2023 EDITION

FREE

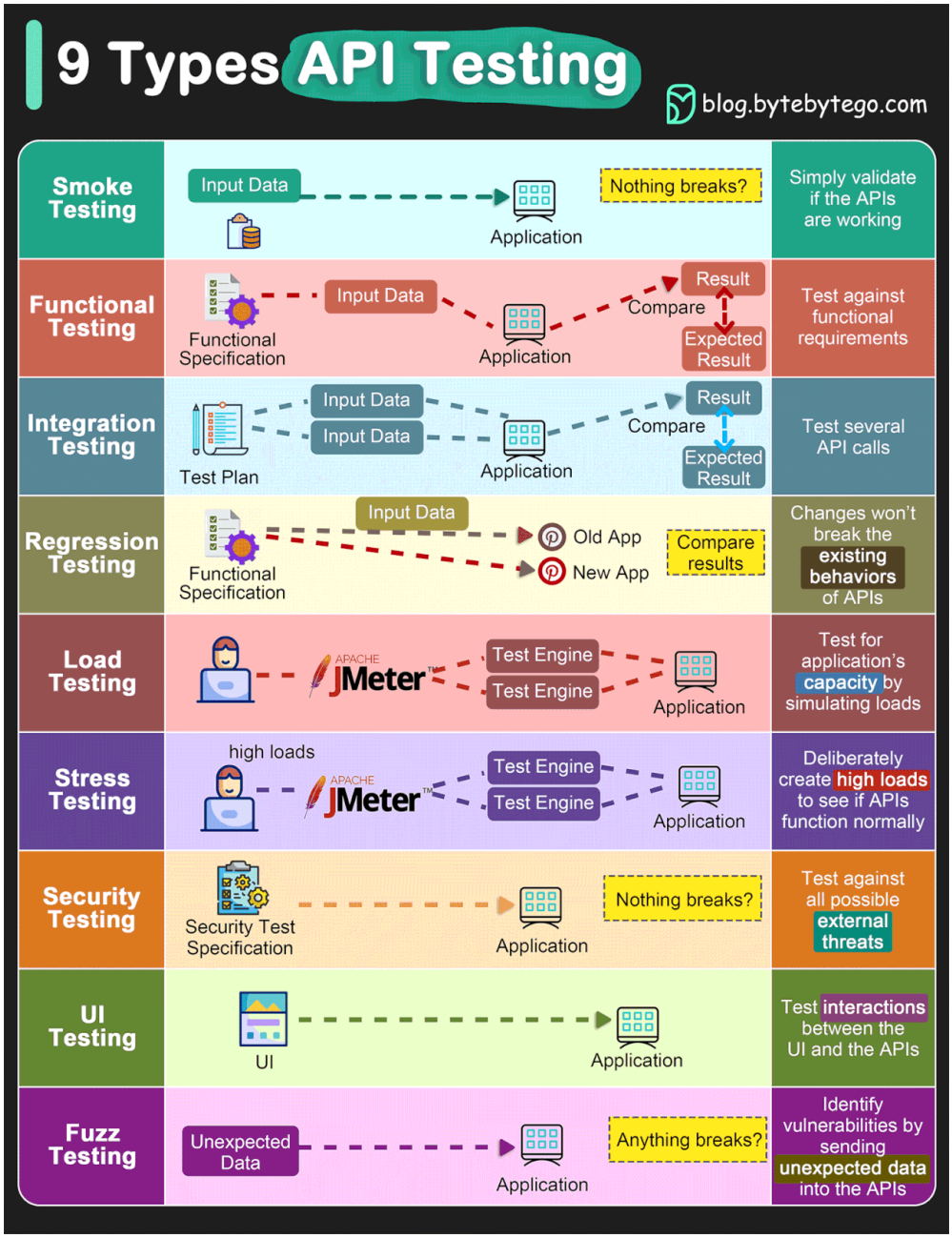
SYSTEM DESIGN

THE BIG ARCHIVE

ByteByteGo

By ByteByteGo.com

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.

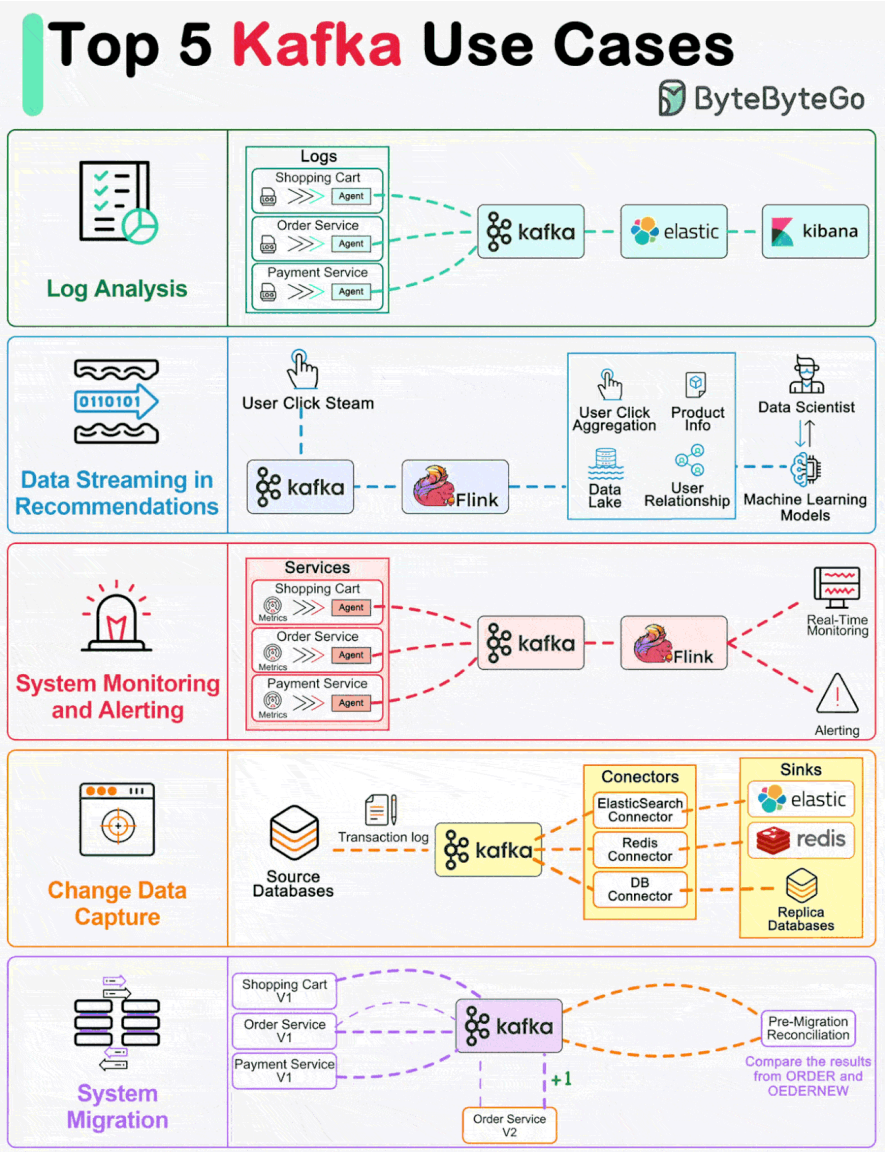
- Security Testing
  This tests the APIs against all possible external threats.

- 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.

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.

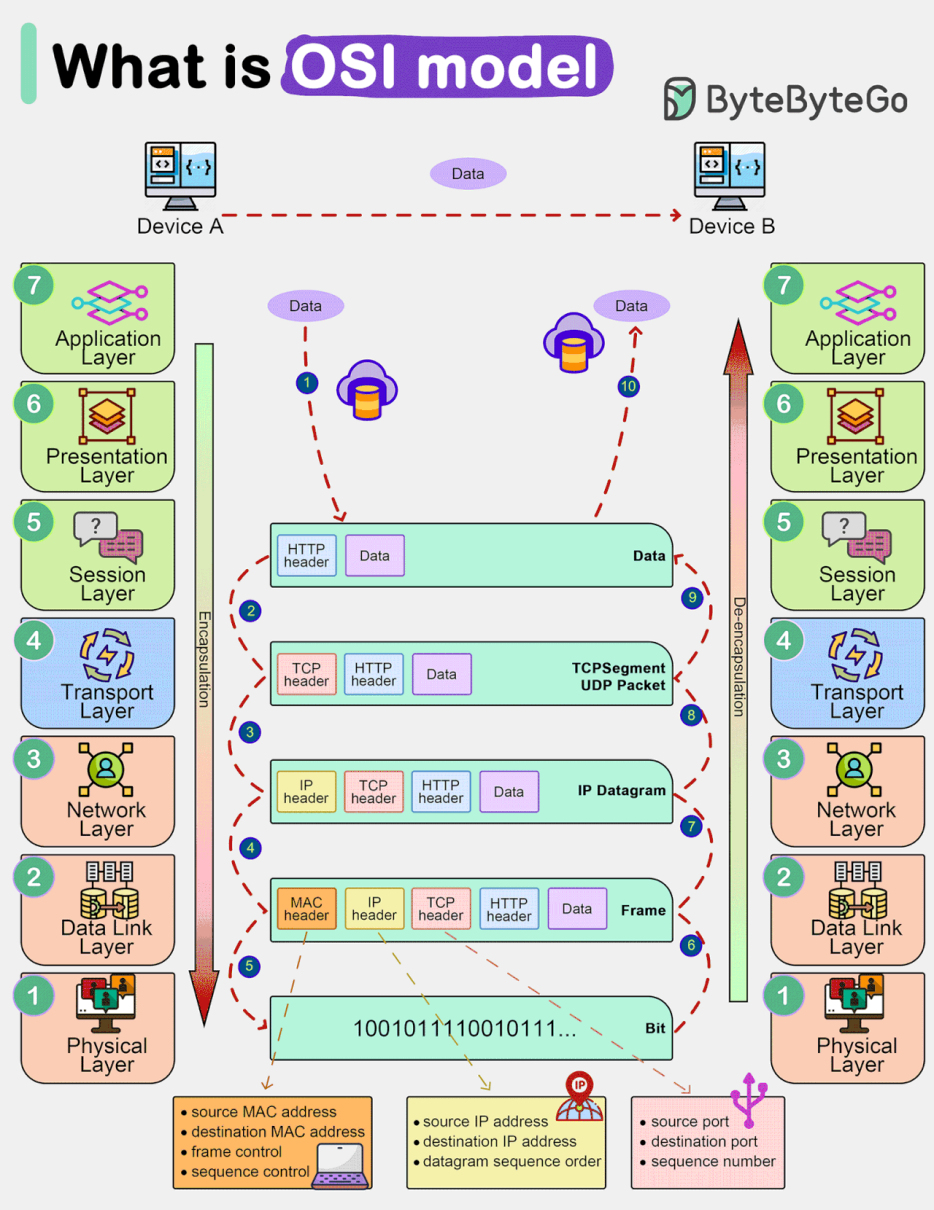


Let's review the popular Kafka use cases.

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

Over to you: Do you have any other Kafka use cases to share?

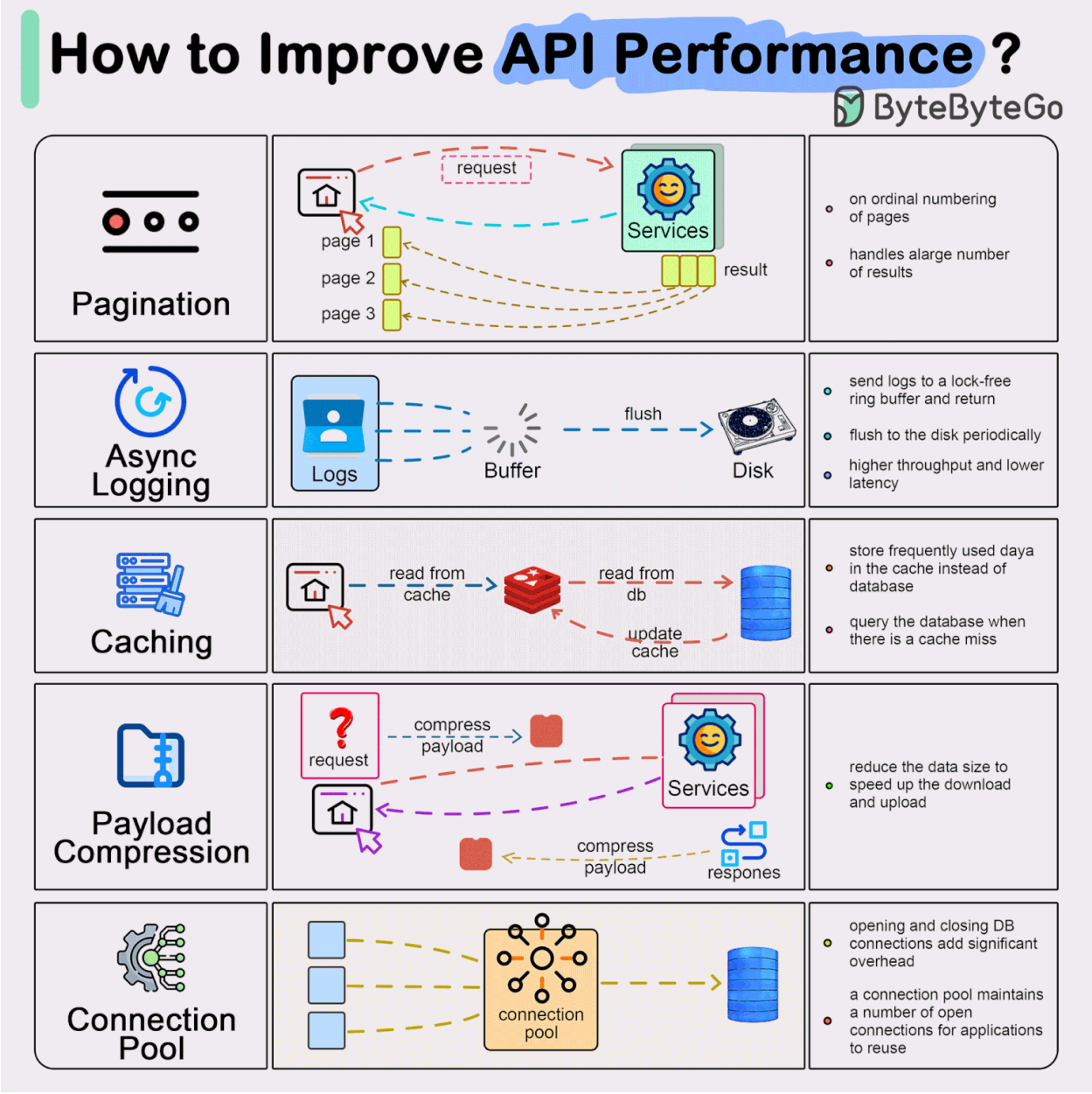
How is data sent over the internet? What does that have to do with the OSI model? How does TCP/IP fit into this?



7 Layers in the OSI model are:

1. Physical Layer
2. Data Link Layer
3. Network Layer
4. Transport Layer
5. Session Layer
6. Presentation Layer
7. Application Layer

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.

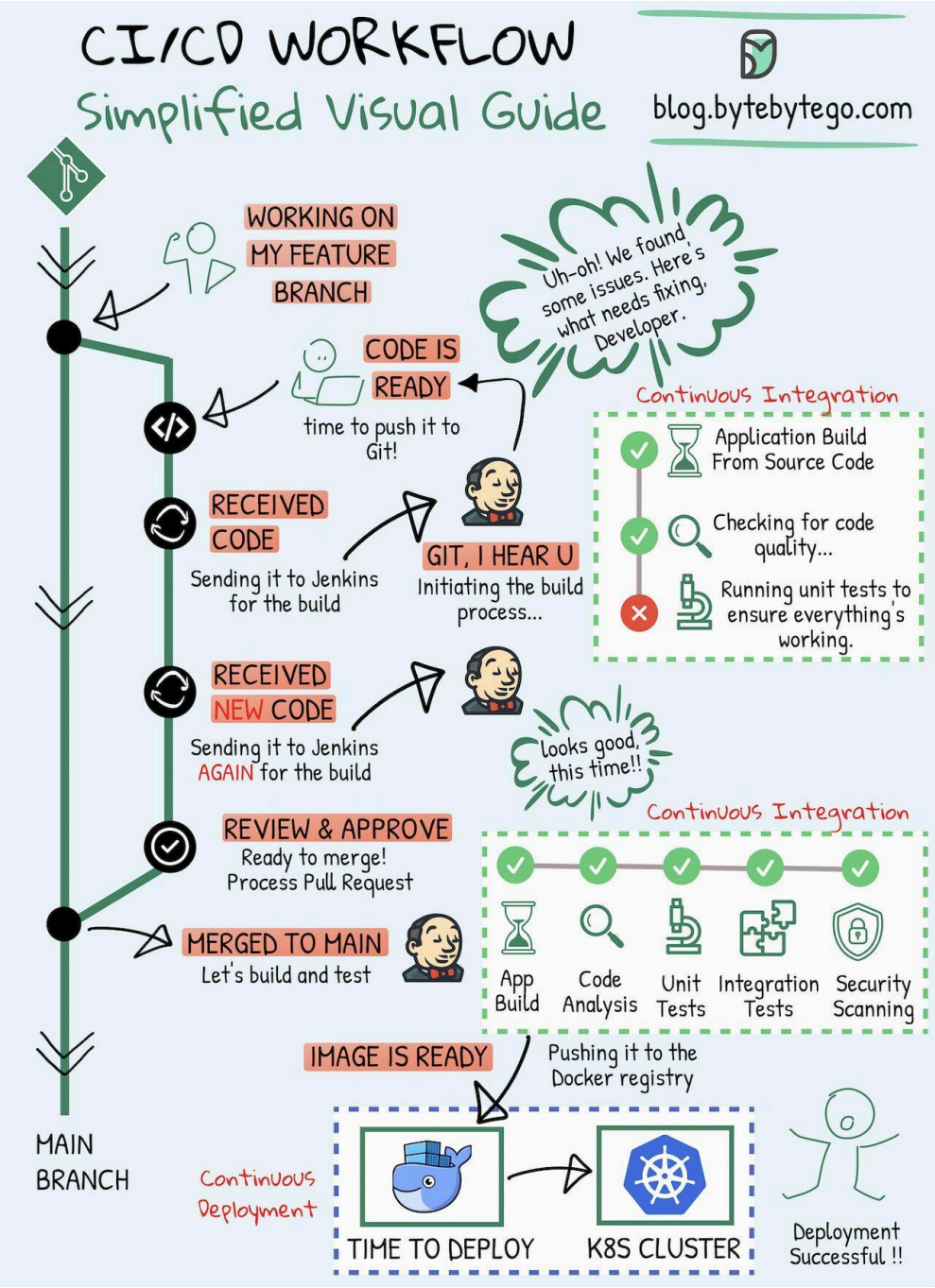
Payload Compression:
To reduce data transmission time, requests and responses can be compressed (e.g., using gzip), making the upload and download processes quicker.

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?

CI/CD Simplified Visual Guide

Whether you're a developer, a DevOps specialist, a tester, or involved in any modern IT role, CI/CD pipelines have become an integral part of the software development process.



Continuous Integration (CI) is a practice where code changes are frequently combined into a shared repository. This process includes automatic checks to ensure the new code works well

with the existing code.

Continuous Deployment (CD) takes care of automatically putting these code changes into real-world use. It makes sure that the process of moving new code to production is smooth and reliable.

This visual guide is designed to help you grasp and enhance your methods for creating and delivering software more effectively.

Over to you: Which tools or strategies do you find most effective in implementing CI/CD in your projects?

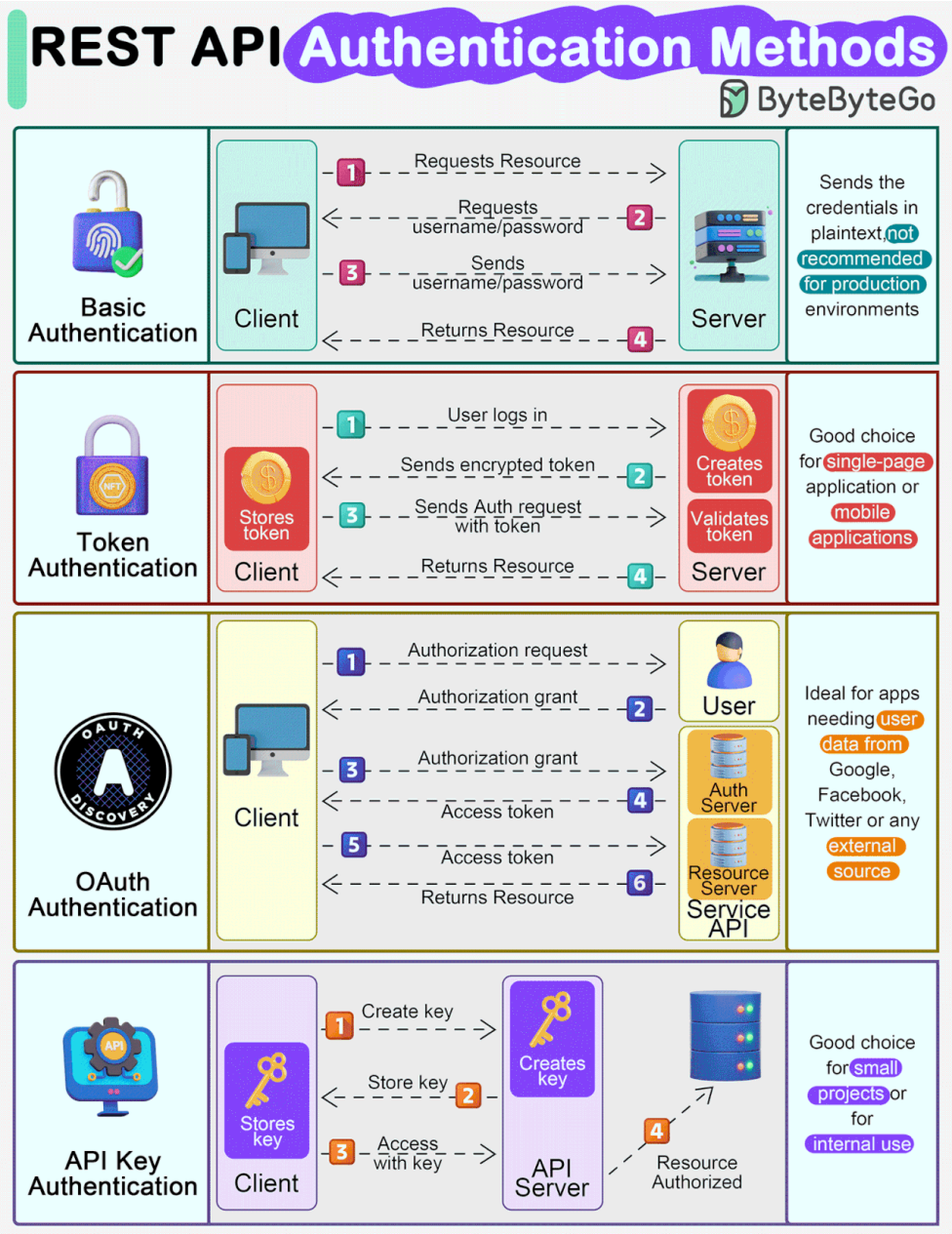
There are over 1,000 engineering blogs. Here are my top 9 favorites:

- 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?

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.

