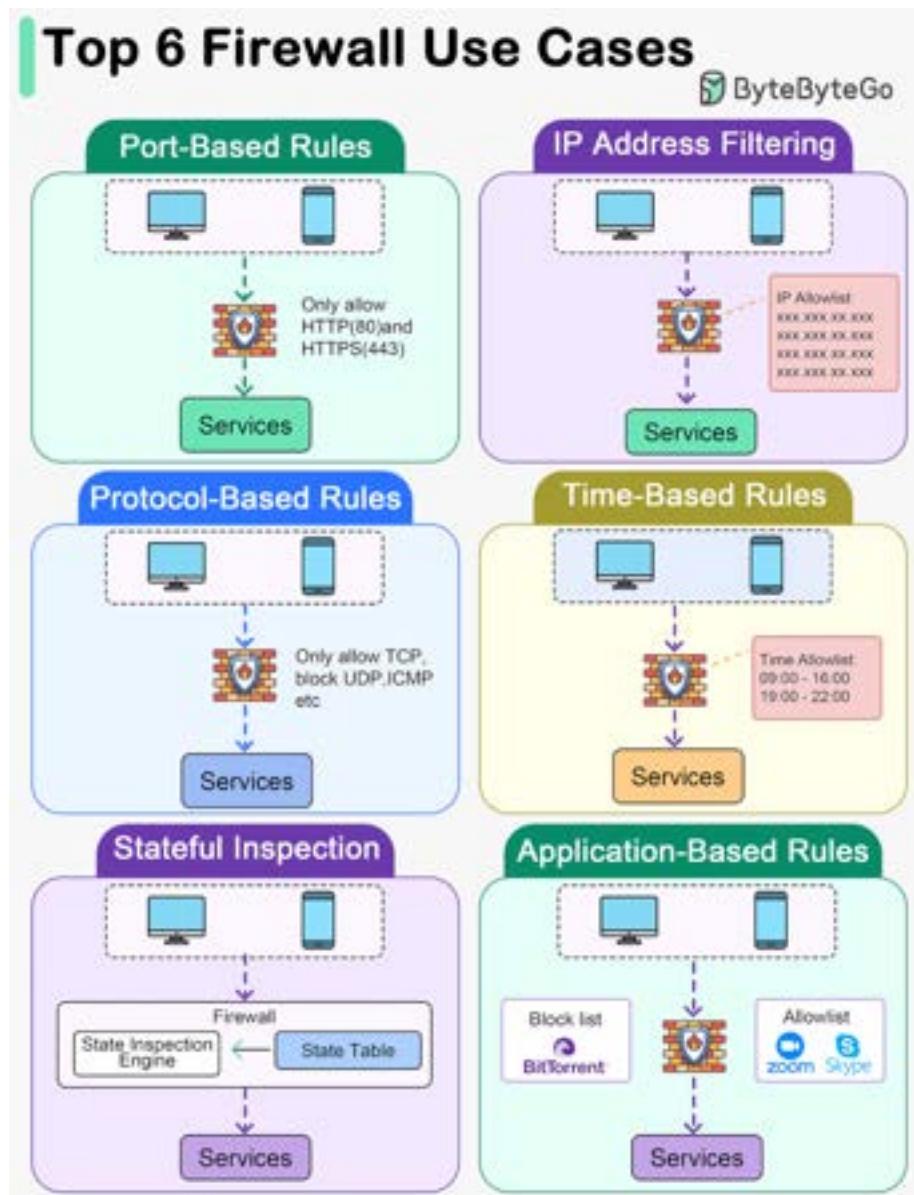
- **Scalability**

Load balancers facilitate horizontal scaling by effectively managing increased traffic. Additional servers can be easily added to the pool, and the load balancer will distribute traffic across all servers.

- **Health Monitoring**

Load balancers continuously monitor the health and performance of servers, removing failed or unhealthy servers from the pool to maintain optimal performance.

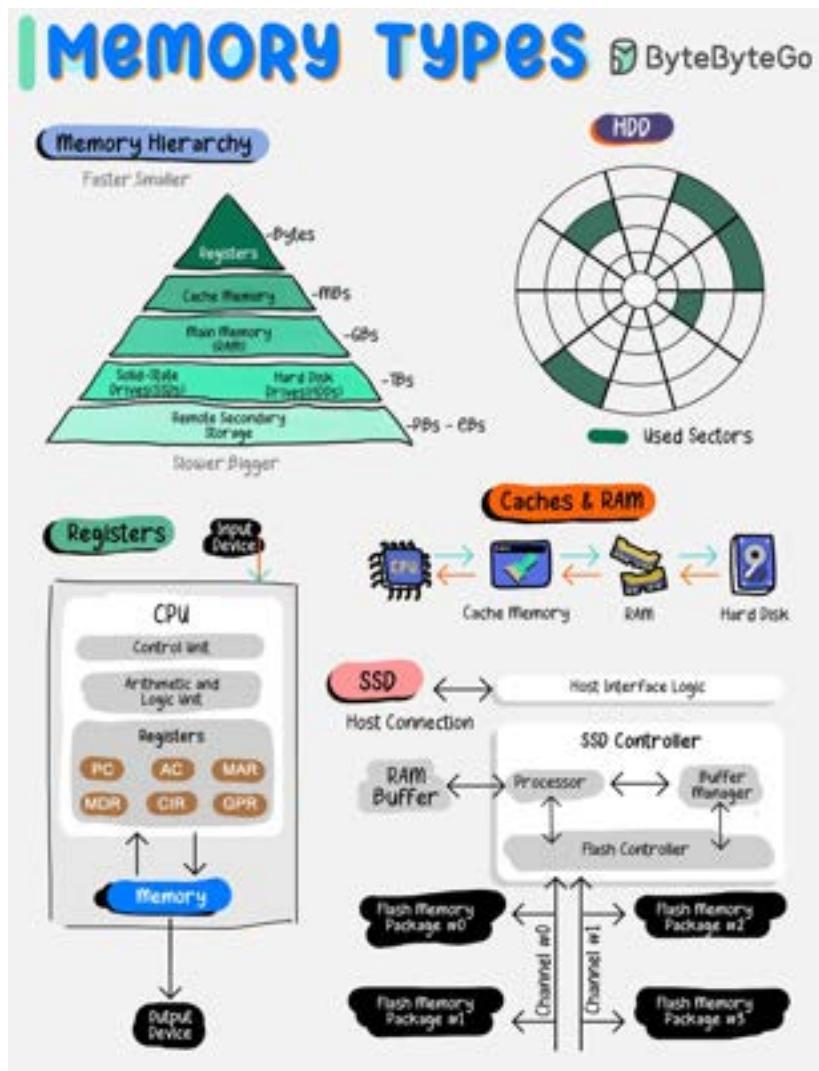
Top 6 Firewall Use Cases



- **Port-Based Rules**
Firewall rules can be set to allow or block traffic based on specific ports. For example, allowing only traffic on ports 80 (HTTP) and 443 (HTTPS) for web browsing.
- **IP Address Filtering**
Rules can be configured to allow or deny traffic based on source or destination IP addresses. This can include whitelisting trusted IP addresses or blacklisting known malicious ones.
- **Protocol-Based Rules**
Firewalls can be configured to allow or block traffic based on specific network protocols such as TCP, UDP, ICMP, etc. For instance, allowing only TCP traffic on port 22 (SSH).

- **Time-Based Rules**
Firewalls can be configured to enforce rules based on specific times or schedules. This can be useful for setting different access rules during business hours versus after-hours.
- **Stateful Inspection**
Stateful Inspection: Stateful firewalls monitor the state of active connections and allow traffic only if it matches an established connection, preventing unauthorized access from the outside.
- **Application-Based Rules**
Some firewalls offer application-level control by allowing or blocking traffic based on specific applications or services. For instance, allowing or restricting access to certain applications like Skype, BitTorrent, etc.

Types of memory. Which ones do you know?



Memory types vary by speed, size, and function, creating a multi-layered architecture that balances cost with the need for rapid data access.

By grasping the roles and capabilities of each memory type, developers and system architects can design systems that effectively leverage the strengths of each storage layer, leading to improved overall system performance and user experience.

Some of the common Memory types are:

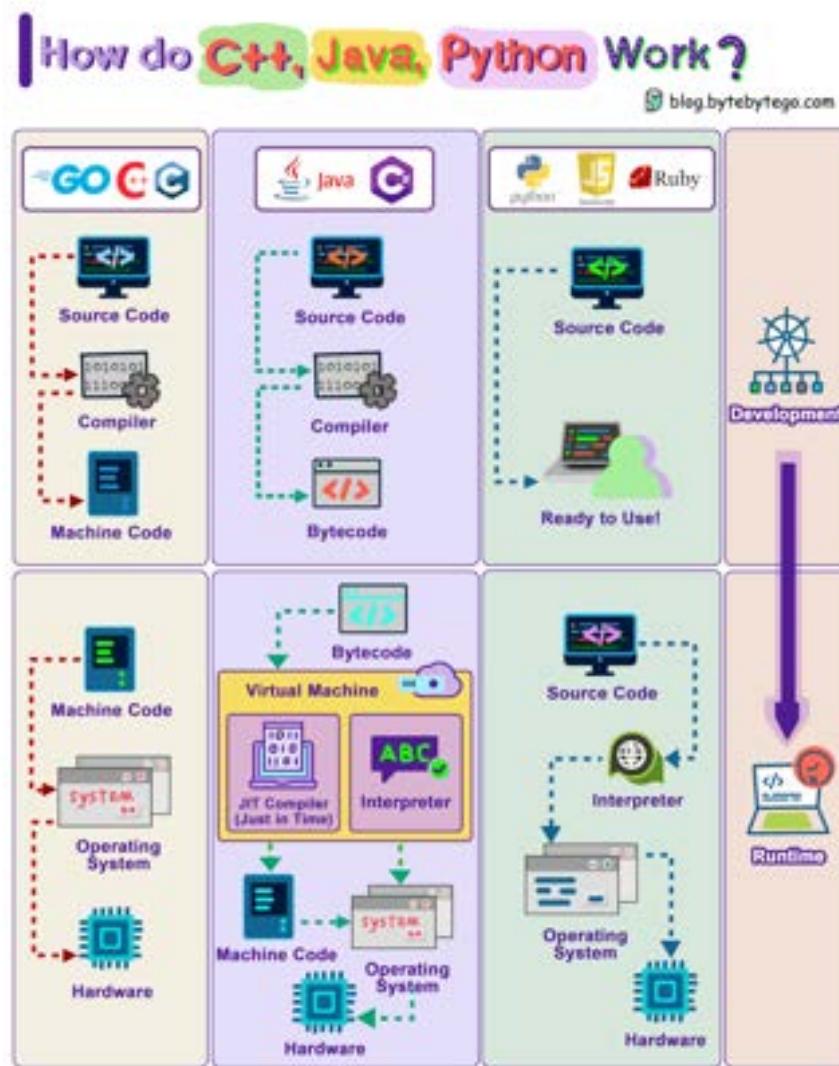
1. **Registers:**
Tiny, ultra-fast storage within the CPU for immediate data access.
2. **Caches:**
Small, quick memory located close to the CPU to speed up data retrieval.

3. Main Memory (RAM):
Larger, primary storage for currently executing programs and data.
4. Solid-State Drives (SSDs):
Fast, reliable storage with no moving parts, used for persistent data.
5. Hard Disk Drives (HDDs):
Mechanical drives with large capacities for long-term storage.
6. Remote Secondary Storage:
Offsite storage for data backup and archiving, accessible over a network.

Over to you: Which memory type resonates most with your tech projects and why? Share your thoughts!

How Do C++, Java, Python Work?

The diagram shows how the compilation and execution work.



Compiled languages are compiled into machine code by the compiler. The machine code can later be executed directly by the CPU. Examples: C, C++, Go.

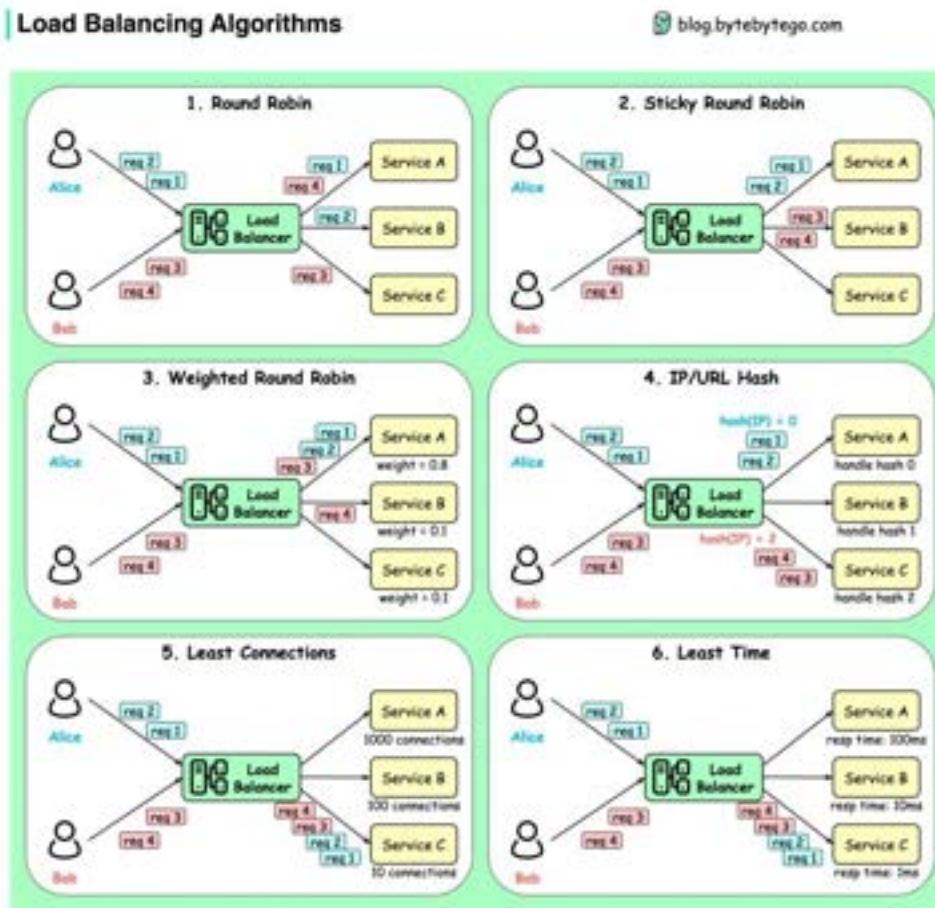
A bytecode language like Java, compiles the source code into bytecode first, then the JVM executes the program. Sometimes the JIT (Just-In-Time) compiler compiles the source code into machine code to speed up the execution. Examples: Java, C#

Interpreted languages are not compiled. They are interpreted by the interpreter during runtime. Examples: Python, Javascript, Ruby

Compiled languages in general run faster than interpreted languages.

Over to you: which type of language do you prefer?

Top 6 Load Balancing Algorithms



• Static Algorithms

1. Round robin

The client requests are sent to different service instances in sequential order. The services are usually required to be stateless.

2. Sticky round-robin

This is an improvement of the round-robin algorithm. If Alice's first request goes to service A, the following requests go to service A as well.

3. Weighted round-robin

The admin can specify the weight for each service. The ones with a higher weight handle more requests than others.

4. Hash

This algorithm applies a hash function on the incoming requests' IP or URL. The requests are routed to relevant instances based on the hash function result.

• Dynamic Algorithms

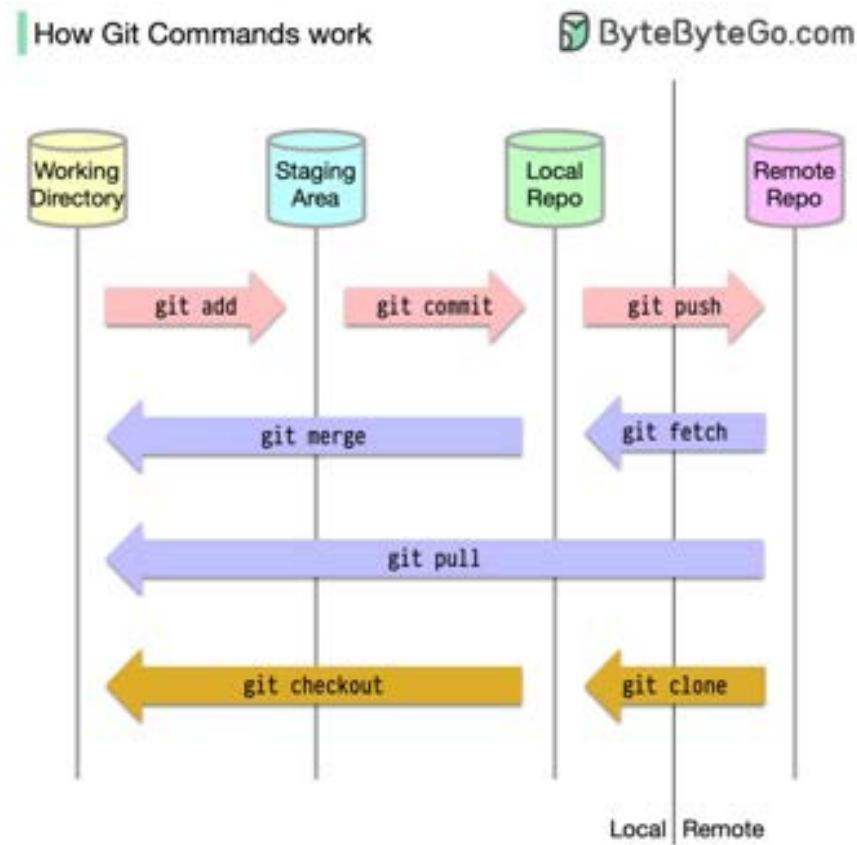
5. Least connections

A new request is sent to the service instance with the least concurrent connections.

6. Least response time

A new request is sent to the service instance with the fastest response time.

How does Git work?



To begin with, it's essential to identify where our code is stored. The common assumption is that there are only two locations - one on a remote server like Github and the other on our local machine. However, this isn't entirely accurate. Git maintains three local storages on our machine, which means that our code can be found in four places:

- Working directory: where we edit files
- Staging area: a temporary location where files are kept for the next commit
- Local repository: contains the code that has been committed
- Remote repository: the remote server that stores the code

Most Git commands primarily move files between these four locations.

Over to you: Do you know which storage location the "git tag" command operates on? This command can add annotations to a commit.

HTTP Cookies Explained With a Simple Diagram

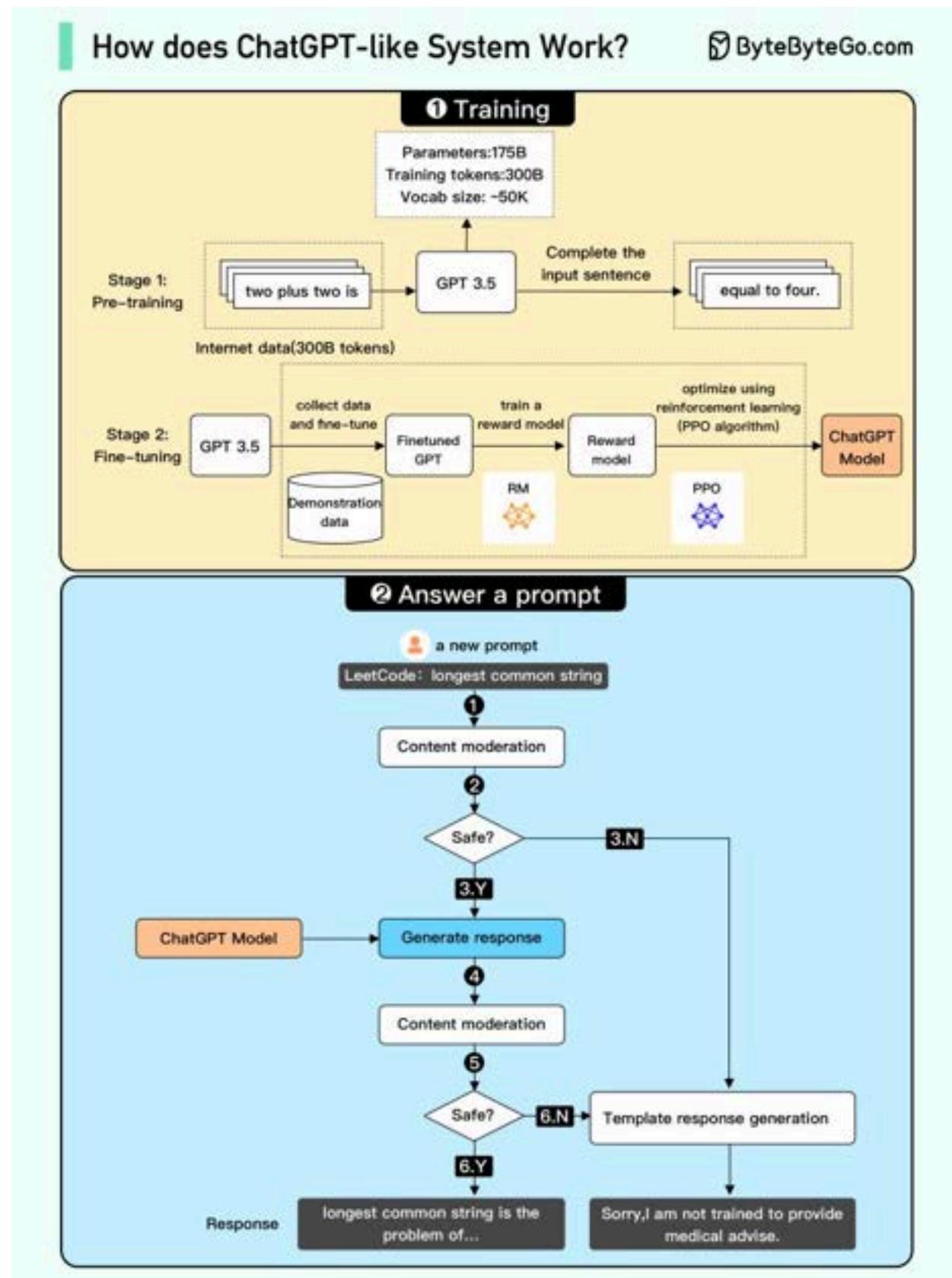


HTTP, the language of the web, is naturally "stateless." But hey, we all want that seamless, continuous browsing experience, right? Enter the unsung heroes - Cookies!

So, here's the scoop in this cookie flyer:

1. HTTP is like a goldfish with no memory - it forgets you instantly! But cookies swoop in to the rescue, adding that "session secret sauce" to your web interactions.
2. Cookies? Think of them as little notes you pass to the web server, saying, "Remember me, please!" And yes, they're stored there, like cherished mementos.
3. Browsers are like cookie bouncers, making sure your cookies don't party crash at the wrong website.
4. Finally, meet the cookie celebrities - SameSite, Name, Value, Secure, Domain, and HttpOnly. They're the cool kids setting the rules in the cookie jar!

How does a ChatGPT-like system work?



We attempted to explain how it works in the diagram below. The process can be broken down into two parts.

1. Training. To train a ChatGPT model, there are two stages:

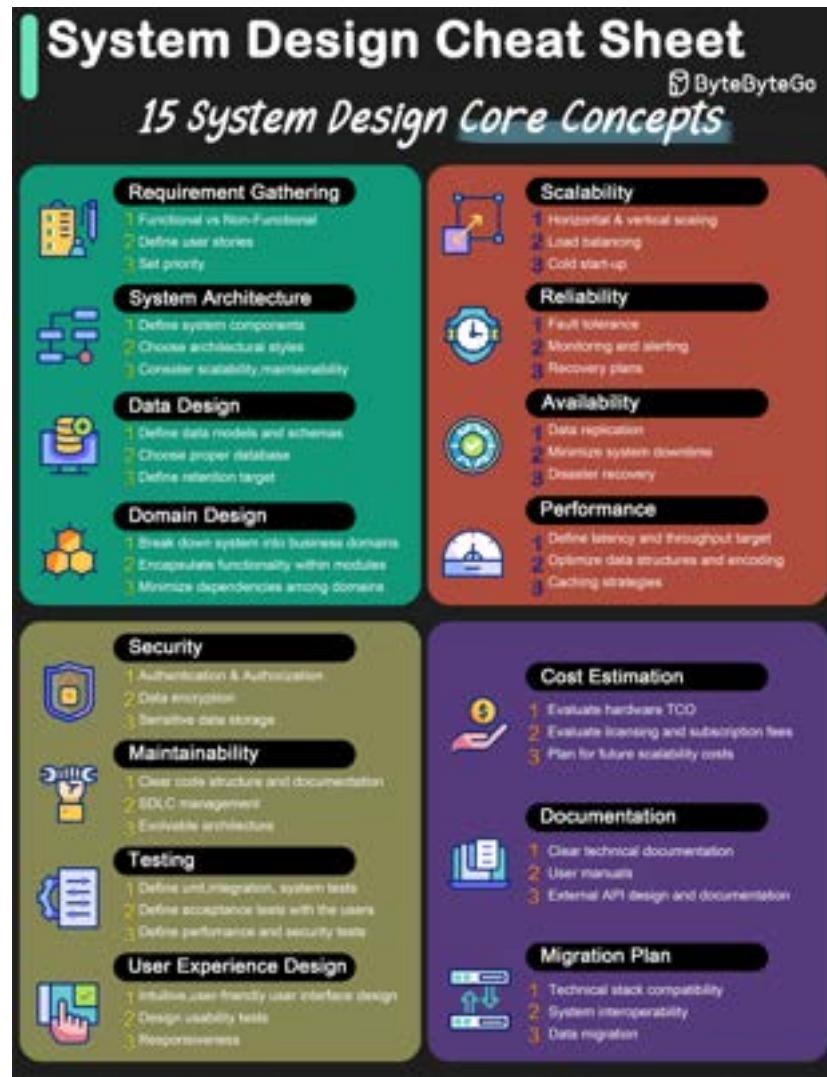
- Pre-training: In this stage, we train a GPT model (decoder-only transformer) on a large chunk of internet data. The objective is to train a model that can predict future words given a sentence in a way that is grammatically correct and semantically meaningful similar to the internet data. After the pre-training stage, the model can complete given sentences, but it is not capable of responding to questions.
- Fine-tuning: This stage is a 3-step process that turns the pre-trained model into a question-answering ChatGPT model:
 1. Collect training data (questions and answers), and fine-tune the pre-trained model on this data. The model takes a question as input and learns to generate an answer similar to the training data.
 2. Collect more data (question, several answers) and train a reward model to rank these answers from most relevant to least relevant.
 3. Use reinforcement learning (PPO optimization) to fine-tune the model so the model's answers are more accurate.

2. Answer a prompt

- Step 1: The user enters the full question, “Explain how a classification algorithm works”.
- Step 2: The question is sent to a content moderation component. This component ensures that the question does not violate safety guidelines and filters inappropriate questions.
- Steps 3-4: If the input passes content moderation, it is sent to the chatGPT model. If the input doesn't pass content moderation, it goes straight to template response generation.
- Step 5-6: Once the model generates the response, it is sent to a content moderation component again. This ensures the generated response is safe, harmless, unbiased, etc.
- Step 7: If the input passes content moderation, it is shown to the user. If the input doesn't pass content moderation, it goes to template response generation and shows a template answer to the user.

A cheat sheet for system designs

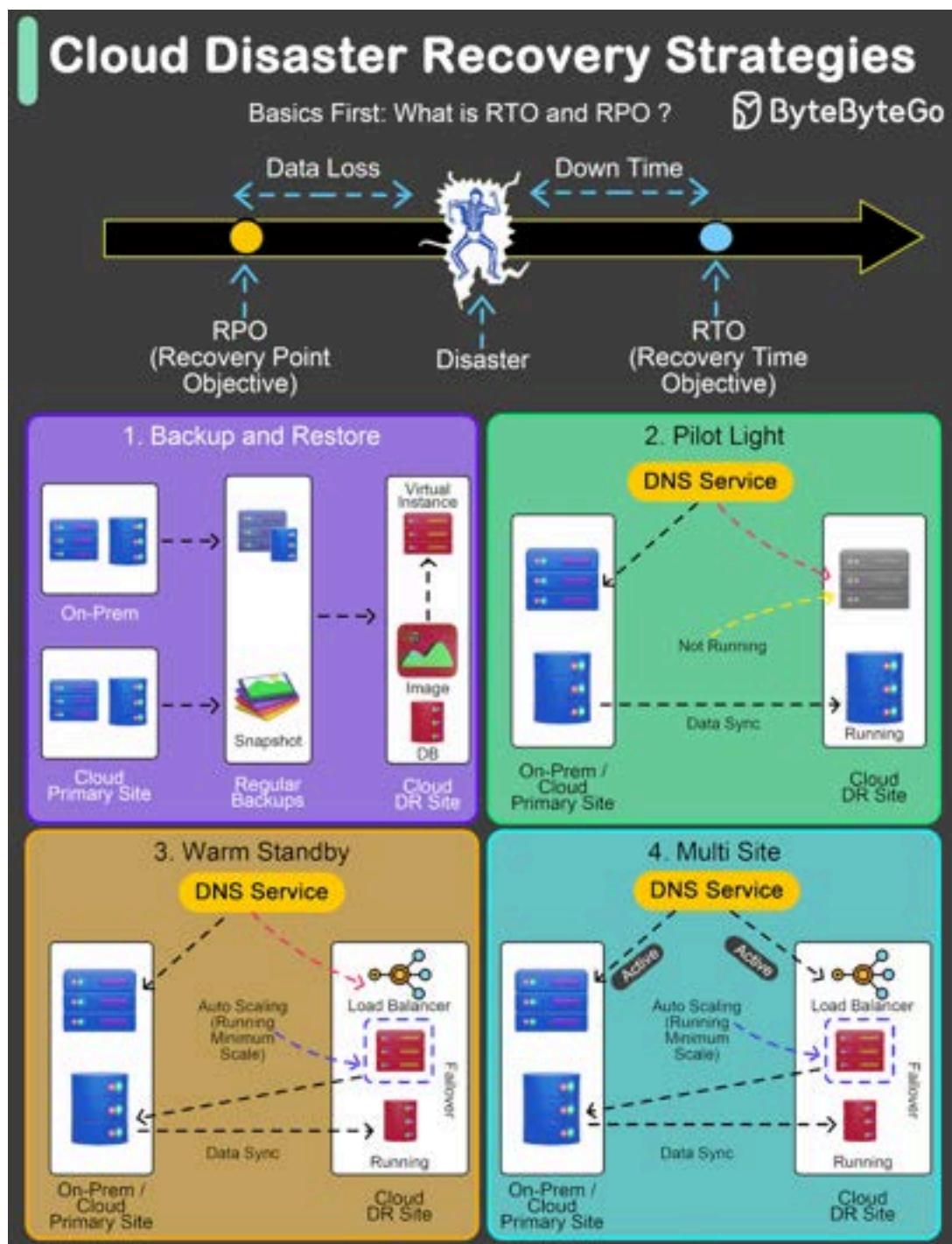
The diagram below lists 15 core concepts when we design systems. The cheat sheet is straightforward to go through one by one. Save it for future reference!



- Requirement gathering
- System architecture
- Data design
- Domain design
- Scalability
- Reliability
- Availability
- Performance
- Security
- Maintainability
- Testing

- User experience design
- Cost estimation
- Documentation
- Migration plan

Cloud Disaster Recovery Strategies



An effective Disaster Recovery (DR) plan is not just a precaution; it's a necessity.

The key to any robust DR strategy lies in understanding and setting two pivotal benchmarks: Recovery Time Objective (RTO) and Recovery Point Objective (RPO).

- Recovery Time Objective (RTO) refers to the maximum acceptable length of time that your application or network can be offline after a disaster.
- Recovery Point Objective (RPO), on the other hand, indicates the maximum acceptable amount of data loss measured in time.

Let's explore four widely adopted DR strategies:

1. Backup and Restore Strategy:

This method involves regular backups of data and systems to facilitate post-disaster recovery.

- Typical RTO: From several hours to a few days.
- Typical RPO: From a few hours up to the time of the last successful backup.

2. Pilot Light Approach:

Maintains crucial components in a ready-to-activate mode, enabling rapid scaling in response to a disaster.

- Typical RTO: From a few minutes to several hours.
- Typical RPO: Depends on how often data is synchronized.

3. Warm Standby Solution:

Establishes a semi-active environment with current data to reduce recovery time.

- Typical RTO: Generally within a few minutes to hours.
- Typical RPO: Up to the last few minutes or hours.

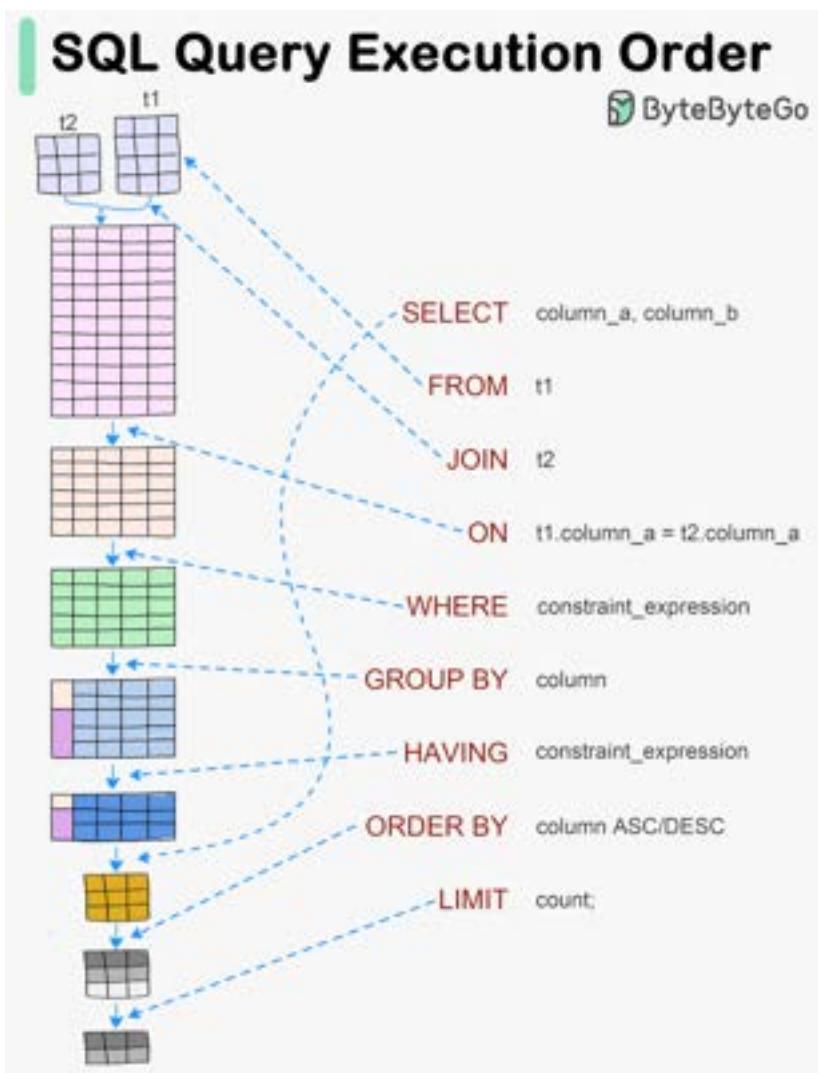
4. Hot Site / Multi-Site Configuration:

Ensures a fully operational, duplicate environment that runs parallel to the primary system.

- Typical RTO: Almost immediate, often just a few minutes.
- Typical RPO: Extremely minimal, usually only a few seconds old.

Over to you: What factors would influence your decision to choose a DR strategy?

Visualizing a SQL query



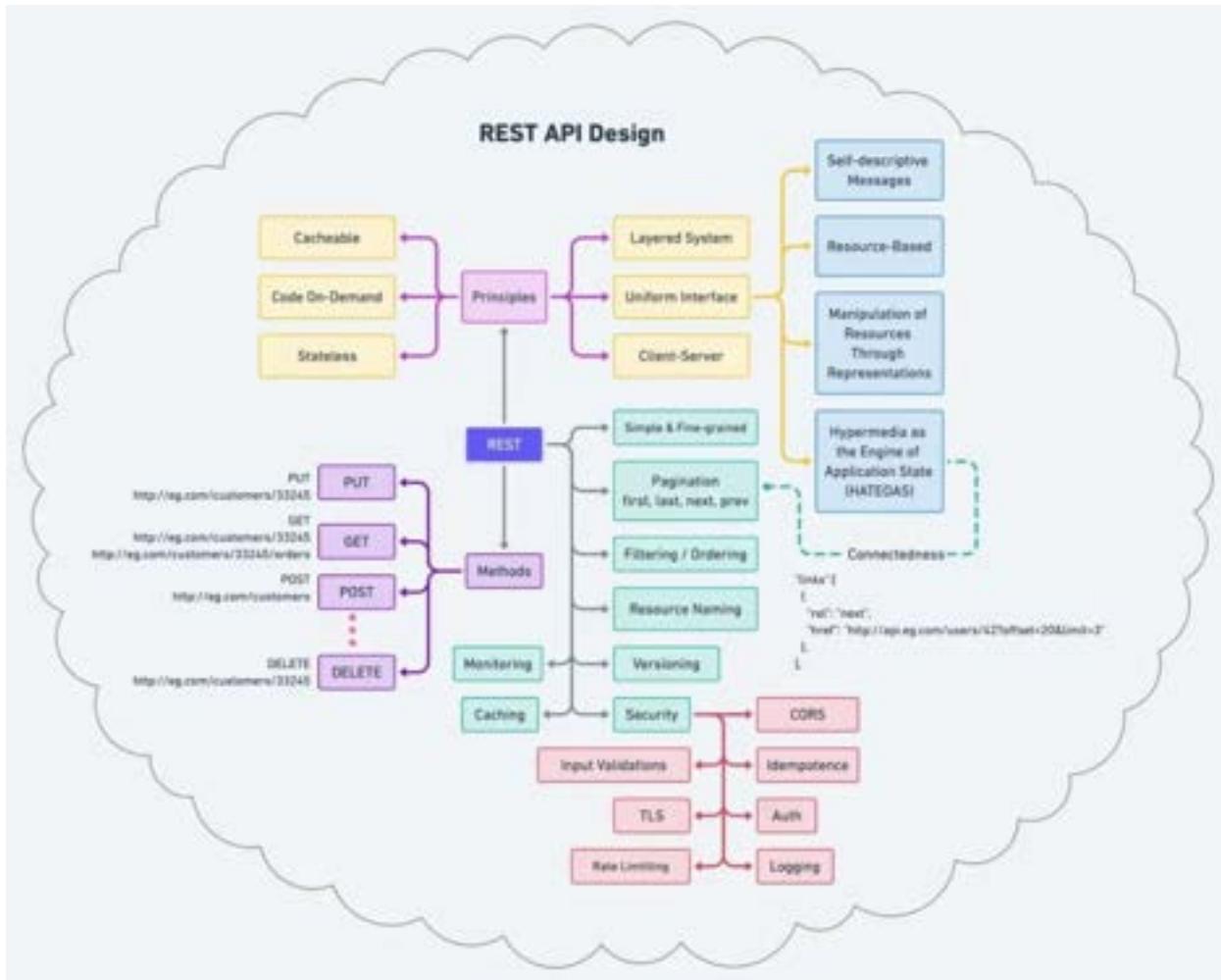
SQL statements are executed by the database system in several steps, including:

- Parsing the SQL statement and checking its validity
- Transforming the SQL into an internal representation, such as relational algebra
- Optimizing the internal representation and creating an execution plan that utilizes index information
- Executing the plan and returning the results

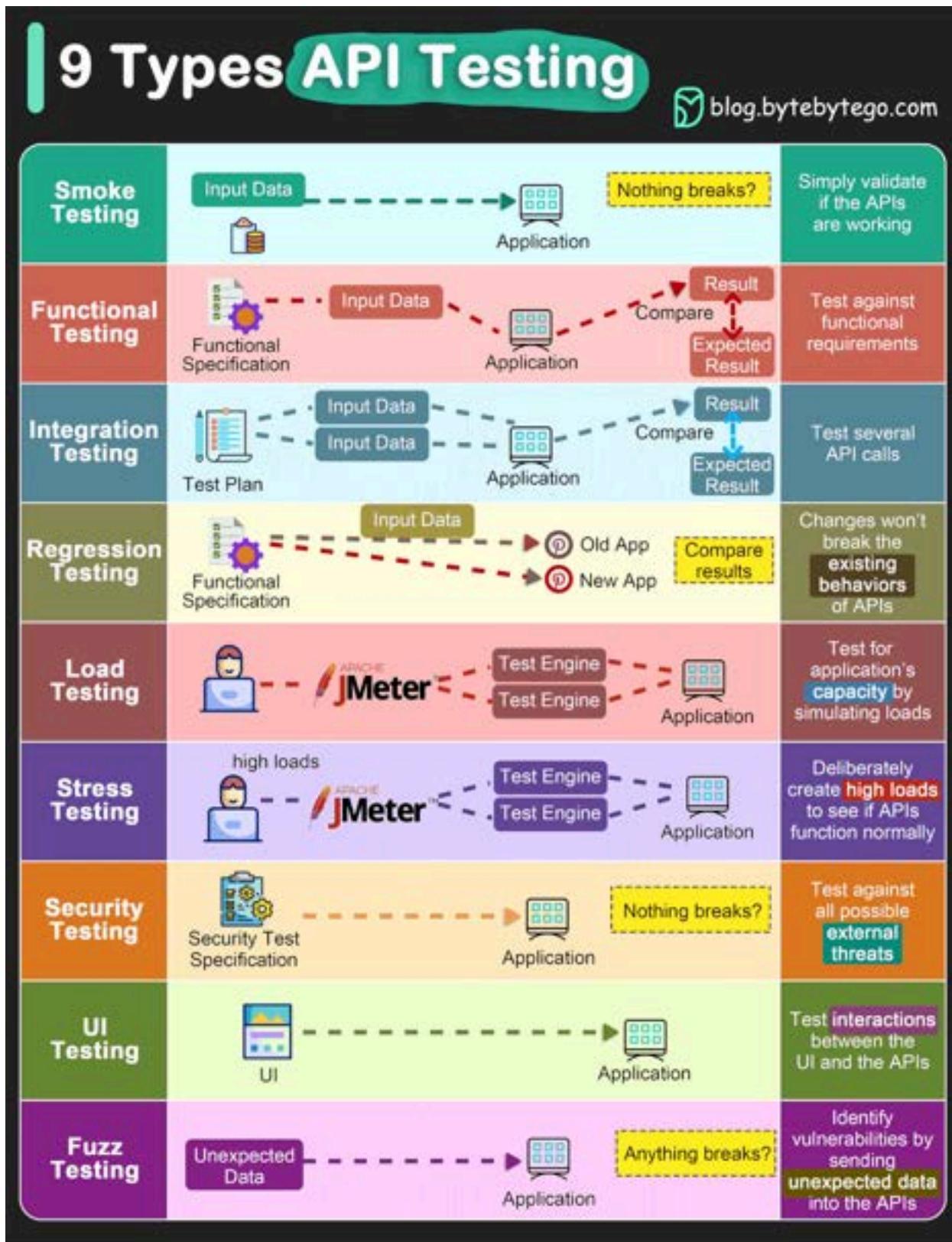
How does REST API work?

What are its principles, methods, constraints, and best practices?

I hope the diagram below gives you a quick overview.



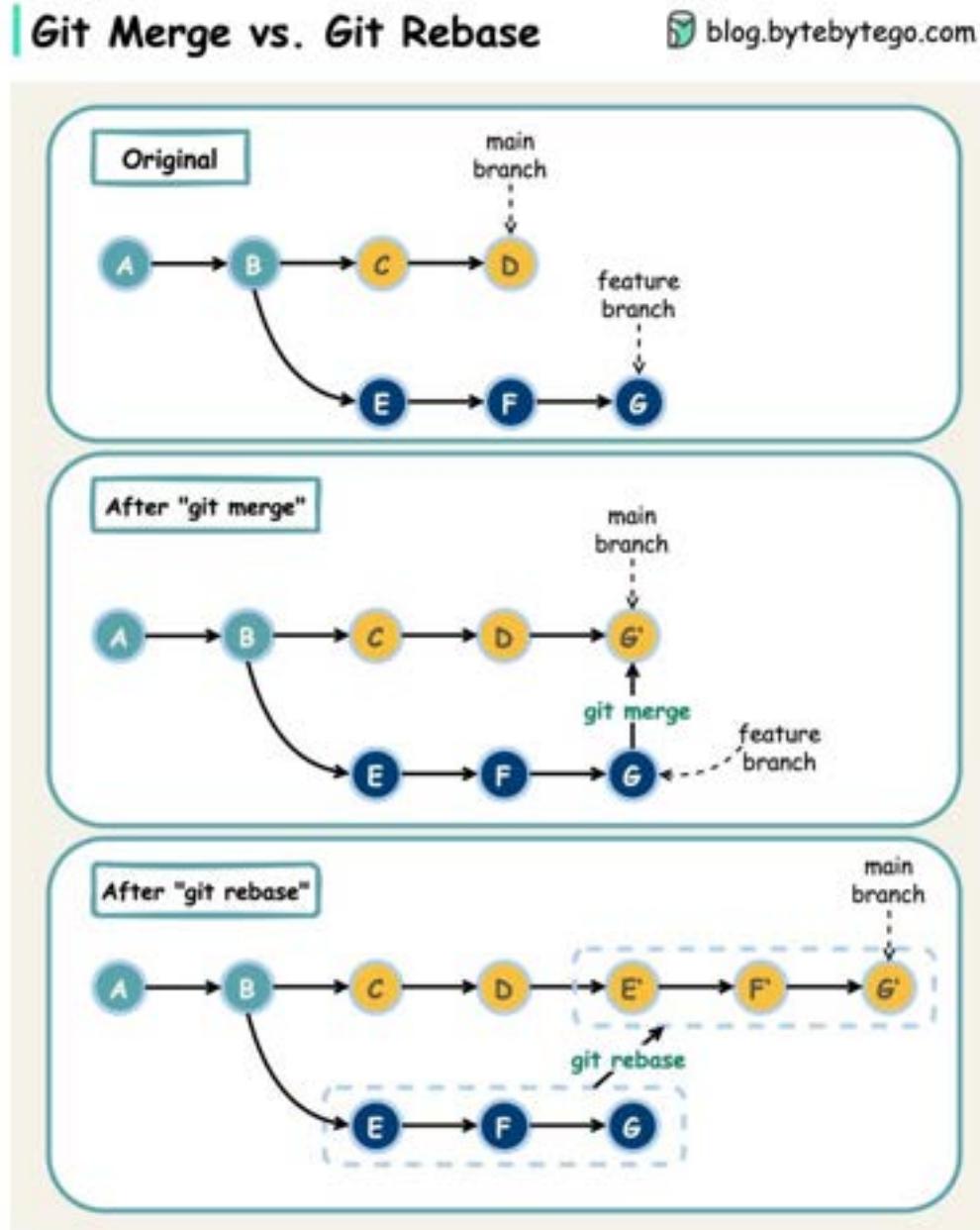
Explaining 9 types of API testing



- **Smoke Testing**
This is done after API development is complete. Simply validate if the APIs are working and nothing breaks.
- **Functional Testing**
This creates a test plan based on the functional requirements and compares the results with the expected results.
- **Integration Testing**
This test combines several API calls to perform end-to-end tests. The intra-service communications and data transmissions are tested.
- **Regression Testing**
This test ensures that bug fixes or new features shouldn't break the existing behaviors of APIs.
- **Load Testing**
This tests applications' performance by simulating different loads. Then we can calculate the capacity of the application.
- **Stress Testing**
We deliberately create high loads to the APIs and test if the APIs are able to function normally.
- **Security Testing**
This tests the APIs against all possible external threats.
- **UI Testing**
This tests the UI interactions with the APIs to make sure the data can be displayed properly.
- **Fuzz Testing**
This injects invalid or unexpected input data into the API and tries to crash the API. In this way, it identifies the API vulnerabilities.

Git Merge vs. Rebase vs. Squash Commit!

What are the differences?



When we **merge changes** from one Git branch to another, we can use 'git merge' or 'git rebase'. The diagram below shows how the two commands work.

Git Merge

This creates a new commit **G'** in the main branch. **G'** ties the histories of both main and feature branches.

Git merge is **non-destructive**. Neither the main nor the feature branch is changed.

Git Rebase

Git rebase moves the feature branch histories to the head of the main branch. It creates new commits E', F', and G' for each commit in the feature branch.

The benefit of rebase is that it has **linear commit history**.

Rebase can be dangerous if “the golden rule of git rebase” is not followed.

The Golden Rule of Git Rebase

Never use it on public branches!

What is a cookie?

What is a Cookie ?

EXPLAIN TECH WITH COMICS
ByteByteGo

Bob enters the cafe for the first time.

MEDIUM-SIZED COFFEE WITH TWO SUGARS

Bob Cashier

HERE IS YOUR COFFEE AND YOUR CARD. BRING THE CARD WITH YOU NEXT TIME

Bob Medium-sized, two sugars Cashier

Bob enters the cafe again

HERE IS MY CARD

Bob Cashier

WELCOME BOB! HERE IS YOUR COFFEE.

Bob Cashier

A COOKIE IS LIKE A CARD, CARRYING USER INFORMATION BETWEEN CLIENT AND SERVER

Client Server

1. login
2. cookie
3. request cookie
4. request

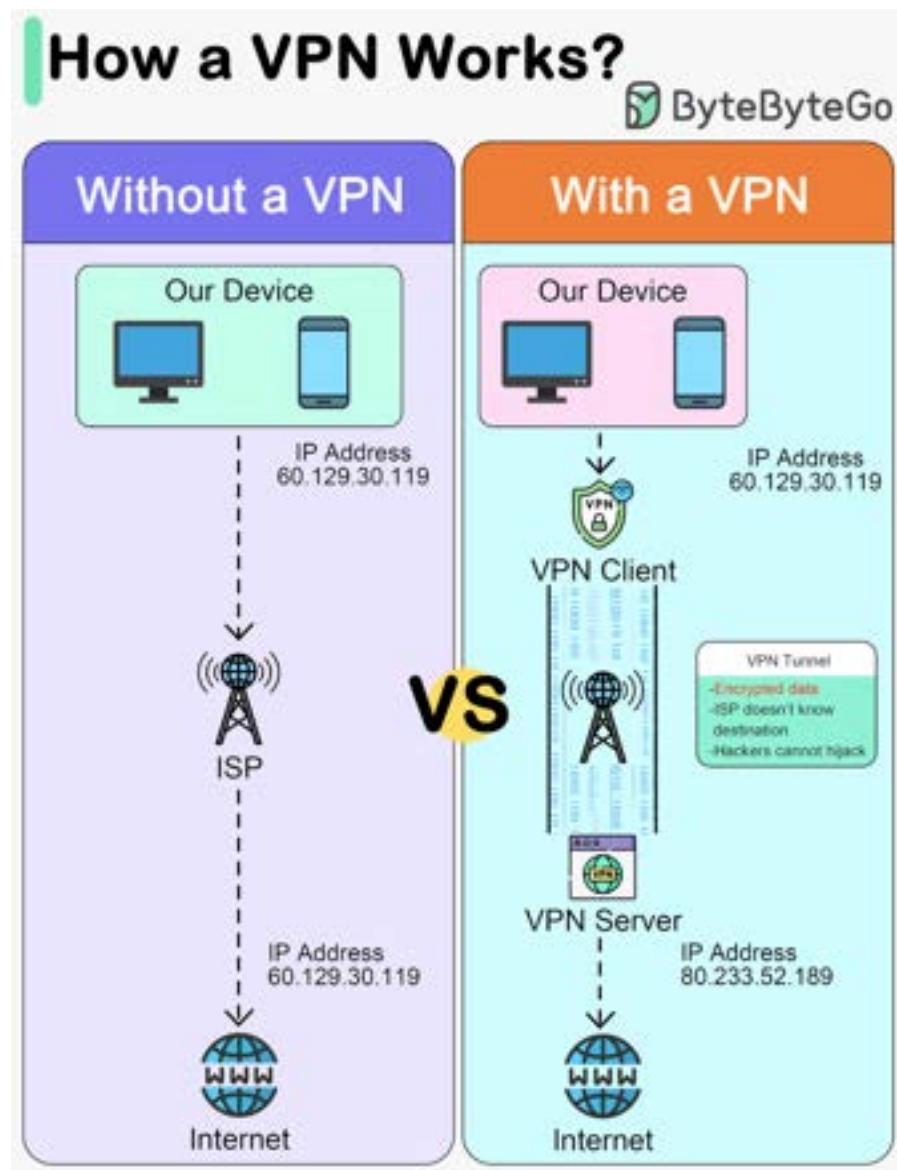
Imagine Bob goes to a coffee shop for the first time, orders a medium-sized espresso with two sugars. The cashier records Bob's identity and preferences on a card and hands it over to Bob with a cup of coffee.

The next time Bob goes to the cafe, he shows the cashier the preference card. The cashier immediately knows who the customer is and what kind of coffee he likes.

A cookie acts as the preference card. When we log in to a website, the server issues a cookie to us with a small amount of data. The cookie is stored on the client side, so the next time we send a request to the server with the cookie, the server knows our identity and preferences immediately without looking into the database.

How does a VPN work?

This diagram below shows how we access the internet with and without VPNs.



A VPN, or Virtual Private Network, is a technology that creates a secure, encrypted connection over a less secure network, such as the public internet. The primary purpose of a VPN is to provide privacy and security to data and communications.

A VPN acts as a tunnel through which the encrypted data goes from one location to another. Any external party cannot see the data transferring.

A VPN works in 4 steps:

- Step 1 - Establish a secure tunnel between our device and the VPN server.

- Step 2 - Encrypt the data transmitted.
- Step 3 - Mask our IP address, so it appears as if our internet activity is coming from the VPN server.
- Step 4 - Our internet traffic is routed through the VPN server.

Advantages of a VPN:

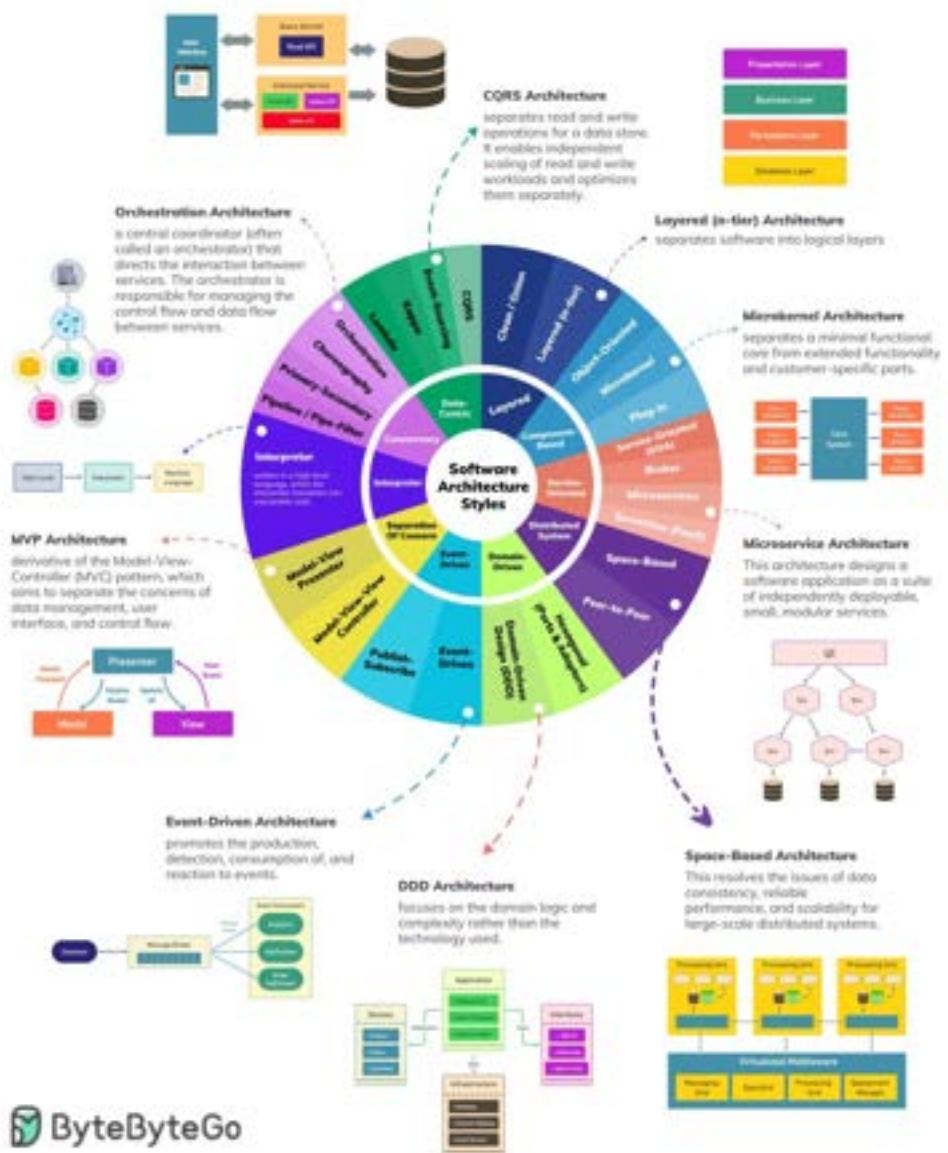
- Privacy
- Anonymity
- Security
- Encryption
- Masking the original IP address

Disadvantages of a VPN:

- VPN blocking
- Slow down connections
- Trust in VPN provider

Top Software Architectural Styles

Software Architecture Styles

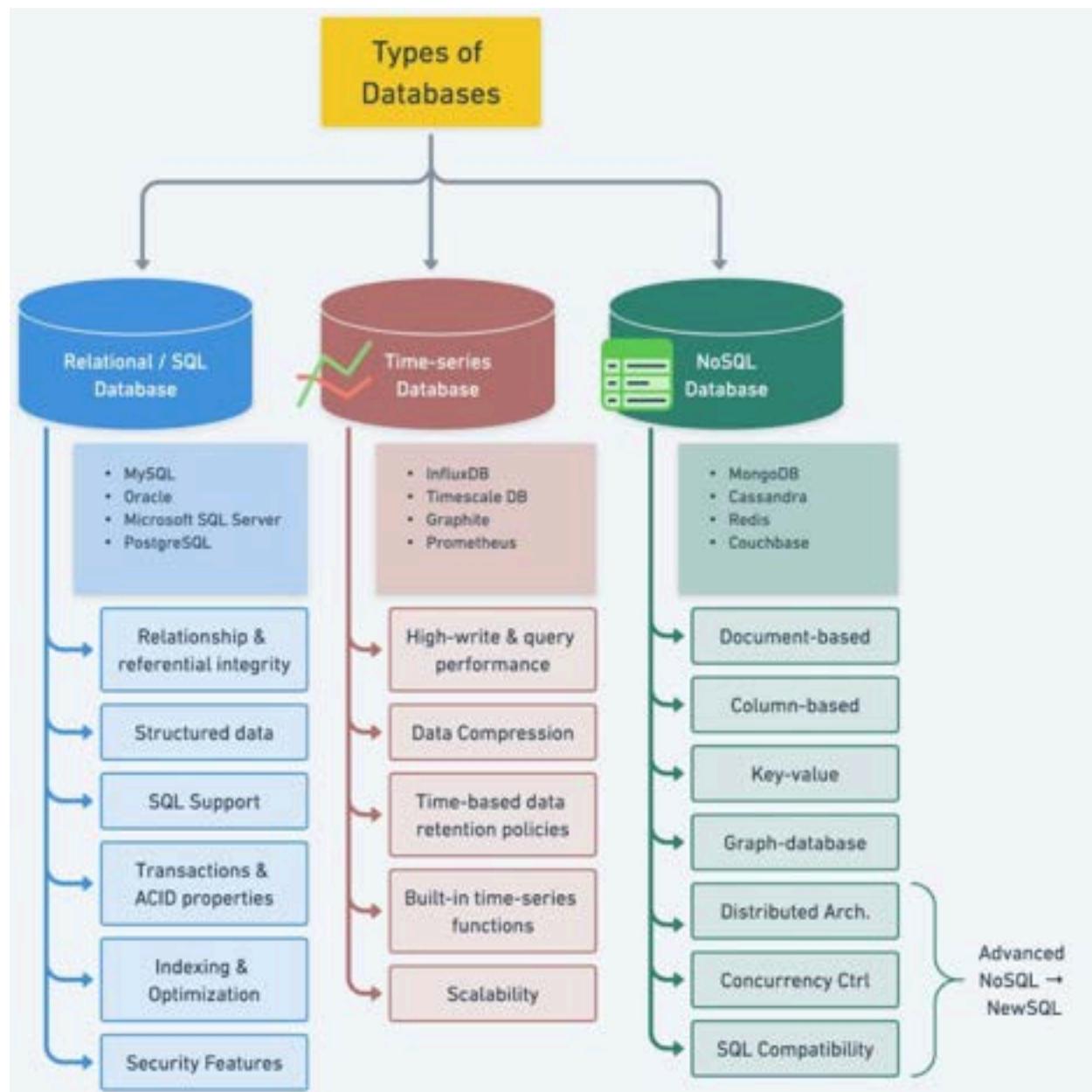


In software development, architecture plays a crucial role in shaping the structure and behavior of software systems. It provides a blueprint for system design, detailing how components interact with each other to deliver specific functionality. They also offer solutions to common problems, saving time and effort and leading to more robust and maintainable systems.

However, with the vast array of architectural styles and patterns available, it can take time to discern which approach best suits a particular project or system. Aims to shed light on these concepts, helping you make informed decisions in your architectural endeavors.

To help you navigate the vast landscape of architectural styles and patterns, there is a cheat sheet that encapsulates all. This cheat sheet is a handy reference guide that you can use to quickly recall the main characteristics of each architectural style and pattern.

Understanding Database Types



To make the best decision for our projects, it is essential to understand the various types of databases available in the market. We need to consider key characteristics of different database types, including popular options for each, and compare their use cases.

Cloud Security Cheat Sheet

Element	aws	Google Cloud	Azure
Infrastructure Security	 Shield  Security Hub  WAF  Certificate Manager	 Cloud Armor  Security Command Center  Cloud Armor  Certificate Manager	 DDoS Protection  Security Center  WAF  Key Vault
Identity Security	 Cloud Trail  IAM  Directory Service  Firewall Manager  Resource Access Manager	 Cloud Audit Logs  IAM  Managed Service Active Directory  Firewall Rules  Resource Manager	 Azure Audit Logs  Active Directory  Active Directory Domain Services  Firewall Manager  Resource Manager
Data Security	 Macie  CloudHSM  KMS  Secrets Manager  Config	 Data Loss Prevention  CloudHSM  KMS  Secret Manager  Security Command Center	 Information Protection  Managed HSM  Key Vault  Key Vault  Microsoft Defender
Business Security	 Fraud Detector  Rekognition  Cognito	 reCAPTCHA Enterprise  Vision AI  Identity Platform	 Microsoft Dynamics Fraud  Computer Vision  Active Directory

Cloud security is the top priority for any business because it ensures the safety and privacy of their digital assets in the cloud.

Having said that, it is not that simple, especially with so many services, applications, and potential threats to consider.

The complexity of modern cloud environments requires diligent planning, robust security measures, and continuous monitoring to protect against data breaches, cyberattacks, and compliance violations.

Businesses must proactively invest in cloud security practices, stay informed about evolving threats, and adapt their strategies to mitigate risks effectively and maintain trust with their customers and partners.

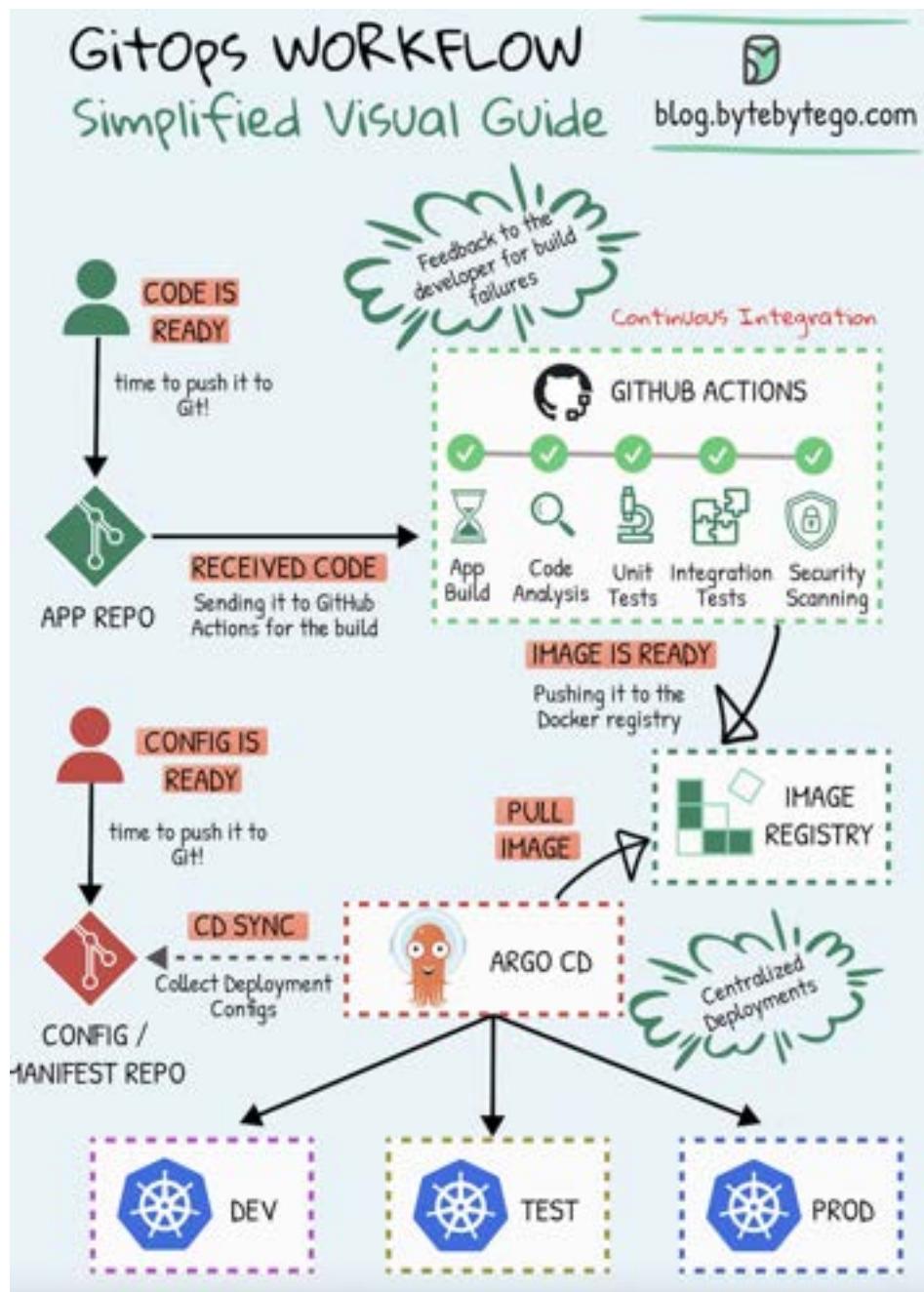
Especially with multi-cloud implementations, the complexity grows. Keeping a watchful eye on the services and resources scattered across multiple cloud providers can be challenging.

It demands a comprehensive understanding of each cloud platform's unique security features, configurations, and best practices.

Happy to introduce the cloud security cheat sheet that maps the cloud services across three popular cloud providers and helps you to quickly navigate the complexities of cloud security.

Over to you: How do you track the security offerings available in the cloud?

GitOps Workflow - Simplified Visual Guide



GitOps brought a shift in how software and infrastructure are managed with Git as the central hub for managing and automating the entire lifecycle of applications and infrastructure.

It's built on the principles of version control, collaboration, and continuous integration and deployment (CI/CD).

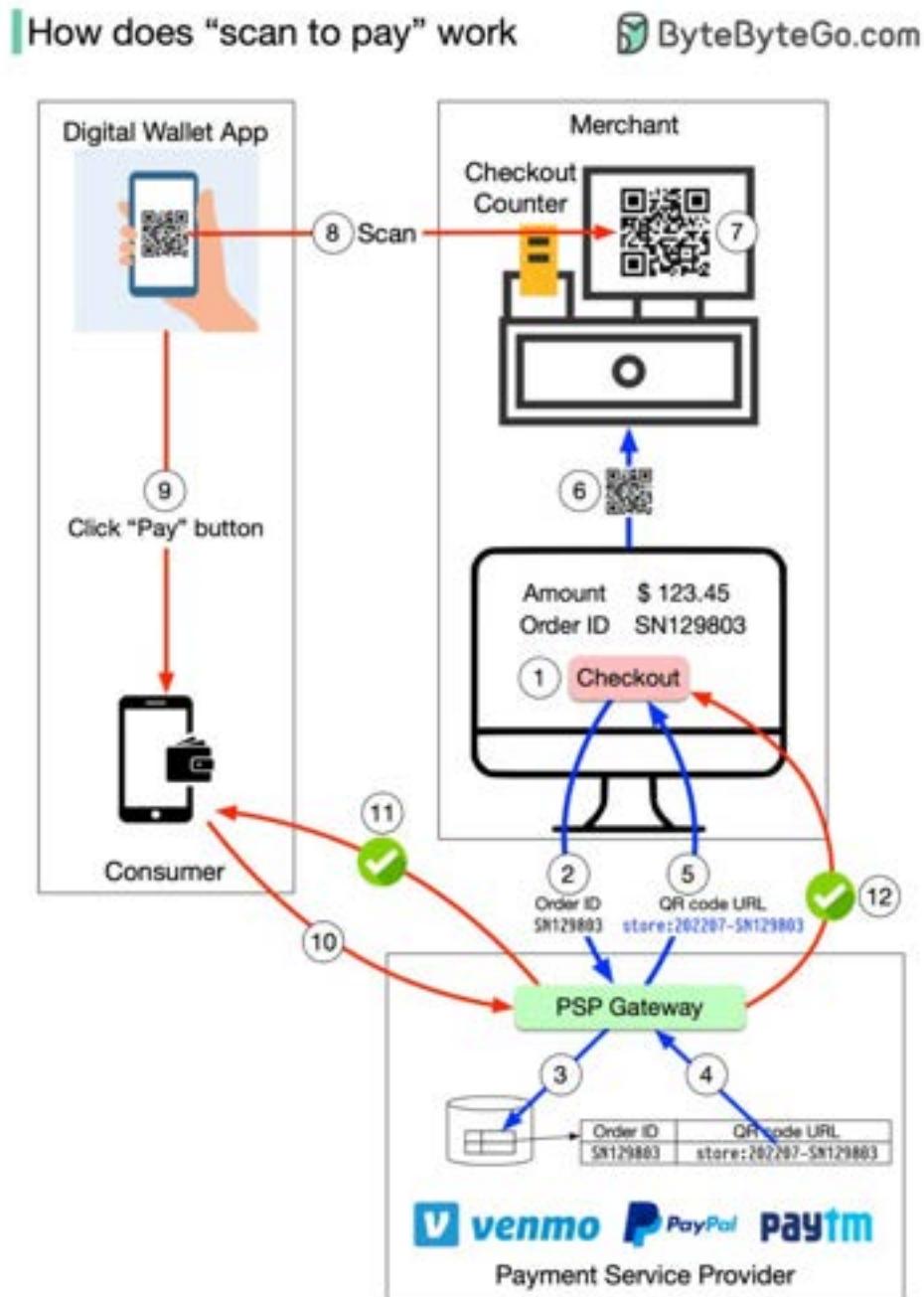
Key features include:

1. Version Control and Collaboration:
Centralizing code, configurations, and infrastructure in Git for control and collaboration.
2. Declarative System:
Describing the system's desired state for easier version control.
3. Automated Delivery:
Automating deployment through Git-triggered processes, closely integrated with CI/CD pipelines.
4. Immutable Infrastructure:
Making changes via Git instead of directly in the live environment to prevent inconsistencies.
5. Observability and Feedback:
Monitoring systems in real-time to align the actual state with Git's declared state.
6. Security and Compliance:
Tracking changes in Git for security and compliance, with role-based access for added control.

Over to you: Do you see GitOps' declarative approach speeding up your deployments?

How does “scan to pay” work?

How do you pay from your digital wallet, such as Paypal, Venmo, Paytm, by scanning the QR code?



To understand the process involved, we need to divide the “scan to pay” process into two sub-processes:

1. Merchant generates a QR code and displays it on the screen
2. Consumer scans the QR code and pays

Here are the steps for generating the QR code:

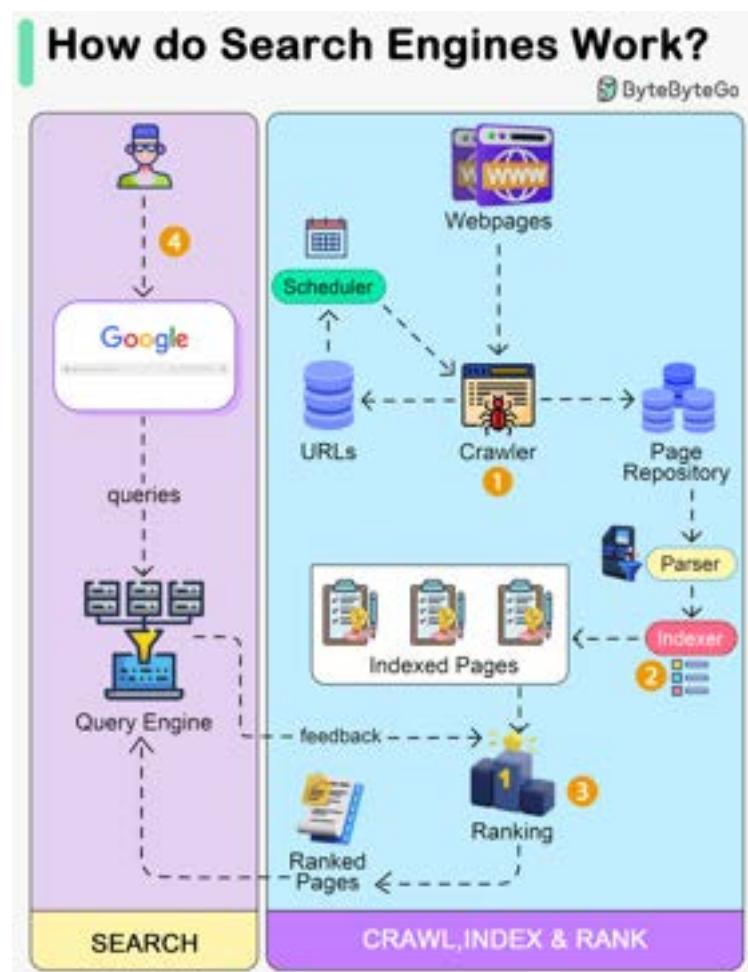
1. When you want to pay for your shopping, the cashier tallies up all the goods and calculates the total amount due, for example, \$123.45. The checkout has an order ID of SN129803. The cashier clicks the “checkout” button.
2. The cashier’s computer sends the order ID and the amount to PSP.
3. The PSP saves this information to the database and generates a QR code URL.
4. PSP’s Payment Gateway service reads the QR code URL.
5. The payment gateway returns the QR code URL to the merchant’s computer.
6. The merchant’s computer sends the QR code URL (or image) to the checkout counter.
7. The checkout counter displays the QR code.

These 7 steps complete in less than a second. Now it’s the consumer’s turn to pay from their digital wallet by scanning the QR code:

1. The consumer opens their digital wallet app to scan the QR code.
2. After confirming the amount is correct, the client clicks the “pay” button.
3. The digital wallet App notifies the PSP that the consumer has paid the given QR code.
4. The PSP payment gateway marks this QR code as paid and returns a success message to the consumer’s digital wallet App.
5. The PSP payment gateway notifies the merchant that the consumer has paid the given QR code.

How do Search Engines Work?

The diagram below shows a high-level walk-through of a search engine.