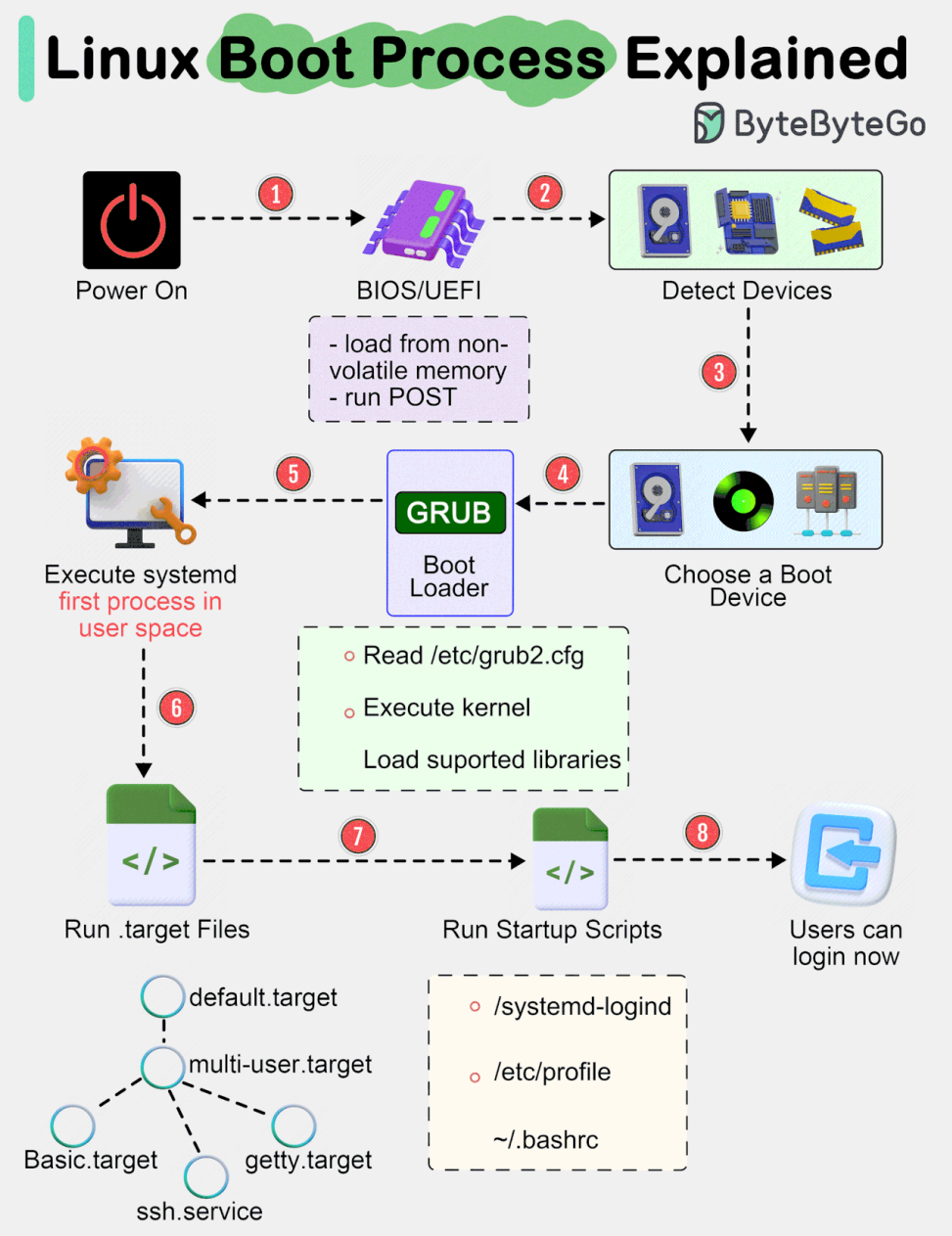
When to use:
Convenient for straightforward access control in less sensitive environments or for granting access to certain functionalities without the need for user-specific permissions.

Over to you:
Which REST API authentication method do you find most effective in ensuring both security and usability for your applications?

Linux Boot Process Illustrated

We've made a video (YouTube Link at the end).

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.

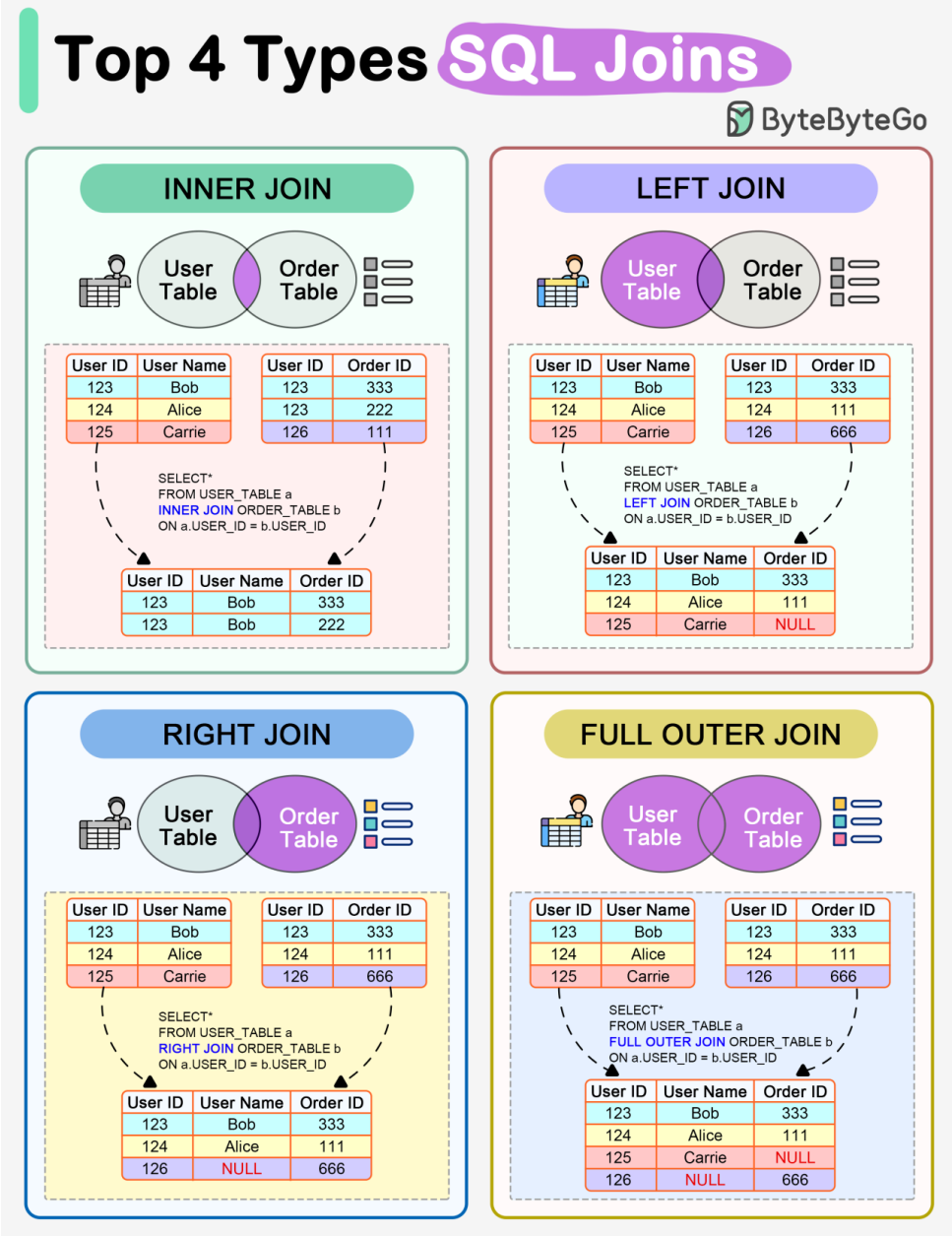
Step 7 - The system runs a set of startup scripts and configures the environment.

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

Watch and subscribe here: https://lnkd.in/ezkZb5Wq

How do SQL Joins Work?

The diagram below shows how 4 types of SQL joins work in detail.



- INNER JOIN
  Returns matching rows in both tables.

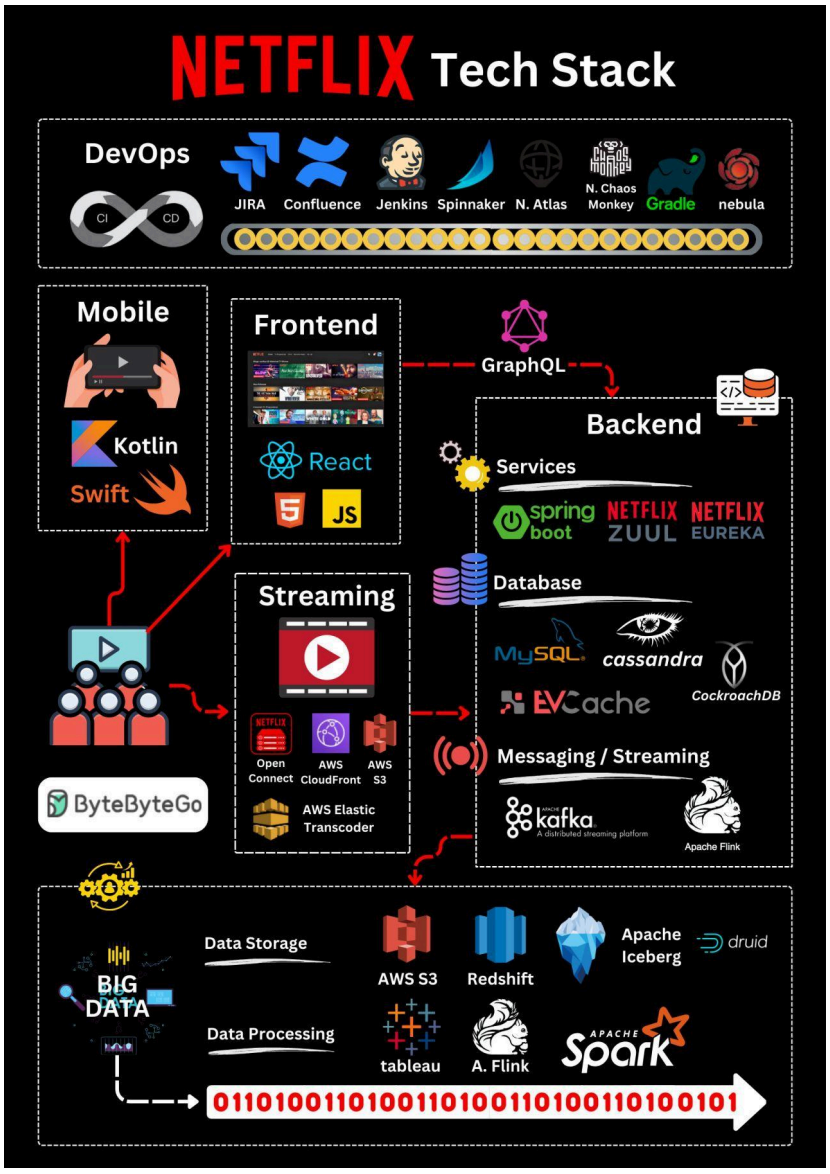
- LEFT JOIN
  Returns all records from the left table, and the matching records from the right table.

- RIGHT JOIN
  Returns all records from the right table, and the matching records from the left table.

- FULL OUTER JOIN
  Returns all records where there is a match in either the left or right table.

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.

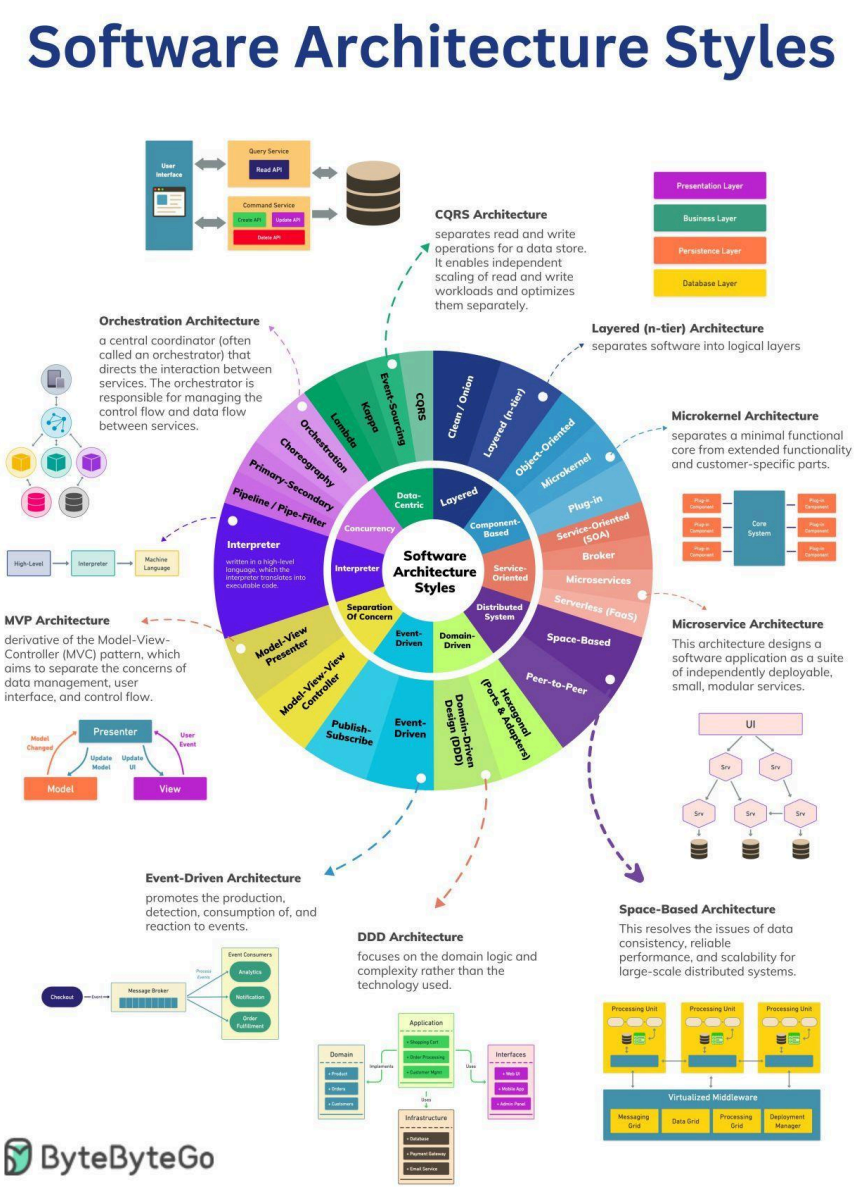
Messaging/streaming: Netflix employs Apache Kafka and Fink for messaging and streaming purposes.

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 Architectural 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.

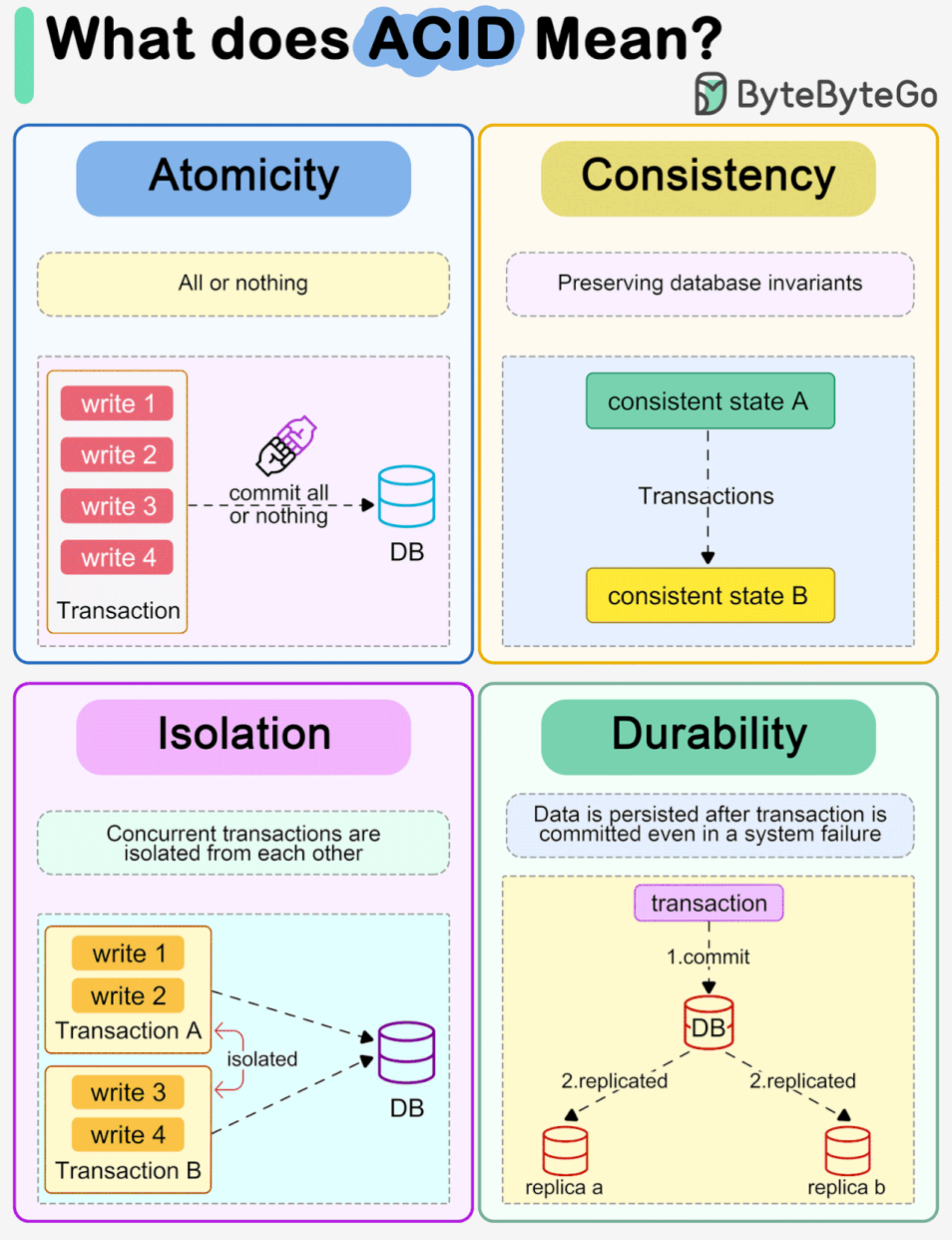
However, with the vast array of architectural styles and patterns available, it can take time to discern which approach best suits a particular project or system. Aims to shed light on these

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.

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.

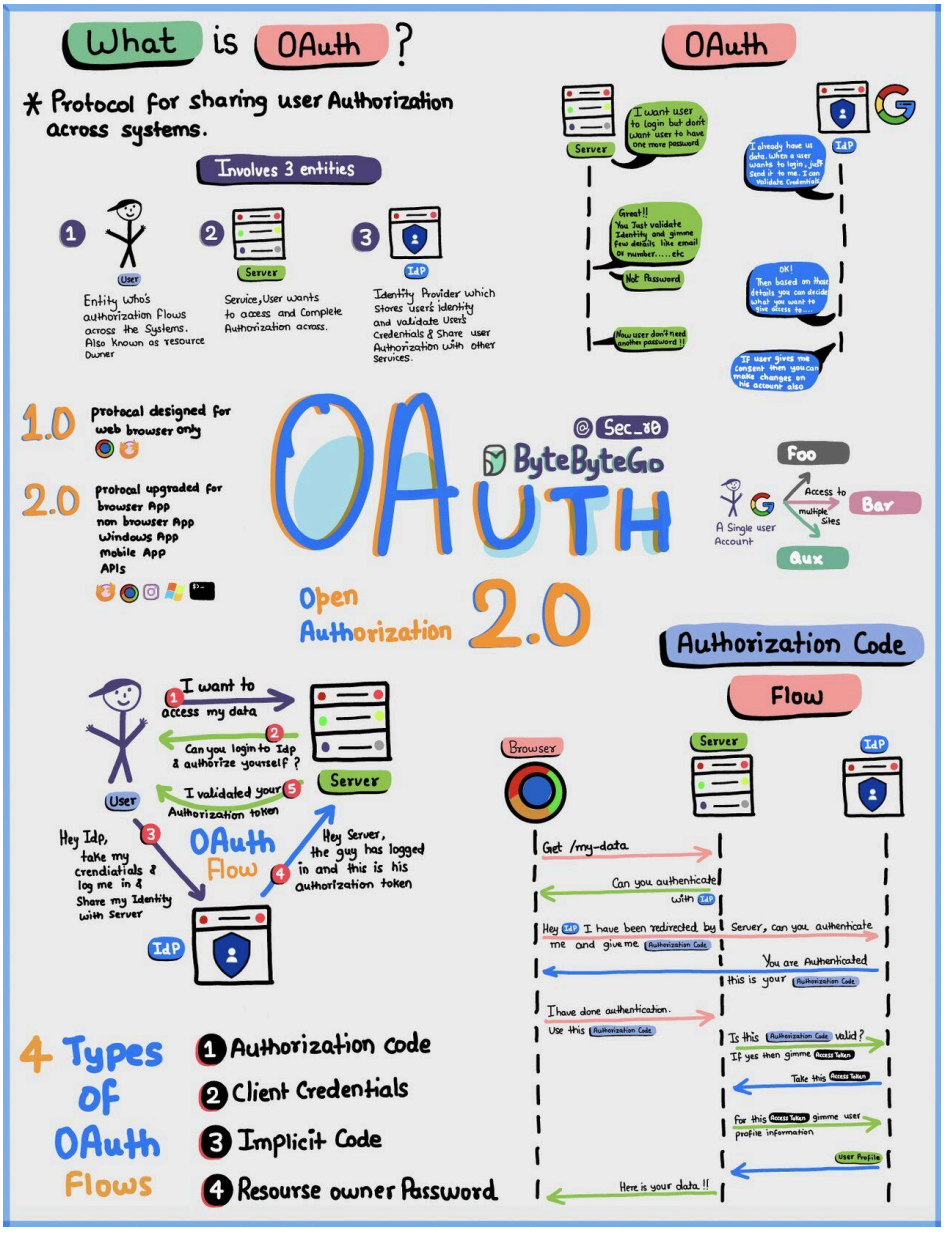
So atomicity means “all or nothing”.

- 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 a 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.

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.

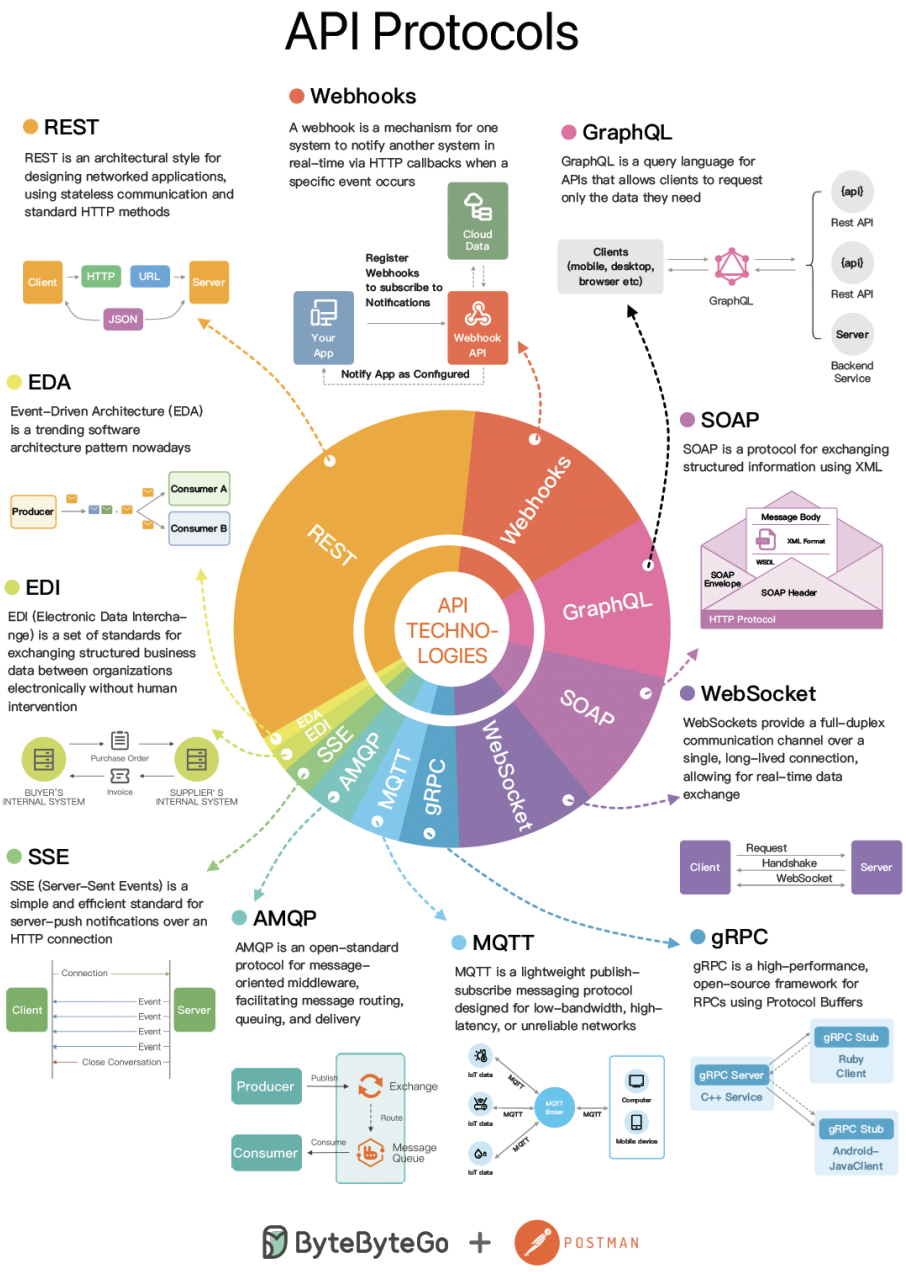
Remember, OAuth 2.0 is all about keeping you and your data safe while making your online experiences seamless and hassle-free across different applications and services.

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.

The Evolving Landscape of API Protocols in 2023

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.

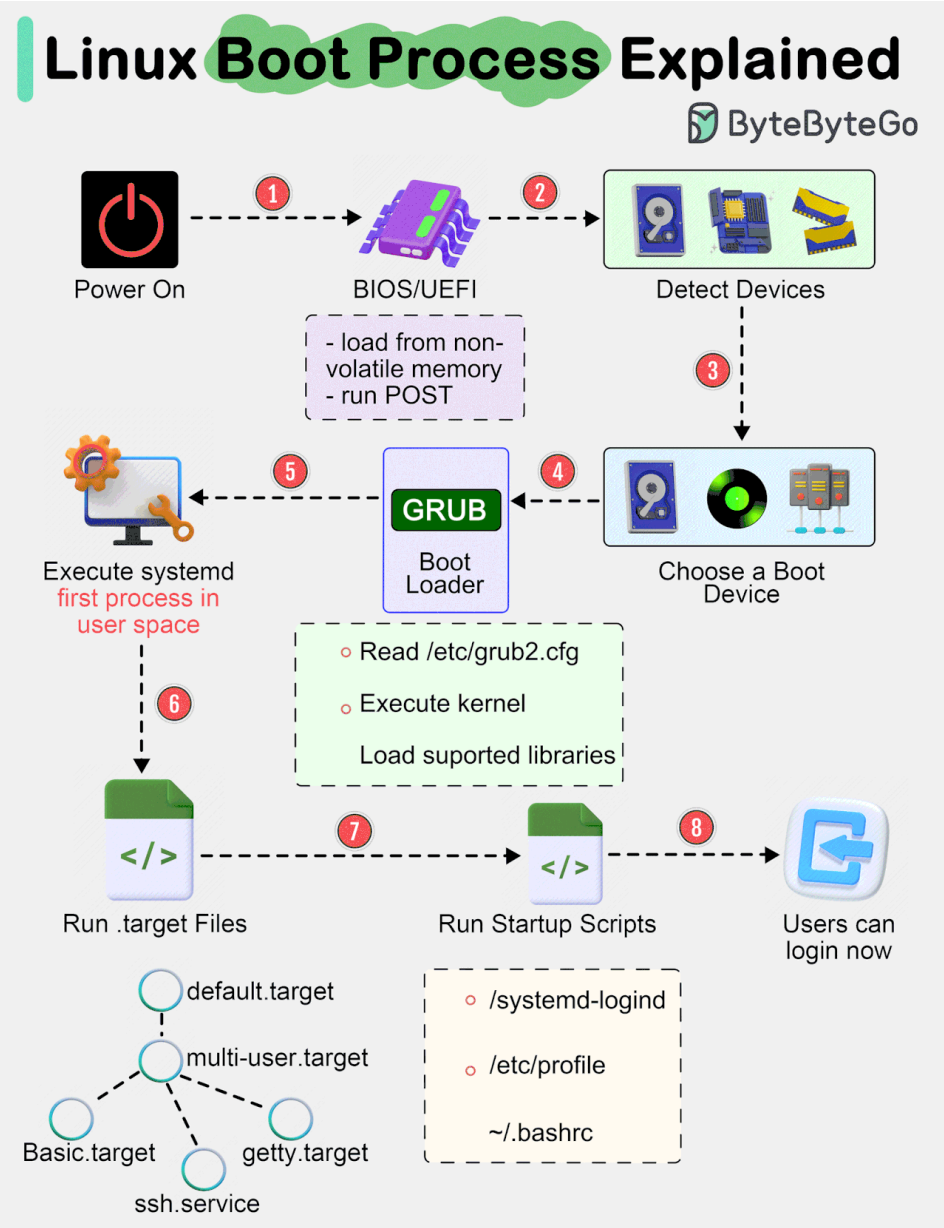


You can read the full blog post here: https://blog.postman.com/api-protocols-in-2023/

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.

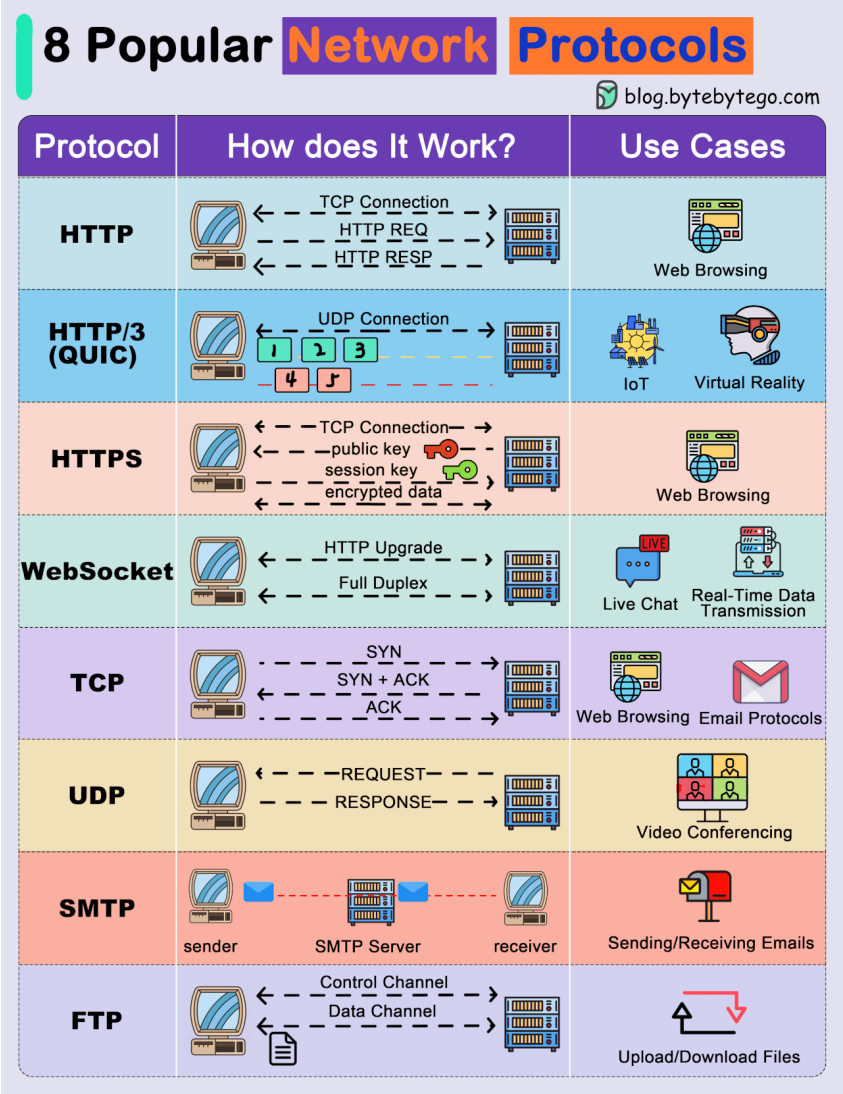
Step 6 - systemd activates the default. target unit by default when the system boots. Other analysis units are executed as well.

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.

Explaining 8 Popular Network Protocols in 1 Diagram.

You can find the link to watch a detailed video explanation at the end of the post.



Network protocols are standard methods of transferring data between two computers in a network.

1. HTTP (HyperText Transfer Protocol)
HTTP is a protocol for fetching resources such as HTML documents. It is the foundation of any data exchange on the Web and it is a client-server protocol.

2. HTTP/3
HTTP/3 is the next major revision of the HTTP. It runs on QUIC, a new transport protocol designed for mobile-heavy internet usage. It relies on UDP instead of TCP, which enables faster

web page responsiveness. VR applications demand more bandwidth to render intricate details of a virtual scene and will likely benefit from migrating to HTTP/3 powered by QUIC.

3. HTTPS (HyperText Transfer Protocol Secure)
HTTPS extends HTTP and uses encryption for secure communications.

4. WebSocket
WebSocket is a protocol that provides full-duplex communications over TCP. Clients establish WebSockets to receive real-time updates from the back-end services. Unlike REST, which always "pulls" data, WebSocket enables data to be "pushed". Applications, like online gaming, stock trading, and messaging apps leverage WebSocket for real-time communication.

5. TCP (Transmission Control Protocol)
TCP is designed to send packets across the internet and ensure the successful delivery of data and messages over networks. Many application-layer protocols are built on top of TCP.

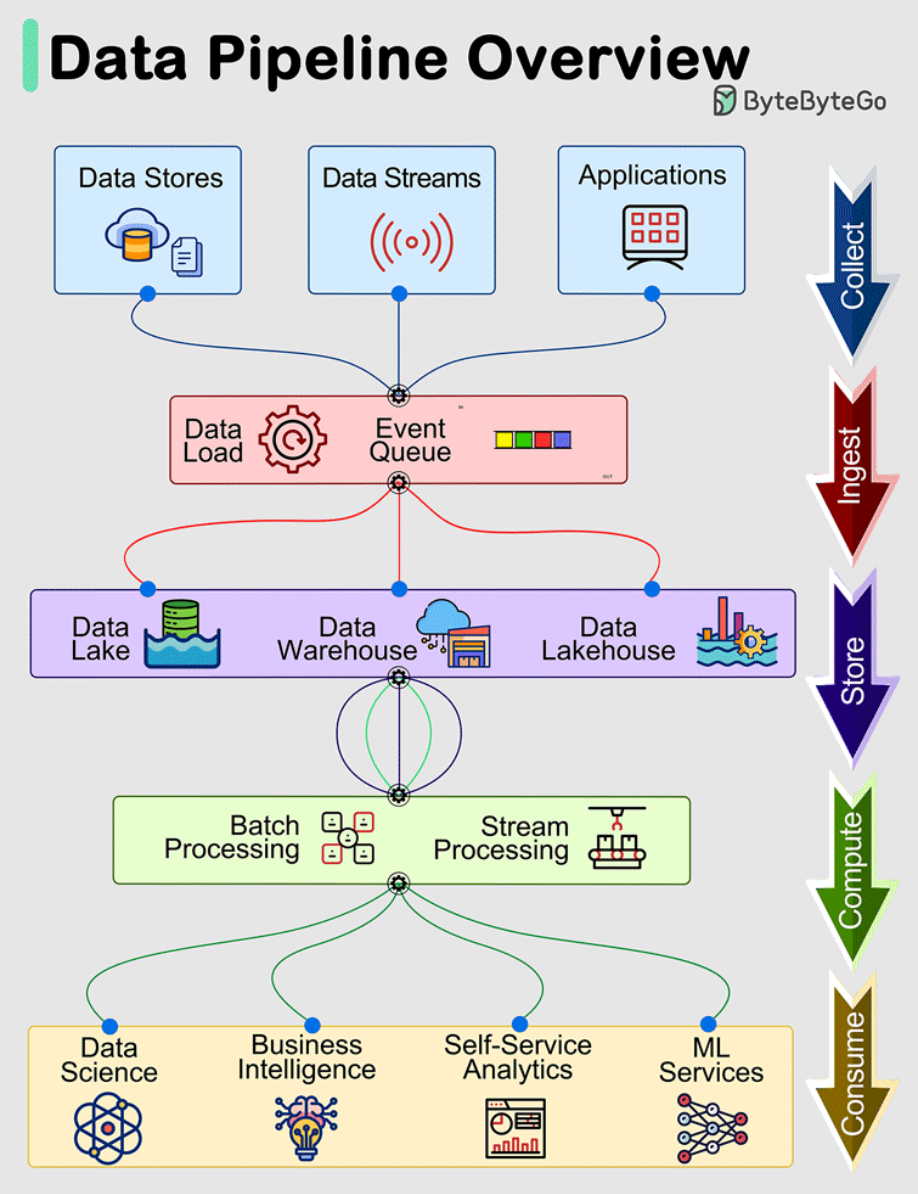
6. UDP (User Datagram Protocol)
UDP sends packets directly to a target computer, without establishing a connection first. UDP is commonly used in time-sensitive communications where occasionally dropping packets is better than waiting. Voice and video traffic are often sent using this protocol.

7. SMTP (Simple Mail Transfer Protocol)
SMTP is a standard protocol to transfer electronic mail from one user to another.

8. FTP (File Transfer Protocol)
FTP is used to transfer computer files between client and server. It has separate connections for the control channel and data channel.

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.

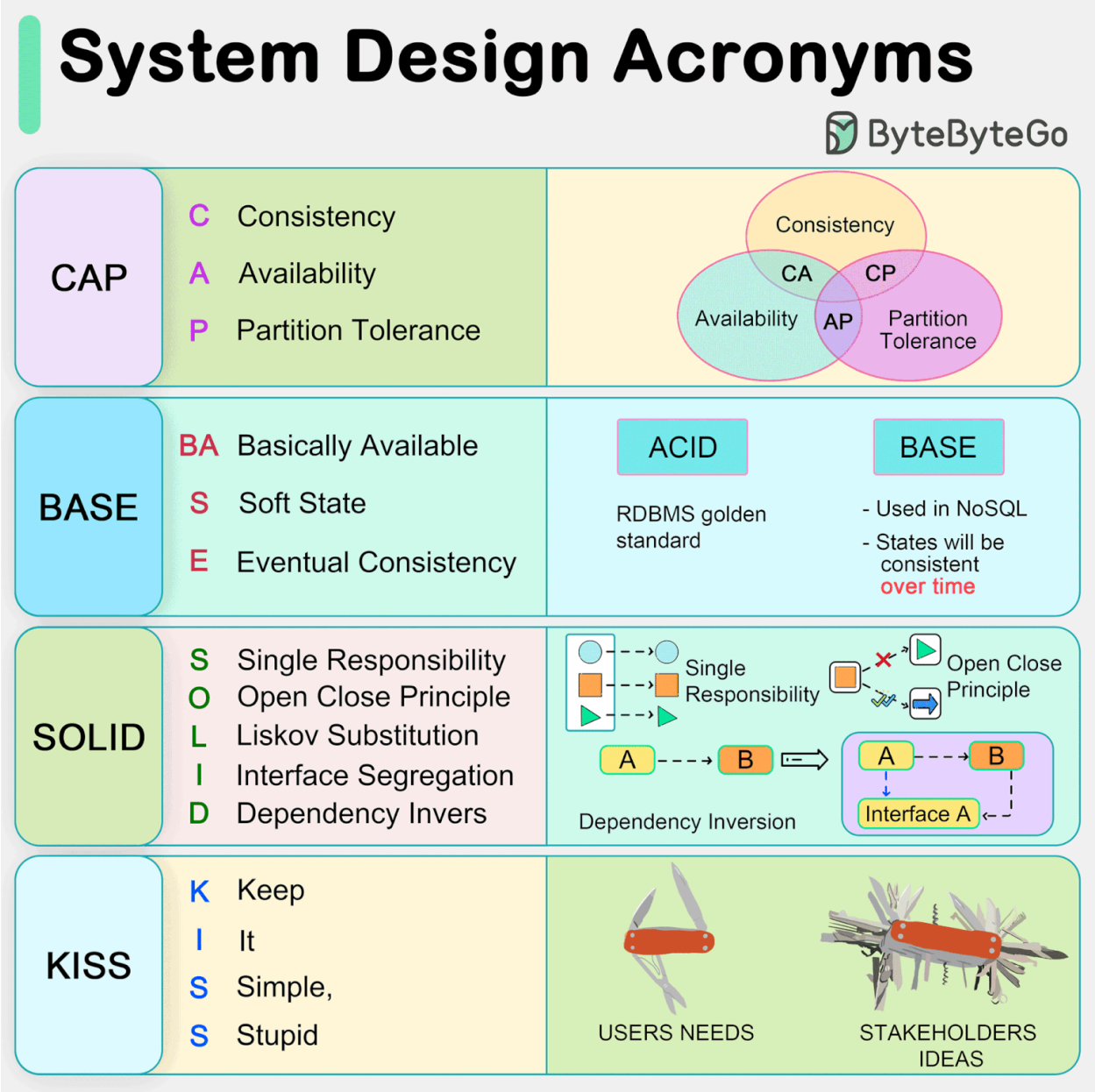
5. Consume:
Processed data is made available for consumption through analytics and visualization tools, operational data stores, decision engines, user-facing applications, dashboards, data science, machine learning services, business intelligence, and self-service analytics.

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?

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.

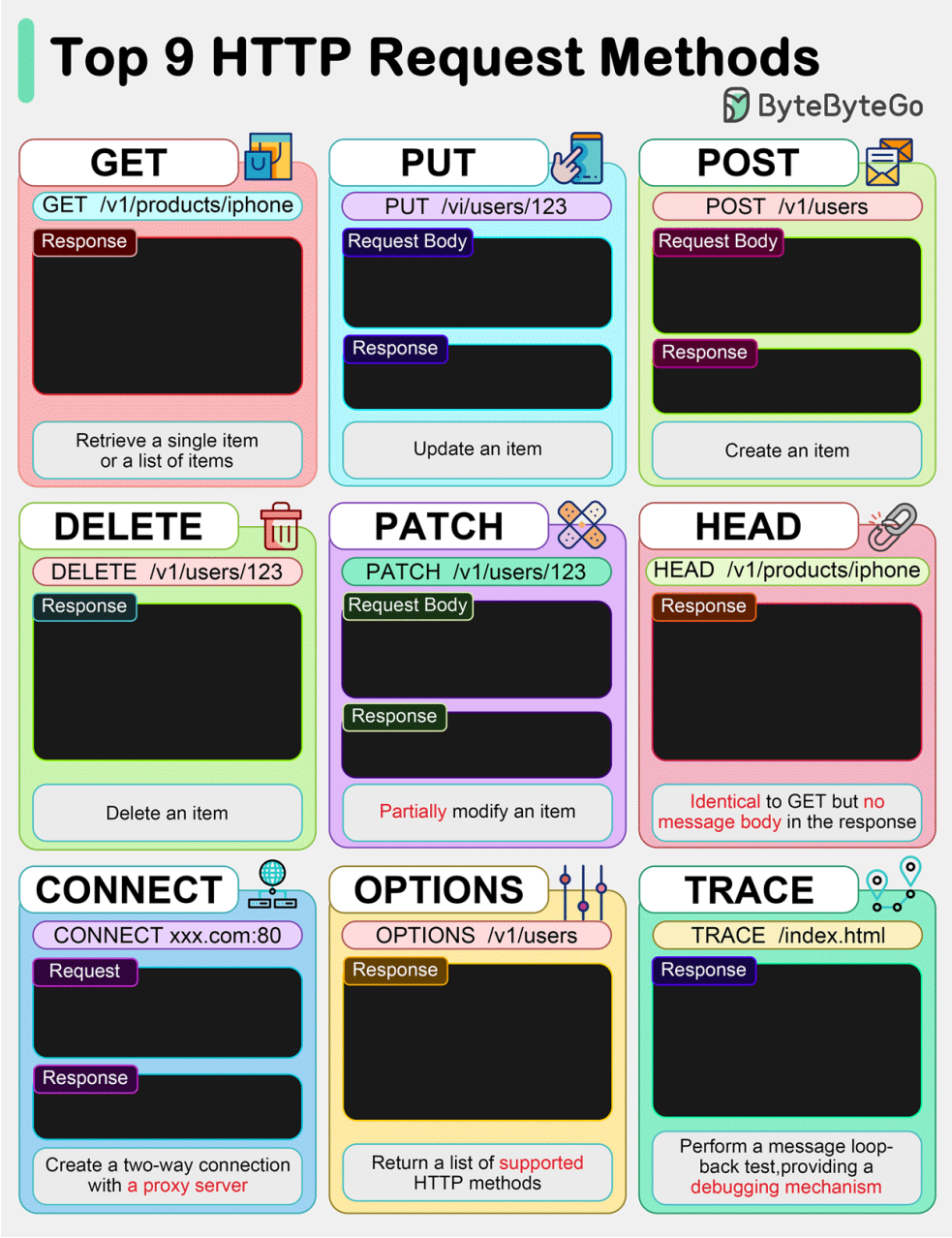
  4. ISP (Interface Segregation Principle)
  Expose multiple interfaces with specific responsibilities.

  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?

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.

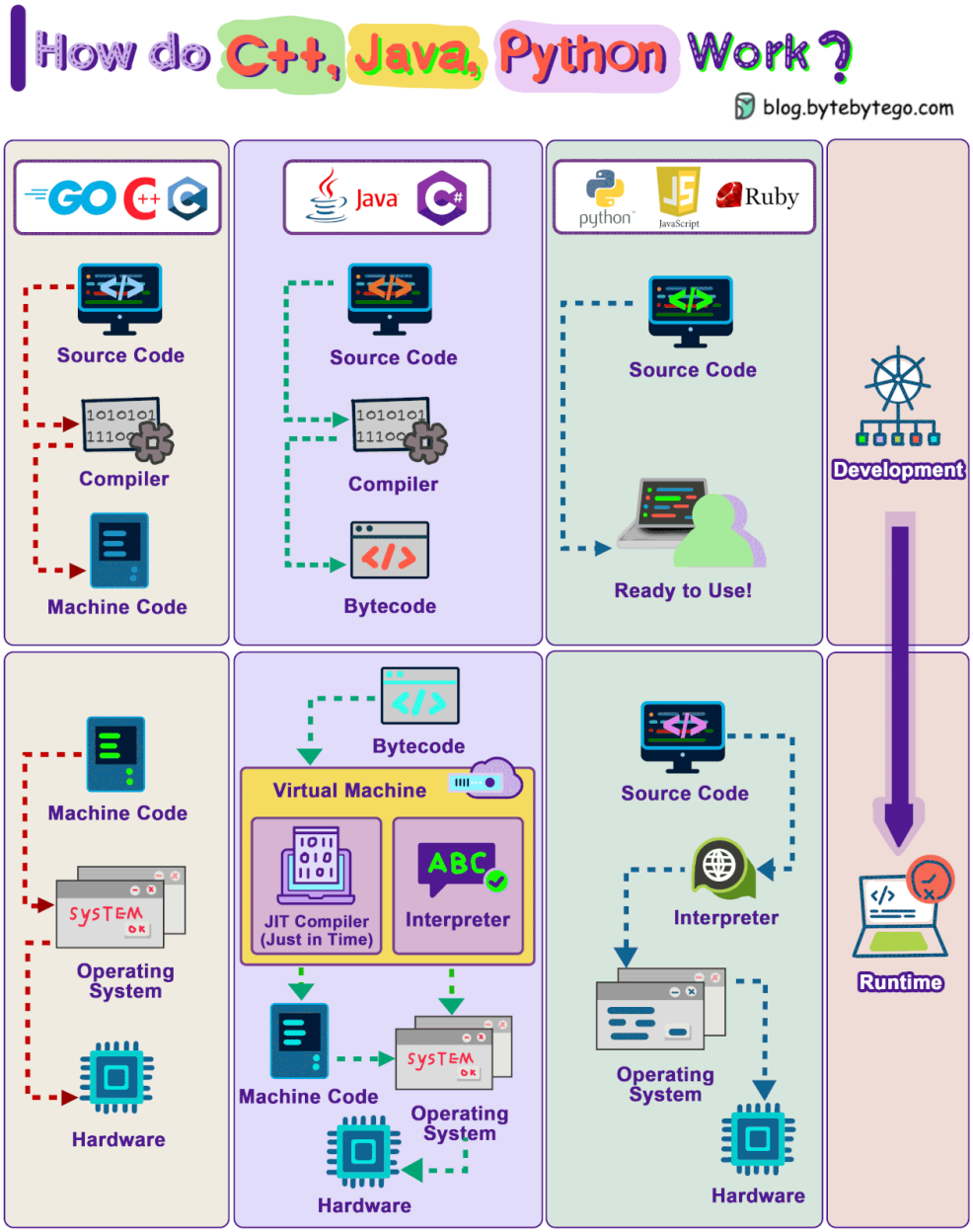
8. HTTP OPTIONS
   This describes the communication options for the target resource.

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?

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 JIT (Just-In-Time) compiler compiles the source code into

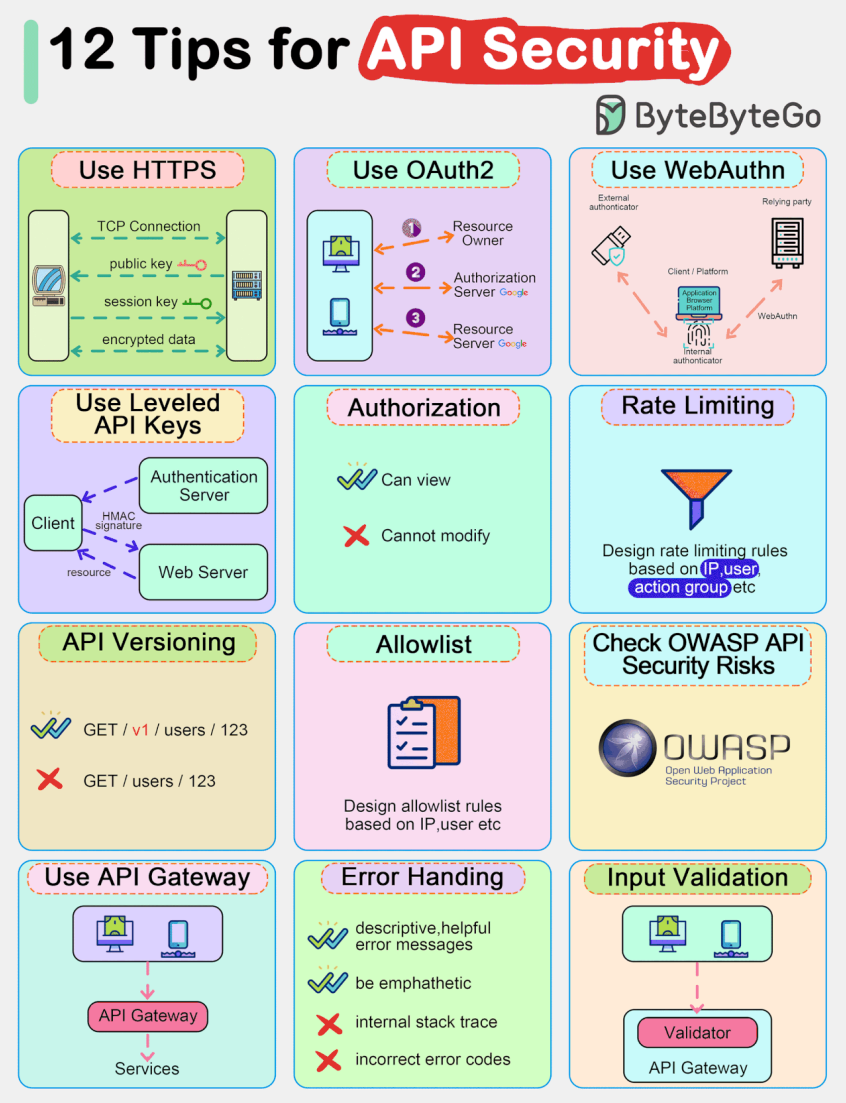
machine code to speed up the execution. Examples: Java, C#

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?