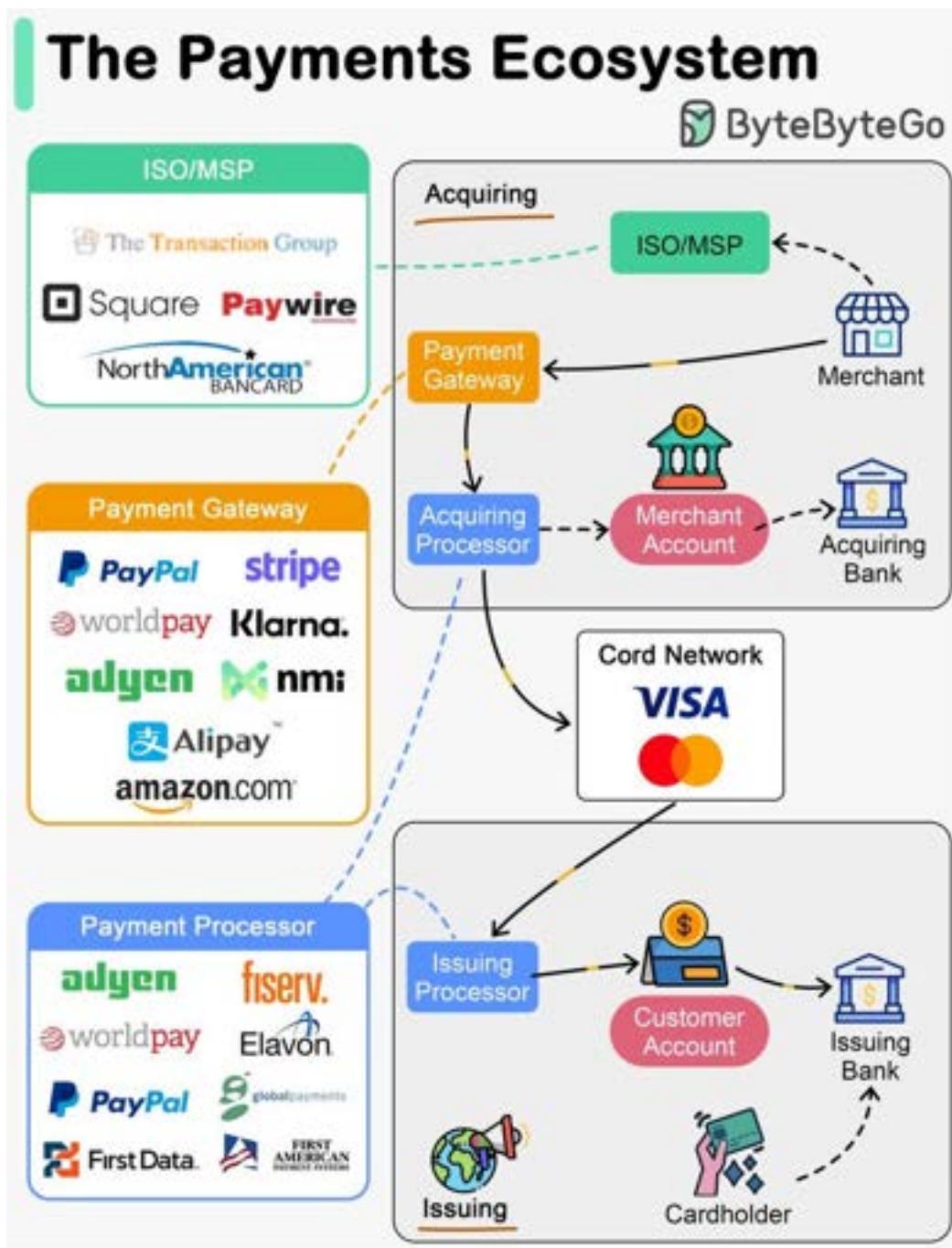


- Step 1 - Crawling
Web Crawlers scan the internet for web pages. They follow the URL links from one page to another and store URLs in the URL store. The crawlers discover new content, including web pages, images, videos, and files.
- Step 2 - Indexing
Once a web page is crawled, the search engine parses the page and indexes the content found on the page in a database. The content is analyzed and categorized. For example, keywords, site quality, content freshness, and many other factors are assessed to understand what the page is about.
- Step 3 - Ranking
Search engines use complex algorithms to determine the order of search results. These algorithms consider various factors, including keywords, pages' relevance, content quality, user engagement, page load speed, and many others. Some search engines also personalize results based on the user's past search history, location, device, and other personal factors.

- Step 4 - Querying

When a user performs a search, the search engine sifts through its index to provide the most relevant results.

The Payments Ecosystem



How do fintech startups find new opportunities among so many payment companies? What do PayPal, Stripe, and Square do exactly?

Steps 0-1: The cardholder opens an account in the issuing bank and gets the debit/credit card. The merchant registers with ISO (Independent Sales Organization) or MSP (Member Service Provider) for in-store sales. ISO/MSP partners with payment processors to open merchant accounts.

Steps 2-5: The acquiring process.

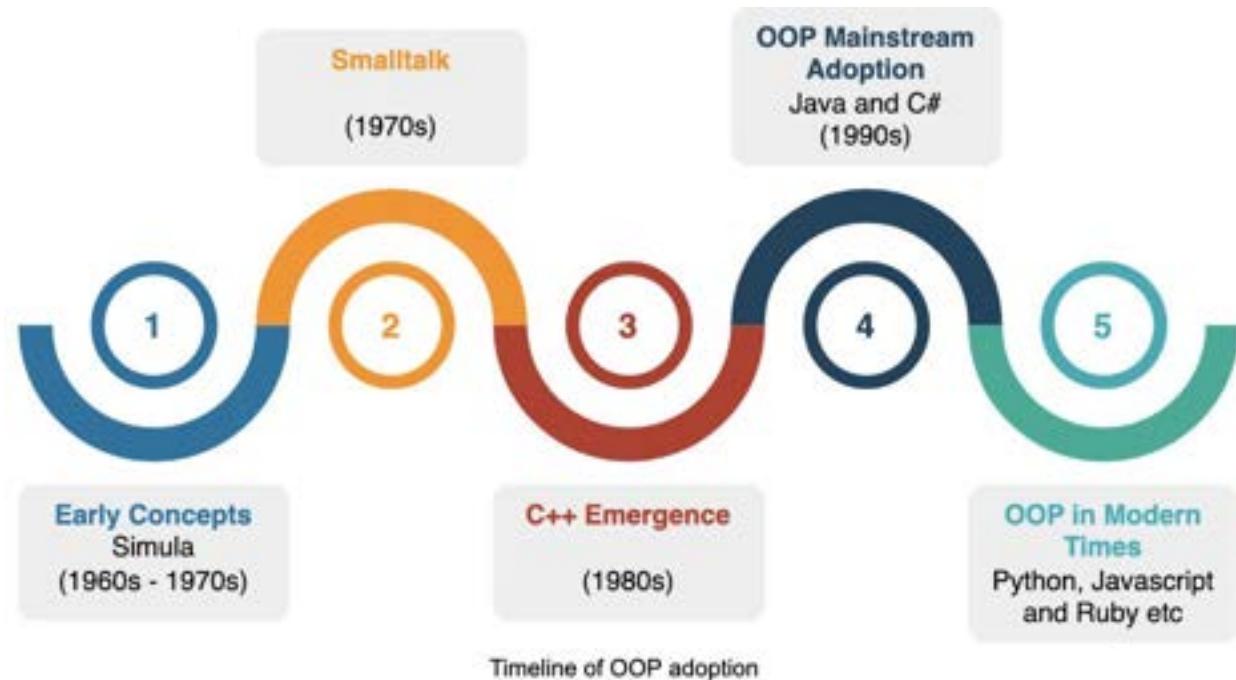
The payment gateway accepts the purchase transaction and collects payment information. It is then sent to a payment processor, which uses customer information to collect payments. The acquiring processor sends the transaction to the card network. It also owns and operates the merchant's account during settlement, which doesn't happen in real-time.

Steps 6-8: The issuing process.

The issuing processor talks to the card network on the issuing bank's behalf. It validates and operates the customer's account.

I've listed some companies in different verticals in the diagram. Notice payment companies usually start from one vertical, but later expand to multiple verticals.

Object-oriented Programming: A Primer

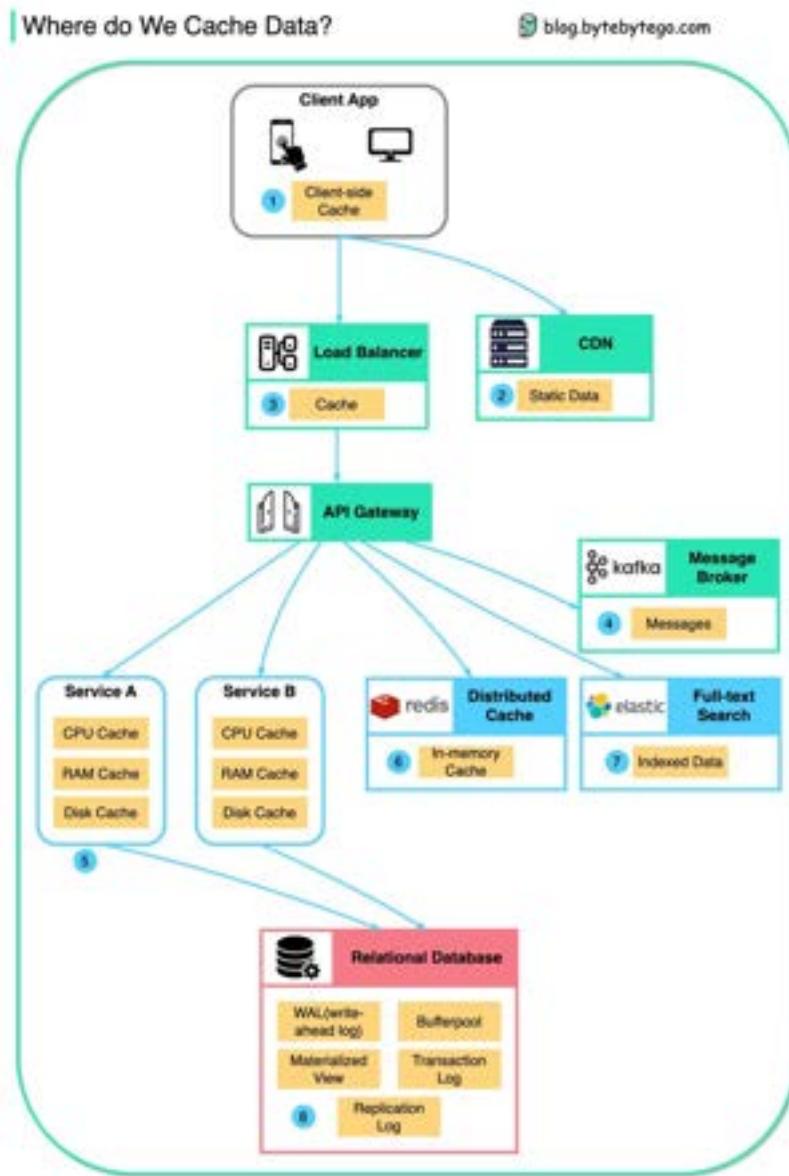


I wrote a [blog post](#) about this topic. It covers:

- Background of OOP
- Classes and Objects
- Cornerstones of Object-Oriented Programming
 - Encapsulation
 - Abstraction
 - Inheritance
 - Polymorphism

Where do we cache data?

Data is cached everywhere, from the front end to the back end!



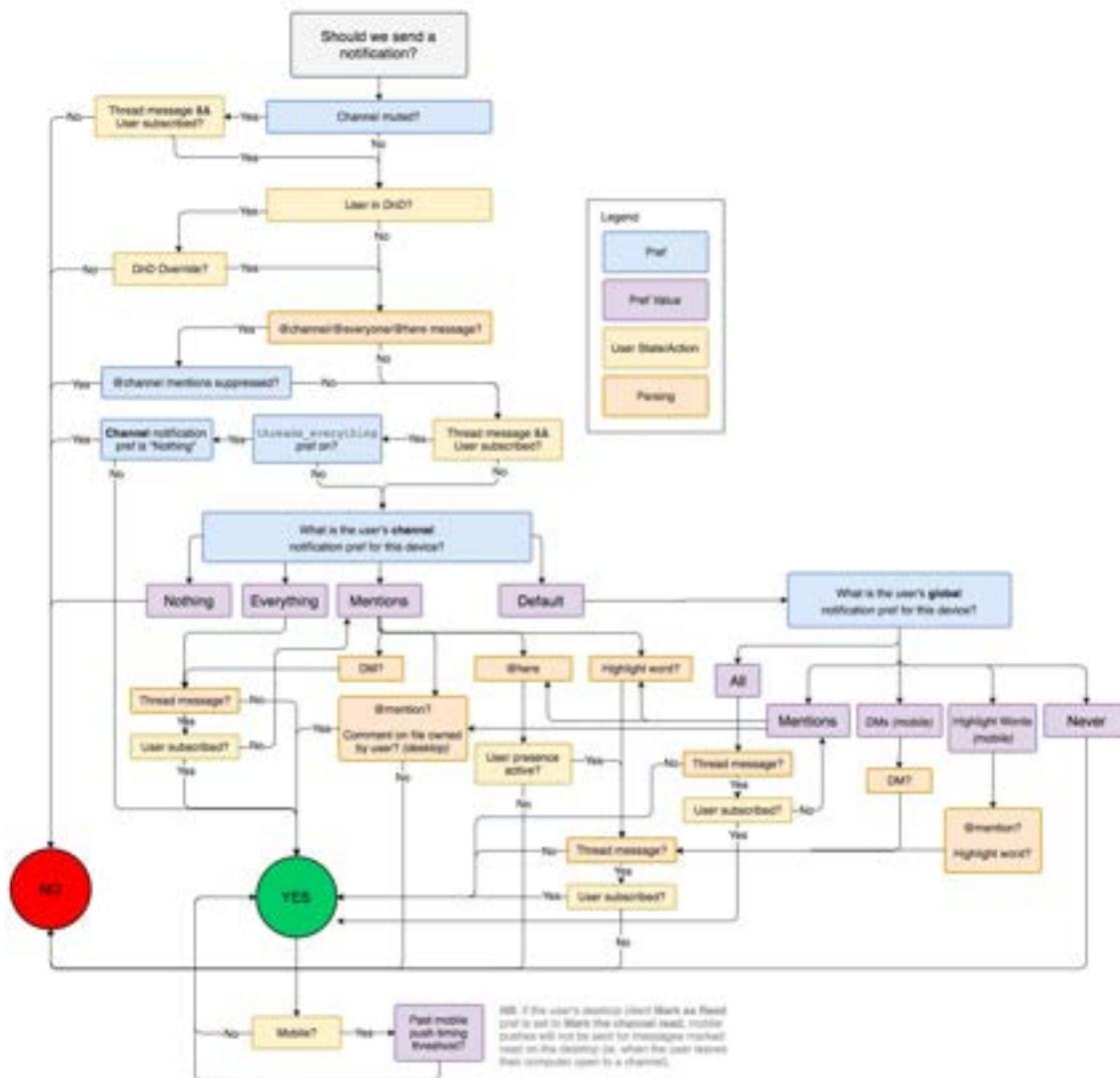
There are **multiple layers** along the flow.

1. Client apps: HTTP responses can be cached by the browser. We request data over HTTP for the first time, and it is returned with an expiry policy in the HTTP header; we request data again, and the client app tries to retrieve the data from the browser cache first.
2. CDN: CDN caches static web resources. The clients can retrieve data from a CDN node nearby.
3. Load Balancer: The load Balancer can cache resources as well.

4. Messaging infra: Message brokers store messages on disk first, and then consumers retrieve them at their own pace. Depending on the retention policy, the data is cached in Kafka clusters for a period of time.
5. Services: There are multiple layers of cache in a service. If the data is not cached in the CPU cache, the service will try to retrieve the data from memory. Sometimes the service has a second-level cache to store data on disk.
6. Distributed Cache: Distributed cache like Redis hold key-value pairs for multiple services in memory. It provides much better read/write performance than the database.
7. Full-text Search: we sometimes need to use full-text searches like Elastic Search for document search or log search. A copy of data is indexed in the search engine as well.
8. Database: Even in the database, we have different levels of caches:
 - WAL(Write-ahead Log): data is written to WAL first before building the B tree index
 - Bufferpool: A memory area allocated to cache query results
 - Materialized View: Pre-compute query results and store them in the database tables for better query performance
 - Transaction log: record all the transactions and database updates
 - Replication Log: used to record the replication state in a database cluster

Over to you: With the data cached at so many levels, how can we guarantee the **sensitive user data** is completely erased from the systems?

Flowchart of how slack decides to send a notification

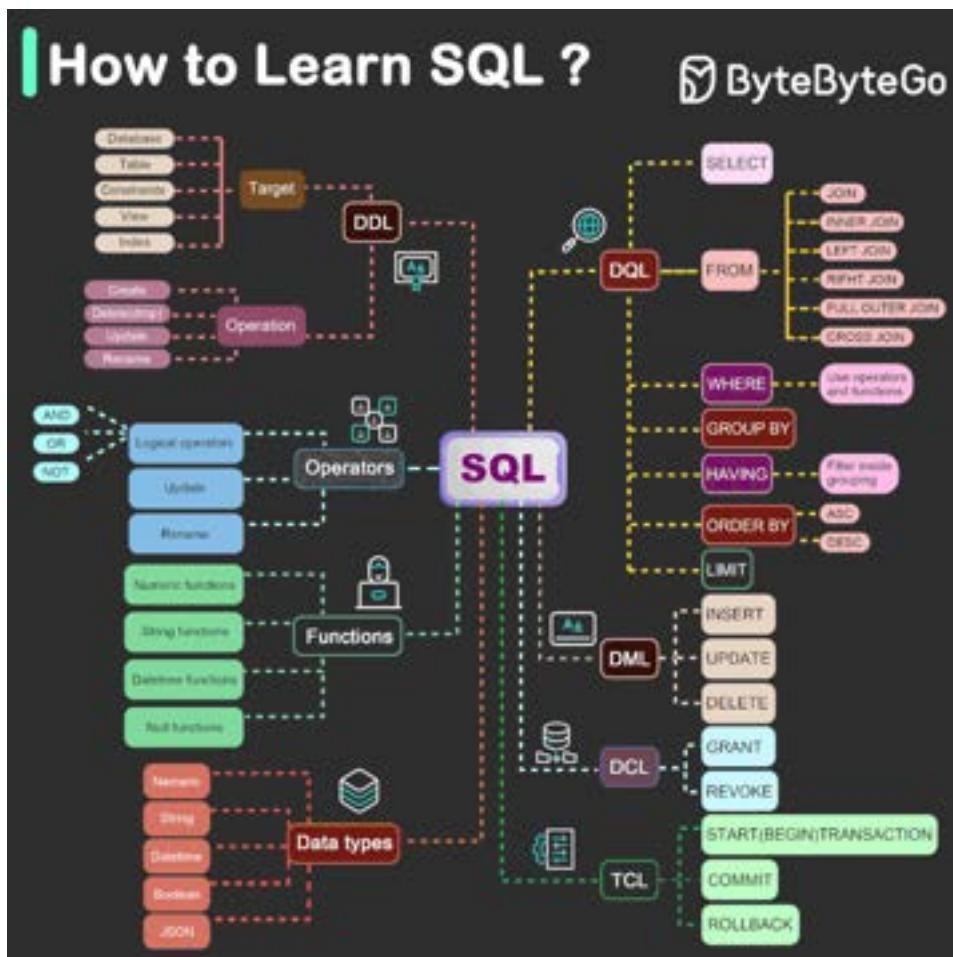


It is a great example of why a simple feature may take much longer to develop than many people think.

When we have a great design, users may not notice the complexity because it feels like the feature just working as intended.

What's your takeaway from this diagram?

What is the best way to learn SQL?



In 1986, SQL (Structured Query Language) became a standard. Over the next 40 years, it became the dominant language for relational database management systems. Reading the latest standard (ANSI SQL 2016) can be time-consuming. How can I learn it?

There are 5 components of the SQL language:

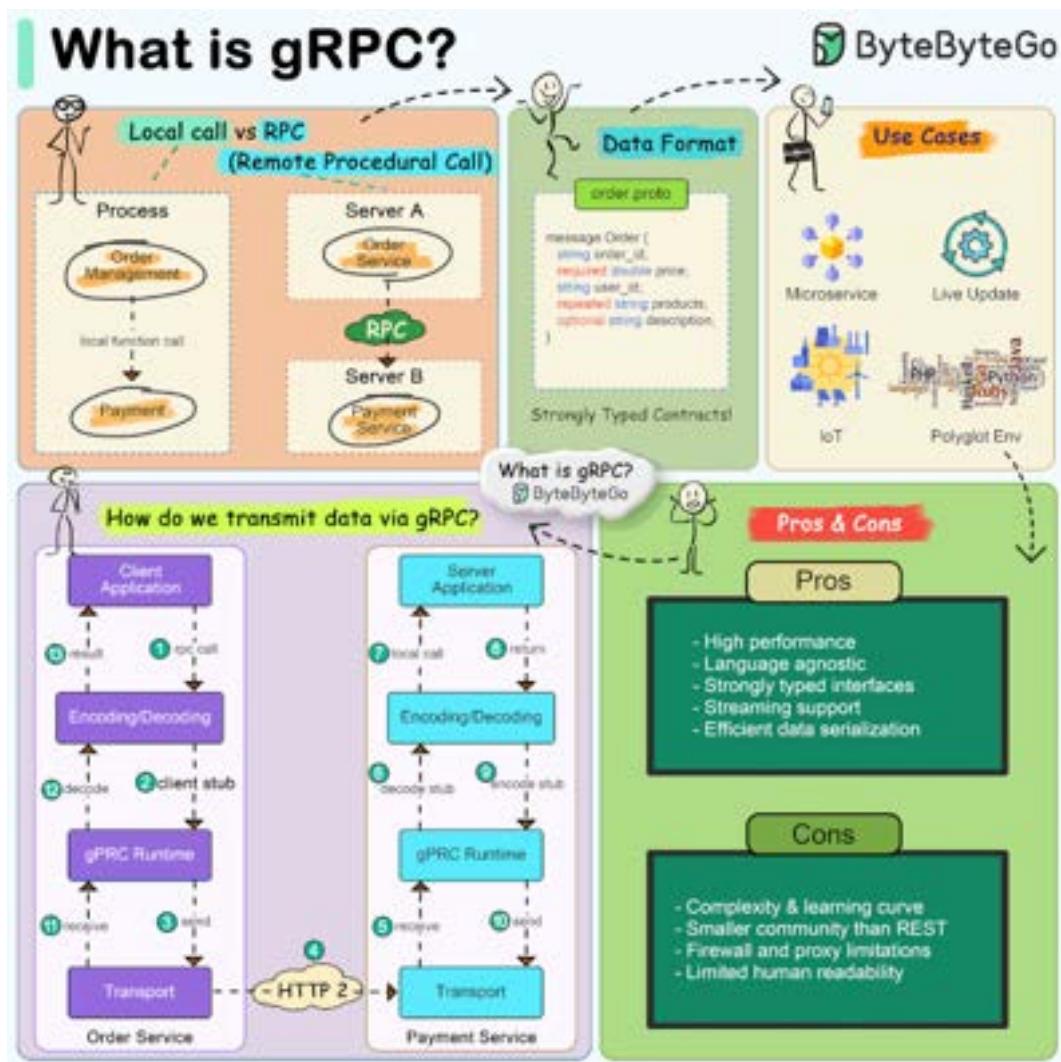
- DDL: data definition language, such as CREATE, ALTER, DROP
- DQL: data query language, such as SELECT
- DML: data manipulation language, such as INSERT, UPDATE, DELETE
- DCL: data control language, such as GRANT, REVOKE
- TCL: transaction control language, such as COMMIT, ROLLBACK

For a backend engineer, you may need to know most of it. As a data analyst, you may need to have a good understanding of DQL. Select the topics that are most relevant to you.

Over to you: What does this SQL statement do in PostgreSQL: "select payload->ids->0 from events"?

What is gRPC?

The diagram below shows important aspects of understanding gRPC.



gRPC is a high-performance, open-source universal RPC (Remote Procedure Call) framework initially developed by Google. It leverages HTTP/2 for transport, Protocol Buffers as the interface description language, and provides features such as authentication, load balancing, and more.

gRPC is designed to enable efficient and robust communication between services in a microservices architecture, making it a popular choice for building distributed systems and APIs.

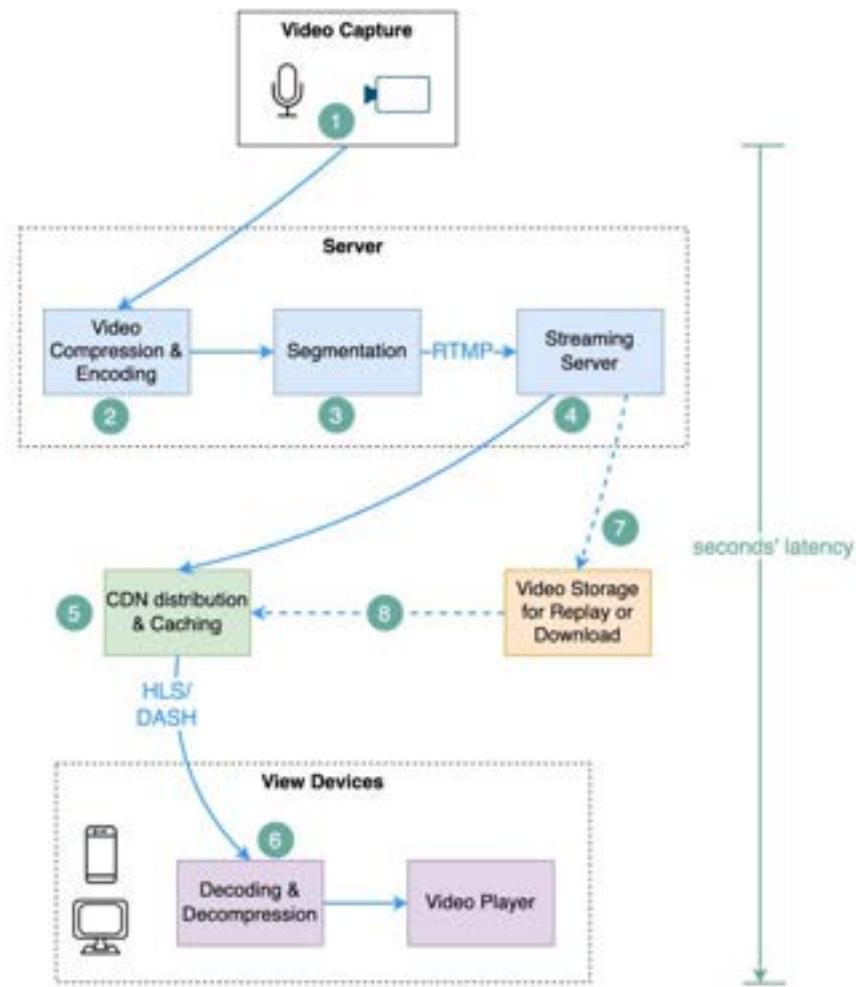
Key Features of gRPC:

1. **Protocol Buffers:** By default, gRPC uses Protocol Buffers (proto files) as its interface definition language (IDL). This makes gRPC messages smaller and faster compared to JSON or XML.
2. **HTTP/2 Based Transport:** gRPC uses HTTP/2 for transport, which allows for many improvements over HTTP/1.x.
3. **Multiple Language Support:** gRPC supports a wide range of programming languages.

4. Bi-Directional Streaming: gRPC supports streaming requests and responses, allowing for the development of sophisticated real-time applications with bidirectional communication like chat services.

How do live streaming platforms like YouTube Live, TikTok

| How does Live Streaming Work? blog.bytebytego.com



Live, or Twitch work?

Live streaming is challenging because the video content is sent over the internet in near real-time. Video processing is compute-intensive. Sending a large volume of video content over the internet takes time. These factors make live streaming challenging.

The diagram below explains what happens behind the scenes to make this possible.

Step 1: The streamer starts their stream. The source could be any video and audio source wired up to an encoder

Step 2: To provide the best upload condition for the streamer, most live streaming platforms provide point-of-presence servers worldwide. The streamer connects to a point-of-presence server closest to them.

Step 3: The incoming video stream is transcoded to different resolutions, and divided into smaller video segments a few seconds in length.

Step 4: The video segments are packaged into different live streaming formats that video players can understand. The most common live-streaming format is HLS, or HTTP Live Streaming.

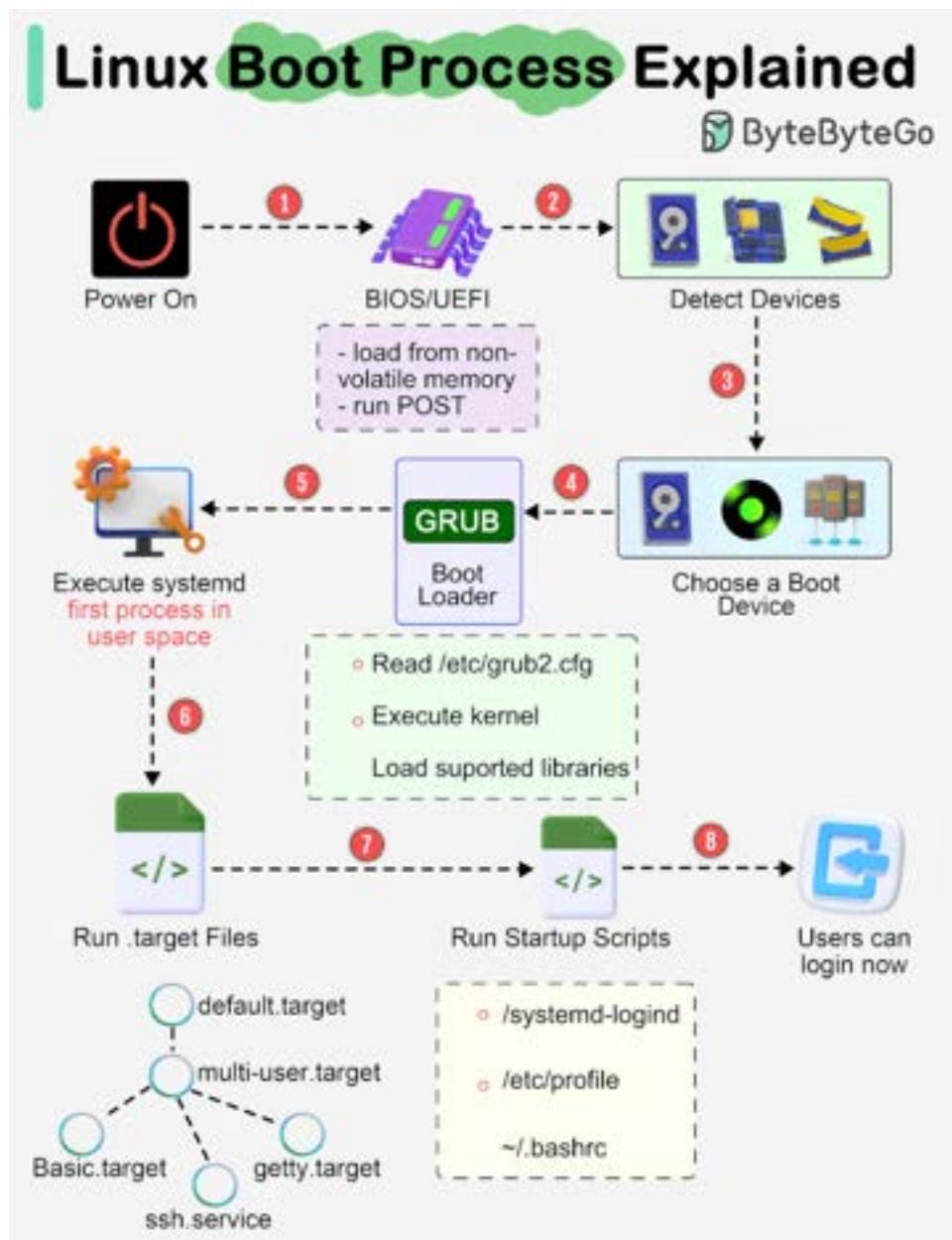
Step 5: The resulting HLS manifest and video chunks from the packaging step are cached by the CDN.

Step 6: Finally, the video starts to arrive at the viewer's video player.

Step 7-8: To support replay, videos can be optionally stored in storage such as Amazon S3.

Linux Boot Process Illustrated

The diagram below shows the steps.



Step 1 - When we turn on the power, BIOS (Basic Input/Output System) or UEFI (Unified Extensible Firmware Interface) firmware is loaded from non-volatile memory, and executes POST (Power On Self Test).

Step 2 - BIOS/UEFI detects the devices connected to the system, including CPU, RAM, and storage.

Step 3 - Choose a booting device to boot the OS from. This can be the hard drive, the network server, or CD ROM.

Step 4 - BIOS/UEFI runs the boot loader (GRUB), which provides a menu to choose the OS or the kernel functions.

Step 5 - After the kernel is ready, we now switch to the user space. The kernel starts up systemd as the first user-space process, which manages the processes and services, probes all remaining hardware, mounts filesystems, and runs a desktop environment.

Step 6 - systemd activates the default. target unit by default when the system boots. Other analysis units are executed as well.

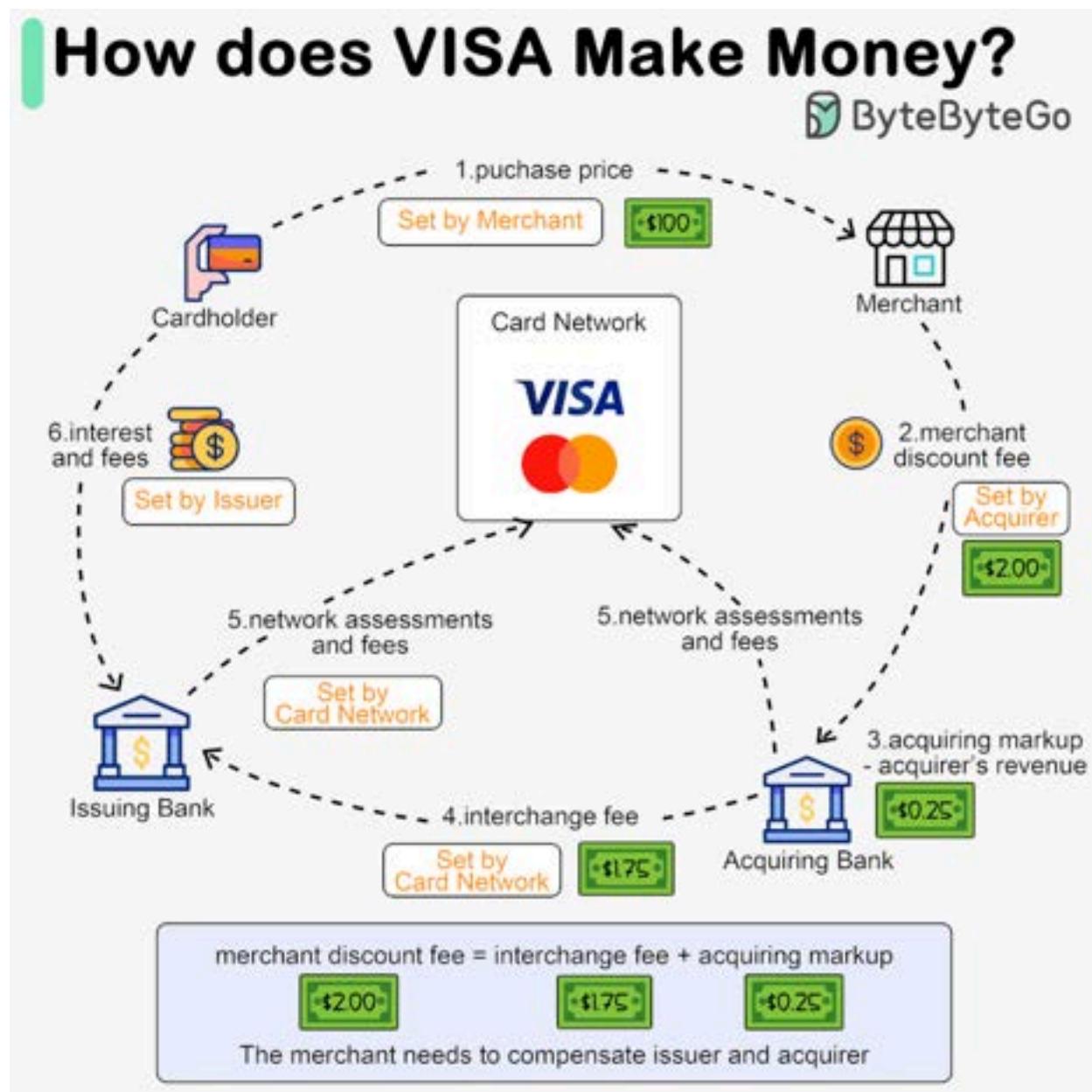
Step 7 - The system runs a set of startup scripts and configures the environment.

Step 8 - The users are presented with a login window. The system is now ready.

How does Visa make money?

Why is the credit card called “the most profitable product in banks”? How does VISA/Mastercard make money?

The diagram below shows the economics of the credit card payment flow.



1. The cardholder pays a merchant \$100 to buy a product.
2. The merchant benefits from the use of the credit card with higher sales volume, and needs to compensate the issuer and the card network for providing the payment service. The acquiring bank sets a fee with the merchant, called the “**merchant discount fee**.”

3 - 4. The acquiring bank keeps \$0.25 as the **acquiring markup**, and \$1.75 is paid to the issuing bank as the **interchange fee**. The merchant discount fee should cover the interchange fee.

The interchange fee is set by the card network because it is less efficient for each issuing bank to negotiate fees with each merchant.

5. The card network sets up the **network assessments and fees** with each bank, which pays the card network for its services every month. For example, VISA charges a 0.11% assessment, plus a \$0.0195 usage fee, for every swipe.

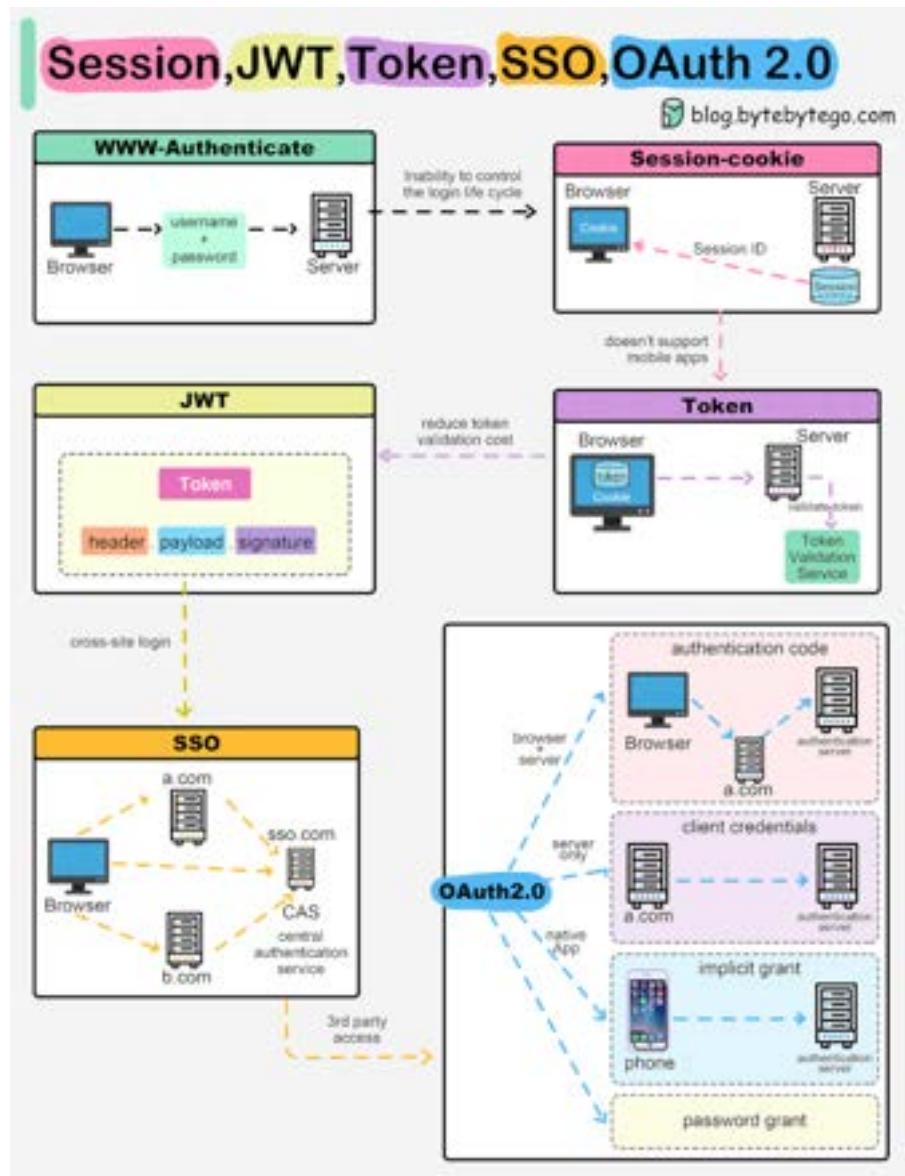
6. The cardholder pays the issuing bank for its services.

Why should the issuing bank be compensated?

- The issuer pays the merchant even if the cardholder fails to pay the issuer.
- The issuer pays the merchant before the cardholder pays the issuer.
- The issuer has other operating costs, including managing customer accounts, providing statements, fraud detection, risk management, clearing & settlement, etc.

Over to you: Does the card network charge the same interchange fee for big merchants as for small merchants?

Session, Cookie, JWT, Token, SSO, and OAuth 2.0 Explained in One Diagram



When you login to a website, your identity needs to be managed. Here is how different solutions work:

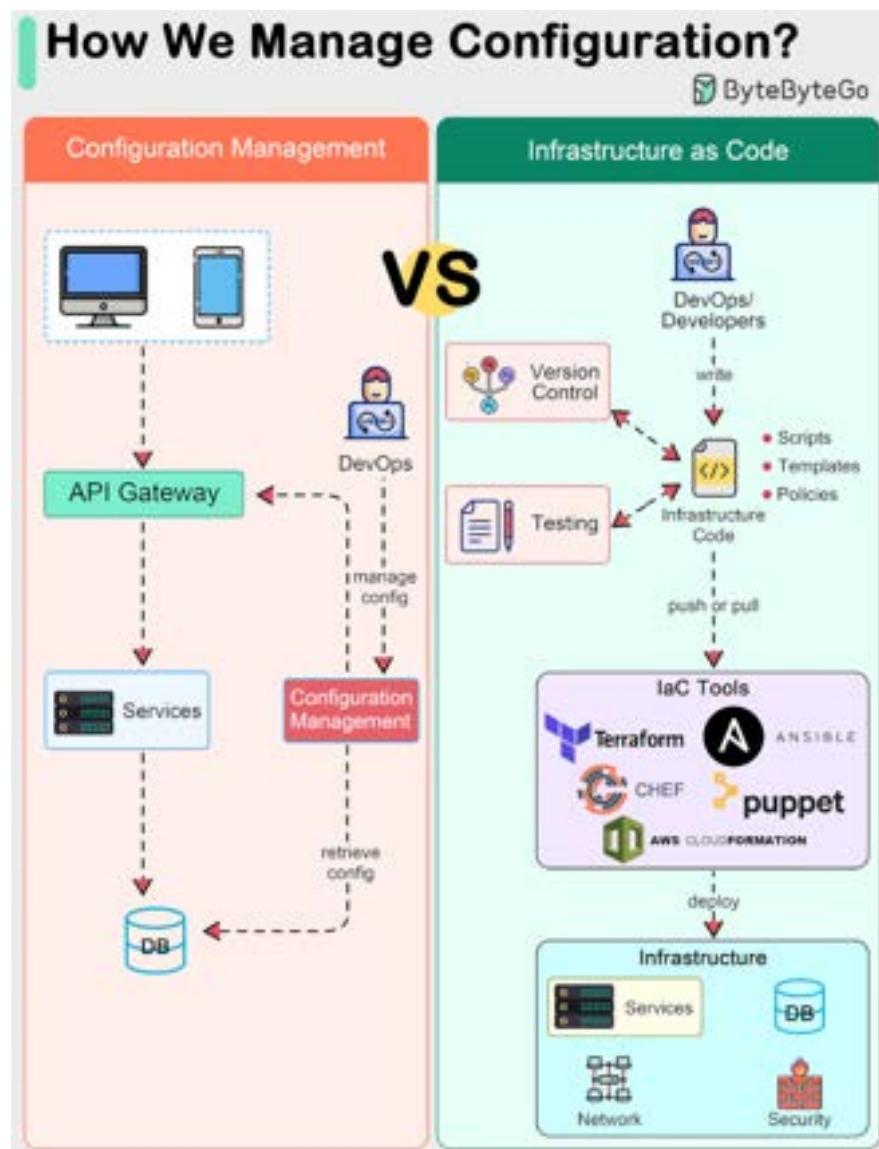
- Session - The server stores your identity and gives the browser a session ID cookie. This allows the server to track login state. But cookies don't work well across devices.
- Token - Your identity is encoded into a token sent to the browser. The browser sends this token on future requests for authentication. No server session storage is required. But tokens need encryption/decryption.
- JWT - JSON Web Tokens standardize identity tokens using digital signatures for trust. The signature is contained in the token so no server session is needed.

- SSO - Single Sign On uses a central authentication service. This allows a single login to work across multiple sites.
- OAuth2 - Allows limited access to your data on one site by another site, without giving away passwords.
- QR Code - Encodes a random token into a QR code for mobile login. Scanning the code logs you in without typing a password.

Over to you: QR code logins are gaining popularity. Do you know how it works?

How do we manage configurations in a system?

The diagram shows a comparison between traditional configuration management and IaC (Infrastructure as Code).



- Configuration Management

The practice is designed to manage and provision IT infrastructure through systematic and repeatable processes. This is critical for ensuring that the system performs as intended.

Traditional configuration management focuses on maintaining the desired state of the system's configuration items, such as servers, network devices, and applications, after they have been provisioned.

It usually involves initial manual setup by DevOps. Changes are managed by step-by-step commands.

- What is IaC?

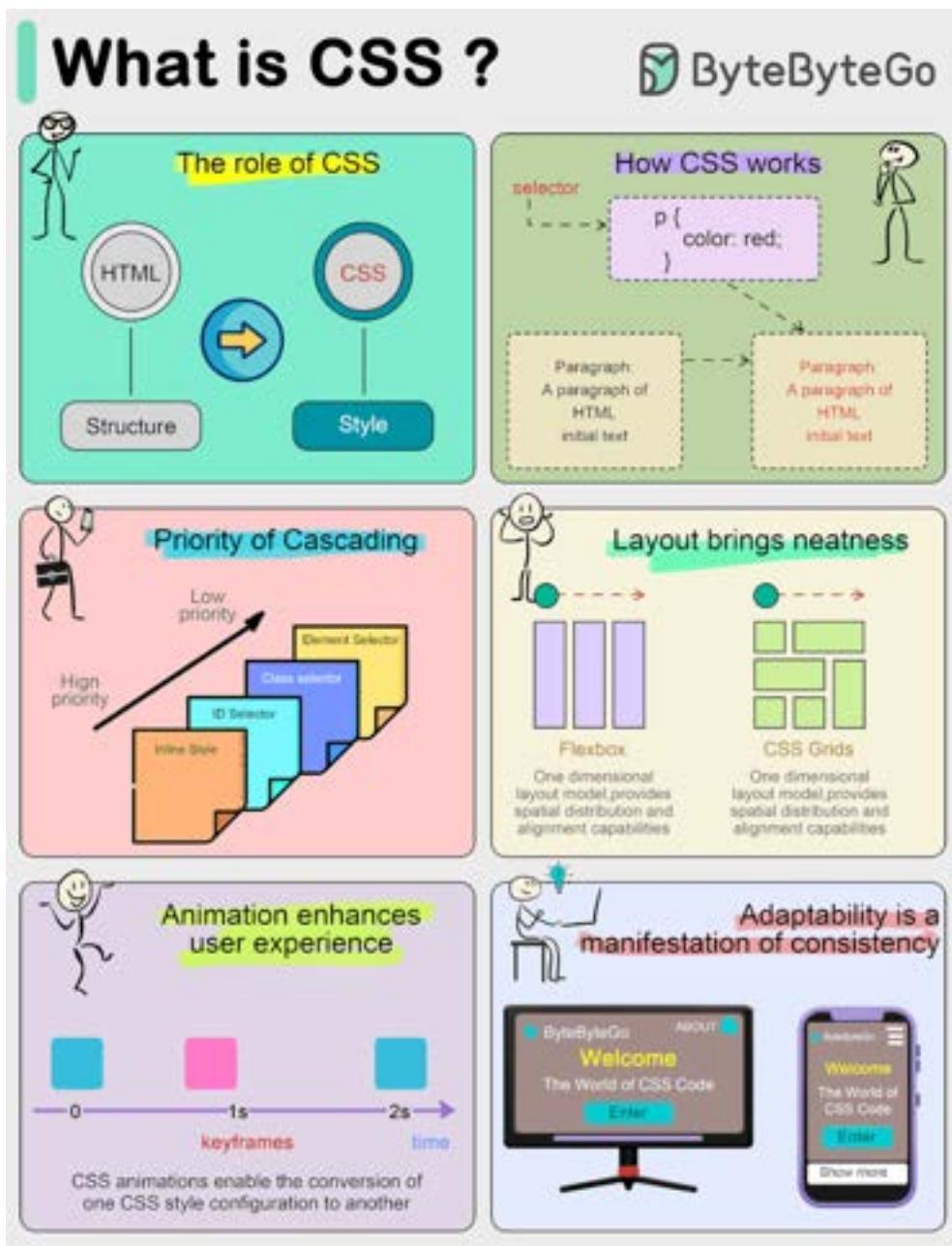
IaC, on the hand, represents a shift in how infrastructure is provisioned and managed, treating infrastructure setup and changes as software development practices.

IaC automates the provisioning of infrastructure, starting and managing the system through code. It often uses a declarative approach, where the desired state of the infrastructure is described.

Tools like Terraform, AWS CloudFormation, Chef, and Puppet are used to define infrastructure in code files that are source controlled.

IaC represents an evolution towards automation, repeatability, and the application of software development practices to infrastructure management.

What is CSS (Cascading Style Sheets)?



Front-end development requires not only content presentation, but also good-looking. CSS is a markup language used to describe how elements on a web page should be rendered.

▶ What CSS does?

CSS separates the content and presentation of a document. In the early days of web development, HTML acted as both content and style.

CSS divides structure (HTML) and style (CSS). This has many benefits, for example, when we change the color scheme of a web page, all we need to do is to tweak the CSS file.

How CSS works?

CSS consists of a selector and a set of properties, which can be thought of as individual rules. Selectors are used to locate HTML elements that we want to change the style of, and properties are the specific style descriptions for those elements, such as color, size, position, etc.

For example, if we want to make all the text in a paragraph blue, we write CSS code like this:

```
p { color: blue; }
```

Here “p” is the selector and “color: blue” is the attribute that declares the color of the paragraph text to be blue.

Cascading in CSS

The concept of cascading is crucial to understanding CSS.

When multiple style rules conflict, the browser needs to decide which rule to use based on a specific prioritization rule. The one with the highest weight wins. The weight can be determined by a variety of factors, including selector type and the order of the source.

Powerful Layout Capabilities of CSS

In the past, CSS was only used for simple visual effects such as text colors, font styles, or backgrounds. Today, CSS has evolved into a powerful layout tool capable of handling complex design layouts.

The “Flexbox” and “Grid” layout modules are two popular CSS layout modules that make it easy to create responsive designs and precise placement of web elements, so web developers no longer have to rely on complex tables or floating layouts.

CSS Animation

Animation and interactive elements can greatly enhance the user experience.

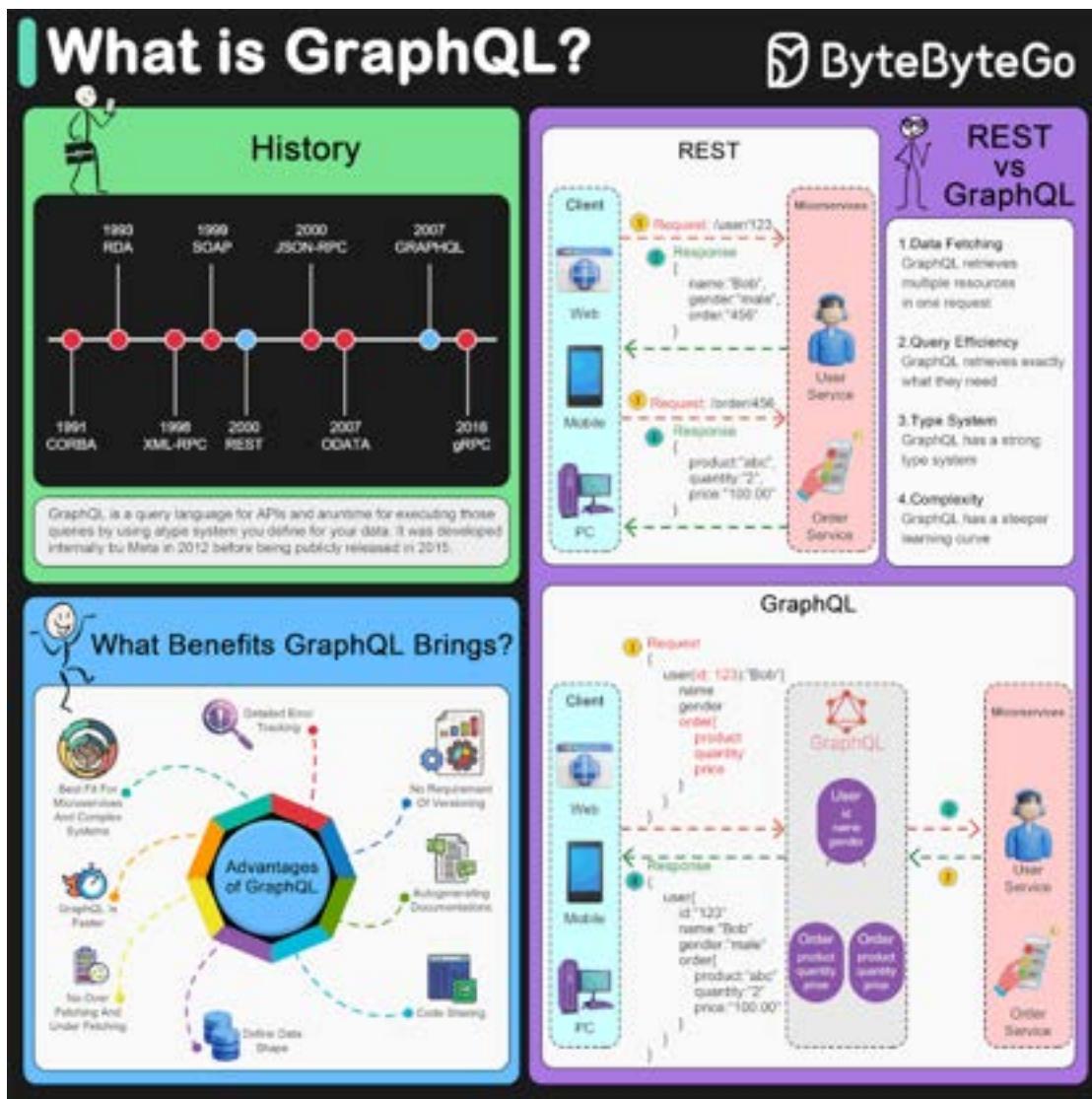
CSS3 introduces animation features that allow us to transform and animate elements without using JavaScript. For example, “@keyframes” rule defines animation sequences, and the ‘transition’ property can be used to set animated transitions from one state to another.

Responsive Design

CSS allows the layout and style of a website to be adapted to different screen sizes and resolutions, so that we can provide an optimized browsing experience for different devices such as cell phones, tablets and computers.

What is GraphQL? Is it a replacement for the REST API?

The diagram below explains different aspects of GraphQL.



GraphQL is a query language for APIs and a runtime for executing those queries by using a type system you define for your data. It was developed internally by Meta in 2012 before being publicly released in 2015.

Unlike the more traditional REST API, GraphQL allows clients to request exactly the data they need, making it possible to fetch data from multiple sources with a single query. This efficiency in data retrieval can lead to improved performance for web and mobile applications.

GraphQL servers sit in between the client and the backend services. It can aggregate multiple REST requests into one query. GraphQL server organizes the resources in a graph.

GraphQL supports queries, mutations (applying data modifications to resources), and subscriptions (receiving notifications on schema modifications).

Benefits of GraphQL:

1. GraphQL is more efficient in data fetching.
2. GraphQL returns more accurate results.
3. GraphQL has a strong type system to manage the structure of entities, reducing errors.
4. GraphQL is suitable for managing complex microservices.

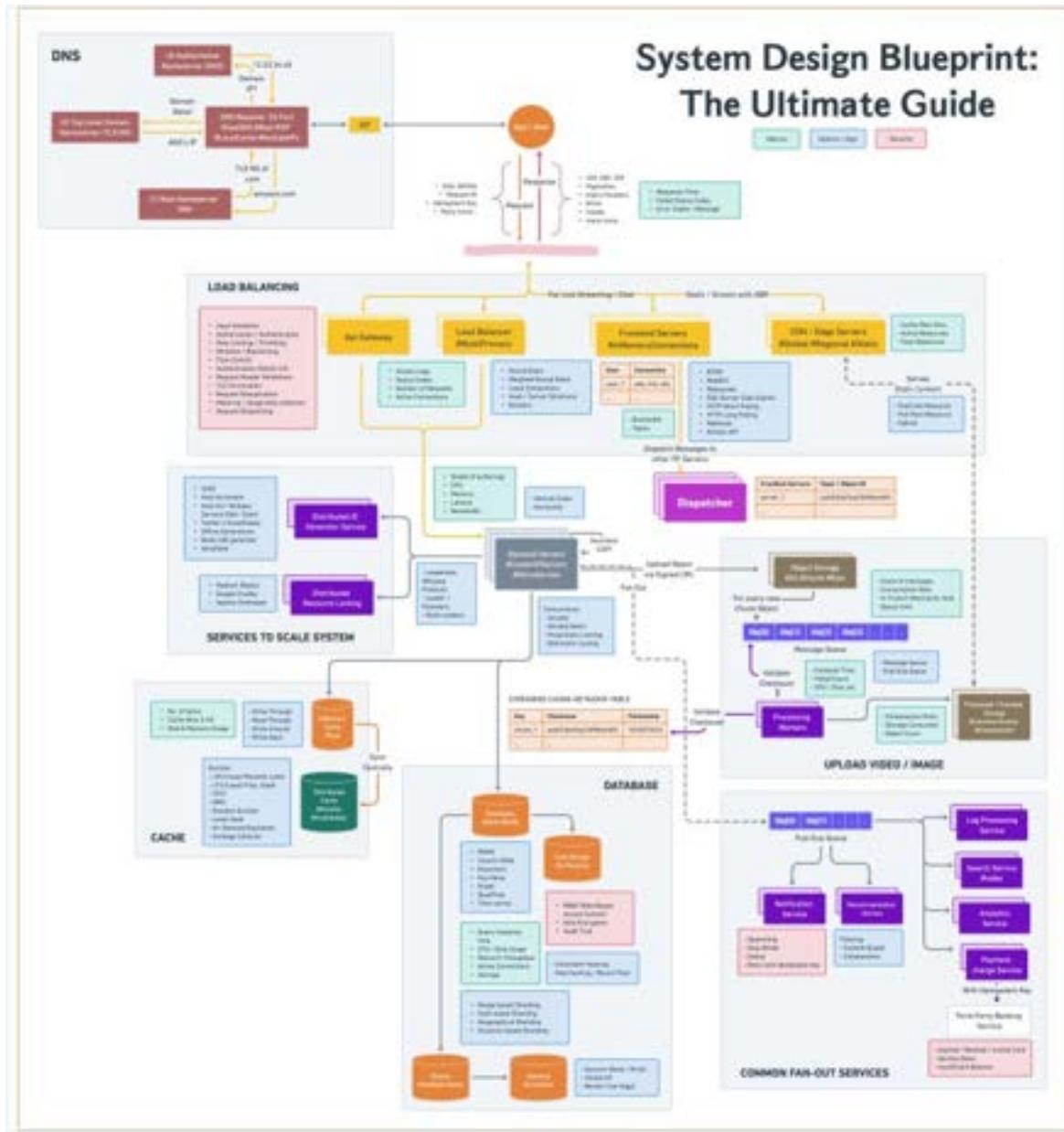
Disadvantages of GraphQL

- Increased complexity.
- Over fetching by design
- Caching complexity

System Design Blueprint: The Ultimate Guide

We've created a template to tackle various system design problems in interviews.

Hope this checklist is useful to guide your discussions during the interview process.

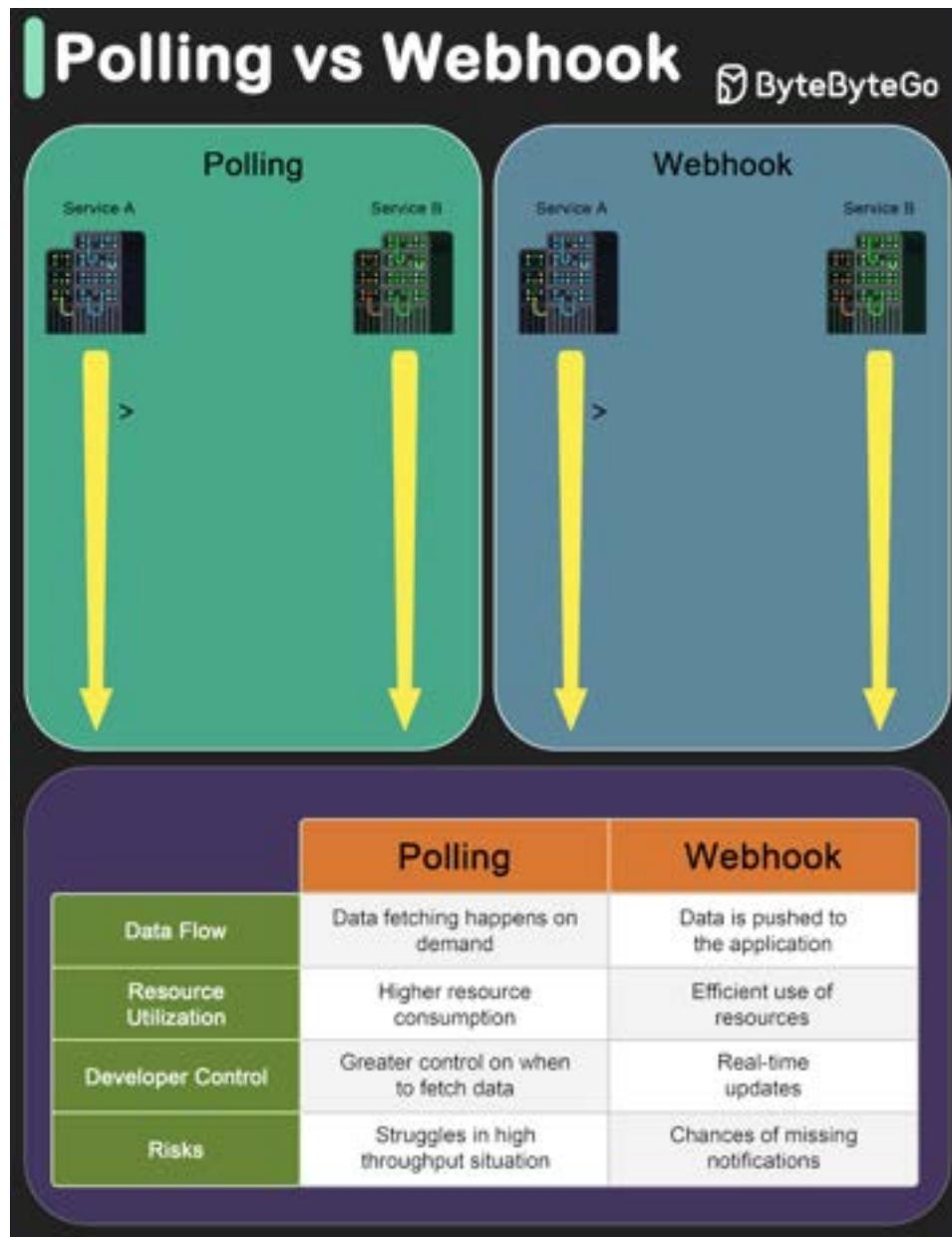


This briefly touches on the following discussion points:

- Load Balancing
 - API Gateway
 - Communication Protocols
 - Content Delivery Network (CDN)

- Database
- Cache
- Message Queue
- Unique ID Generation
- Scalability
- Availability
- Performance
- Security
- Fault Tolerance and Resilience
- And more

Polling Vs Webhooks



Polling

Polling involves repeatedly checking the external service or endpoint at fixed intervals to retrieve updated information.

It's like constantly asking, "Do you have something new for me?" even where there might not be any update.

This approach is resource-intensive and inefficient.

Also, you get updates only when you ask for it, thereby missing any real-time information.

However, developers have more control over when and how the data is fetched.

- Webhooks

Webhooks are like having a built-in notification system.

You don't continuously ask for information.

Instead you create an endpoint in your application server and provide it as a callback to the external service (such as a payment processor or a shipping vendor)

Every time something interesting happens, the external service calls the endpoint and provides the information.

This makes webhooks ideal for dealing with real-time updates because data is pushed to your application as soon as it's available.

So, when to use Polling or Webhook?

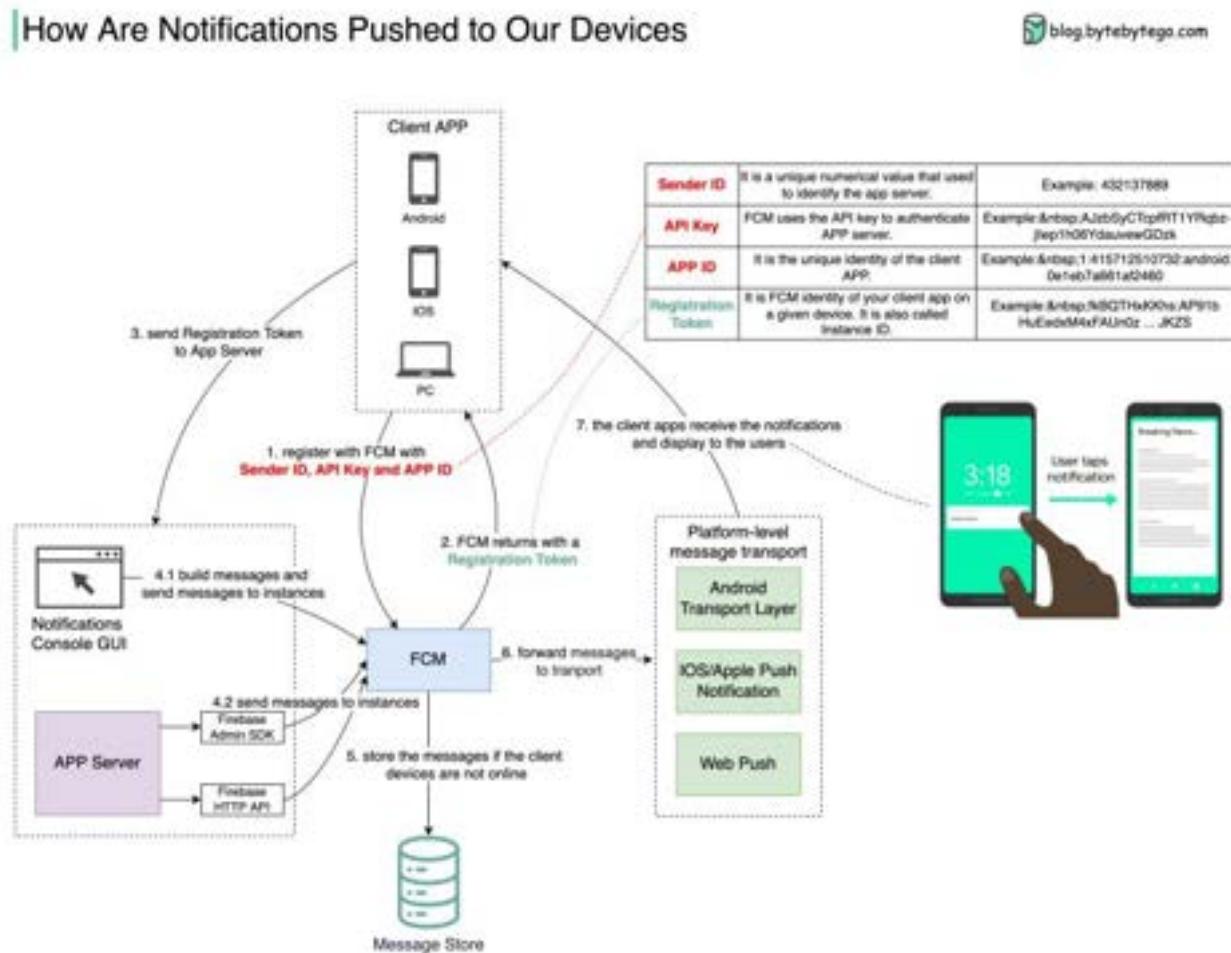
Polling is a solid option when there is some infrastructural limitation that prevents the use of webhooks. Also, with webhooks there is a risk of missed notifications due to network issues, hence proper retry mechanisms are needed.

Webhooks are recommended for applications that need instant data delivery. Also, webhooks are efficient in terms of resource utilization especially in high throughput environments.

How are notifications pushed to our phones or PCs?

A messaging solution (Firebase) can be used to support the notification push.

The diagram below shows how Firebase Cloud Messaging (FCM) works.



FCM is a cross-platform messaging solution that can compose, send, queue, and route notifications reliably. It provides a unified API between message senders (app servers) and receivers (client apps). The app developer can use this solution to drive user retention.

Steps 1 - 2: When the client app starts for the first time, the client app sends credentials to FCM, including Sender ID, API Key, and App ID. FCM generates Registration Token for the client app instance (so the Registration Token is also called Instance ID). This token must be included in the notifications.

Step 3: The client app sends the Registration Token to the app server. The app server caches the token for subsequent communications. Over time, the app server has too many tokens to maintain, so the recommended practice is to store the token with timestamps and to remove stale tokens from time to time.

Step 4: There are two ways to send messages. One is to compose messages directly in the console GUI (Step 4.1,) and the other is to send the messages from the app server (Step 4.2.) We can use the Firebase Admin SDK or HTTP for the latter.

Step 5: FCM receives the messages, and queues the messages in the storage if the devices are not online.

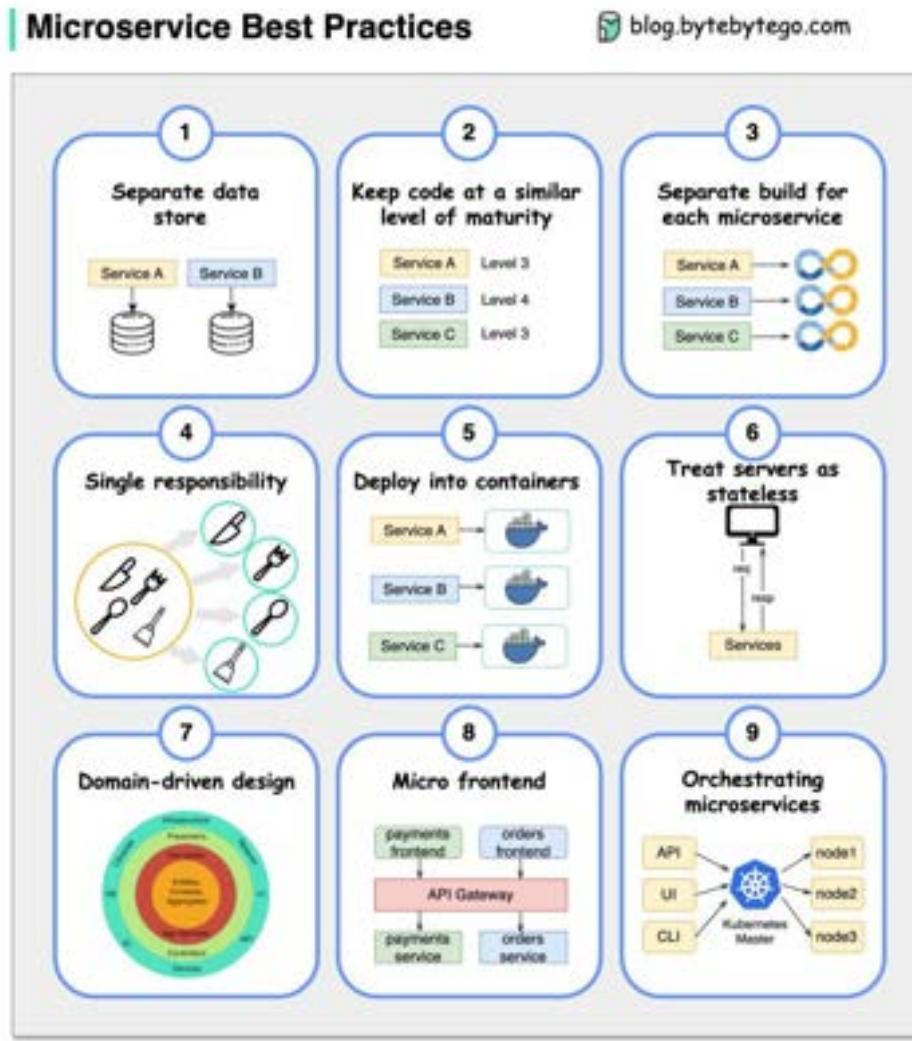
Step 6: FCM forwards the messages to platform-level transport. This transport layer handles platform-specific configurations.

Step 7: The messages are routed to the targeted devices. The notifications can be displayed according to the configurations sent from the app server [1].

Over to you: We can also send messages to a “topic” (just like Kafka) in Step 4. When should the client app subscribe to the topic?

Reference Material: [Google firebase documentation](#)

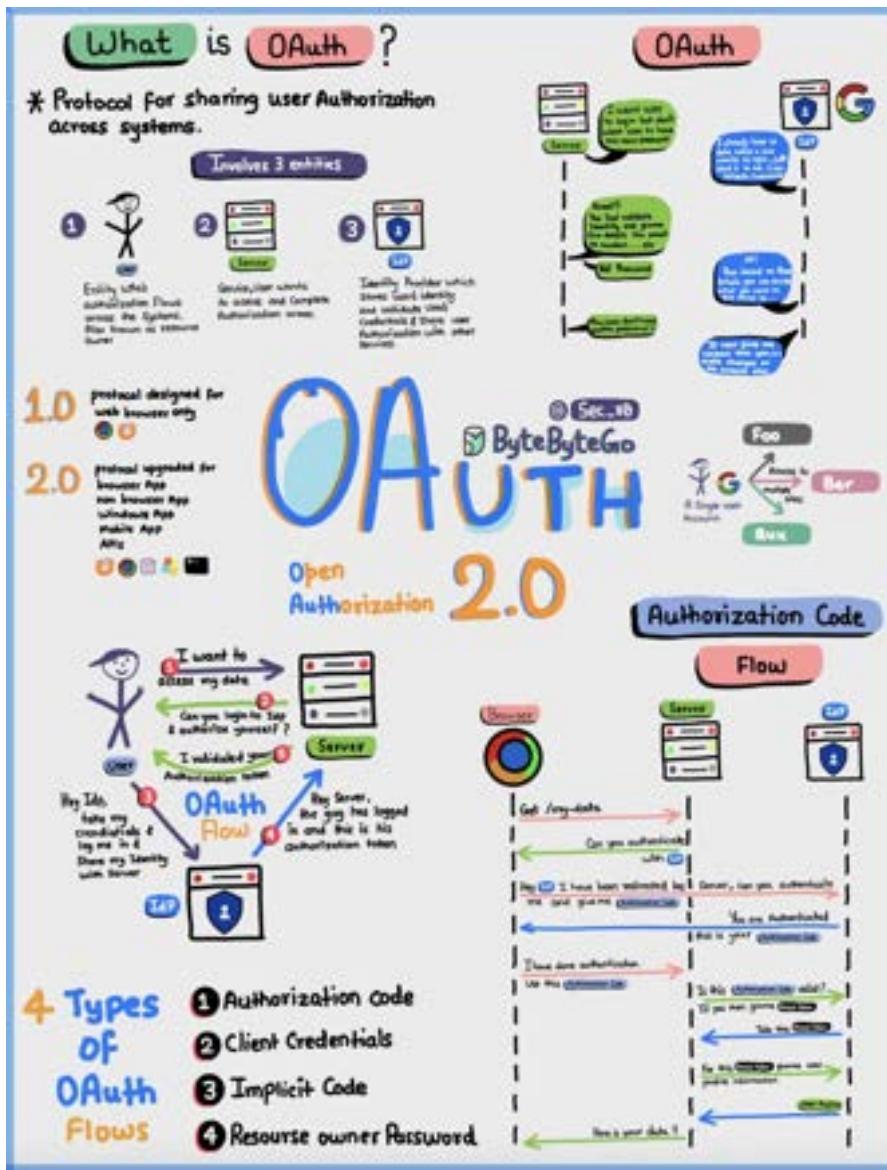
9 best practices for developing microservices



When we develop microservices, we need to follow the following best practices:

1. Use separate data storage for each microservice
2. Keep code at a similar level of maturity
3. Separate build for each microservice
4. Assign each microservice with a single responsibility
5. Deploy into containers
6. Design stateless services
7. Adopt domain-driven design
8. Design micro frontend
9. Orchestrating microservices

Oauth 2.0 Explained With Simple Terms



OAuth 2.0 is a powerful and secure framework that allows different applications to securely interact with each other on behalf of users without sharing sensitive credentials.

The entities involved in OAuth are the User, the Server, and the Identity Provider (IDP).

What Can an OAuth Token Do?

When you use OAuth, you get an OAuth token that represents your identity and permissions. This token can do a few important things:

Single Sign-On (SSO): With an OAuth token, you can log into multiple services or apps using just one login, making life easier and safer.

Authorization Across Systems: The OAuth token allows you to share your authorization or access rights across various systems, so you don't have to log in separately everywhere.