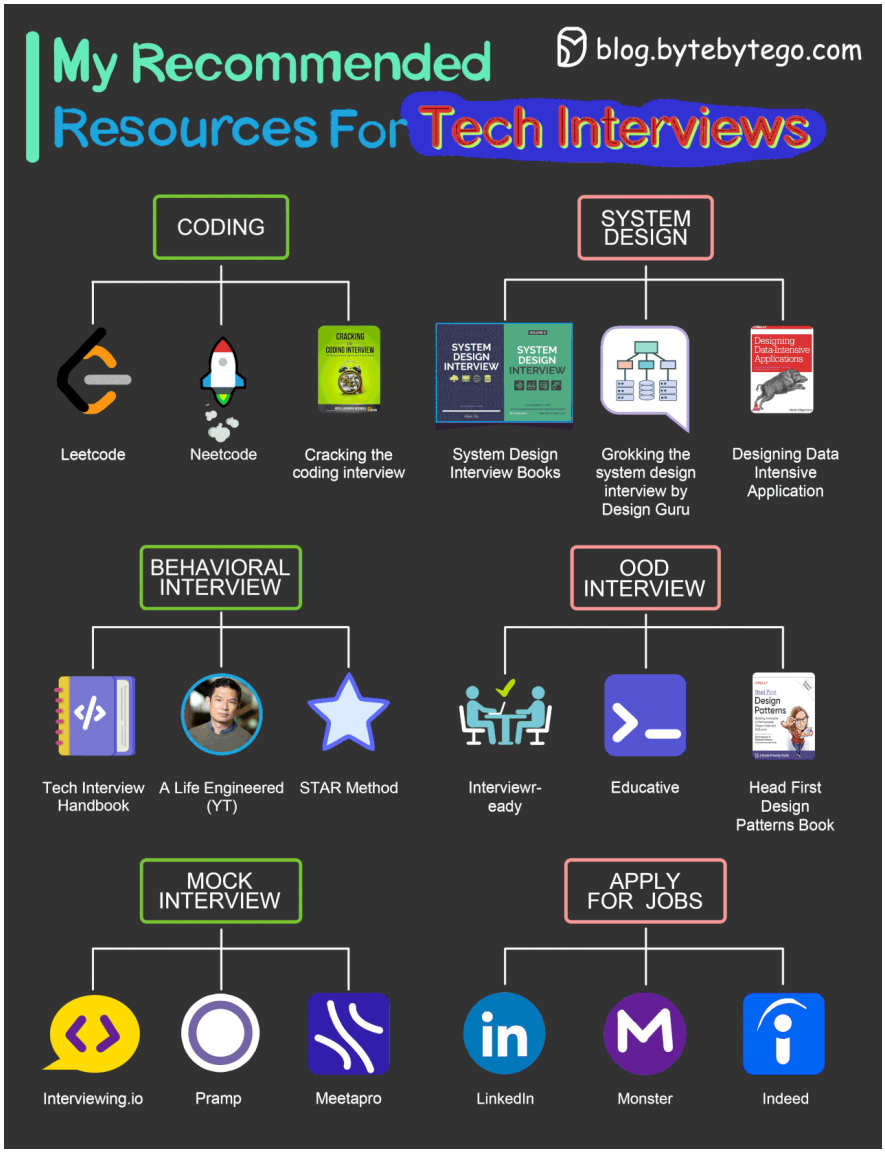
Top 12 Tips for API Security



- Use HTTPS
- Use OAuth2
- Use WebAuthn
- Use Leveled API Keys
- Authorization
- Rate Limiting
- API Versioning
- Whitelisting
- Check OWASP API Security Risks
- Use API Gateway
- Error Handling
- Input Validation

Our recommended materials to crack your next tech interview

You can find the link to watch a detailed video explanation at the end of the post.



Coding

- Leetcode
- Cracking the coding interview book
- Neetcode

System Design Interview

- System Design Interview book 1, 2 by Alex Xu
- 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

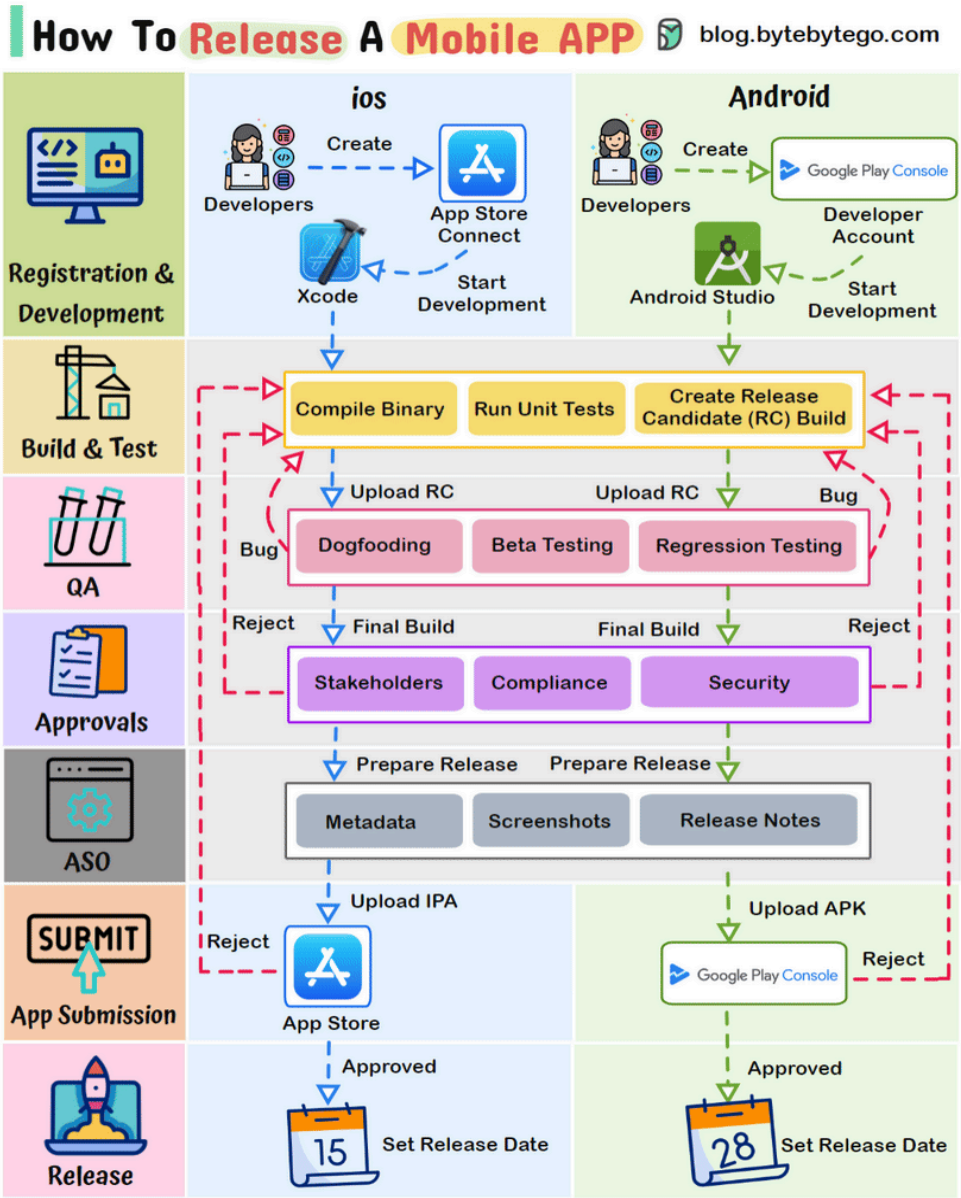
Apply for Jobs

- Linkedin
- Monster
- Indeed

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

How To Release A Mobile App

The mobile app release process differs from conventional methods. This illustration simplifies the journey to help you understand.



Typical Stages in a Mobile App Release Process:

1. Registration & Development (iOS & Android):
- Enroll in Apple's Developer Program and Google Play Console as iOS and Android developer
- Code using platform-specific tools: Swift/Obj-C for iOS, and Java/Kotlin for Android

2. Build & Test (iOS & Android):
Compile the app's binary, run extensive tests on both platforms to ensure functionality and performance. Create a release candidate build.

3. QA:
- Internally test the app for issue identification (dogfooding)
- Beta test with external users to collect feedback
- Conduct regression testing to maintain feature stability

4. Internal Approvals:
- Obtain approval from stakeholders and key team members.
- Comply with app store guidelines and industry regulations
- Obtain security approvals to safeguard user data and privacy

5. App Store Optimization (ASO):
- Optimize metadata, including titles, descriptions, and keywords, for better search visibility
- Design captivating screenshots and icons to entice users
- Prepare engaging release notes to inform users about new features and updates

6. App Submission To Store:
- Submit the iOS app via App Store Connect following Apple's guidelines
- Submit the Android app via Google Play Console, adhering to Google's policies
- Both platforms may request issues resolution for approval

7. Release:
- Upon approval, set a release date to coordinate the launch on both iOS and Android platforms

Over to you:
What's the most challenging phase you've encountered in the mobile app release process?

A handy cheat sheet for the most popular cloud services (2023 edition)

What's included?

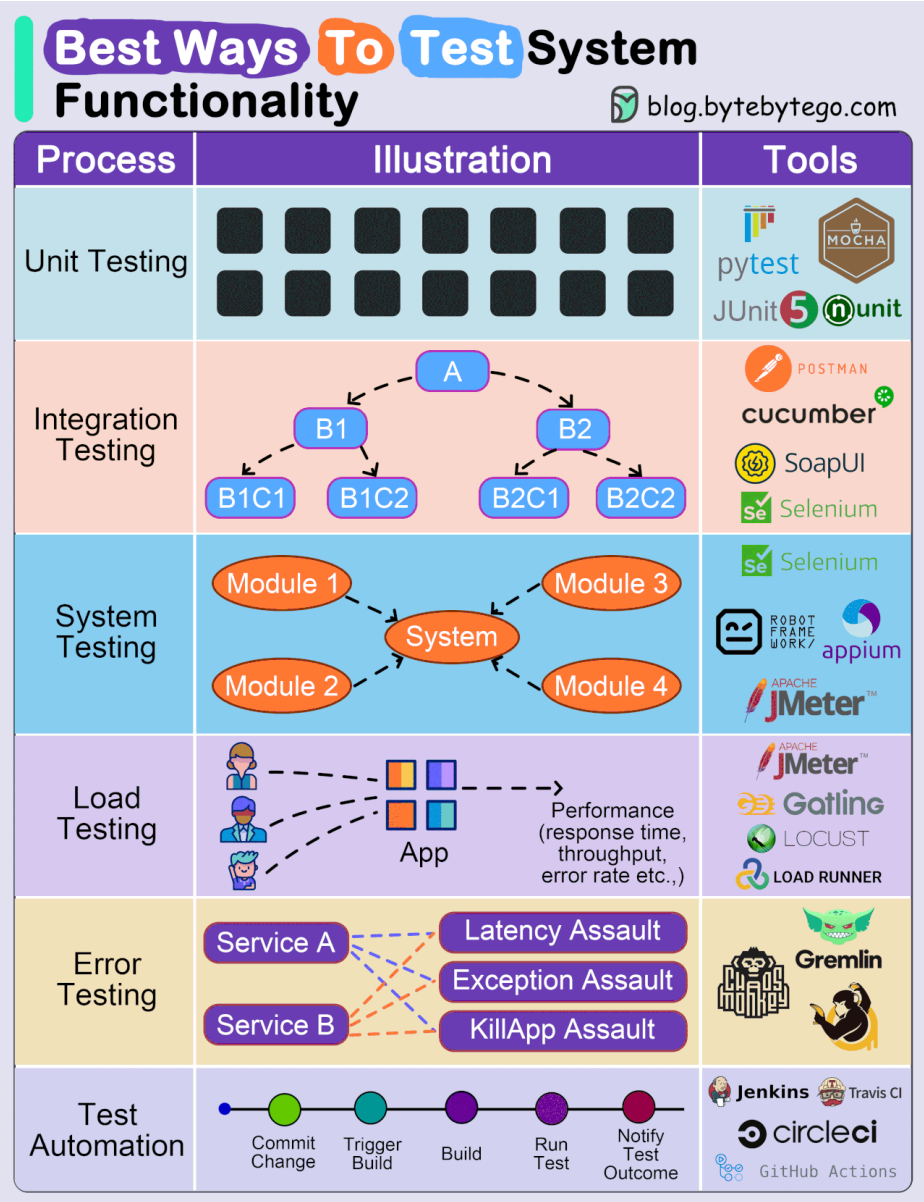
- AWS, Azure, Google Cloud, Oracle Cloud, Alibaba Cloud
- Cloud servers
- Databases

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

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

Best ways to test system functionality

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.

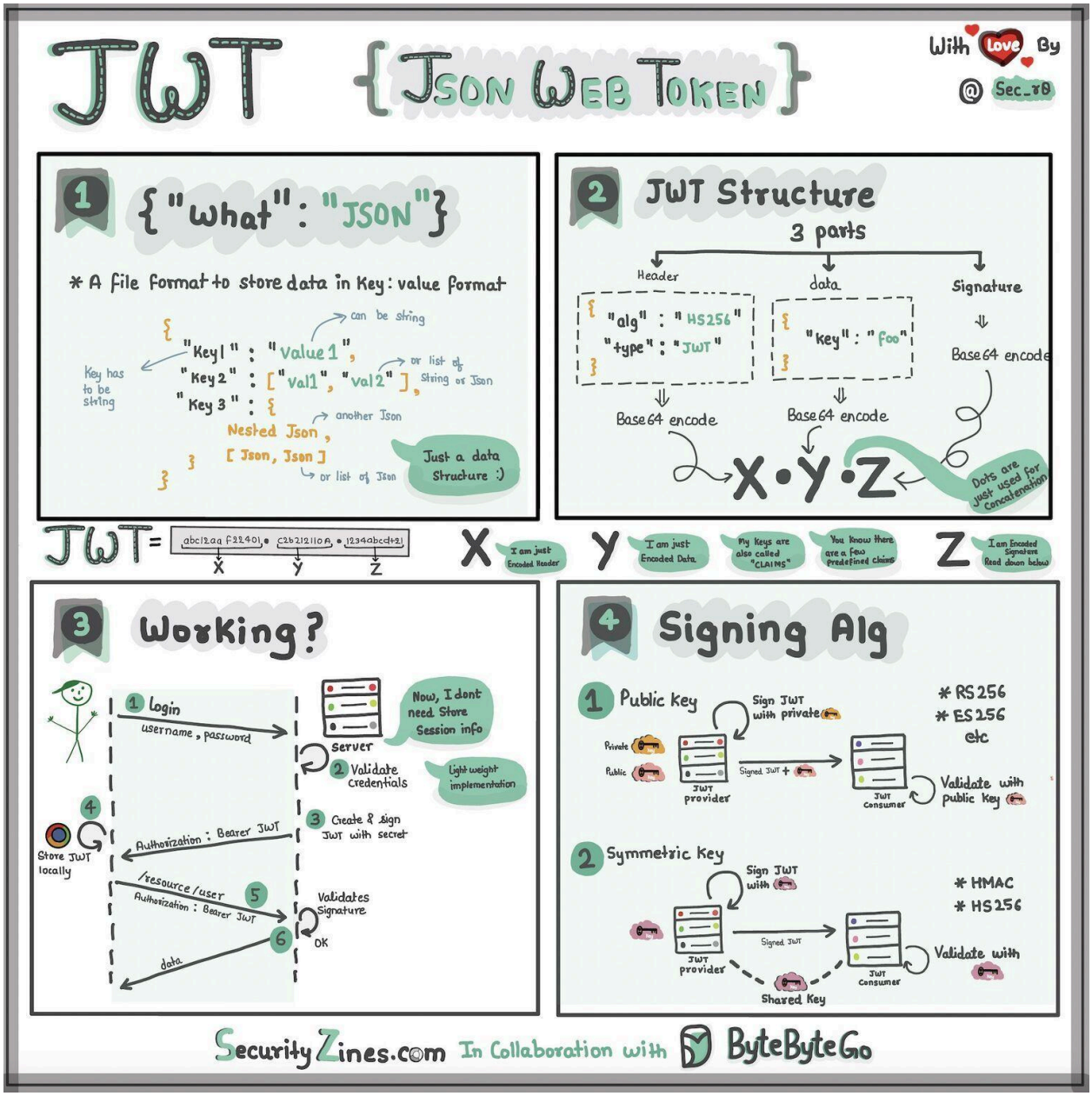
6. Test Automation: Automates test case execution for efficiency, repeatability, and error reduction.

Over to you:

- How do you approach testing system functionality in your software development or engineering projects?

- What’s your company's release process look like?

Explaining JSON Web Token (JWT) to a 10 year old Kid



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.

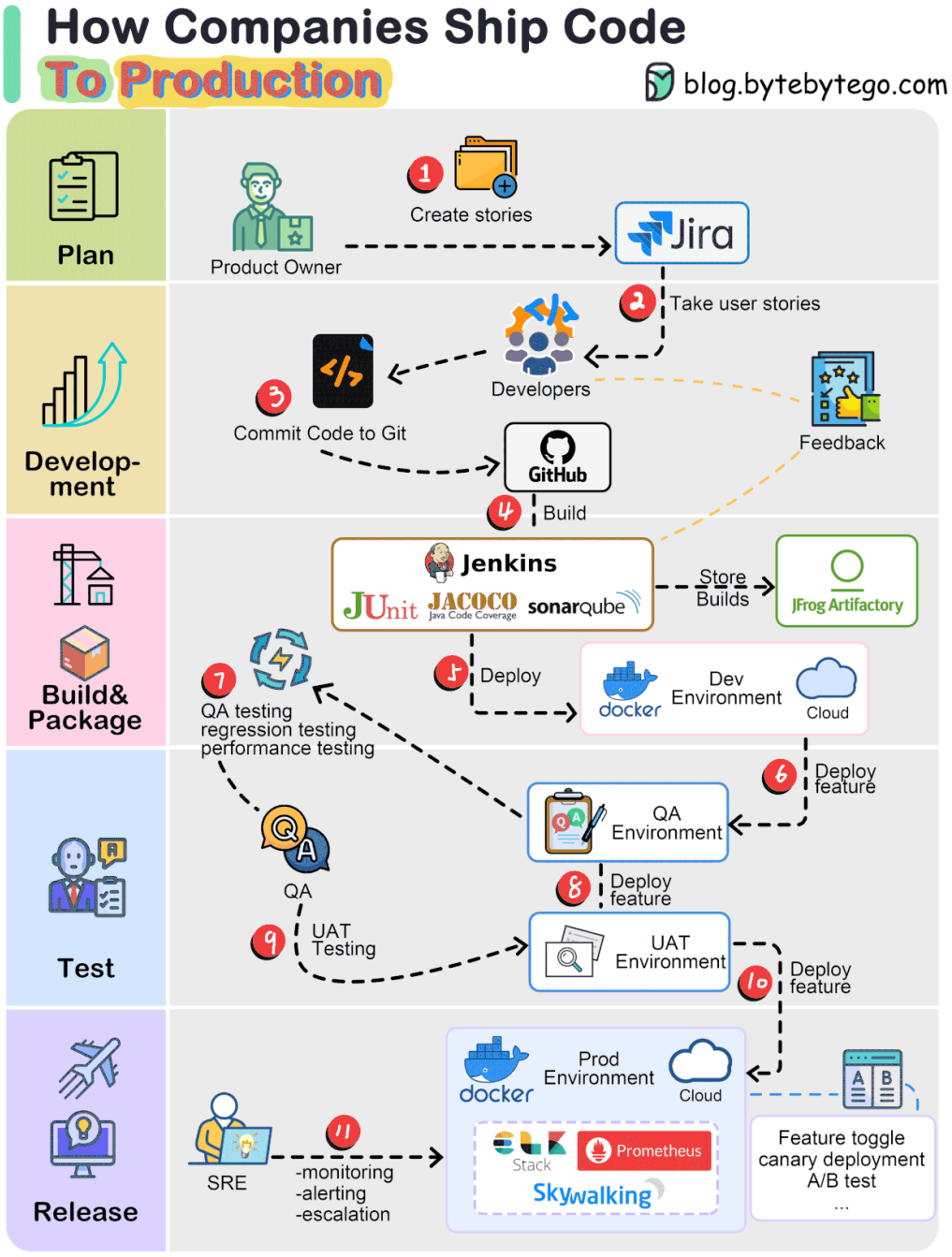
Now, the signature is what makes the JWT secure. It's like a special seal that only the sender knows how to create. The signature is created using a secret code, kind of like a password. This signature ensures that nobody can tamper with the contents of the JWT without the sender knowing about it.

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?

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.

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

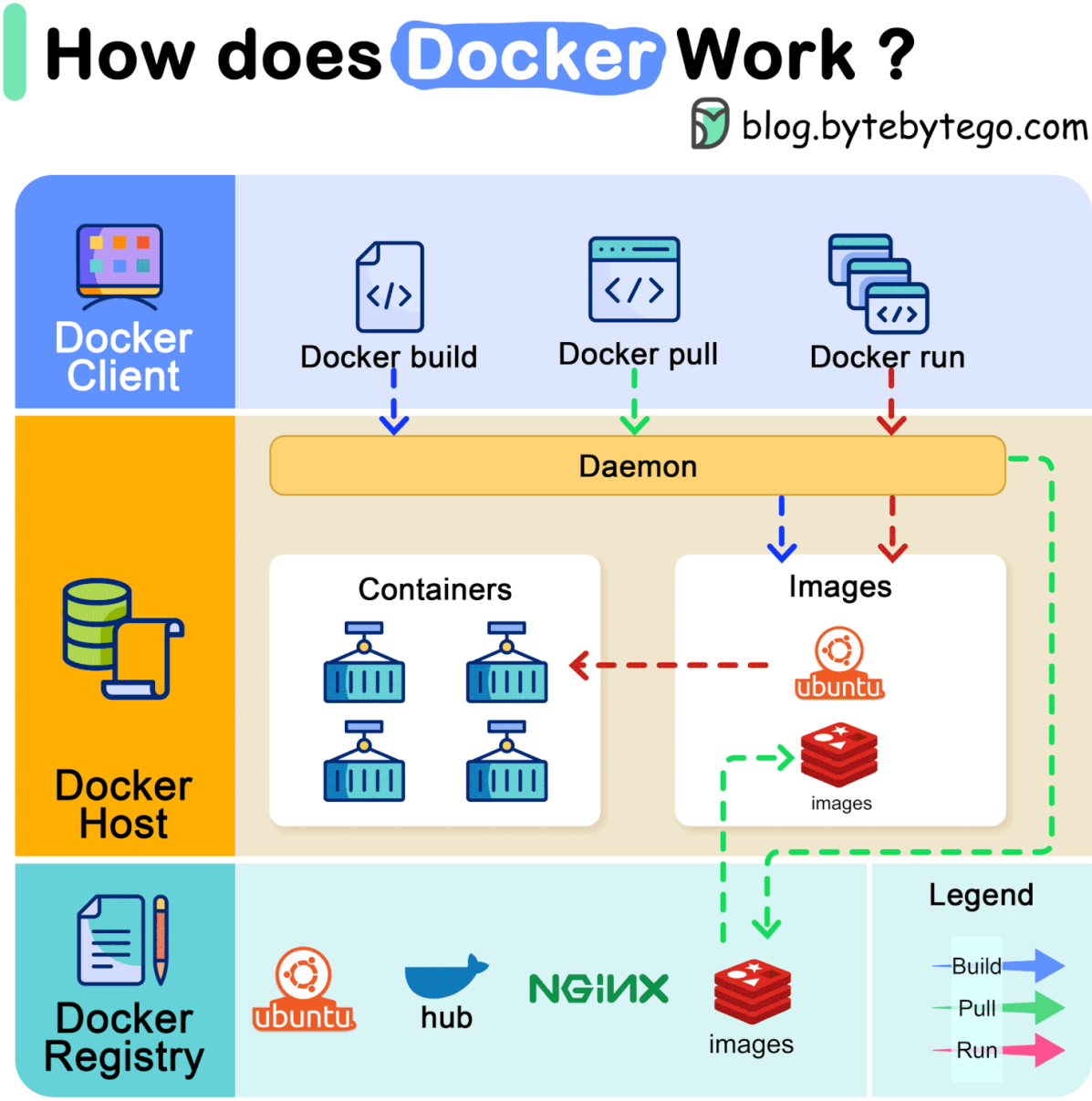
Step 9: If the UAT testing is successful, the builds become release candidates and will be deployed to the production environment on schedule.