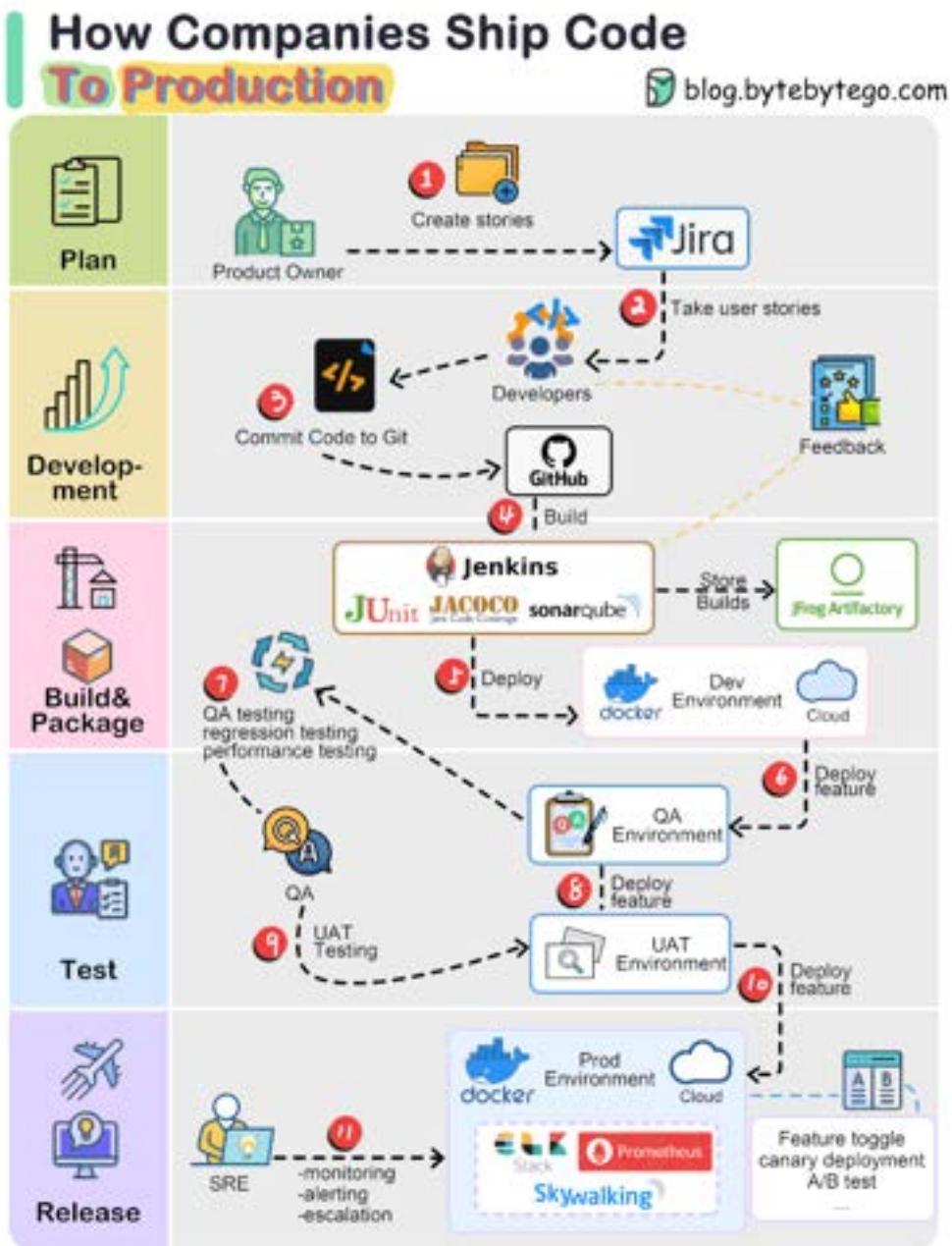
Accessing User Profile: Apps with an OAuth token can access certain parts of your user profile that you allow, but they won't see everything.

Remember, OAuth 2.0 is all about keeping you and your data safe while making your online experiences seamless and hassle-free across different applications and services.

Over to you: Imagine you have a magical power to grant one wish to OAuth 2.0. What would that be? Maybe your suggestions actually lead to OAuth 3.

How do companies ship code to production?

The diagram below illustrates the typical workflow.



Step 1: The process starts with a product owner creating user stories based on requirements.

Step 2: The dev team picks up the user stories from the backlog and puts them into a sprint for a two-week dev cycle.

Step 3: The developers commit source code into the code repository Git.

Step 4: A build is triggered in Jenkins. The source code must pass unit tests, code coverage threshold, and gates in SonarQube.

Step 5: Once the build is successful, the build is stored in artifactory. Then the build is deployed into the dev environment.

Step 6: There might be multiple dev teams working on different features. The features need to be tested independently, so they are deployed to QA1 and QA2.

Step 7: The QA team picks up the new QA environments and performs QA testing, regression testing, and performance testing.

Step 8: Once the QA builds pass the QA team's verification, they are deployed to the UAT environment.

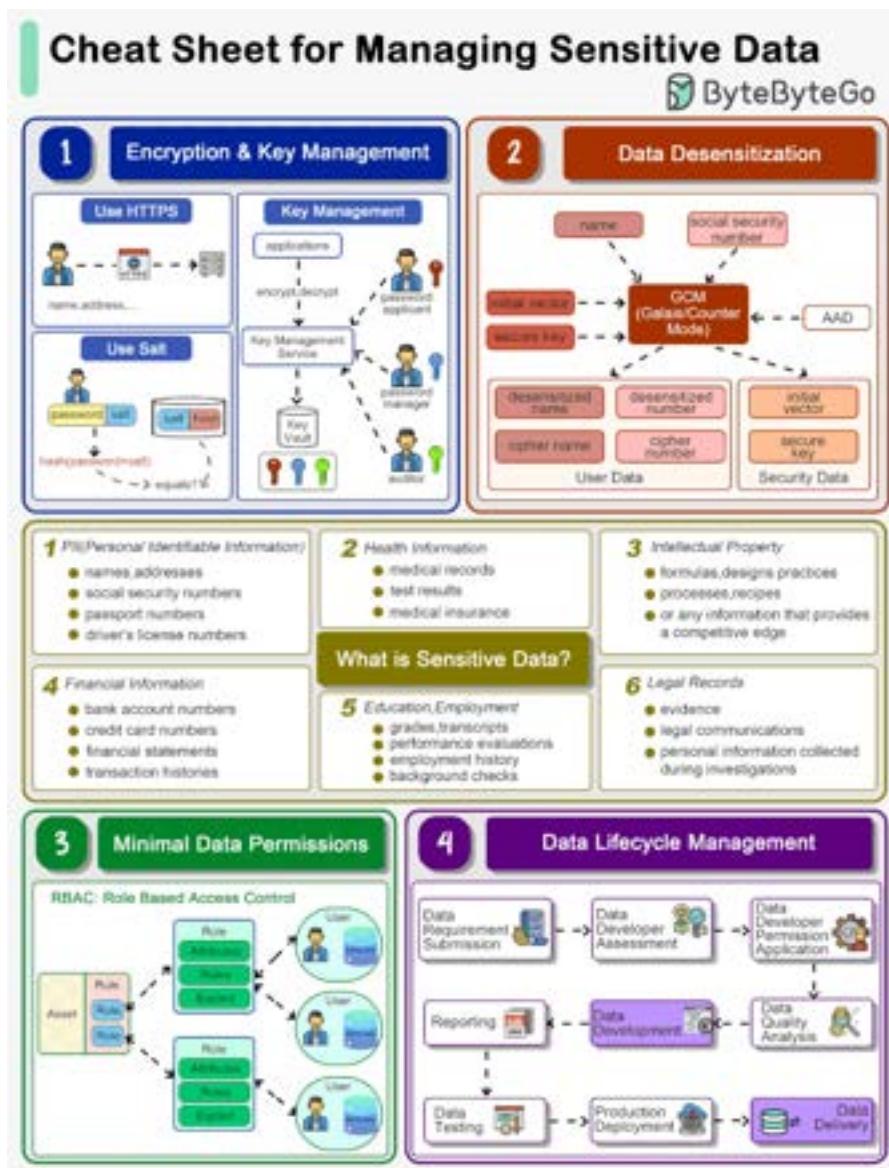
Step 9: If the UAT testing is successful, the builds become release candidates and will be deployed to the production environment on schedule.

Step 10: SRE (Site Reliability Engineering) team is responsible for prod monitoring.

Over to you: what's your company's release process look like?

How do we manage sensitive data in a system?

The cheat sheet below shows a list of guidelines.



◆ What is Sensitive Data?

Personal Identifiable Information (PII), health information, intellectual property, financial information, education and legal records are all sensitive data.

Most countries have laws and regulations that require the protection of sensitive data. For example, the General Data Protection Regulation (GDPR) in the European Union sets stringent rules for data protection and privacy. Non-compliance with such regulations can result in hefty fines, legal actions, and sanctions against the violating entity.

When we design systems, we need to design for data protection.

◆ Encryption & Key Management

The data transmission needs to be encrypted using SSL. Passwords shouldn't be stored in plain text.

For key storage, we design different roles including password applicant, password manager and auditor, all holding one piece of the key. We will need all three keys to open a lock.

◆ Data Desensitization

Data desensitization, also known as data anonymization or data sanitization, refers to the process of removing or modifying personal information from a dataset so that individuals cannot be readily identified. This practice is crucial in protecting individuals' privacy and ensuring compliance with data protection laws and regulations. Data desensitization is often used when sharing data externally, such as for research or statistical analysis, or even internally within an organization, to limit access to sensitive information.

Algorithms like GCM store cipher data and keys separately so that hackers are not able to decipher the user data.

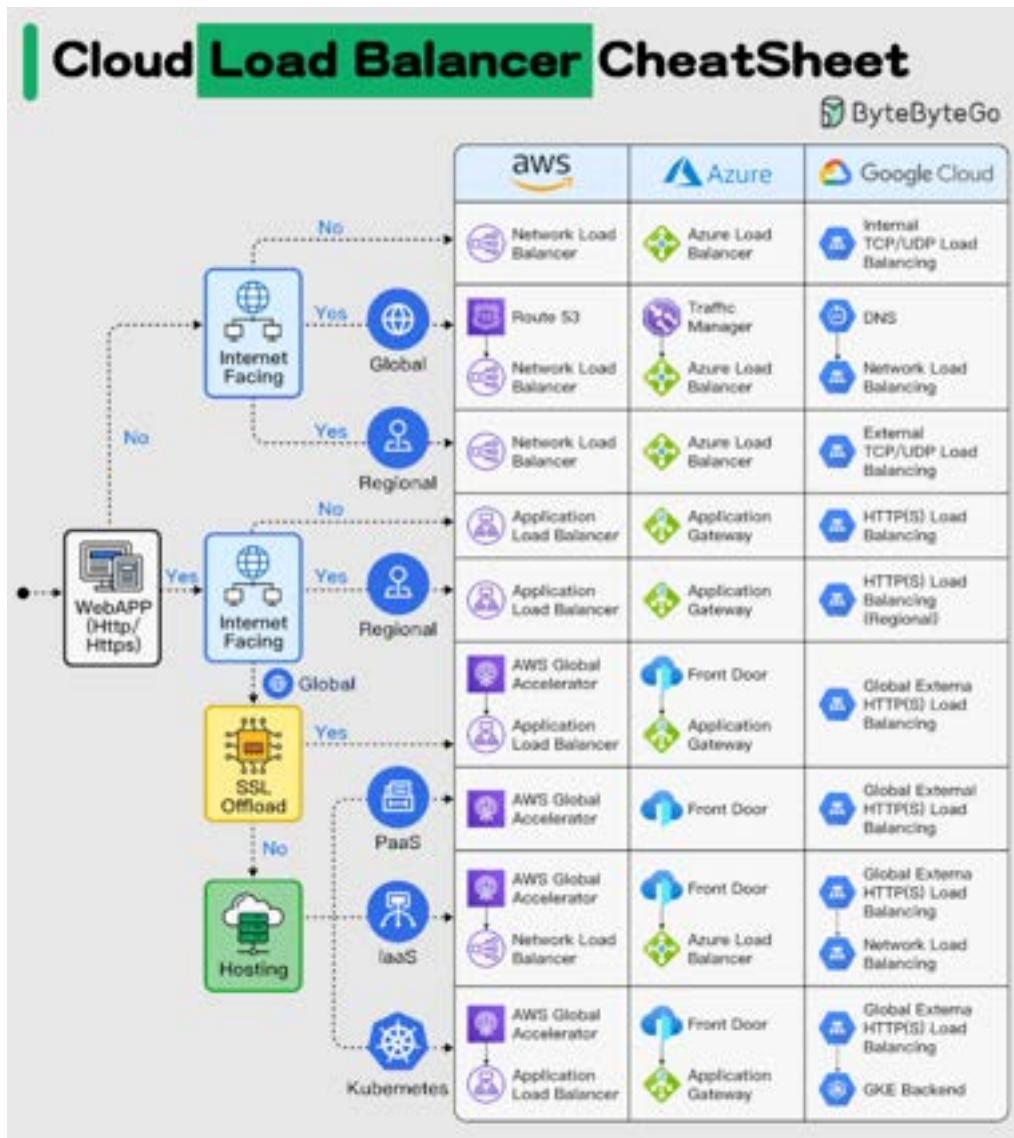
◆ Minimal Data Permissions

To protect sensitive data, we should grant minimal permissions to the users. Often we design Role-Based Access Control (RBAC) to restrict access to authorized users based on their roles within an organization. It is a widely used access control mechanism that simplifies the management of user permissions, ensuring that users have access to only the information and resources necessary for their roles.

◆ Data Lifecycle Management

When we develop data products like reports or data feeds, we need to design a process to maintain data quality. Data developers should be granted with necessary permissions during development. After the data is online, they should be revoked from the data access.

Cloud Load Balancer Cheat Sheet



Efficient load balancing is vital for optimizing the performance and availability of your applications in the cloud.

However, managing load balancers can be overwhelming, given the various types and configuration options available.

In today's multi-cloud landscape, mastering load balancing is essential to ensure seamless user experiences and maximize resource utilization, especially when orchestrating applications across multiple cloud providers. Having the right knowledge is key to overcoming these challenges and achieving consistent, reliable application delivery.

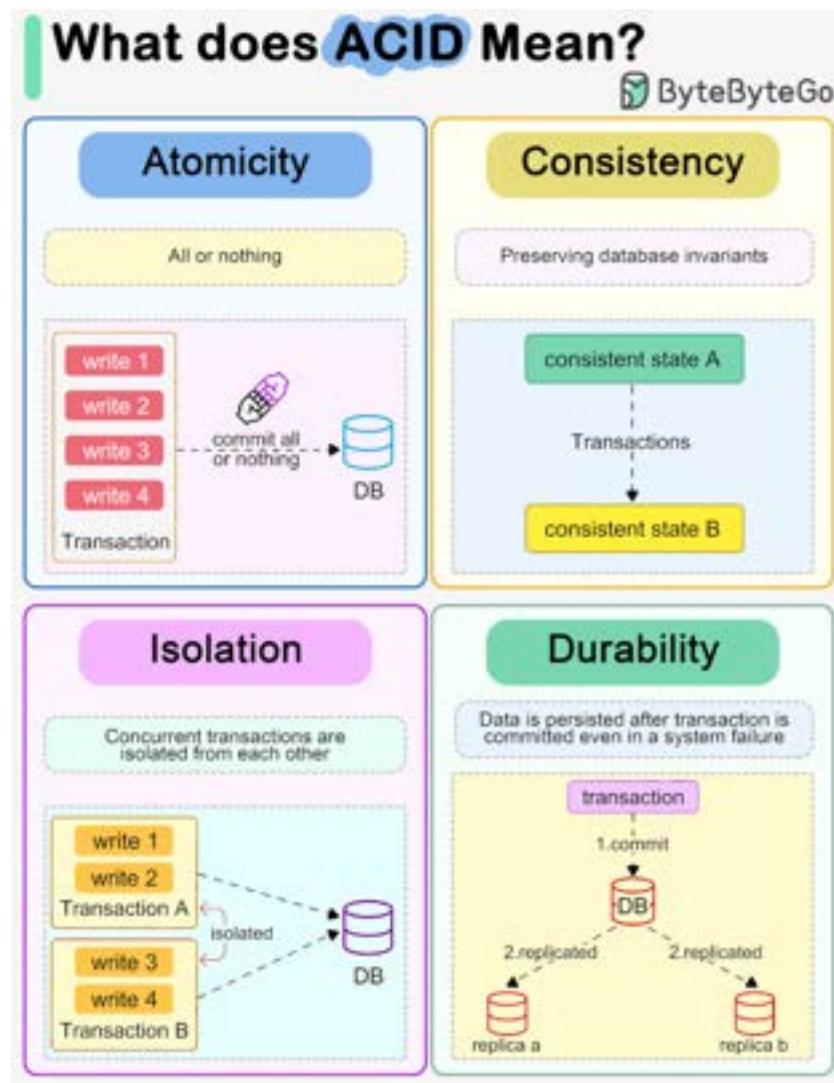
In selecting the appropriate load balancer type, it's essential to consider factors such as application traffic patterns, scalability requirements, and security considerations. By carefully evaluating your specific use case, you can make informed decisions that enhance your cloud infrastructure's efficiency and reliability.

This Cloud Load Balancer cheat sheet would help you in simplifying the decision-making process and helping you implement the most effective load balancing strategy for your cloud-based applications.

Over to you: What factors do you believe are most crucial in choosing the right load balancer type for your applications?

What does ACID mean?

The diagram below explains what ACID means in the context of a database transaction.



◆ Atomicity

The writes in a transaction are executed all at once and cannot be broken into smaller parts. If there are faults when executing the transaction, the writes in the transaction are rolled back.

So atomicity means “all or nothing”.

◆ Consistency

Unlike “consistency” in CAP theorem, which means every read receives the most recent write or an error, here consistency means preserving database invariants. Any data written by a transaction must be valid according to all defined rules and maintain the database in a good state.

◆ Isolation

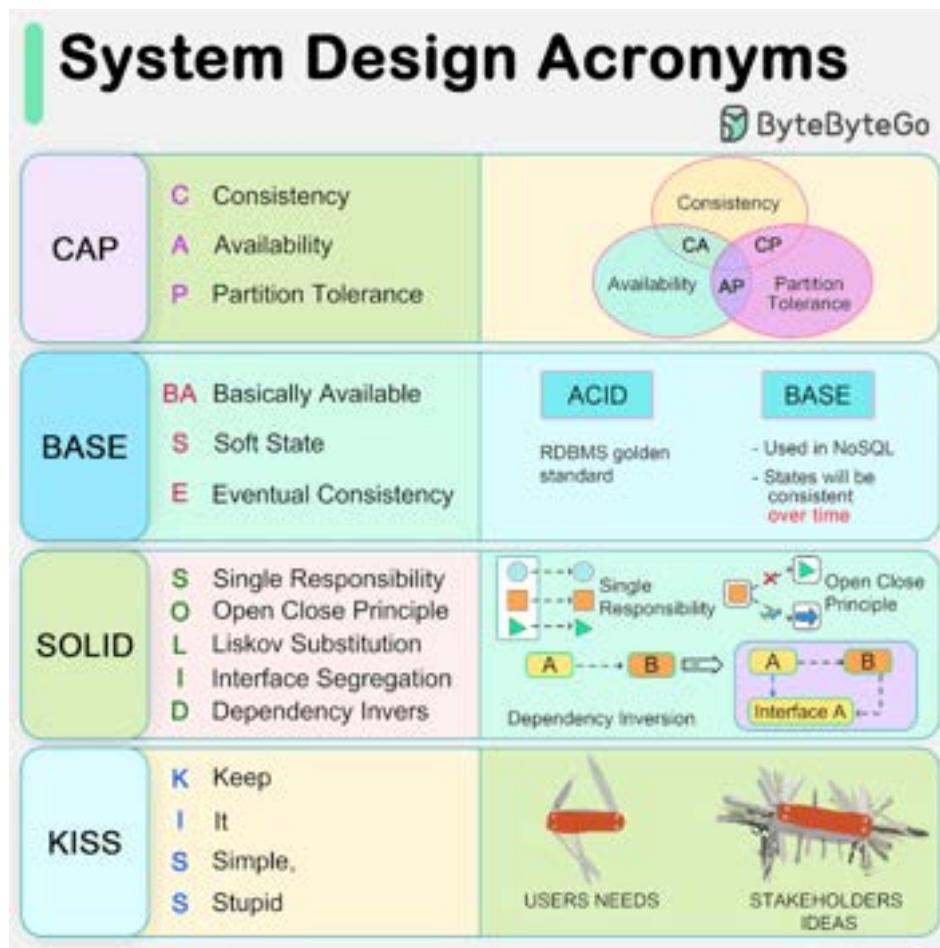
When there are concurrent writes from two different transactions, the two transactions are isolated from each other. The most strict isolation is “serializability”, where each transaction acts like it is the only transaction running in the database. However, this is hard to implement in reality, so we often adopt loser isolation level.

- ◆ Durability

Data is persisted after a transaction is committed even in a system failure. In a distributed system, this means the data is replicated to some other nodes.

CAP, BASE, SOLID, KISS, What do these acronyms mean?

The diagram below explains the common acronyms in system designs.



◆ CAP

CAP theorem states that any distributed data store can only provide two of the following three guarantees:

1. Consistency - Every read receives the most recent write or an error.
2. Availability - Every request receives a response.
3. Partition tolerance - The system continues to operate in network faults.

However, this theorem was criticized for being too narrow for distributed systems, and we shouldn't use it to categorize the databases. Network faults are guaranteed to happen in distributed systems, and we must deal with this in any distributed systems.

You can read more on this in "Please stop calling databases CP or AP" by Martin Kleppmann.

◆ BASE

The ACID (Atomicity-Consistency-Isolation-Durability) model used in relational databases is too strict for NoSQL databases. The BASE principle offers more flexibility, choosing availability over consistency. It states that the states will eventually be consistent.

◆ SOLID

SOLID principle is quite famous in OOP. There are 5 components to it.

1. SRP (Single Responsibility Principle)

Each unit of code should have one responsibility.

2. OCP (Open Close Principle)

Units of code should be open for extension but closed for modification.

3. LSP (Liskov Substitution Principle)

A subclass should be able to be substituted by its base class.

4. ISP (Interface Segregation Principle)

Expose multiple interfaces with specific responsibilities.

5. DIP (Dependency Inversion Principle)

Use abstractions to decouple dependencies in the system.

◆ KISS

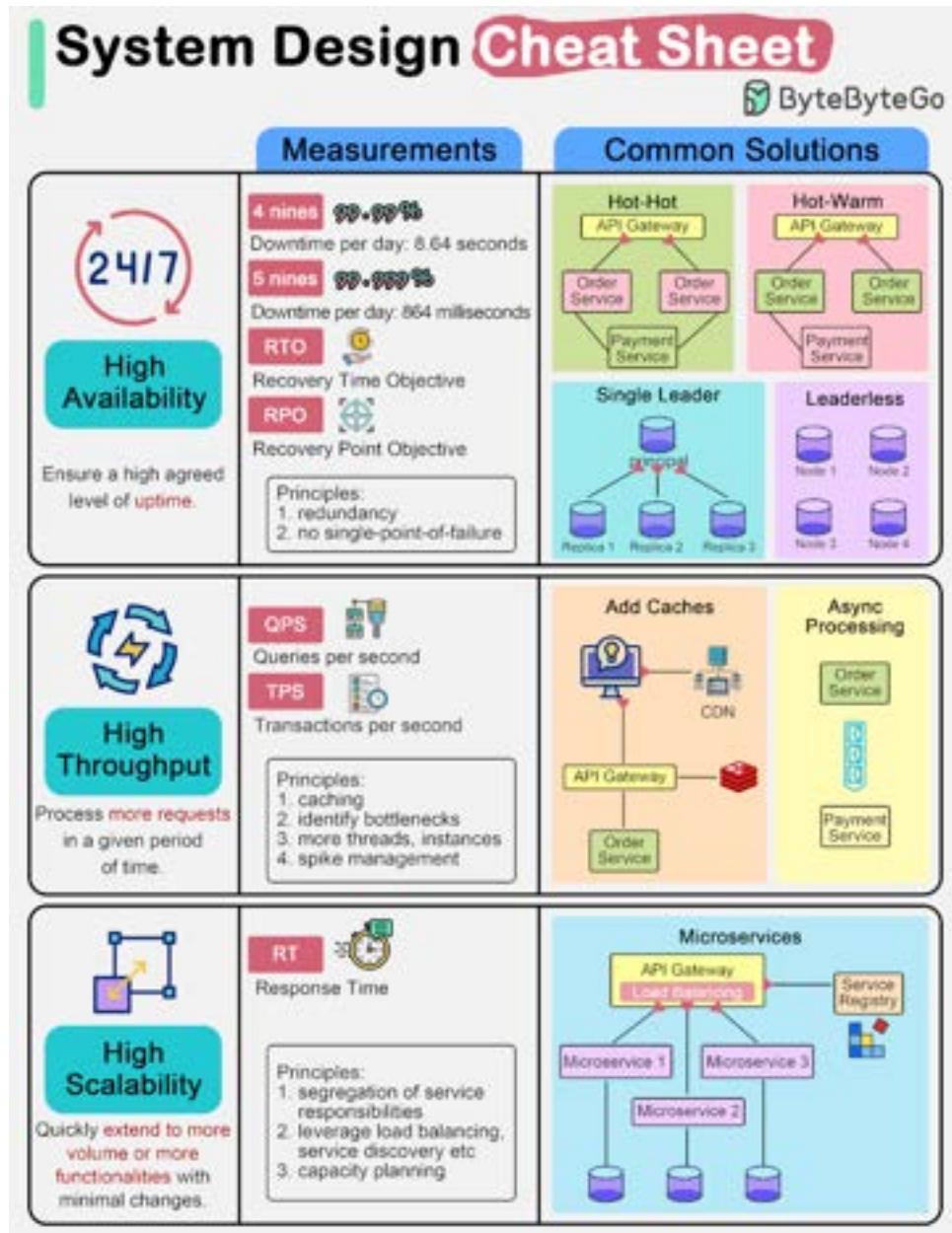
"Keep it simple, stupid!" is a design principle first noted by the U.S. Navy in 1960. It states that most systems work best if they are kept simple.

Over to you: Have you invented any acronyms in your career?

System Design cheat sheet

We are often asked to design for high availability, high scalability, and high throughput. What do they mean exactly?

The diagram below is a system design cheat sheet with common solutions.



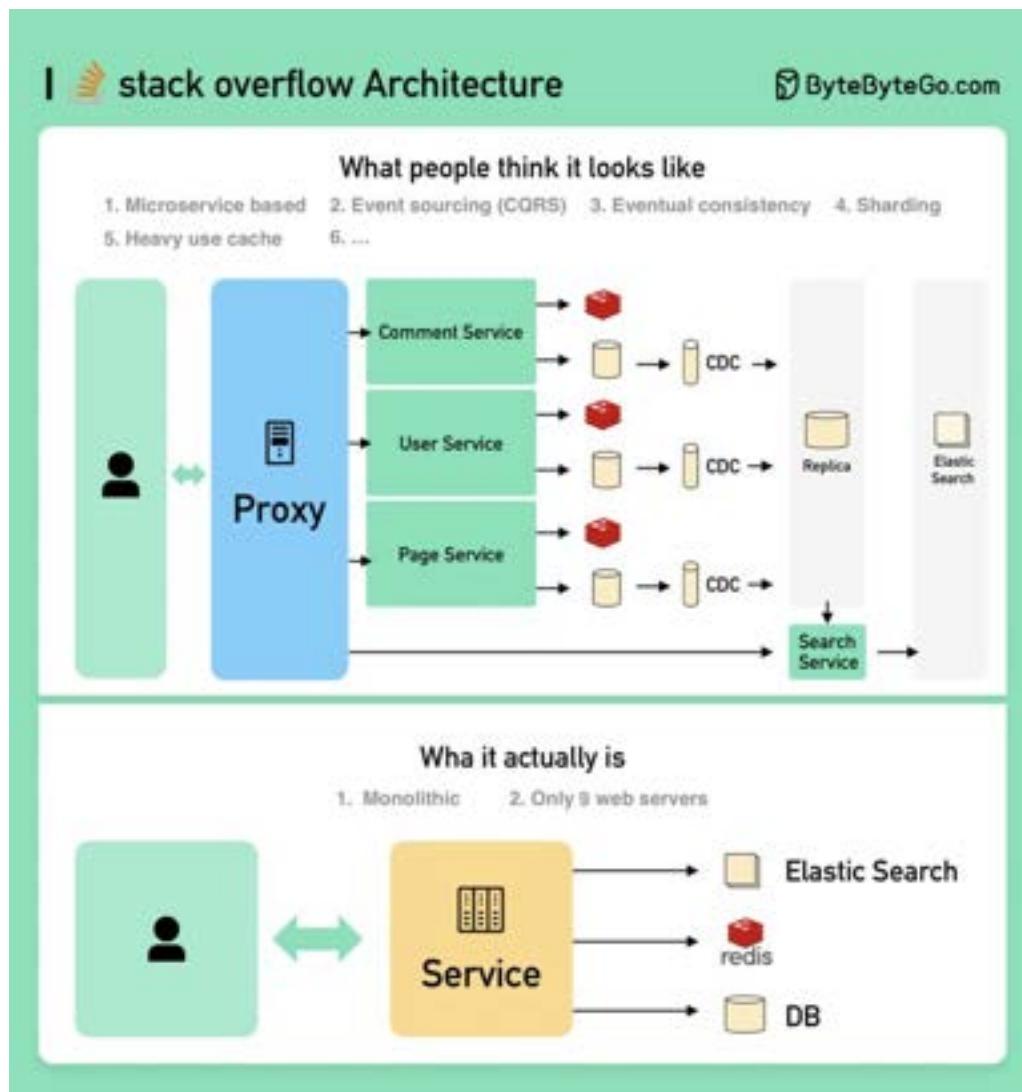
1. High Availability

This means we need to ensure a high agreed level of uptime. We often describe the design target as “3 nines” or “4 nines”. “4 nines”, 99.99% uptime, means the service can only be down 8.64 seconds per day.

To achieve high availability, we need to design redundancy in the system. There are several ways to do this:

- Hot-hot: two instances receive the same input and send the output to the downstream service. In case one side is down, the other side can immediately take over. Since both sides send output to the downstream, the downstream system needs to dedupe.
 - Hot-warm: two instances receive the same input and only the hot side sends the output to the downstream service. In case the hot side is down, the warm side takes over and starts to send output to the downstream service.
 - Single-leader cluster: one leader instance receives data from the upstream system and replicates to other replicas.
 - Leaderless cluster: there is no leader in this type of cluster. Any write will get replicated to other instances. As long as the number of write instances plus the number of read instances are larger than the total number of instances, we should get valid data.
2. High Throughput
- This means the service needs to handle a high number of requests given a period of time. Commonly used metrics are QPS (query per second) or TPS (transaction per second).
- To achieve high throughput, we often add caches to the architecture so that the request can return without hitting slower I/O devices like databases or disks. We can also increase the number of threads for computation-intensive tasks. However, adding too many threads can deteriorate the performance. We then need to identify the bottlenecks in the system and increase its throughput. Using asynchronous processing can often effectively isolate heavy-lifting components.
3. High Scalability
- This means a system can quickly and easily extend to accommodate more volume (horizontal scalability) or more functionalities (vertical scalability). Normally we watch the response time to decide if we need to scale the system.

How will you design the Stack Overflow website?



If your answer is on-premise servers and monolith (on the right), you would likely fail the interview, but that's how it is built in reality!

What people think it should look like

The interviewer is probably expecting something on the left side.

1. Microservice is used to decompose the system into small components.
2. Each service has its own database. Use cache heavily.
3. The service is sharded.
4. The services talk to each other asynchronously through message queues.
5. The service is implemented using Event Sourcing with CQRS.
6. Showing off knowledge in distributed systems such as eventual consistency, CAP theorem, etc.

What it actually is

Stack Overflow serves all the traffic with only 9 on-premise web servers, and it's a monolith! It has its own servers and does not run on the cloud.

This is contrary to all our popular beliefs these days.

Over to you: what is good architecture, the one that looks fancy during the interview or the one that works in reality?

A nice cheat sheet of different cloud services

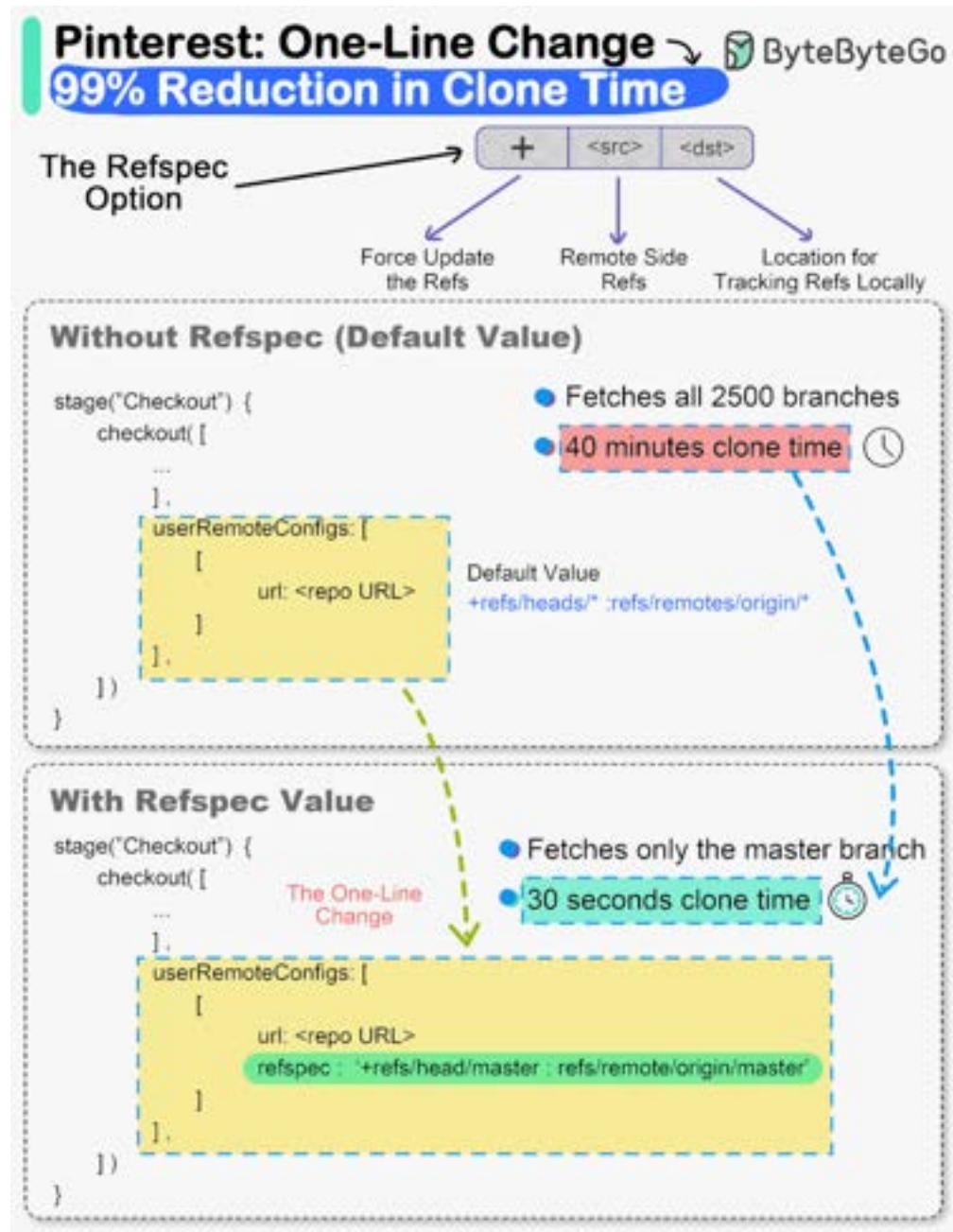


What's included?

- AWS, Azure, Google Cloud, Oracle Cloud, Alibaba Cloud
- Cloud servers
- Databases

- Message queues and streaming platforms
- Load balancing, DNS routing software
- Security
- Monitoring

The one-line change that reduced clone times by a whopping 99%, says Pinterest



While it may sound cliché, small changes can definitely create a big impact.

The Engineering Productivity team at Pinterest witnessed this first-hand.

They made a small change in the Jenkins build pipeline of their monorepo codebase called Pinboard.

And it brought down clone times from 40 minutes to a staggering 30 seconds.

For reference, Pinboard is the oldest and largest monorepo at Pinterest. Some facts about it:

- 350K commits
- 20 GB in size when cloned fully
- 60K git pulls on every business day

Cloning monorepos having a lot of code and history is time consuming. This was exactly what was happening with Pinboard.

The build pipeline (written in Groovy) started with a “Checkout” stage where the repository was cloned for the build and test steps.

The clone options were set to shallow clone, no fetching of tags and only fetching the last 50 commits.

But it missed a vital piece of optimization.

The Checkout step didn’t use the Git refspec option.

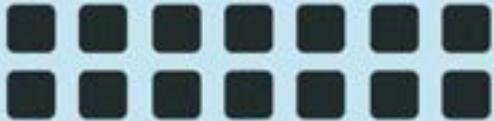
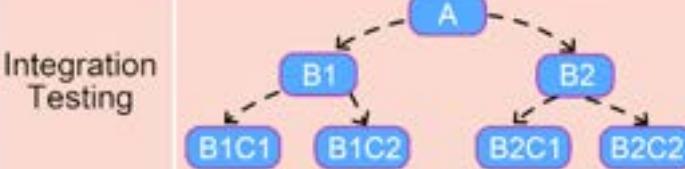
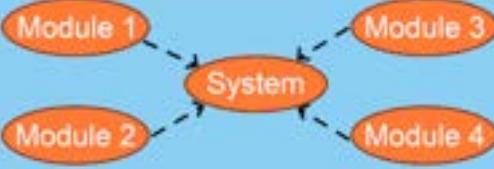
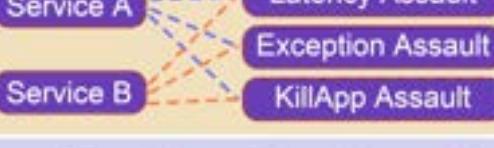
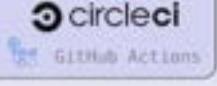
This meant that Git was effectively fetching all refs for every build. For the Pinboard monorepo, it meant fetching more than 2500 branches.

So - what was the fix?

The team simply added the refspec option and specified which ref they cared about. It was the “master” branch in this case.

This single change allowed Git clone to deal with only one branch and significantly reduced the overall build time of the monorepo.

Best ways to test system functionality

Best Ways To Test System Functionality		
Process	Illustration	Tools
Unit Testing		
Integration Testing		
System Testing		
Load Testing		
Error Testing		
Test Automation		

Testing system functionality is a crucial step in software development and engineering processes.

It ensures that a system or software application performs as expected, meets user requirements, and operates reliably.

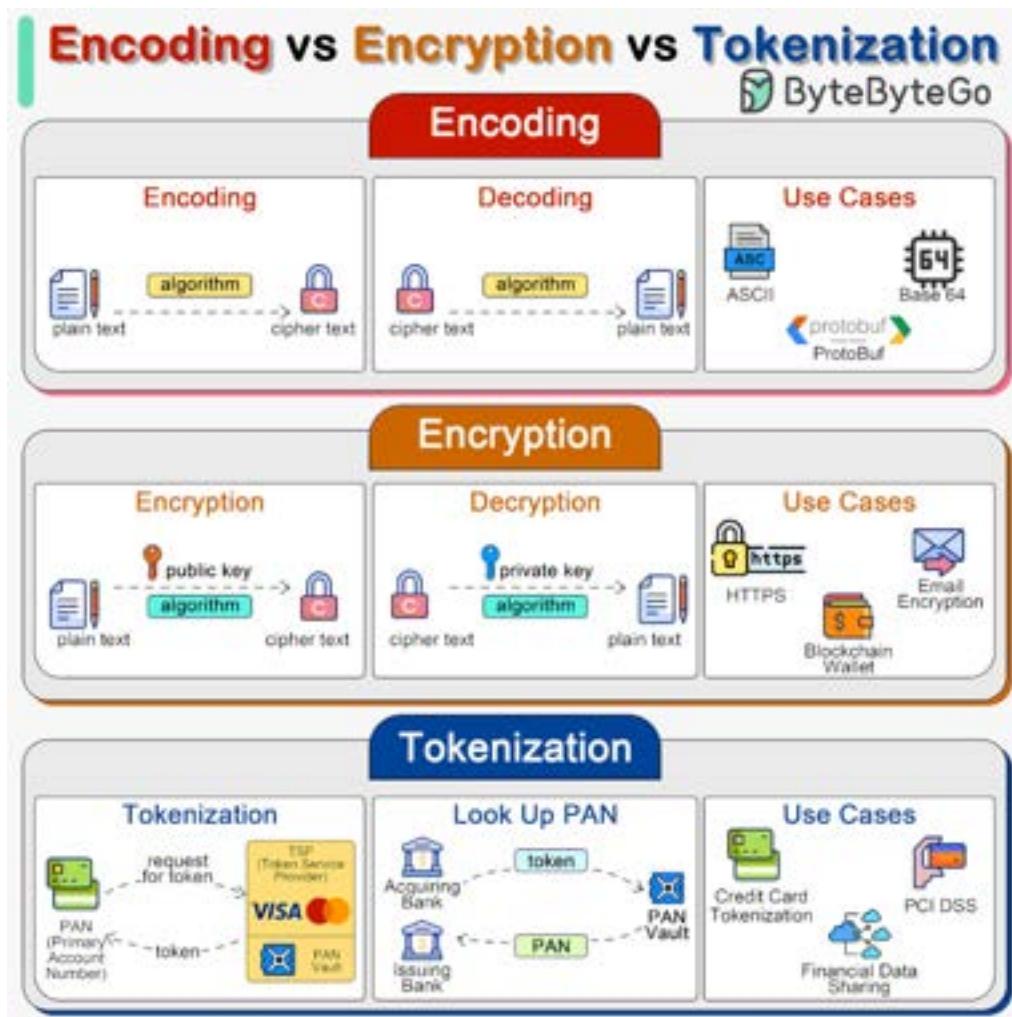
Here we delve into the best ways:

1. Unit Testing: Ensures individual code components work correctly in isolation.
2. Integration Testing: Verifies that different system parts function seamlessly together.

3. System Testing: Assesses the entire system's compliance with user requirements and performance.
4. Load Testing: Tests a system's ability to handle high workloads and identifies performance issues.
5. Error Testing: Evaluates how the software handles invalid inputs and error conditions.
6. Test Automation: Automates test case execution for efficiency, repeatability, and error reduction.

Over to you: How do you approach testing system functionality in your software development or engineering projects?

Encoding vs Encryption vs Tokenization



Encoding, encryption, and tokenization are three distinct processes that handle data in different ways for various purposes, including data transmission, security, and compliance.

In system designs, we need to select the right approach for handling sensitive information.

♦ Encoding

Encoding converts data into a different format using a scheme that can be easily reversed. Examples include Base64 encoding, which encodes binary data into ASCII characters, making it easier to transmit data over media that are designed to deal with textual data.

Encoding is not meant for securing data. The encoded data can be easily decoded using the same scheme without the need for a key.

♦ Encryption

Encryption involves complex algorithms that use keys for transforming data. Encryption can be symmetric (using the same key for encryption and decryption) or asymmetric (using a public key for encryption and a private key for decryption).

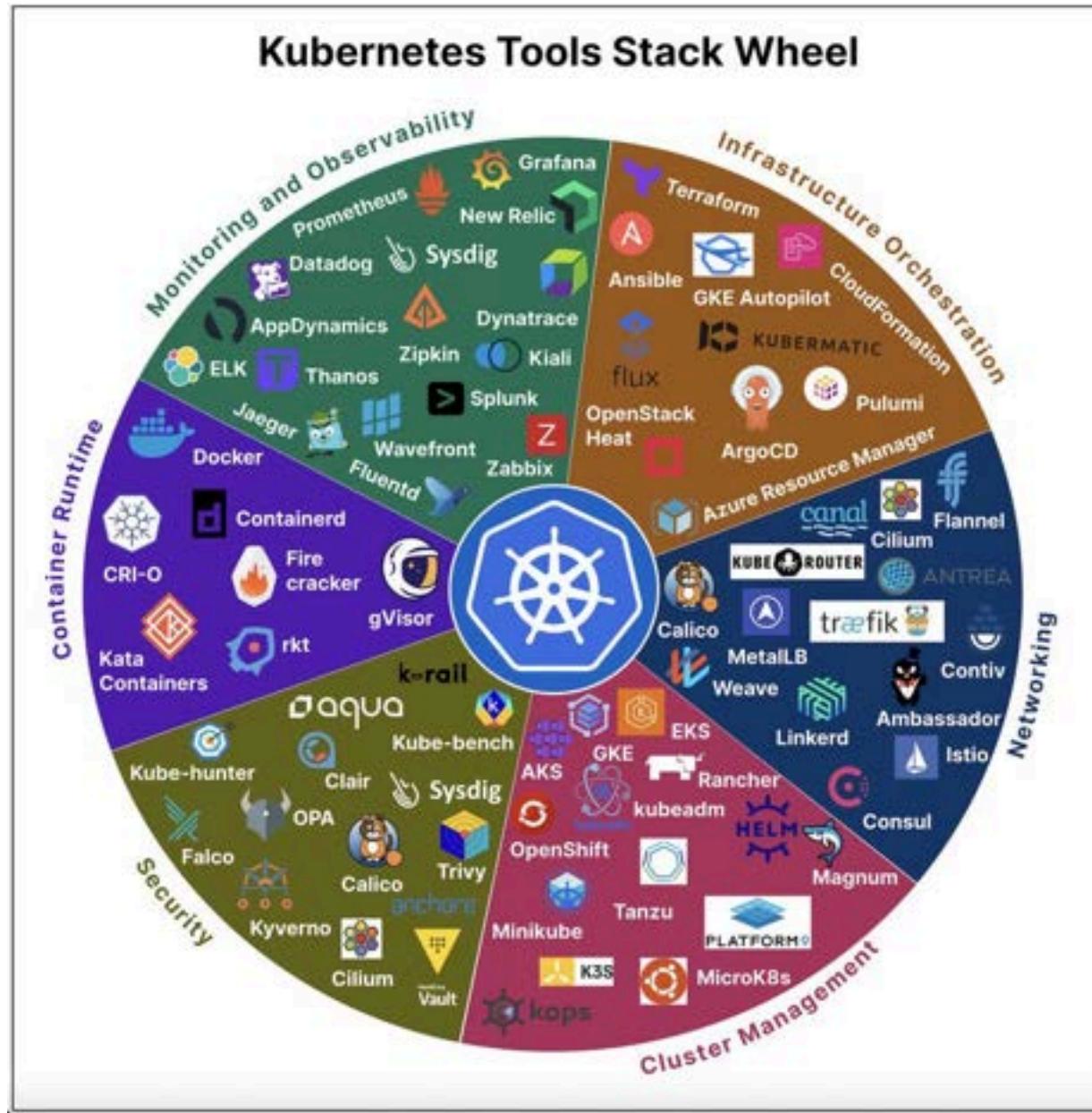
Encryption is designed to protect data confidentiality by transforming readable data (plaintext) into an unreadable format (ciphertext) using an algorithm and a secret key. Only those with the correct key can decrypt and access the original data.

◆ Tokenization

Tokenization is the process of substituting sensitive data with non-sensitive placeholders called tokens. The mapping between the original data and the token is stored securely in a token vault. These tokens can be used in various systems and processes without exposing the original data, reducing the risk of data breaches.

Tokenization is often used for protecting credit card information, personal identification numbers, and other sensitive data. Tokenization is highly secure, as the tokens do not contain any part of the original data and thus cannot be reverse-engineered to reveal the original data. It is particularly useful for compliance with regulations like PCI DSS.

Kubernetes Tools Stack Wheel



Kubernetes tools continually evolve, offering enhanced capabilities and simplifying container orchestration. The innumerable choice of tools speaks about the vastness and the scope of this dynamic ecosystem, catering to diverse needs in the world of containerization.

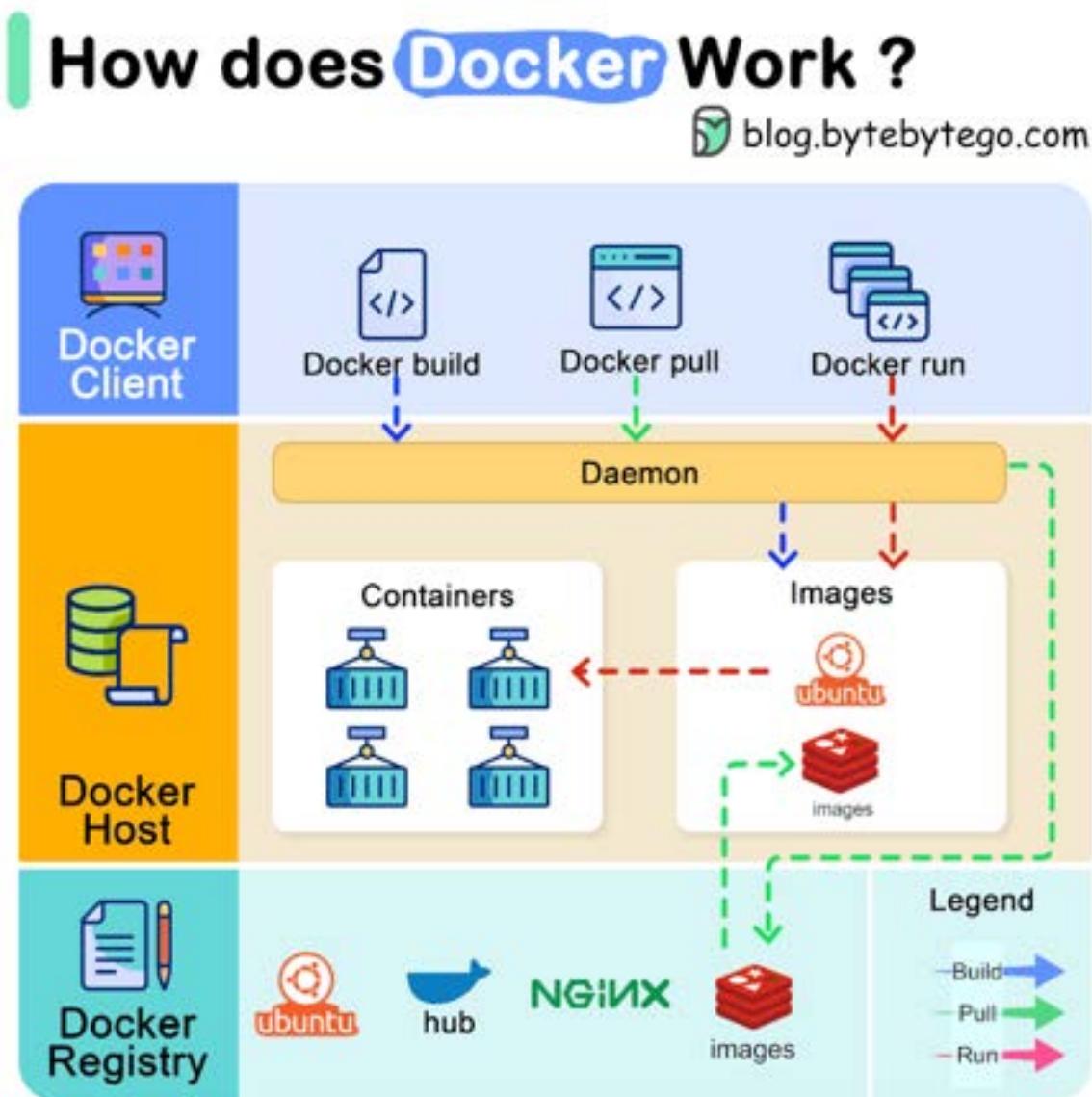
In fact, getting to know about the existing tools themselves can be a significant endeavor. With new tools and updates being introduced regularly, staying informed about their features, compatibility, and best practices becomes essential for Kubernetes practitioners, ensuring they can make informed decisions and adapt to the ever-changing landscape effectively.

This tool stack streamlines the decision-making process and keeps up with that evolution, ultimately helping you to choose the right combination of tools for your use cases.

Over to you: I am sure there would be a few awesome tools that are missing here. Which one would you like to add?

How does Docker work?

The diagram below shows the architecture of Docker and how it works when we run “docker build”, “docker pull” and “docker run”.



There are 3 components in Docker architecture:

- Docker client

The docker client talks to the Docker daemon.

- Docker host

The Docker daemon listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes.

- ◆ Docker registry

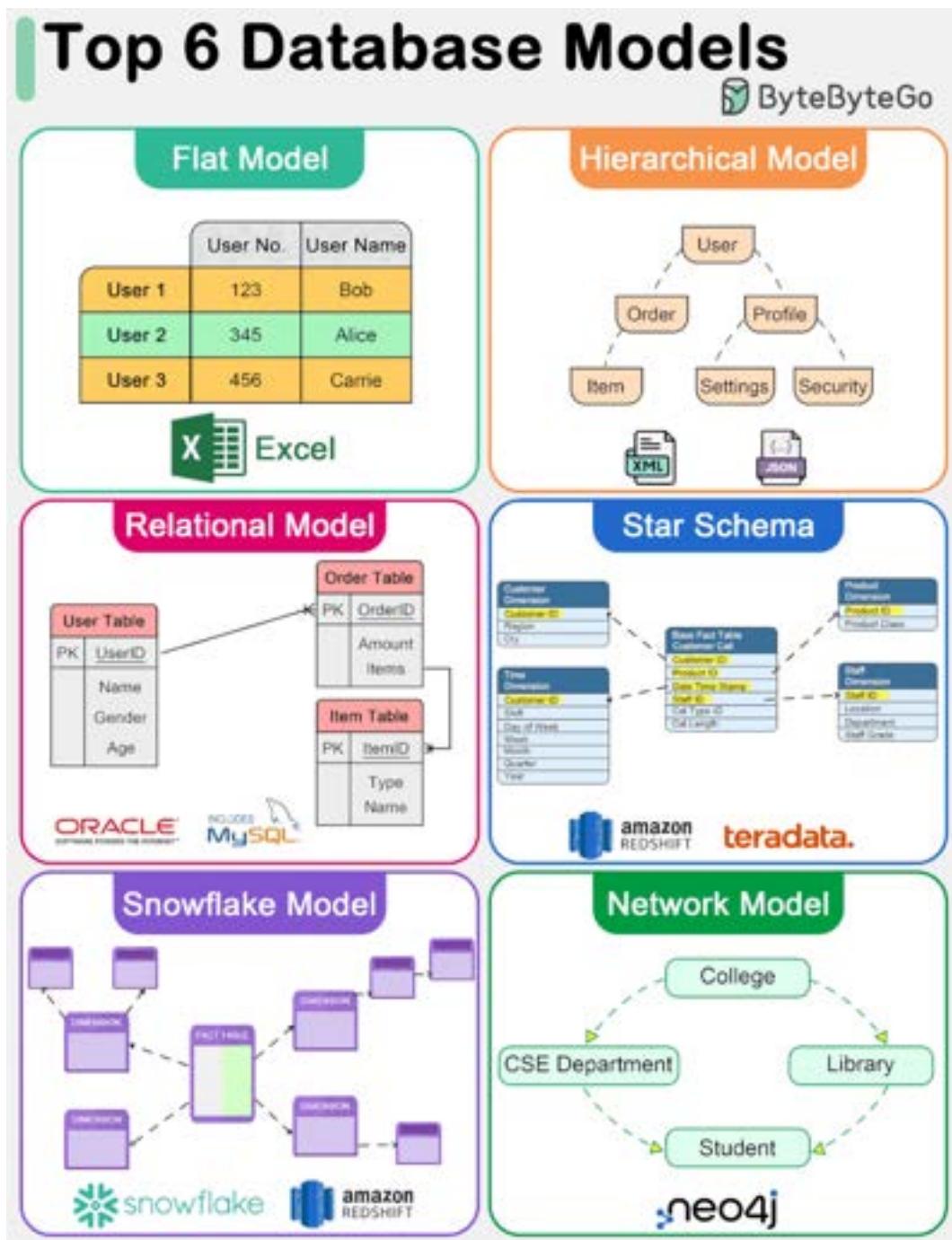
A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use.

Let's take the "docker run" command as an example.

1. Docker pulls the image from the registry.
2. Docker creates a new container.
3. Docker allocates a read-write filesystem to the container.
4. Docker creates a network interface to connect the container to the default network.
5. Docker starts the container.

Top 6 Database Models

The diagram below shows top 6 data models.



Flat Model

The flat data model is one of the simplest forms of database models. It organizes data into a single table where each row represents a record and each column represents an attribute. This model is

similar to a spreadsheet and is straightforward to understand and implement. However, it lacks the ability to efficiently handle complex relationships between data entities.

◆ Hierarchical Model

The hierarchical data model organizes data into a tree-like structure, where each record has a single parent but can have multiple children. This model is efficient for scenarios with a clear "parent-child" relationship among data entities. However, it struggles with many-to-many relationships and can become complex and rigid.

◆ Relational Model

Introduced by E.F. Codd in 1970, the relational model represents data in tables (relations), consisting of rows (tuples) and columns (attributes). It supports data integrity and avoids redundancy through the use of keys and normalization. The relational model's strength lies in its flexibility and the simplicity of its query language, SQL (Structured Query Language), making it the most widely used data model for traditional database systems. It efficiently handles many-to-many relationships and supports complex queries and transactions.

◆ Star Schema

The star schema is a specialized data model used in data warehousing for OLAP (Online Analytical Processing) applications. It features a central fact table that contains measurable, quantitative data, surrounded by dimension tables that contain descriptive attributes related to the fact data. This model is optimized for query performance in analytical applications, offering simplicity and fast data retrieval by minimizing the number of joins needed for queries.

◆ Snowflake Model

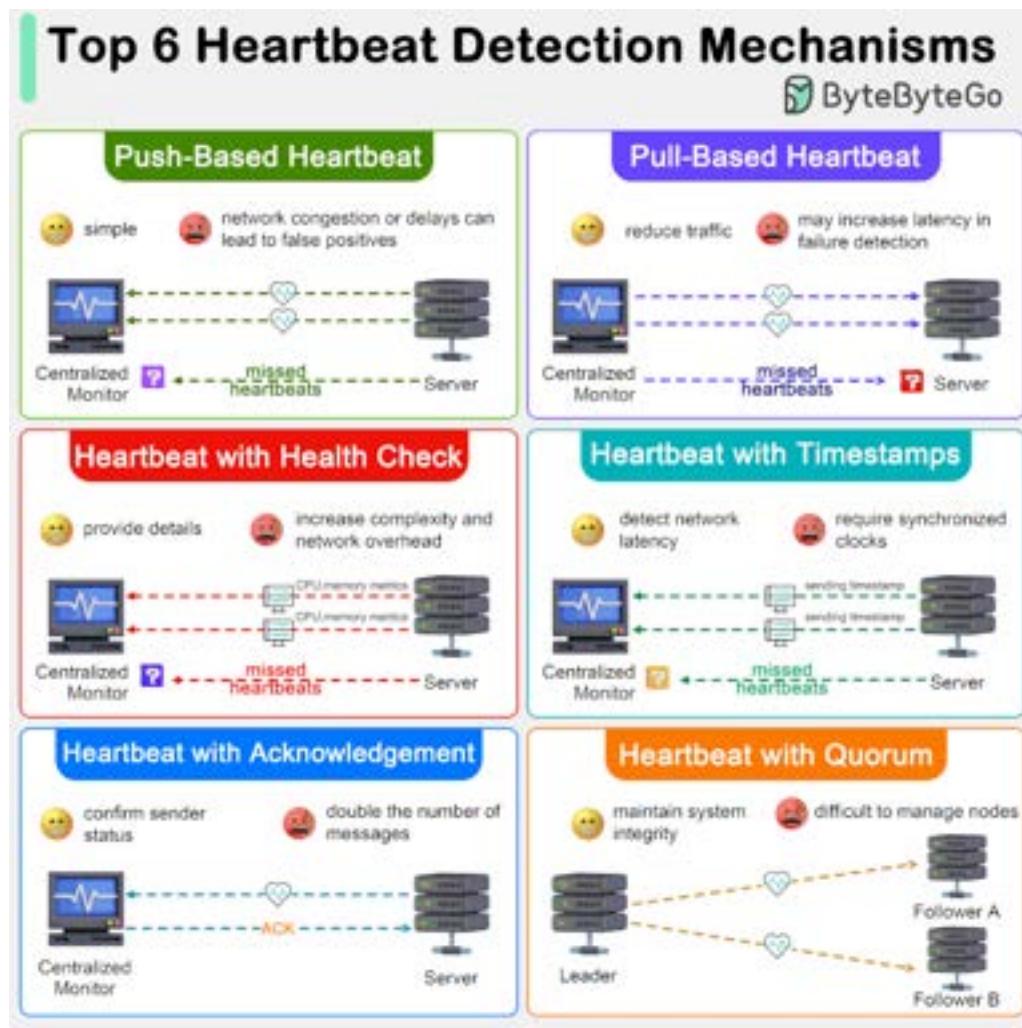
The snowflake model is a variation of the star schema where the dimension tables are normalized into multiple related tables, reducing redundancy and improving data integrity. This results in a structure that resembles a snowflake. While the snowflake model can lead to more complex queries due to the increased number of joins, it offers benefits in terms of storage efficiency and can be advantageous in scenarios where dimension tables are large or frequently updated.

◆ Network Model

The network data model allows each record to have multiple parents and children, forming a graph structure that can represent complex relationships between data entities. This model overcomes some of the hierarchical model's limitations by efficiently handling many-to-many relationships.

How do we detect node failures in distributed systems?

The diagram below shows top 6 Heartbeat Detection Mechanisms.



Heartbeat mechanisms are crucial in distributed systems for monitoring the health and status of various components. Here are several types of heartbeat detection mechanisms commonly used in distributed systems:

◆ Push-Based Heartbeat

The most basic form of heartbeat involves a periodic signal sent from one node to another or to a monitoring service. If the heartbeat signals stop arriving within a specified interval, the system assumes that the node has failed. This is simple to implement, but network congestion can lead to false positives.

◆ Pull-Based Heartbeat

Instead of nodes sending heartbeats actively, a central monitor might periodically "pull" status information from nodes. It reduces network traffic but might increase latency in failure detection.

- ◆ Heartbeat with Health Check

This includes diagnostic information about the node's health in the heartbeat signal. This information can include CPU usage, memory usage, or application-specific metrics. It Provides more detailed information about the node, allowing for more nuanced decision-making. However, it Increases complexity and potential for larger network overhead.

- ◆ Heartbeat with Timestamps

Heartbeats that include timestamps can help the receiving node or service determine not just if a node is alive, but also if there are network delays affecting communication.

- ◆ Heartbeat with Acknowledgement

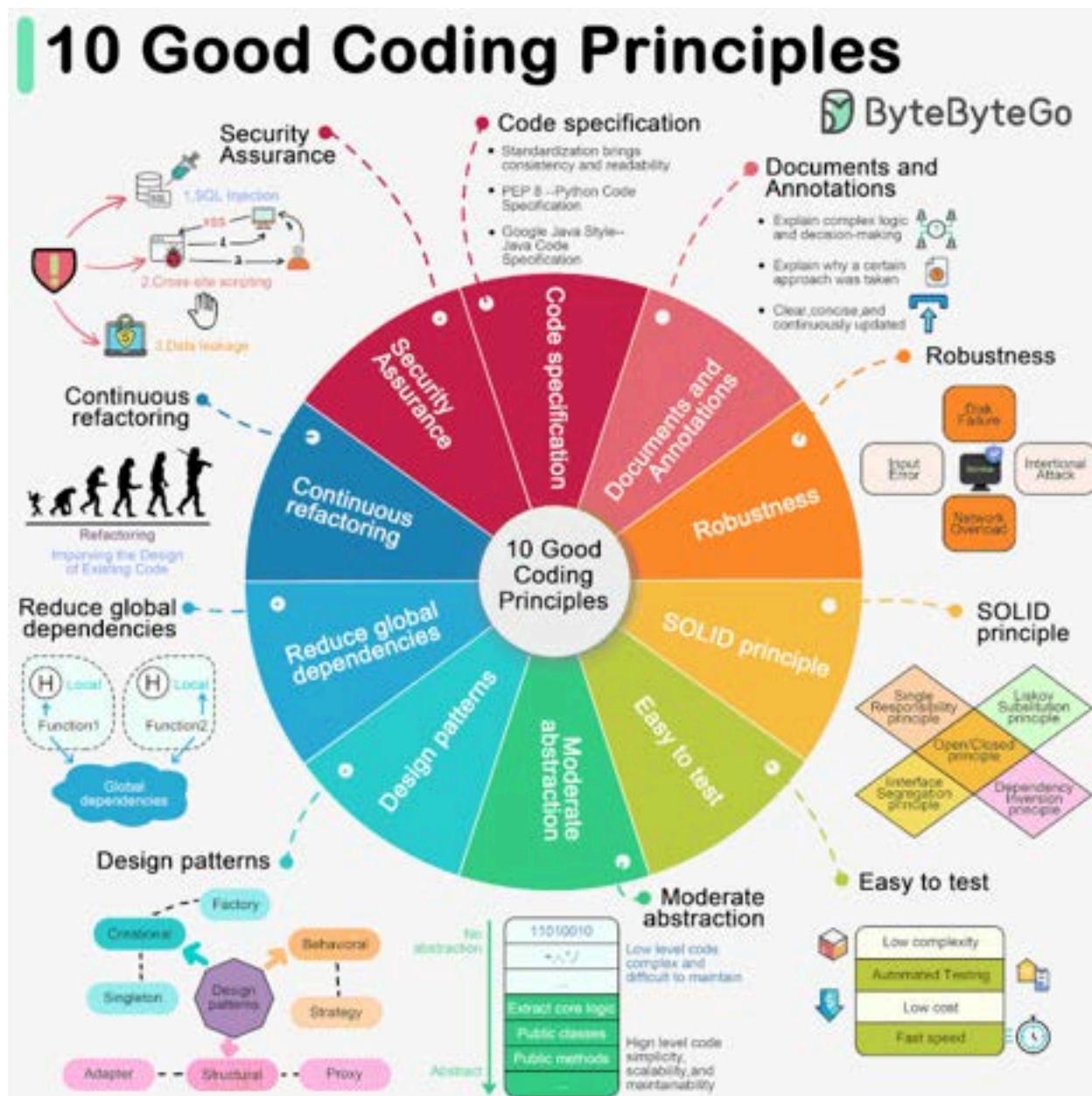
The receiver of the heartbeat message must send back an acknowledgment in this model. This ensures that not only is the sender alive, but the network path between the sender and receiver is also functional.

- ◆ Heartbeat with Quorum

In some distributed systems, especially those involving consensus protocols like Paxos or Raft, the concept of a quorum (a majority of nodes) is used. Heartbeats might be used to establish or maintain a quorum, ensuring that a sufficient number of nodes are operational for the system to make decisions. This brings complexity in implementation and managing quorum changes as nodes join or leave the system.

10 Good Coding Principles to improve code quality

Software development requires good system designs and coding standards. We list 10 good coding principles in the diagram below.



1. Follow Code Specifications

When we write code, it is important to follow the industry's well-established norms, like "PEP 8", "Google Java Style", adhering to a set of agreed-upon code specifications ensures that the quality of the code is consistent and readable.

2. Documentation and Comments

Good code should be clearly documented and commented to explain complex logic and decisions, and comments should explain why a certain approach was taken ("Why") rather than what exactly is being done ("What"). Documentation and comments should be clear, concise, and continuously updated.

3. Robustness

Good code should be able to handle a variety of unexpected situations and inputs without crashing or producing unpredictable results. Most common approach is to catch and handle exceptions.

4. Follow the SOLID principle

"Single Responsibility", "Open/Closed", "Liskov Substitution", "Interface Segregation", and "Dependency Inversion" - these five principles (SOLID for short) are the cornerstones of writing code that scales and is easy to maintain.

5. Make Testing Easy

Testability of software is particularly important. Good code should be easy to test, both by trying to reduce the complexity of each component, and by supporting automated testing to ensure that it behaves as expected.

6. Abstraction

Abstraction requires us to extract the core logic and hide the complexity, thus making the code more flexible and generic. Good code should have a moderate level of abstraction, neither over-designed nor neglecting long-term expandability and maintainability.

7. Utilize Design Patterns, but don't over-design

Design patterns can help us solve some common problems. However, every pattern has its applicable scenarios. Overusing or misusing design patterns may make your code more complex and difficult to understand.

8. Reduce Global Dependencies

We can get bogged down in dependencies and confusing state management if we use global variables and instances. Good code should rely on localized state and parameter passing. Functions should be side-effect free.

9. Continuous Refactoring

Good code is maintainable and extensible. Continuous refactoring reduces technical debt by identifying and fixing problems as early as possible.

10. Security is a Top Priority

Good code should avoid common security vulnerabilities.

Over to you: which one do you prefer, and with which one do you disagree?

15 Open-Source Projects That Changed the World



To come up with the list, we tried to look at the overall impact these projects have created on the industry and related technologies. Also, we've focused on projects that have led to a big change in the day-to-day lives of many software developers across the world.

Web Development

- Node.js: The cross-platform server-side Javascript runtime that brought JS to server-side development
- React: The library that became the foundation of many web development frameworks.

- Apache HTTP Server: The highly versatile web server loved by enterprises and startups alike. Served as inspiration for many other web servers over the years.

Data Management

- PostgreSQL: An open-source relational database management system that provided a high-quality alternative to costly systems
- Redis: The super versatile data store that can be used a cache, message broker and even general-purpose storage
- Elasticsearch: A scale solution to search, analyze and visualize large volumes of data

Developer Tools

- Git: Free and open-source version control tool that allows developer collaboration across the globe.
- VSCode: One of the most popular source code editors in the world
- Jupyter Notebook: The web application that lets developers share live code, equations, visualizations and narrative text.

Machine Learning & Big Data

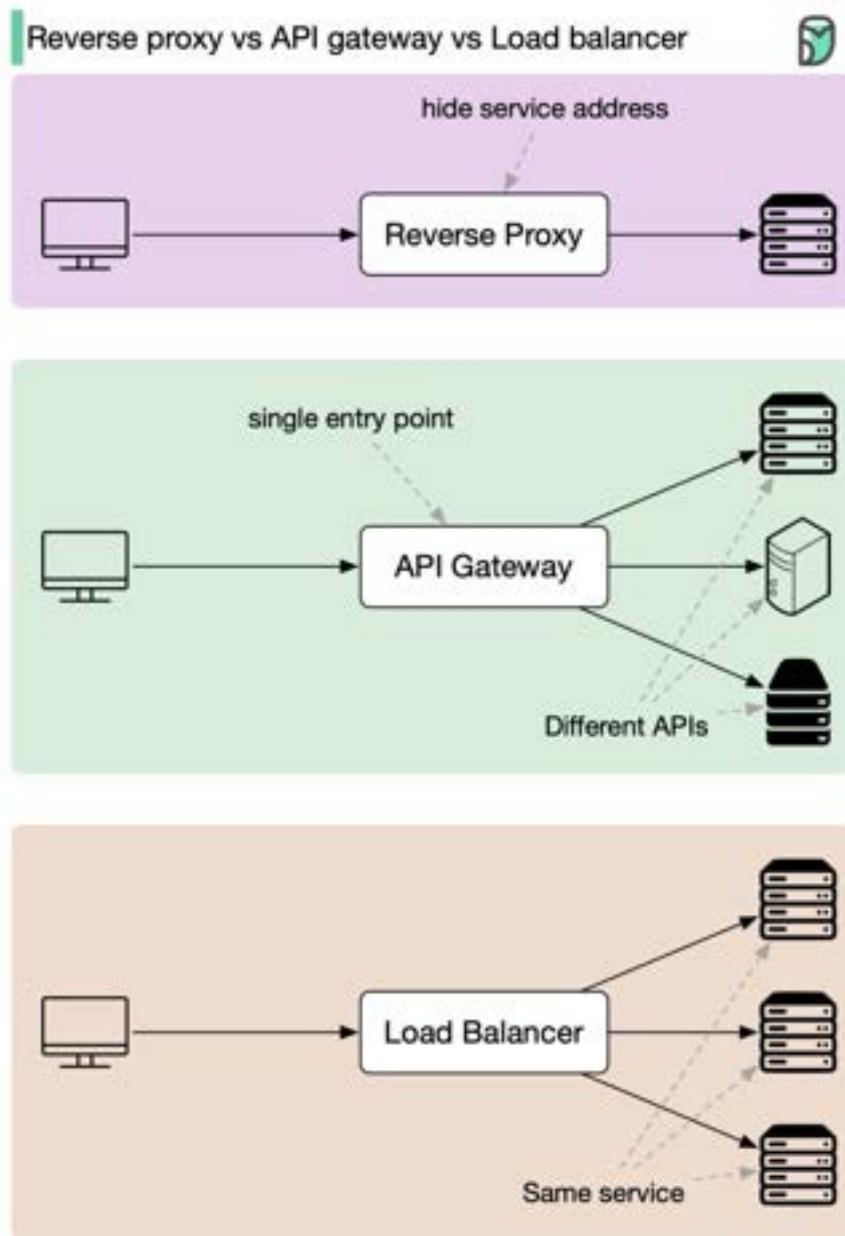
- Tensorflow: The leading choice to leverage machine learning techniques
- Apache Spark: Standard tool for big data processing and analytics platforms
- Kafka: Standard platform for building real-time data pipelines and applications.

DevOps & Containerization

- Docker: The open source solution that allows developers to package and deploy applications in a consistent and portable way.
- Kubernetes: The heart of Cloud-Native architecture and a platform to manage multiple containers
- Linux: The OS that democratized the world of software development.

Over to you: Do you agree with the list? What did we miss?

Reverse proxy vs. API gateway vs. load balancer



As modern websites and applications are like busy beehives, we use a variety of tools to manage the buzz. Here we'll explore three superheroes: Reverse Proxy, API Gateway, and Load Balancer.

- ◆ **Reverse Proxy: change identity**

- Fetching data secretly, keeping servers hidden.
- Perfect for shielding sensitive websites from cyber-attacks and prying eyes.

- ◆ **API Gateway: postman**

- Delivers requests to the right services.
- Ideal for bustling applications with numerous intercommunicating services.

◆ Load Balancer: traffic cop

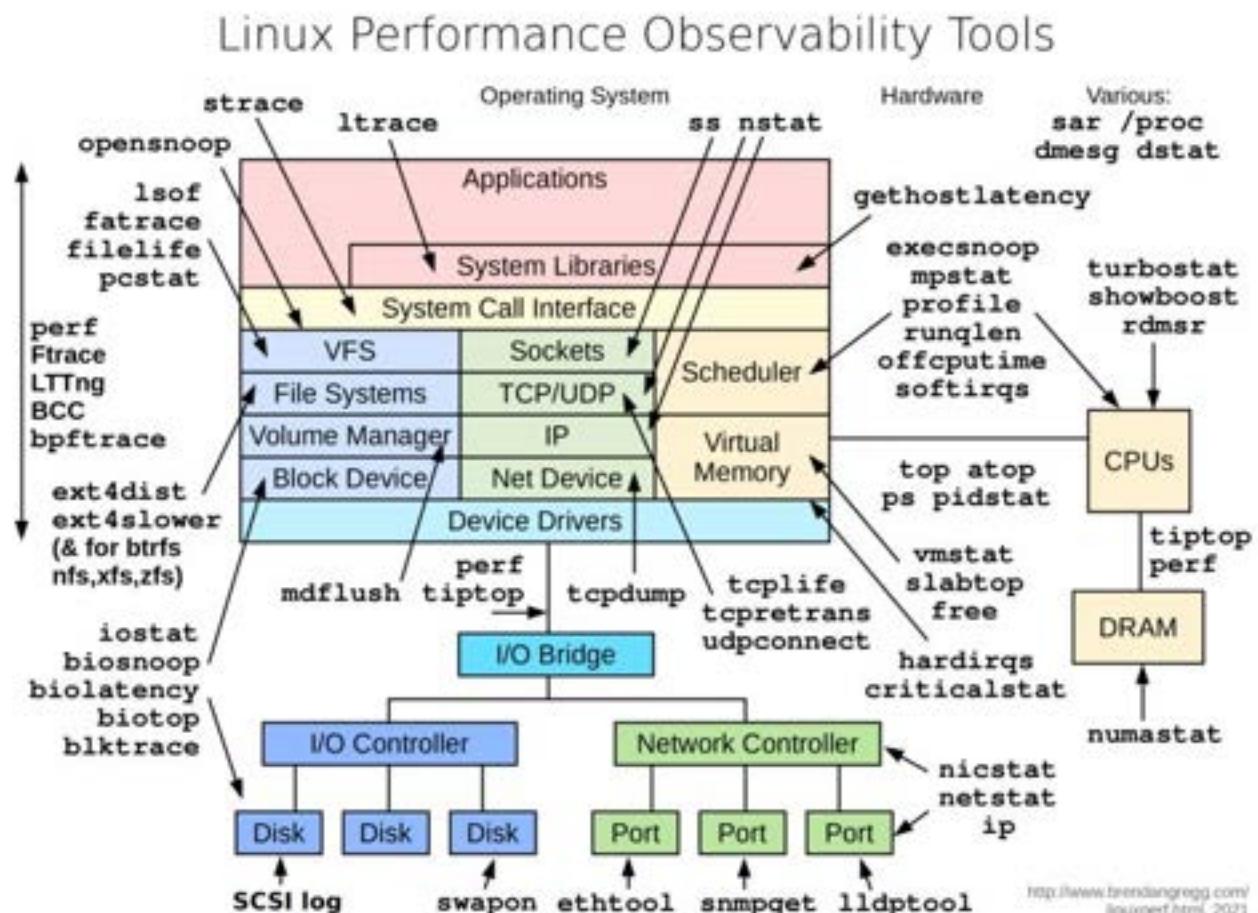
- Directs traffic evenly across servers, preventing bottlenecks
- Essential for popular websites with heavy traffic and high demand.

In a nutshell, choose a Reverse Proxy for stealth, an API Gateway for organized communications, and a Load Balancer for traffic control. Sometimes, it's wise to have all three - they make a super team that keeps your digital kingdom safe and efficient.