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 does Docker Work? Is Docker still relevant?

We just made a video on this topic.



Docker's architecture comprises three main components:

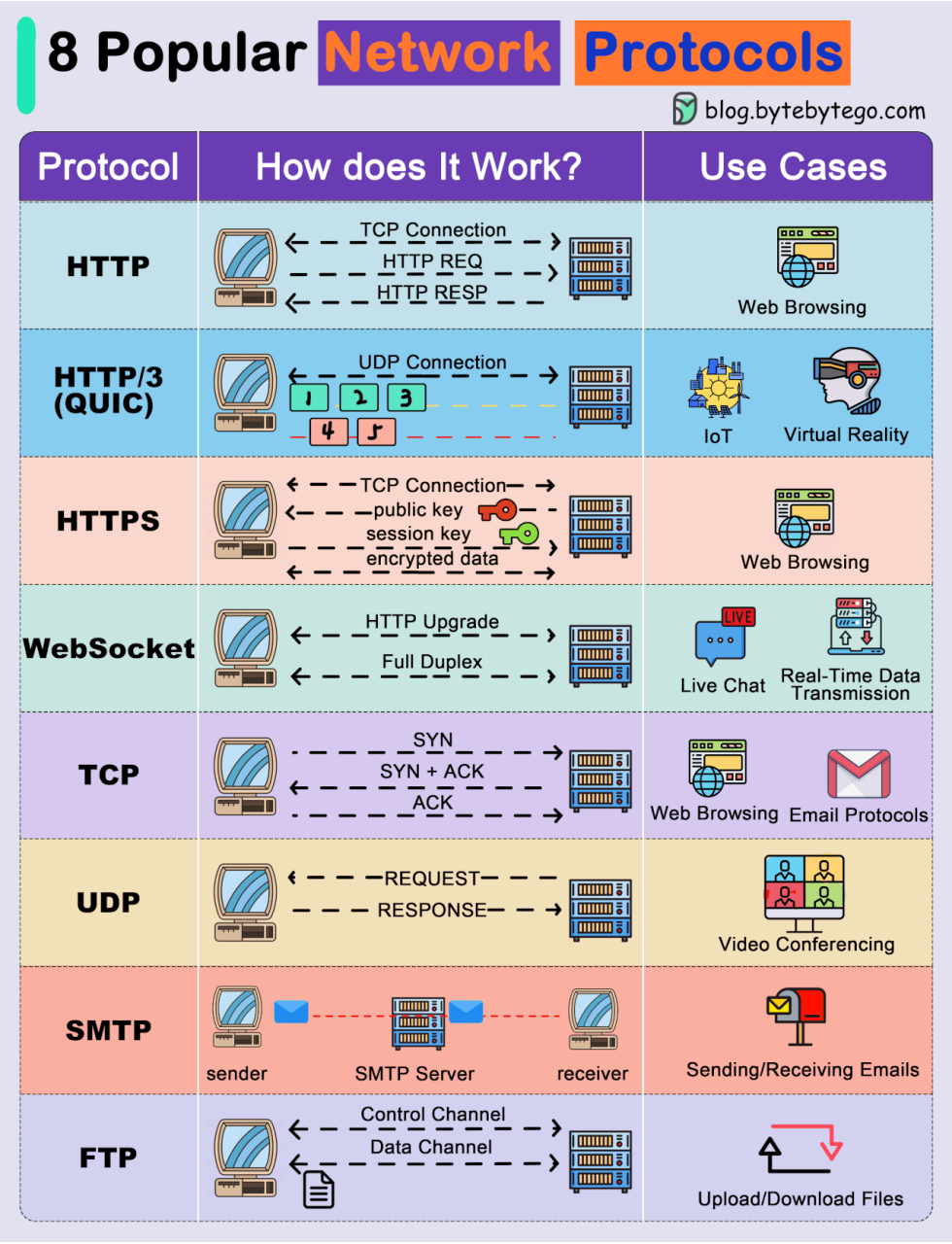
- Docker Client
  This is the interface through which users interact. It communicates with the Docker daemon.

- Docker Host
  Here, the Docker daemon listens for Docker API requests and manages various Docker objects, including images, containers, networks, and volumes.

- Docker Registry
  This is where Docker images are stored. Docker Hub, for instance, is a widely-used public registry.

Explaining 8 Popular Network Protocols in 1 Diagram

Network protocols are standard methods of transferring data between two computers in a network.



1. HTTP (HyperText Transfer Protocol)
   HTTP is a protocol for fetching resources such as HTML documents. It is the foundation of any data exchange on the Web and it is a client-server protocol.

2. HTTP/3
   HTTP/3 is the next major revision of the HTTP. It runs on QUIC, a new transport protocol designed for mobile-heavy internet usage. It relies on UDP instead of TCP, which enables faster web page responsiveness. VR applications demand more bandwidth to render intricate details of a virtual scene and will likely benefit from migrating to HTTP/3 powered by QUIC.

3. HTTPS (HyperText Transfer Protocol Secure)
   HTTPS extends HTTP and uses encryption for secure communications.

4. WebSocket
   WebSocket is a protocol that provides full-duplex communications over TCP. Clients establish WebSockets to receive real-time updates from the back-end services. Unlike REST, which always “pulls” data, WebSocket enables data to be “pushed”. Applications, like online gaming, stock trading, and messaging apps leverage WebSocket for real-time communication.

5. TCP (Transmission Control Protocol)
   TCP is is designed to send packets across the internet and ensure the successful delivery of data and messages over networks. Many application-layer protocols build on top of TCP.

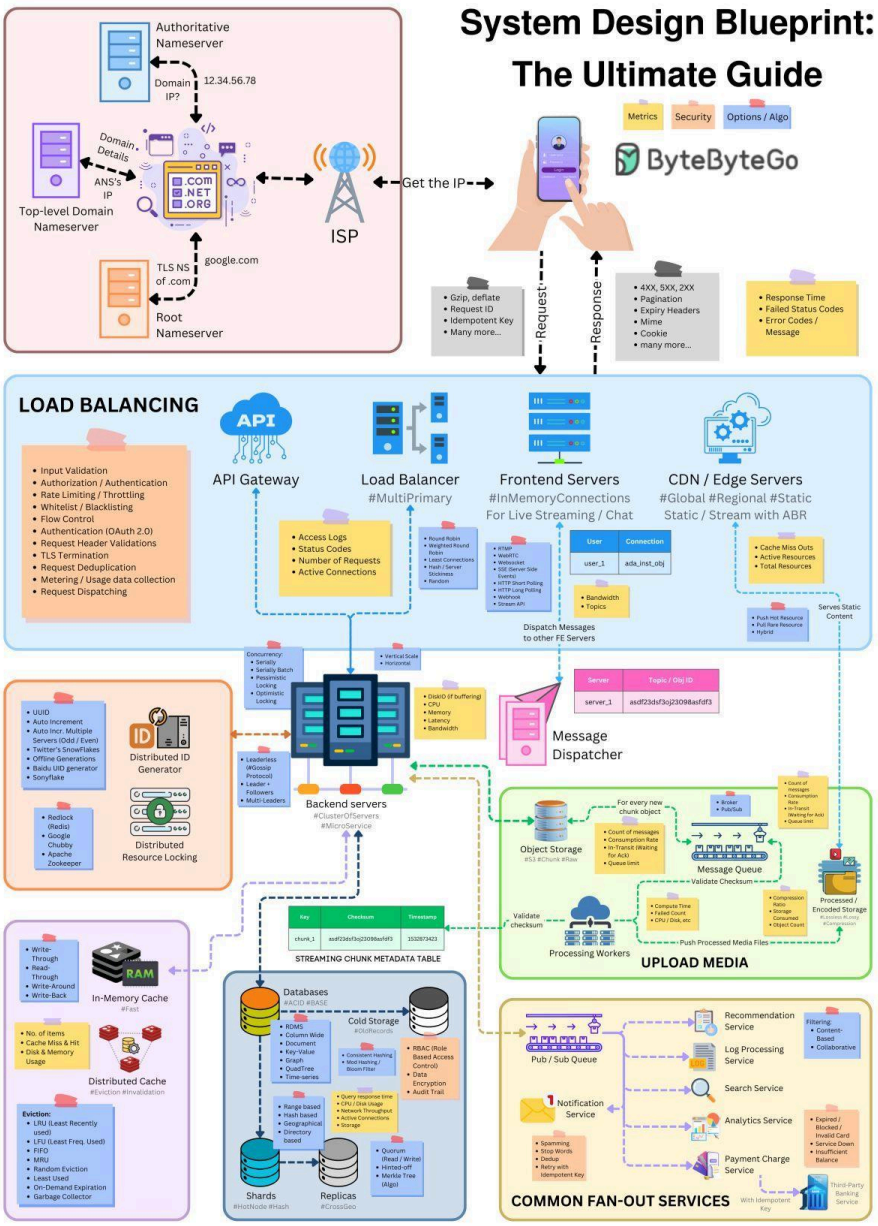
6. UDP (User Datagram Protocol)
   UDP sends packets directly to a target computer, without establishing a connection first. UDP is commonly used in time-sensitive communications where occasionally dropping packets is better than waiting. Voice and video traffic are often sent using this protocol.

7. SMTP (Simple Mail Transfer Protocol)
   SMTP is a standard protocol to transfer electronic mail from one user to another.

8. FTP (File Transfer Protocol)
   FTP is used to transfer computer files between client and server. It has separate connections for the control channel and data channel.

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.

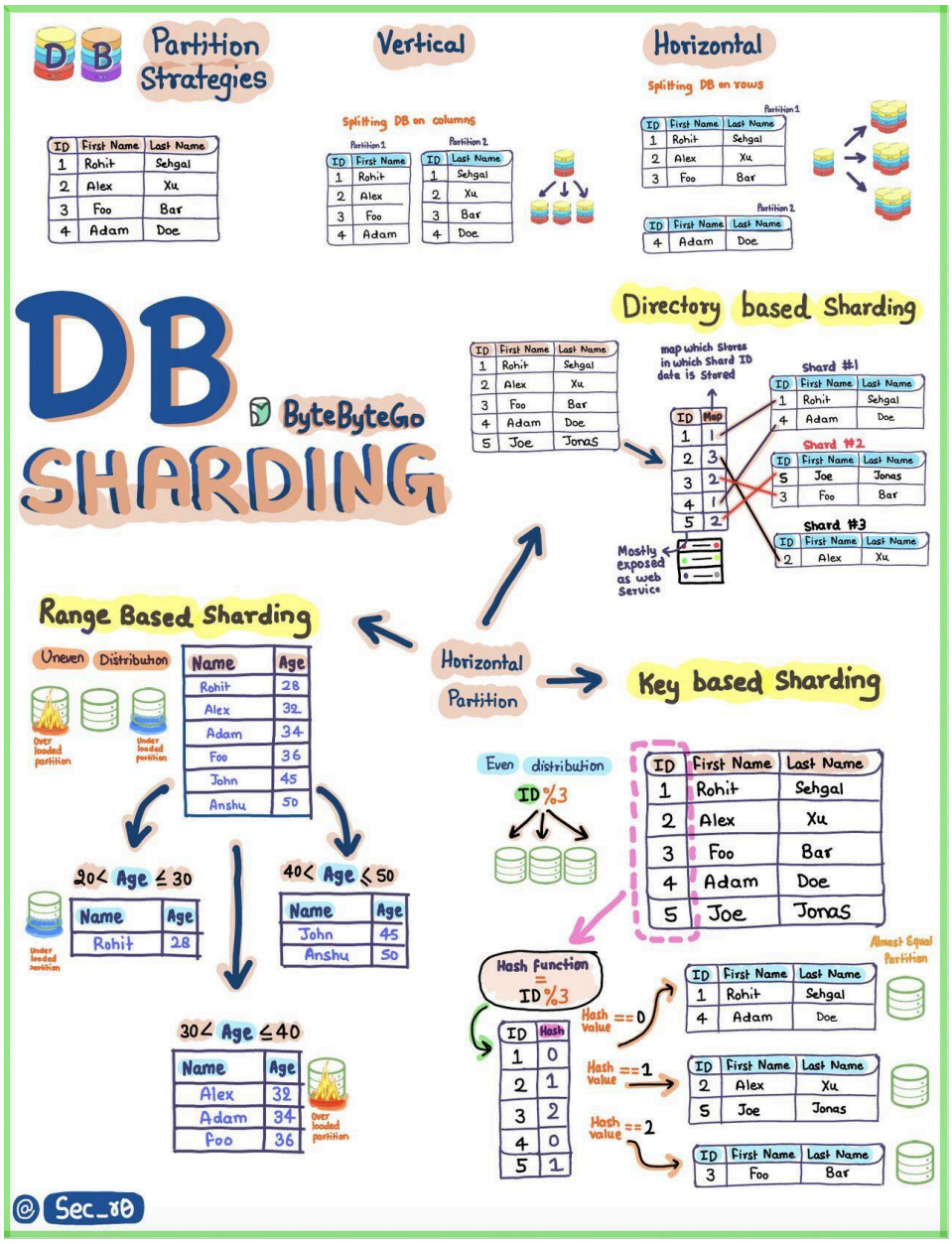
This briefly touches on the following discussion points:

- 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

Key Concepts to Understand Database Sharding

In this concise and visually engaging resource, we break down the key concepts of database partitioning, explaining both vertical and horizontal strategies.



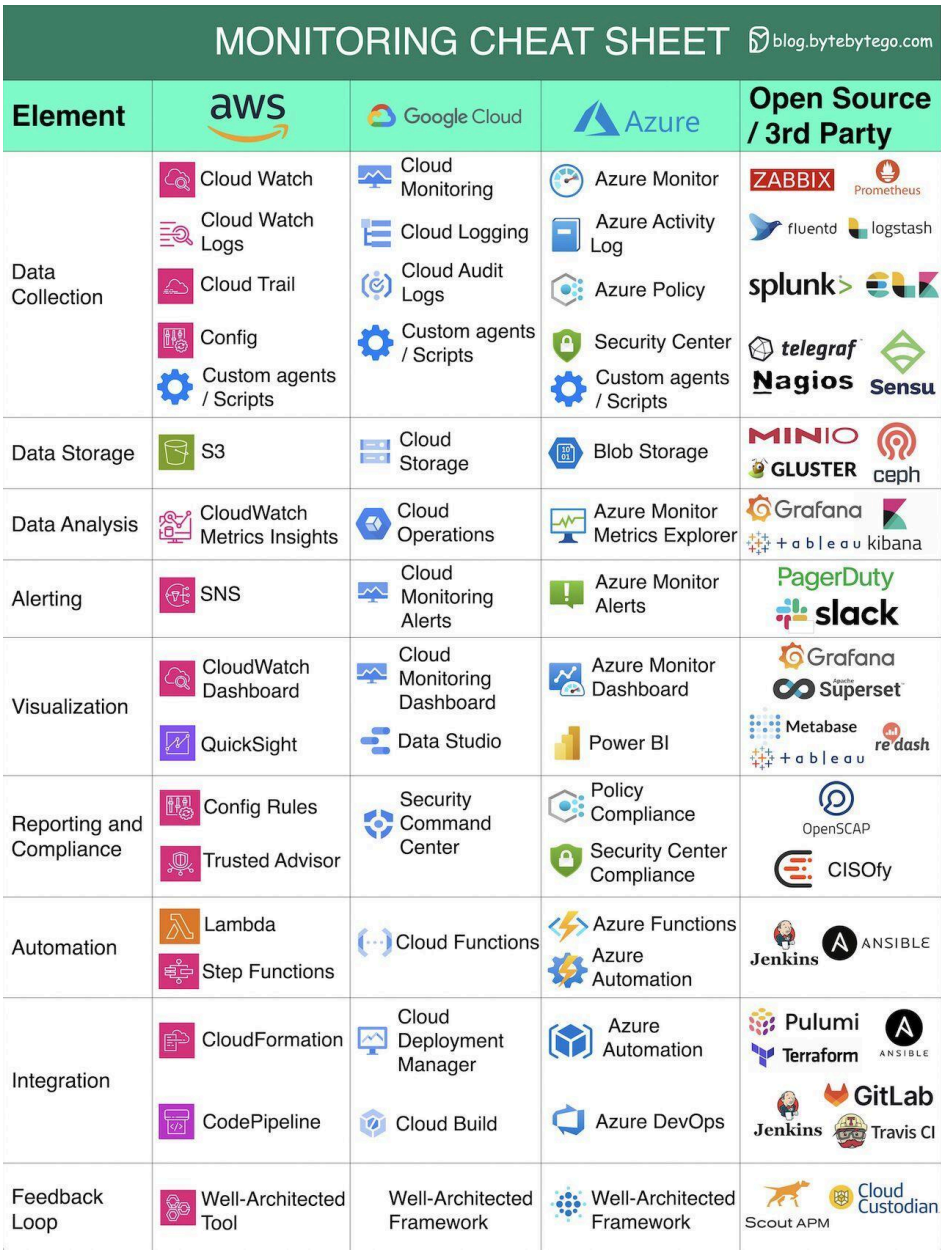
1. Range-Based Sharding: Splitting your data into distinct ranges. Think of it as organizing your books by genre on separate shelves.

2. Key-Based Sharding (with a dash of %3 hash): Imagine each piece of data having a unique key, and we distribute them based on a specific rule. It's like sorting your playing cards by suit and number.
3. Directory-Based Sharding: A directory, like a phone book, helps you quickly find the information you need. Similarly, this technique uses a directory to route data efficiently.

Over to you: What are some other ways to scale a database?

A nice cheat sheet of different monitoring infrastructure in cloud services

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.



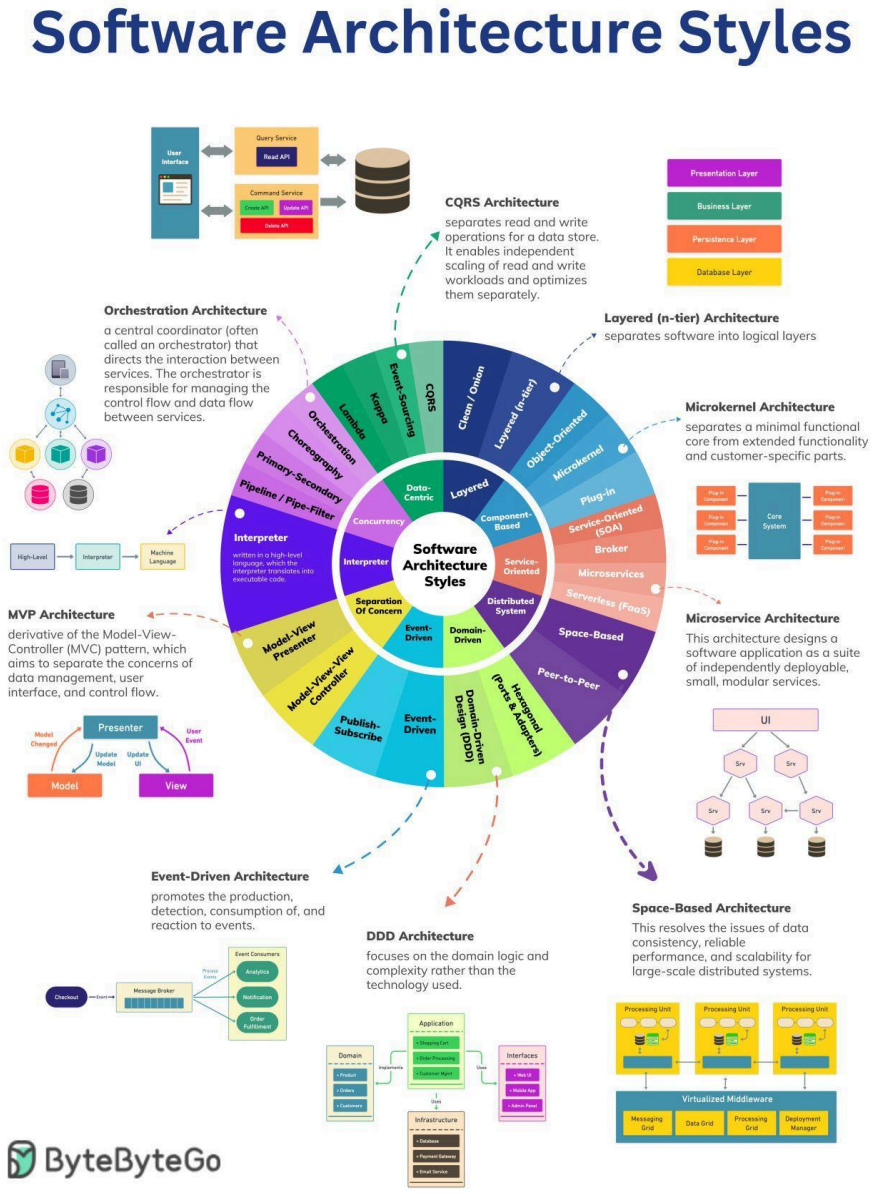
Let's delve into the essential monitoring aspects covered:

- 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?

Top 5 Software Architectural Patterns

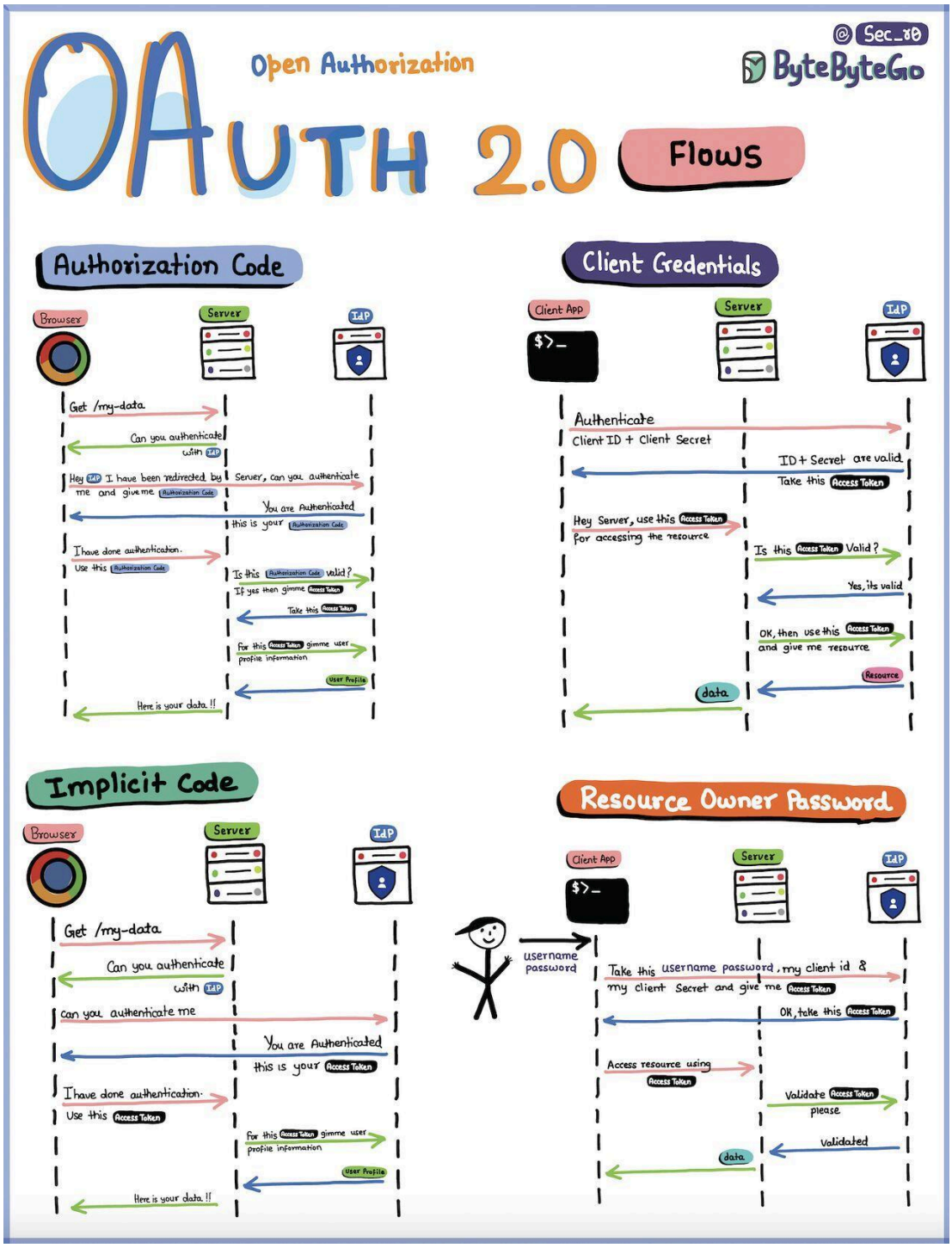


In software development, architecture plays a crucial role in shaping the structure and behavior of software systems. It provides a blueprint for system design, detailing how components interact with each other to deliver specific functionality. They also offer solutions to common problems, saving time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

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?