Linux Performance Observability Tools

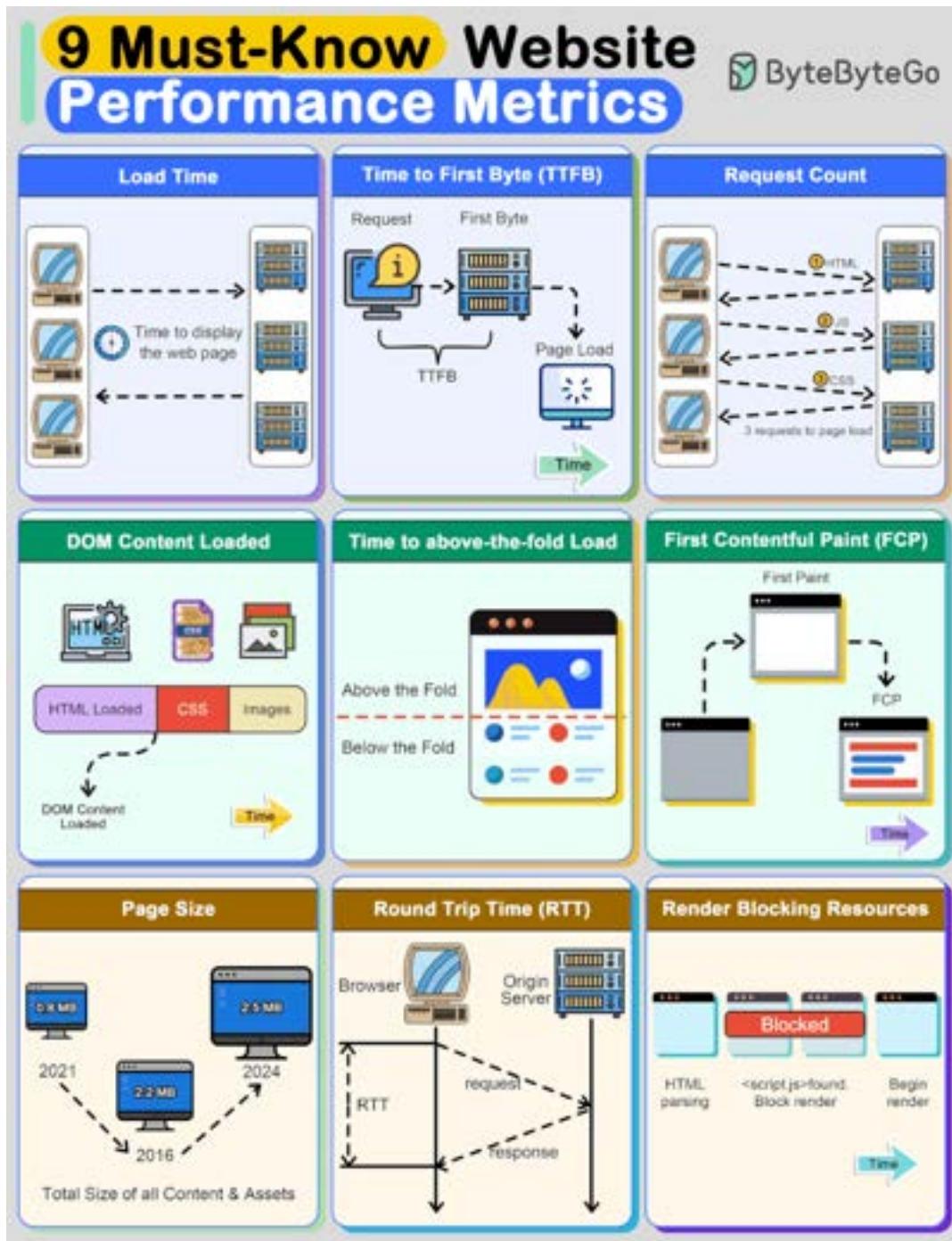
Popular interview question: how to diagnose a mysterious process that's taking too much CPU, memory, IO, etc?

The diagram below illustrates helpful tools in a Linux system.



- 'vmstat' - reports information about processes, memory, paging, block IO, traps, and CPU activity.
- 'iostat' - reports CPU and input/output statistics of the system.
- 'netstat' - displays statistical data related to IP, TCP, UDP, and ICMP protocols.
- 'lsof' - lists open files of the current system.
- 'pidstat' - monitors the utilization of system resources by all or specified processes, including CPU, memory, device IO, task switching, threads, etc.

Top 9 website performance metrics you cannot ignore



Request Count: The number of HTTP requests a browser has to make to fully load the page. The lower this count, the faster a website will feel to the user.

DOMContentLoaded (DCL): This is the time it takes for the full HTML code of a webpage to be loaded. The faster this happens, the faster users can see useful functionality. This time doesn't include loading CSS and other assets

Time to above-the-fold load: "Above the fold" is the area of a webpage that fits in a browser window without a user having to scroll down. This is the content that is first seen by the user and often dictates whether they'll continue reading the webpage.

First Contentful Paint (FCP): This is the time at which content first begins to be "painted" by the browser. It can be a text, image, or even background color.

Page Size: This is the total file size of all content and assets that appear on the page. Over the last several years, the page size of websites has been growing constantly. The bigger the size of a webpage, the longer it will take to load

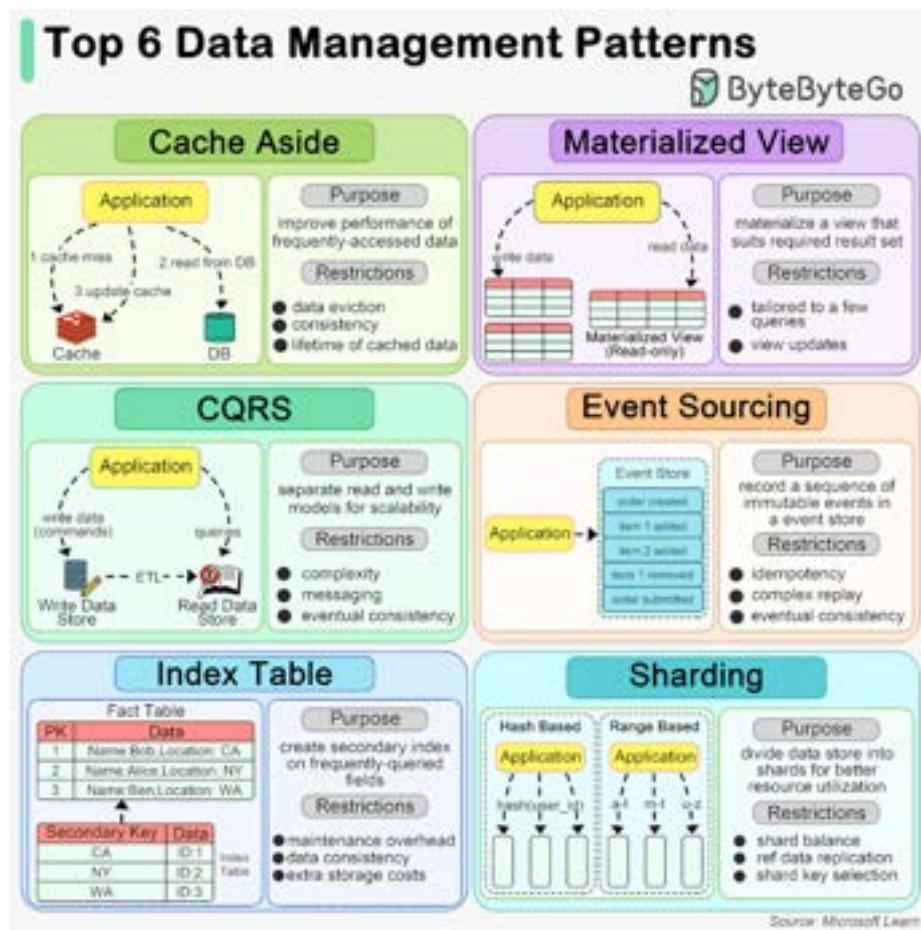
Round Trip Time (RTT): This is the amount of time a round trip takes. A round trip constitutes a request traveling from the browser to the origin server and the response from the server going to the browser. Reducing RTT is one of the key approaches to improving a website's performance.

Render Blocking Resources: Some resources block other parts of the page from being loaded. It's important to track the number of such resources. The more render-blocking resources a webpage has, the greater the delay for the browser to load the page.

Over to you - What other website performance metrics do you track?

How do we manage data?

Here are the top 6 data management patterns.



Cache Aside

When an application needs to access data, it first checks the cache. If the data is not present (a cache miss), it fetches the data from the data store, stores it in the cache, and then returns the data to the user. This pattern is particularly useful for scenarios where data is read frequently but updated less often.

Materialized View

A Materialized View is a database object that contains the results of a query. It is physically stored, meaning the data is actually computed and stored on disk, as opposed to being dynamically generated upon each request. This can significantly speed up query times for complex calculations or aggregations that would otherwise need to be computed on the fly. Materialized views are especially beneficial in data warehousing and business intelligence scenarios where query performance is critical.

CQRS

CQRS is an architectural pattern that separates the models for reading and writing data. This means that the data structures used for querying data (reads) are separated from the structures used for updating data (writes). This separation allows for optimization of each operation independently, improving performance, scalability, and security. CQRS can be particularly useful in complex systems where the read and write operations have very different requirements.

- ◆ Event Sourcing

Event Sourcing is a pattern where changes to the application state are stored as a sequence of events. Instead of storing just the current state of data in a domain, Event Sourcing stores a log of all the changes (events) that have occurred over time. This allows the application to reconstruct past states and provides an audit trail of changes. Event Sourcing is beneficial in scenarios requiring complex business transactions, auditability, and the ability to rollback or replay events.

- ◆ Index Table

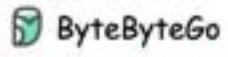
The Index Table pattern involves creating additional tables in a database that are optimized for specific query operations. These tables act as secondary indexes and are designed to speed up the retrieval of data without requiring a full scan of the primary data store. Index tables are particularly useful in scenarios with large datasets and where certain queries are performed frequently.

- ◆ Sharding

Sharding is a data partitioning pattern where data is divided into smaller, more manageable pieces, or "shards", each of which can be stored on different database servers. This pattern is used to distribute the data across multiple machines to improve scalability and performance. Sharding is particularly effective in high-volume applications, as it allows for horizontal scaling, spreading the load across multiple servers to handle more users and transactions.

Comparing Different API Clients: Postman vs. Insomnia vs. ReadyAPI vs. Thunder Client vs. Hoppscotch

API Clients Comparison Table



Criteria	Postman	Insomnia	ReadyAPI	Thunderclient	Hoppscotch
Protocols	REST, SOAP, GraphQL, gRPC, WebSocket, MQTT	REST, SOAP, GraphQL, gRPC, WebSocket	REST, SOAP, GraphQL, gRPC, WebSocket, MQTT	Focused on RESTful APIs	REST, GraphQL, WebSocket, MQTT
User Interface (UI)	Modern & Intuitive Design	Modern & Intuitive Design	Simple & Minimalistic Design	Simple & Minimalistic Design	Simple & Minimalistic Design
Features	Comprehensive enterprise-level development	Extensive features and growing	Limited; suitable for small projects	Limited; suitable for small projects	Limited; suitable for small projects
VS Code Support	Yes	No	Yes	Yes	Yes
Performance	Fast & Efficient	Fast & Efficient	Lightweight & Responsive	Lightweight & Responsive	Lightweight & Responsive
Team Collaboration	Workspaces	Workspaces	Workspaces	No	Workspaces
Third-party Integration	Github, Slack, CircleCI, AWS API Gateway, New Relic	Github, Jenkins, CircleCI, Travis CI	Github, Slack, Jenkins, JIRA	Github	Github
Data-Driven Testing	Yes	Yes	Yes	Limited: VS Code	No
Security Testing	Yes	No	Yes	No	No
Mock Servers	Yes	No	Yes	No built-in mock server	No built-in mock server
Version Control	Available in paid plans	Available in paid plans	Available in paid plans	No	No
Monitoring & Analytics	Yes	No	Yes	Limited: VS Code	No
AI Support	Documentation, tests, scripts	Test automation	Test automation	No	No
Community & Support	Large	Active	Active	Limited: VS Code	Active
Open Source	Partially	Yes	No	No	Yes

Postman is a widely used API lifecycle platform. It emerges as a comprehensive and versatile API client suitable for enterprise-level development. Its support for a wide range of protocols, robust feature set, and strong performance make it a top choice for complex projects. With an intuitive design, collaboration features, and a large community, Postman excels in scenarios requiring extensive functionality and community support.

Insomnia is a powerful API client with extensive features and being completely open-source makes it a good choice for developers seeking flexibility and continuous growth. Insomnia is suited for those who value an open-source environment and an active community.

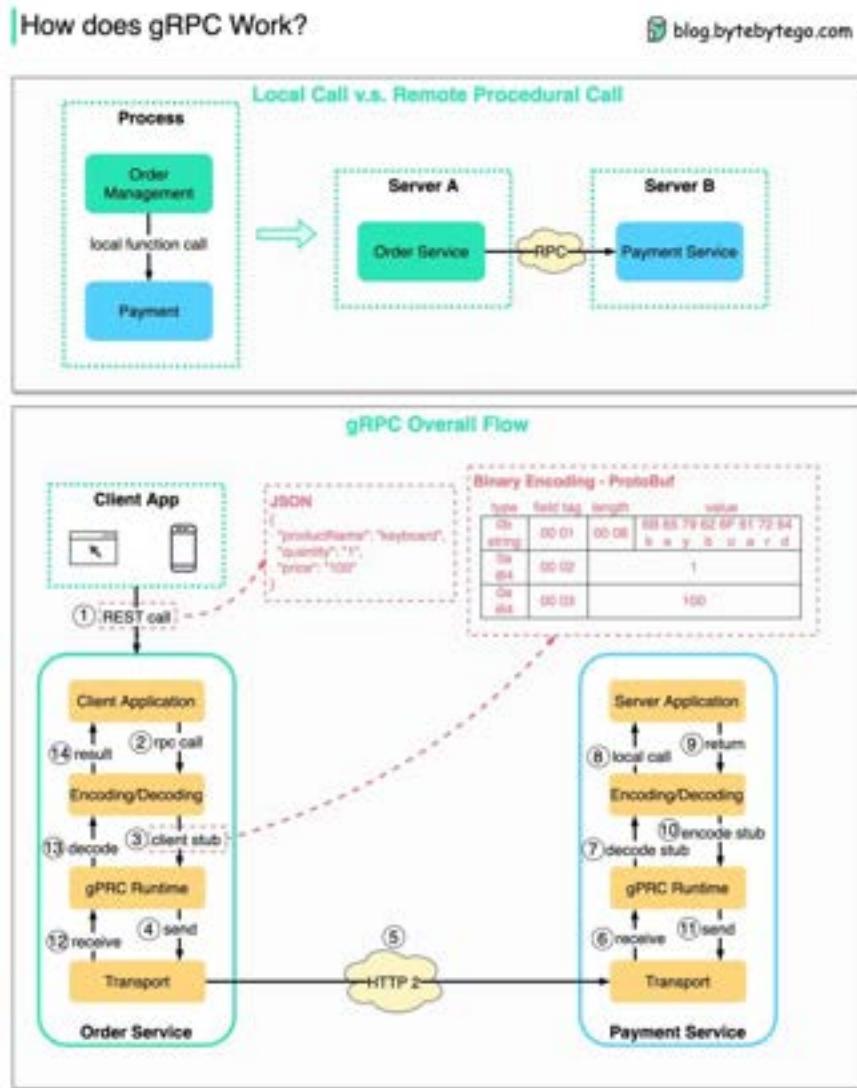
ReadyAPI, with its simplicity and focus on smaller projects, is an ideal choice for scenarios where a lightweight and responsive tool is preferred. It provides essential features, making it suitable for

projects with less complexity. However, it may not be the best fit for larger, more intricate endeavors that require extensive functionality.

ThunderClient, a VS Code plugin, is free and user-friendly, catering to developers who prefer an integrated testing environment. However, it lacks extensive features and community support, crucial for larger or complex projects, rendering it more appropriate for smaller teams with simpler requirements. Additionally, its reliance on Visual Studio Code may restrict its appeal to users who prefer alternative development environments. Experienced users accustomed to feature-rich tools may encounter a learning curve and might find ThunderClient lacking in certain functionalities.

Hopscotch, a free and open-source tool, focuses on functionality over design, offering a lightweight web version with support for various protocols. While it lacks extensive documentation and community support, it provides a cost-effective solution for developers seeking simplicity.

How does gRPC work?



RPC (Remote Procedure Call) is called “remote” because it enables communications between remote services when services are deployed to different servers under microservice architecture. From the user’s point of view, it acts like a local function call.

The diagram below illustrates the overall data flow for gRPC.

Step 1: A REST call is made from the client. The request body is usually in JSON format.

Steps 2 - 4: The order service (gRPC client) receives the REST call, transforms it, and makes an RPC call to the payment service. gPRC encodes the **client stub** into a binary format and sends it to the low-level transport layer.

Step 5: gRPC sends the packets over the network via HTTP2. Because of binary encoding and network optimizations, gRPC is said to be 5X faster than JSON.

Steps 6 - 8: The payment service (gRPC server) receives the packets from the network, decodes them, and invokes the server application.

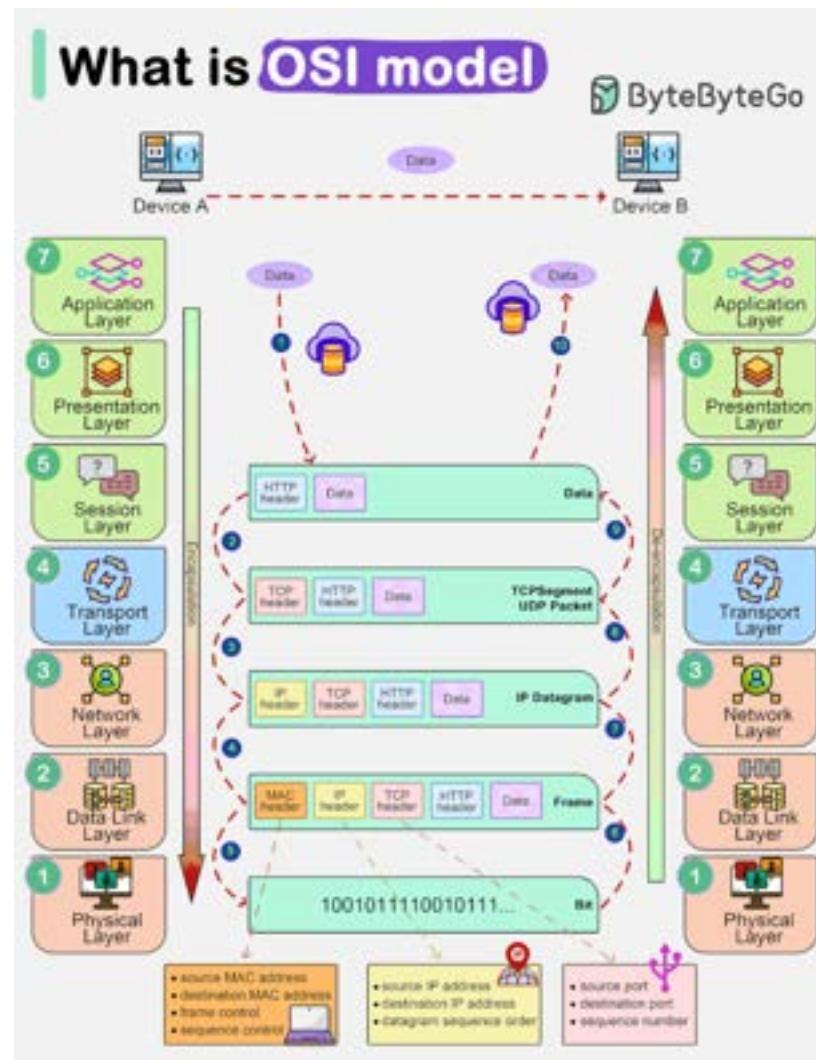
Steps 9 - 11: The result is returned from the server application, and gets encoded and sent to the transport layer.

Steps 12 - 14: The order service receives the packets, decodes them, and sends the result to the client application.

Over to you: Have you used gPRC in your project? What are some of its limitations?

How is data sent over the network? Why do we need so many layers in the OSI model?

The diagram below shows how data is encapsulated and de-encapsulated when transmitting over the network.



Step 1: When Device A sends data to Device B over the network via the HTTP protocol, it is first added an HTTP header at the application layer.

Step 2: Then a TCP or a UDP header is added to the data. It is encapsulated into TCP segments at the transport layer. The header contains the source port, destination port, and sequence number.

Step 3: The segments are then encapsulated with an IP header at the network layer. The IP header contains the source/destination IP addresses.

Step 4: The IP datagram is added a MAC header at the data link layer, with source/destination MAC addresses.

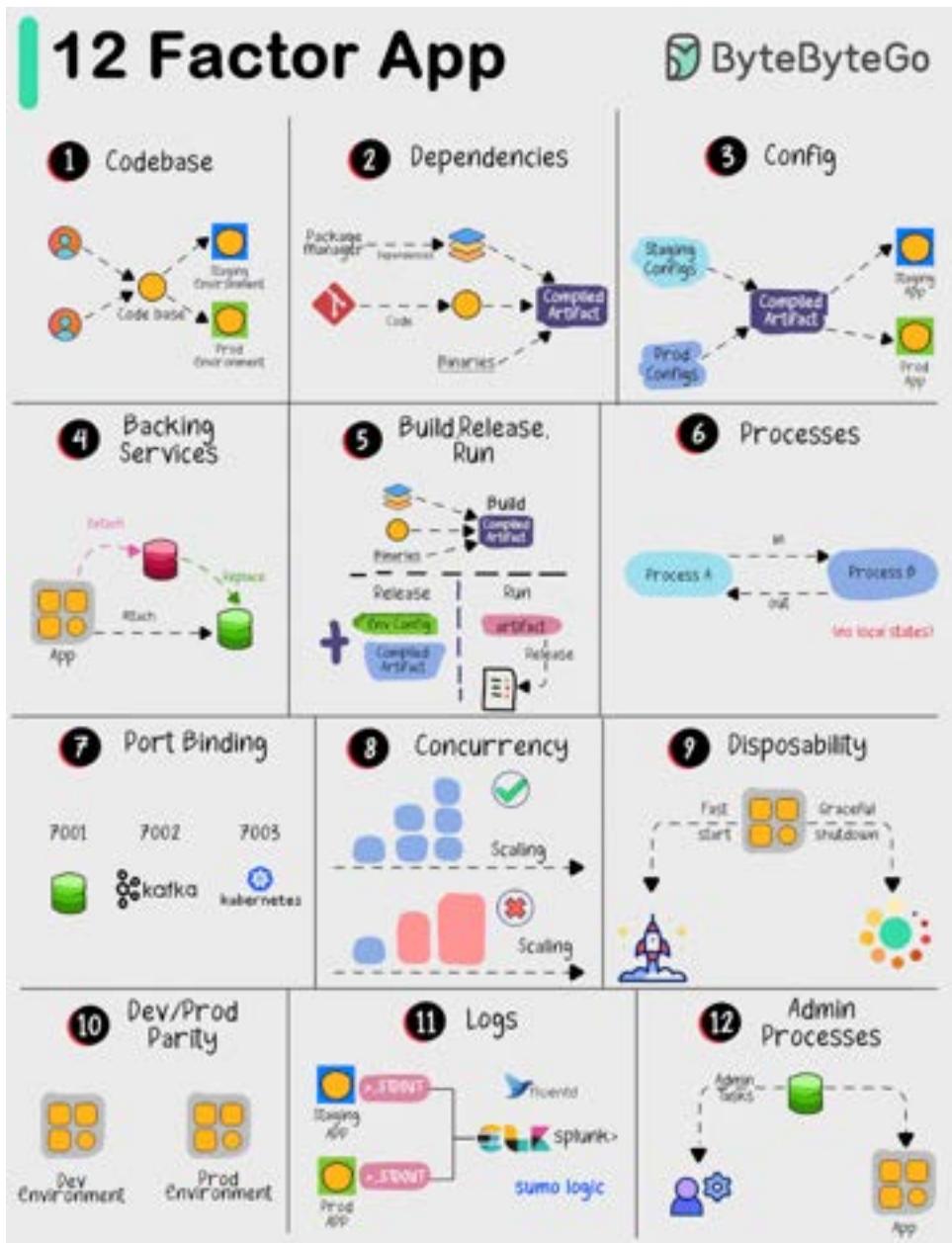
Step 5: The encapsulated frames are sent to the physical layer and sent over the network in binary bits.

Steps 6-10: When Device B receives the bits from the network, it performs the de-encapsulation process, which is a reverse processing of the encapsulation process. The headers are removed layer by layer, and eventually, Device B can read the data.

We need layers in the network model because each layer focuses on its own responsibilities. Each layer can rely on the headers for processing instructions and does not need to know the meaning of the data from the last layer.

Over to you: Do you know which layer is responsible for resending lost data?

Have you heard of the 12-Factor App?



The "12 Factor App" offers a set of best practices for building modern software applications. Following these 12 principles can help developers and teams in building reliable, scalable, and manageable applications.

Here's a brief overview of each principle:

1. Codebase:

Have one place to keep all your code, and manage it using version control like Git.

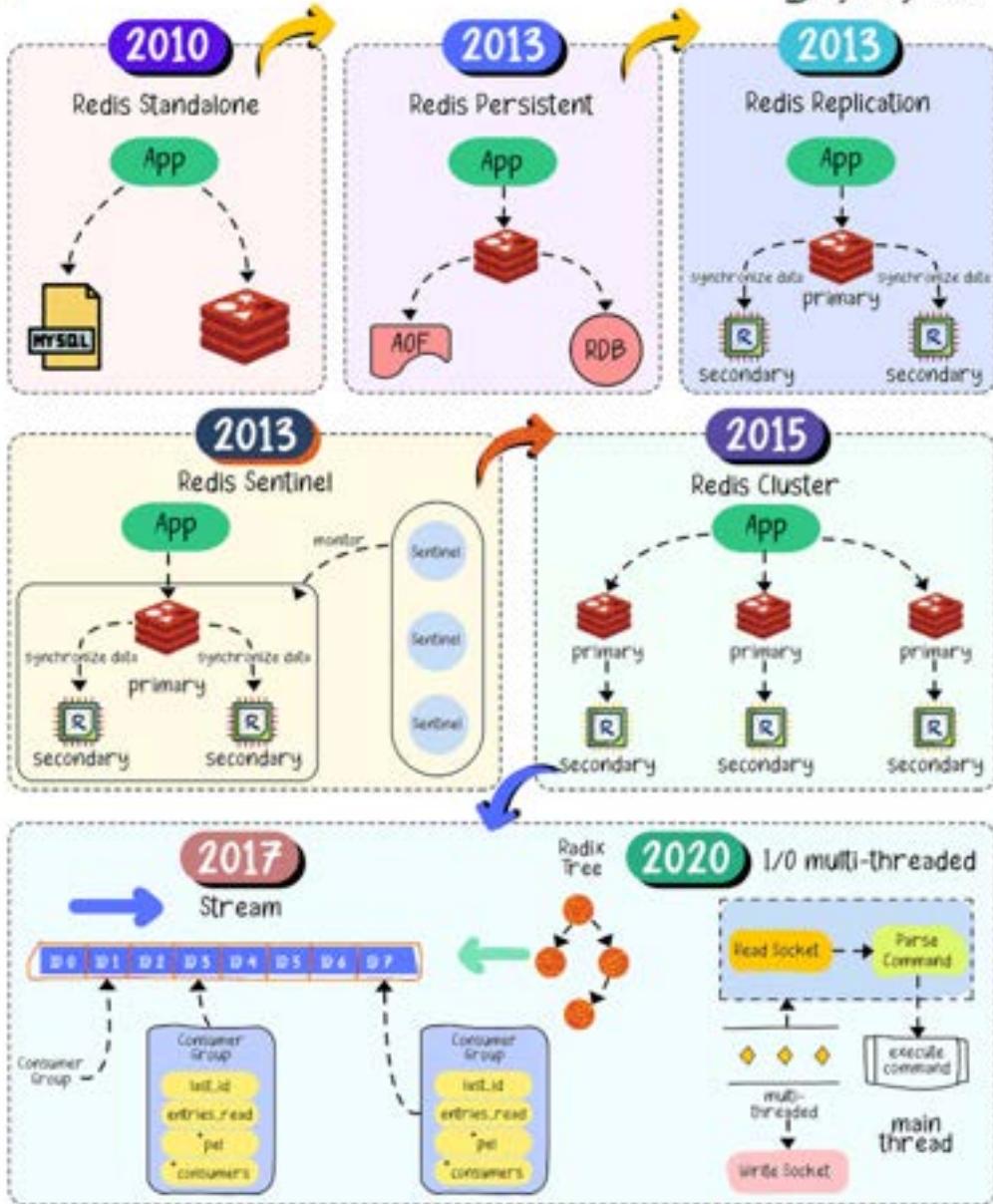
2. Dependencies:
List all the things your app needs to work properly, and make sure they're easy to install.
3. Config:
Keep important settings like database credentials separate from your code, so you can change them without rewriting code.
4. Backing Services:
Use other services (like databases or payment processors) as separate components that your app connects to.
5. Build, Release, Run:
Make a clear distinction between preparing your app, releasing it, and running it in production.
6. Processes:
Design your app so that each part doesn't rely on a specific computer or memory. It's like making LEGO blocks that fit together.
7. Port Binding:
Let your app be accessible through a network port, and make sure it doesn't store critical information on a single computer.
8. Concurrency:
Make your app able to handle more work by adding more copies of the same thing, like hiring more workers for a busy restaurant.
9. Disposability:
Your app should start quickly and shut down gracefully, like turning off a light switch instead of yanking out the power cord.
10. Dev/Prod Parity:
Ensure that what you use for developing your app is very similar to what you use in production, to avoid surprises.
11. Logs:
Keep a record of what happens in your app so you can understand and fix issues, like a diary for your software.
12. Admin Processes:
Run special tasks separately from your app, like doing maintenance work in a workshop instead of on the factory floor.

Over to you: Where do you think these principles can have the most impact in improving software development practices?

How does Redis architecture evolve?

How Redis Architecture Evolves ?

ByteByteGo



Redis is a popular in-memory cache. How did it evolve to the architecture it is today?

2010 - Standalone Redis

When Redis 1.0 was released in 2010, the architecture was quite simple. It is usually used as a cache to the business application.

However, Redis stores data in memory. When we restart Redis, we will lose all the data and the traffic directly hits the database.

◆ 2013 - Persistence

When Redis 2.8 was released in 2013, it addressed the previous restrictions. Redis introduced RDB in-memory snapshots to persist data. It also supports AOF (Append-Only-File), where each write command is written to an AOF file.

◆ 2013 - Replication

Redis 2.8 also added replication to increase availability. The primary instance handles real-time read and write requests, while replica synchronizes the primary's data.

◆ 2013 - Sentinel

Redis 2.8 introduced Sentinel to monitor the Redis instances in real time. is a system designed to help managing Redis instances. It performs the following four tasks: monitoring, notification, automatic failover and configuration provider.

◆ 2015 - Cluster

In 2015, Redis 3.0 was released. It added Redis clusters.

A Redis cluster is a distributed database solution that manages data through sharding. The data is divided into 16384 slots, and each node is responsible for a portion of the slot.

◆ Looking Ahead

Redis is popular because of its high performance and rich data structures that dramatically reduce the complexity of developing a business application.

In 2017, Redis 5.0 was released, adding the stream data type.

In 2020, Redis 6.0 was released, introducing the multi-threaded I/O in the network module. Redis model is divided into the network module and the main processing module. The Redis developers the network module tends to become a bottleneck in the system.

Over to you - have you used Redis before? If so, for what use case?

Cloud Cost Reduction Techniques



Irrational Cloud Cost is the biggest challenge many organizations are battling as they navigate the complexities of cloud computing.

Efficiently managing these costs is crucial for optimizing cloud usage and maintaining financial health.

The following techniques can help businesses effectively control and minimize their cloud expenses.

1. Reduce Usage:

Fine-tune the volume and scale of resources to ensure efficiency without compromising on the performance of applications (e.g., downsizing instances, minimizing storage space, consolidating services).

2. Terminate Idle Resources:

Locate and eliminate resources that are not in active use, such as dormant instances, databases, or storage units.

3. Right Sizing:

Adjust instance sizes to adequately meet the demands of your applications, ensuring neither underuse nor overuse.

4. Shutdown Resources During Off-Peak Times:

Set up automatic mechanisms or schedules for turning off non-essential resources when they are not in use, especially during low-activity periods.

5. Reserve to Reduce Rate:

Adopt cost-effective pricing models like Reserved Instances or Savings Plans that align with your specific workload needs.

Bonus Tip: Consider using Spot Instances and lower-tier storage options for additional cost savings.

6. Optimize Data Transfers:

Utilize methods such as data compression and Content Delivery Networks (CDNs) to cut down on bandwidth expenses, and strategically position resources to reduce data transfer costs, focusing on intra-region transfers.

Over to you: Which technique fits in well with your current cloud infra setup?

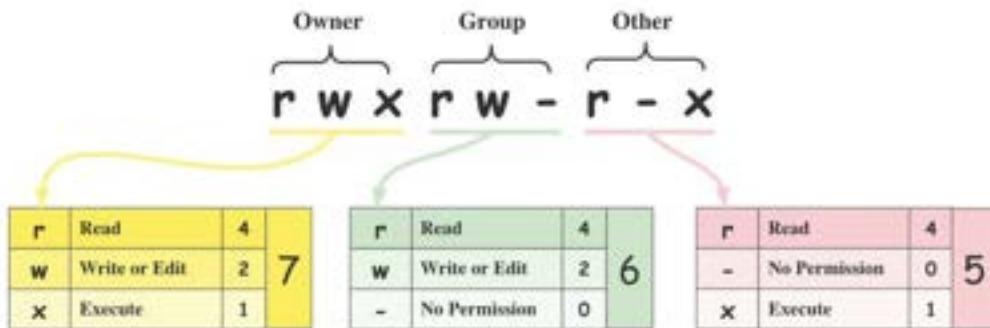
Linux file permission illustrated

To understand Linux file permissions, we need to understand Ownership and Permission.

Linux File Permissions

 blog.bytebytogo.com

Binary	Octal	String Representation	Permissions
000	0 (0+0+0)	---	No Permission
001	1 (0+0+1)	--x	Execute
010	2 (0+2+0)	-w-	Write
011	3 (0+2+1)	-wx	Write + Execute
100	4 (4+0+0)	r--	Read
101	5 (4+0+1)	r-x	Read + Execute
110	6 (4+2+0)	rw-	Read + Write
111	7 (4+2+1)	rwx	Read + Write + Execute



Ownership

Every file or directory is assigned 3 types of owner:

- Owner: the owner is the user who created the file or directory.
- Group: a group can have multiple users. All users in the group have the same permissions to access the file or directory.
- Other: other means those users who are not owners or members of the group.

Permission

There are only three types of permissions for a file or directory.

- Read (r): the read permission allows the user to read a file.
- Write (w): the write permission allows the user to change the content of the file.
- Execute (x): the execute permission allows a file to be executed.

Over to you: what are some of the commonly used Linux commands to change file permissions?

There are over 1,000 engineering blogs. Here are my top 9 favorites

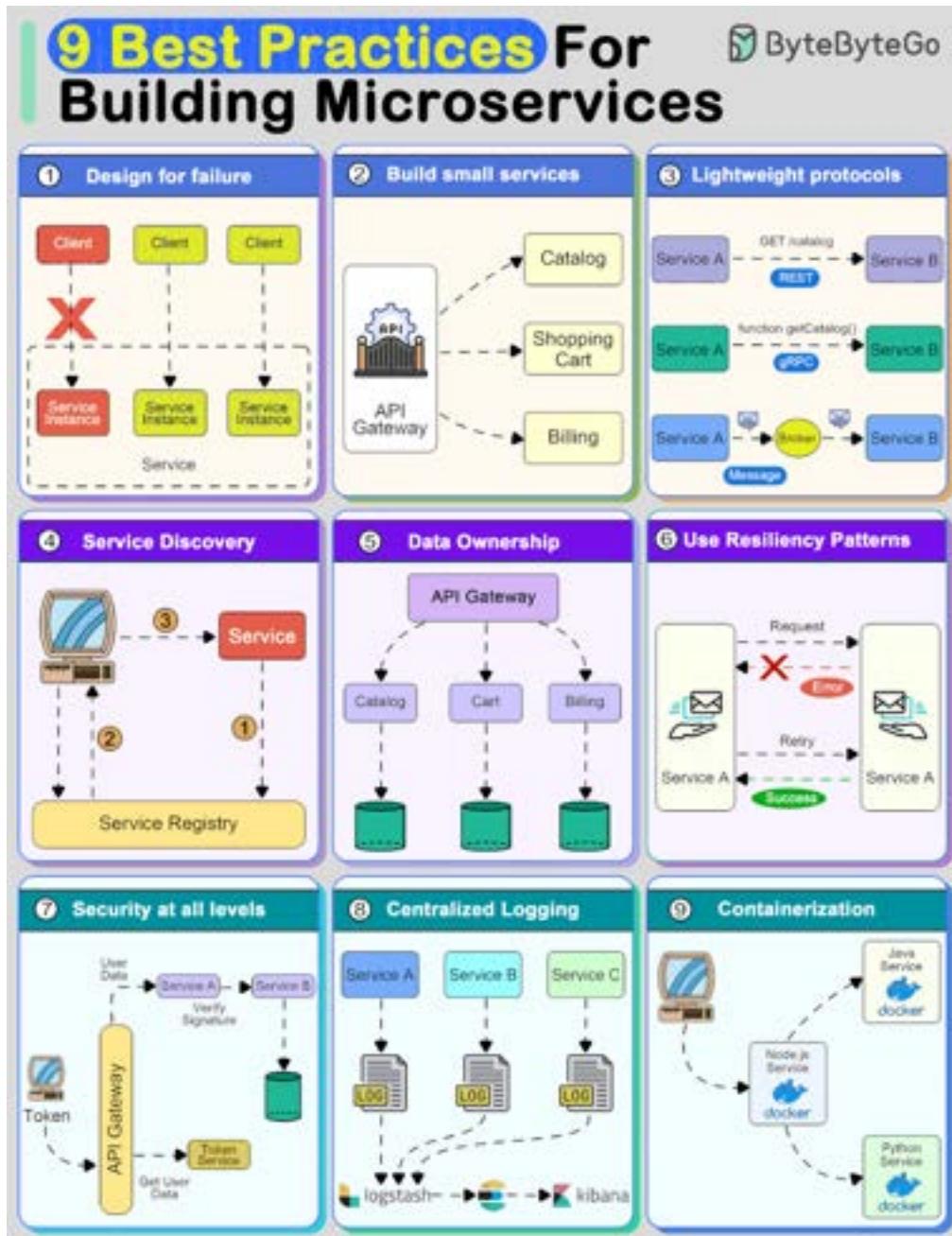


- Netflix TechBlog
- Uber Blog
- Cloudflare Blog
- Engineering at Meta
- LinkedIn Engineering
- Discord Blog
- AWS Architecture
- Slack Engineering
- Stripe Blog

Over to you - What are some of your favorite engineering blogs?

9 Best Practices for Building Microservices

Creating a system using microservices is extremely difficult unless you follow some strong principles.



1 - Design For Failure

A distributed system with microservices is going to fail.

You must design the system to tolerate failure at multiple levels such as infrastructure, database, and individual services. Use circuit breakers, bulkheads, or graceful degradation methods to deal with failures.

2 - Build Small Services

A microservice should not do multiple things at once.

A good microservice is designed to do one thing well.

3 - Use lightweight protocols for communication

Communication is the core of a distributed system.

Microservices must talk to each other using lightweight protocols. Options include REST, gRPC, or message brokers.

4 - Implement service discovery

To communicate with each other, microservices need to discover each other over the network.

Implement service discovery using tools such as Consul, Eureka, or Kubernetes Services

5 - Data Ownership

In microservices, data should be owned and managed by the individual services.

The goal should be to reduce coupling between services so that they can evolve independently.

6 - Use resiliency patterns

Implement specific resiliency patterns to improve the availability of the services.

Examples: retry policies, caching, and rate limiting.

7 - Security at all levels

In a microservices-based system, the attack surface is quite large. You must implement security at every level of the service communication path.

8 - Centralized logging

Logs are important to finding issues in a system. With multiple services, they become critical.

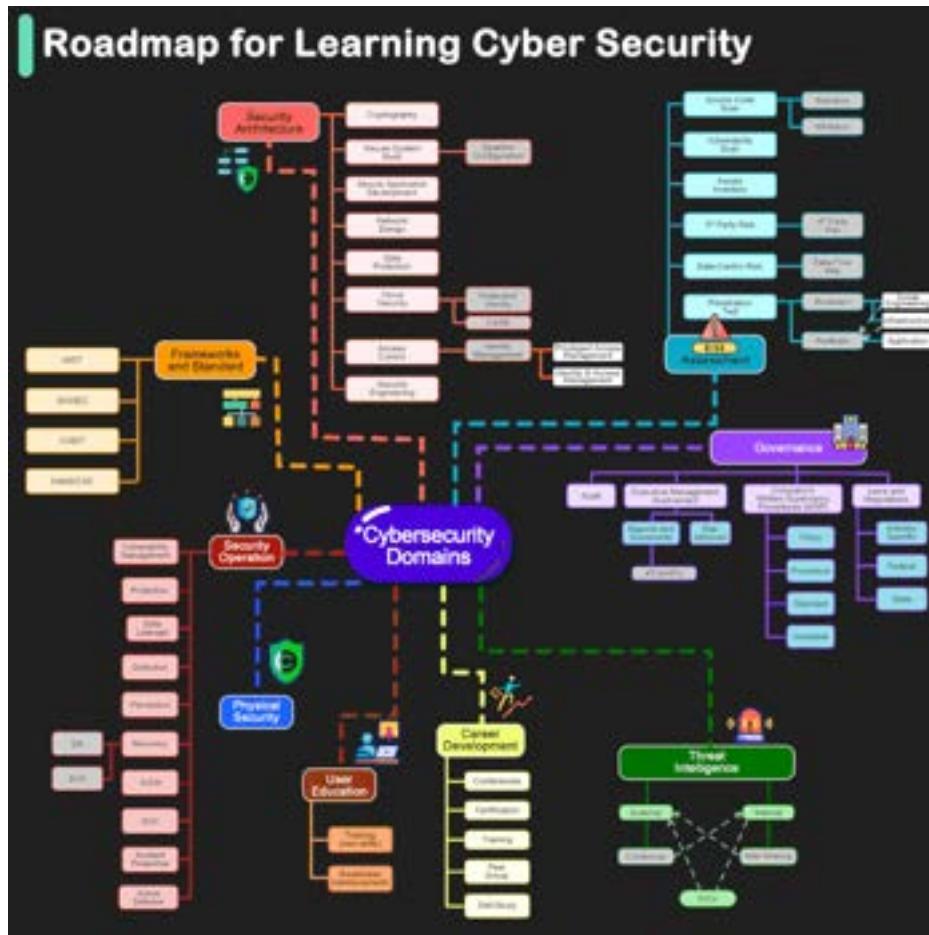
9 - Use containerization techniques

To deploy microservices in an isolated manner, use containerization techniques.

Tools like Docker and Kubernetes can help with this as they are meant to simplify the scaling and deployment of a microservice.

Over to you: what other best practice would you recommend?

Roadmap for Learning Cyber Security

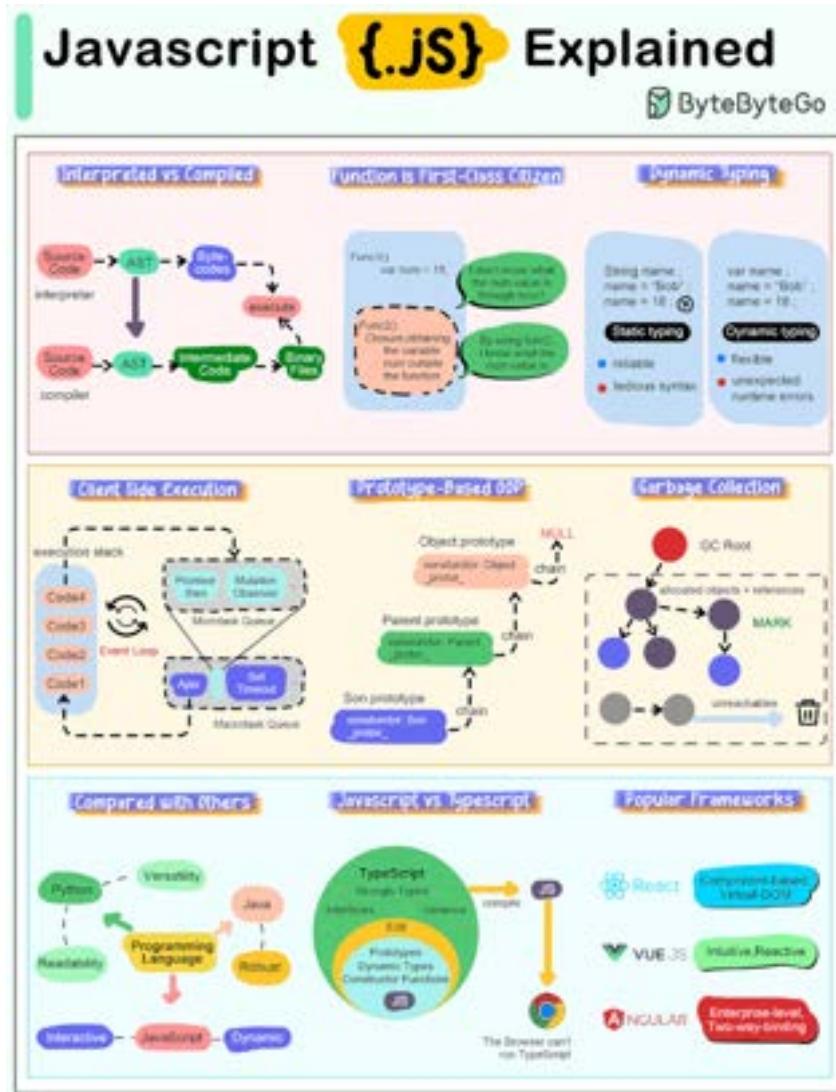


By [Henry Jiang](#). Redrawn by ByteByteGo.

Cybersecurity is crucial for protecting information and systems from theft, damage, and unauthorized access. Whether you're a beginner or looking to advance your technical skills, there are numerous resources and paths you can take to learn more about cybersecurity. Here are some structured suggestions to help you get started or deepen your knowledge:

- ◆ Security Architecture
- ◆ Frameworks & Standards
- ◆ Application Security
- ◆ Risk Assessment
- ◆ Enterprise Risk Management
- ◆ Threat Intelligence
- ◆ Security Operation

How does Javascript Work?



The cheat sheet below shows most important characteristics of Javascript.

◆ Interpreted Language

JavaScript code is executed by the browser or JavaScript engine rather than being compiled into machine language beforehand. This makes it highly portable across different platforms. Modern engines such as V8 utilize Just-In-Time (JIT) technology to compile code into directly executable machine code.

◆ Function is First-Class Citizen

In JavaScript, functions are treated as first-class citizens, meaning they can be stored in variables, passed as arguments to other functions, and returned from functions.

- ◆ **Dynamic Typing**

JavaScript is a loosely typed or dynamic language, meaning we don't have to declare a variable's type ahead of time, and the type can change at runtime.

- ◆ **Client-Side Execution**

JavaScript supports asynchronous programming, allowing operations like reading files, making HTTP requests, or querying databases to run in the background and trigger callbacks or promises when complete. This is particularly useful in web development for improving performance and user experience.

- ◆ **Prototype-Based OOP**

Unlike class-based object-oriented languages, JavaScript uses prototypes for inheritance. This means that objects can inherit properties and methods from other objects.

- ◆ **Automatic Garbage Collection**

Garbage collection in JavaScript is a form of automatic memory management. The primary goal of garbage collection is to reclaim memory occupied by objects that are no longer in use by the program, which helps prevent memory leaks and optimizes the performance of the application.

- ◆ **Compared with Other Languages**

JavaScript is special compared to programming languages like Python or Java because of its position as a major language for web development.

While Python is known to provide good code readability and versatility, and Java is known for its structure and robustness, JavaScript is an interpreted language that runs directly on the browser without compilation, emphasizing flexibility and dynamism.

- ◆ **Relationship with Typescript**

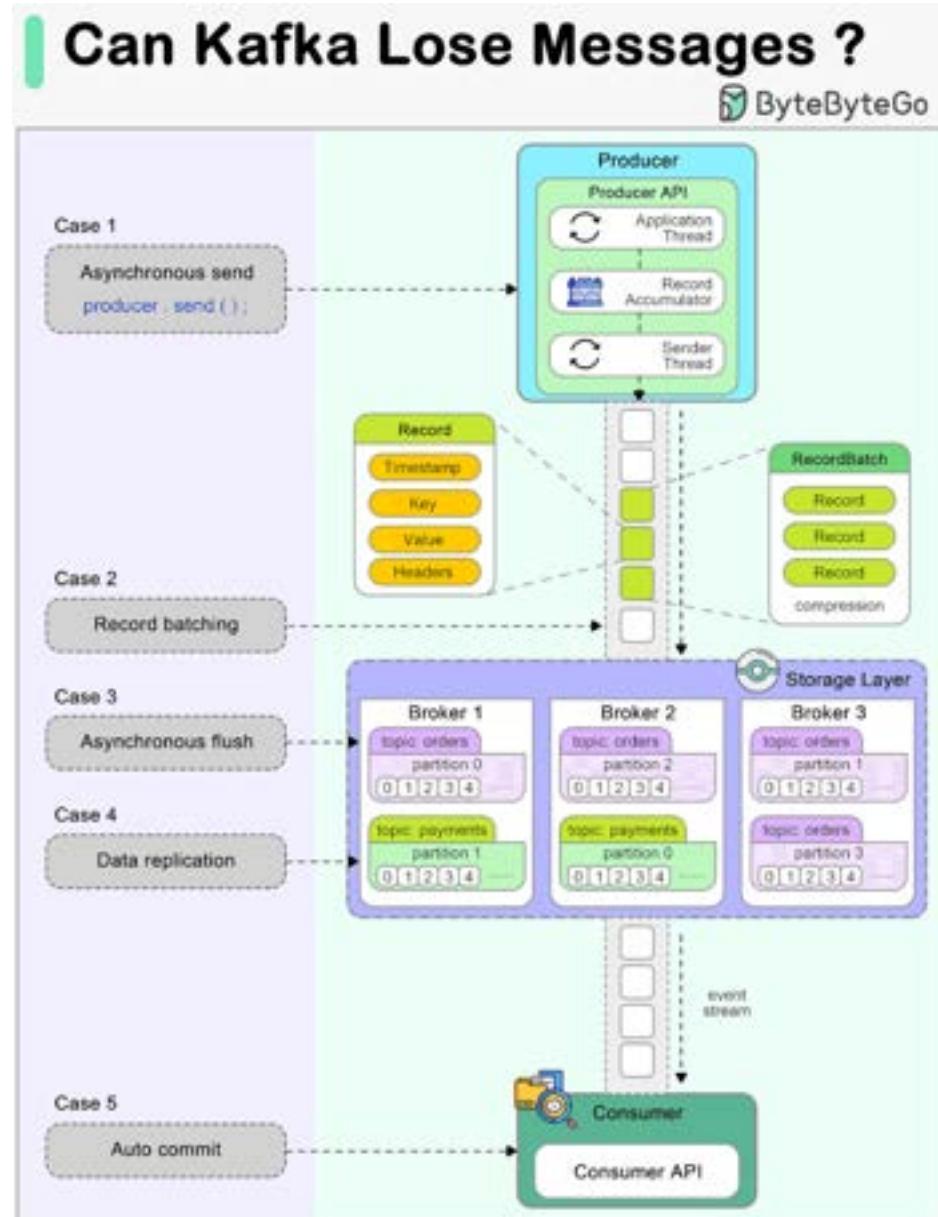
TypeScript is a superset of JavaScript, which means that it extends JavaScript by adding features to the language, most notably type annotations. This relationship allows any valid JavaScript code to also be considered valid TypeScript code.

- ◆ **Popular Javascript Frameworks**

React is known for its flexibility and large number of community-driven plugins, while Vue is clean and intuitive with highly integrated and responsive features. Angular, on the other hand, offers a strict set of development specifications for enterprise-level JS development.

Can Kafka Lose Messages?

Error handling is one of the most important aspects of building reliable systems.



Today, we will discuss an important topic: Can Kafka lose messages?

A common belief among many developers is that Kafka, by its very design, guarantees no message loss. However, understanding the nuances of Kafka's architecture and configuration is essential to truly grasp how and when it might lose messages, and more importantly, how to prevent such scenarios.

The diagram below shows how a message can be lost during its lifecycle in Kafka.

◆ Producer

When we call `producer.send()` to send a message, it doesn't get sent to the broker directly. There are two threads and a queue involved in the message-sending process:

1. Application thread
2. Record accumulator
3. Sender thread (I/O thread)

We need to configure proper 'acks' and 'retries' for the producer to make sure messages are sent to the broker.

◆ Broker

A broker cluster should not lose messages when it is functioning normally. However, we need to understand which extreme situations might lead to message loss:

1. The messages are usually flushed to the disk asynchronously for higher I/O throughput, so if the instance is down before the flush happens, the messages are lost.
2. The replicas in the Kafka cluster need to be properly configured to hold a valid copy of the data. The determinism in data synchronization is important.

◆ Consumer

Kafka offers different ways to commit messages. Auto-committing might acknowledge the processing of records before they are actually processed. When the consumer is down in the middle of processing, some records may never be processed.

A good practice is to combine both synchronous and asynchronous commits, where we use asynchronous commits in the processing loop for higher throughput and synchronous commits in exception handling to make sure the last offset is always committed.

You're Decent at Linux if You Know What Those Directories Mean :)



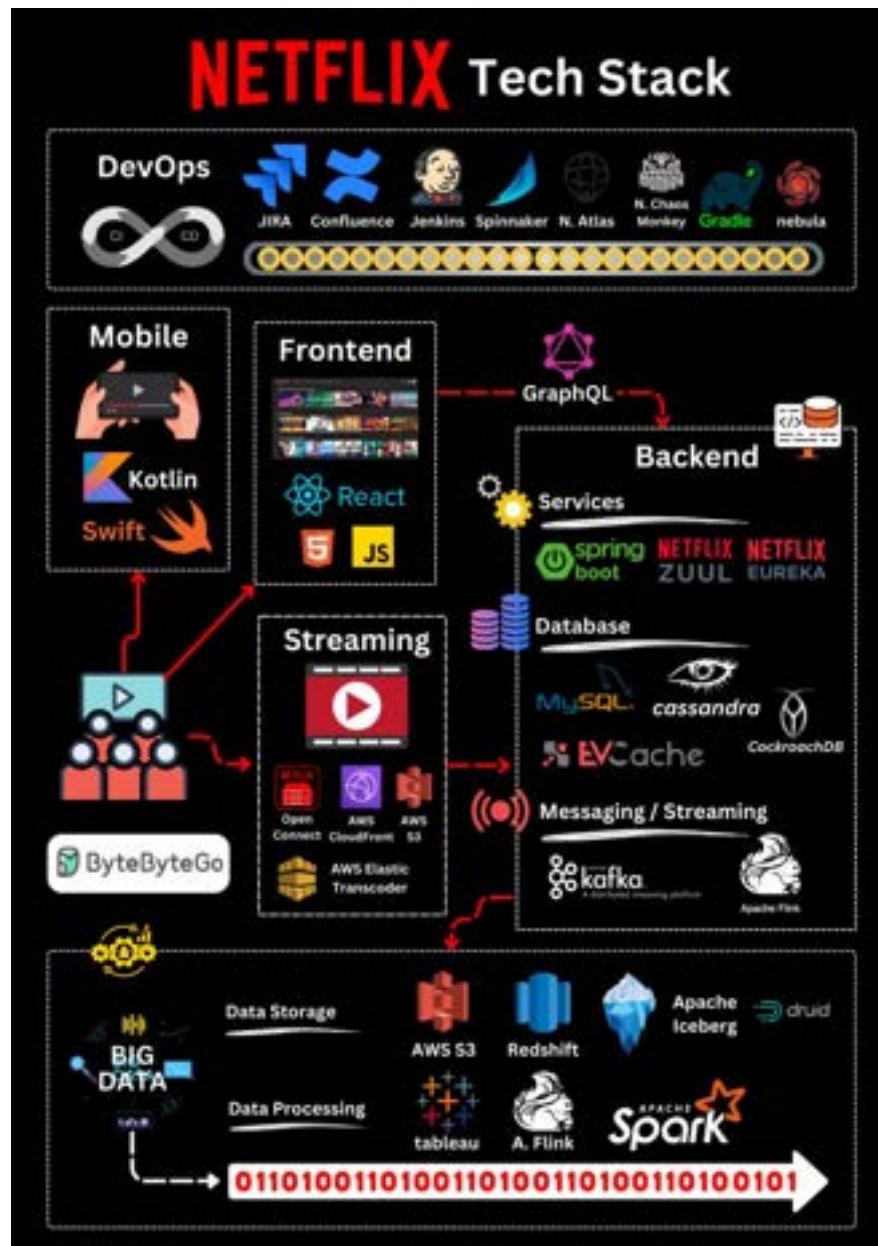
The Linux file system used to resemble an unorganized town where individuals constructed their houses wherever they pleased. However, in 1994, the Filesystem Hierarchy Standard (FHS) was introduced to bring order to the Linux file system.

By implementing a standard like the FHS, software can ensure a consistent layout across various Linux distributions. Nonetheless, not all Linux distributions strictly adhere to this standard. They often incorporate their own unique elements or cater to specific requirements.

To become proficient in this standard, you can begin by exploring. Utilize commands such as "cd" for navigation and "ls" for listing directory contents. Imagine the file system as a tree, starting from the root (/). With time, it will become second nature to you, transforming you into a skilled Linux administrator.

Have fun exploring!

Netflix's Tech Stack



This post is based on research from many Netflix engineering blogs and open-source projects. If you come across any inaccuracies, please feel free to inform us.

Mobile and web: Netflix has adopted Swift and Kotlin to build native mobile apps. For its web application, it uses React.

Frontend/server communication: GraphQL.

Backend services: Netflix relies on ZUUL, Eureka, the Spring Boot framework, and other technologies.

Databases: Netflix utilizes EV cache, Cassandra, CockroachDB, and other databases.

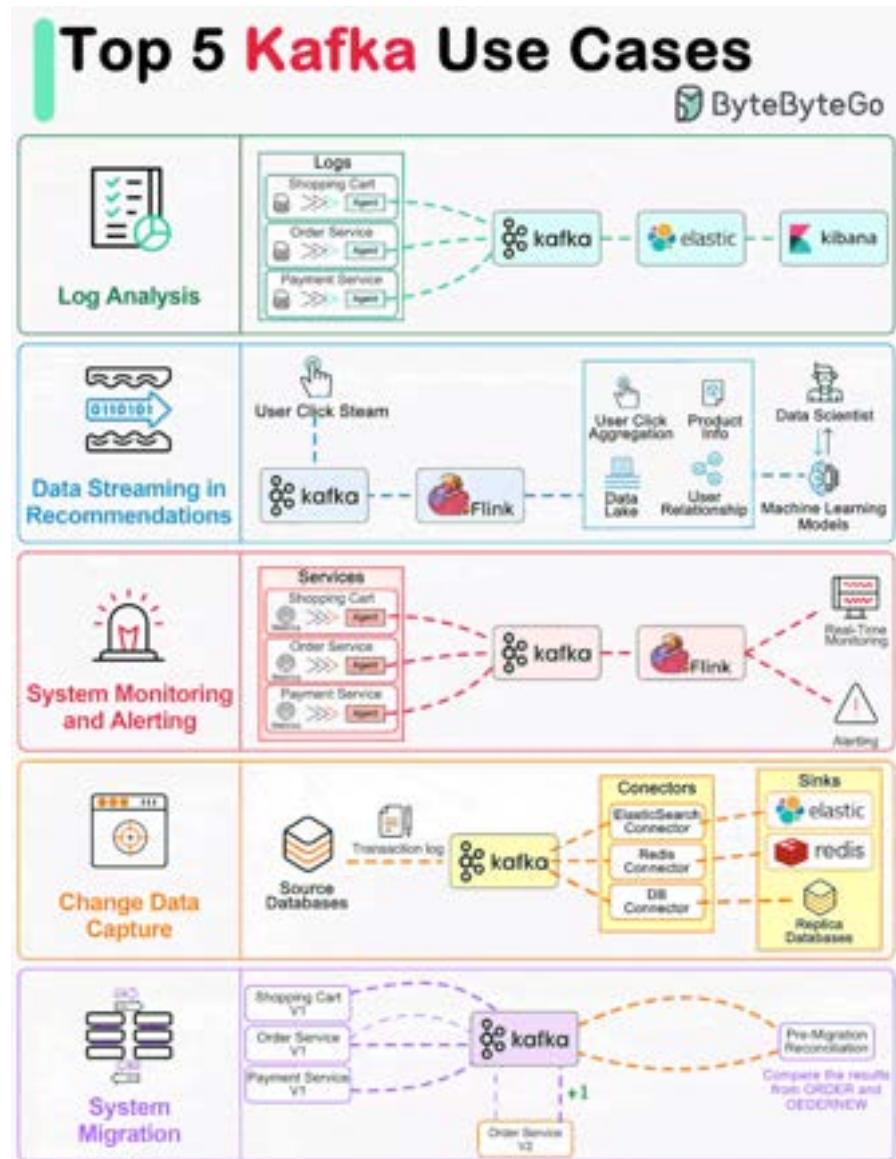
Messaging/streaming: Netflix employs Apache Kafka and Flink for messaging and streaming purposes.

Video storage: Netflix uses S3 and Open Connect for video storage.

Data processing: Netflix utilizes Flink and Spark for data processing, which is then visualized using Tableau. Redshift is used for processing structured data warehouse information.

CI/CD: Netflix employs various tools such as JIRA, Confluence, PagerDuty, Jenkins, Gradle, Chaos Monkey, Spinnaker, Altas, and more for CI/CD processes.

Top 5 Kafka use cases



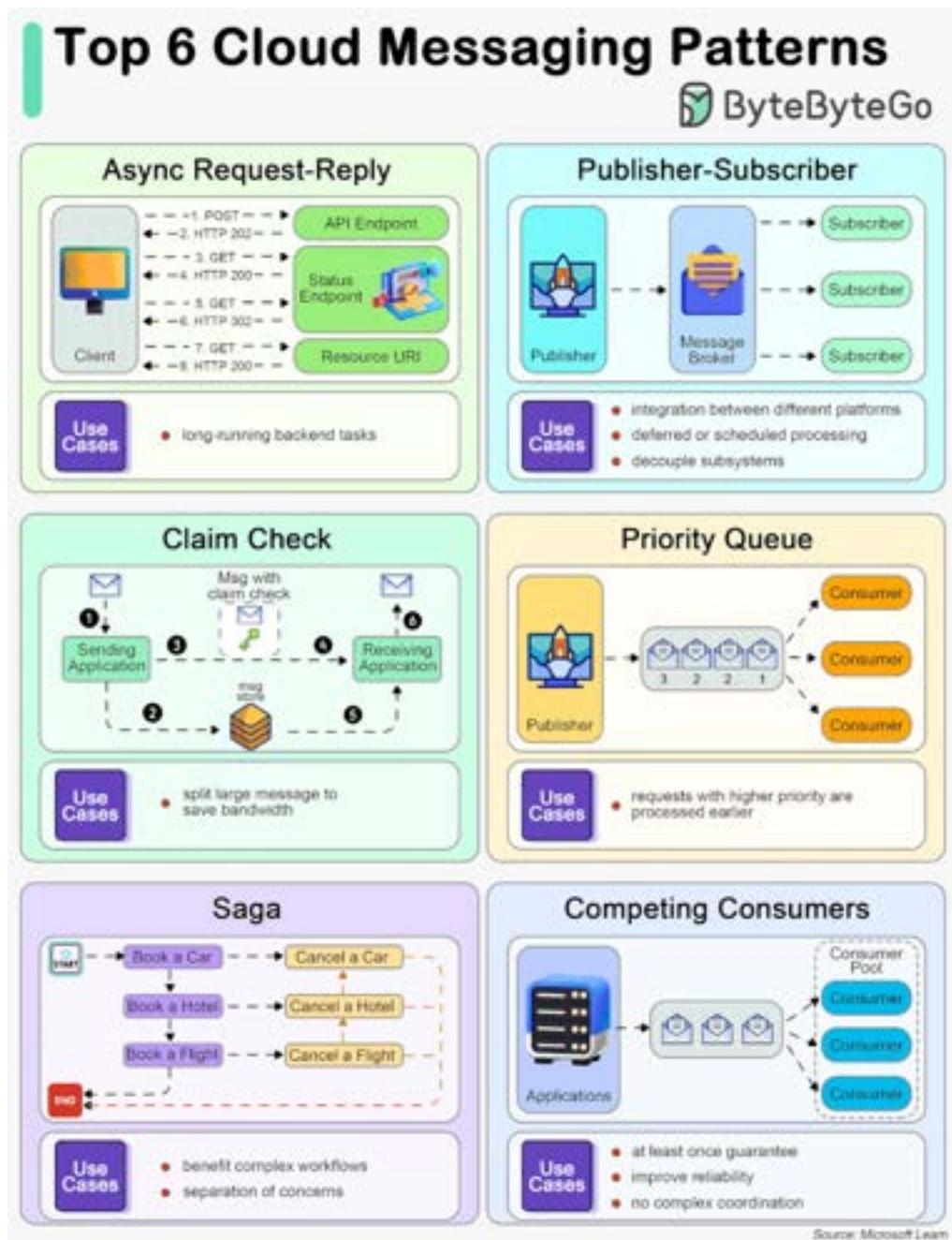
Kafka was originally built for massive log processing. It retains messages until expiration and lets consumers pull messages at their own pace.

Let's review the popular Kafka use cases.

- Log processing and analysis
- Data streaming in recommendations
- System monitoring and alerting
- CDC (Change data capture)

- System migration

Top 6 Cloud Messaging Patterns.



How do services communicate with each other? The diagram below shows 6 cloud messaging patterns.

- Asynchronous Request-Reply

This pattern aims at providing determinism for long-running backend tasks. It decouples backend processing from frontend clients.

In the diagram below, the client makes a synchronous call to the API, triggering a long-running operation on the backend. The API returns an HTTP 202 (Accepted) status code, acknowledging that the request has been received for processing.

◆ Publisher-Subscriber

This pattern targets decoupling senders from consumers, and avoiding blocking the sender to wait for a response.

◆ Claim Check

This pattern solves the transmission of large messages. It stores the whole message payload into a database and transmits only the reference to the message, which will be used later to retrieve the payload from the database.

◆ Priority Queue

This pattern prioritizes requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority.

◆ Saga

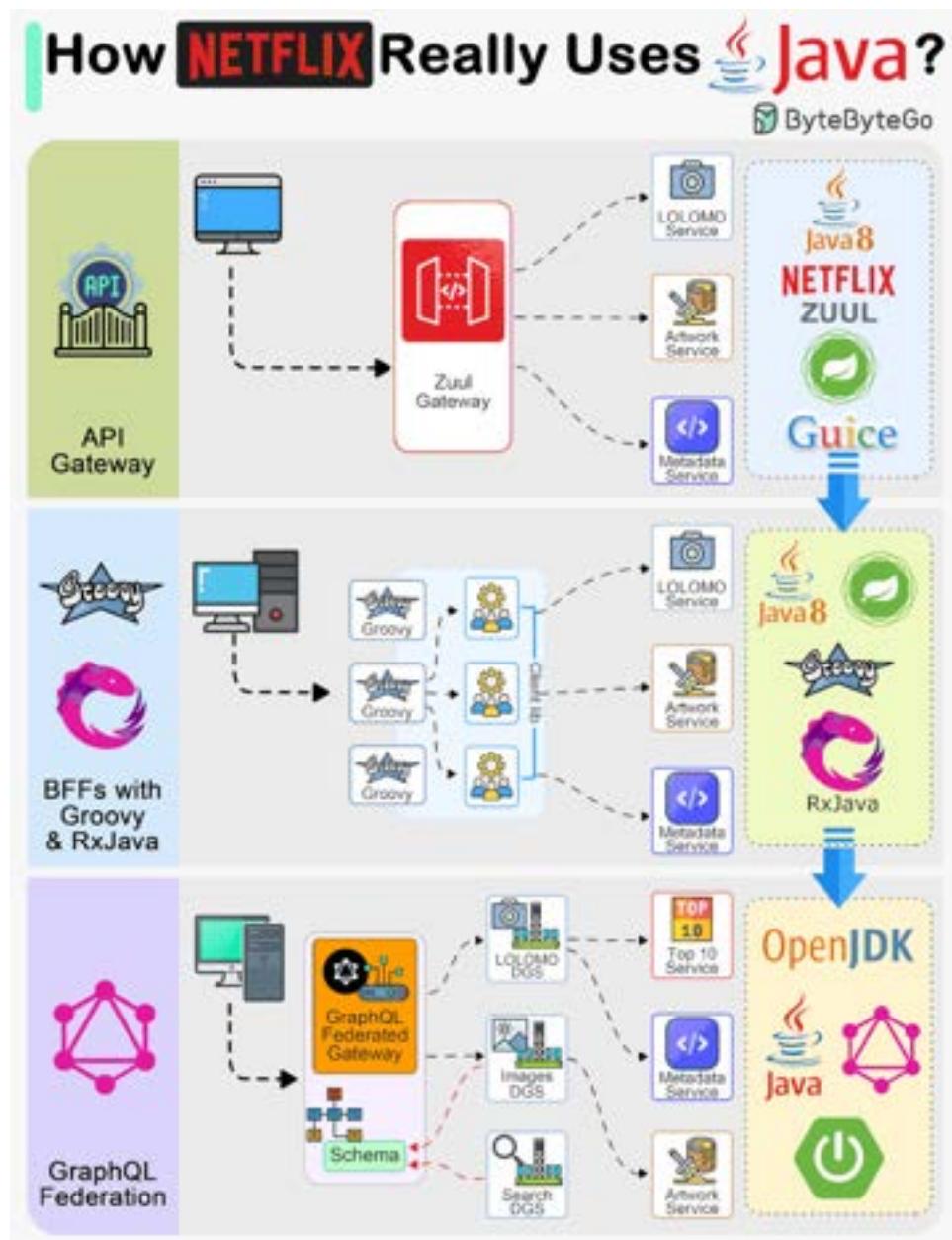
Saga is used to manage data consistency across multiple services in distributed systems, especially in microservices architectures where each service manages its own database.

The saga pattern addresses the challenge of maintaining data consistency without relying on distributed transactions, which are difficult to scale and can negatively impact system performance.

◆ Competing Consumers

This pattern enables multiple concurrent consumers to process messages received on the same messaging channel. There is no need to configure complex coordination between the consumers. However, this pattern cannot guarantee message ordering.

How Netflix Really Uses Java?



Netflix is predominantly a Java shop.

Every backend application (including internal apps, streaming, and movie production apps) at Netflix is a Java application.

However, the Java stack is not static and has gone through multiple iterations over the years.

Here are the details of those iterations:

1 - API Gateway

Netflix follows a microservices architecture. Every piece of functionality and data is owned by a microservice built using Java (initially version 8)

This means that rendering one screen (such as the List of List of Movies or LOLOMO) involved fetching data from 10s of microservices. But making all these calls from the client created a performance problem.

Netflix initially used the API Gateway pattern using Zuul to handle the orchestration.

2 - BFFs with Groovy & RxJava

Using a single gateway for multiple clients was a problem for Netflix because each client (such as TV, mobile apps, or web browser) had subtle differences.

To handle this, Netflix used the Backend-for-Frontend (BFF) pattern. Zuul was moved to the role of a proxy

In this pattern, every frontend or UI gets its own mini backend that performs the request fanout and orchestration for multiple services.

The BFFs were built using Groovy scripts and the service fanout was done using RxJava for thread management.

3 - GraphQL Federation

The Groovy and RxJava approach required more work from the UI developers in creating the Groovy scripts. Also, reactive programming is generally hard.

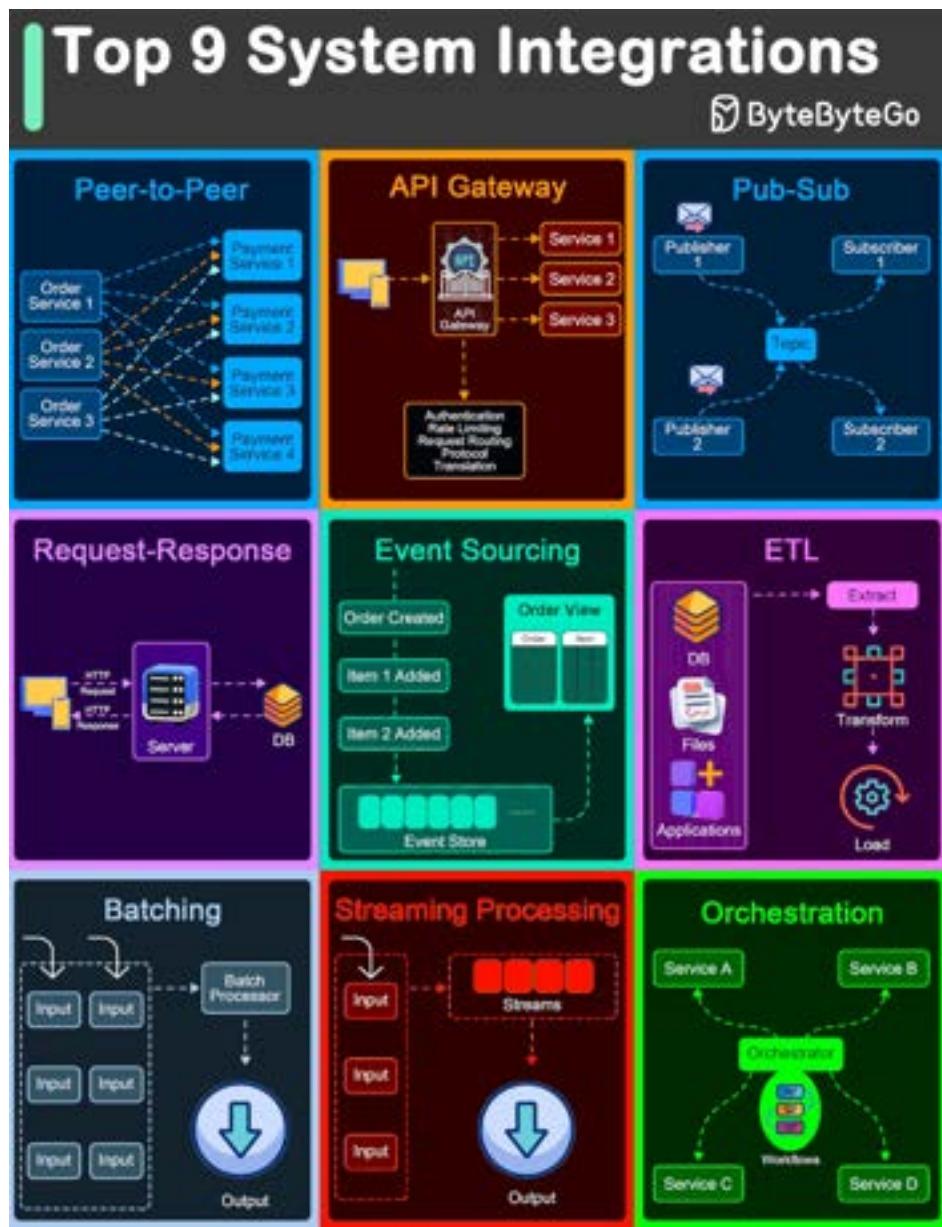
Recently, Netflix moved to GraphQL Federation. With GraphQL, a client can specify exactly what set of fields it needs, thereby solving the problem of overfetching and underfetching with REST APIs.

The GraphQL Federation takes care of calling the necessary microservices to fetch the data.

These microservices are called Domain Graph Service (DGS) and are built using Java 17, Spring Boot 3, and Spring Boot Netflix OSS packages. The move from Java 8 to Java 17 resulted in 20% CPU gains.

More recently, Netflix has started to migrate to Java 21 to take advantage of features like virtual threads.

Top 9 Architectural Patterns for Data and Communication Flow



◆ Peer-to-Peer

The Peer-to-Peer pattern involves direct communication between two components without the need for a central coordinator.

◆ API Gateway

An API Gateway acts as a single entry point for all client requests to the backend services of an application.

Pub-Sub

The Pub-Sub pattern decouples the producers of messages (publishers) from the consumers of messages (subscribers) through a message broker.

Request-Response

This is one of the most fundamental integration patterns, where a client sends a request to a server and waits for a response.

Event Sourcing

Event Sourcing involves storing the state changes of an application as a sequence of events.

ETL

ETL is a data integration pattern used to gather data from multiple sources, transform it into a structured format, and load it into a destination database.

Batching

Batching involves accumulating data over a period or until a certain threshold is met before processing it as a single group.

Streaming Processing

Streaming Processing allows for the continuous ingestion, processing, and analysis of data streams in real-time.

Orchestration

Orchestration involves a central coordinator (an orchestrator) managing the interactions between distributed components or services to achieve a workflow or business process.

What Are the Most Important AWS Services To Learn?

Most Important AWS Services To Learn					ByteByteGo
Cloud Computing Essentials	Simple Storage Service (S3)	Elastic Compute Cloud (EC2)			
Cloud First Steps	Management Console	IAM	CloudWatch		
Computing Solutions	Elastic Compute Cloud (EC2)	Lambda	Fargate	ECS	
Cloud Economics	Cost Explorer	Budgets	Cost & Usage Report		
Networking Concepts	Virtual Private Cloud (VPC)	Route 53	API Gateway		
Connecting VPCs	VPC Peering	Transit Gateway	Direct Connect	VPN Connection	
First NoSQL Database	DynamoDB				
Storage and File Systems	Simple Storage Service (S3)	EBS	EFS	FSx	
Auto-healing and Scaling Applications	EC2 Auto Scaling	ELB	CloudFront		
Serverless Architectures	Simple Storage Service (S3)	Lambda	API Gateway	Step Functions	
DevOps and Continuous Delivery	Code Commit	Code Build	Code Deploy	Code Pipeline	
Relational Database Solutions	RDS	Aurora			
Monitoring and Observability	CloudWatch	CloudTrail	Config	X-Ray	
Messaging and Queuing	SNS	SQS	MQ		
Security and Compliance	IAM	WAF	Shield	GuardDuty	
	Secrets Manager	KMS	NACL	Security Groups	
Migration to the Cloud	Application Migration Service	DMS	Snowball	DataSync	
Machine Learning and AI	Comprehend	SageMaker	Lex	Rekognition	

Since its inception in 2006, AWS has rapidly evolved from simple offerings like S3 and EC2 to an expansive, versatile cloud ecosystem.

Today, AWS provides a highly reliable, scalable infrastructure platform with over 200 services in the cloud, powering hundreds of thousands of businesses in 190 countries around the world.

For both newcomers and seasoned professionals, navigating the broad set of AWS services is no small feat.

From computing power, storage options, and networking capabilities to database management, analytics, and machine learning, AWS provides a wide array of tools that can be daunting to understand and master.

Each service is tailored to specific needs and use cases, requiring a deep understanding of not just the services themselves, but also how they interact and integrate within an IT ecosystem.

This attached illustration can serve as both a starting point and a quick reference for anyone looking to demystify AWS and focus their efforts on the services that matter most.

It provides a visual roadmap, outlining the foundational services that underpin cloud computing essentials, as well as advanced services catering to specific needs like serverless architectures, DevOps, and machine learning.

8 Key Data Structures That Power Modern Databases

8 Data Structures That Power Your Databases		ByteByteGo.com	
Types	Illustration	Use Case	Note
Skiplist		In-memory	used in Redis
Hash Index		In-memory	Most common in-memory index solution
SSTable		Disk-based	Immutable data structure. Seldom used alone.
LSM tree		Memory + Disk	High write throughput. Disk compaction may impact performance
B-tree		Disk-based	Most popular database index implementation
Inverted index		Search document	Used in document search engine such as Lucene
Suffix tree		Search string	Used in string search, such as string suffix match
R-tree		Search multi-dimension shape	Such as the nearest neighbor

- Skiplist: a common in-memory index type. Used in Redis
- Hash index: a very common implementation of the “Map” data structure (or “Collection”)
- SSTable: immutable on-disk “Map” implementation
- LSM tree: Skiplist + SSTable. High write throughput
- B-tree: disk-based solution. Consistent read/write performance
- Inverted index: used for document indexing. Used in Lucene
- Suffix tree: for string pattern search
- R-tree: multi-dimension search, such as finding the nearest neighbor

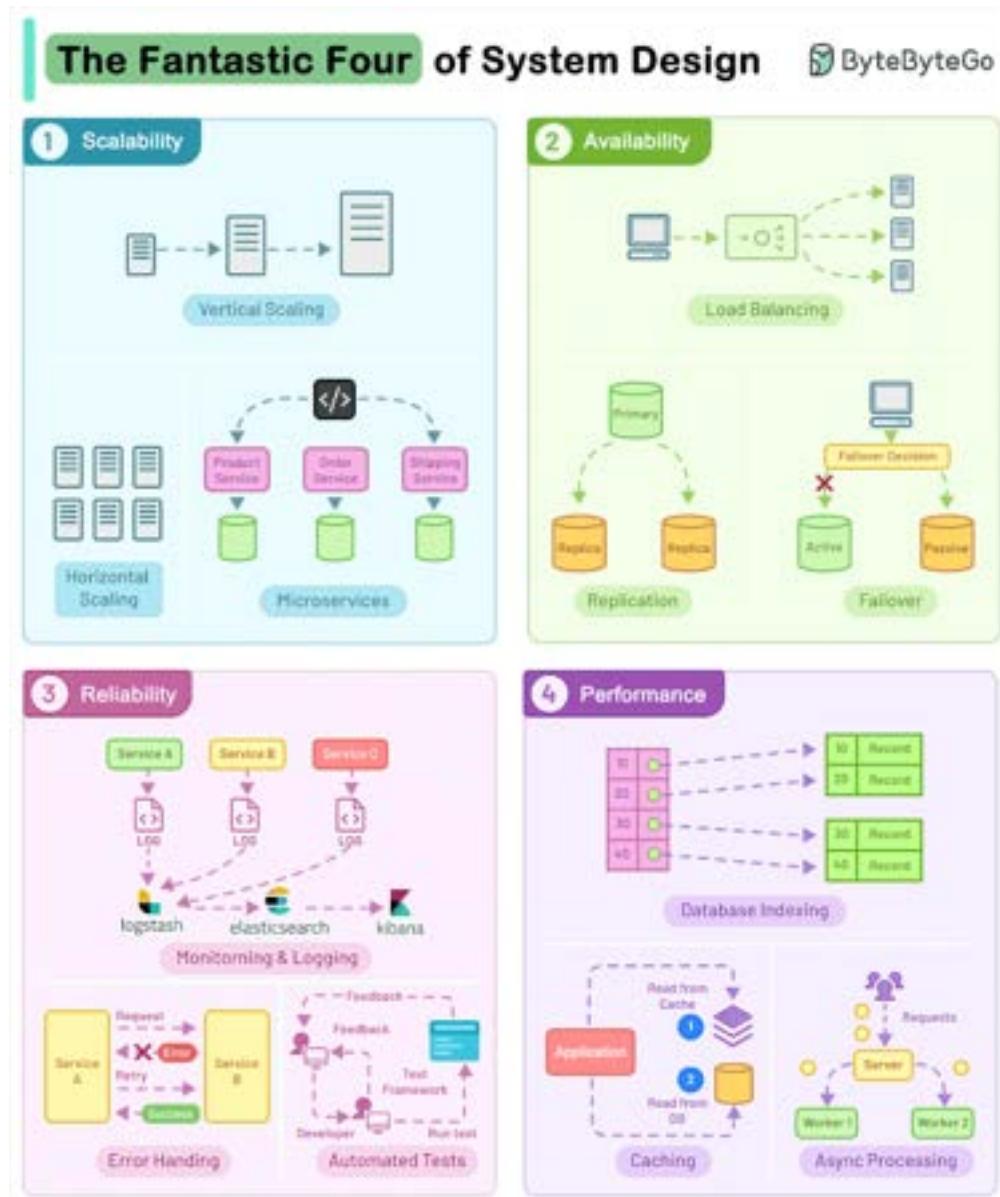
How do we design effective and safe APIs?

Design Effective & Safe APIs		
Design a Shopping Cart		
Use resource names (nouns)	✗ GET /querycarts/123	✓ GET /carts/123
Use plurals	✗ GET /cart/123	✓ GET /carts/123
Idempotency	✗ POST /carts	✓ POST /carts?{requestId: 4321}
Use versioning	✗ GET /carts/v1/123	✓ GET /v1/carts/123
Query after soft deletion	✗ GET /carts	✓ GET /carts?includeDeleted=true
Pagination	✗ GET /carts	✓ GET /carts?pageSize=xx&pageToken=xx
Sorting	✗ GET /items	✓ GET /items?sort_by=time
Filtering	✗ GET /items	✓ GET /items?filter=color:red
Secure Access	✗ X-API-KEY:xxx	✓ X-API-KEY:xxx X-EXPIRY:xxx X-REQUEST-SIGNATURE:xxx <small>https://URL + QueryString + Expiry + Body</small>
Resource cross reference	✗ GET /carts/123?item=321	✓ GET /carts/123/items/321
Add an item to a cart	✗ POST /carts/123?addItem=321	✓ POST /carts/123/items/add [itemId: "items/321"]
Rate limit	✗ No rate limit - DDos	✓ Design rate limiting rules based on IP, user, action group etc

The diagram below shows typical API designs with a shopping cart example.

Note that API design is not just URL path design. Most of the time, we need to choose the proper resource names, identifiers, and path patterns. It is equally important to design proper HTTP header fields or to design effective rate-limiting rules within the API gateway.

Who are the Fantastic Four of System Design?



Scalability, Availability, Reliability, and Performance.

They are the most critical components to crafting successful software systems.

Let's look at each of them with implementation techniques:

1 - Scalability

Scalability ensures that your application can handle more load without compromising performance.

2 - Availability

Availability makes sure that your application is always ready to serve the users and downtime is minimal.

3 - Reliability

Reliability is about building software that consistently delivers correct results.

4 - Performance

Performance is the ability of a system to carry out its tasks at an expected rate under peak load using available resources.

Over to you: What are the other pillars of system design and strategies you've come across?

How do we design a secure system?



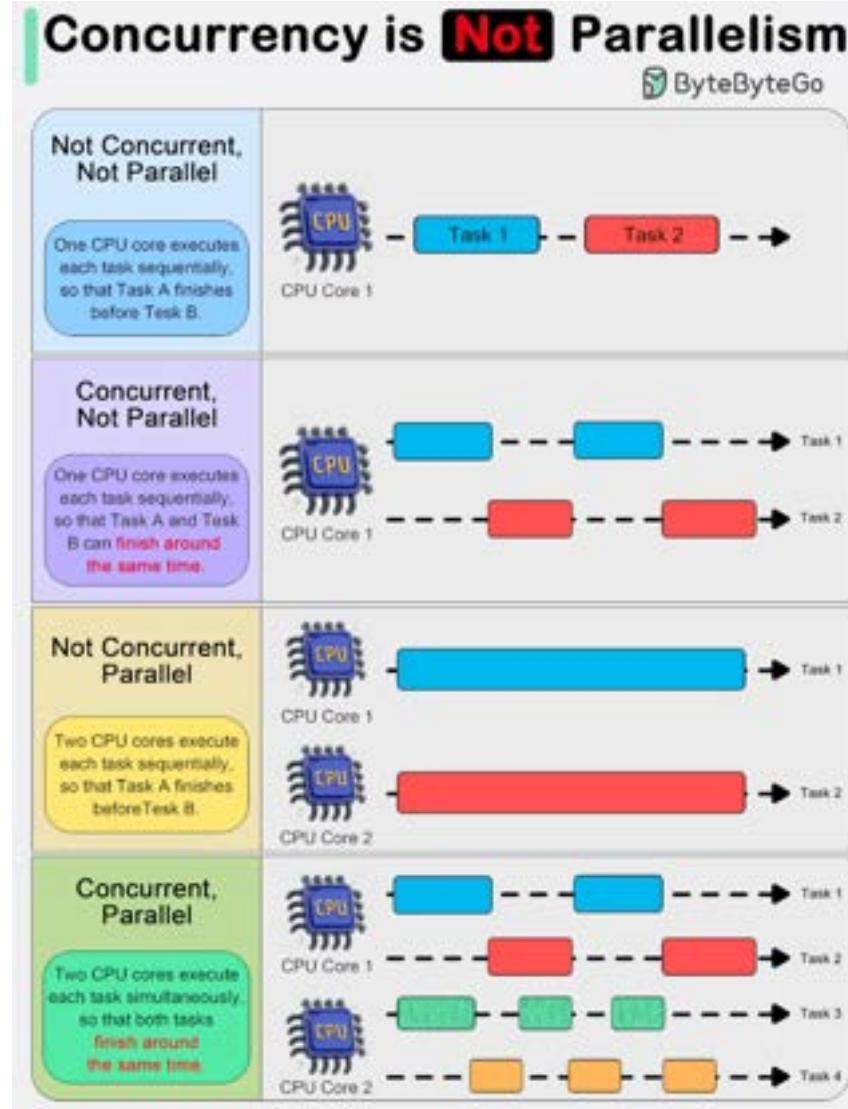
Designing secure systems is important for a multitude of reasons, spanning from protecting sensitive information to ensuring the stability and reliability of the infrastructure. As developers, we should design and implement these security guidelines by default.

The diagram below is a pragmatic cheat sheet with the use cases and key design points.

- ◆ Authentication
- ◆ Authorization
- ◆ Encryption
- ◆ Vulnerability
- ◆ Audit & Compliance
- ◆ Network Security
- ◆ Terminal Security
- ◆ Emergency Responses

- ◆ Container Security
- ◆ API Security
- ◆ 3rd-Party Vendor Management
- ◆ Disaster Recovery

Things Every Developer Should Know: Concurrency is NOT parallelism.



In system design, it is important to understand the difference between concurrency and parallelism.

As Rob Pike (one of the creators of GoLang) stated: "Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once." This distinction emphasizes that concurrency is more about the **design** of a program, while parallelism is about the **execution**.

Concurrency is about dealing with multiple things at once. It involves structuring a program to handle multiple tasks simultaneously, where the tasks can start, run, and complete in overlapping time periods, but not necessarily at the same instant.

Concurrency is about the composition of independently executing processes and describes a program's ability to manage multiple tasks by making progress on them without necessarily completing one before it starts another.

Parallelism, on the other hand, refers to the simultaneous execution of multiple computations. It is the technique of running two or more tasks or computations at the same time, utilizing multiple processors or cores within a computer to perform several operations concurrently. Parallelism requires hardware with multiple processing units, and its primary goal is to increase the throughput and computational speed of a system.

In practical terms, concurrency enables a program to remain responsive to input, perform background tasks, and handle multiple operations in a seemingly simultaneous manner, even on a single-core processor. It's particularly useful in I/O-bound and high-latency operations where programs need to wait for external events, such as file, network, or user interactions.

Parallelism, with its ability to perform multiple operations at the same time, is crucial in CPU-bound tasks where computational speed and throughput are the bottlenecks. Applications that require heavy mathematical computations, data analysis, image processing, and real-time processing can significantly benefit from parallel execution.

HTTPS, SSL Handshake, and Data Encryption Explained to Kids.



HTTPS: Safeguards your data from eavesdroppers and breaches. Understand how encryption and digital certificates create an impregnable shield.

SSL Handshake: Behind the Scenes — Witness the cryptographic protocols that establish a secure connection. Experience the intricate exchange of keys and negotiation.

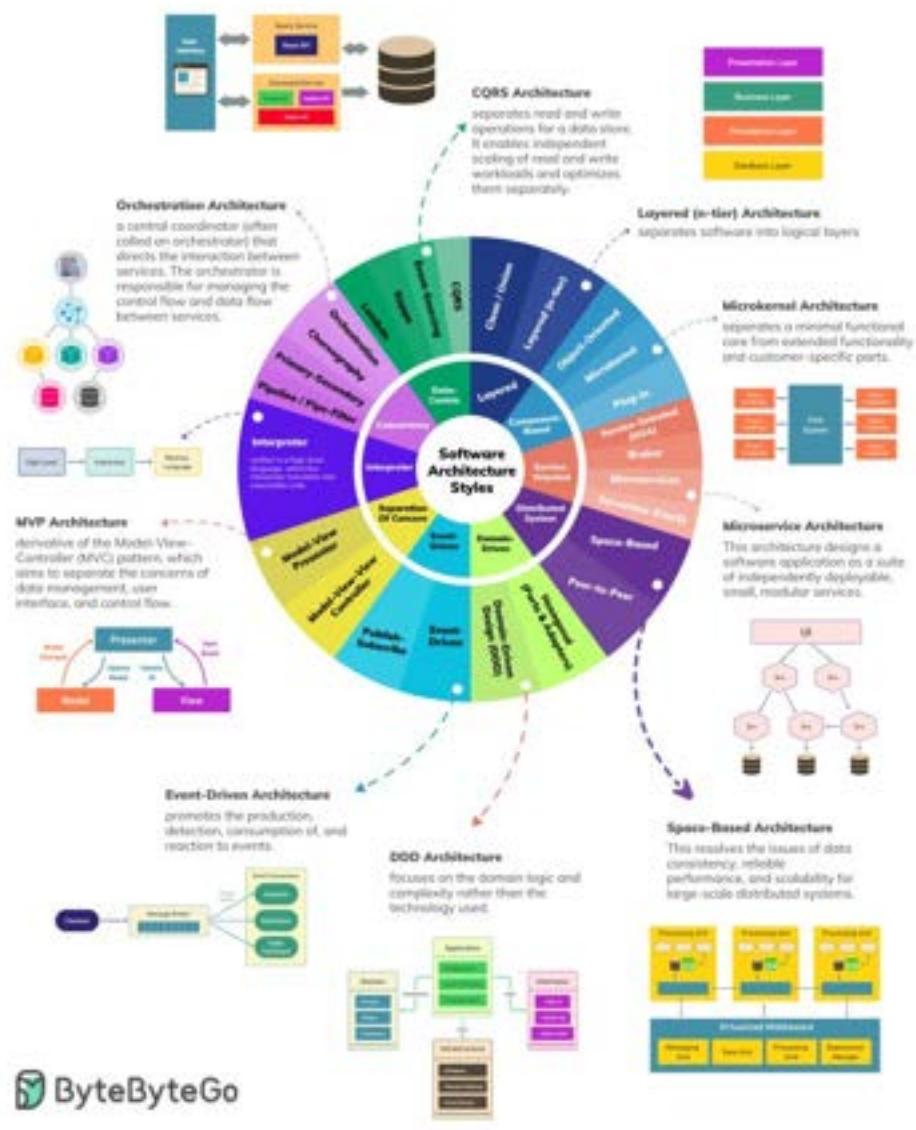
Secure Data Transmission: Navigating the Tunnel — Journey through the encrypted tunnel forged by HTTPS. Learn how your information travels while shielded from cyber threats.

HTML's Role: Peek into HTML's role in structuring the web. Uncover how hyperlinks and content come together seamlessly. And why is it called HYPER TEXT.

Over to you: In this ever-evolving digital landscape, what emerging technologies do you foresee shaping the future of cybersecurity or the web?

Top 5 Software Architectural Patterns

Software Architecture Styles

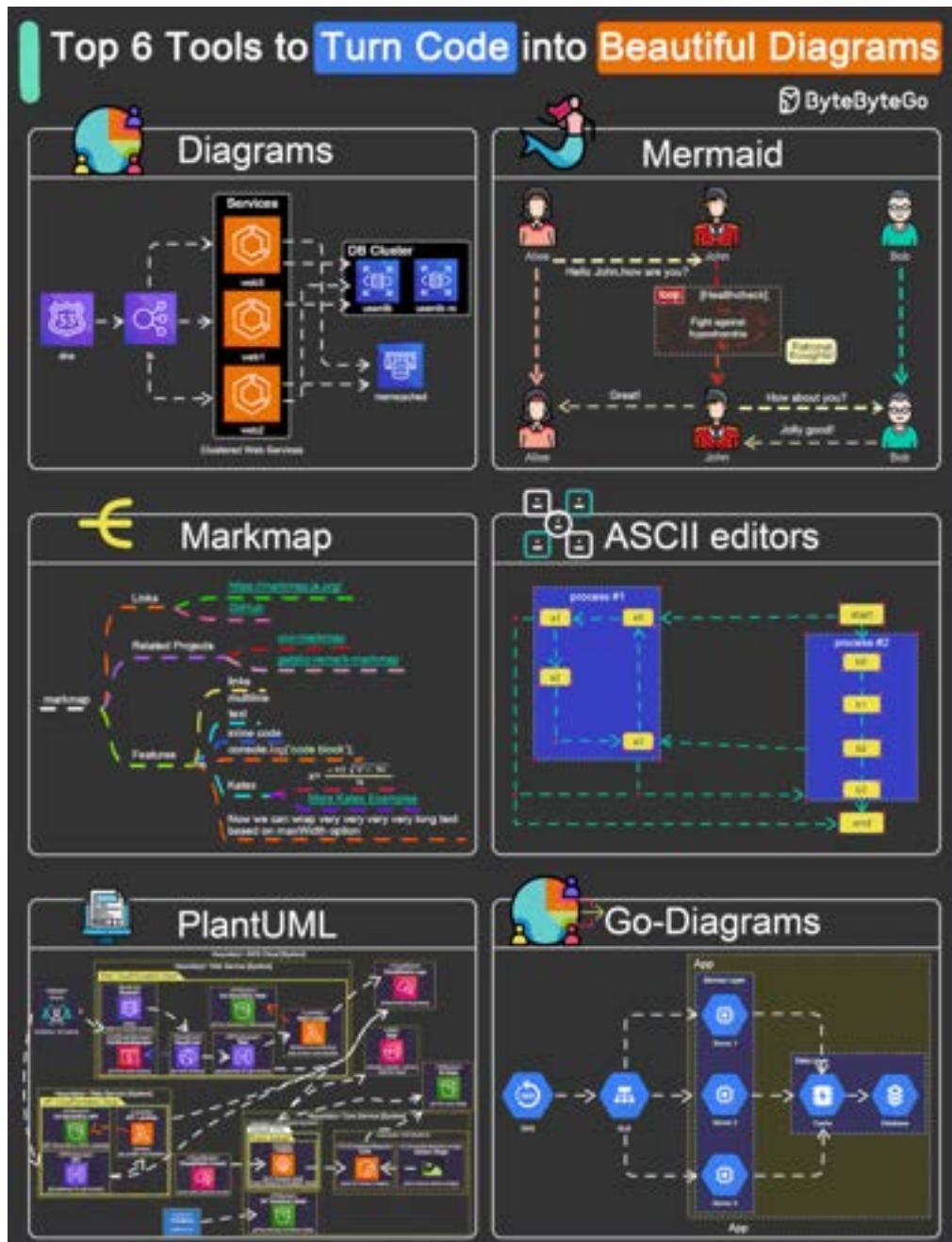


In software development, architecture plays a crucial role in shaping the structure and behavior of software systems. It provides a blueprint for system design, detailing how components interact with each other to deliver specific functionality. They also offer solutions to common problems, saving time and effort and leading to more robust and maintainable systems.

However, with the vast array of architectural styles and patterns available, it can take time to discern which approach best suits a particular project or system. Aims to shed light on these concepts, helping you make informed decisions in your architectural endeavors.

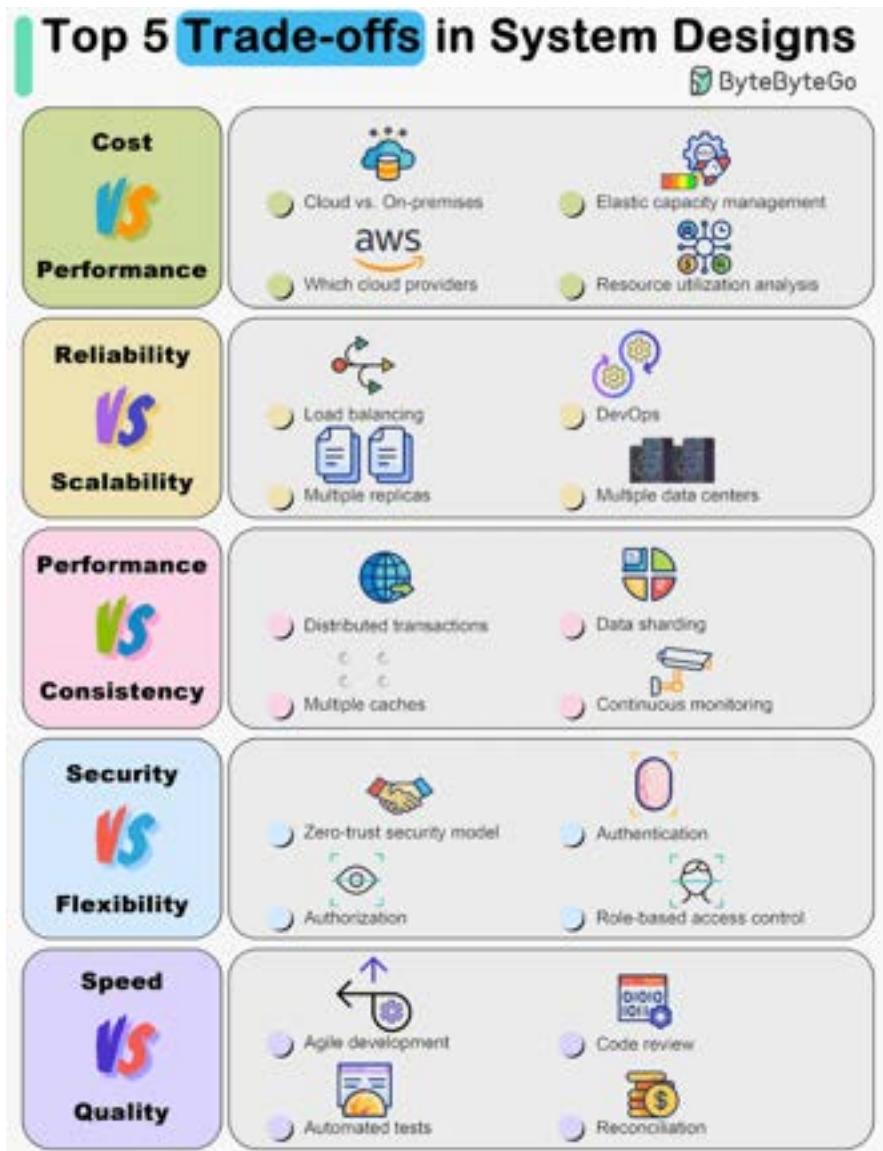
To help you navigate the vast landscape of architectural styles and patterns, there is a cheat sheet that encapsulates all. This cheat sheet is a handy reference guide that you can use to quickly recall the main characteristics of each architectural style and pattern.

Top 6 Tools to Turn Code into Beautiful Diagrams



- Diagrams
- Go Diagrams
- Mermaid
- PlantUML
- ASCII diagrams
- Markmap

Everything is a trade-off.



Everything is a compromise.

There is no right or wrong design.

The diagram below shows some of the most important trade-offs.

- ◆ Cost vs. Performance
- ◆ Reliability vs. Scalability
- ◆ Performance vs. Consistency
- ◆ Security vs. Flexibility
- ◆ Development Speed vs. Quality

Over to you: What trade-offs have you made in the past?

What is DevSecOps?



DevSecOps emerged as a natural evolution of DevOps practices with a focus on integrating security into the software development and deployment process. The term "DevSecOps" represents the convergence of Development (Dev), Security (Sec), and Operations (Ops) practices, emphasizing the importance of security throughout the software development lifecycle.

The diagram below shows the important concepts in DevSecOps.

- 1 . Automated Security Checks
- 2 . Continuous Monitoring
- 3 . CI/CD Automation
- 4 . Infrastructure as Code (IaC)
- 5 . Container Security
- 6 . Secret Management
- 7 . Threat Modeling
- 8 . Quality Assurance (QA) Integration
- 9 . Collaboration and Communication
- 10 . Vulnerability Management

Top 8 Cache Eviction Strategies.



• LRU (Least Recently Used)

LRU eviction strategy removes the least recently accessed items first. This approach is based on the principle that items accessed recently are more likely to be accessed again in the near future.

• MRU (Most Recently Used)

Contrary to LRU, the MRU algorithm removes the most recently used items first. This strategy can be useful in scenarios where the most recently accessed items are less likely to be accessed again soon.

- ◆ SLRU (Segmented LRU)

SLRU divides the cache into two segments: a probationary segment and a protected segment. New items are initially placed into the probationary segment. If an item in the probationary segment is accessed again, it is promoted to the protected segment.

- ◆ LFU (Least Frequently Used)

LFU algorithm evicts the items with the lowest access frequency.

- ◆ FIFO (First In First Out)

FIFO is one of the simplest caching strategies, where the cache behaves in a queue-like manner, evicting the oldest items first, regardless of their access patterns or frequency.

- ◆ TTL (Time-to-Live)

While not strictly an eviction algorithm, TTL is a strategy where each cache item is given a specific lifespan.

- ◆ Two-Tiered Caching

In Two-Tiered Caching strategy, we use an in-memory cache for the first layer and a distributed cache for the second layer.

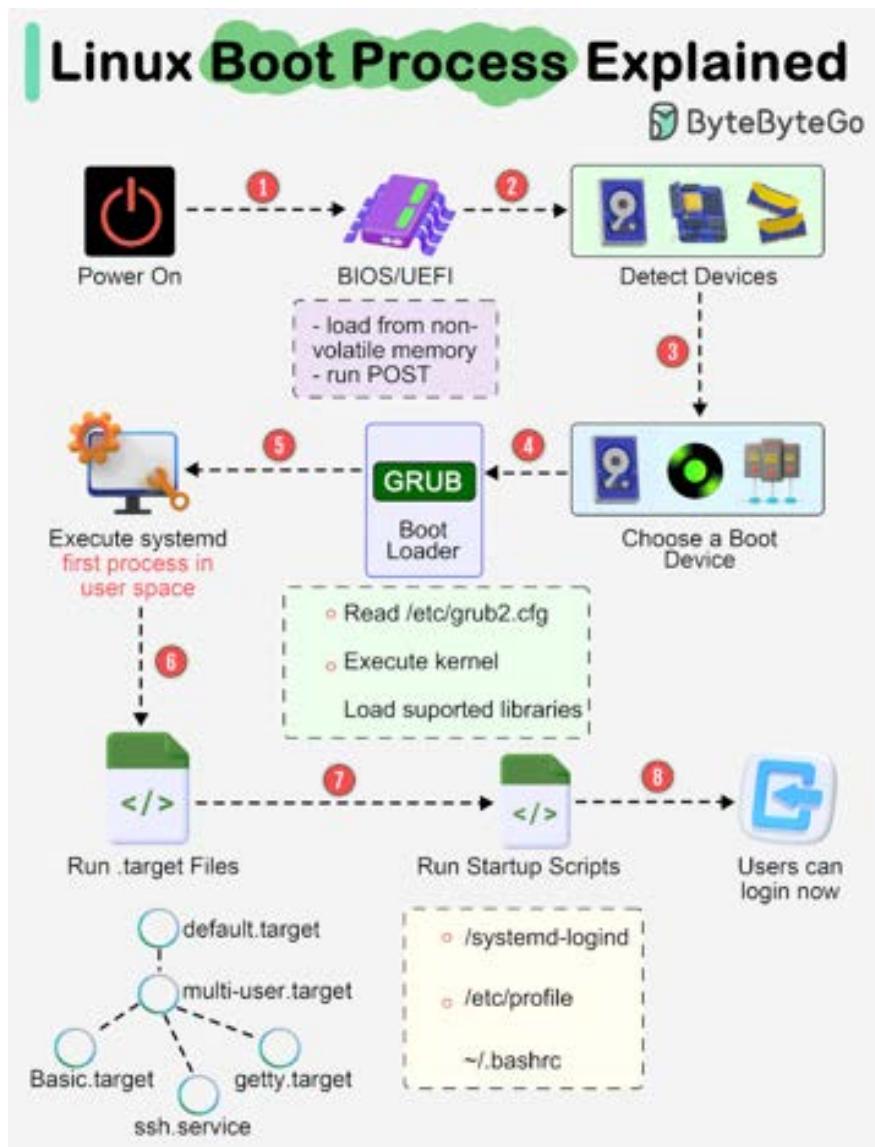
- ◆ RR (Random Replacement)

Random Replacement algorithm randomly selects a cache item and evicts it to make space for new items. This method is also simple to implement and does not require tracking access patterns or frequencies.

Linux Boot Process Explained

Almost every software engineer has used Linux before, but only a handful know how its Boot Process works :) Let's dive in.

The diagram below shows the steps.



Step 1 - When we turn on the power, BIOS (Basic Input/Output System) or UEFI (Unified Extensible Firmware Interface) firmware is loaded from non-volatile memory, and executes POST (Power On Self Test).

Step 2 - BIOS/UEFI detects the devices connected to the system, including CPU, RAM, and storage.

Step 3 - Choose a booting device to boot the OS from. This can be the hard drive, the network server, or CD ROM.

Step 4 - BIOS/UEFI runs the boot loader (GRUB), which provides a menu to choose the OS or the kernel functions.

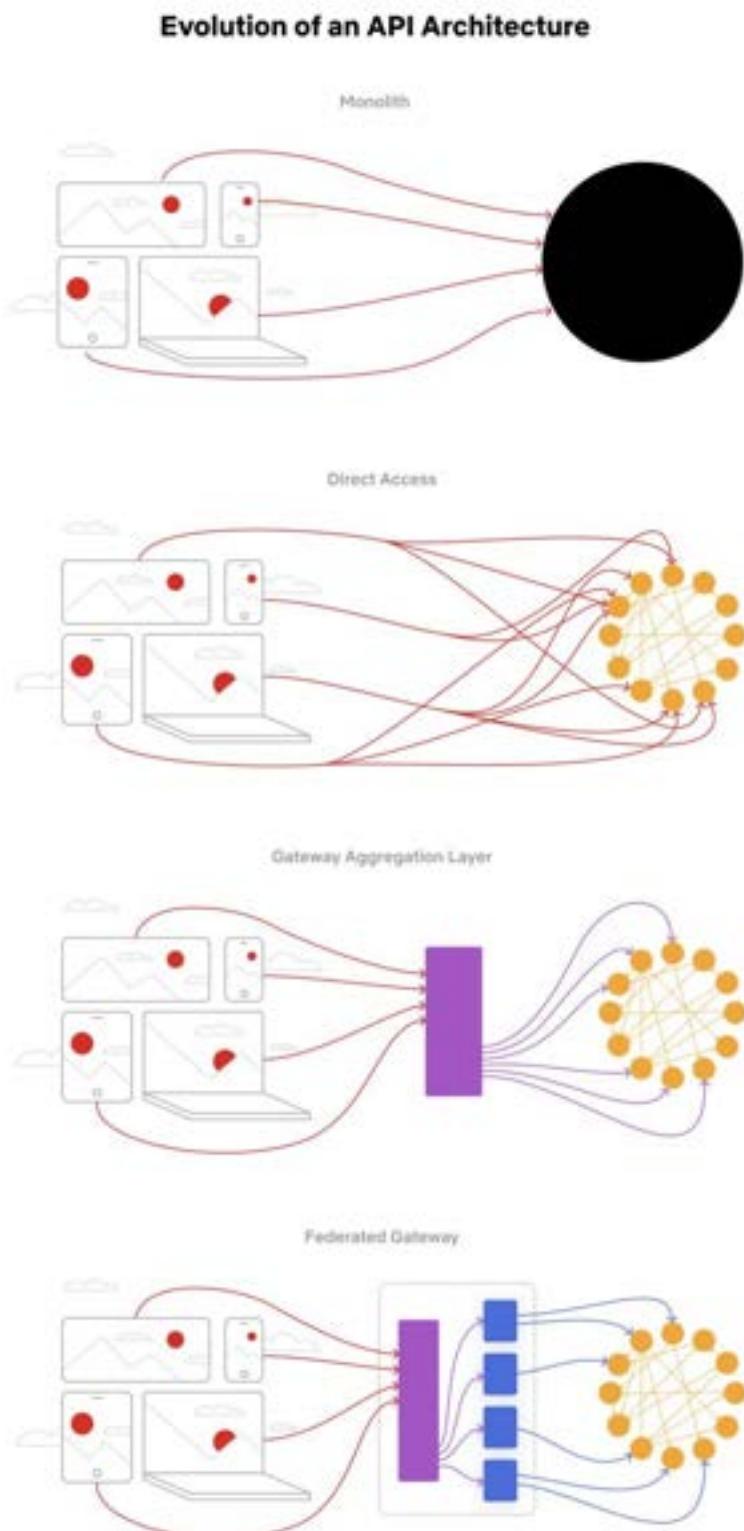
Step 5 - After the kernel is ready, we now switch to the user space. The kernel starts up systemd as the first user-space process, which manages the processes and services, probes all remaining hardware, mounts filesystems, and runs a desktop environment.

Step 6 - systemd activates the default. target unit by default when the system boots. Other analysis units are executed as well.

Step 7 - The system runs a set of startup scripts and configure the environment.

Step 8 - The users are presented with a login window. The system is now ready.

Unusual Evolution of the Netflix API Architecture



The Netflix API architecture went through 4 main stages.

Monolith. The application is packaged and deployed as a monolith, such as a single Java WAR file, Rails app, etc. Most startups begin with a monolith architecture.

Direct access. In this architecture, a client app can make requests directly to the microservices. With hundreds or even thousands of microservices, exposing all of them to clients is not ideal.

Gateway aggregation layer. Some use cases may span multiple services, we need a gateway aggregation layer. Imagine the Netflix app needs 3 APIs (movie, production, talent) to render the frontend. The gateway aggregation layer makes it possible.

Federated gateway. As the number of developers grew and domain complexity increased, developing the API aggregation layer became increasingly harder. GraphQL federation allows Netflix to set up a single GraphQL gateway that fetches data from all the other APIs.

Over to you - why do you think Netflix uses GraphQL instead of RESTful?

References:

- [1] How Netflix Scales its API with GraphQL Federation: bit.ly/3MPuAsi (image source)
- [2] Why You Can't Talk About Microservices Without Mentioning Netflix: bit.ly/3LKn0On

GET, POST, PUT... Common HTTP “verbs” in one figure



1. HTTP GET

This retrieves a resource from the server. It is idempotent. Multiple identical requests return the same result.

2. HTTP PUT

This updates or Creates a resource. It is idempotent. Multiple identical requests will update the same resource.

3. HTTP POST

This is used to create new resources. It is not idempotent, making two identical POST will duplicate the resource creation.

4. HTTP DELETE

This is used to delete a resource. It is idempotent. Multiple identical requests will delete the same resource.

5. HTTP PATCH

The PATCH method applies partial modifications to a resource.

6. HTTP HEAD

The HEAD method asks for a response identical to a GET request but without the response body.

7. HTTP CONNECT

The CONNECT method establishes a tunnel to the server identified by the target resource.

8. HTTP OPTIONS

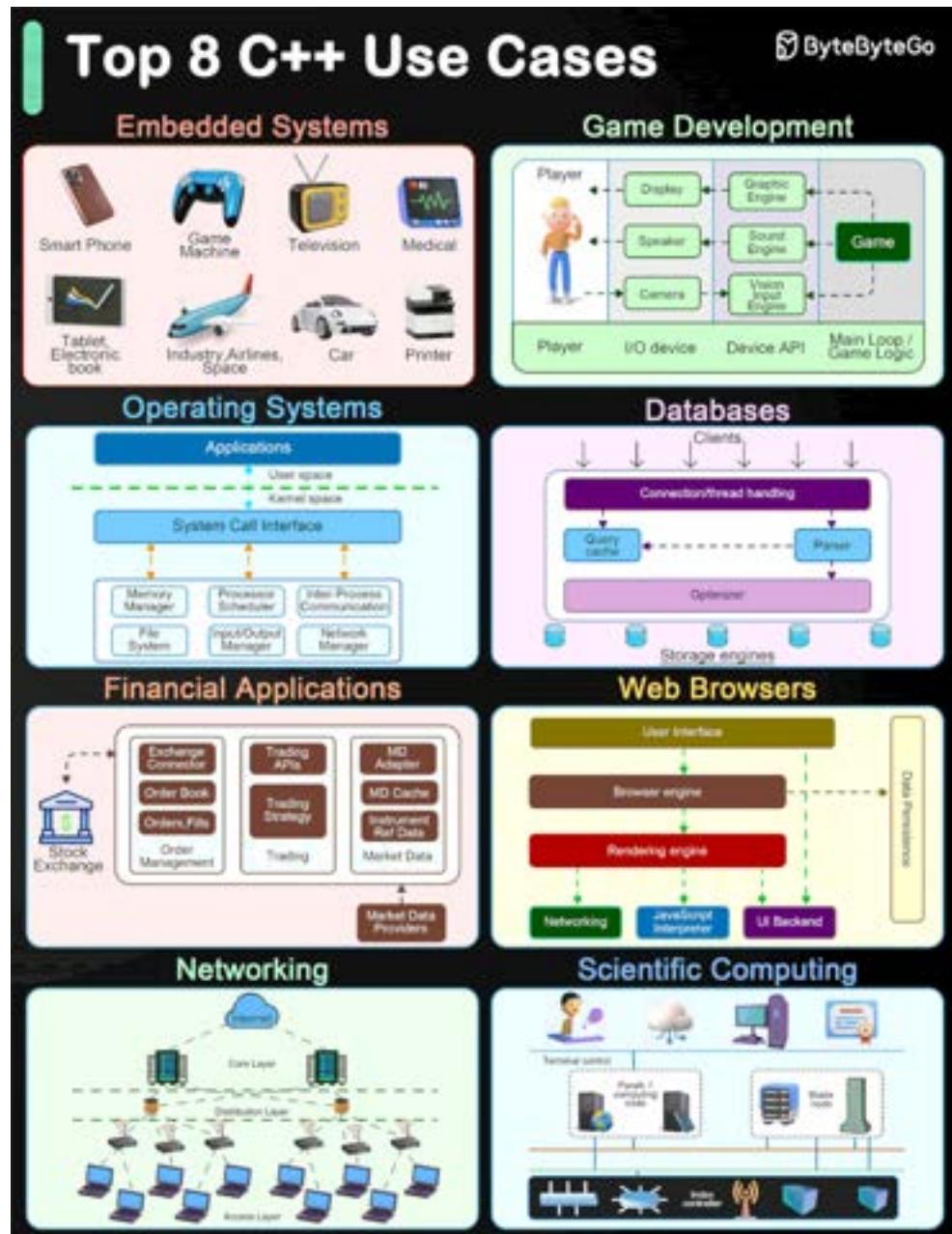
This describes the communication options for the target resource.

9. HTTP TRACE

This performs a message loop-back test along the path to the target resource.

Over to you: What other HTTP verbs have you used?

Top 8 C++ Use Cases



C++ is a highly versatile programming language that is suitable for a wide range of applications.

• Embedded Systems

The language's efficiency and fine control over hardware resources make it excellent for embedded systems development.

• Game Development

C++ is a staple in the game development industry due to its performance and efficiency.

◆ Operating Systems

C++ provides extensive control over system resources and memory, making it ideal for developing operating systems and low-level system utilities.

◆ Databases

Many high-performance database systems are implemented in C++ to manage memory efficiently and ensure fast execution of queries.

◆ Financial Applications

◆ Web Browsers

C++ is used in the development of web browsers and their components, such as rendering engines.

◆ Networking

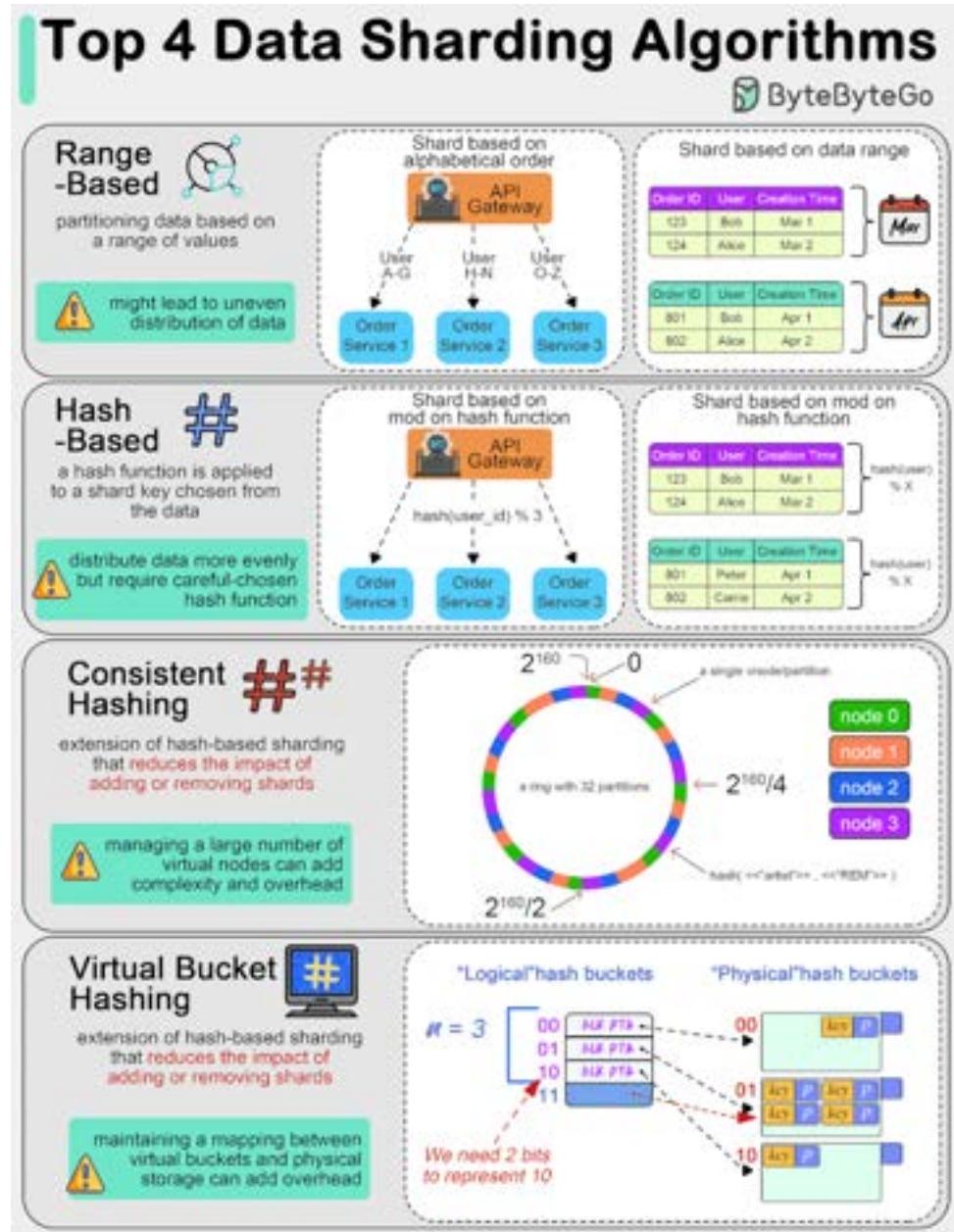
C++ is often used for developing network devices and simulation tools.

◆ Scientific Computing

C++ finds extensive use in scientific computing and engineering applications that require high performance and precise control over computational resources.

Over to you - What did we miss?

Top 4 data sharding algorithms explained.



We are dealing with massive amounts of data. Often we need to split data into smaller, more manageable pieces, or “shards”. Here are some of the top data sharding algorithms commonly used:

Range-Based Sharding

This involves partitioning data based on a range of values. For example, customer data can be sharded based on alphabetical order of last names, or transaction data can be sharded based on date ranges.

- ◆ Hash-Based Sharding

In this method, a hash function is applied to a shard key chosen from the data (like a customer ID or transaction ID).

This tends to distribute data more evenly across shards compared to range-based sharding. However, we need to choose a proper hash function to avoid hash collision.

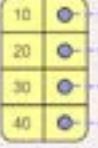
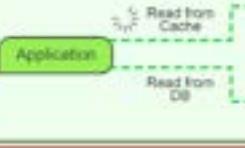
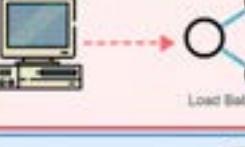
- ◆ Consistent Hashing

This is an extension of hash-based sharding that reduces the impact of adding or removing shards. It distributes data more evenly and minimizes the amount of data that needs to be relocated when shards are added or removed.

- ◆ Virtual Bucket Sharding

Data is mapped into virtual buckets, and these buckets are then mapped to physical shards. This two-level mapping allows for more flexible shard management and rebalancing without significant data movement.

10 years ago, Amazon found that every 100ms of latency cost them 1% in sales.

Top 5 Strategies to Reduce Latency		
ByteByteGo		
Process	Illustration	How?
Database Indexing		<ul style="list-style-type: none"> 1. Make sure to create the right indexes 2. Optimize & refactor slow running queries
Caching		<ul style="list-style-type: none"> 1. Store frequently accessed data in a cache 2. Minimize costly database lookups
Load Balancing		<ul style="list-style-type: none"> 1. Distribute requests evenly between multiple servers 2. Use the right LB type and the appropriate algorithm
Content Delivery Networks		<ul style="list-style-type: none"> 1. Cache static content near the end-users 2. Reduce geographic distance to reduce latency
Async Processing		<ul style="list-style-type: none"> 1. Don't block the request flow for long running tasks 2. Execute such tasks in the background
Data Compression		<ul style="list-style-type: none"> 1. Compress data before sending over the network 2. Use libraries like gzip and zlib

That's a staggering \$5.7 billion in today's terms.

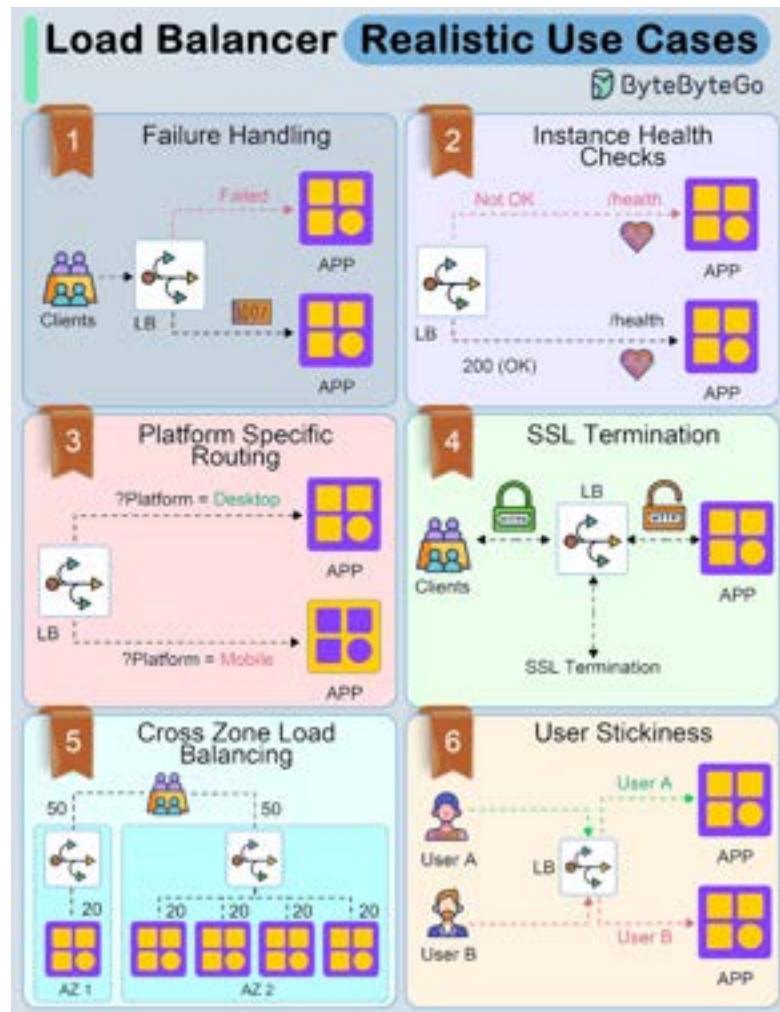
For high-scale user-facing systems, high latency is a big loss of revenue.

Here are the top strategies to reduce latency:

- 1 - Database Indexing
- 2 - Caching
- 3 - Load Balancing
- 4 - Content Delivery Network
- 5 - Async Processing
- 6 - Data Compression

Over to you: What other strategies to reduce latency have you seen?

Load Balancer Realistic Use Cases You May Not Know



Load balancers are inherently dynamic and adaptable, designed to efficiently address multiple purposes and use cases in network traffic and server workload management.

Let's explore some of the use cases:

1. Failure Handling:

Automatically redirects traffic away from malfunctioning elements to maintain continuous service and reduce service interruptions.

2. Instance Health Checks:

Continuously evaluates the functionality of instances, directing incoming requests exclusively to those that are fully operational and efficient.

3. Platform Specific Routing:

Routes requests from different device types (like mobiles, desktops) to specialized backend systems, providing customized responses based on platform.

4. SSL Termination:

Handles the encryption and decryption of SSL traffic, reducing the processing burden on backend infrastructure.

5. Cross Zone Load Balancing:

Distributes incoming traffic across various geographic or network zones, increasing the system's resilience and capacity for handling large volumes of requests.

6. User Stickiness:

Maintains user session integrity and tailored user interactions by consistently directing requests from specific users to designated backend servers.

Over to you:

Which of these use cases would you consider adding to your network to enhance system reliability and why?

25 Papers That Completely Transformed the Computer World.

25 Papers That Completely Transformed the Computer World

ByteByteGo

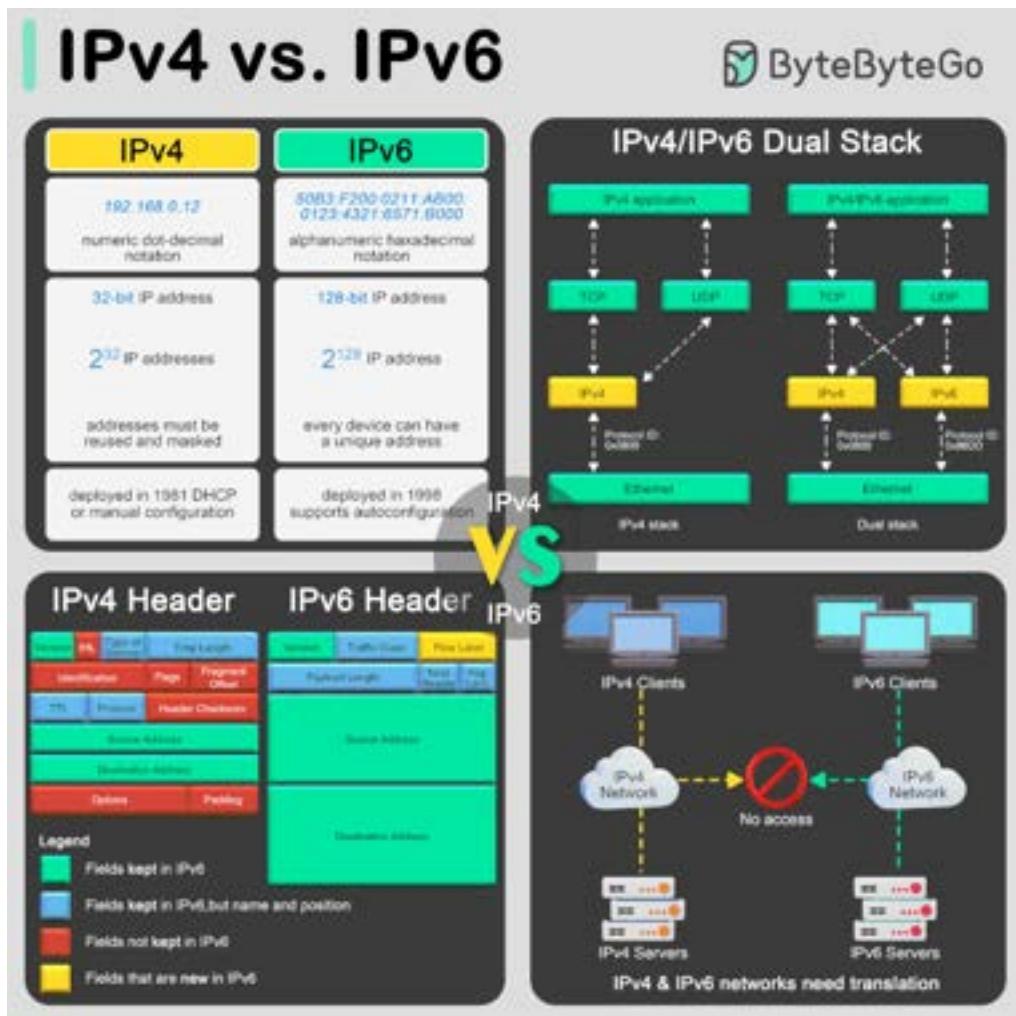
- [Dynamo](#): Amazon's Highly Available Key Value Store
- [Google File System](#): Insights into a highly scalable file system
- [Scaling Memcached at Facebook](#): A look at the complexities of Caching at a cluster
- [BigTable](#): The design principles behind a distributed storage system for structured data
- [Borg](#): Large Scale Cluster Management at Google
- [Cassandra](#): A look at the design and architecture of a distributed NoSQL database
- [Attention Is All You Need](#): Into a new deep learning architecture known as the transformer
- [Kafka](#): Internals of the distributed messaging platform
- [FoundationDB](#): A look at how a distributed database
- [Amazon Aurora](#): To learn how Amazon provides high-availability and performance
- [Spanner](#): Design and architecture of Google's globally distributed database
- [MapReduce](#): A detailed look at how MapReduce enables parallel processing
- [Shard Manager](#): Understanding the generic shard management framework
- [Dapper](#): Insights into Google's distributed systems tracing infrastructure
- [Flink](#): A detailed look at the unified architecture of stream and batch data processing
- [A Comprehensive Survey on Vector Databases](#): Algorithms powering the vector database
- [Zanzibar](#): A global system for managing access control lists at Google
- [Monarch](#): Architecture of Google's globally distributed in-memory time series database
- [Thrift](#): Explore the design choices behind Facebook's code-generation tool
- [Bitcoin](#): The ground-breaking introduction to the peer-to-peer electronic cash system
- [WTF](#): Who to Follow Service at Twitter
- [MyRocks](#): LSM-Tree Database Storage Engine
- [GoTo Considered Harmful](#): Understand why GoTo statements are problematic
- [Raft Consensus Algorithm](#): To learn about the more understandable consensus algorithm
- [Time Clocks and Ordering of Events](#): Explains the concept of time and event ordering

1. Dynamo - Amazon's Highly Available Key Value Store
2. Google File System: Insights into a highly scalable file system
3. Scaling Memcached at Facebook: A look at the complexities of Caching
4. BigTable: The design principles behind a distributed storage system
5. Borg - Large Scale Cluster Management at Google

6. Cassandra: A look at the design and architecture of a distributed NoSQL database
 7. Attention Is All You Need: Into a new deep learning architecture known as the transformer
 8. Kafka: Internals of the distributed messaging platform
 9. FoundationDB: A look at how a distributed database
 10. Amazon Aurora: To learn how Amazon provides high-availability and performance
 11. Spanner: Design and architecture of Google's globally distributed database
 12. MapReduce: A detailed look at how MapReduce enables parallel processing of massive volumes of data
 13. Shard Manager: Understanding the generic shard management framework
 14. Dapper: Insights into Google's distributed systems tracing infrastructure
 15. Flink: A detailed look at the unified architecture of stream and batch processing
 16. A Comprehensive Survey on Vector Databases
 17. Zanzibar: A look at the design, implementation and deployment of a global system for managing access control lists at Google
 18. Monarch: Architecture of Google's in-memory time series database
-
19. Thrift: Explore the design choices behind Facebook's code-generation tool
 20. Bitcoin: The ground-breaking introduction to the peer-to-peer electronic cash system
 21. WTF - Who to Follow Service at Twitter: Twitter's (now X) user recommendation system
 22. MyRocks: LSM-Tree Database Storage Engine
 23. GoTo Considered Harmful
 24. Raft Consensus Algorithm: To learn about the more understandable consensus algorithm
 25. Time Clocks and Ordering of Events: The extremely important paper that explains the concept of time and event ordering in a distributed system

Over to you: I'm sure we missed many important papers. Which ones do you think should be included?

IPv4 vs. IPv6, what are the differences?



The transition from Internet Protocol version 4 (IPv4) to Internet Protocol version 6 (IPv6) is primarily driven by the need for more internet addresses, alongside the desire to streamline certain aspects of network management.

Format and Length

IPv4 uses a 32-bit address format, which is typically displayed as four decimal numbers separated by dots (e.g., 192.168.0. 12). The 32-bit format allows for approximately 4.3 billion unique addresses, a number that is rapidly proving insufficient due to the explosion of internet-connected devices.

In contrast, IPv6 utilizes a 128-bit address format, represented by eight groups of four hexadecimal digits separated by colons (e.g., 50B3:F200:0211:AB00:0123:4321:6571:B000). This expansion allows for approximately much more addresses, ensuring the internet's growth can continue unabated.

◆ Header

The IPv4 header is more complex and includes fields such as the header length, service type, total length, identification, flags, fragment offset, time to live (TTL), protocol, header checksum, source and destination IP addresses, and options.

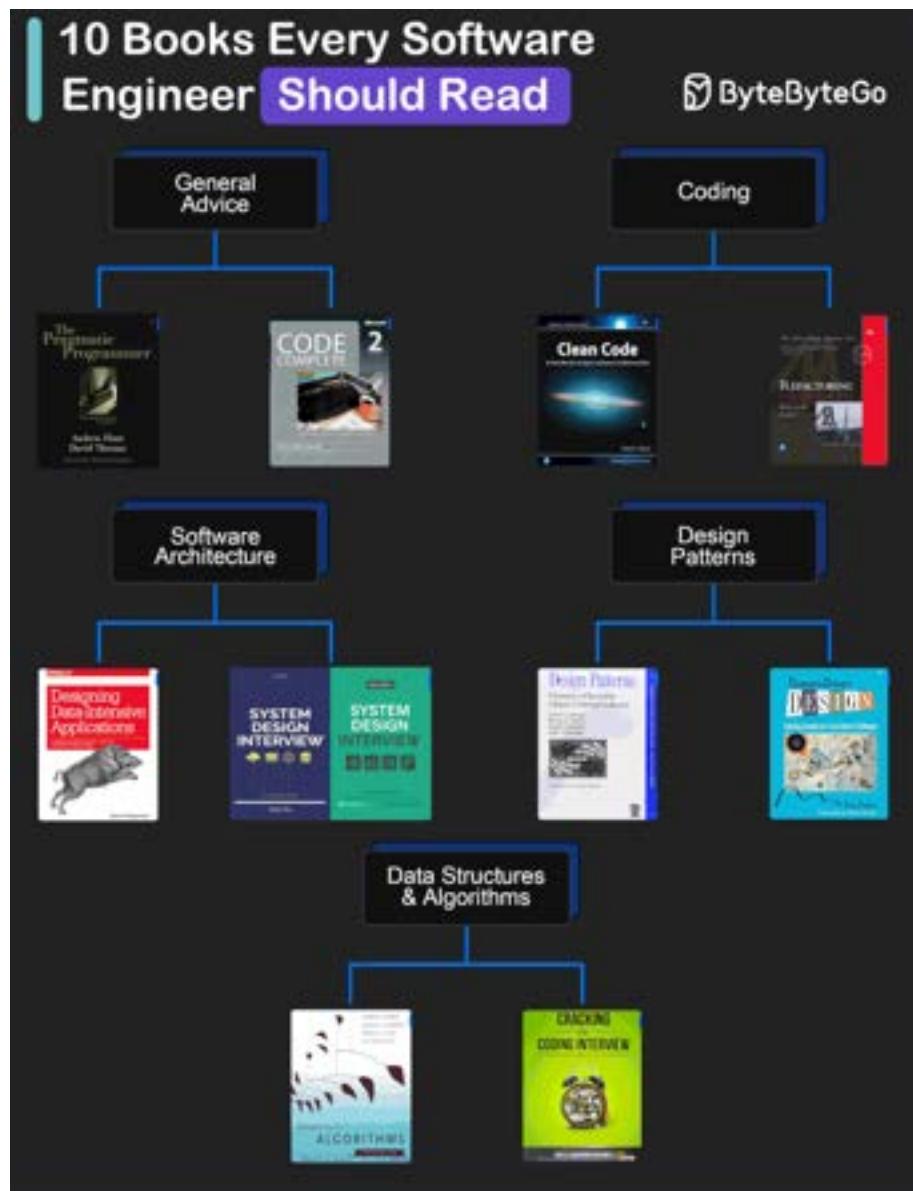
IPv6 headers are designed to be simpler and more efficient. The fixed header size is 40 bytes and includes less frequently used fields in optional extension headers. The main fields include version, traffic class, flow label, payload length, next header, hop limit, and source and destination addresses. This simplification helps improve packet processing speeds.

◆ Translation between IPv4 and IPv6

As the internet transitions from IPv4 to IPv6, mechanisms to allow these protocols to coexist have become essential:

- Dual Stack: This technique involves running IPv4 and IPv6 simultaneously on the same network devices. It allows seamless communication in both protocols, depending on the destination address availability and compatibility. The dual stack is considered one of the best approaches for the smooth transition from IPv4 to IPv6.

My Favorite 10 Books for Software Developers



General Advice

- 1 - The Pragmatic Programmer by Andrew Hunt and David Thomas
- 2 - Code Complete by Steve McConnell: Often considered a bible for software developers, this comprehensive book covers all aspects of software development, from design and coding to testing and maintenance.

Coding

- 1 - Clean Code by Robert C. Martin
- 2 - Refactoring by Martin Fowler

Software Architecture

- 1 - Designing Data-Intensive Applications by Martin Kleppmann
- 2 - System Design Interview (our own book :))

Design Patterns

- 1 - Design Patterns by Eric Gamma and Others
- 2 - Domain-Driven Design by Eric Evans

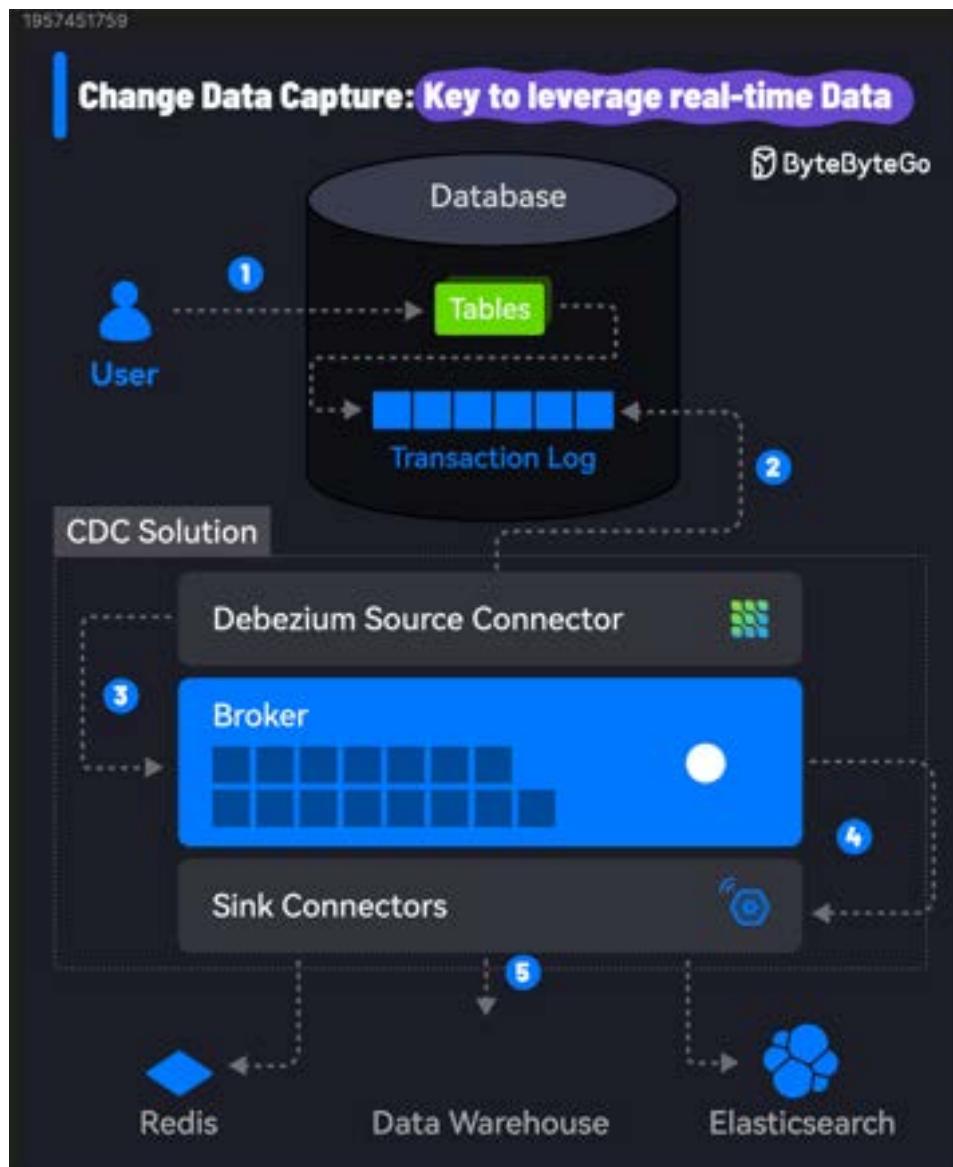
Data Structures and Algorithms

- 1 - Introduction to Algorithms by Cormen, Leiserson, Rivest, and Stein
- 2 - Cracking the Coding Interview by Gayle Laakmann McDowell

Over to you: What is your favorite book?

Change Data Capture: key to leverage real-time Data

90% of the world's data was created in the last two years and this growth will only get faster.



However, the biggest challenge is to leverage this data in real-time. Constant data changes make databases, data lakes, and data warehouses out of sync.

CDC or Change Data Capture can help you overcome this challenge.

CDC identifies and captures changes made to the data in a database, allowing you to replicate and sync data across multiple systems.

So, how does Change Data Capture work? Here's a step-by-step breakdown:

1 - Data Modification: A change is made to the data in the source database. It could be an insert, update, or delete operation on a table.

2 - Change Capture: A CDC tool monitors the database transaction logs to capture the modifications. It uses the source connector to connect to the database and read the logs.

3 - Change Processing: The captured changes are processed and transformed into a format suitable for the downstream systems.

4 - Change Propagation: The processed changes are published to a message queue and propagated to the target systems, such as data warehouses, analytics platforms, distributed caches like Redis, and so on.

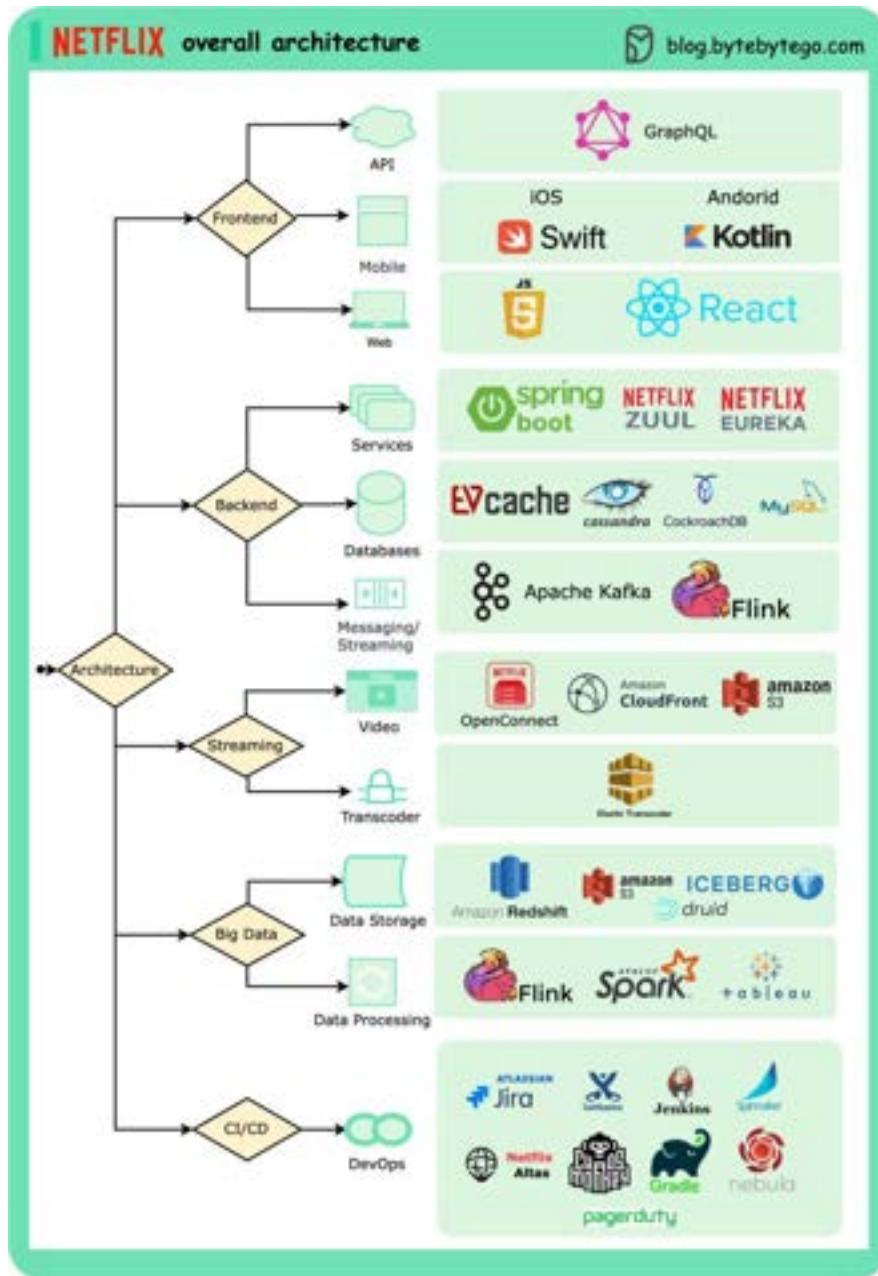
5 - Real-Time Integration: The CDC tool uses its sink connector to consume the log and update the target systems. The changes are received in real time, allowing for conflict-free data analysis and decision-making.

Users only need to take care of step 1 while all other steps are transparent.

A popular CDC solution uses Debezium with Kafka Connect to stream data changes from the source to target systems using Kafka as the broker. Debezium has connectors for most databases such as MySQL, PostgreSQL, Oracle, etc.

Over to you: have you leveraged CDC in your application before?

Netflix's Overall Architecture



This post is based on research from many Netflix engineering blogs and open-source projects. If you come across any inaccuracies, please feel free to inform us.

Mobile and web: Netflix has adopted Swift and Kotlin to build native mobile apps. For its web application, it uses React.

Frontend/server communication: Netflix uses GraphQL.

Backend services: Netflix relies on ZUUL, Eureka, the Spring Boot framework, and other technologies.

Databases: Netflix utilizes EV cache, Cassandra, CockroachDB, and other databases.

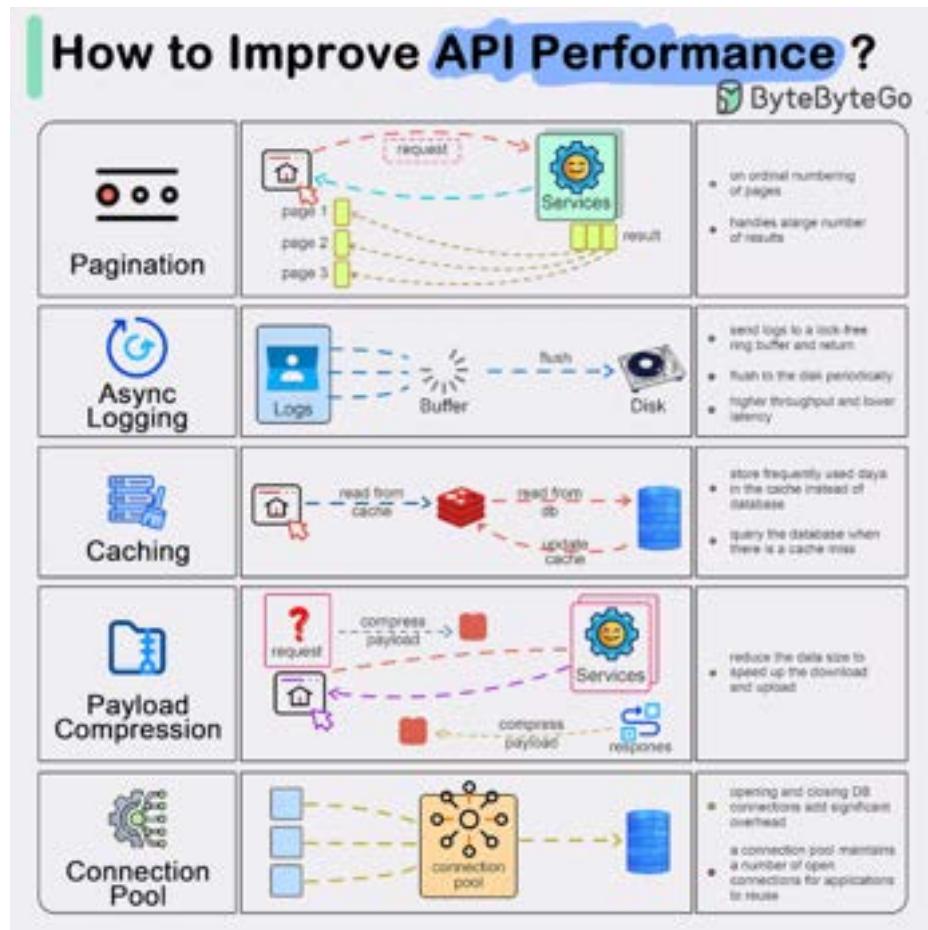
Messaging/streaming: Netflix employs Apache Kafka and Flink for messaging and streaming purposes.

Video storage: Netflix uses S3 and Open Connect for video storage.

Data processing: Netflix utilizes Flink and Spark for data processing, which is then visualized using Tableau. Redshift is used for processing structured data warehouse information.

CI/CD: Netflix employs various tools such as JIRA, Confluence, PagerDuty, Jenkins, Gradle, Chaos Monkey, Spinnaker, Altas, and more for CI/CD processes.

Top 5 common ways to improve API performance.



Result Pagination:

This method is used to optimize large result sets by streaming them back to the client, enhancing service responsiveness and user experience.

Asynchronous Logging:

This approach involves sending logs to a lock-free buffer and returning immediately, rather than dealing with the disk on every call. Logs are periodically flushed to the disk, significantly reducing I/O overhead.

Data Caching:

Frequently accessed data can be stored in a cache to speed up retrieval. Clients check the cache before querying the database, with data storage solutions like Redis offering faster access due to in-memory storage.