How did AWS grow from just a few services in 2006 to over 200 fully-featured services?

Let's take a look.

Since 2006, it has become a cloud computing leader, offering foundational infrastructure, platforms, and advanced capabilities like serverless computing and AI.

This expansion empowered innovation, allowing complex applications without extensive hardware management. AWS also explored edge and quantum computing, staying at tech's forefront.

This evolution mirrors cloud computing's shift from niche to essential, benefiting global businesses with efficiency and scalability

Happy to present the curated list of AWS services introduced over the years below.

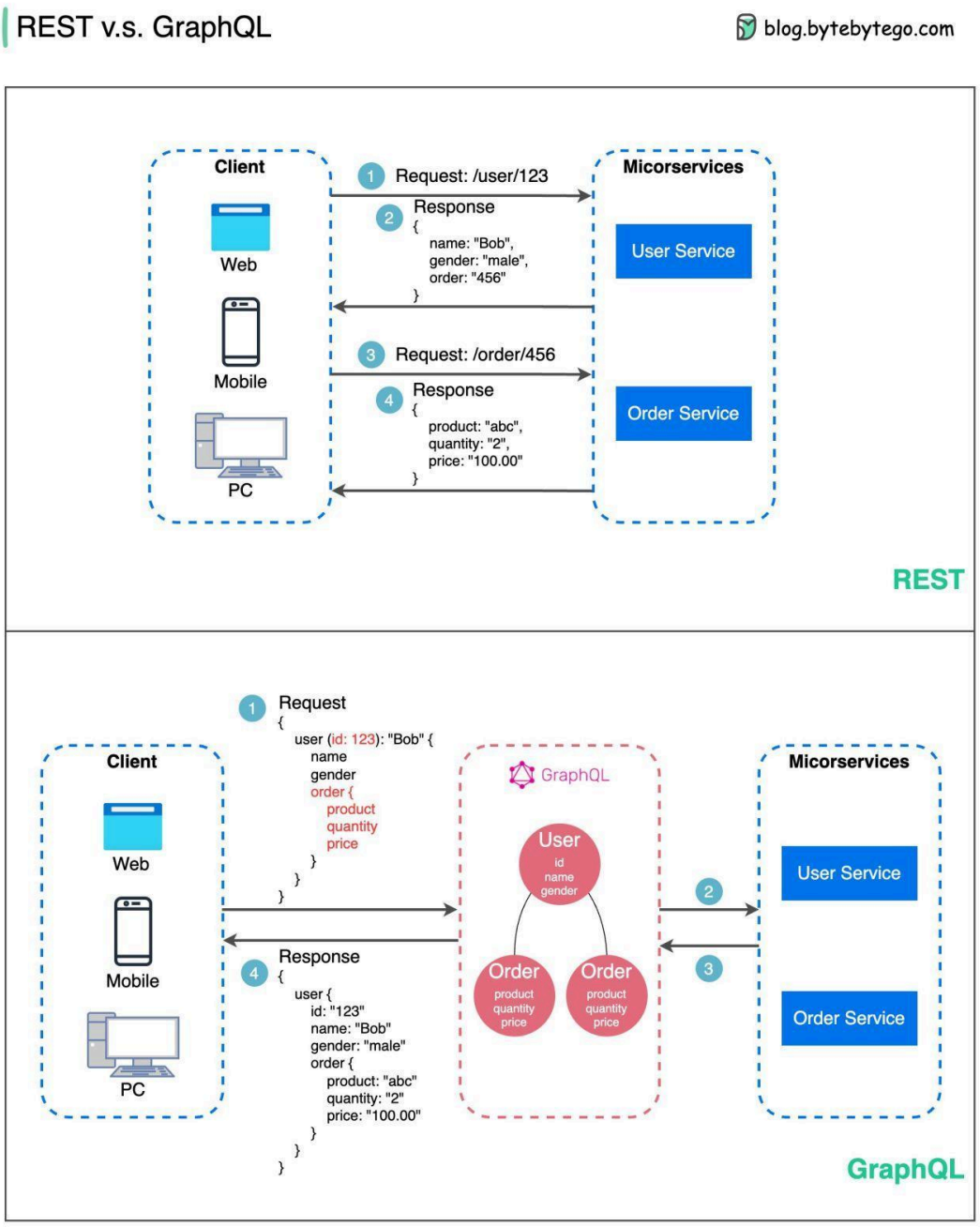
Note:

- The announcement or preview year differs from the public release year for certain services. In these cases, we've noted the service under the release year
- Unreleased services noted in announcement years

Over to you: Are you excited about all the new services, or do you find it overwhelming?

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

The diagram below shows the quick comparison between REST and GraphQL.



- GraphQL is a query language for APIs developed by Meta. It provides a complete description of the data in the API and gives clients the power to ask for exactly what they need.

- GraphQL servers sit in between the client and the backend services.

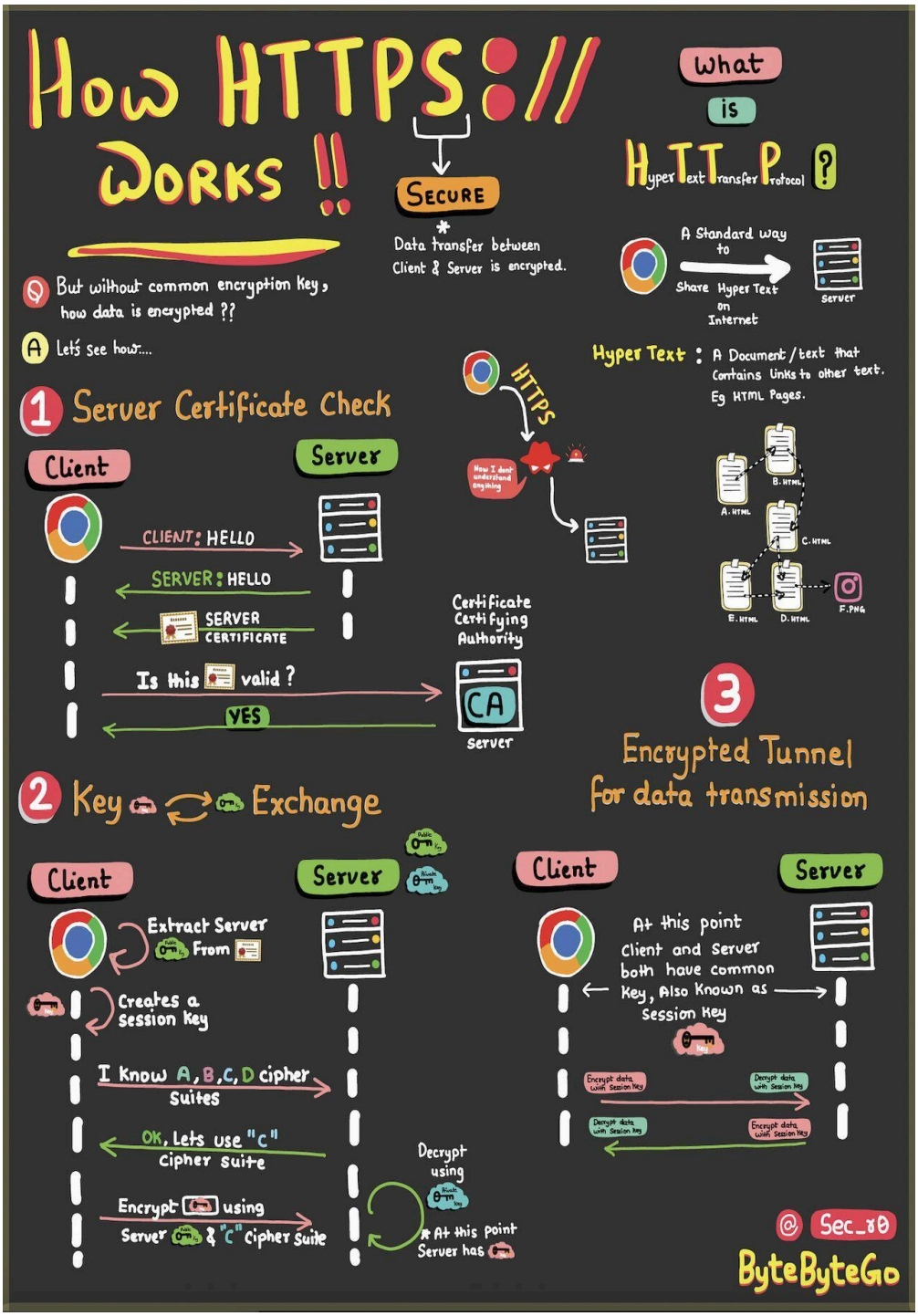
- GraphQL can aggregate multiple REST requests into one query. GraphQL server organizes the resources in a graph.

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

Over to you:

1. Is GraphQL a database technology?
2. Do you recommend GraphQL? Why/why not?

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.

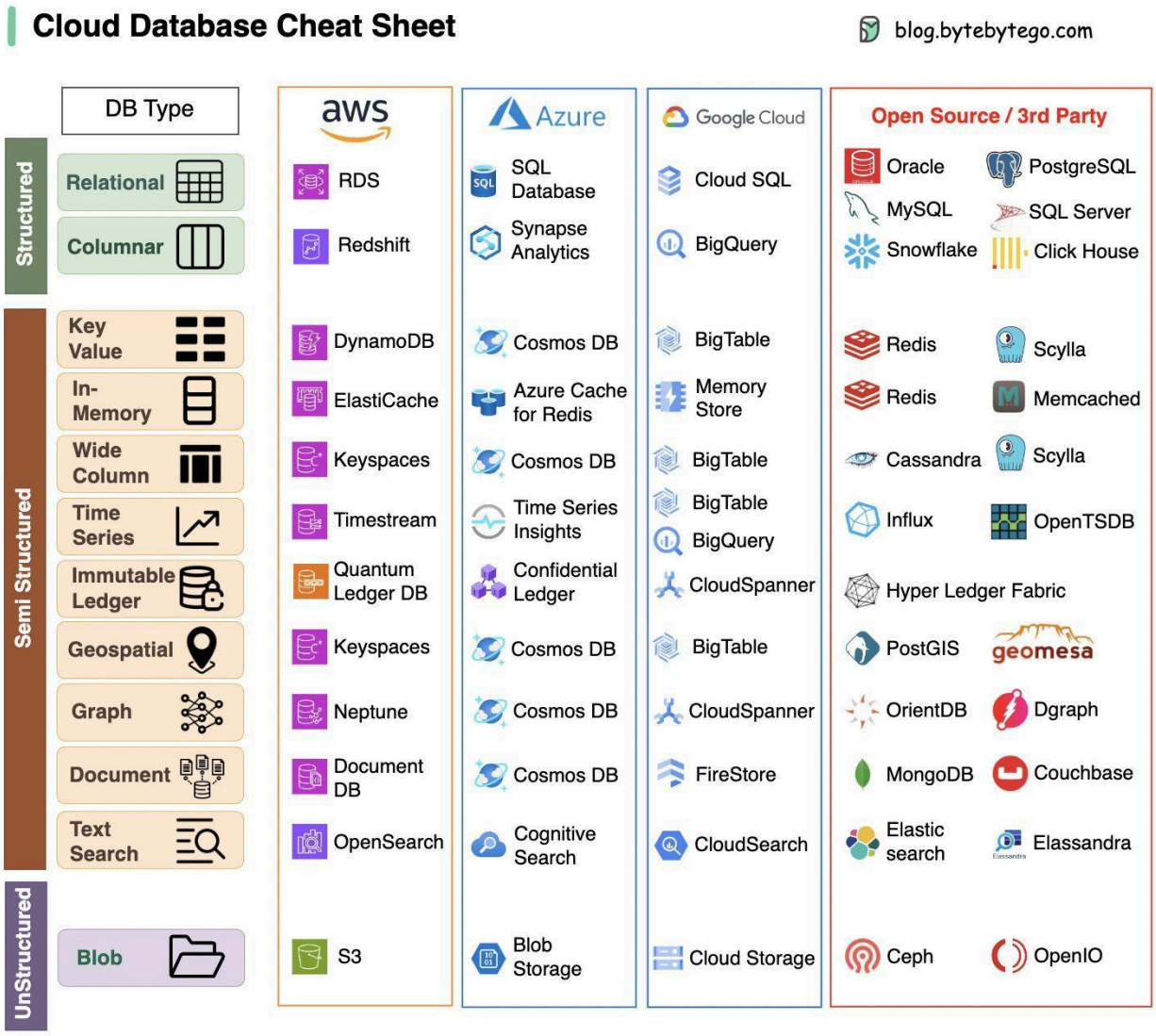
SSL Handshake: Behind the Scenes — Witness the cryptographic protocols that establish a secure connection. Experience the intricate exchange of keys and negotiation.

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?

A nice cheat sheet of different databases in cloud services



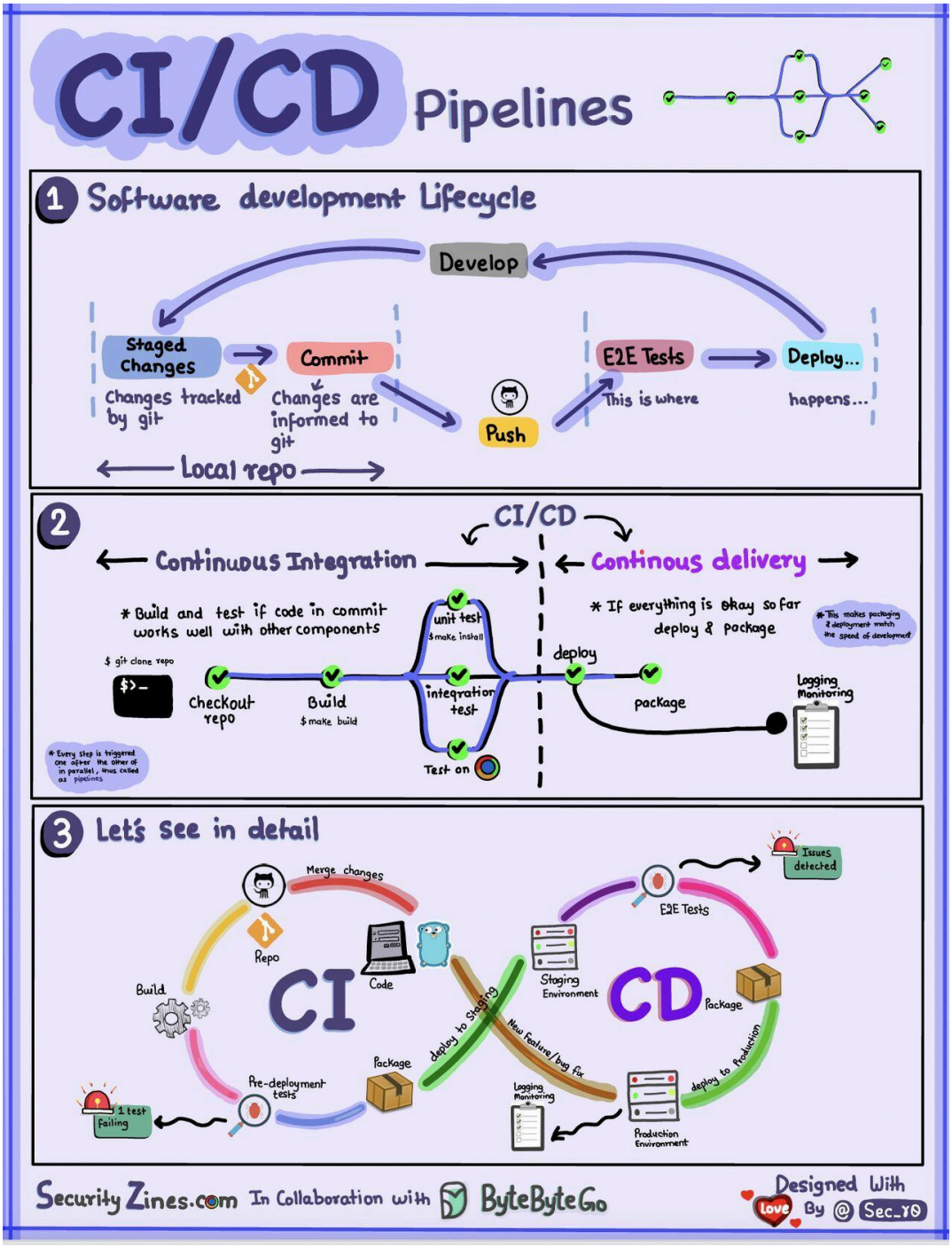
Choosing the right database for your project is a complex task. The multitude of database options, each suited to distinct use cases, can quickly lead to decision fatigue.

We hope this cheat sheet provides high level direction to pinpoint the right service that aligns with your project's needs and avoid potential pitfalls.

Note: Google has limited documentation for their database use cases. Even though we did our best to look at what was available and arrived at the best option, some of the entries may be not accurate.

Over to you: Which database have you used in the past, and for what use cases?

CI/CD Pipeline Explained in Simple Terms



Section 1 - SDLC with CI/CD

The software development life cycle (SDLC) consists of several key stages: development, testing, deployment, and maintenance. CI/CD automates and integrates these stages to enable faster, more reliable releases.

When code is pushed to a git repository, it triggers an automated build and test process. End-to-end (e2e) test cases are run to validate the code. If tests pass, the code can be automatically deployed to staging/production. If issues are found, the code is sent back to development for bug fixing. This automation provides fast feedback to developers and reduces risk of bugs in production.

Section 2 - Difference between CI and CD
Continuous Integration (CI) automates the build, test, and merge process. It runs tests whenever code is committed to detect integration issues early. This encourages frequent code commits and rapid feedback.

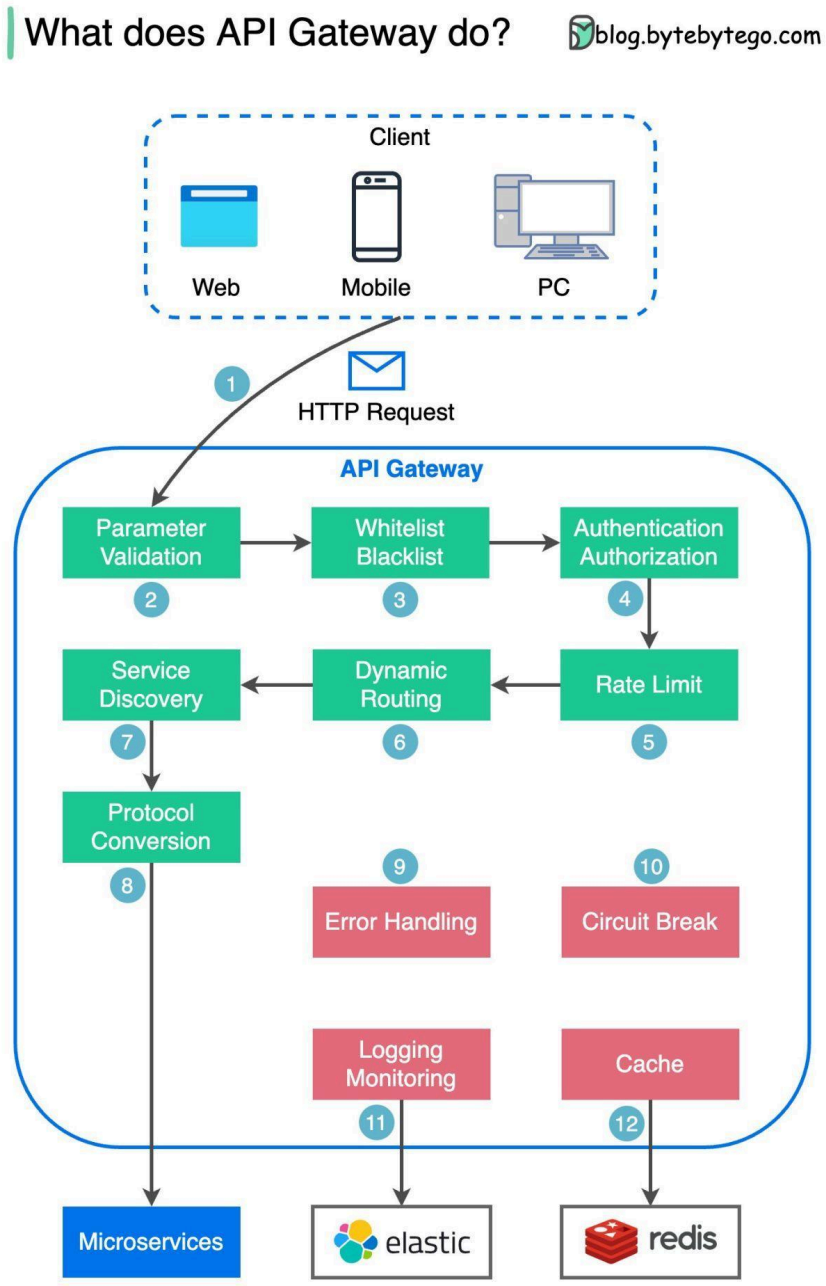
Continuous Delivery (CD) automates release processes like infrastructure changes and deployment. It ensures software can be released reliably at any time through automated workflows. CD may also automate the manual testing and approval steps required before production deployment.

Section 3 - CI/CD Pipeline
A typical CI/CD pipeline has several connected stages:

- Developer commits code changes to source control
- CI server detects changes and triggers build
- Code is compiled, tested (unit, integration tests)
- Test results reported to developer
- On success, artifacts are deployed to staging environments
- Further testing may be done on staging before release
- CD system deploys approved changes to production

What does API gateway do?

The diagram below shows the detail.



Step 1 - The client sends an HTTP request to the API gateway.

Step 2 - The API gateway parses and validates the attributes in the HTTP request.

Step 3 - The API gateway performs allow-list/deny-list checks.

Step 4 - The API gateway talks to an identity provider for authentication and authorization.

Step 5 - The rate limiting rules are applied to the request. If it is over the limit, the request is rejected.

Steps 6 and 7 - Now that the request has passed basic checks, the API gateway finds the relevant service to route to by path matching.

Step 8 - The API gateway transforms the request into the appropriate protocol and sends it to backend microservices.

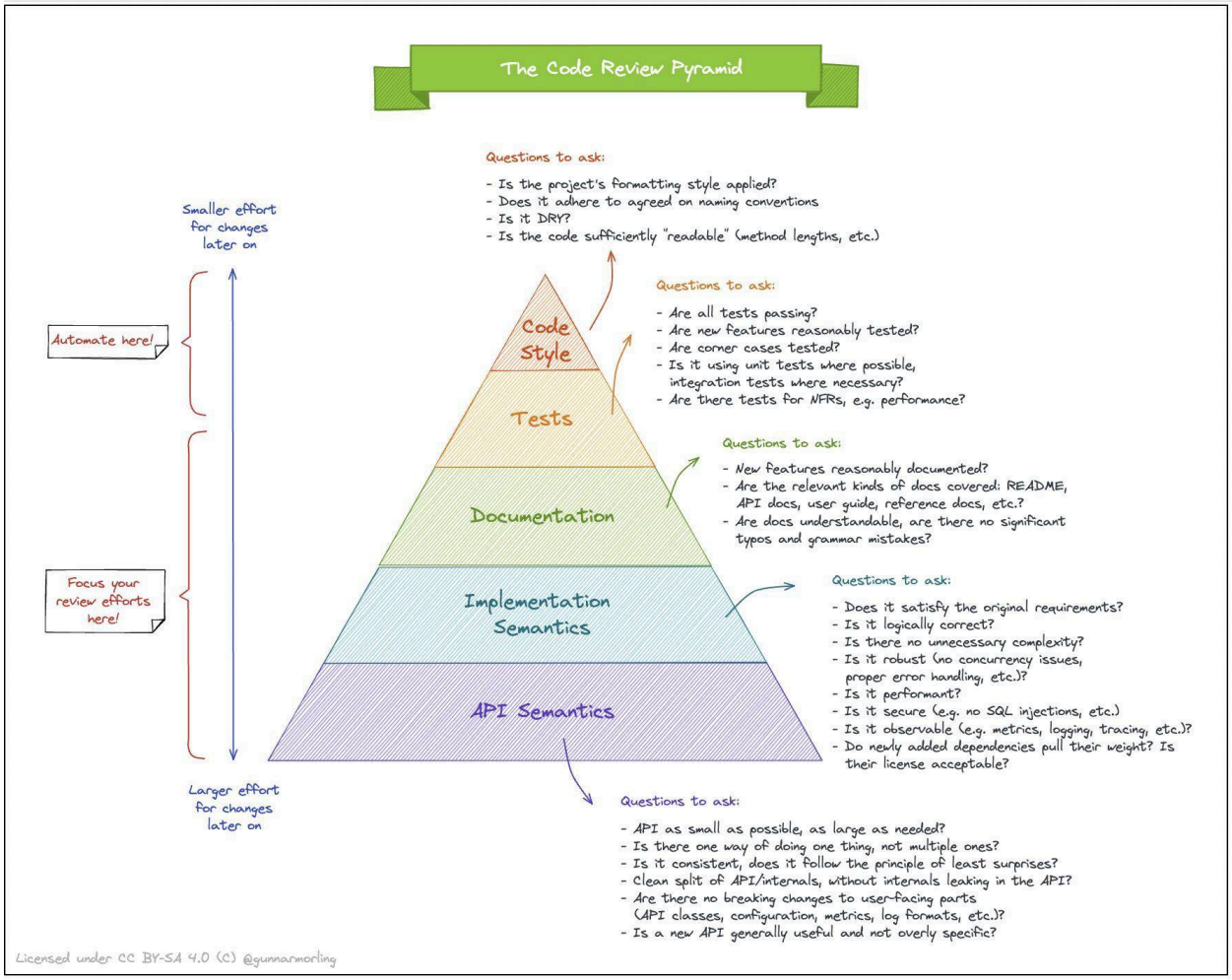
Steps 9-12: The API gateway can handle errors properly, and deals with faults if the error takes a longer time to recover (circuit break). It can also leverage ELK (Elastic-Logstash-Kibana) stack for logging and monitoring. We sometimes cache data in the API gateway.