Payload Compression:

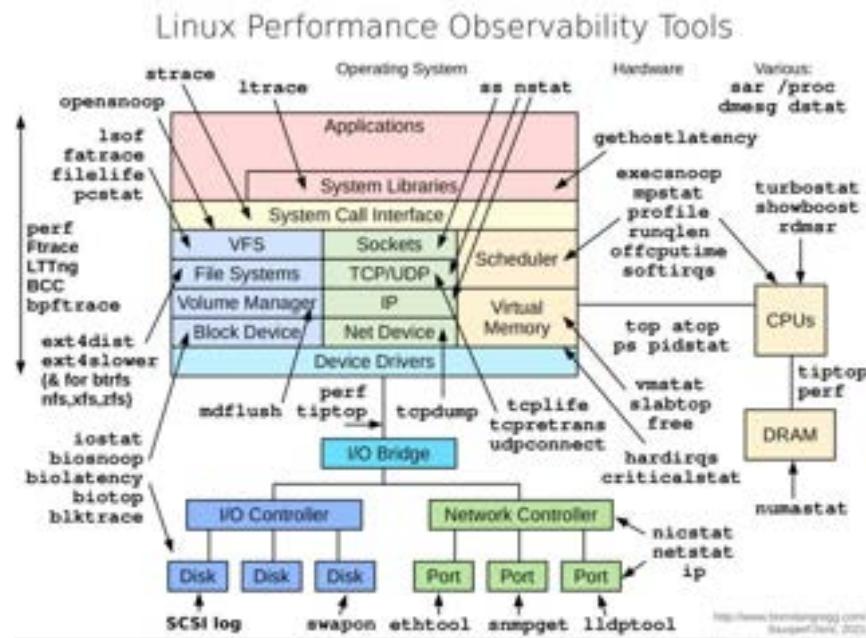
To reduce data transmission time, requests and responses can be compressed (e.g., using gzip), making the upload and download processes quicker.

Connection Pooling:

This technique involves using a pool of open connections to manage database interaction, which reduces the overhead associated with opening and closing connections each time data needs to be loaded. The pool manages the lifecycle of connections for efficient resource use.

Over to you: What other ways do you use to improve API performance?

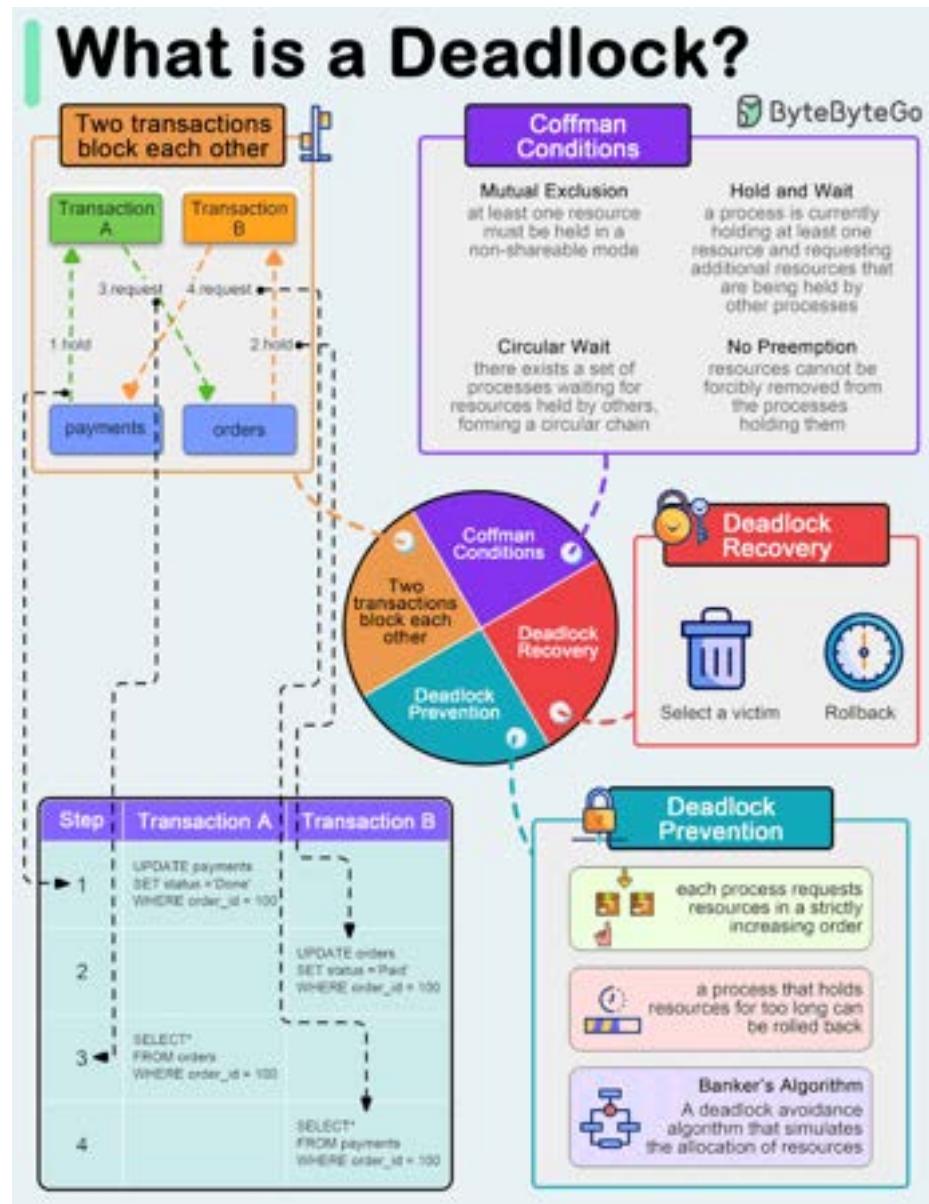
Popular interview question: how to diagnose a mysterious process that's taking too much CPU, memory, IO, etc?



The diagram below illustrates helpful tools in a Linux system.

- 'vmstat' - reports information about processes, memory, paging, block IO, traps, and CPU activity.
- 'iostat' - reports CPU and input/output statistics of the system.
- 'netstat' - displays statistical data related to IP, TCP, UDP, and ICMP protocols.
- 'lsof' - lists open files of the current system.
- 'pidstat' - monitors the utilization of system resources by all or specified processes, including CPU, memory, device IO, task switching, threads, etc.

What is a deadlock?



A deadlock occurs when two or more transactions are waiting for each other to release locks on resources they need to continue processing. This results in a situation where neither transaction can proceed, and they end up waiting indefinitely.

◆ Coffman Conditions

The Coffman conditions, named after Edward G. Coffman, Jr., who first outlined them in 1971, describe four necessary conditions that must be present simultaneously for a deadlock to occur:

- Mutual Exclusion

- Hold and Wait
- No Preemption
- Circular Wait

◆ Deadlock Prevention

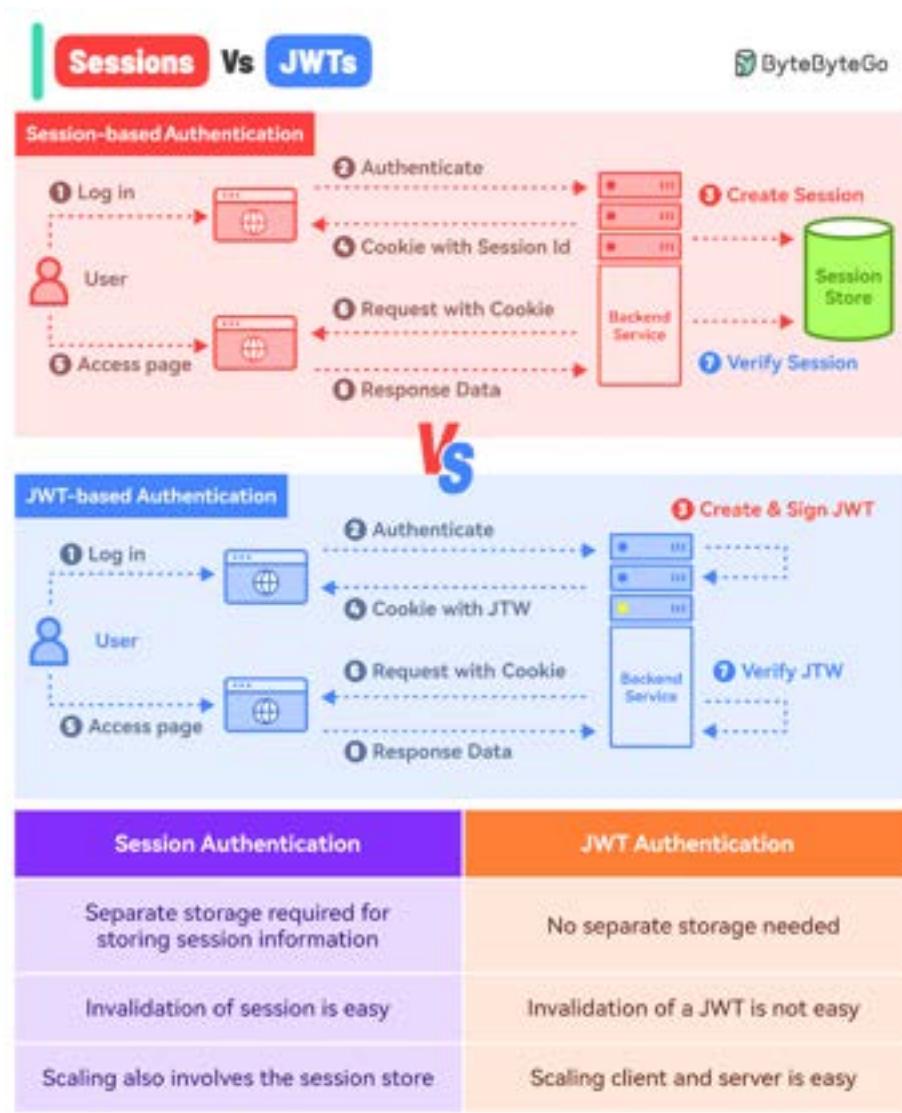
- Resource ordering: impose a total ordering of all resource types, and require that each process requests resources in a strictly increasing order.
- Timeouts: A process that holds resources for too long can be rolled back.
- Banker's Algorithm: A deadlock avoidance algorithm that simulates the allocation of resources to processes and helps in deciding whether it is safe to grant a resource request based on the future availability of resources, thus avoiding unsafe states.

◆ Deadlock Recovery

- Selecting a victim: Most modern Database Management Systems (DBMS) and Operating Systems implement sophisticated algorithms for detecting deadlocks and selecting victims, often allowing customization of the victim selection criteria via configuration settings. The selection can be based on resource utilization, transaction priority, cost of rollback etc.
- Rollback: The database may roll back the entire transaction or just enough of it to break the deadlock. Rolled-back transactions can be restarted automatically by the database management system.

Over to you: have you solved any tricky deadlock issues?

What's the difference between Session-based authentication and JWTs?



Here's a simple breakdown for both approaches:

Session-Based Authentication

In this approach, you store the session information in a database or session store and hand over a session ID to the user.

Think of it like a passenger getting just the Ticket ID of their flight while all other details are stored in the airline's database.

Here's how it works:

- 1 - The user makes a login request and the frontend app sends the request to the backend server.
- 2 - The backend creates a session using a secret key and stores the data in session storage.
- 3 - The server sends a cookie back to the client with the unique session ID.
- 4 - The user makes a new request and the browser sends the session ID along with the request.
- 5 - The server authenticates the user using the session ID.

JWT-Based Authentication

In the JWT-based approach, you don't store the session information in the session store.

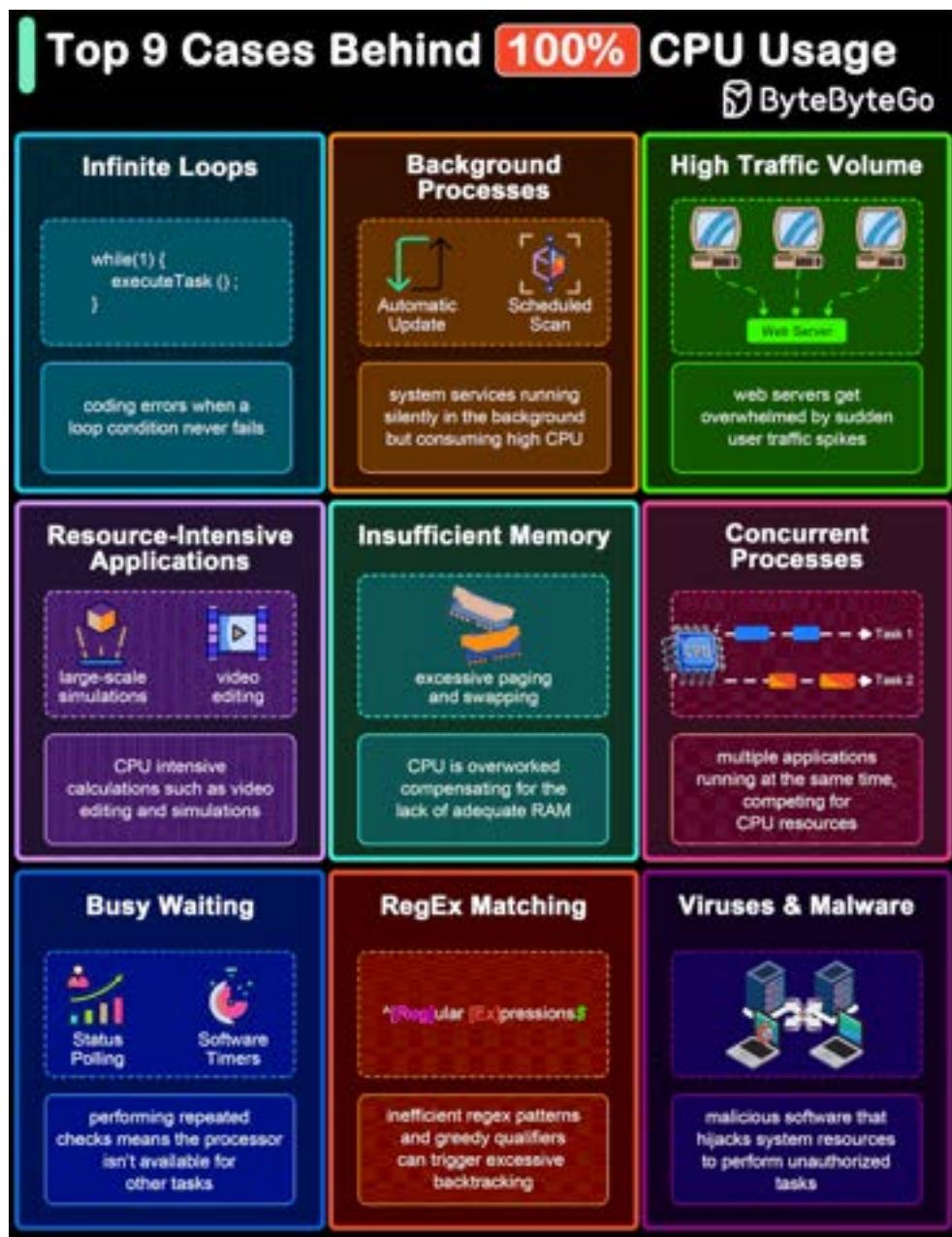
The entire information is available within the token.

Think of it like getting the flight ticket along with all the details available on the ticket but encoded.

Here's how it works:

- 1 - The user makes a login request and it goes to the backend server.
- 2 - The server verifies the credentials and issues a JWT. The JWT is signed using a private key and no session storage is involved.
- 3 - The JWT is passed to the client, either as a cookie or in the response body. Both approaches have their pros and cons but we've gone with the cookie approach.
- 4 - For every subsequent request, the browser sends the cookie with the JWT.
- 5 - The server verifies the JWT using the secret private key and extracts the user info.

Top 9 Cases Behind 100% CPU Usage.



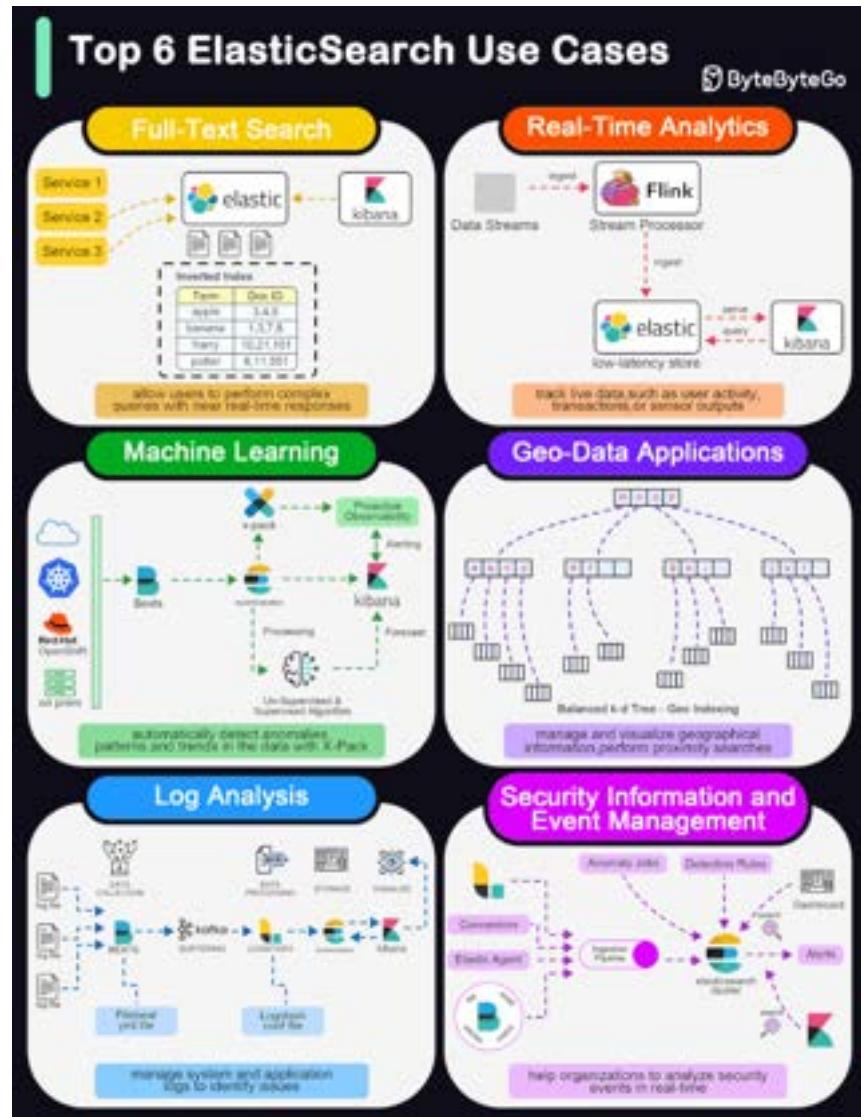
The diagram below shows common culprits that can lead to 100% CPU usage. Understanding these can help in diagnosing problems and improving system efficiency.

1. Infinite Loops
2. Background Processes
3. High Traffic Volume
4. Resource-Intensive Applications

5. Insufficient Memory
6. Concurrent Processes
7. Busy Waiting
8. Regular Expression Matching
9. Malware and Viruses

Over to you: Did we miss anything important?

Top 6 ElasticSearch Use Cases.



Elasticsearch is widely used for its powerful and versatile search capabilities. The diagram below shows the top 6 use cases:

• Full-Text Search

Elasticsearch excels in full-text search scenarios due to its robust, scalable, and fast search capabilities. It allows users to perform complex queries with near real-time responses.

• Real-Time Analytics

Elasticsearch's ability to perform analytics in real-time makes it suitable for dashboards that track live data, such as user activity, transactions, or sensor outputs.

• Machine Learning

With the addition of the machine learning feature in X-Pack, Elasticsearch can automatically detect anomalies, patterns, and trends in the data.

- ◆ **Geo-Data Applications**

Elasticsearch supports geo-data through geospatial indexing and searching capabilities. This is useful for applications that need to manage and visualize geographical information, such as mapping and location-based services.

- ◆ **Log and Event Data Analysis**

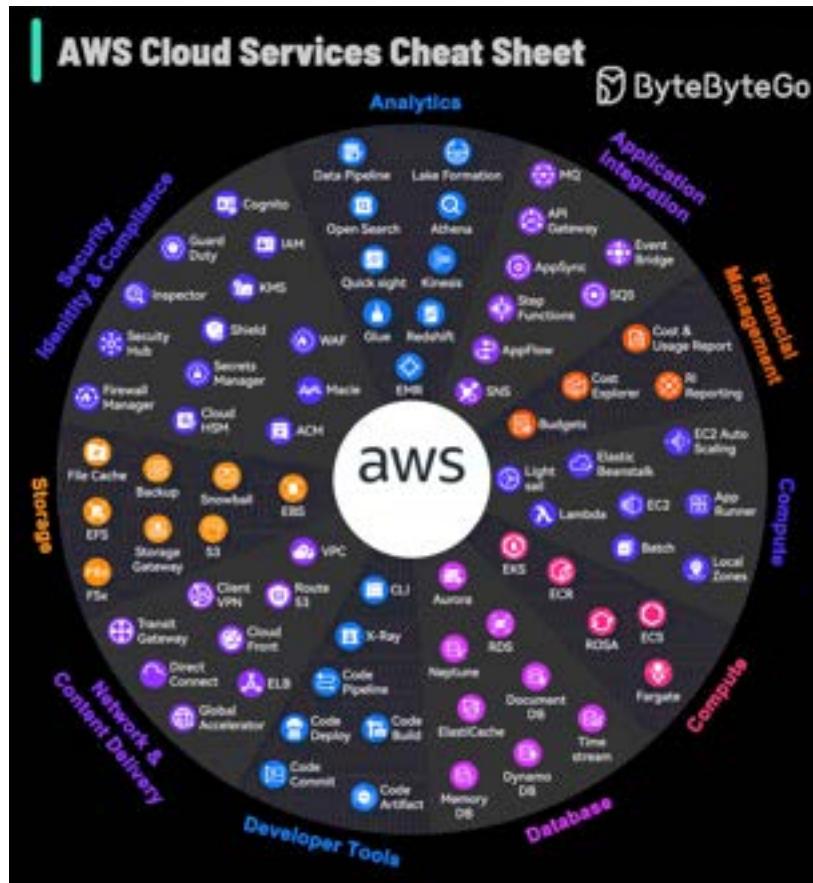
Organizations use Elasticsearch to aggregate, monitor, and analyze logs and event data from various sources. It's a key component of the ELK stack (Elasticsearch, Logstash, Kibana), which is popular for managing system and application logs to identify issues and monitor system health.

- ◆ **Security Information and Event Management (SIEM)**

Elasticsearch can be used as a tool for SIEM, helping organizations to analyze security events in real time.

Over to you: What did we miss?

AWS Services Cheat Sheet



AWS grew from an in-house project to the market leader in cloud services, offering so many different services that even experts can find it a lot to take in.

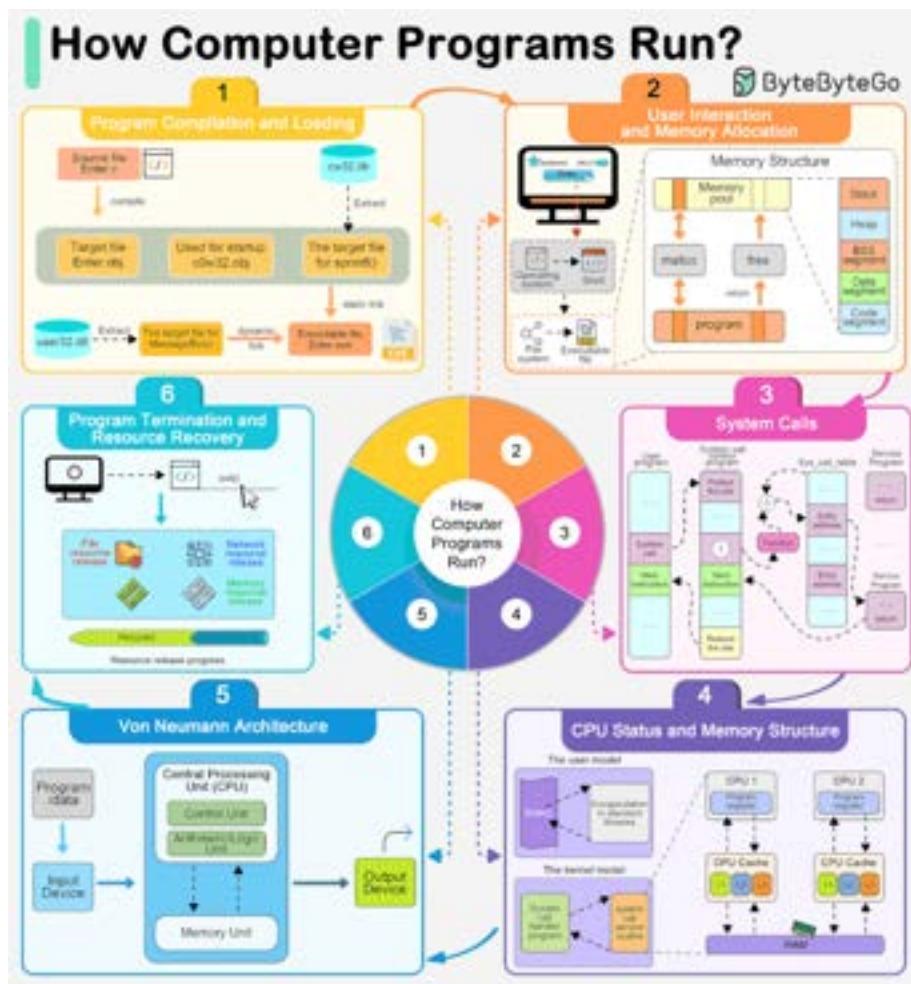
The platform not only caters to foundational cloud needs but also stays at the forefront of emerging technologies such as machine learning and IoT, establishing itself as a bedrock for cutting-edge innovation. AWS continuously refines its array of services, ensuring advanced capabilities for security, scalability, and operational efficiency are available.

For those navigating the complex array of options, this AWS Services Guide is a helpful visual aid.

It simplifies the exploration of AWS's expansive landscape, making it accessible for users to identify and leverage the right tools for their cloud-based endeavors.

Over to you: What improvements would you like to see in AWS services based on your usage?

How do computer programs run?



The diagram shows the steps.

User interaction and command initiation

By double-clicking a program, a user is instructing the operating system to launch an application via the graphical user interface.

Program Preloading

Once the execution request has been initiated, the operating system first retrieves the program's executable file.

The operating system locates this file through the file system and loads it into memory in preparation for execution.

Dependency resolution and loading

Most modern applications rely on a number of shared libraries, such as dynamic link libraries (DLLs).

- ◆ Allocating memory space

The operating system is responsible for allocating space in memory.

- ◆ Initializing the Runtime Environment

After allocating memory, the operating system and execution environment (e.g., Java's JVM or the .NET Framework) will initialize various resources needed to run the program.

- ◆ System Calls and Resource Management

The entry point of a program (usually a function named `main`) is called to begin execution of the code written by the programmer.

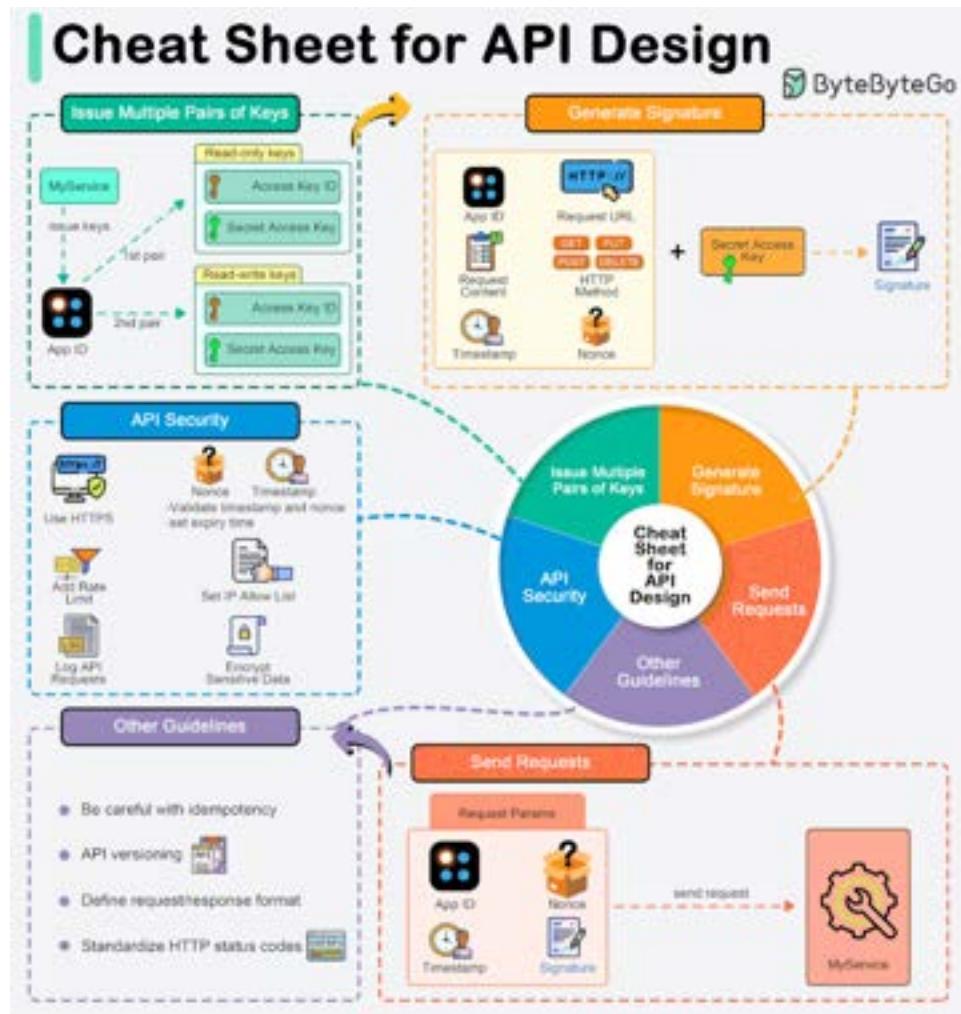
- ◆ Von Neumann Architecture

In the Von Neumann architecture, the CPU executes instructions stored in memory.

- ◆ Program termination

Eventually, when the program has completed its task, or the user actively terminates the application, the program will begin a cleanup phase. This includes closing open file descriptors, freeing up network resources, and returning memory to the system.

A cheat sheet for API designs.



APIs expose business logic and data to external systems, so designing them securely and efficiently is important.

◆ API key generation

We normally generate one unique app ID for each client and generate different pairs of public key (access key) and private key (secret key) to cater to different authorizations. For example, we can generate one pair of keys for read-only access and another pair for read-write access.

◆ Signature generation

Signatures are used to verify the authenticity and integrity of API requests. They are generated using the secret key and typically involve the following steps:

- Collect parameters
- Create a string to sign

- Hash the string: Use a cryptographic hash function, like HMAC (Hash-based Message Authentication Code) in combination with SHA-256, to hash the string using the secret key.

- ◆ Send the requests

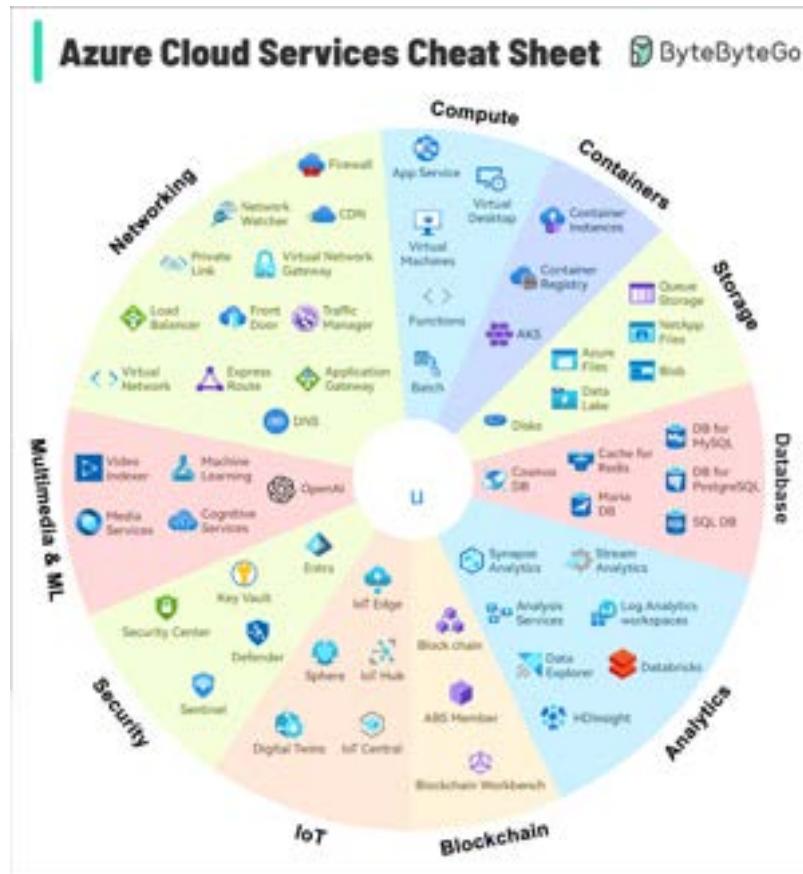
When designing an API, deciding what should be included in HTTP request parameters is crucial. Include the following in the request parameters:

- Authentication Credentials
- Timestamp: To prevent replay attacks.
- Request-specific Data: Necessary to process the request, such as user IDs, transaction details, or search queries.
- Nonces: Randomly generated strings included in each request to ensure that each request is unique and to prevent replay attacks.

- ◆ Security guidelines

To safeguard APIs against common vulnerabilities and threats, adhere to these security guidelines.

Azure Services Cheat Sheet



Launching in 2010, Microsoft Azure has quickly grown to hold the No. 2 position in market share by evolving from basic offerings to a comprehensive, flexible cloud ecosystem.

Today, Azure not only supports traditional cloud applications but also caters to emerging technologies such as AI, IoT, and blockchain, making it a crucial platform for innovation and development.

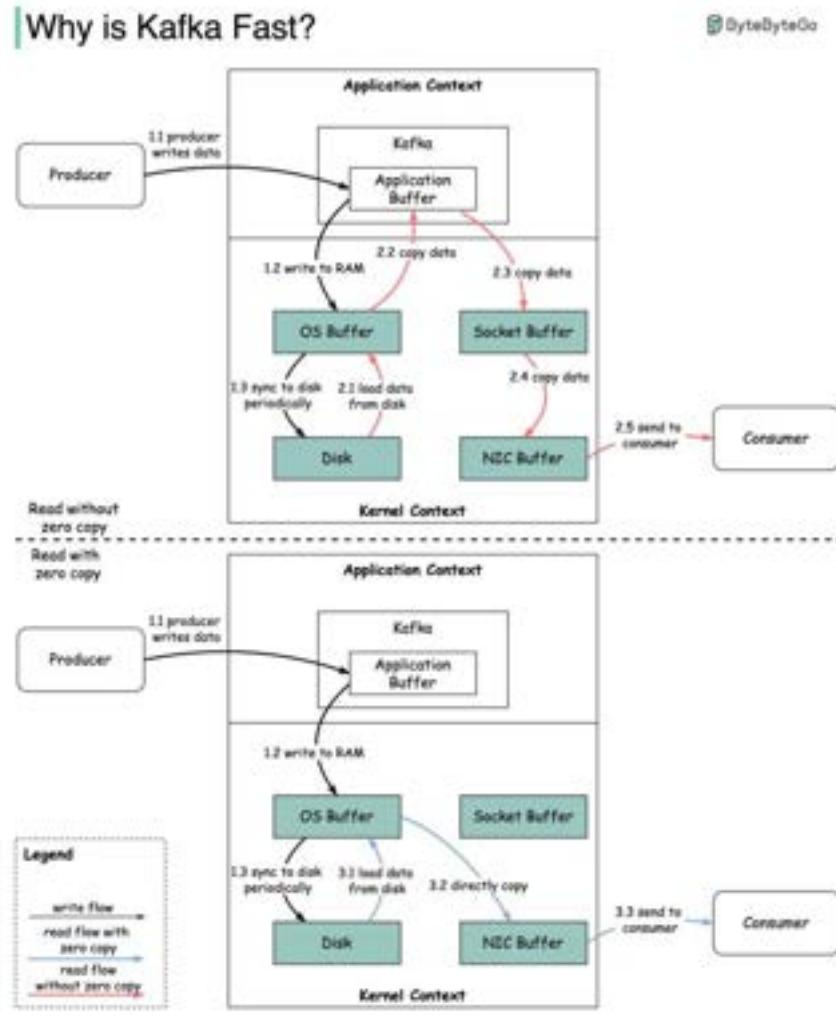
As it evolves, Azure continues to enhance its capabilities to provide advanced solutions for security, scalability, and efficiency, meeting the demands of modern enterprises and startups alike. This expansion allows organizations to adapt and thrive in a rapidly changing digital landscape.

The attached illustration can serve as both an introduction and a quick reference for anyone aiming to understand Azure.

Over to you: How does your experience with Azure compare to that with AWS?

Over to you: Does the card network charge the same interchange fee for big merchants as for small merchants?

Why is Kafka fast?



There are many design decisions that contributed to Kafka's performance. In this post, we'll focus on two. We think these two carried the most weight.

1. The first one is Kafka's reliance on Sequential I/O.
2. The second design choice that gives Kafka its performance advantage is its focus on efficiency: zero copy principle.

The diagram below illustrates how the data is transmitted between producer and consumer, and what zero-copy means.

- ◆ Step 1.1 - 1.3: Producer writes data to the disk
- ◆ Step 2: Consumer reads data without zero-copy

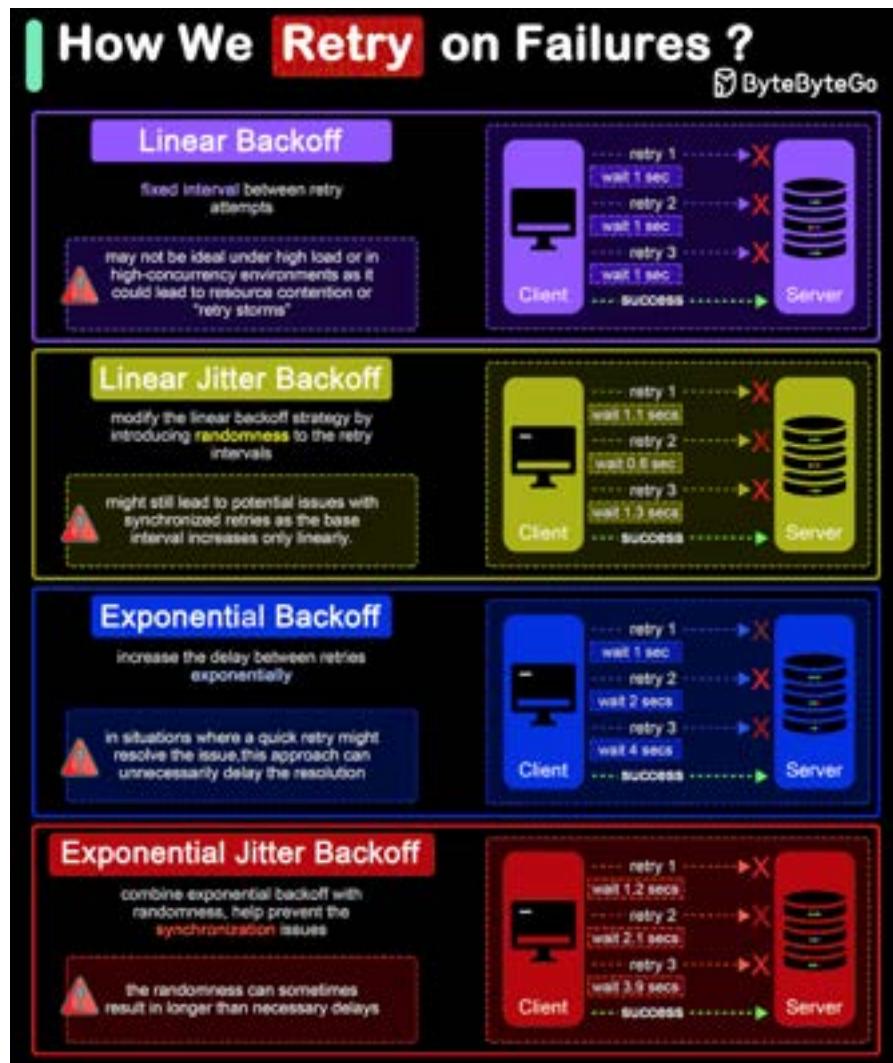
- 2.1: The data is loaded from disk to OS cache
- 2.2 The data is copied from OS cache to Kafka application
- 2.3 Kafka application copies the data into the socket buffer
- 2.4 The data is copied from socket buffer to network card
- 2.5 The network card sends data out to the consumer

◆ Step 3: Consumer reads data with zero-copy

- 3.1: The data is loaded from disk to OS cache
- 3.2 OS cache directly copies the data to the network card via sendfile() command
- 3.3 The network card sends data out to the consumer

Zero copy is a shortcut to save multiple data copies between the application context and kernel context.

How do we retry on failures?



In distributed systems and networked applications, retry strategies are crucial for handling transient errors and network instability effectively. The diagram shows 4 common retry strategies.

◆ Linear Backoff

Linear backoff involves waiting for a progressively increasing fixed interval between retry attempts.

Advantages: Simple to implement and understand.

Disadvantages: May not be ideal under high load or in high-concurrency environments as it could lead to resource contention or "retry storms".

◆ Linear Jitter Backoff

Linear jitter backoff modifies the linear backoff strategy by introducing randomness to the retry intervals. This strategy still increases the delay linearly but adds a random "jitter" to each interval.

Advantages: The randomness helps spread out the retry attempts over time, reducing the chance of synchronized retries across instances.

Disadvantages: Although better than simple linear backoff, this strategy might still lead to potential issues with synchronized retries as the base interval increases only linearly.

◆ Exponential Backoff

Exponential backoff involves increasing the delay between retries exponentially. The interval might start at 1 second, then increase to 2 seconds, 4 seconds, 8 seconds, and so on, typically up to a maximum delay. This approach is more aggressive in spacing out retries than linear backoff.

Advantages: Significantly reduces the load on the system and the likelihood of collision or overlap in retry attempts, making it suitable for high-load environments.

Disadvantages: In situations where a quick retry might resolve the issue, this approach can unnecessarily delay the resolution.

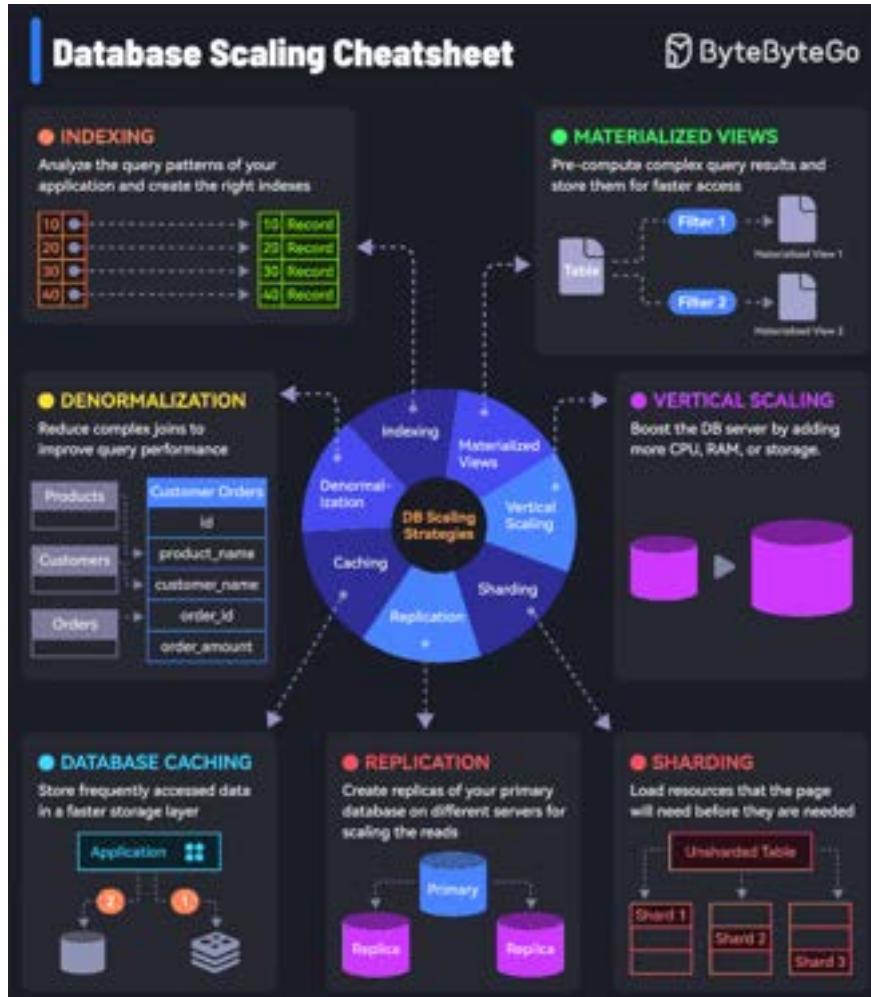
◆ Exponential Jitter Backoff

Exponential jitter backoff combines exponential backoff with randomness. After each retry, the backoff interval is exponentially increased, and then a random jitter is applied. The jitter can be either additive (adding a random amount to the exponential delay) or multiplicative (multiplying the exponential delay by a random factor).

Advantages: Offers all the benefits of exponential backoff, with the added advantage of reducing retry collisions even further due to the introduction of jitter.

Disadvantages: The randomness can sometimes result in longer than necessary delays, especially if the jitter is significant.

7 must-know strategies to scale your database.



1 - Indexing:

Check the query patterns of your application and create the right indexes.

2 - Materialized Views:

Pre-compute complex query results and store them for faster access.

3 - Denormalization:

Reduce complex joins to improve query performance.

4 - Vertical Scaling

Boost your database server by adding more CPU, RAM, or storage.

5 - Caching

Store frequently accessed data in a faster storage layer to reduce database load.

6 - Replication

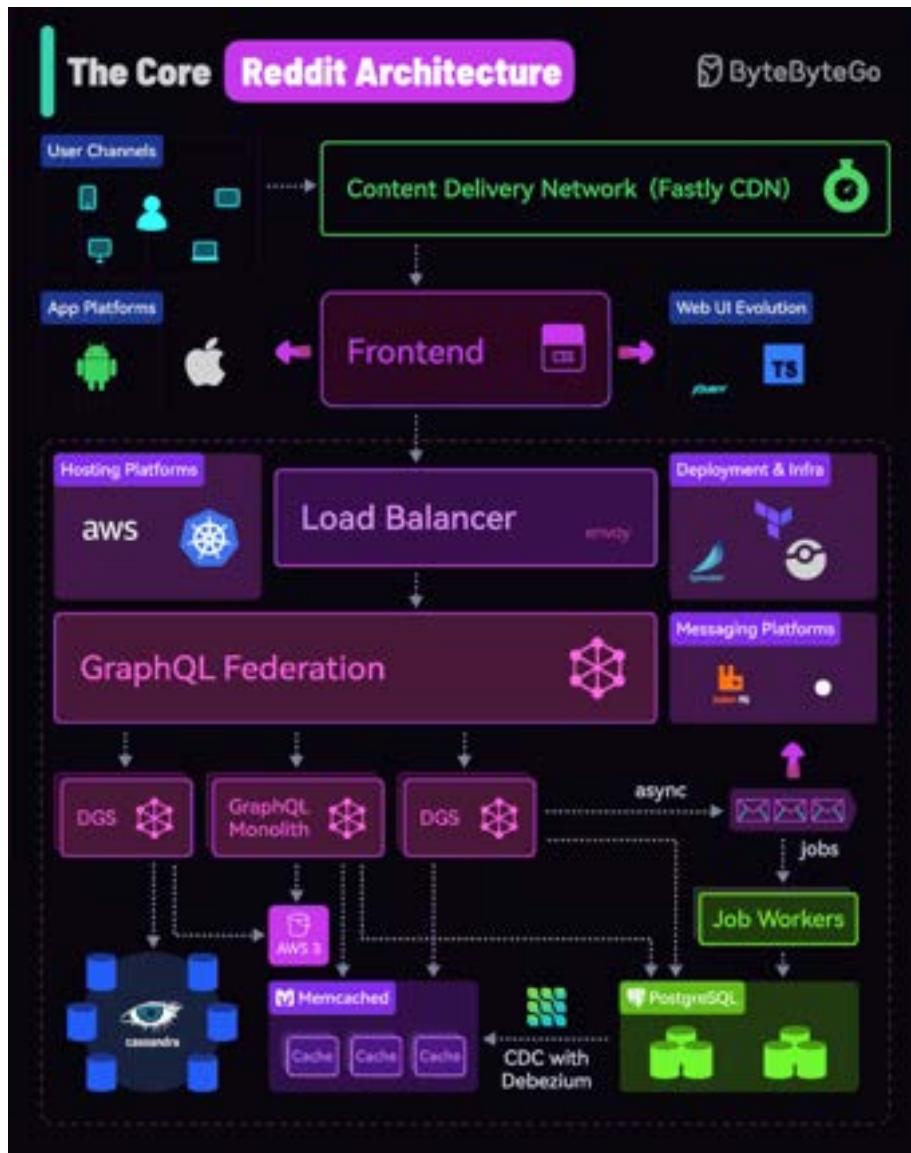
Create replicas of your primary database on different servers for scaling the reads.

7 - Sharding

Split your database tables into smaller pieces and spread them across servers. Used for scaling the writes as well as the reads.

Over to you: What other strategies do you use for scaling your databases?

Reddit's Core Architecture that helps it serve over 1 billion users every month.



This information is based on research from many Reddit engineering blogs. But since architecture is ever-evolving, things might have changed in some aspects.

The main points of Reddit's architecture are as follows:

- 1 - Reddit uses a Content Delivery Network (CDN) from Fastly as a front for the application

2 - Reddit started using jQuery in early 2009. Later on, they started using Typescript and have now moved to modern Node.js frameworks. Over the years, Reddit has also built mobile apps for Android and iOS.

3 - Within the application stack, the load balancer sits in front and routes incoming requests to the appropriate services.

4 - Reddit started as a Python-based monolithic application but has since started moving to microservices built using Go.

5 - Reddit heavily uses GraphQL for its API layer. In early 2021, they started moving to GraphQL Federation, which is a way to combine multiple smaller GraphQL APIs known as Domain Graph Services (DGS). In 2022, the GraphQL team at Reddit added several new Go subgraphs for core Reddit entities thereby splitting the GraphQL monolith.

6 - From a data storage point of view, Reddit relies on Postgres for its core data model. To reduce the load on the database, they use memcached in front of Postgres. Also, they use Cassandra quite heavily for new features mainly because of its resiliency and availability properties.

7 - To support data replication and maintain cache consistency, Reddit uses Debezium to run a Change Data Capture process.

8 - Expensive operations such as a user voting or submitting a link are deferred to an async job queue via RabbitMQ and processed by job workers. For content safety checks and moderation, they use Kafka to transfer data in real-time to run rules over them.

9 - Reddit uses AWS and Kubernetes as the hosting platform for its various apps and internal services.

10 - For deployment and infrastructure, they use Spinnaker, Drone CI, and Terraform.

Over to you: what other aspects do you know about Reddit's architecture?

Everything You Need to Know About Cross-Site Scripting (XSS).



XSS, a prevalent vulnerability, occurs when malicious scripts are injected into web pages, often through input fields. Check out the diagram below for a deeper dive into how this vulnerability emerges when user input is improperly handled and subsequently returned to the client, leaving systems vulnerable to exploitation.

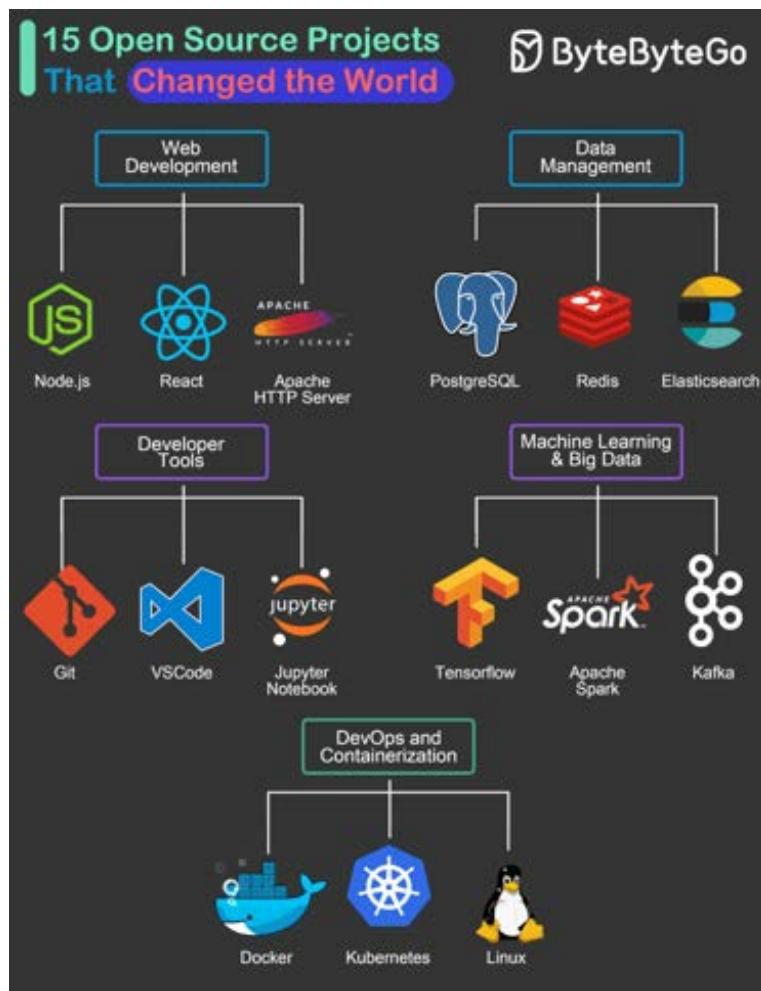
Understanding the distinction between Reflective and Stored XSS is crucial. Reflective XSS involves immediate execution of the injected script, while Stored XSS persists over time, posing long-term threats. Dive into the diagrams for a comprehensive comparison of these attack vectors.

Imagine this scenario: A cunning hacker exploits XSS to clandestinely harvest user credentials, such as cookies, from their browser, potentially leading to unauthorized access and data breaches. It's a chilling reality.

But fret not! Our flyer also delves into effective mitigation strategies, empowering you to fortify your systems against XSS attacks. From input validation and output encoding to implementing strict Content Security Policies (CSP), we've got you covered.

Over to you: How can we amplify user awareness to proactively prevent falling victim to XSS attacks? Share your insights and strategies below! Let's collaboratively bolster our web defenses and foster a safer digital environment.

15 Open-Source Projects That Changed the World



To come up with the list, we tried to look at the overall impact these projects have created on the industry and related technologies. Also, we've focused on projects that have led to a big change in the day-to-day lives of many software developers across the world.

Web Development

- Node.js: The cross-platform server-side Javascript runtime that brought JS to server-side development
- React: The library that became the foundation of many web development frameworks.
- Apache HTTP Server: The highly versatile web server loved by enterprises and startups alike. Served as inspiration for many other web servers over the years.

Data Management

- PostgreSQL: An open-source relational database management system that provided a high-quality alternative to costly systems

- Redis: The super versatile data store that can be used a cache, message broker and even general-purpose storage
- Elasticsearch: A scale solution to search, analyze and visualize large volumes of data

Developer Tools

- Git: Free and open-source version control tool that allows developer collaboration across the globe.
- VSCode: One of the most popular source code editors in the world
- Jupyter Notebook: The web application that lets developers share live code, equations, visualizations and narrative text.

Machine Learning & Big Data

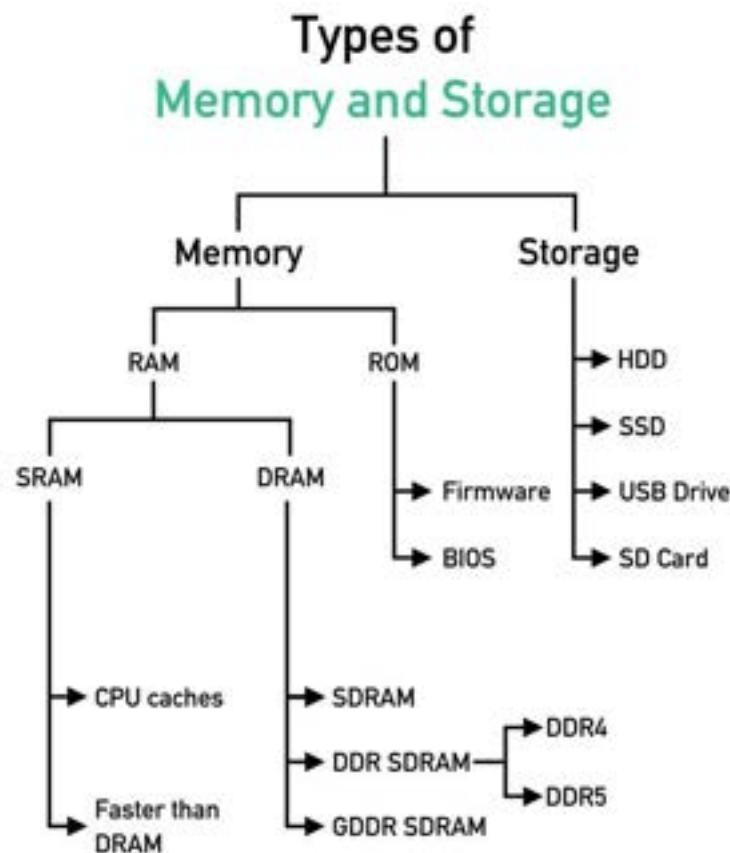
- Tensorflow: The leading choice to leverage machine learning techniques
- Apache Spark: Standard tool for big data processing and analytics platforms
- Kafka: Standard platform for building real-time data pipelines and applications.

DevOps & Containerization

- Docker: The open source solution that allows developers to package and deploy applications in a consistent and portable way.
- Kubernetes: The heart of Cloud-Native architecture and a platform to manage multiple containers
- Linux: The OS that democratized the world of software development.

Over to you: Do you agree with the list? What did we miss?

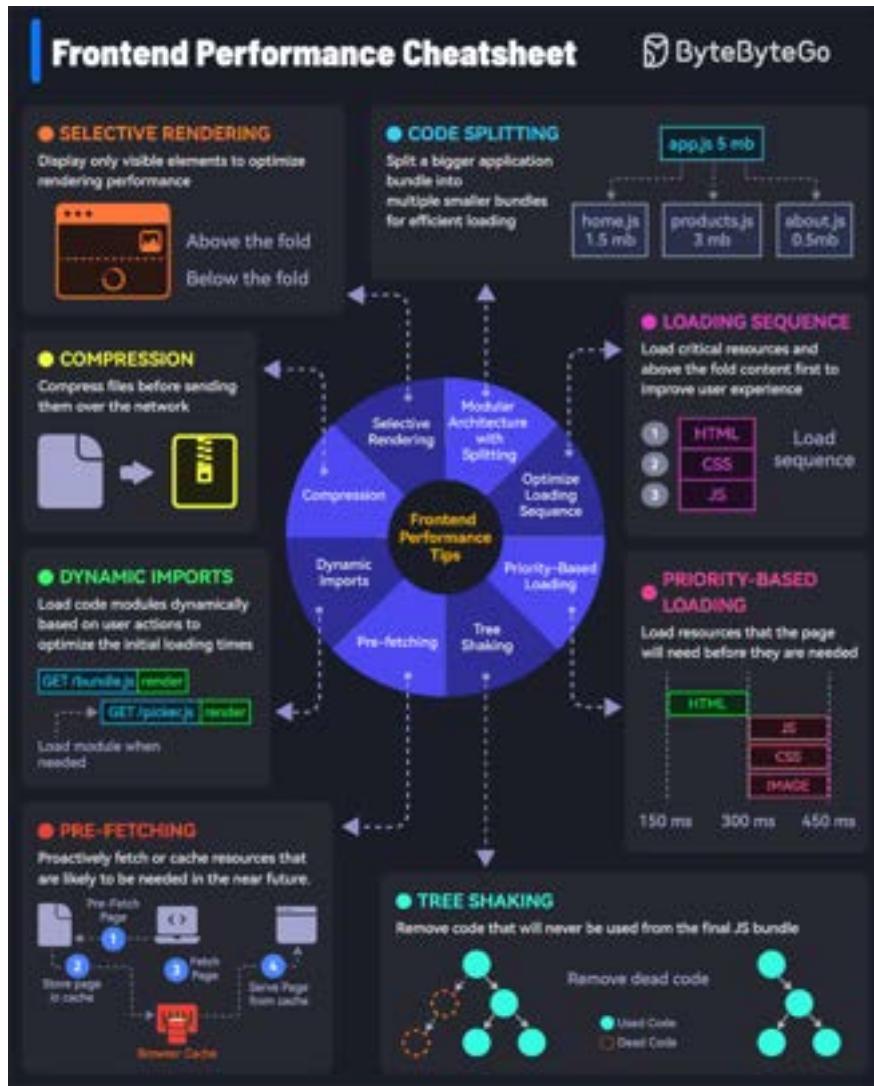
Types of Memory and Storage



 ByteByteGo

- The fundamental duo: RAM and ROM
- DDR4 and DDR5
- Firmware and BIOS
- SRAM and DRAM
- HDD, SSD, USB Drive, SD Card

How to load your websites at lightning speed?



Check out these 8 tips to boost frontend performance:

1 - Compression

Compress files and minimize data size before transmission to reduce network load.

2 - Selective Rendering/Windowing

Display only visible elements to optimize rendering performance. For example, in a dynamic list, only render visible items.

3 - Modular Architecture with Code Splitting

Split a bigger application bundle into multiple smaller bundles for efficient loading.

4 - Priority-Based Loading

Prioritize essential resources and visible (or above-the-fold) content for a better user experience.

5 - Pre-loading

Fetch resources in advance before they are requested to improve loading speed.

6 - Tree Shaking or Dead Code Removal

Optimize the final JS bundle by removing dead code that will never be used.

7 - Pre-fetching

Proactively fetch or cache resources that are likely to be needed soon.

8 - Dynamic Imports

Load code modules dynamically based on user actions to optimize the initial loading times.

Over to you: What other frontend performance tips would you add to this cheat sheet?

25 Papers That Completely Transformed the Computer World.

25 Papers That Completely Transformed the Computer World

ByteByteGo

- [Dynamo](#): Amazon's Highly Available Key Value Store
- [Google File System](#): Insights into a highly scalable file system
- [Scaling Memcached at Facebook](#): A look at the complexities of Caching at a cluster
- [BigTable](#): The design principles behind a distributed storage system for structured data
- [Borg](#): Large Scale Cluster Management at Google
- [Cassandra](#): A look at the design and architecture of a distributed NoSQL database
- [Attention Is All You Need](#): Into a new deep learning architecture known as the transformer
- [Kafka](#): Internals of the distributed messaging platform
- [FoundationDB](#): A look at how a distributed database
- [Amazon Aurora](#): To learn how Amazon provides high-availability and performance
- [Spanner](#): Design and architecture of Google's globally distributed database
- [MapReduce](#): A detailed look at how MapReduce enables parallel processing
- [Shard Manager](#): Understanding the generic shard management framework
- [Dapper](#): Insights into Google's distributed systems tracing infrastructure
- [Flink](#): A detailed look at the unified architecture of stream and batch data processing
- [A Comprehensive Survey on Vector Databases](#): Algorithms powering the vector database
- [Zanzibar](#): A global system for managing access control lists at Google
- [Monarch](#): Architecture of Google's globally distributed in-memory time series database
- [Thrift](#): Explore the design choices behind Facebook's code-generation tool
- [Bitcoin](#): The ground-breaking introduction to the peer-to-peer electronic cash system
- [WTF](#): Who to Follow Service at Twitter
- [MyRocks](#): LSM-Tree Database Storage Engine
- [GoTo Considered Harmful](#): Understand why GoTo statements are problematic
- [Raft Consensus Algorithm](#): To learn about the more understandable consensus algorithm
- [Time Clocks and Ordering of Events](#): Explains the concept of time and event ordering

1. Dynamo - Amazon's Highly Available Key Value Store
2. Google File System: Insights into a highly scalable file system
3. Scaling Memcached at Facebook: A look at the complexities of Caching
4. BigTable: The design principles behind a distributed storage system
5. Borg - Large Scale Cluster Management at Google
6. Cassandra: A look at the design and architecture of a distributed NoSQL database
7. Attention Is All You Need: Into a new deep learning architecture known as the transformer
8. Kafka: Internals of the distributed messaging platform
9. FoundationDB: A look at how a distributed database
10. Amazon Aurora: To learn how Amazon provides high-availability and performance
11. Spanner: Design and architecture of Google's globally distributed databases

12. MapReduce: A detailed look at how MapReduce enables parallel processing of massive volumes of data
13. Shard Manager: Understanding the generic shard management framework
14. Dapper: Insights into Google's distributed systems tracing infrastructure
15. Flink: A detailed look at the unified architecture of stream and batch processing
16. A Comprehensive Survey on Vector Databases
17. Zanzibar: A look at the design, implementation and deployment of a global system for managing access control lists at Google
18. Monarch: Architecture of Google's in-memory time series database
19. Thrift: Explore the design choices behind Facebook's code-generation tool
20. Bitcoin: The ground-breaking introduction to the peer-to-peer electronic cash system
21. WTF - Who to Follow Service at Twitter: Twitter's (now X) user recommendation system
22. MyRocks: LSM-Tree Database Storage Engine
23. GoTo Considered Harmful
24. Raft Consensus Algorithm: To learn about the more understandable consensus algorithm
25. Time Clocks and Ordering of Events: The extremely important paper that explains the concept of time and event ordering in a distributed system

Over to you: I'm sure we missed many important papers. Which ones do you think should be included?

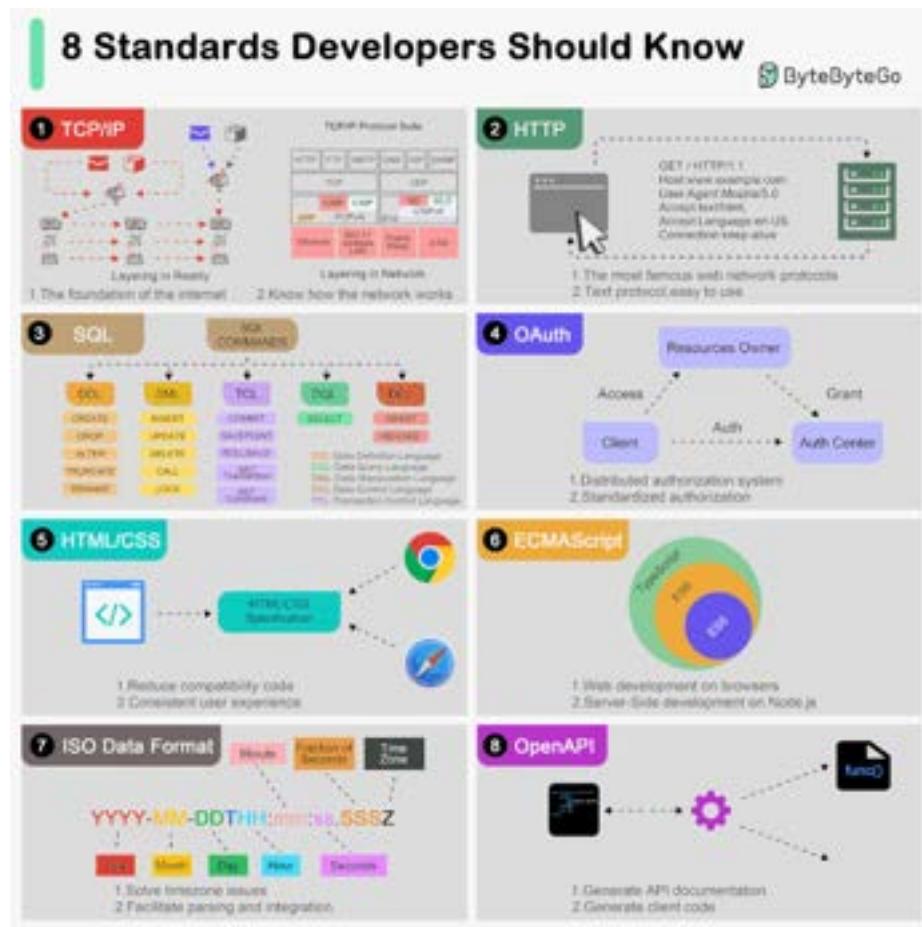
10 Essential Components of a Production Web Application.



- 1 - It all starts with CI/CD pipelines that deploy code to the server instances. Tools like Jenkins and GitHub help over here.
- 2 - The user requests originate from the web browser. After DNS resolution, the requests reach the app servers.
- 3 - Load balancers and reverse proxies (such as Nginx & HAProxy) distribute user requests evenly across the web application servers.
- 4 - The requests can also be served by a Content Delivery Network (CDN).
- 5 - The web app communicates with backend services via APIs.
- 6 - The backend services interact with database servers or distributed caches to provide the data.
- 7 - Resource-intensive and long-running tasks are sent to job workers using a job queue.
- 8 - The full-text search service supports the search functionality. Tools like Elasticsearch and Apache Solr can help here.
- 9 - Monitoring tools (such as Sentry, Grafana, and Prometheus) store logs and help analyze data to ensure everything works fine.
- 10 - In case of issues, alerting services notify developers through platforms like Slack for quick resolution.

Over to you: What other components would you add to the architecture of a production web app?

Top 8 Standards Every Developer Should Know.



◆ TCP/IP

Developed by the IETF organization, the TCP/IP protocol is the foundation of the Internet and one of the best-known networking standards.

◆ HTTP

The IETF has also developed the HTTP protocol, which is essential for all web developers.

◆ SQL

Structured Query Language (SQL) is a domain-specific language used to manage data.

◆ OAuth

OAuth (Open Authorization) is an open standard for access delegation commonly used to grant websites or applications limited access to user information without exposing their passwords.

◆ HTML/CSS

With HTML, web pages are rendered uniformly across browsers, which reduces development effort spent on compatibility issues.HTML tags.

CSS standards are often used in conjunction with HTML.

- ◆ ECMAScript

ECMAScript is a standardized scripting language specification that serves as the foundation for several programming languages, the most well-known being JavaScript.

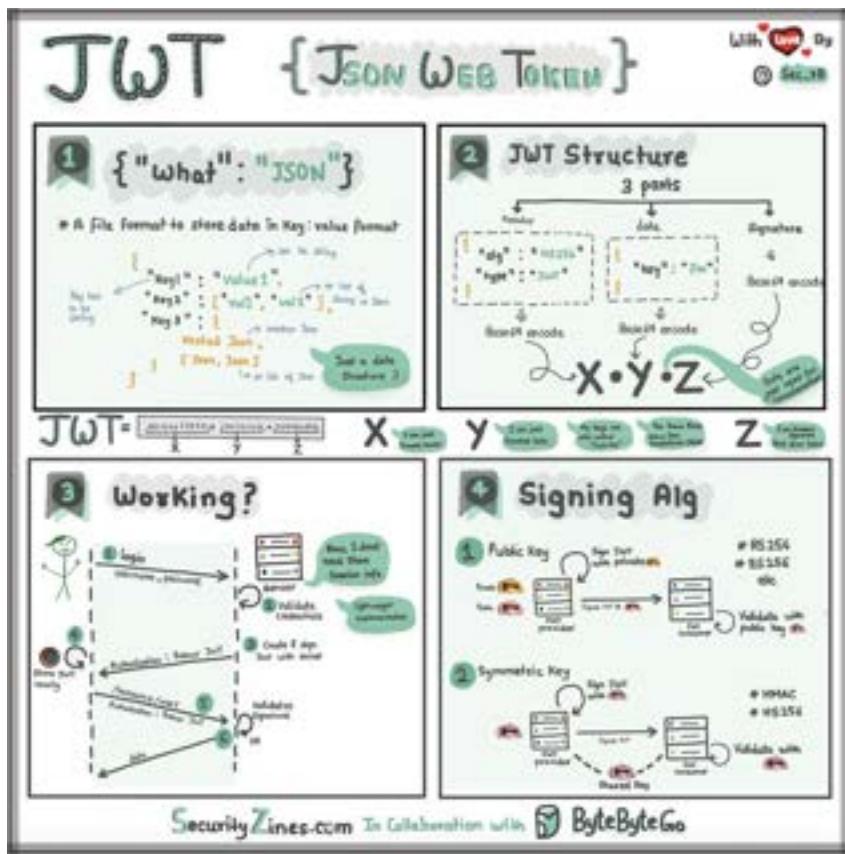
- ◆ ISO Date

It is common for developers to have problems with inconsistent time formats on a daily basis. ISO 8601 is a date and time format standard developed by the ISO (International Organization for Standardization) to provide a common format for exchanging date and time data across borders, cultures, and industries.

- ◆ OpenAPI

OpenAPI, also known as the OpenAPI Specification (OAS), is a standardized format for describing and documenting RESTful APIs.

Explaining JSON Web Token (JWT) with simple terms.



Imagine you have a special box called a JWT. Inside this box, there are three parts: a header, a payload, and a signature.

The header is like the label on the outside of the box. It tells us what type of box it is and how it's secured. It's usually written in a format called JSON, which is just a way to organize information using curly braces {} and colons : .

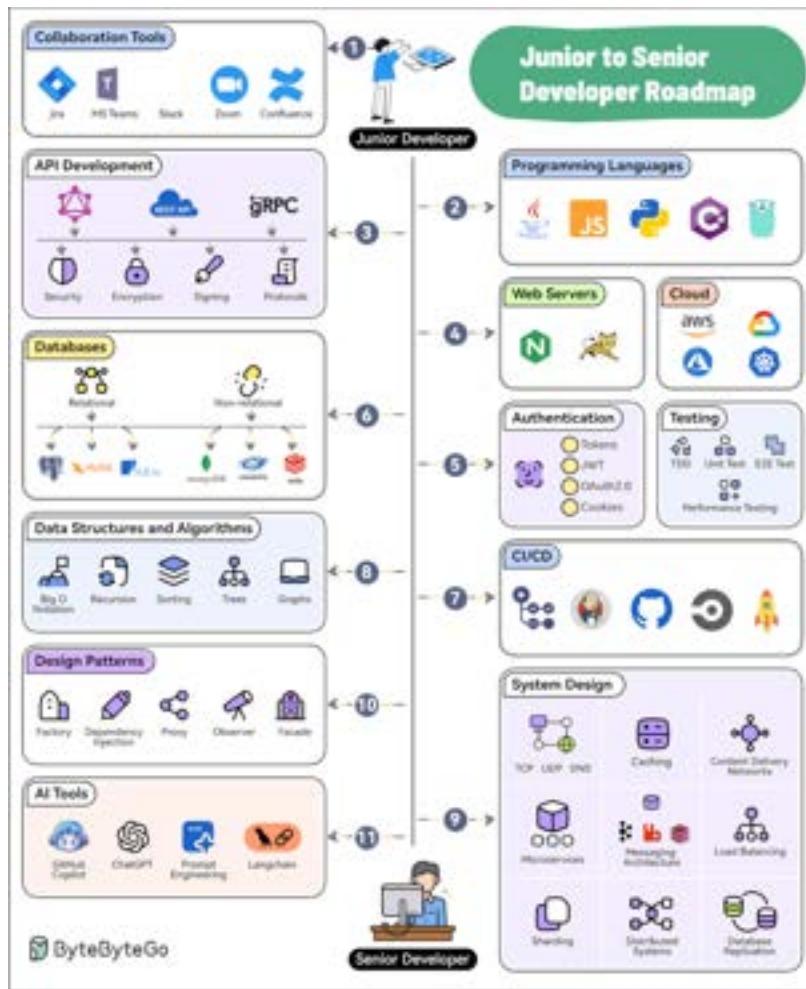
The payload is like the actual message or information you want to send. It could be your name, age, or any other data you want to share. It's also written in JSON format, so it's easy to understand and work with.

Now, the signature is what makes the JWT secure. It's like a special seal that only the sender knows how to create. The signature is created using a secret code, kind of like a password. This signature ensures that nobody can tamper with the contents of the JWT without the sender knowing about it.

When you want to send the JWT to a server, you put the header, payload, and signature inside the box. Then you send it over to the server. The server can easily read the header and payload to understand who you are and what you want to do.

Over to you: When should we use JWT for authentication? What are some other authentication methods?

11 steps to go from Junior to Senior Developer.



1 - Collaboration Tools

Software development is a social activity. Learn to use collaboration tools like Jira, Confluence, Slack, MS Teams, Zoom, etc.

2 - Programming Languages

Pick and master one or two programming languages. Choose from options like Java, Python, JavaScript, C#, Go, etc.

3 - API Development

Learn the ins and outs of API Development approaches such as REST, GraphQL, and gRPC.

4 - Web Servers and Hosting

Know about web servers as well as cloud platforms like AWS, Azure, GCP, and Kubernetes

5 - Authentication and Testing

Learn how to secure your applications with authentication techniques such as JWTs, OAuth2, etc. Also, master testing techniques like TDD, E2E Testing, and Performance Testing

6 - Databases

Learn to work with relational (Postgres, MySQL, and SQLite) and non-relational databases (MongoDB, Cassandra, and Redis).

7 - CI/CD

Pick tools like GitHub Actions, Jenkins, or CircleCI to learn about continuous integration and continuous delivery.

8 - Data Structures and Algorithms

Master the basics of DSA with topics like Big O Notation, Sorting, Trees, and Graphs.

9 - System Design

Learn System Design concepts such as Networking, Caching, CDNs, Microservices, Messaging, Load Balancing, Replication, Distributed Systems, etc.