Over to you:

1. What's the difference between a load balancer and an API gateway?
2. Do we need to use different API gateways for PC, mobile and browser separately?

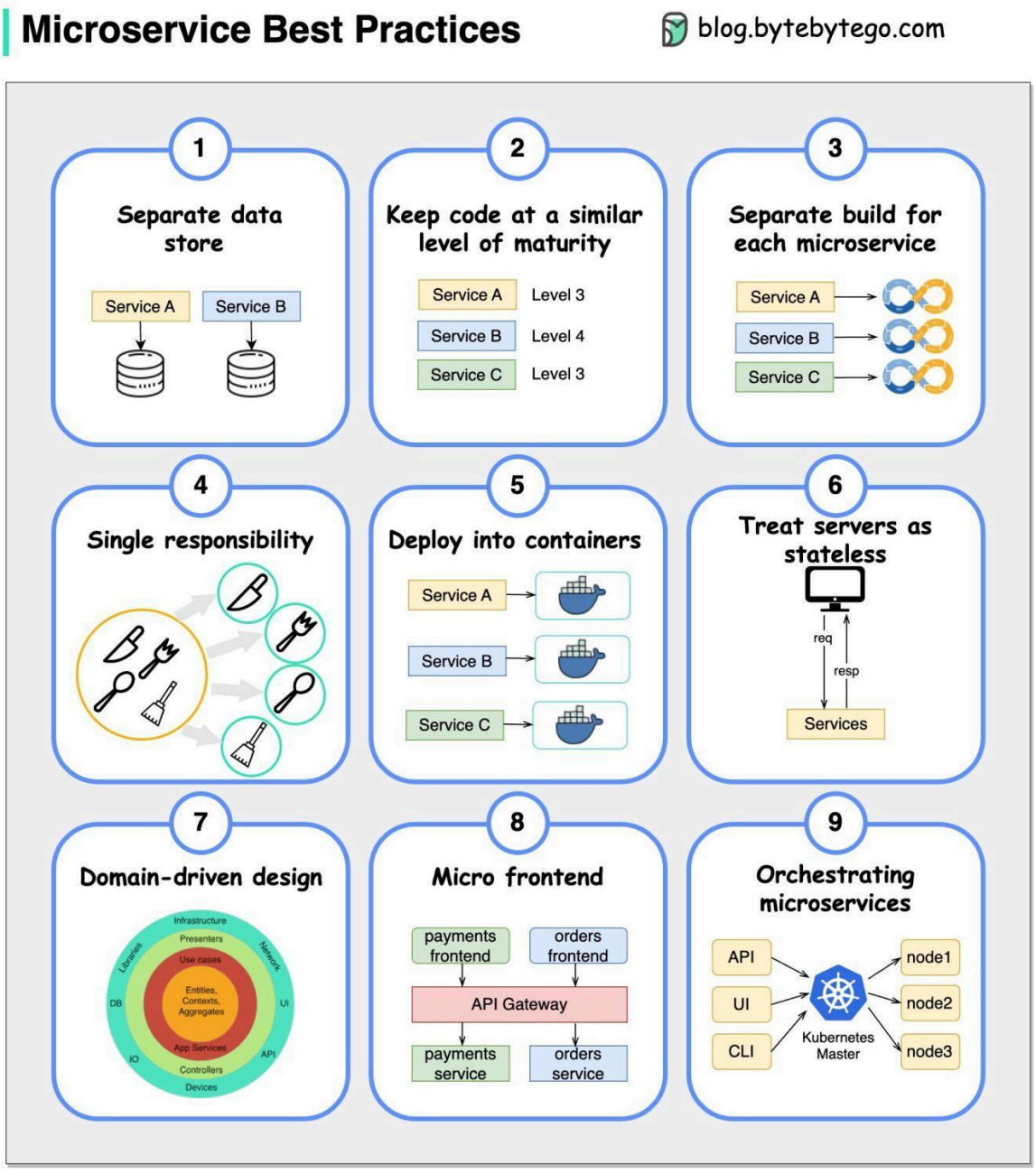
The Code Review Pyramid

By Gunnar Morling



Over to you - Any other tips for effective code review?

A picture is worth a thousand words: 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

Over to you - what else should be included?

What are the greenest programming languages?

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

The study below runs 10 benchmark problems in 28 languages[1]. It measures the runtime, memory usage, and energy consumption of each language. The abstract of the paper is shown below.

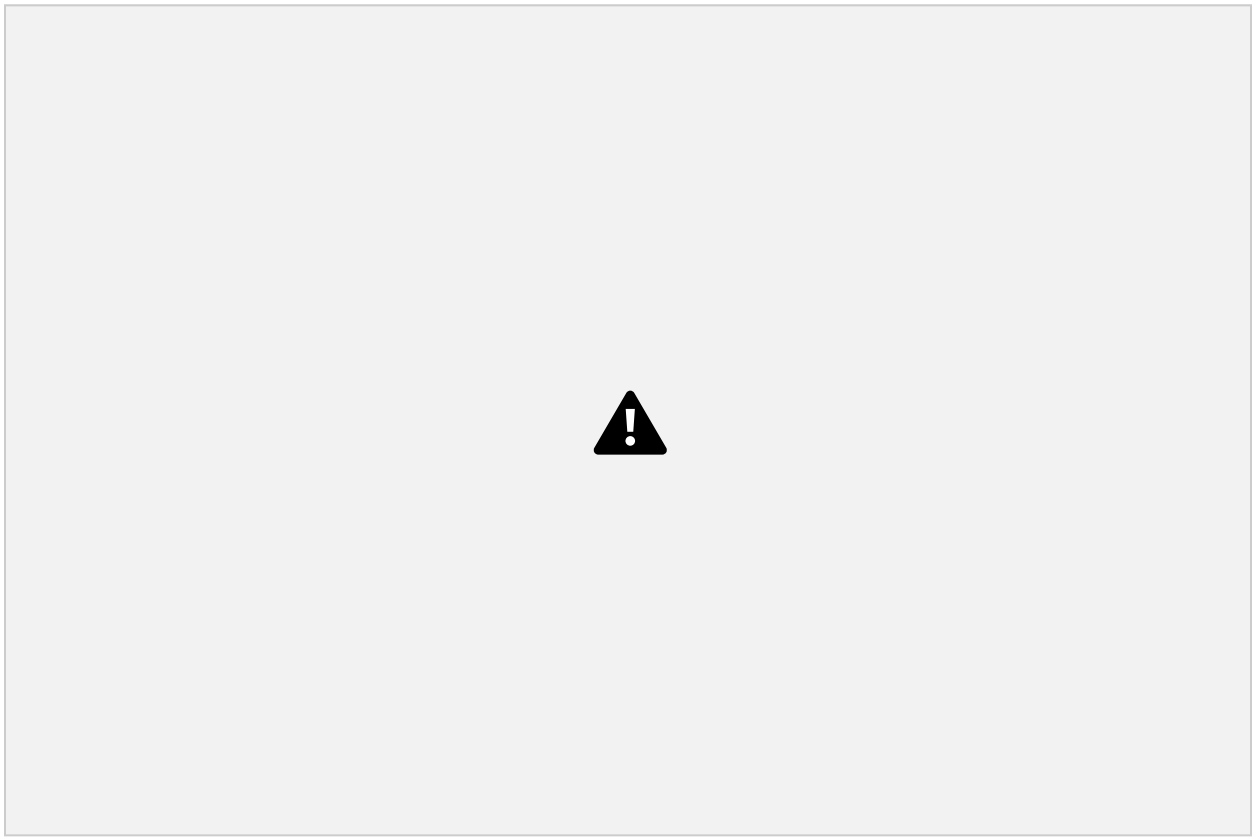
“This paper presents a study of the runtime, memory usage and energy consumption of twenty seven well-known software languages. We monitor the performance of such languages using ten different programming problems, expressed in each of the languages. Our results show interesting findings, such as, slower/faster languages consuming less/more energy, and how memory usage influences energy consumption. We show how to use our results to provide software engineers support to decide which language to use when energy efficiency is a concern”.[2]

Most environmentally friendly languages: C, Rust, and C++
Least environmentally-friendly languages: Ruby, Python, Perl

Over to you: What do you think of the accuracy of this analysis?

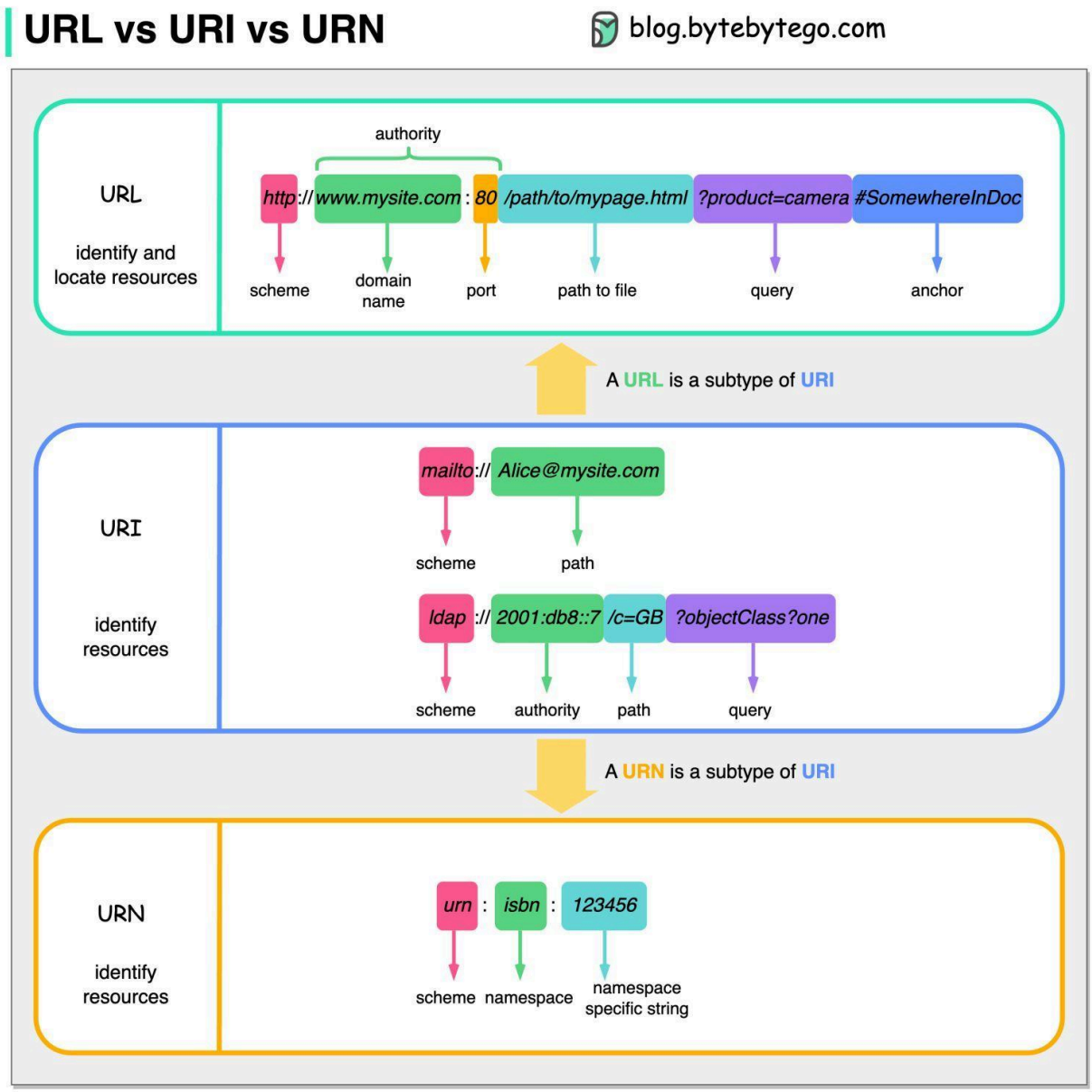
An amazing illustration of how to build a resilient three-tier architecture on AWS

Image Credit: Ankit Jodhani



URL, URI, URN - Do you know the differences?

The diagram below shows a comparison of URL, URI, and URN.



- URI
  URI stands for Uniform Resource Identifier. It identifies a logical or physical resource on the web. URL and URN are subtypes of URI. URL locates a resource, while URN names a resource.

  A URI is composed of the following parts:

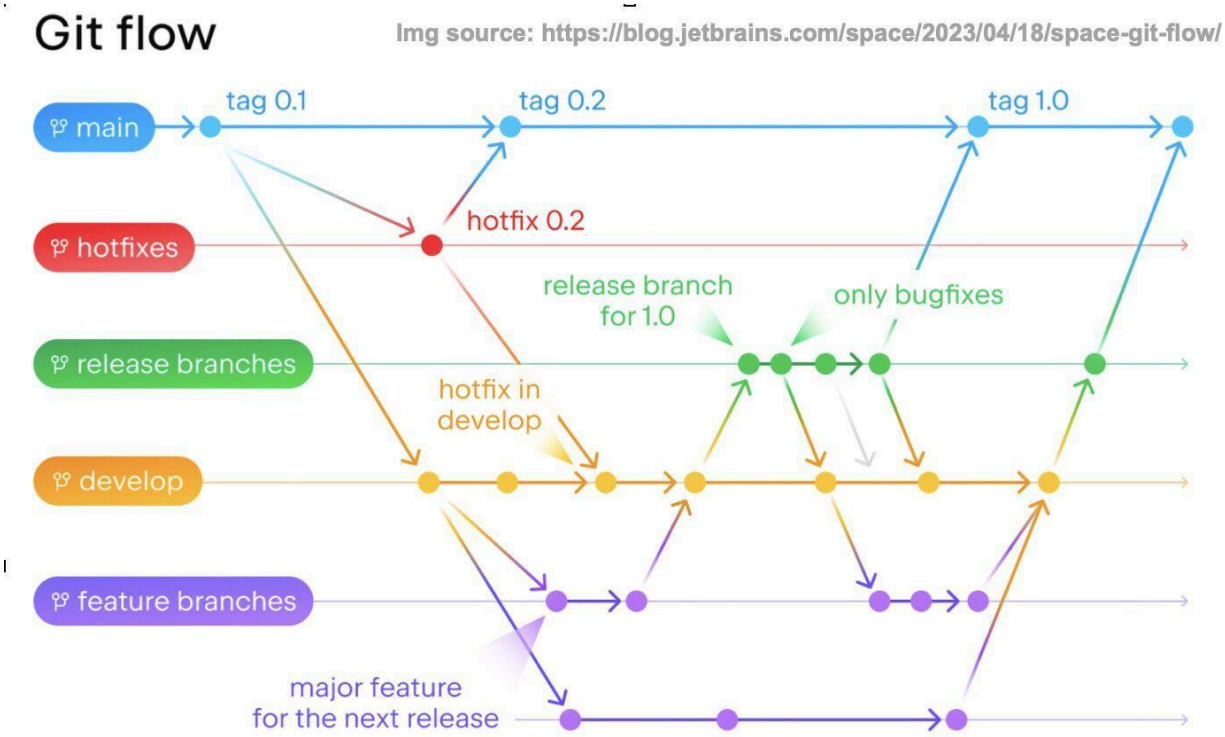
scheme:[//authority]path[?query][#fragment]

- URL
  URL stands for Uniform Resource Locator, the key concept of HTTP. It is the address of a unique resource on the web. It can be used with other protocols like FTP and JDBC.

- URN
  URN stands for Uniform Resource Name. It uses the urn scheme. URNs cannot be used to locate a resource. A simple example given in the diagram is composed of a namespace and a namespace-specific string.

If you would like to learn more detail on the subject, I would recommend W3C's clarification.

What branching strategies does your team use?



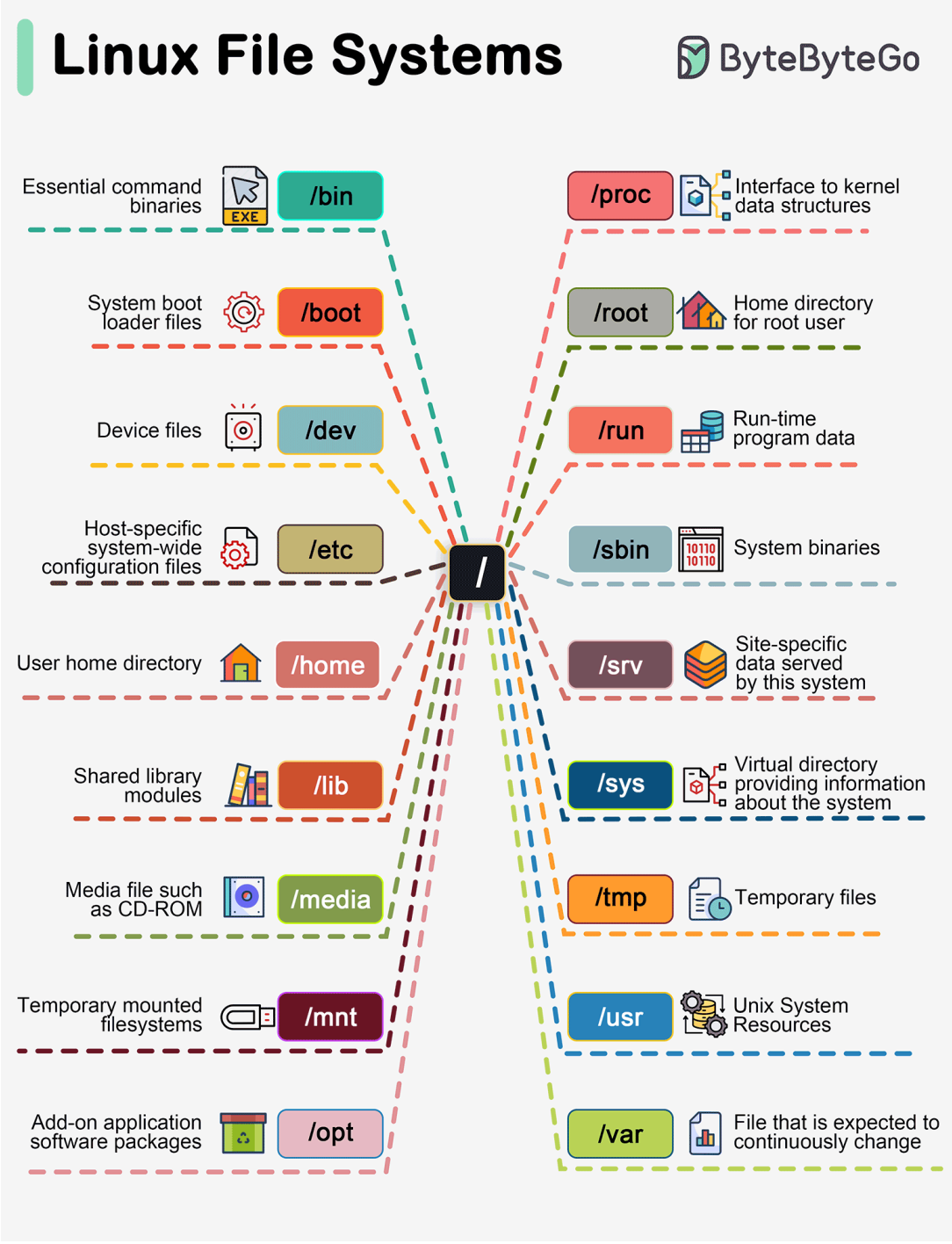
- The `main` branch is for production code only.
- The `develop` branch is for development code.
- `feature` branches are created from the `develop` branch.
- `hotfix` branches are created from the `main` branch.
- `release` branches are created from the `develop` branch.

Teams often employ various branching strategies for managing their code, such as Git flow, feature branches, and trunk-based development.

Out of these options, Git flow or its variations are the most widely favored methods. The illustration by Jetbrains explains how it works.

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.

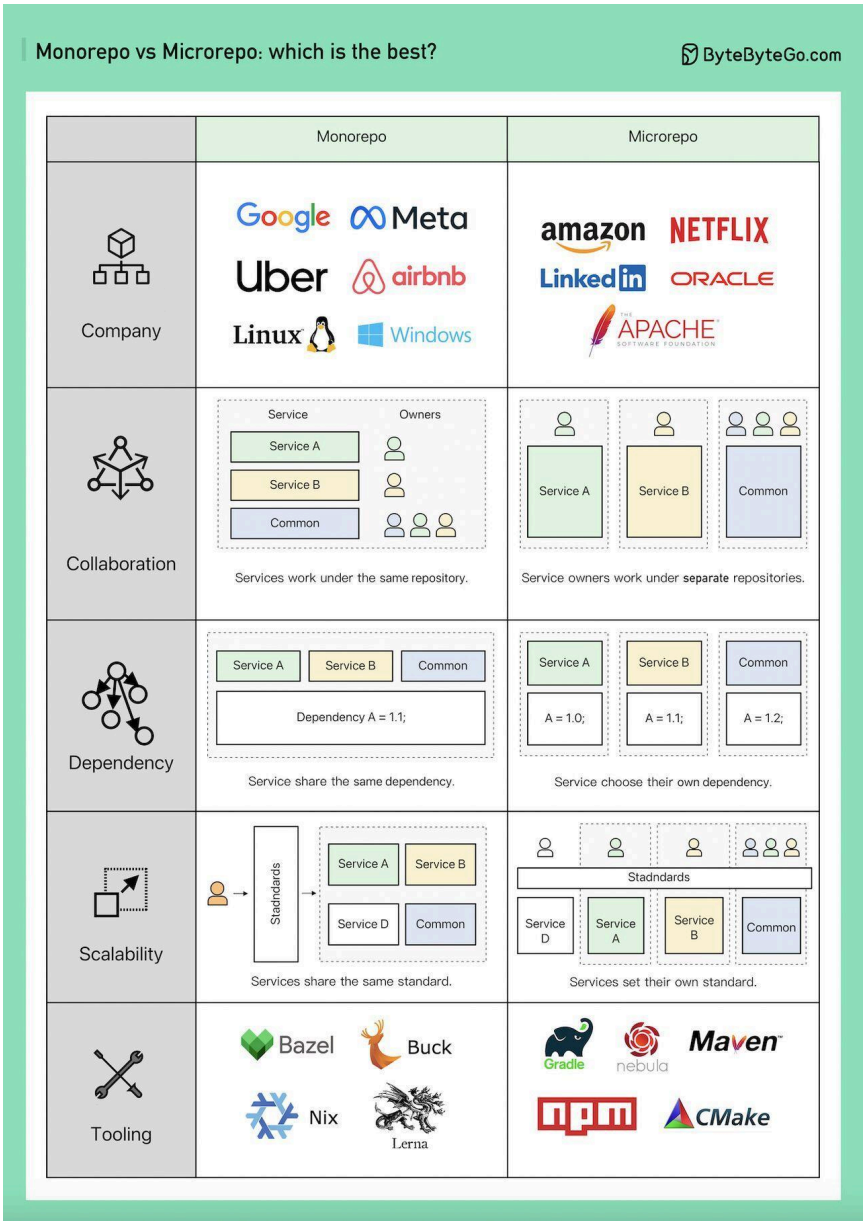
Have fun exploring!

Over to you: What Linux commands are useful for navigating and examining files?

Do you believe that Google, Meta, Uber, and Airbnb put almost all of their code in one repository?

This practice is called a monorepo.

Monorepo vs. Microrepo. Which is the best? Why do different companies choose different options?