10 - Design patterns

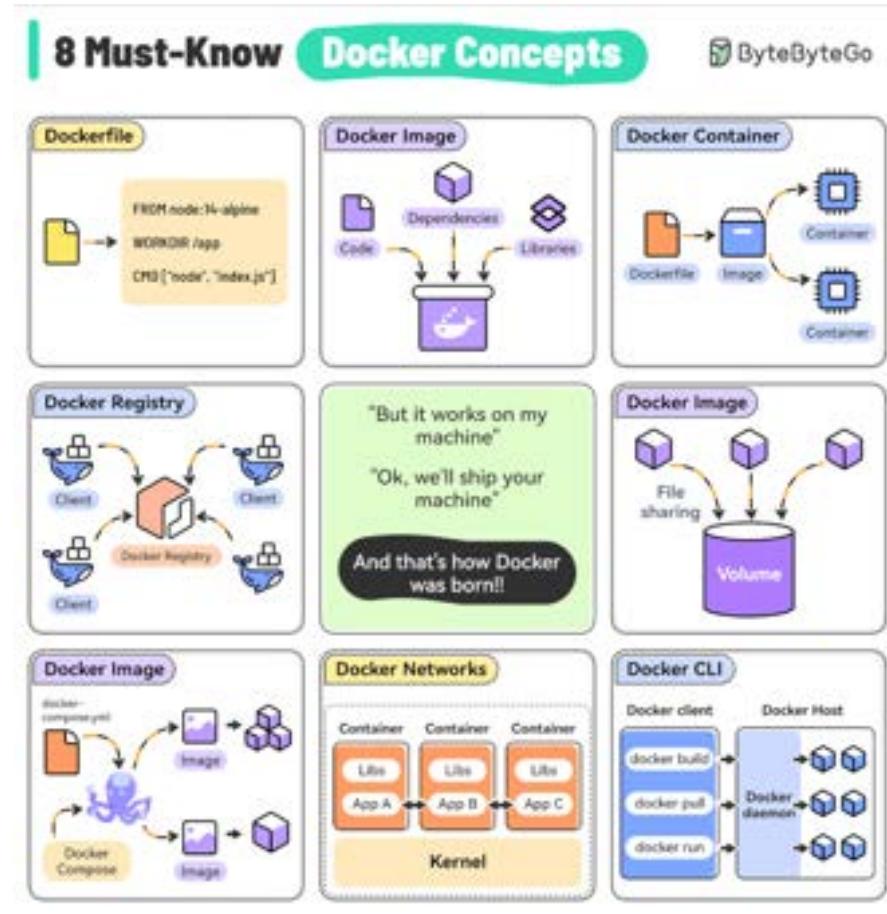
Master the application of design patterns such as dependency injection, factory, proxy, observers, and facade.

11 - AI Tools

To future-proof your career, learn to leverage AI tools like GitHub Copilot, ChatGPT, Langchain, and Prompt Engineering.

Over to you: What else would you add to the roadmap?

Top 8 must-know Docker concepts



1 - Dockerfile: It contains the instructions to build a Docker image by specifying the base image, dependencies, and run command.

2 - Docker Image: A lightweight, standalone package that includes everything (code, libraries, and dependencies) needed to run your application. Images are built from a Dockerfile and can be versioned.

3 - Docker Container: A running instance of a Docker image. Containers are isolated from each other and the host system, providing a secure and reproducible environment for running your apps.

4 - Docker Registry: A centralized repository for storing and distributing Docker images. For example, Docker Hub is the default public registry but you can also set up private registries.

5 - Docker Volumes: A way to persist data generated by containers. Volumes are outside the container's file system and can be shared between multiple containers.

6 - Docker Compose: A tool for defining and running multi-container Docker applications, making it easy to manage the entire stack.

7 - Docker Networks: Used to enable communication between containers and the host system. Custom networks can isolate containers or enable selective communication.

8 - Docker CLI: The primary way to interact with Docker, providing commands for building images, running containers, managing volumes, and performing other operations.

Over to you: What other concept should one know about Docker?

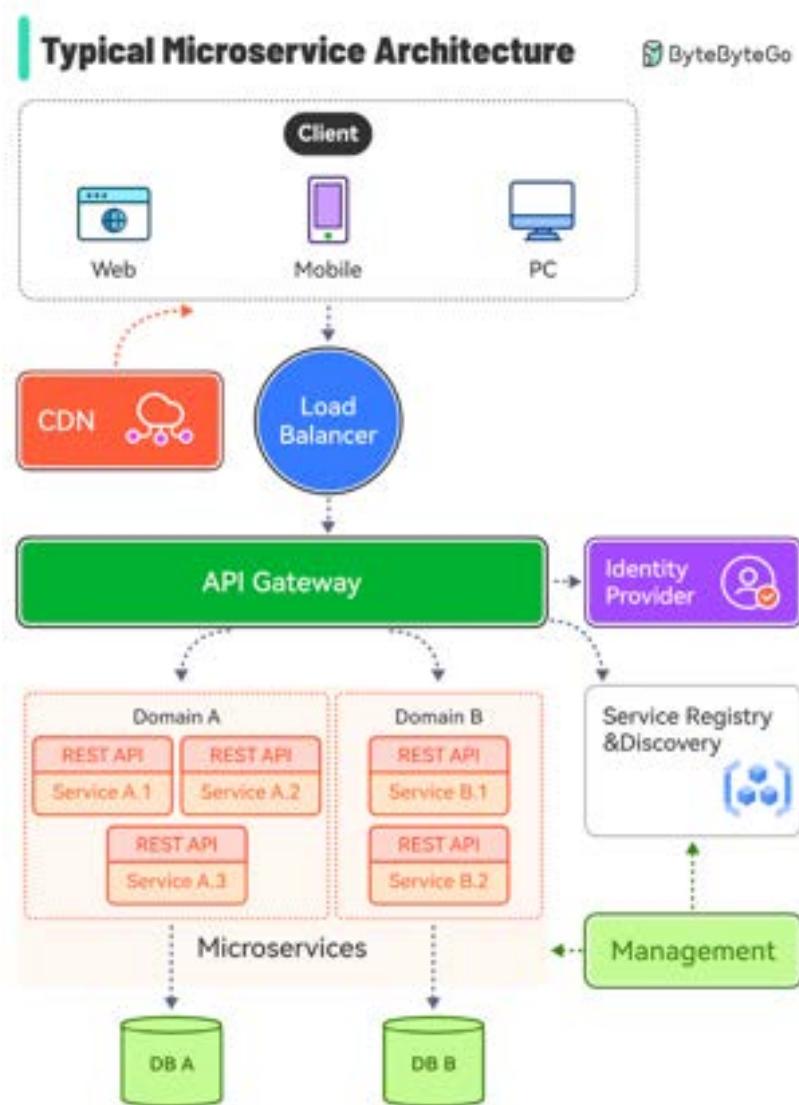
Top 10 Most Popular Open-Source Databases



This list is based on factors like adoption, industry impact, and the general awareness of the database among the developer community.

- 1 - MySQL
- 2 - PostgreSQL
- 3 - MariaDB
- 4 - Apache Cassandra
- 5 - Neo4j
- 6 - SQLite
- 7 - CockroachDB
- 8 - Redis
- 9 - MongoDB
- 10 - Couchbase

What does a typical microservice architecture look like?



The diagram below shows a typical microservice architecture.

- Load Balancer: This distributes incoming traffic across multiple backend services.
- CDN (Content Delivery Network): CDN is a group of geographically distributed servers that hold static content for faster delivery. The clients look for content in CDN first, then progress to backend services.

- ◆ API Gateway: This handles incoming requests and routes them to the relevant services. It talks to the identity provider and service discovery.
- ◆ Identity Provider: This handles authentication and authorization for users.
- ◆ Service Registry & Discovery: Microservice registration and discovery happen in this component, and the API gateway looks for relevant services in this component to talk to.
- ◆ Management: This component is responsible for monitoring the services.
- ◆ Microservices: Microservices are designed and deployed in different domains. Each domain has its database.

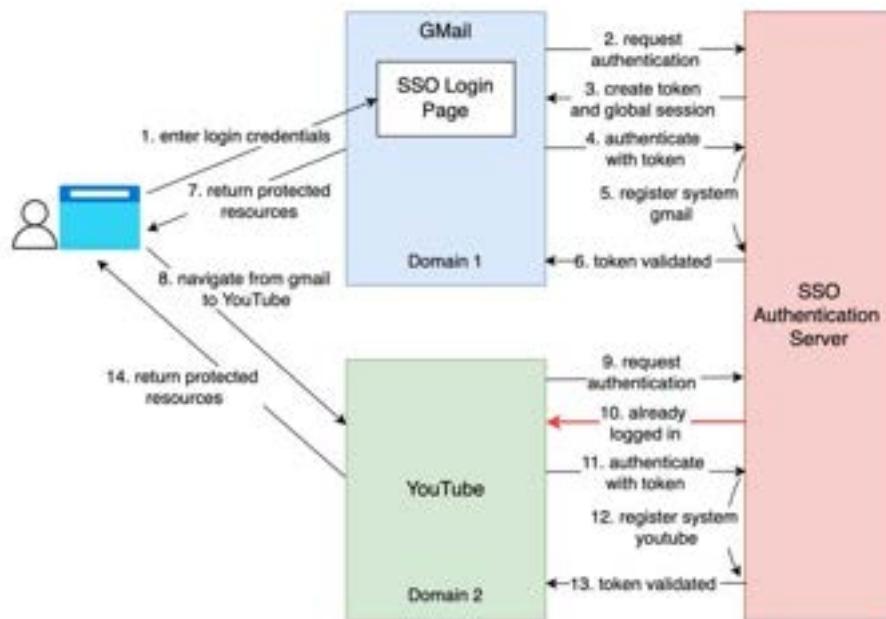
Over to you:

- 1). What are the drawbacks of the microservice architecture?
- 2). Have you seen a monolithic system be transformed into microservice architecture? How long does it take?

What is SSO (Single Sign-On)?

How does SSO Work?

blog.bytebytego.com



Basically, Single Sign-On (SSO) is an authentication scheme. It allows a user to log in to different systems using a single ID.

The diagram below illustrates how SSO works.

Step 1: A user visits Gmail, or any email service. Gmail finds the user is not logged in and so redirects them to the SSO authentication server, which also finds the user is not logged in. As a result, the user is redirected to the SSO login page, where they enter their login credentials.

Steps 2-3: The SSO authentication server validates the credentials, creates the global session for the user, and creates a token.

Steps 4-7: Gmail validates the token in the SSO authentication server. The authentication server registers the Gmail system, and returns "valid." Gmail returns the protected resource to the user.

Step 8: From Gmail, the user navigates to another Google-owned website, for example, YouTube.

Steps 9-10: YouTube finds the user is not logged in, and then requests authentication. The SSO authentication server finds the user is already logged in and returns the token.

Step 11-14: YouTube validates the token in the SSO authentication server. The authentication server registers the YouTube system, and returns “valid.” YouTube returns the protected resource to the user.

The process is complete and the user gets back access to their account.

Over to you:

Question 1: have you implemented SSO in your projects? What is the most difficult part?

Question 2: what's your favorite sign-in method and why?

What makes HTTP2 faster than HTTP1?



The key features of HTTP2 play a big role in this. Let's look at them:

1 - Binary Framing Layer

HTTP2 encodes the messages into binary format.

This allows the messages into smaller units called frames, which are then sent over the TCP connection, resulting in more efficient processing.

2 - Multiplexing

The Binary Framing allows full request and response multiplexing.

Clients and servers can interleave frames during transmissions and reassemble them on the other side.

3 - Stream Prioritization

With stream prioritization, developers can customize the relative weight of requests or streams to make the server send more frames for higher-priority requests.

4 - Server Push

Since HTTP2 allows multiple concurrent responses to a client's request, a server can send additional resources along with the requested page to the client.

5 - HPACK Header Compression

HTTP2 uses a special compression algorithm called HPACK to make the headers smaller for multiple requests, thereby saving bandwidth.

Of course, despite these features, HTTP2 can also be slow depending on the exact technical scenario. Therefore, developers need to test and optimize things to maximize the benefits of HTTP2.

Over to you: Have you used HTTP2 in your application?

Log Parsing Cheat Sheet

Top 6 Log Parsing Commands

ByteByteGo

 GREP	\$grep <pattern> <file.log> GREP searches any given input files, selecting lines that match one or more patterns.	① find file names that match <code>\$grep -l "bytebytego" *log</code> ② case insensitive word match <code>\$grep -wi "bytebytego" test.log</code> ③ show line numbers <code>\$grep -n "bytebytego" test.log</code> ④ invert matches <code>\$grep -v "bytebytego" test.log</code> ⑤ take patterns from a file <code>\$grep -f pattern.txt test.log</code> ⑥ search recursively in a dir <code>\$grep -R "bytebytego" /home</code>
 CUT	\$cut -d"," -f 3 <file.log> CUT cuts out selected portions of each line from each file and writes them to the standard output.	① cut first 3 bytes <code>\$cut -b 1,2,3 file.log</code> ② select 2nd column delimited by a space <code>\$cut -d" " -f 2 test.log</code> ③ specify characters position <code>\$cut -c 1-8 test.log</code>
 SED	\$sed s/<regex>/<replace>/g SED reads the specified files, modifying the input as specified by a list of commands.	① substitute a string <code>\$sed s/bytebytego/g/o/ test.log</code> ② replace the 2nd occurrence <code>\$sed s/bytebytego/g/2 test.log</code> ③ replace string on the 4th line <code>\$sed '4 s/bytebytego/g/' test.log</code> ④ replace string on a range of lines <code>\$sed '2-4 s/bytebytego/g/' test.log</code> ⑤ delete a line <code>\$sed '4d' test.log</code>
 AWK	\$awk '{print \$4}' <file.log> AWK scans each input file for lines that match any of a set of patterns.	① print matched lines <code>\$awk '/bytebytego/ {print}' test.log</code> ② split a line into fields <code>\$awk '{print \$1,\$3}' test.log</code> ③ print lines 2 to 7 <code>\$awk 'NR==2, NR==7 {print NR, \$0}' test.log</code> ④ print lines with more than 10 characters <code>\$awk 'length(\$0)>10' test.log</code> ⑤ find a string in a column <code>\$awk 'if(\$4=="byte" print \$0)' test.log</code>
 SORT	\$sort <file.log> SORT sorts text and binary files by lines.	① output to a file <code>\$sort -o output.txt input.txt</code> ② sort in reverse order <code>\$sort -r test.log</code> ③ sort numerically <code>\$sort -n test.log</code> ④ sort based on the 3rd column <code>\$sort -k 3n test.log</code> ⑤ check if a file is ordered <code>\$sort -c test.log</code> ⑥ sort and remove duplicates <code>\$sort -u test.log</code>
 UNIQ	\$uniq <file.log> UNIQ reads the specified input file comparing adjacent lines, and writes a copy of each unique input line to the output file.	① tell how many times a line is repeated <code>\$uniq -c test.log</code> ② print repeated lines <code>\$uniq -d test.log</code> ③ print unique lines <code>\$uniq -u test.log</code> ④ skip the first two fields <code>\$uniq -f 2 test.log</code> ⑤ compare case-insensitive <code>\$uniq -i test.log</code>

The diagram below lists the top 6 log parsing commands.

1. GREP

GREP searches any given input files, selecting lines that match one or more patterns.

2. CUT

CUT cuts out selected portions of each line from each file and writes them to the standard output.

3. SED

SED reads the specified files, modifying the input as specified by a list of commands.

4. AWK

AWK scans each input file for lines that match any of a set of patterns.

5. SORT

SORT sorts text and binary files by lines.

6. UNIQ

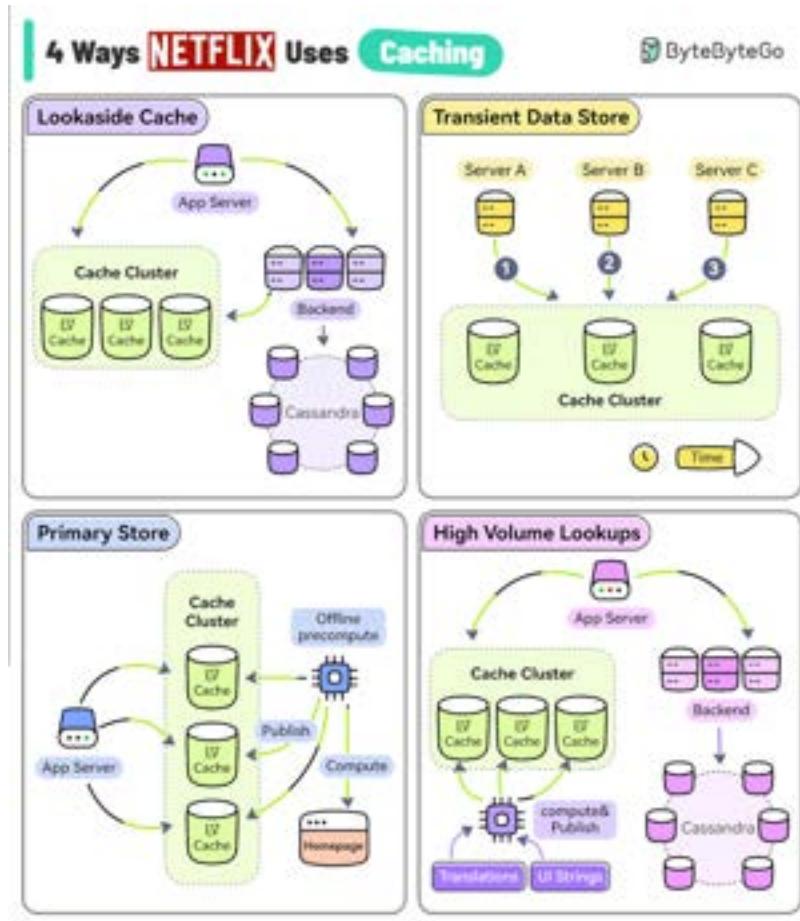
UNIQ reads the specified input file comparing adjacent lines and writes a copy of each unique input line to the output file.

These commands are often used in combination to quickly find useful information from the log files. For example, the below commands list the timestamps (column 2) when there is an exception happening for xxService.

```
grep "xxService" service.log | grep "Exception" | cut -d" " -f 2
```

Over to you: What other commands do you use when you parse logs?

4 Ways Netflix Uses Caching to Hold User Attention



The goal of Netflix is to keep you streaming for as long as possible. But a user's typical attention span is just 90 seconds.

They use EVCache (a distributed key-value store) to reduce latency so that the users don't lose interest.

However, EVCache has multiple use cases at Netflix.

1 - Lookaside Cache

When the application needs some data, it first tries the EVCache client and if the data is not in the cache, it goes to the backend service and the Cassandra database to fetch the data.

The service also keeps the cache updated for future requests.

2 - Transient Data Store

Netflix uses EVCache to keep track of transient data such as playback session information.

One application service might start the session while the other may update the session followed by a session closure at the very end.

3 - Primary Store

Netflix runs large-scale pre-compute systems every night to compute a brand-new home page for every profile of every user based on watch history and recommendations.

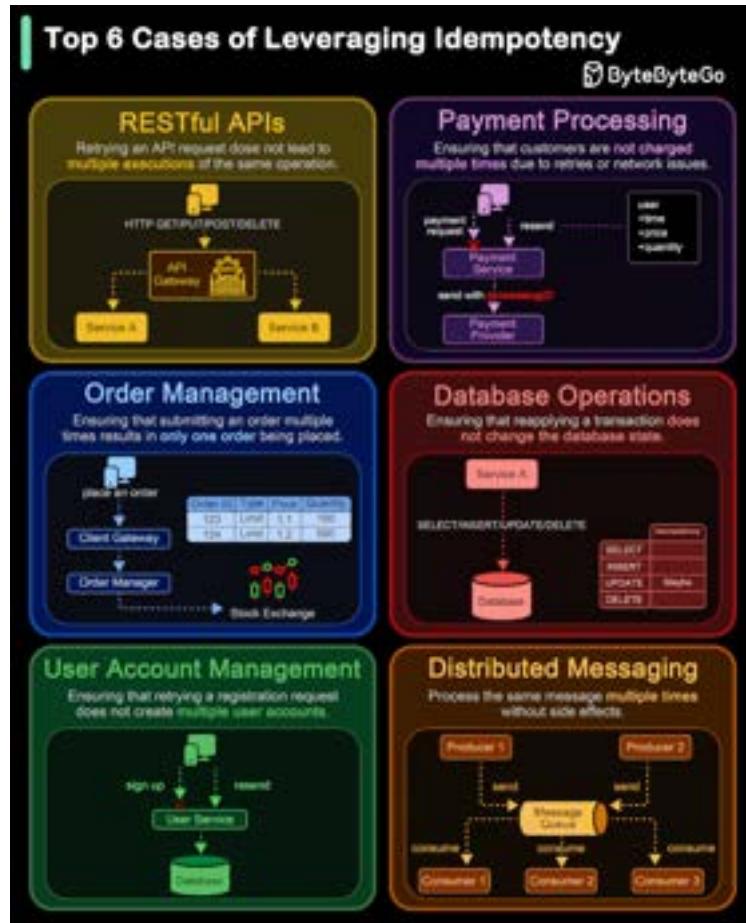
All of that data is written into the EVCache cluster from where the online services read the data and build the homepage.

4 - High Volume Data

Netflix has data that has a high volume of access and also needs to be highly available. For example, UI strings and translations that are shown on the Netflix home page.

A separate process asynchronously computes and publishes the UI string to EVCache from where the application can read it with low latency and high availability.

Top 6 Cases to Apply Idempotency.



Idempotency is essential in various scenarios, particularly where operations might be retried or executed multiple times. Here are the top 6 use cases where idempotency is crucial:

1. RESTful API Requests

We need to ensure that retrying an API request does not lead to multiple executions of the same operation. Implement idempotent methods (like PUT and DELETE) to maintain consistent resource states.

2. Payment Processing

We need to ensure that customers are not charged multiple times due to retries or network issues. Payment gateways often need to retry transactions; idempotency ensures only one charge is made.

3. Order Management Systems

We need to ensure that submitting an order multiple times results in only one order being placed. We design a safe mechanism to prevent duplicate inventory deductions or updates.

4. Database Operations

We need to ensure that reapplying a transaction does not change the database state beyond the initial application.

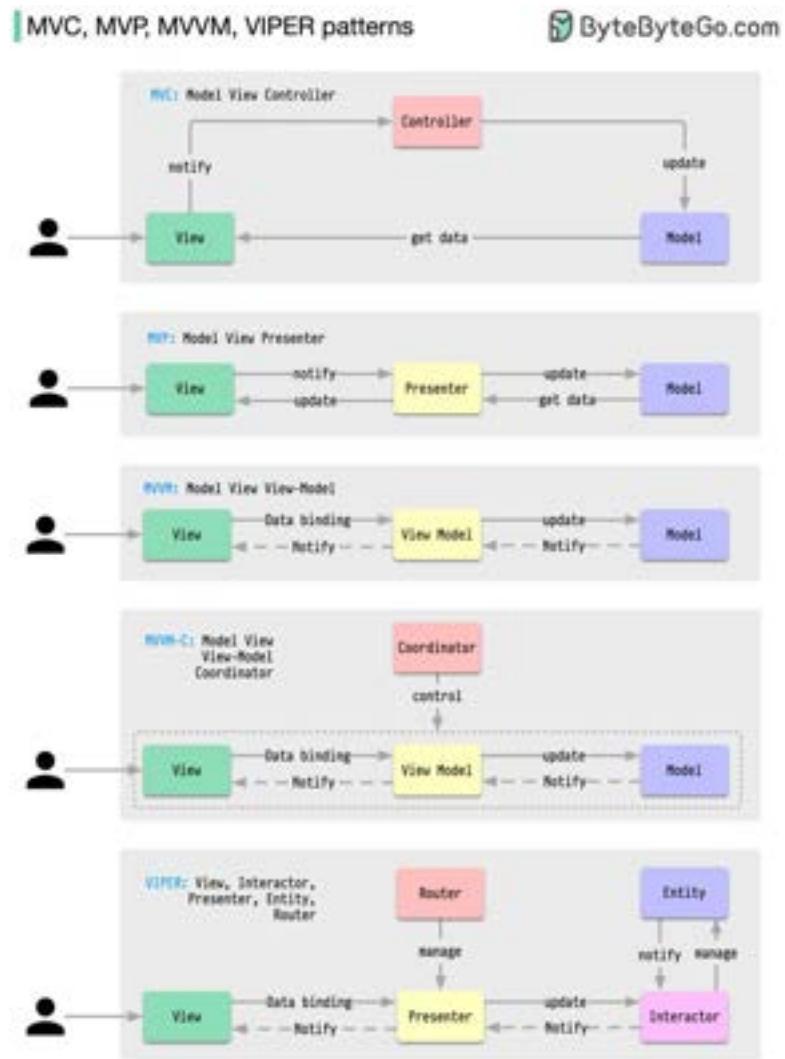
5. User Account Management

We need to ensure that retrying a registration request does not create multiple user accounts. Also, we need to ensure that multiple password reset requests result in a single reset action.

6. Distributed Systems and Messaging

We need to ensure that reprocessing messages from a queue does not result in duplicate processing. We Implement handlers that can process the same message multiple times without side effects.

MVC, MVP, MVVM, MVVM-C, and VIPER architecture patterns



These architecture patterns are among the most commonly used in app development, whether on iOS or Android platforms. Developers have introduced them to overcome the limitations of earlier patterns. So, how do they differ?

- MVC, the oldest pattern, dates back almost 50 years
- Every pattern has a "view" (V) responsible for displaying content and receiving user input
- Most patterns include a "model" (M) to manage business data
- "Controller," "presenter," and "view-model" are translators that mediate between the view and the model ("entity" in the VIPER pattern)
- These translators can be quite complex to write, so various patterns have been proposed to make them more maintainable

What are the differences among database locks?



In database management, locks are mechanisms that prevent concurrent access to data to ensure data integrity and consistency.

Here are the common types of locks used in databases:

1. Shared Lock (S Lock)

It allows multiple transactions to read a resource simultaneously but not modify it. Other transactions can also acquire a shared lock on the same resource.

2. Exclusive Lock (X Lock)

It allows a transaction to both read and modify a resource. No other transaction can acquire any type of lock on the same resource while an exclusive lock is held.

3. Update Lock (U Lock)

It is used to prevent a deadlock scenario when a transaction intends to update a resource.

4. Schema Lock

It is used to protect the structure of database objects.

5. Bulk Update Lock (BU Lock)

It is used during bulk insert operations to improve performance by reducing the number of locks required.

6. Key-Range Lock

It is used in indexed data to prevent phantom reads (inserting new rows into a range that a transaction has already read).

7. Row-Level Lock

It locks a specific row in a table, allowing other rows to be accessed concurrently.

8. Page-Level Lock

It locks a specific page (a fixed-size block of data) in the database.

9. Table-Level Lock

It locks an entire table. This is simple to implement but can reduce concurrency significantly.

How do we Perform Pagination in API Design?



Pagination is crucial in API design to handle large datasets efficiently and improve performance. Here are six popular pagination techniques:

◆ Offset-based Pagination:

This technique uses an offset and a limit parameter to define the starting point and the number of records to return.

- Example: GET /orders?offset=0&limit=3
- Pros: Simple to implement and understand.
- Cons: Can become inefficient for large offsets, as it requires scanning and skipping rows.

◆ Cursor-based Pagination:

This technique uses a cursor (a unique identifier) to mark the position in the dataset. Typically, the cursor is an encoded string that points to a specific record.

- Example: GET /orders?cursor=xxx
- Pros: More efficient for large datasets, as it doesn't require scanning skipped records.
- Cons: Slightly more complex to implement and understand.

◆ Page-based Pagination:

This technique specifies the page number and the size of each page.

- Example: GET /items?page=2&size=3
- Pros: Easy to implement and use.
- Cons: Similar performance issues as offset-based pagination for large page numbers.

◆ Keyset-based Pagination:

This technique uses a key to filter the dataset, often the primary key or another indexed column.

- Example: GET /items?after_id=102&limit=3
- Pros: Efficient for large datasets and avoids performance issues with large offsets.
- Cons: Requires a unique and indexed key, and can be complex to implement.

◆ Time-based Pagination:

This technique uses a timestamp or date to paginate through records.

- Example: GET /items?start_time=xxx&end_time=yyy
- Pros: Useful for datasets ordered by time, ensures no records are missed if new ones are added.
- Cons: Requires a reliable and consistent timestamp.

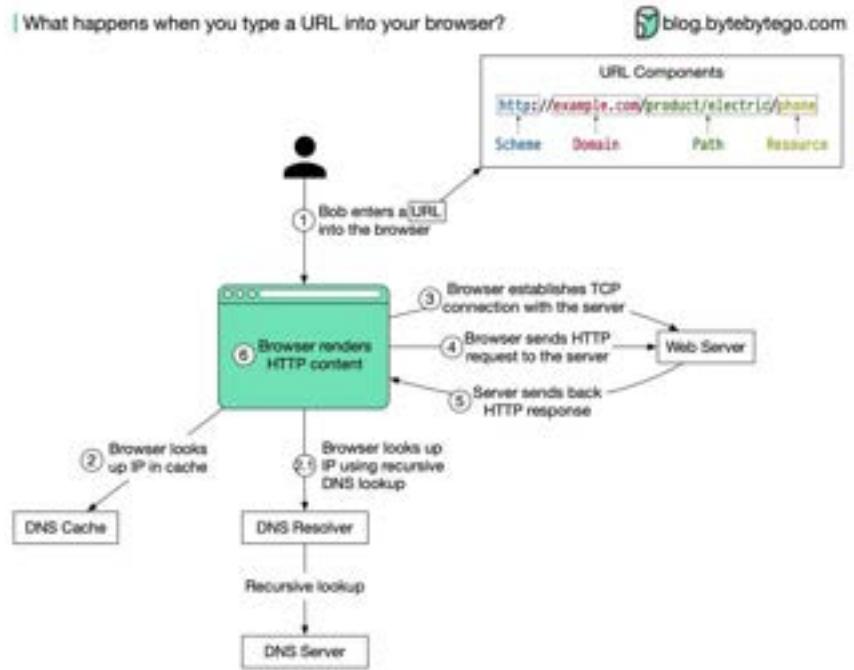
◆ Hybrid Pagination:

This technique combines multiple pagination techniques to leverage their strengths.

Example: Combining cursor and time-based pagination for efficient scrolling through time-ordered records.

- Example: GET /items?cursor=abc&start_time=xxx&end_time=yyy
- Pros: Can offer the best performance and flexibility for complex datasets.
- Cons: More complex to implement and requires careful design.

What happens when you type a URL into your browser?



The diagram below illustrates the steps.

1. Bob enters a URL into the browser and hits Enter. In this example, the URL is composed of 4 parts:

- ◆ scheme - *http://*. This tells the browser to send a connection to the server using HTTP.
- ◆ domain - *example.com*. This is the domain name of the site.
- ◆ path - *product/electric*. It is the path on the server to the requested resource: phone.
- ◆ resource - *phone*. It is the name of the resource Bob wants to visit.

2. The browser looks up the IP address for the domain with a domain name system (DNS) lookup. To make the lookup process fast, data is cached at different layers: browser cache, OS cache, local network cache, and ISP cache.

2.1 If the IP address cannot be found at any of the caches, the browser goes to DNS servers to do a recursive DNS lookup until the IP address is found (this will be covered in another post).

3. Now that we have the IP address of the server, the browser establishes a TCP connection with the server.

4. The browser sends an HTTP request to the server. The request looks like this:

GET /phone HTTP/1.1

Host: example.com

5. The server processes the request and sends back the response. For a successful response (the status code is 200). The HTML response might look like this:

HTTP/1.1 200 OK

Date: Sun, 30 Jan 2022 00:01:01 GMT

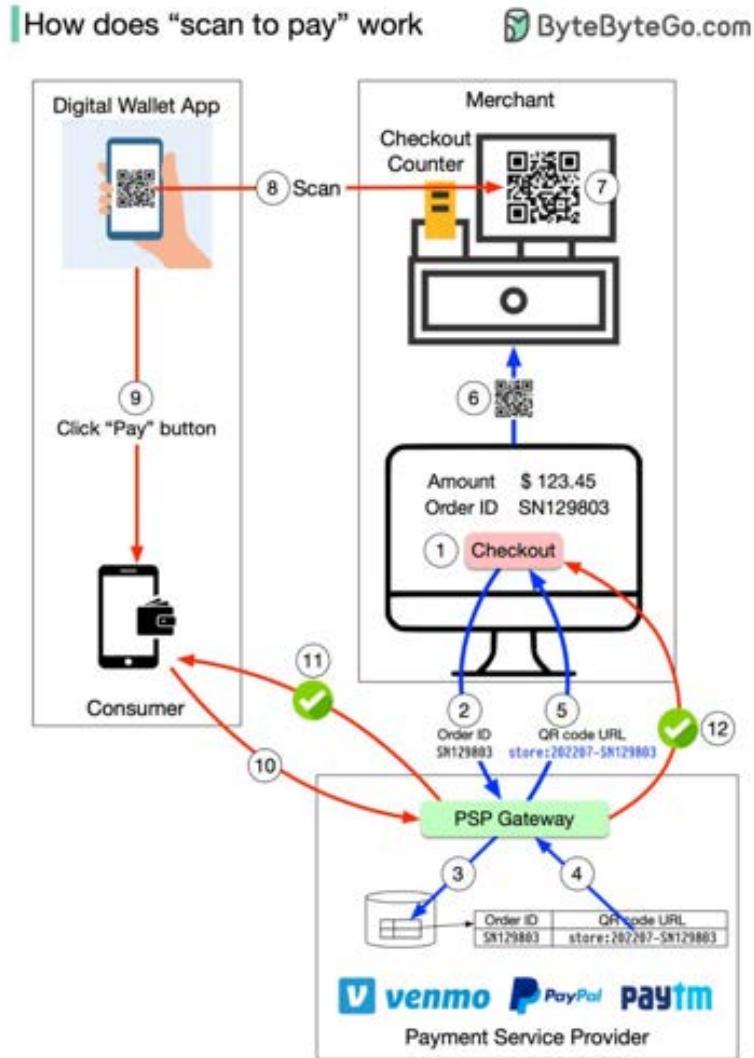
Server: Apache

Content-Type: text/html; charset=utf-8

```
<!DOCTYPE html>
<html lang="en">
Hello world
</html>
```

6. The browser renders the HTML content.

How do you pay from your digital wallet by scanning the QR code?



To understand the process involved, we need to divide the “scan to pay” process into two sub-processes:

1. Merchant generates a QR code and displays it on the screen
2. Consumer scans the QR code and pays

Here are the steps for generating the QR code:

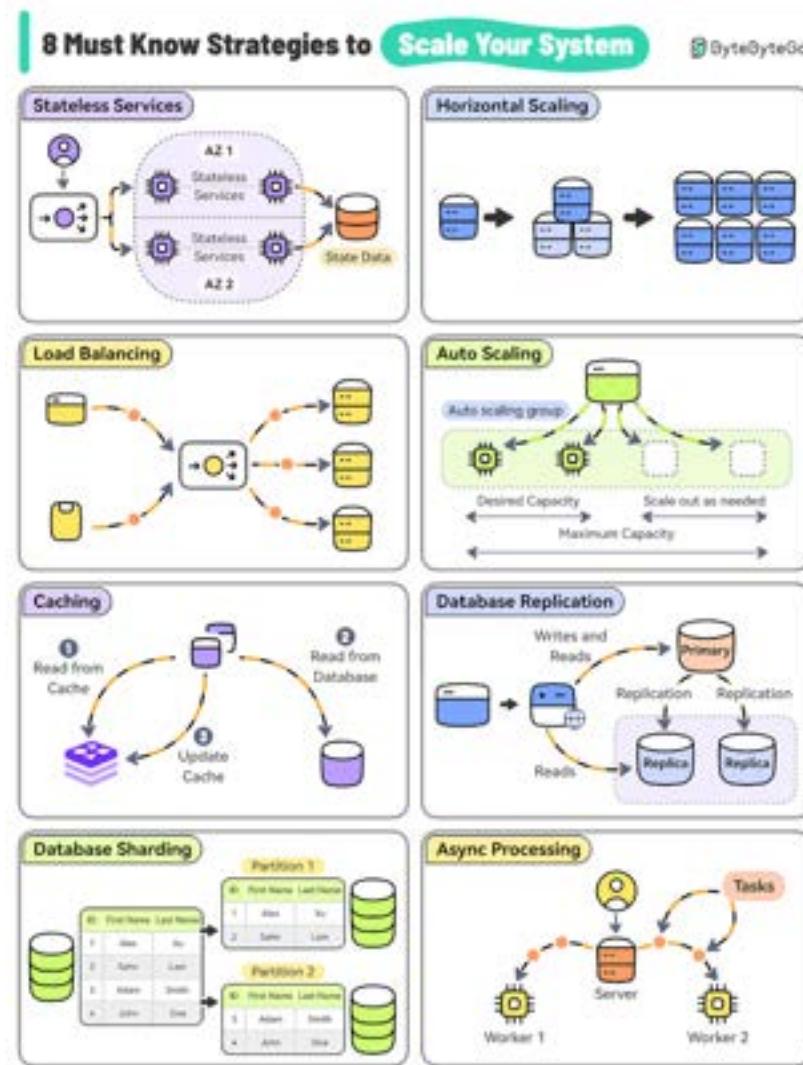
1. When you want to pay for your shopping, the cashier tallies up all the goods and calculates the total amount due, for example, \$123.45. The checkout has an order ID of SN129803. The cashier clicks the “checkout” button.
2. The cashier’s computer sends the order ID and the amount to PSP.
3. The PSP saves this information to the database and generates a QR code URL.

4. PSP's Payment Gateway service reads the QR code URL.
5. The payment gateway returns the QR code URL to the merchant's computer.
6. The merchant's computer sends the QR code URL (or image) to the checkout counter.
7. The checkout counter displays the QR code.

These 7 steps complete in less than a second. Now it's the consumer's turn to pay from their digital wallet by scanning the QR code:

1. The consumer opens their digital wallet app to scan the QR code.
2. After confirming the amount is correct, the client clicks the "pay" button.
3. The digital wallet App notifies the PSP that the consumer has paid the given QR code.
4. The PSP payment gateway marks this QR code as paid and returns a success message to the consumer's digital wallet App.
5. The PSP payment gateway notifies the merchant that the consumer has paid the given QR code.

What do Amazon, Netflix, and Uber have in common?



They are extremely good at scaling their system whenever needed.

Here are 8 must-know strategies to scale your system.

1 - Stateless Services

Design stateless services because they don't rely on server-specific data and are easier to scale.

2 - Horizontal Scaling

Add more servers so that the workload can be shared.

3 - Load Balancing

Use a load balancer to distribute incoming requests evenly across multiple servers.

4 - Auto Scaling

Implement auto-scaling policies to adjust resources based on real-time traffic.

5 - Caching

Use caching to reduce the load on the database and handle repetitive requests at scale.

6 - Database Replication

Replicate data across multiple nodes to scale the read operations while improving redundancy.

7 - Database Sharding

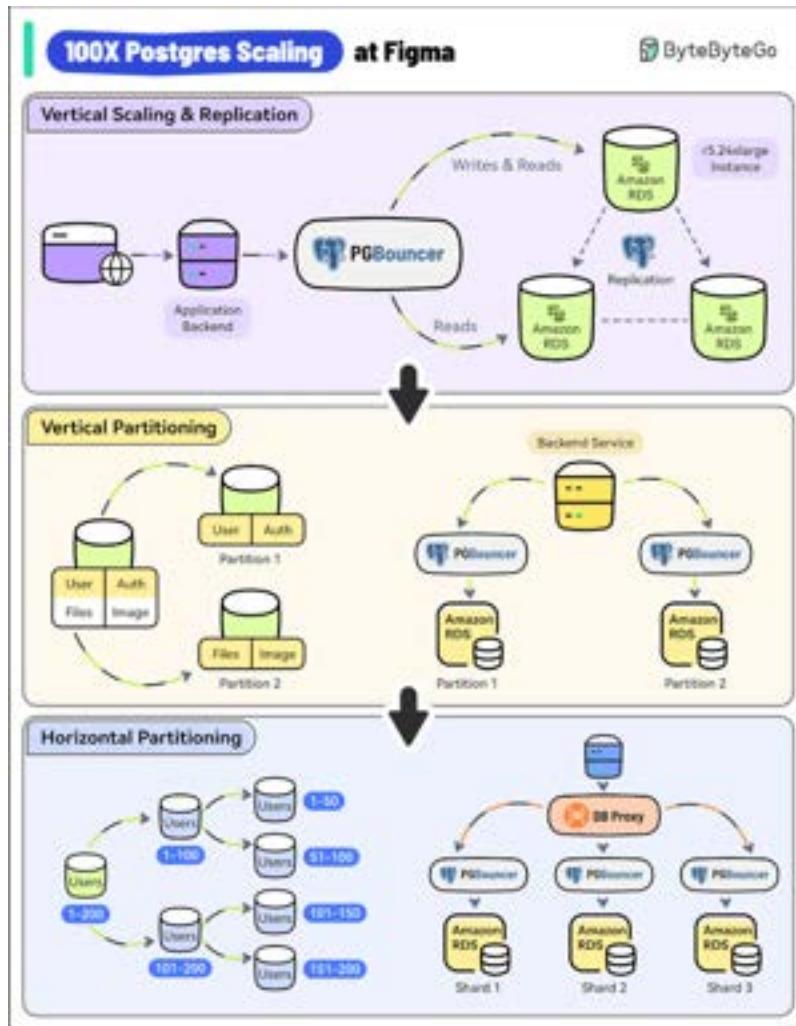
Distribute data across multiple instances to scale the writes as well as reads.

8 - Async Processing

Move time-consuming and resource-intensive tasks to background workers using async processing to scale out new requests.

Over to you: Which other strategies have you used?

100X Postgres Scaling at Figma.



With 3 million monthly users, Figma's user base has increased by 200% since 2018.

As a result, its Postgres database witnessed a whopping 100X growth.

1 - Vertical Scaling and Replication

Figma used a single, large Amazon RDS database.

As a first step, they upgraded to the largest instance available (from r5.12xlarge to r5.24xlarge).

They also created multiple read replicas to scale read traffic and added PgBouncer as a connection pooler to limit the impact of a growing number of connections.

2 - Vertical Partitioning

The next step was vertical partitioning.

They migrated high-traffic tables like “Figma Files” and “Organizations” into their separate databases.

Multiple PgBouncer instances were used to manage the connections for these separate databases.

3 - Horizontal Partitioning

Over time, some tables crossed several terabytes of data and billions of rows.

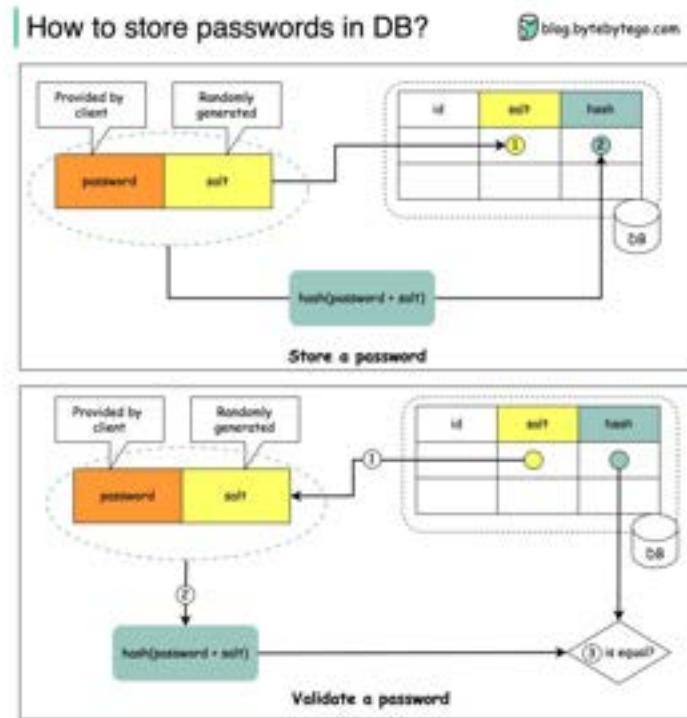
Postgres Vacuum became an issue and max IOPS exceeded the limits of Amazon RDS at the time.

To solve this, Figma implemented horizontal partitioning by splitting large tables across multiple physical databases.

A new DBProxy service was built to handle routing and query execution.

Over to you - Would you have done something differently?

How to store passwords safely in the database and how to validate a password?



Things NOT to do

- Storing passwords in plain text is not a good idea because anyone with internal access can see them.
- Storing password hashes directly is not sufficient because it is prone to precomputation attacks, such as rainbow tables.
- To mitigate precomputation attacks, we salt the passwords.

What is salt?

According to OWASP guidelines, “a salt is a unique, randomly generated string that is added to each password as part of the hashing process”.

How to store a password and salt?

① A salt is not meant to be secret and it can be stored in plain text in the database. It is used to ensure the hash result is unique to each password.

② The password can be stored in the database using the following format: $\text{hash}(\text{password} + \text{salt})$.

How to validate a password?

To validate a password, it can go through the following process:

- ① A client enters the password.
- ② The system fetches the corresponding salt from the database.
- ③ The system appends the salt to the password and hashes it. Let's call the hashed value H1.
- ④ The system compares H1 and H2, where H2 is the hash stored in the database. If they are the same, the password is valid.

Over to you: what other mechanisms can we use to ensure password safety?

Cybersecurity 101 in one picture.



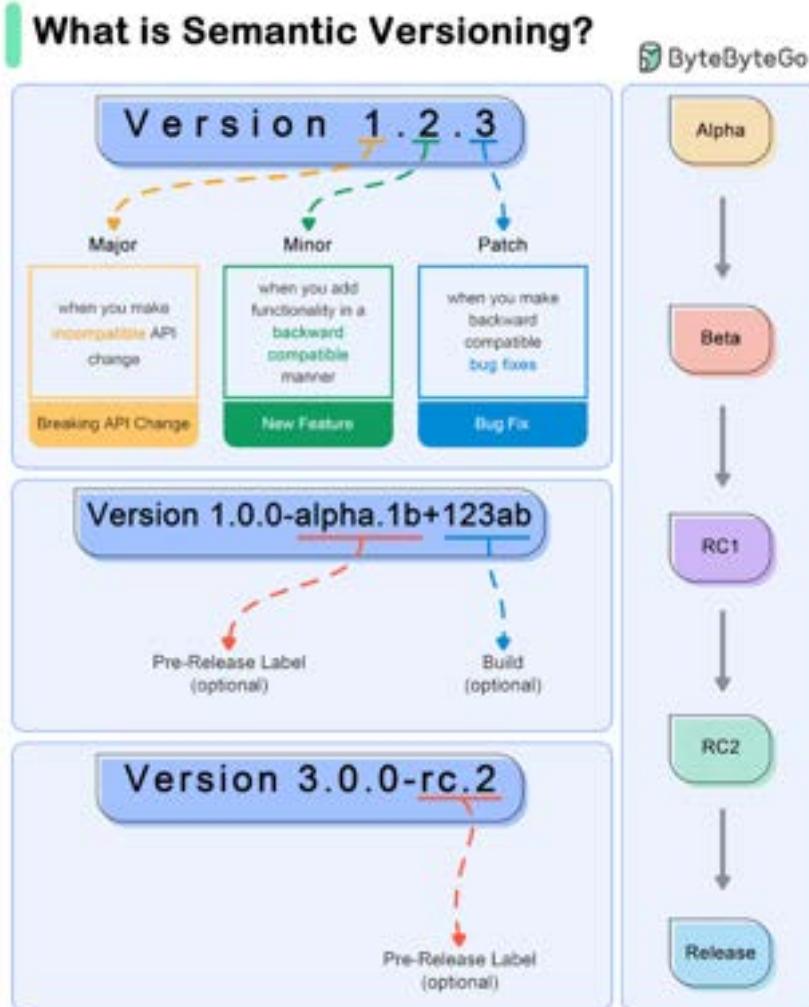
1. Introduction to Cybersecurity
2. The CIA Triad
3. Common Cybersecurity Threats
4. Basic Defense Mechanisms

To combat these threats, several basic defense mechanisms are employed:

- Firewalls: Network security devices that monitor and control incoming and outgoing network traffic.
- Antivirus Software: Programs designed to detect and remove malware.
- Encryption: The process of converting information into a code to prevent unauthorized access.

5. Cybersecurity Frameworks

What do version numbers mean?



Semantic Versioning (SemVer) is a versioning scheme for software that aims to convey meaning about the underlying changes in a release.

- ◆ SemVer uses a three-part version number: MAJOR.MINOR.PATCH.
 - MAJOR version: Incremented when there are incompatible API changes.
 - MINOR version: Incremented when functionality is added in a backward-compatible manner.
 - PATCH version: Incremented when backward-compatible bug fixes are made.

◆ Example Workflow

1 - Initial Development Phase

Start with version 0.1.0.

2 - First Stable Release

Reach a stable release: 1.0.0.

3 - Subsequent Changes

Patch Release: A bug fix is needed for 1.0.0. Update to 1.0.1.

Minor Release: A new, backward-compatible feature is added to 1.0.3. Update to 1.1.0.

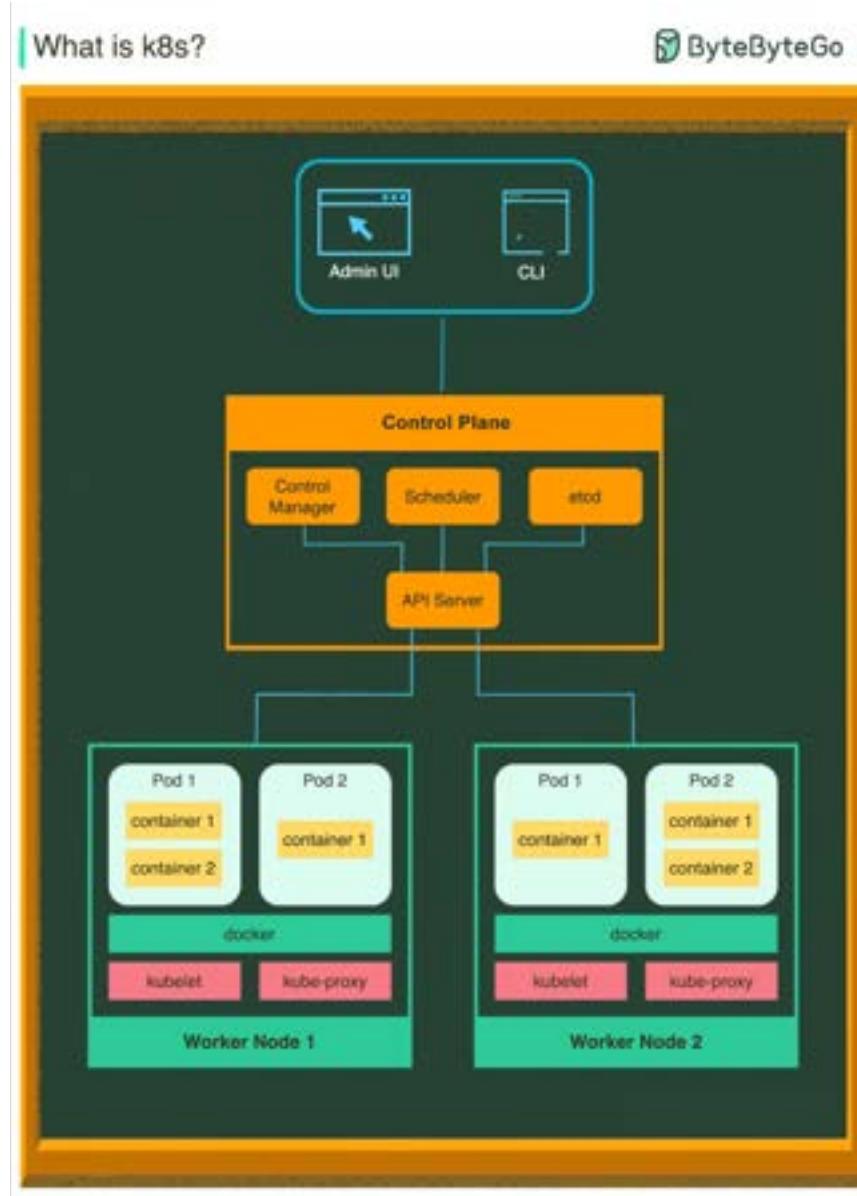
Major Release: A significant change that is not backward-compatible is introduced in 1.2.2. Update to 2.0.0.

4 - Special Versions and Pre-releases

Pre-release Versions: 1.0.0-alpha, 1.0.0-beta, 1.0.0-rc.1.

Build Metadata: 1.0.0+20130313144700.

What is k8s (Kubernetes)?



k8s is a container orchestration system. It is used for container deployment and management. Its design is greatly impacted by Google's internal system Borg.

A k8s cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.

The worker node(s) host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

◆ Control Plane Components

1. API Server

The API server talks to all the components in the k8s cluster. All the operations on pods are executed by talking to the API server.

2. Scheduler

The scheduler watches the workloads on pods and assigns loads on newly created pods.

3. Controller Manager

The controller manager runs the controllers, including Node Controller, Job Controller, EndpointSlice Controller, and ServiceAccount Controller.

4. etcd

etcd is a key-value store used as Kubernetes' backing store for all cluster data.

◆ Nodes

1. Pods

A pod is a group of containers and is the smallest unit that k8s administers. Pods have a single IP address applied to every container within the pod.

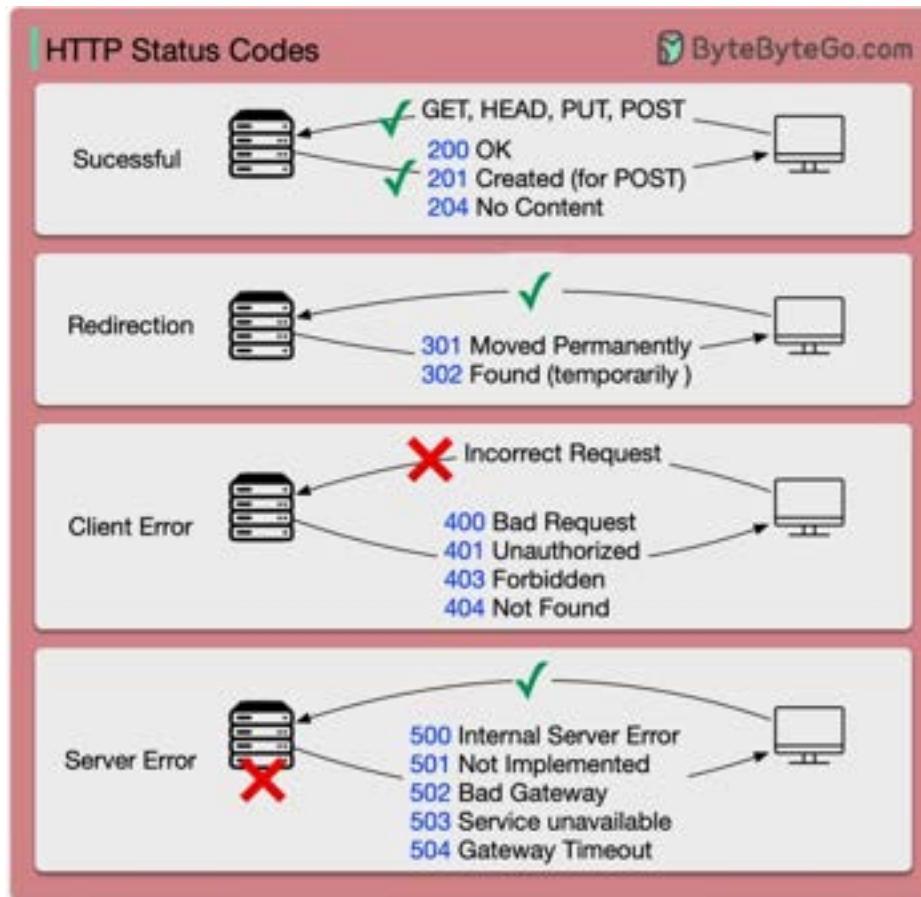
2. Kubelet

An agent that runs on each node in the cluster. It ensures containers are running in a Pod.

3. Kube Proxy

kube-proxy is a network proxy that runs on each node in your cluster. It routes traffic coming into a node from the service. It forwards requests for work to the correct containers.

HTTP Status Code You Should Know



The response codes for HTTP are divided into five categories:

Informational (100-199)

Success (200-299)

Redirection (300-399)

Client Error (400-499)

Server Error (500-599)

These codes are defined in RFC 9110. To save you from reading the entire document (which is about 200 pages), here is a summary of the most common ones:

Over to you: HTTP status code 401 is for Unauthorized. Can you explain the difference between authentication and authorization, and which one does code 401 check for?

18 Most-used Linux Commands You Should Know

18 Most-used Linux Commands			ByteByteGo.com
ls	cd	mkdir	
list files and directories	change current directory	create new directory	
rm	mv	chmod	
remove files or directories	move or rename files or change file or directory	change file or directories permission	
cp	find	grep	
copy files or directories	search for files or directories	search for a pattern in files	
vi	cat	tar	
edit files using text editor	display the content of files	manipulate tarball archive files	
ps	kill	top	
display process information	terminate process by sending a signal	display process and resource usage	
ifconfig	ping	du	
configure network interfaces	test network connectivity between hosts	estimate file space usage	

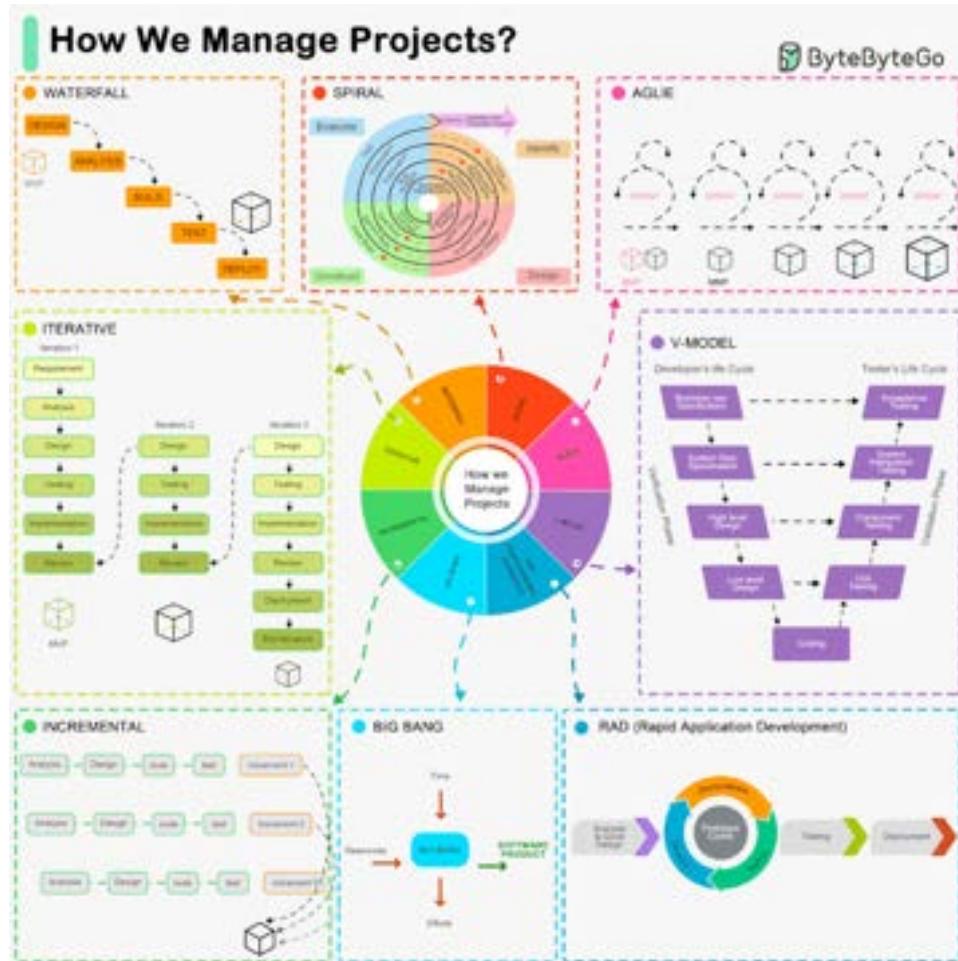
Linux commands are instructions for interacting with the operating system. They help manage files, directories, system processes, and many other aspects of the system. You need to become familiar with these commands in order to navigate and maintain Linux-based systems efficiently and effectively. The following are some popular Linux commands:

- **ls** - List files and directories
- **cd** - Change the current directory
- **mkdir** - Create a new directory
- **rm** - Remove files or directories

- cp - Copy files or directories
- mv - Move or rename files or directories
- chmod - Change file or directory permissions
- grep - Search for a pattern in files
- find - Search for files and directories
- tar - manipulate tarball archive files
- vi - Edit files using text editors
- cat - display the content of files
- top - Display processes and resource usage
- ps - Display processes information
- kill - Terminate a process by sending a signal
- du - Estimate file space usage
- ifconfig - Configure network interfaces
- ping - Test network connectivity between hosts

Over to you: What is your favorite Linux command?

Iterative, Agile, Waterfall, Spiral Model, RAD Model... What are the differences?



The Software Development Life Cycle (SDLC) is a framework that outlines the process of developing software in a systematic way. Here are some of the most common ones:

1 - Waterfall Model:

- A linear and sequential approach.
- Divides the project into distinct phases: Requirements, Design, Implementation, Verification, and Maintenance.

2 - Agile Model:

- Development is done in small, manageable increments called sprints.
- Common Agile methodologies include Scrum, Kanban, and Extreme Programming (XP).

3 - V-Model (Validation and Verification Model):

- An extension of the Waterfall model.
- Each development phase is associated with a testing phase, forming a V shape.

4 - Iterative Model:

- Focuses on building a system incrementally.
- Each iteration builds upon the previous one until the final product is achieved.

5 - Spiral Model:

- Combines iterative development with systematic aspects of the Waterfall model.
- Each cycle involves planning, risk analysis, engineering, and evaluation.

6 - Big Bang Model:

- All coding is done with minimal planning, and the entire software is integrated and tested at once.

7 - RAD Model (Rapid Application Development):

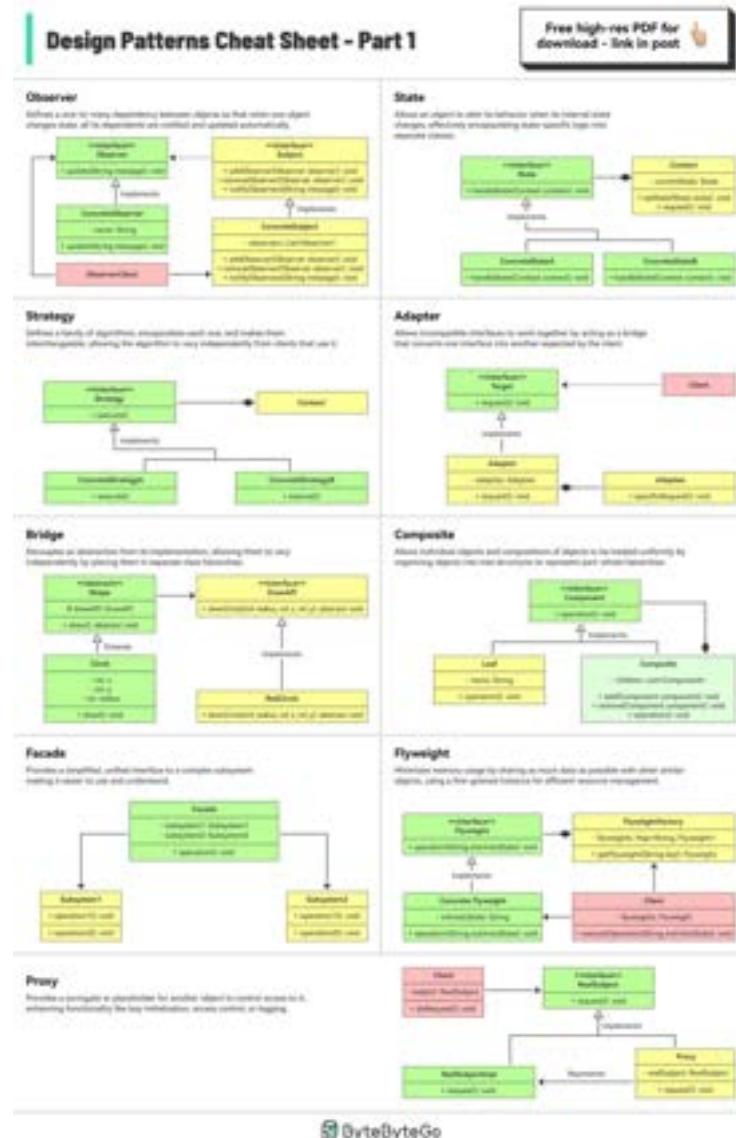
- Emphasizes rapid prototyping and quick feedback.
- Focuses on quick development and delivery.

8 - Incremental Model:

- The product is designed, implemented, and tested incrementally until the product is finished.

Each of these models has its advantages and disadvantages, and the choice of which to use often depends on the specific requirements and constraints of the project at hand.

Design Patterns Cheat Sheet - Part 1 and Part 2

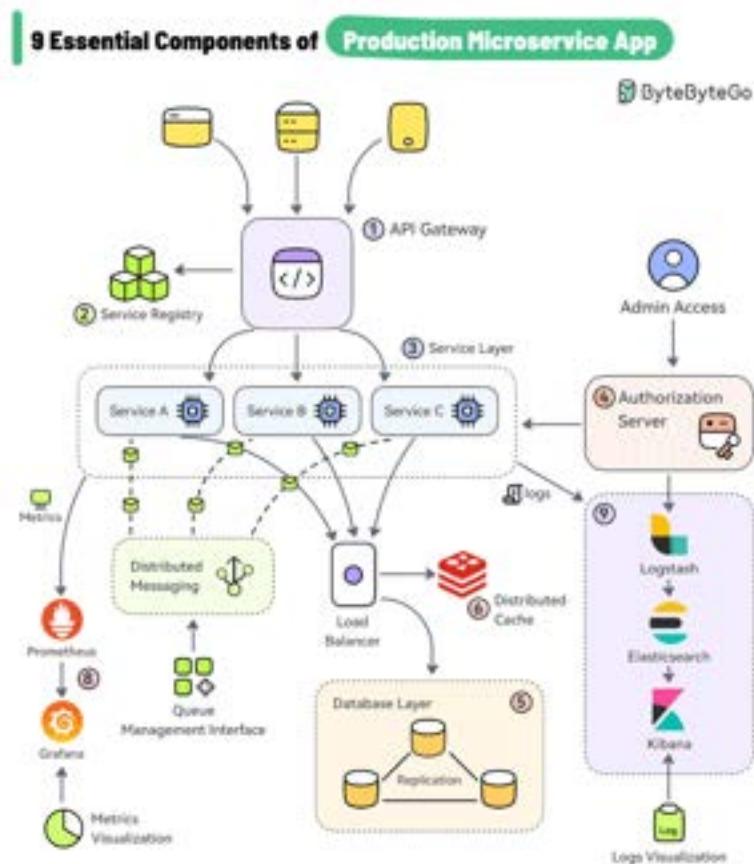


The cheat sheet briefly explains each pattern and how to use it.

What's included?

- Factory
- Builder
- Prototype
- Singleton
- Chain of Responsibility
- And many more!

9 Essential Components of a Production Microservice Application



1 - API Gateway

The gateway provides a unified entry point for client applications. It handles routing, filtering, and load balancing.

2 - Service Registry

The service registry contains the details of all the services. The gateway discovers the service using the registry. For example, Consul, Eureka, Zookeeper, etc.

3 - Service Layer

Each microservices serves a specific business function and can run on multiple instances. These services can be built using frameworks like Spring Boot, NestJS, etc.

4 - Authorization Server

Used to secure the microservices and manage identity and access control. Tools like Keycloak, Azure AD, and Okta can help over here.

5 - Data Storage

Databases like PostgreSQL and MySQL can store application data generated by the services.

6 - Distributed Caching

Caching is a great approach for boosting the application performance. Options include caching solutions like Redis, Couchbase, Memcached, etc.

7 - Async Microservices Communication

Use platforms such as Kafka and RabbitMQ to support async communication between microservices.

8 - Metrics Visualization

Microservices can be configured to publish metrics to Prometheus and tools like Grafana can help visualize the metrics.

9 - Log Aggregation and Visualization

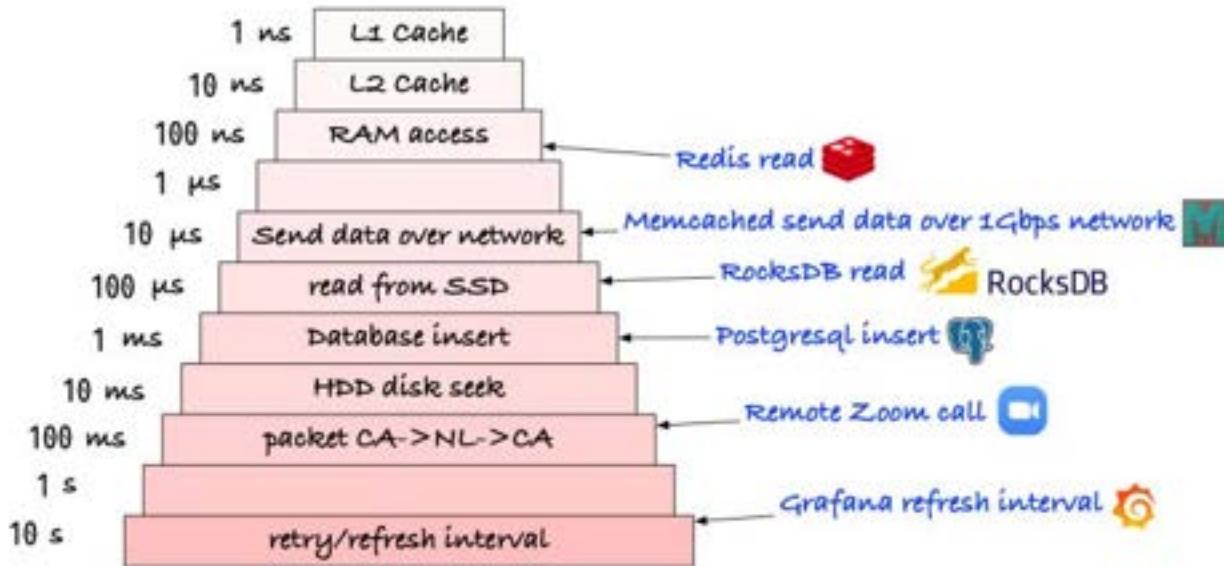
Logs generated by the services are aggregated using Logstash, stored in Elasticsearch, and visualized with Kibana.

Over to you: What else would you add to your production microservice architecture?

Which latency numbers you should know?

Latency Numbers You Should Know

 ByteByteGo.com



Please note those are not precise numbers. They are based on some online benchmarks (Jeff Dean's latency numbers + some other sources).

- L1 and L2 caches: 1 ns, 10 ns

E.g.: They are usually built onto the microprocessor chip. Unless you work with hardware directly, you probably don't need to worry about them.

- RAM access: 100 ns

E.g.: It takes around 100 ns to read data from memory. Redis is an in-memory data store, so it takes about 100 ns to read data from Redis.

- Send 1K bytes over 1 Gbps network: 10 us

E.g.: It takes around 10 us to send 1KB of data from Memcached through the network.

- Read from SSD: 100 us

E.g.: RocksDB is a disk-based K/V store, so the read latency is around 100 us on SSD.

- Database insert operation: 1 ms.

E.g.: Postgresql commit might take 1ms. The database needs to store the data, create the index, and flush logs. All these actions take time.

- Send packet CA->Netherlands->CA: 100 ms

E.g.: If we have a long-distance Zoom call, the latency might be around 100 ms.

- ◆ Retry/refresh internal: 1-10s

E.g: In a monitoring system, the refresh interval is usually set to 5~10 seconds.

Notes

1 ns = 10^{-9} seconds

1 us = 10^{-6} seconds = 1,000 ns

1 ms = 10^{-3} seconds = 1,000 us = 1,000,000 ns

API Gateway 101



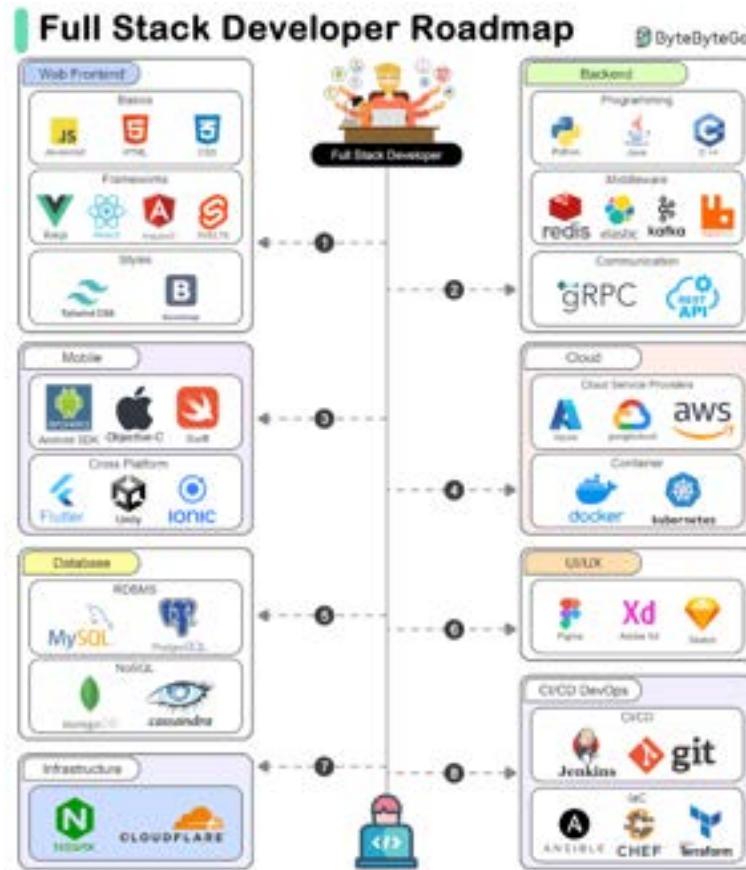
An API gateway is a server that acts as an API front-end, receiving API requests, enforcing throttling and security policies, passing requests to the back-end service, and then returning the appropriate result to the client.

It is essentially a middleman between the client and the server, managing and optimizing API traffic.

Key Functions of an API Gateway

- Request Routing: Directs incoming API requests to the appropriate backend service.
- Load Balancing: Distributes requests across multiple servers to ensure no single server is overwhelmed.
- Security: Implements security measures like authentication, authorization, and data encryption.
- Rate Limiting and Throttling: Controls the number of requests a client can make within a certain period.
- API Composition: Combines multiple backend API requests into a single frontend request to optimize performance.
- Caching: Stores responses temporarily to reduce the need for repeated processing.

A Roadmap for Full-Stack Development.



A full-stack developer needs to be proficient in a wide range of technologies and tools across different areas of software development. Here's a comprehensive look at the technical stacks required for a full-stack developer.

1. Frontend Development

Frontend development involves creating the user interface and user experience of a web application.

2. Backend Development

Backend development involves managing the server-side logic, databases, and integration of various services.

3. Database Development

Database development involves managing data storage, retrieval, and manipulation.

4. Mobile Development

Mobile development involves creating applications for mobile devices.

5. Cloud Computing

Cloud computing involves deploying and managing applications on cloud platforms.

- ◆ 6. UI/UX Design

UI/UX design involves designing the user interface and experience of applications.

- ◆ 7. Infrastructure and DevOps

Infrastructure and DevOps involve managing the infrastructure, deployment, and continuous integration/continuous delivery (CI/CD) of applications.

OAuth 2.0 Flows



Authorization Code Flow: The most common OAuth flow. After user authentication, the client receives an authorization code and exchanges it for an access token and refresh token.

Client Credentials Flow: Designed for single-page applications. The access token is returned directly to the client without an intermediate authorization code.

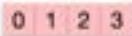
Implicit Code Flow: Designed for single-page applications. The access token is returned directly to the client without an intermediate authorization code.

Resource Owner Password Grant Flow: Allows users to provide their username and password directly to the client, which then exchanges them for an access token.

Over to you - So which one do you think is something that you should use next in your application?

10 Key Data Structures We Use Every Day

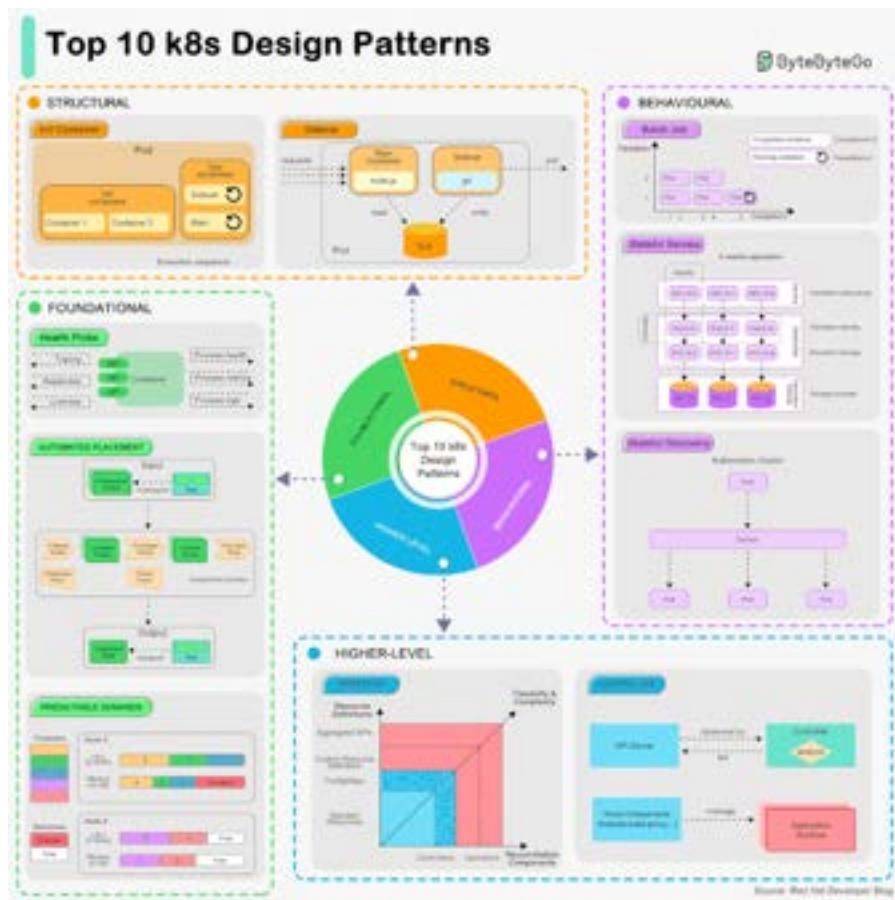
10 Data Structures Used in Daily Life ByteByteGo.com

Data Structure	Illustration	Use Cases
List		Twitter feeds
Array		Math operations Large data sets
Stack		Undo/Redo of word editor
Queue		Printer jobs User actions in game
Heap		Task scheduling
Tree		HTML document AI decision
Suffix Tree		Search string in document
Graph		Friendship tracking Path finding
R-tree		Nearest neighbour
Hash Table		Caching systems

- list: keep your Twitter feeds
- stack: support undo/redo of the word editor
- queue: keep printer jobs, or send user actions in-game
- hash table: cashing systems
- Array: math operations
- heap: task scheduling
- tree: keep the HTML document, or for AI decision
- suffix tree: for searching string in a document
- graph: for tracking friendship, or path finding
- r-tree: for finding the nearest neighbor
- vertex buffer: for sending data to GPU for rendering

Over to you: Which additional data structures have we overlooked?

Top 10 k8s Design Patterns



♦ Foundational Patterns

These patterns are the fundamental principles for applications to be automated on k8s, regardless of the application's nature.

1. Health Probe Pattern

This pattern requires that every container must implement observable APIs for the platform to manage the application.

2. Predictable Demands Pattern

This pattern requires that we should declare application requirements and runtime dependencies. Every container should declare its resource profile.

3. Automated Placement Pattern

This pattern describes the principles of Kubernetes' scheduling algorithm.

♦ Structural Patterns

These patterns focus on structuring and organizing containers in a Pod.

4. Init Container Pattern

This pattern has a separate life cycle for initialization-related tasks.

5. Sidecar Pattern

This pattern extends a container's functionalities without changing it.

◆ Behavioral Patterns

These patterns describe the life cycle management of a Pod. Depending on the type of the workload, it can run as a service or a batch job.

6. Batch Job Pattern

This pattern is used to manage isolated atomic units of work.

7. Stateful Service Pattern

This pattern creates distributed stateful applications.

8. Service Discovery Pattern

This pattern describes how clients discover the services.

◆ Higher-Level Patterns

These patterns focus on higher-level application management.

9. Controller Pattern

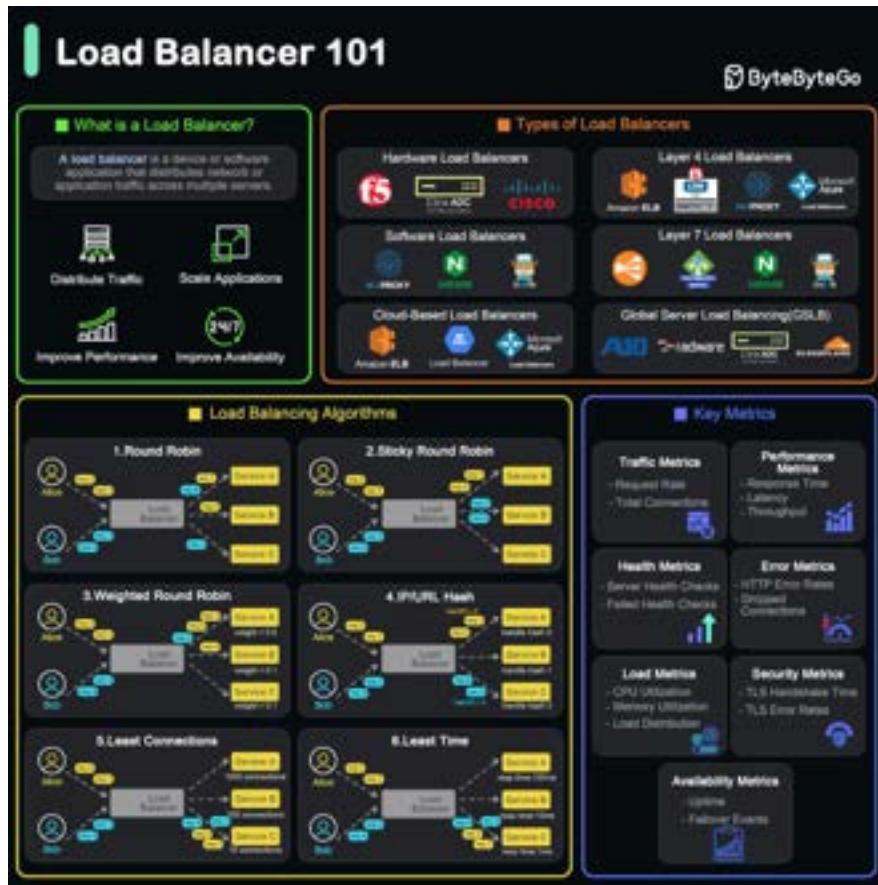
This pattern monitors the current state and reconciles with the declared target state.

10. Operator Pattern

This pattern defines operational knowledge in an algorithmic and automated form.

Reference: developers.redhat.com/blog/2020/05/11/top-10-must-know-kubernetes-design-patterns

What is a Load Balancer?



A load balancer is a device or software application that distributes network or application traffic across multiple servers.

◆ What Does a Load Balancer Do?

1. Distributes Traffic
2. Ensures Availability and Reliability
3. Improves Performance
4. Scales Applications

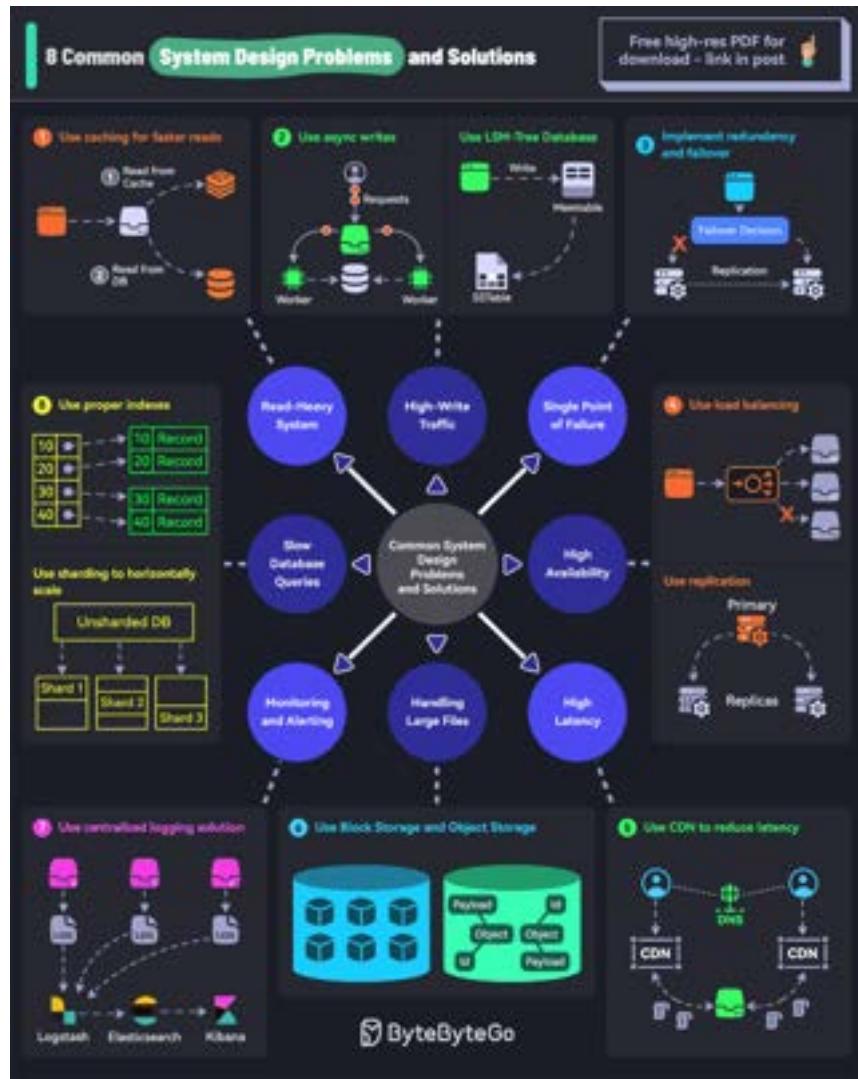
◆ Types of Load Balancers

1. **Hardware Load Balancers**: These are physical devices designed to distribute traffic across servers.
2. **Software Load Balancers**: These are applications that can be installed on standard hardware or virtual machines.

3. Cloud-based Load Balancers: Provided by cloud service providers, these load balancers are integrated into the cloud infrastructure. Examples include AWS Elastic Load Balancer, Google Cloud Load Balancing, and Azure Load Balancer.
4. Layer 4 Load Balancers (Transport Layer): Operate at the transport layer (OSI Layer 4) and make forwarding decisions based on IP address and TCP/UDP ports.
5. Layer 7 Load Balancers (Application Layer): Operate at the application layer (OSI Layer 7) .
6. Global Server Load Balancing (GSLB): Distributes traffic across multiple geographical locations to improve redundancy and performance on a global scale.

8 Common System Design Problems and Solutions

Do you know those 8 common problems in large-scale production systems and their solutions?
Time to test your skills!



1 - Read-Heavy System

Use caching to make the reads faster.

2 - High-Write Traffic

Use async workers to process the writes
Use databases powered by LSM-Trees

3 - Single Point of Failure

Implement redundancy and failover mechanisms for critical components like databases.

4 - High Availability

Use load balancing to ensure that requests go to healthy server instances.

Use database replication to improve durability and availability.

5 - High Latency

Use a content delivery network to reduce latency

6 - Handling Large Files

Use block storage and object storage to handle large files and complex data.

7 - Monitoring and Alerting

Use a centralized logging system using something like the ELK stack.

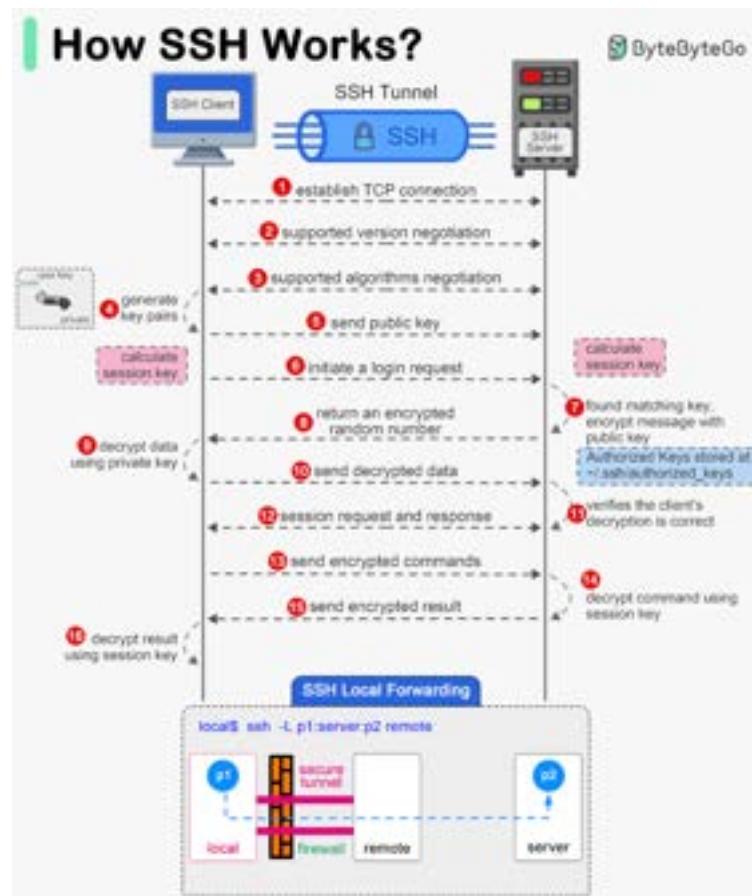
8 - Slower Database Queries

Use proper indexes to optimize queries.

Use sharding to scale the database horizontally.

Over to you: What other common problems and solutions have you seen?

How does SSH work?



SSH (Secure Shell) is a network protocol used to securely connect to remote machines over an unsecured network. It encrypts the connection and provides various mechanisms for authentication and data transfer.

SSH has two versions: SSH-1 and SSH-2. SSH-2 was standardized by the IETF.

It has three main layers: Transport Layer, Authentication Layer, and Connection Layer.

1. Transport Layer

The Transport Layer provides encryption, integrity, and data protection to ensure secure communication between the client and server.

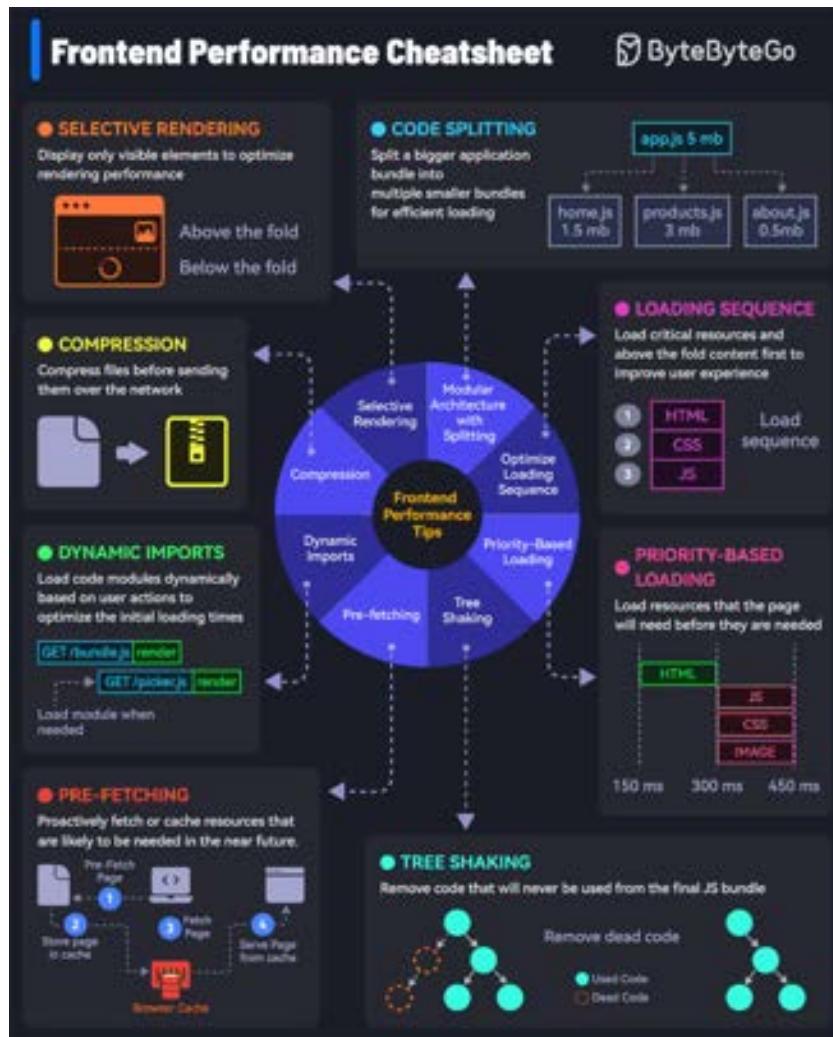
2. Authentication Layer

The Authentication Layer verifies the identity of the client to ensure that only authorized users can access the server.

3. Connection Layer

The Connection Layer multiplexes the encrypted and authenticated communication into multiple logical channels.

How to load your websites at lightning speed?



Check out these 8 tips to boost frontend performance:

1 - Compression

Compress files and minimize data size before transmission to reduce network load.

2 - Selective Rendering/Windowing

Display only visible elements to optimize rendering performance. For example, in a dynamic list, only render visible items.

3 - Modular Architecture with Code Splitting

Split a bigger application bundle into multiple smaller bundles for efficient loading.

4 - Priority-Based Loading

Prioritize essential resources and visible (or above-the-fold) content for a better user experience.

5 - Pre-loading

Fetch resources in advance before they are requested to improve loading speed.

6 - Tree Shaking or Dead Code Removal

Optimize the final JS bundle by removing dead code that will never be used.

7 - Pre-fetching

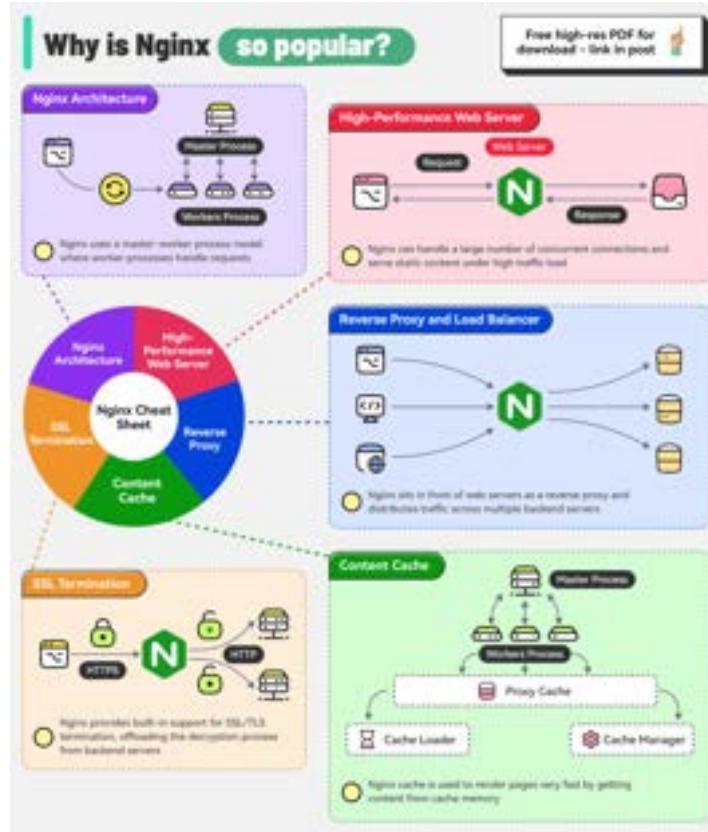
Proactively fetch or cache resources that are likely to be needed soon.

8 - Dynamic Imports

Load code modules dynamically based on user actions to optimize the initial loading times.

Over to you: What other frontend performance tips would you add to this cheat sheet?

Why is Nginx so popular?



Nginx is a high-performance web server and reverse proxy.

It follows a master-worker process model that contributes to its stability, scalability, and efficient resource utilization.

The master process is responsible for reading the configuration and managing worker processes. Worker processes handle incoming connections using an event-driven non-blocking I/O model.

Due to its architecture, Nginx excels in supporting multiple features such as:

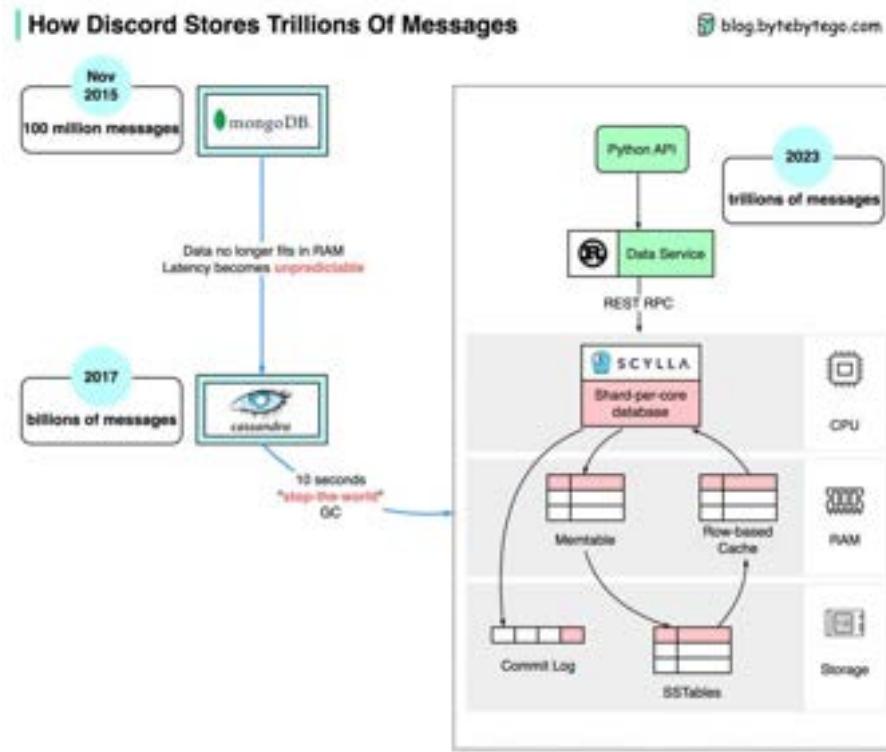
1 - High-Performance Web Server

2 - Reverse Proxy and Load Balancing

3 - Content Cache

4 - SSL Termination

Over to you: Do you know any other features supported by Nginx?



How Discord Stores Trillions of Messages

The diagram below shows the evolution of message storage at Discord:

MongoDB → Cassandra → ScyllaDB

In 2015, the first version of Discord was built on top of a single MongoDB replica. Around Nov 2015, MongoDB stored 100 million messages and the RAM couldn't hold the data and index any longer. The latency became unpredictable. Message storage needs to be moved to another database. Cassandra was chosen.

In 2017, Discord had 12 Cassandra nodes and stored billions of messages.

At the beginning of 2022, it had 177 nodes with trillions of messages. At this point, latency was unpredictable, and maintenance operations became too expensive to run.

There are several reasons for the issue:

- Cassandra uses the LSM tree for the internal data structure. The reads are more expensive than the writes. There can be many concurrent reads on a server with hundreds of users, resulting in hotspots.
- Maintaining clusters, such as compacting SSTables, impacts performance.

- Garbage collection pauses would cause significant latency spikes

ScyllaDB is Cassandra compatible database written in C++. Discord redesigned its architecture to have a monolithic API, a data service written in Rust, and ScyllaDB-based storage.

The p99 read latency in ScyllaDB is 15ms compared to 40-125ms in Cassandra. The p99 write latency is 5ms compared to 5-70ms in Cassandra.

Over to you: What kind of NoSQL database have you used? How do you like it?

How does Garbage Collection work?



Garbage collection is an automatic memory management feature used in programming languages to reclaim memory no longer used by the program.

Java

Java provides several garbage collectors, each suited for different use cases:

1. Serial Garbage Collector: Best for single-threaded environments or small applications.
2. Parallel Garbage Collector: Also known as the "Throughput Collector."
3. CMS (Concurrent Mark-Sweep) Garbage Collector: Low-latency collector aiming to minimize pause times.
4. G1 (Garbage-First) Garbage Collector: Aims to balance throughput and latency.
5. Z Garbage Collector (ZGC): A low-latency garbage collector designed for applications that require large heap sizes and minimal pause times.

Python

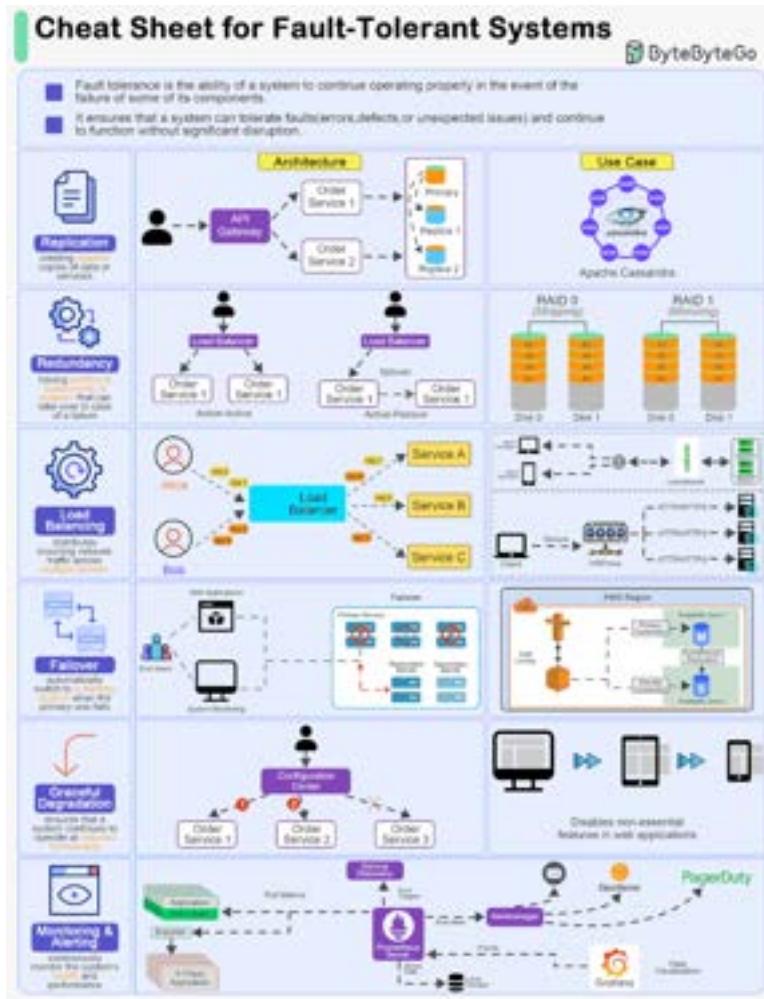
Python's garbage collection is based on reference counting and a cyclic garbage collector:

1. Reference Counting: Each object has a reference count; when it reaches zero, the memory is freed.
2. Cyclic Garbage Collector: Handles circular references that can't be resolved by reference counting.

GoLang

Concurrent Mark-and-Sweep Garbage Collector: Go's garbage collector operates concurrently with the application, minimizing stop-the-world pauses.

A Cheat Sheet for Designing Fault-Tolerant Systems.



Designing fault-tolerant systems is crucial for ensuring high availability and reliability in various applications. Here are six top principles of designing fault-tolerant systems:

1. Replication

Replication involves creating multiple copies of data or services across different nodes or locations.

2. Redundancy

Redundancy refers to having additional components or systems that can take over in case of a failure.

3. Load Balancing

Load balancing distributes incoming network traffic across multiple servers to ensure no single server becomes a point of failure.

4. Failover Mechanisms

Failover mechanisms automatically switch to a standby system or component when the primary one fails.

5. Graceful Degradation

Graceful degradation ensures that a system continues to operate at reduced functionality rather than completely failing when some components fail.

6. Monitoring and Alerting

Continuously monitor the system's health and performance, and set up alerts for any anomalies or failures.

If you don't know trade-offs, you DON'T KNOW system design.



10 System Design Tradeoffs You Cannot Ignore

1 - Vertical vs Horizontal Scaling

Vertical scaling is adding more resources (CPU, RAM) to an existing server.

Horizontal scaling means adding more servers to the pool.

2 - SQL vs NoSQL

SQL databases organize data into tables of rows and columns.

NoSQL is ideal for applications that need a flexible schema.

3 - Batch vs Stream Processing

Batch processing involves collecting data and processing it all at once. For example, daily billing processes.

Stream processing processes data in real time. For example, fraud detection processes.

4 - Normalization vs Denormalization

Normalization splits data into related tables to ensure that each piece of information is stored only once.

Denormalization combines data into fewer tables for better query performance.

5 - Consistency vs Availability

Consistency is the assurance of getting the most recent data every single time.

Availability is about ensuring that the system is always up and running, even if some parts are having problems.

6 - Strong vs Eventual Consistency

Strong consistency is when data updates are immediately reflected.

Eventual consistency is when data updates are delayed before being available across nodes.

7 - REST vs GraphQL

With REST endpoints, you gather data by accessing multiple endpoints.

With GraphQL, you get more efficient data fetching with specific queries but the design cost is higher.

8 - Stateful vs Stateless

A stateful system remembers past interactions.

A stateless system does not keep track of past interactions.

9 - Read-Through vs Write-Through Cache

A read-through cache loads data from the database in case of a cache miss.

A write-through cache simultaneously writes data updates to the cache and storage.

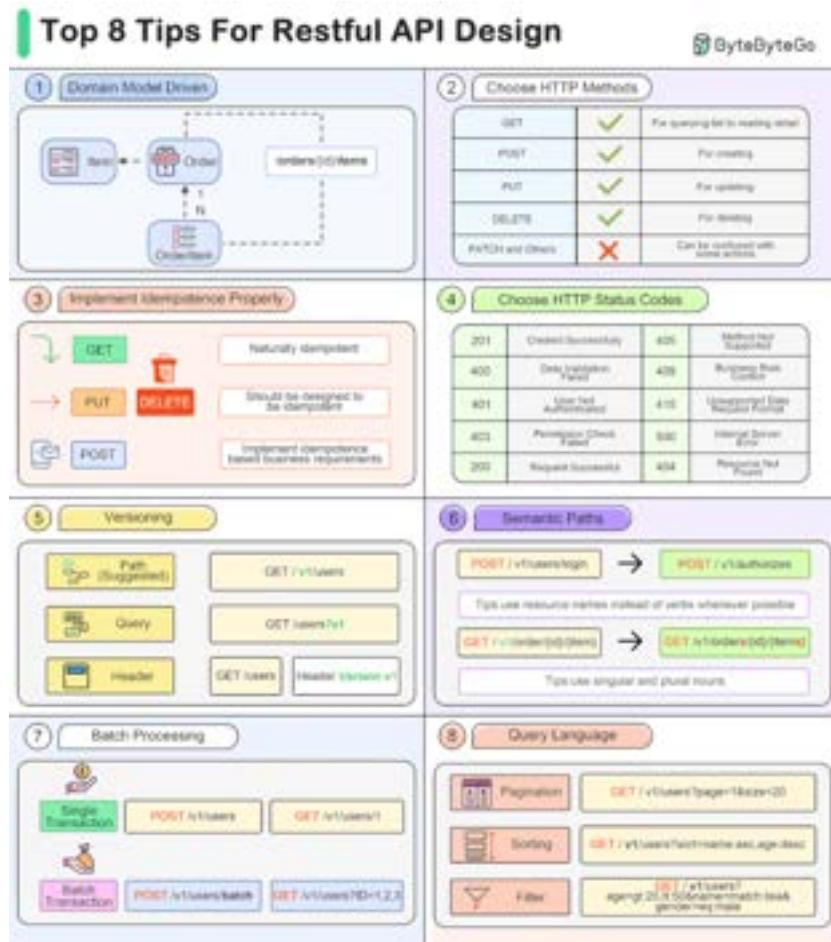
10 - Sync vs Async Processing

In synchronous processing, tasks are performed one after another.

In asynchronous processing, tasks can run in the background. New tasks can be started without waiting for a new task.

Over to you: Which other tradeoffs have you encountered?

8 Tips for Efficient API Design.



◆ Domain Model Driven

When designing the path structure of a RESTful API, we can refer to the domain model.

◆ Choose Proper HTTP Methods

Defining a few basic HTTP Methods can simplify the API design. For example, PATCH can often be a problem for teams.

◆ Implement Idempotence Properly

Designing for idempotence in advance can improve the robustness of an API. GET method is idempotent, but POST needs to be designed properly to be idempotent.

- ◆ Choose Proper HTTP Status Codes

Define a limited number of HTTP status codes to use to simplify application development.

- ◆ Versioning

Designing the version number for the API in advance can simplify upgrade work.

- ◆ Semantic Paths

Using semantic paths makes APIs easier to understand, so that users can find the correct APIs in the documentation.

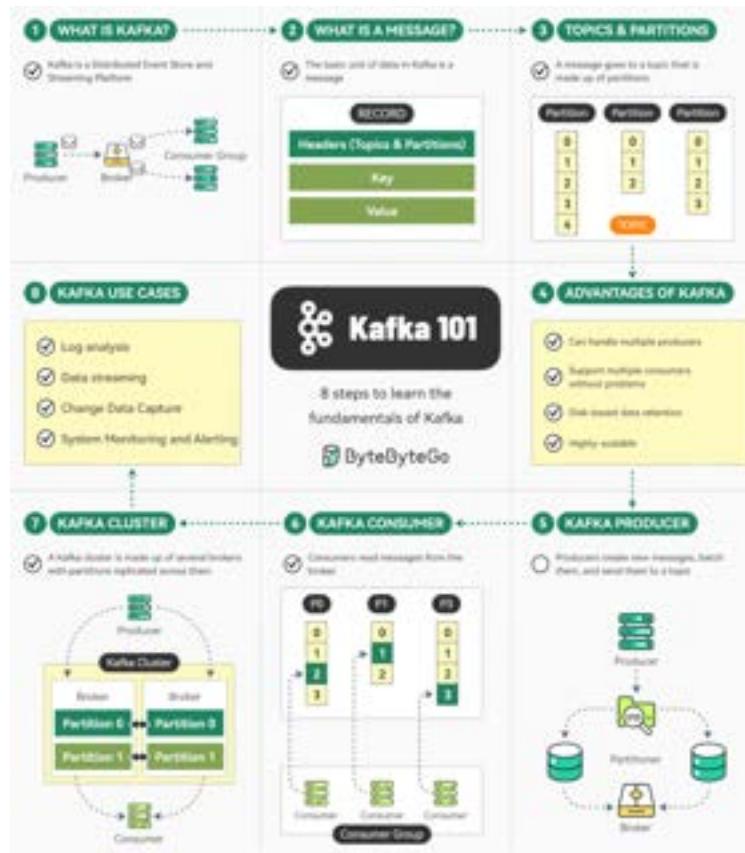
- ◆ Batch Processing

Use batch/bulk as a keyword and place it at the end of the path.

- ◆ Query Language

Designing a set of query rules makes the API more flexible. For example, pagination, sorting, filtering etc.

The Ultimate Kafka 101 You Cannot Miss



Kafka is super-popular but can be overwhelming in the beginning.

Here are 8 simple steps that can help you understand the fundamentals of Kafka.

1 - What is Kafka?

Kafka is a distributed event store and a streaming platform. It began as an internal project at LinkedIn and now powers some of the largest data pipelines in the world in orgs like Netflix, Uber, etc.

2 - Kafka Messages

Message is the basic unit of data in Kafka. It's like a record in a table consisting of headers, key, and value.

3 - Kafka Topics and Partitions

Every message goes to a particular Topic. Think of the topic as a folder on your computer. Topics also have multiple partitions.

4 - Advantages of Kafka

Kafka can handle multiple producers and consumers, while providing disk-based data retention and high scalability.

5 - Kafka Producer

Producers in Kafka create new messages, batch them, and send them to a Kafka topic. They also take care of balancing messages across different partitions.

6 - Kafka Consumer

Kafka consumers work together as a consumer group to read messages from the broker.

7 - Kafka Cluster

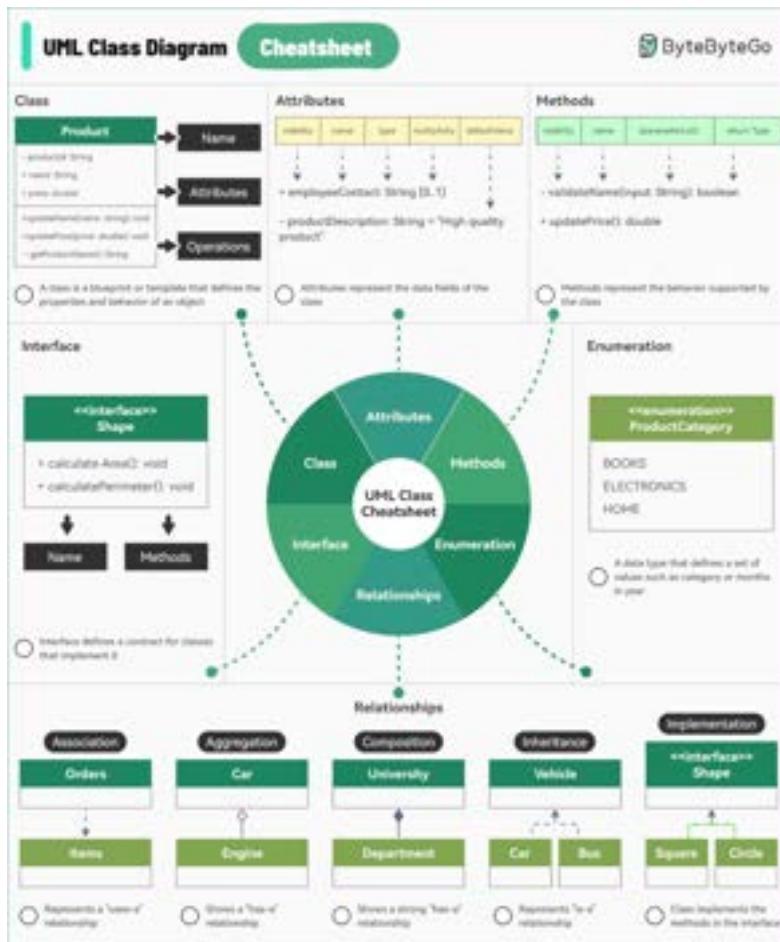
A Kafka cluster consists of several brokers where each partition is replicated across multiple brokers to ensure high availability and redundancy.

8 - Use Cases of Kafka

Kafka can be used for log analysis, data streaming, change data capture, and system monitoring.

Over to you: What else would you add to get a better understanding of Kafka?

A Cheatsheet for UML Class Diagrams



UML is a standard way to visualize the design of your system and class diagrams are used across the industry.

They consist of:

1 - Class

Acts as the blueprint that defines the properties and behavior of an object.

2 - Attributes

Attributes in a UML class diagram represent the data fields of the class.

3 - Methods

Methods in a UML class diagram represent the behavior that a class can perform.

4 - Interfaces

Defines a contract for classes that implement it. Includes a set of methods that the implementing classes must provide.

5 - Enumeration

A special data type that defines a set of named values such as product category or months in a year.

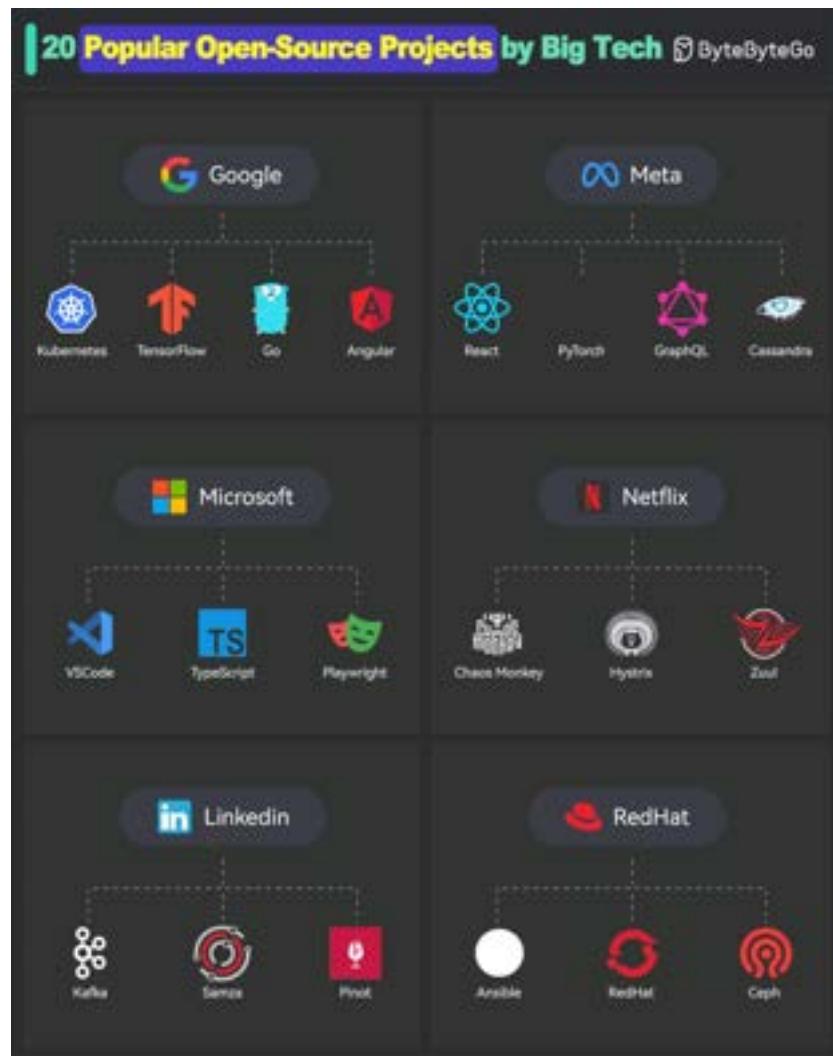
6 - Relationships

Determines how one class is related to another. Some common relationships are as follows:

- Association
- Aggregation
- Composition
- Inheritance
- Implementation

Over to you: What other building blocks have you seen in UML class diagrams?

20 Popular Open Source Projects Started or Supported By Big Companies



1 - Google

- Kubernetes
- TensorFlow
- Go
- Angular

2 - Meta

- React
- PyTorch
- GraphQL
- Cassandra

3 - Microsoft

- VSCode
- TypeScript
- Playwright

4 - Netflix

- Chaos Monkey
- Hystrix
- Zuul

5 - LinkedIn

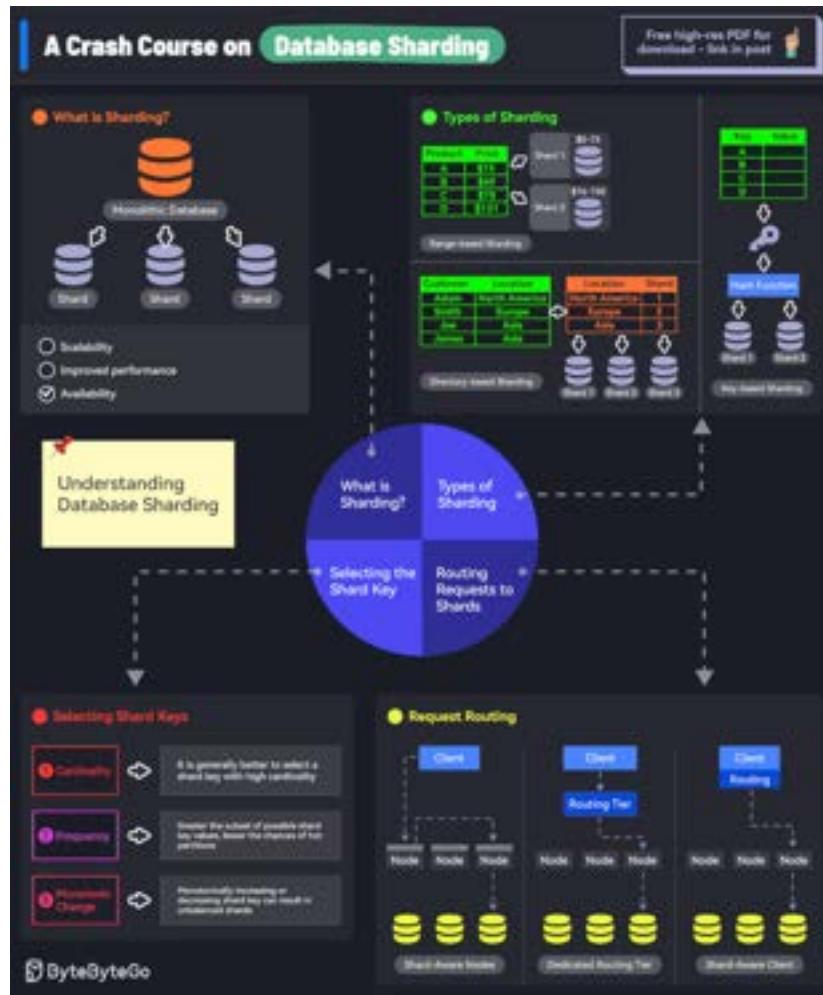
- Kafka
- Samza
- Pinot

6 - RedHat

- Ansible
- OpenShift
- Ceph Storage

Over to you: Which other project would you add to the list?

A Crash Course on Database Sharding



1 What is Sharding?

Sharding is an architectural pattern that addresses the challenges of managing and querying large datasets in databases. It involves splitting a large database into smaller, more manageable parts called shards.

The benefits of sharding are scalability, improved performance, and better availability.

2 Types of Sharding

Three main sharding strategies are as follows:

- Range-based Sharding: Split database rows based on a range of values.
- Key/Hash-based Sharding: Assign a particular key to a shard using a hash function
- Directory-based Sharding: Relies on a lookup table to determine the distribution of records across shards.

3 Selecting the Shard Key

- Choosing an appropriate shard key is crucial for an effective sharding strategy. Designers should consider several factors such as:
 - Cardinality: Number of possible values that a shard key can have. It's better to have a shard key with high cardinality.
 - Frequency: Represents how often a particular shard key value appears. Higher frequency can result in hotspots.
 - Monotonic Change: Refers to the shard key value increasing or decreasing over time. Monotonic increases or decreases can result in unbalanced shards.

4 Request Routing

- With sharding, the most critical consideration is determining which query should go to which shard. There are three main approaches:
 - Shard-aware Node: The client can contact any node and the node will serve/redirect the request to the correct shard.
 - Routing Tier: Client requests go to a dedicated routing tier that determines the node responsible for handling the request.
 - Shard-aware Client: Clients are aware of the shard distribution across the nodes.

Is PostgreSQL eating the database world?



It seems that no matter what the use case, PostgreSQL supports it. When in doubt, you can simply use PostgreSQL.

1 - TimeSeries

PostgreSQL embraces Timescale, a powerful time-series database extension for efficient handling of time-stamped data.

2 - Machine Learning

With pgVector and PostgresML, Postgres can support machine learning capabilities and vector similarity searches.

3 - OLAP

Postgres can support OLAP with tools such as Hydra, Citus, and pg_analytics.

4 - Derived

Even derived databases such as DuckDB, FerretDB, CockroachDB, AlloyDB, YugaByte DB, Supabase, etc provide PostgreSQL.

5 - GeoSpatial

PostGIS extends PostgreSQL with geospatial capabilities, enabling you to easily store, query, and analyze geographic data.

6 - Search

Postgres extensions like pgroonga, ParadeDB, and ZomboDB provide full-text search, text indexing, and data parsing capabilities.

7 - Federated

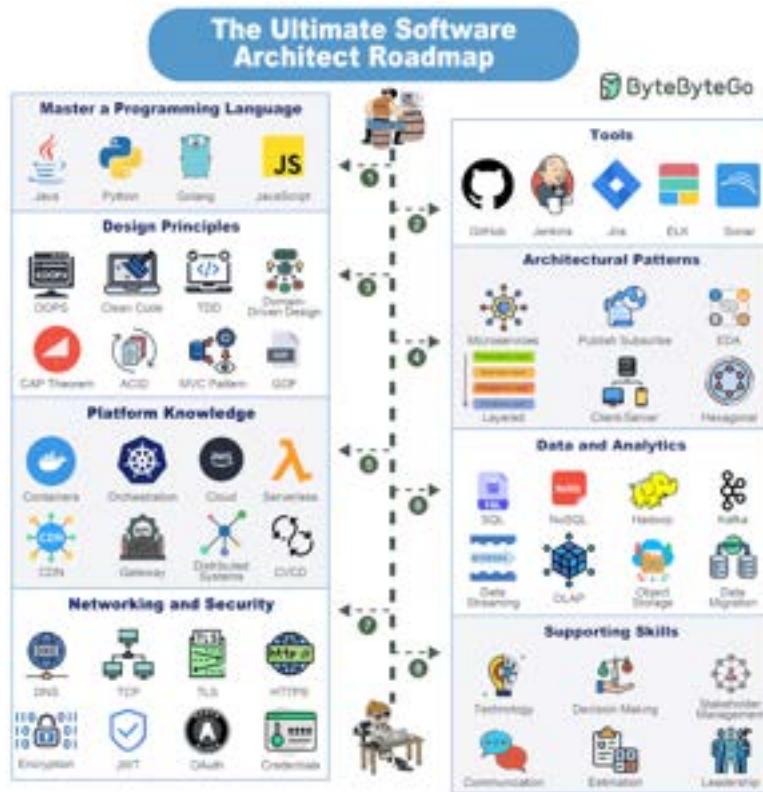
Postgres seamlessly integrates with various data sources such as MongoDB, MySQL, Redis, Oracle, ParquetDB, SQLite, etc, enabling federated querying and data access.

8 - Graph

Apache AGE and EdgeDB are graph databases built on top of PostgreSQL. Also, pg_graphql is an extension that provides GraphQL support for Postgres.

Over to you: Have you seen any other use cases of PostgreSQL?

The Ultimate Software Architect Knowledge Map



Becoming a Software Architect is a journey where you are always learning. But there are some things you must definitely strive to know.

1 - Master a Programming Language

Look to master 1-2 programming languages such as Java, Python, Golang, JavaScript, etc.

2 - Tools

Build proficiency with key tools such as GitHub, Jenkins, Jira, ELK, Sonar, etc.

3 - Design Principles

Learn about important design principles such as OOPS, Clean Code, TDD, DDD, CAP Theorem, MVC Pattern, ACID, and GOF.

4 - Architectural Principles

Become proficient in multiple architectural patterns such as Microservices, Publish-Subscribe, Layered, Event-Driven, Client-Server, Hexagonal, etc.

5 - Platform Knowledge

Get to know about several platforms such as containers, orchestration, cloud, serverless, CDN, API Gateways, Distributed Systems, and CI/CD

6 - Data Analytics

Build a solid knowledge of data and analytics components like SQL and NoSQL databases, data streaming solutions with Kafka, object storage, data migration, OLAP, and so on.

7 - Networking and Security

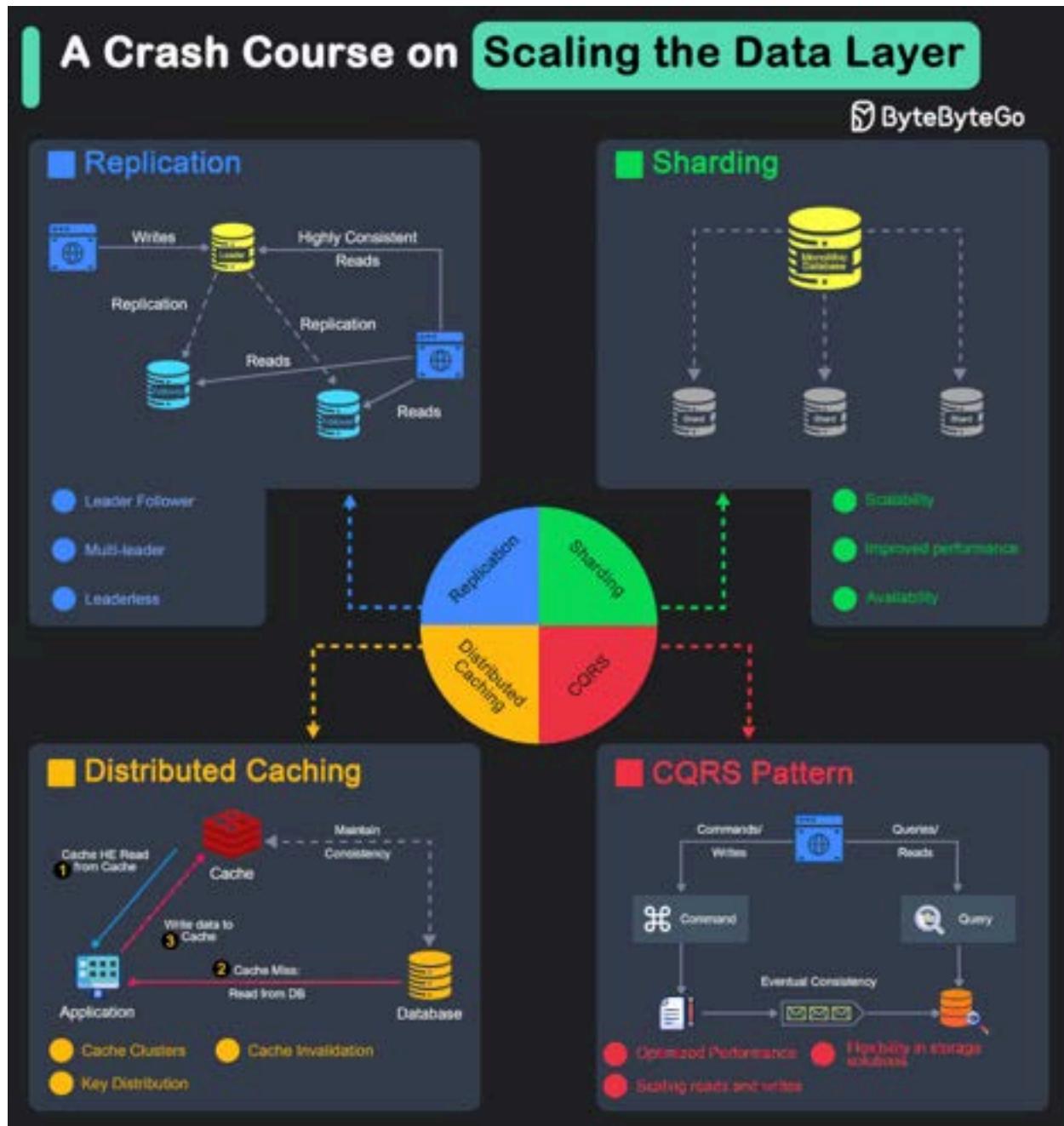
Learn about networking and security concepts such as DNS, TCP, TLS, HTTPS, Encryption, JWT, OAuth, and Credential Management.

8 - Supporting Skills

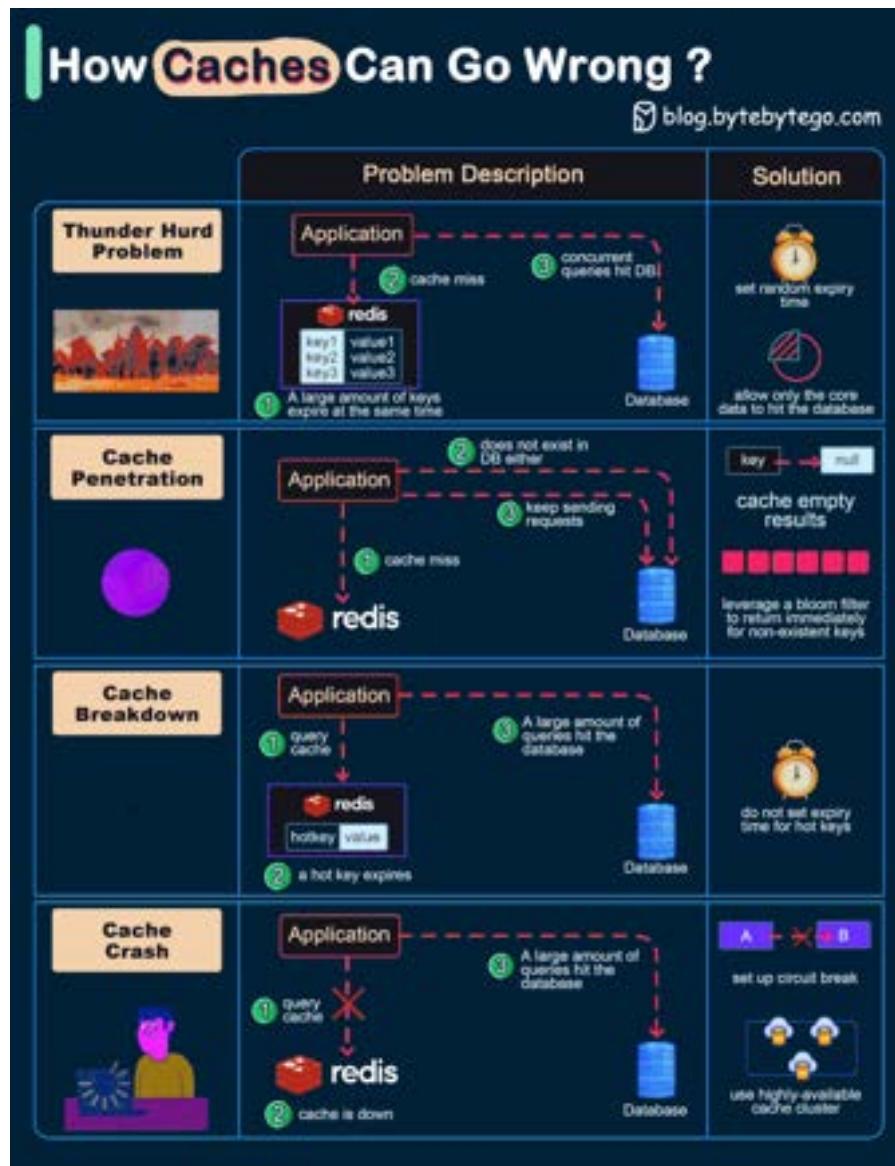
Apart from technical, software architects also need several supporting skills such as decision-making, technology knowledge, stakeholder management, communication, estimation, leadership, etc.

Over to you - What else would you add to the roadmap?

A Crash Course on Scaling the Data Layer



How can Cache Systems go wrong?



The diagram below shows 4 typical cases where caches can go wrong and their solutions.

1. Thunder herd problem

This happens when a large number of keys in the cache expire at the same time. Then the query requests directly hit the database, which overloads the database.

There are two ways to mitigate this issue: one is to avoid setting the same expiry time for the keys, adding a random number in the configuration; the other is to allow only the core business data to hit the database and prevent non-core data to access the database until the cache is back up.

2. Cache penetration

This happens when the key doesn't exist in the cache or the database. The application cannot retrieve relevant data from the database to update the cache. This problem creates a lot of pressure on both the cache and the database.

To solve this, there are two suggestions. One is to cache a null value for non-existent keys, avoiding hitting the database. The other is to use a bloom filter to check the key existence first, and if the key doesn't exist, we can avoid hitting the database.

3. Cache breakdown

This is similar to the thunder herd problem. It happens when a hot key expires. A large number of requests hit the database.

Since the hot keys take up 80% of the queries, we do not set an expiration time for them.

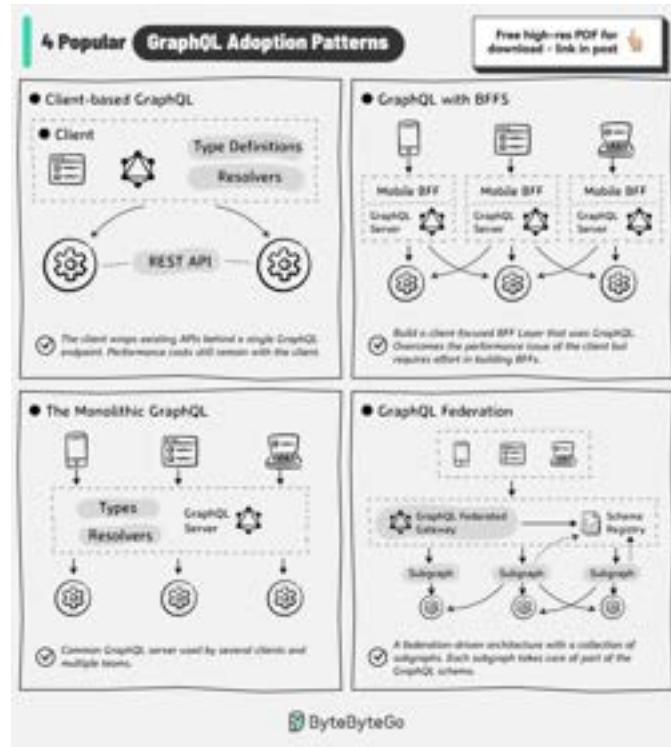
4. Cache crash

This happens when the cache is down and all the requests go to the database.

There are two ways to solve this problem. One is to set up a circuit breaker, and when the cache is down, the application services cannot visit the cache or the database. The other is to set up a cluster for the cache to improve cache availability.

Over to you: Have you met any of these issues in production?

4 Popular GraphQL Adoption Patterns



Typically, teams begin their GraphQL journey with a basic architecture where a client application queries a single GraphQL server.

However, multiple patterns are available:

1 - Client-based GraphQL

The client wraps existing APIs behind a single GraphQL endpoint. This approach improves the developer experience but the client still bears the performance costs of aggregating data.

2 - GraphQL with BFFs

BFF or Backend-for-Frontends adds a new layer where each client has a dedicated BFF service. GraphQL is a natural fit to build a client-focused intermediary layer.

Performance and developer experience for the clients is improved but there's a tradeoff in building and maintaining BFFs.

3 - The Monolithic GraphQL

Multiple teams share one codebase for a GraphQL server used by several clients. Also, a single team owns a GraphQL API that is accessed by multiple client teams.

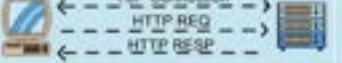
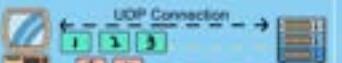
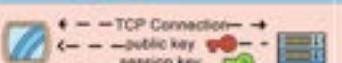
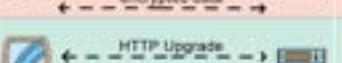
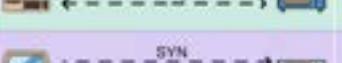
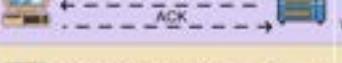
4 - GraphQL Federation

This involves consolidating multiple graphs into a supergraph.

GraphQL Federated Gateway takes care of routing the requests to the downstream subgraph services that take care of a specific part of the GraphQL schema. This approach maintains ownership of data with the domain team while avoiding duplication of effort.

Over to you: Which GraphQL adoption approach have you seen or used?

Top 8 Popular Network Protocols

8 Popular Network Protocols		
Protocol	How does It Work?	Use Cases
HTTP		 Web Browsing
HTTP/3 (QUIC)		 IoT Virtual Reality
HTTPS		 Web Browsing
WebSocket		 Live Chat Real-Time Data Transmission
TCP		 Web Browsing Email Protocols
UDP		 Video Conferencing
SMTP		 Sending/Receiving Emails
FTP		 Upload/Download Files

Network protocols are standard methods of transferring data between two computers in a network.

1. HTTP (HyperText Transfer Protocol)
2. HTTP/3
3. HTTPS (HyperText Transfer Protocol Secure)
4. WebSocket
5. TCP (Transmission Control Protocol)
6. UDP (User Datagram Protocol)
7. SMTP (Simple Mail Transfer Protocol)
8. FTP (File Transfer Protocol)

11 Things I learned about API Development from POST/CON



2024 by Postman.

These learnings can be put into three major buckets:

API Development Toolkit

- API workflows are a critical requirement to build real-world applications. Postman Flows enables developers to automate API workflow visually without writing any code.
- Monitoring and observability have become harder. Postman Insights delivers a fully automated, API-first approach to monitoring.
- TTFC or Time to First Call is a crucial parameter for developers. Postman's Related Request feature can get you a 200 OK response in less than a second.

- Authorization is a key friction point in API development. Postman reduces the friction by providing a streamlined auth experience by encrypting and storing credentials locally.
- AI is transforming API development. Postbot helps you write tests for your APIs, document APIs, debug requests, and visualize responses.

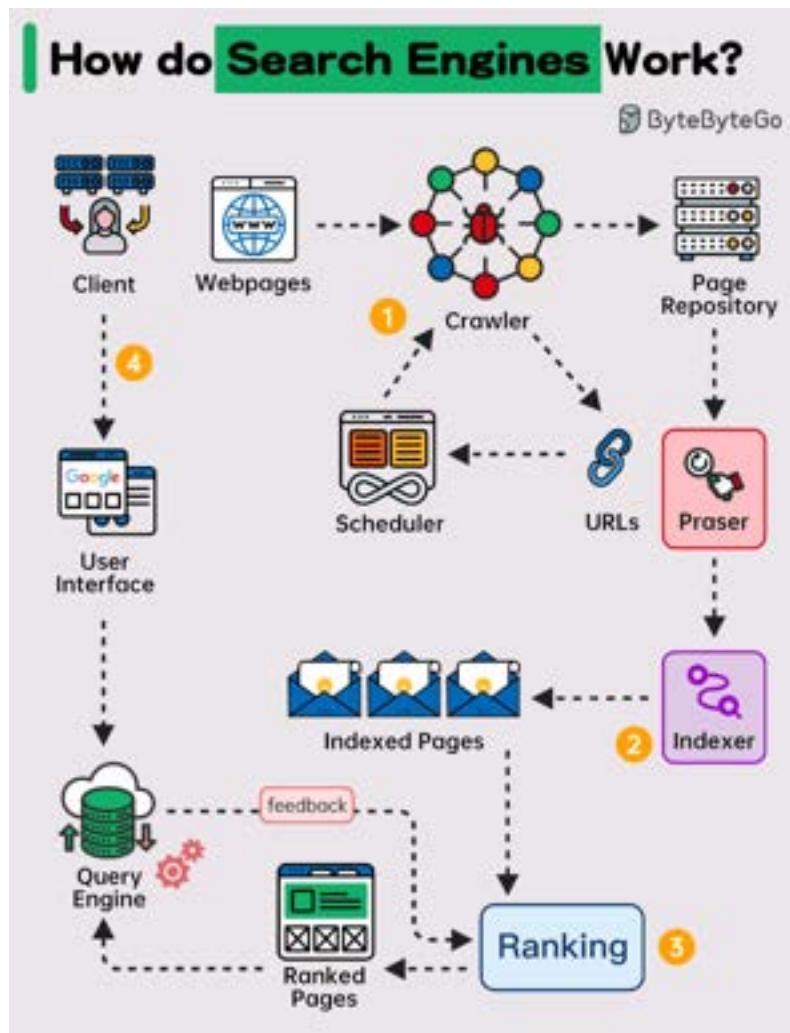
API Collaboration

- A streamlined developer workflow is crucial for modern API development. Postman supports this goal with scripting, tests, visualizers, and team collaboration.
- API collaboration should bring producers and consumers together. Postman enables collaboration using collections, workspaces, and private API networks.
- Designing a delightful API experience is a cross-functional effort that goes beyond writing good documentation.

API Community Growth

- Developers like to test their APIs right inside their editor. Postman's VS Code extension makes it a reality.
- Product teams want to market their APIs to the global community. Postman's public workspaces makes the process seamless.
- Postman API Network can be a game-changer for organizations to reach new customers.

How do Search Engines really Work?



The diagram below shows a high-level walk-through of a search engine.

▶ Step 1 - Crawling

Web Crawlers scan the internet for web pages. They follow the URL links from one page to another and store URLs in the URL store. The crawlers discover new content, including web pages, images, videos, and files.

▶ Step 2 - Indexing

Once a web page is crawled, the search engine parses the page and indexes the content found on the page in a database. The content is analyzed and categorized. For example, keywords, site quality, content freshness, and many other factors are assessed to understand what the page is about.

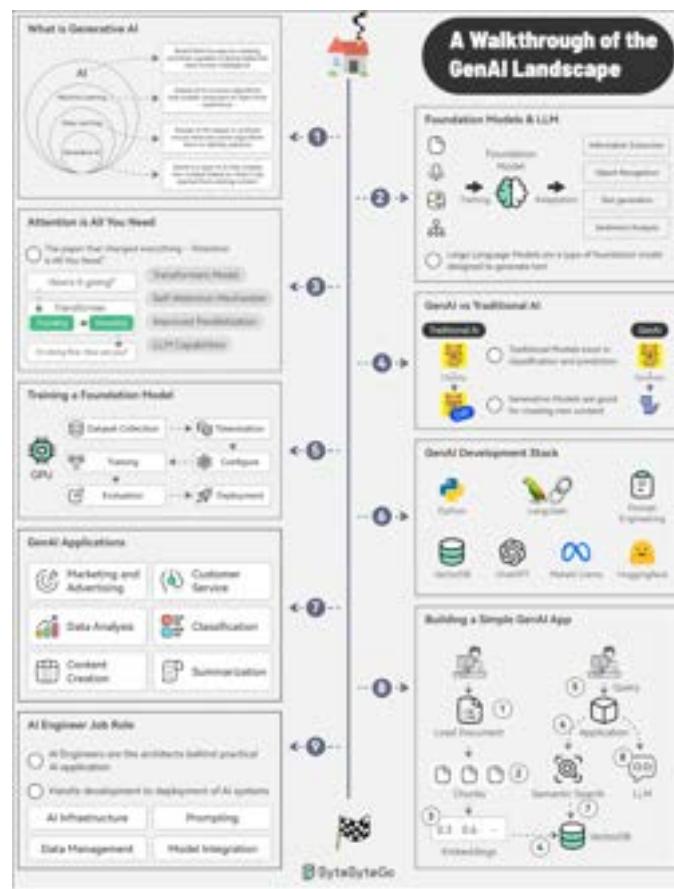
Step 3 - Ranking

Search engines use complex algorithms to determine the order of search results. These algorithms consider various factors, including keywords, pages' relevance, content quality, user engagement, page load speed, and many others. Some search engines also personalize results based on the user's past search history, location, device, and other personal factors.

Step 4 - Querying

When a user performs a search, the search engine sifts through its index to provide the most relevant results.

The Ultimate Walkthrough of the Generative AI Landscape



Generative AI and LLMs are fast becoming a game-changer in the business world. Everyone wants to learn more about it.

The landscape covers the following points:

- 1 - What is GenAI?
- 2 - Foundational Models and LLMs
- 3 - “Attention is All You Need” and its impact
- 4 - GenAI vs Traditional AI
- 5 - How to train a foundation model?
- 6 - The GenAI Development Stack (LLMs, Frameworks, Programming Languages, etc.)
- 7 - GenAI Applications
- 8 - Designing a simple GenAI application
- 9 - The AI Engineer Job Role

Over to you: What else would you add to the GenAI landscape?

Cheatsheet for Relatioable Database Design



Cheatsheet on Relational Database Design

A relational database is a type of database that organizes data into structured tables, also known as relations. These tables consist of rows (records) and columns (fields).

Some key points to know about Relational Database Design

1 - SQL

SQL is the standard programming language used to interact with relational databases. It supports fundamental operations for data manipulation, data definition, and data control.

2 - Fundamental RDBMS Concepts

There are some fundamental RDBMS concepts such as table, row, column, primary key, foreign key, join, index, and view.

3 - Keys in Relational Databases

Different types of keys are as follows:

Primary Key: A column or combination of columns uniquely identifying each record in a table.

Surrogate Key: Artificial key generated by the database system or a globally unique identifier that has no inherent meaning to the data.

Foreign Key: A column or a combination of columns in one table that references the primary key of another table.

4 - Relation Types

Relationships between tables play a key role in defining how data is connected. Three main types of relationships are:

One-to-One Relationship: A record in one table is associated with one record in another table.

One-to-Many Relationship: A record in one table is associated with multiple records in another table

Many-to-Many Relationship: Records in both tables can have multiple records in the other table.

5 - Joins

Joins act as bridges, connecting different tables based on their relationship. They are extremely useful when you need to retrieve data from multiple tables. There are 3 main types of joins:

Inner Join

Right Outer Join

Left Outer Join

Over to you: What else should you know about relational database design?

My Favorite 10 Soft Skill Books that Can Help You Become a Better Developer



Productivity & Personal Development

- 1 - Deep Work by Cal Newport
- 2 - Atomic Habits by James Clear
- 3 - The Effective Executive by Peter Drucker

Communication Skills

- 1 - Crucial Conversations by Kerry Patterson et al.
- 2 - How to Win Friends and Influence People by Dale Carnegie

Leadership & Team Dynamics

- 1 - Extreme Ownership by Jocko Willink and Leif Babin
- 2 - The Five Dysfunctions of a Team by Patrick Lencioni

3 - Start with Why by Simon Sinek

Design & Craftsmanship

1 - The Clean Coder by Robert Martin

2 - The Design of Everyday Things by Dan Norman

Over to you: What is your favorite book?

REST API Authentication Methods



Authentication in REST APIs acts as the crucial gateway, ensuring that solely authorized users or applications gain access to the API's resources.

Some popular authentication methods for REST APIs include:

1. Basic Authentication:

Involves sending a username and password with each request, but can be less secure without encryption.

When to use:

Suitable for simple applications where security and encryption aren't the primary concern or when used over secured connections.

2. Token Authentication:

Uses generated tokens, like JSON Web Tokens (JWT), exchanged between client and server, offering enhanced security without sending login credentials with each request.

When to use:

Ideal for more secure and scalable systems, especially when avoiding sending login credentials with each request is a priority.

3. OAuth Authentication:

Enables third-party limited access to user resources without revealing credentials by issuing access tokens after user authentication.

When to use:

Ideal for scenarios requiring controlled access to user resources by third-party applications or services.

4. API Key Authentication:

Assigns unique keys to users or applications, sent in headers or parameters; while simple, it might lack the security features of token-based or OAuth methods.

When to use:

Convenient for straightforward access control in less sensitive environments or for granting access to certain functionalities without the need for user-specific permissions.

Over to you:

Which REST API authentication method do you find most effective in ensuring both security and usability for your applications?

How to Design a System Like YouTube?



Here's a 9-step process:

- 1 - The user creates a video upload request and provides the video files along with the details about the video.
- 2 - The raw video files are uploaded to an Object Storage (such as S3).
- 3 - Also, the metadata is saved in a database as well as a cache for faster retrieval when needed.
- 4 - The raw video files are sent for transcoding to a special transcoding server. Transcoding is the process of encoding the videos into compatible bitrates and formats for streaming.
- 5 - The transcoded video is uploaded to another object storage.
- 6 - The notification for transcoding completion is sent to a special service via a message queue.

7 - The Transcoding Status Handler updates the metadata DB and cache with the latest details of the video.

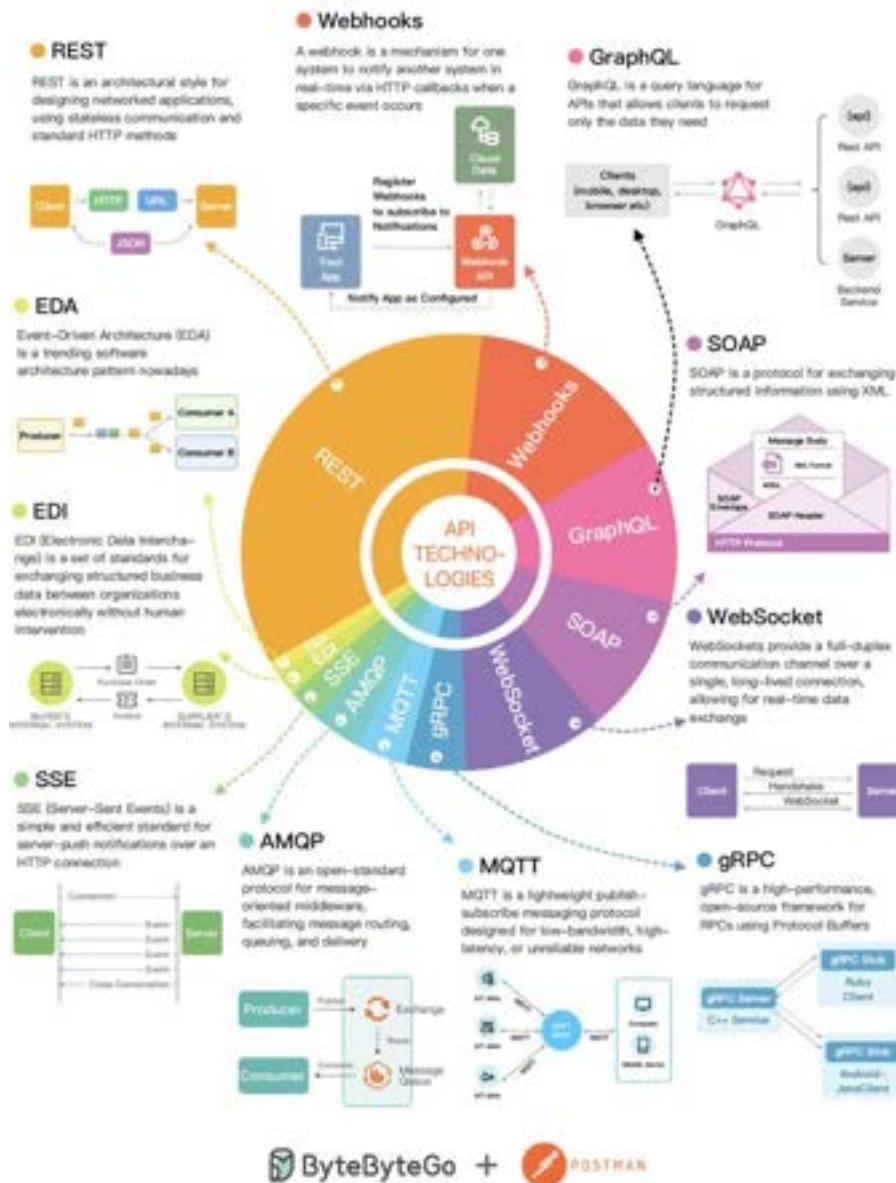
8 - The user raises a video streaming request that goes to a Content Delivery Network (CDN).

9 - The CDN fetches the video from the object storage for streaming. It also caches the video locally for subsequent streaming requests.

Over to you: What else would you add to make the YouTube-like system?

The Evolving Landscape of API Protocols

API Protocols



This is a brief summary of the blog post I wrote for Postman.

In this blog post, I cover the six most popular API protocols: REST, Webhooks, GraphQL, SOAP, WebSocket, and gRPC. The discussion includes the benefits and challenges associated with each protocol.