Monorepo isn't new; Linux and Windows were both created using Monorepo. To improve scalability and build speed, Google developed its internal dedicated toolchain to scale it faster and strict coding quality standards to keep it consistent.

Amazon and Netflix are major ambassadors of the Microservice philosophy. This approach naturally separates the service code into separate repositories. It scales faster but can lead to governance pain points later on.

Within Monorepo, each service is a folder, and every folder has a BUILD config and OWNERS permission control. Every service member is responsible for their own folder.

On the other hand, in Microrepo, each service is responsible for its repository, with the build config and permissions typically set for the entire repository.

In Monorepo, dependencies are shared across the entire codebase regardless of your business, so when there's a version upgrade, every codebase upgrades their version.

In Microrepo, dependencies are controlled within each repository. Businesses choose when to upgrade their versions based on their own schedules.

Monorepo has a standard for check-ins. Google's code review process is famously known for setting a high bar, ensuring a coherent quality standard for Monorepo, regardless of the business.

Microrepo can either set their own standard or adopt a shared standard by incorporating best practices. It can scale faster for business, but the code quality might be a bit different.

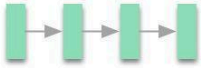
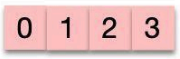
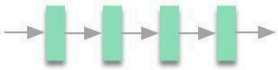
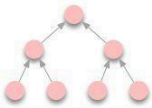
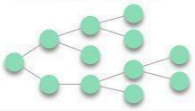
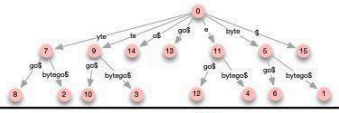
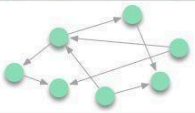
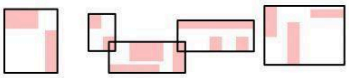
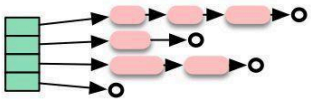
Google engineers built Bazel, and Meta built Buck. There are other open-source tools available, including Nix, Lerna, and others.

Over the years, Microrepo has had more supported tools, including Maven and Gradle for Java, NPM for NodeJS, and CMake for C/C++, among others.

Over to you: Which option do you think is better? Which code repository strategy does your company use?

What are the data structures used in daily life?

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
- heap: task scheduling
- tree: keep the HTML document, or for AI decision

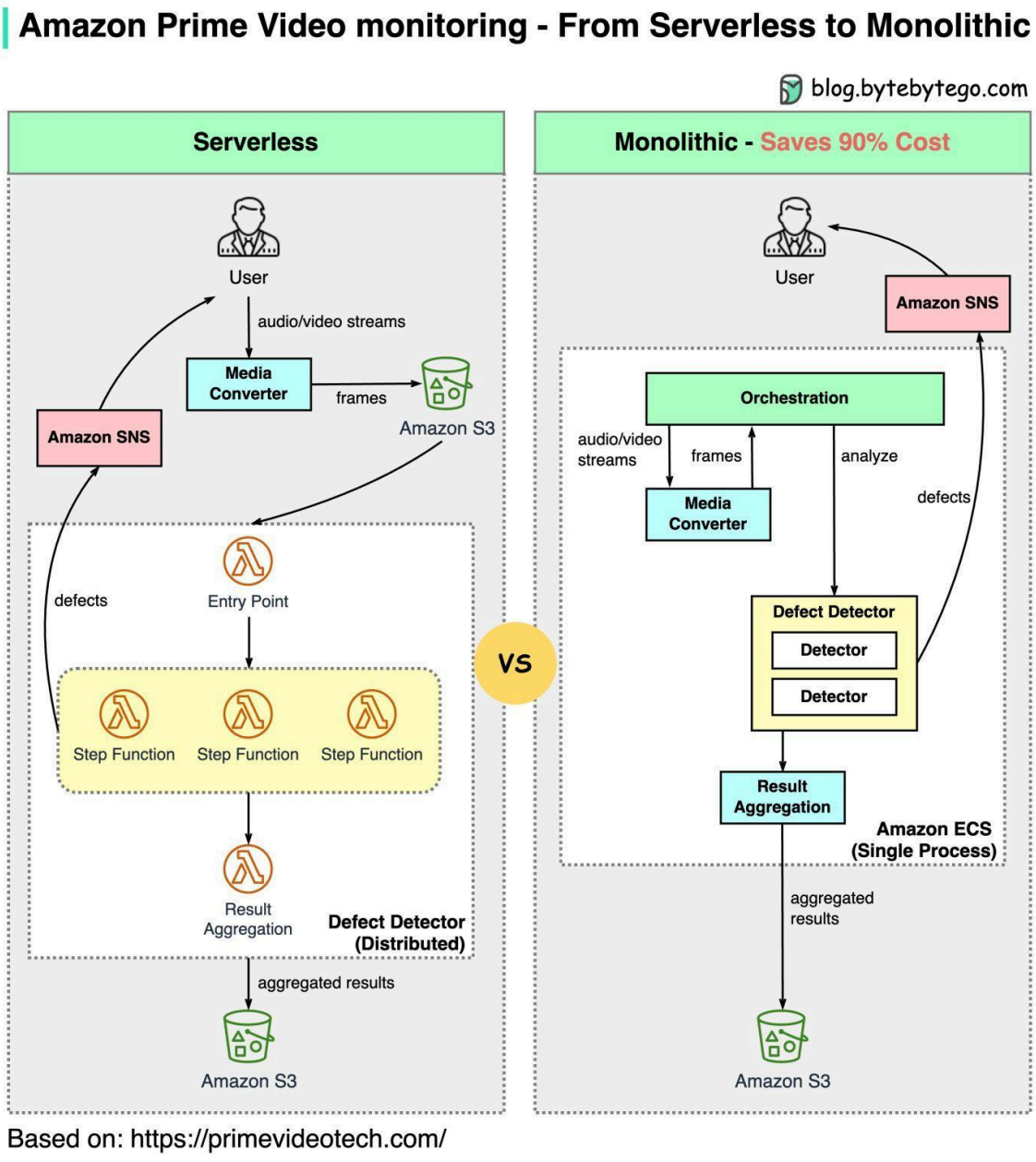
- suffix tree: for searching string in a document
- graph: for tracking friendship, or path finding
- r-tree: for finding the nearest neighbor
- vertex buffer: for sending data to GPU for rendering

To conclude, data structures play an important role in our daily lives, both in our technology and in our experiences. Engineers should be aware of these data structures and their use cases to create effective and efficient solutions.

Over to you: Which additional data structures have we overlooked?

Why did Amazon Prime Video monitoring move from serverless to monolithic? How can it save 90% cost?

The diagram below shows the architecture comparison before and after the migration.



Based on: https://primevideotech.com/

What is Amazon Prime Video Monitoring Service?

Prime Video service needs to monitor the quality of thousands of live streams. The monitoring tool automatically analyzes the streams in real time and identifies quality issues like block

corruption, video freeze, and sync problems. This is an important process for customer satisfaction.

There are 3 steps: media converter, defect detector, and real-time notification.

- What is the problem with the old architecture?

  The old architecture was based on Amazon Lambda, which was good for building services quickly. However, it was not cost-effective when running the architecture at a high scale. The two most expensive operations are:

  1. The orchestration workflow - AWS step functions charge users by state transitions and the orchestration performs multiple state transitions every second.

  2. Data passing between distributed components - the intermediate data is stored in Amazon S3 so that the next stage can download. The download can be costly when the volume is high.

- Monolithic architecture saves 90% cost

  A monolithic architecture is designed to address the cost issues. There are still 3 components, but the media converter and defect detector are deployed in the same process, saving the cost of passing data over the network. Surprisingly, this approach to deployment architecture change led to 90% cost savings!

  This is an interesting and unique case study because microservices have become a go-to and fashionable choice in the tech industry. It's good to see that we are having more discussions about evolving the architecture and having more honest discussions about its pros and cons. Decomposing components into distributed microservices comes with a cost.

- What did Amazon leaders say about this?
  Amazon CTO Werner Vogels: “Building **evolvable software systems** is a strategy, not a religion. And revisiting your architectures with an open mind is a must.”

  Ex Amazon VP Sustainability Adrian Cockcroft: “The Prime Video team had followed a path I call **Serverless First**…I don’t advocate **Serverless Only**”.

👉 Over to you: Does microservice architecture solve an architecture problem or an organizational problem?

18 Most-used Linux Commands You Should Know

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

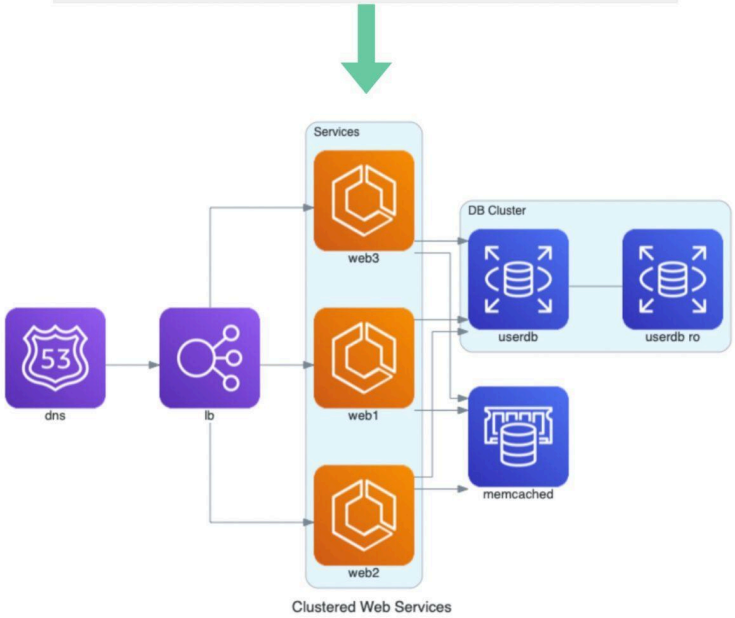
Over to you: What is your favorite Linux command?

Would it be nice if the code we wrote automatically turned into architecture diagrams?

I recently discovered a Github repo that does exactly this: Diagram as Code for prototyping cloud system architectures.

Diagram as Code

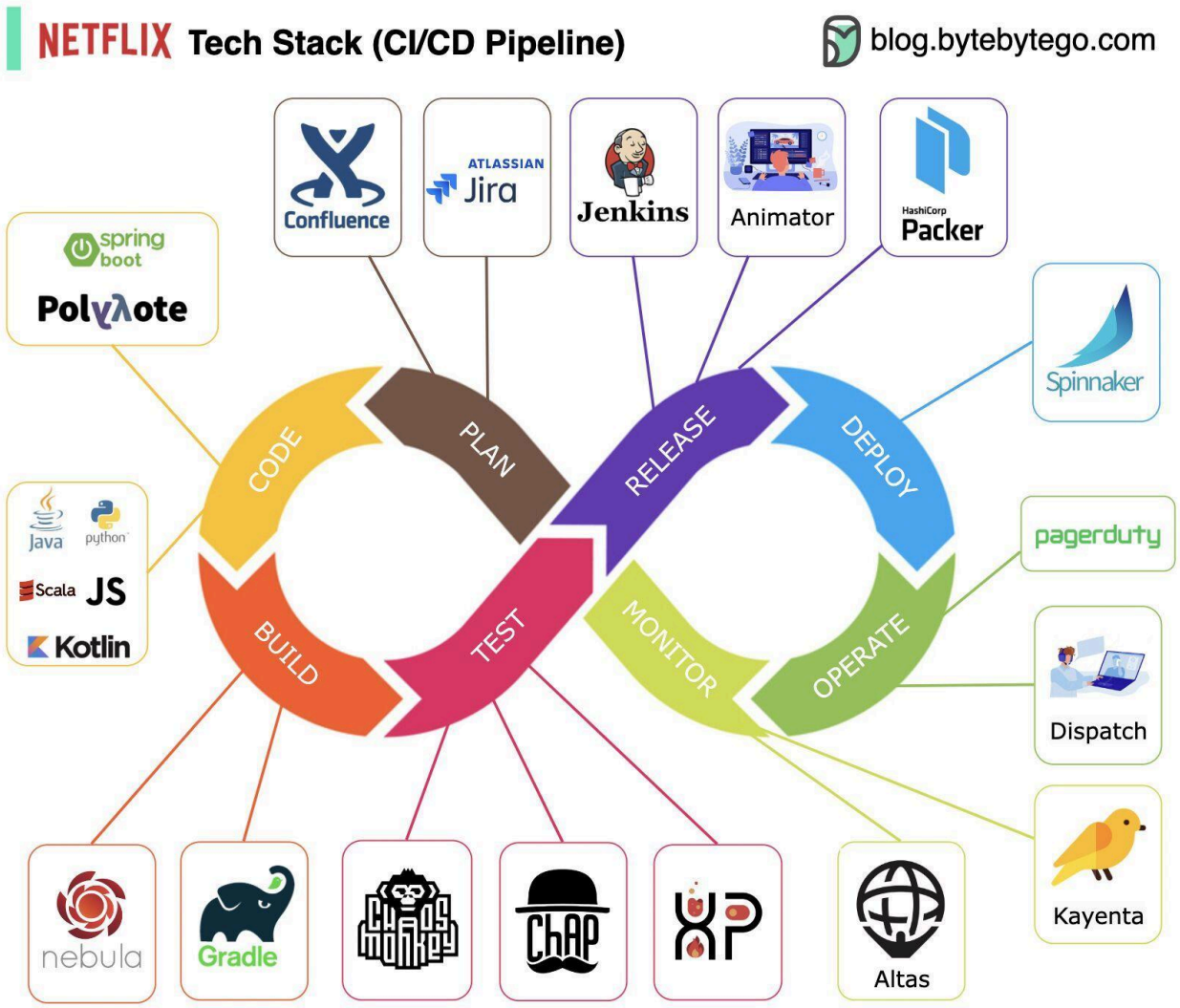
```
from diagrams import Cluster, Diagram
from diagrams.aws.compute import ECS
from diagrams.aws.database import ElastiCache, RDS
from diagrams.aws.network import ELB
from diagrams.aws.network import Route53

with Diagram("Clustered Web Services", show=False):
    dns = Route53("dns")
    lb = ELB("lb")

    with Cluster("Services"):
        svc_group = [ECS("web1"),
                     ECS("web2"),
                     ECS("web3")]

    with Cluster("DB Cluster"):
        db_primary = RDS("userdb")
        db_primary - [RDS("userdb ro")]

    memcached = ElastiCache("memcached")

    dns >> lb >> svc_group
    svc_group >> db_primary
    svc_group >> memcached
```



What does it do?

- Draw the cloud system architecture in Python code.
- Diagrams can also be rendered directly inside the Jupyter Notebooks.
- No design tools are needed.
- Supports the following providers: AWS, Azure, GCP, Kubernetes, Oracle Cloud, etc.

Netflix Tech Stack - Part 1 (CI/CD Pipeline)



Planning: Netflix Engineering uses JIRA for planning and Confluence for documentation.

Coding: Java is the primary programming language for the backend service, while other languages are used for different use cases.

Build: Gradle is mainly used for building, and Gradle plugins are built to support various use cases.

Packaging: Package and dependencies are packed into an Amazon Machine Image (AMI) for release.

Testing: Testing emphasizes the production culture's focus on building chaos tools.

Deployment: Netflix uses its self-built Spinnaker for canary rollout deployment.

Monitoring: The monitoring metrics are centralized in Atlas, and Kayenta is used to detect anomalies.

Incident report: Incidents are dispatched according to priority, and PagerDuty is used for incident handling.

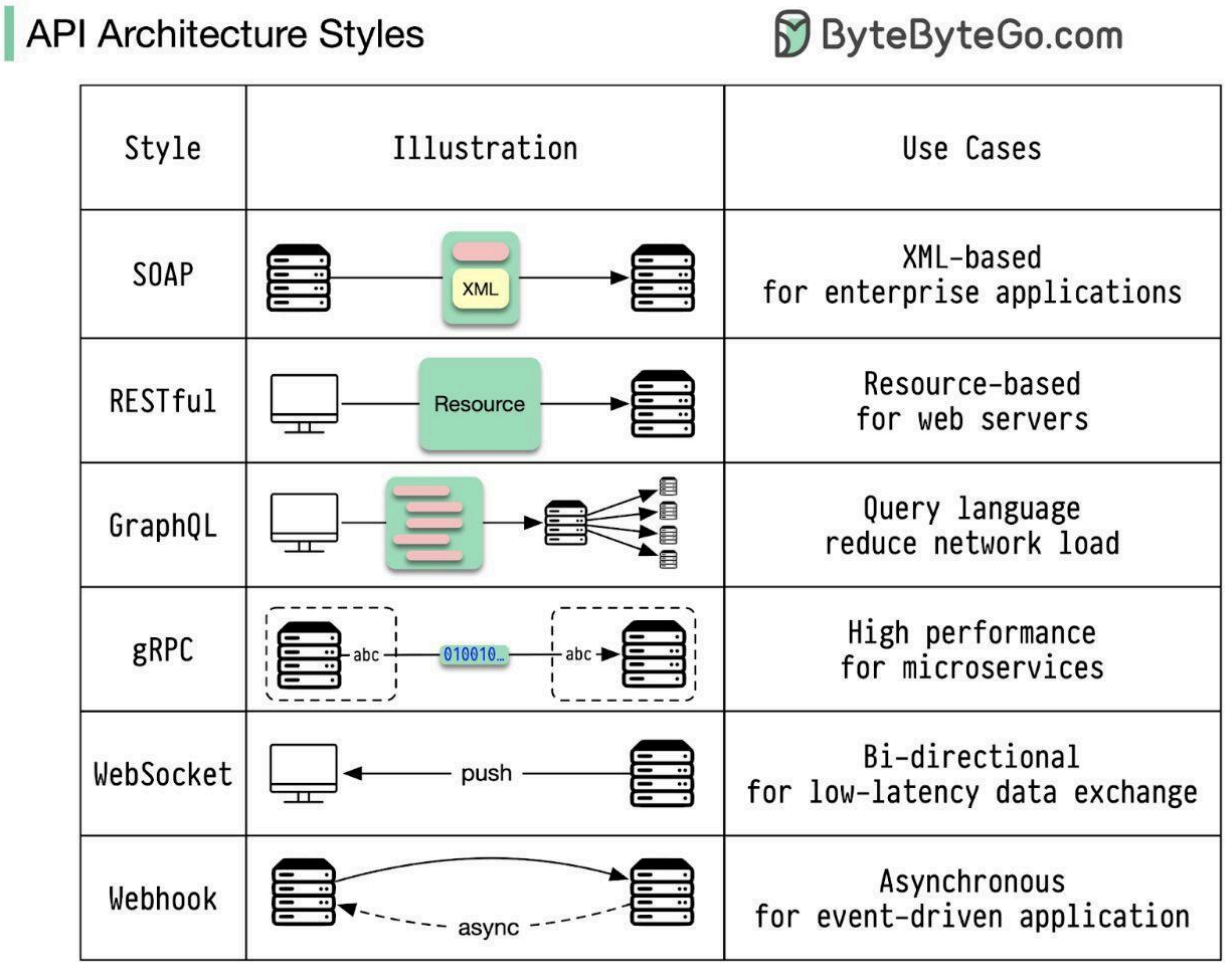
18 Key Design Patterns Every Developer Should Know

Patterns are reusable solutions to common design problems, resulting in a smoother, more efficient development process. They serve as blueprints for building better software structures. These are some of the most popular patterns:

- Abstract Factory: Family Creator - Makes groups of related items.

- Builder: Lego Master - Builds objects step by step, keeping creation and appearance separate.
- Prototype: Clone Maker - Creates copies of fully prepared examples.
- Singleton: One and Only - A special class with just one instance.
- Adapter: Universal Plug - Connects things with different interfaces.
- Bridge: Function Connector - Links how an object works to what it does.
- Composite: Tree Builder - Forms tree-like structures of simple and complex parts.
- Decorator: Customizer - Adds features to objects without changing their core.
- Facade: One-Stop-Shop - Represents a whole system with a single, simplified interface.
- Flyweight: Space Saver - Shares small, reusable items efficiently.
- Proxy: Stand-In Actor - Represents another object, controlling access or actions.
- Chain of Responsibility: Request Relay - Passes a request through a chain of objects until handled.
- Command: Task Wrapper - Turns a request into an object, ready for action.
- Iterator: Collection Explorer - Accesses elements in a collection one by one.
- Mediator: Communication Hub - Simplifies interactions between different classes.
- Memento: Time Capsule - Captures and restores an object's state.
- Observer: News Broadcaster - Notifies classes about changes in other objects.
- Visitor: Skillful Guest - Adds new operations to a class without altering it.

How many API architecture styles do you know?



Architecture styles define how different components of an application programming interface (API) interact with one another. As a result, they ensure efficiency, reliability, and ease of integration with other systems by providing a standard approach to designing and building APIs. Here are the most used styles:

- SOAP:
  Mature, comprehensive, XML-based
  Best for enterprise applications

- RESTful:
  Popular, easy-to-implement, HTTP methods
  Ideal for web services

- GraphQL:
  Query language, request specific data
  Reduces network overhead, faster responses

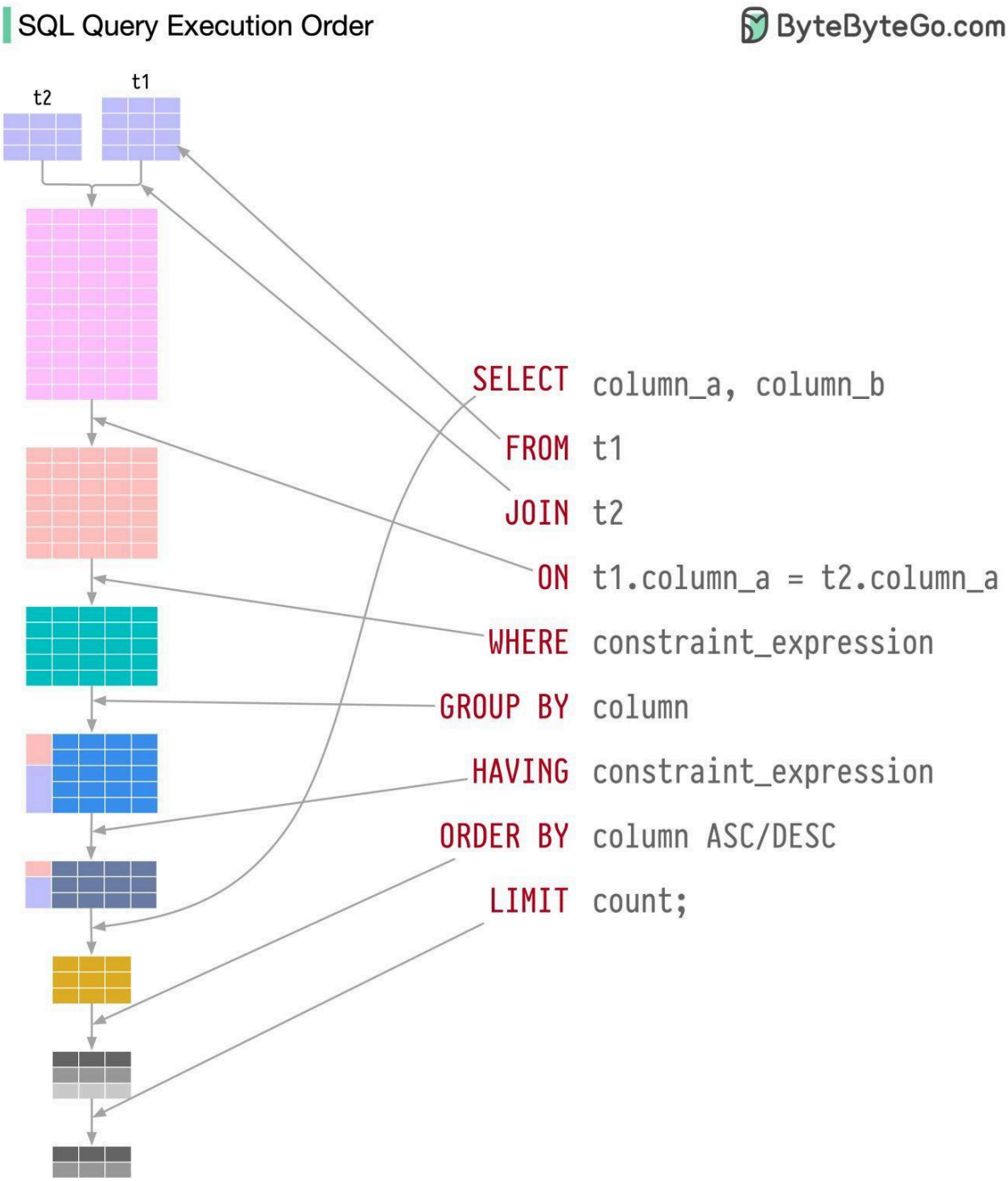
- gRPC:
  Modern, high-performance, Protocol Buffers
  Suitable for microservices architectures

- WebSocket:
  Real-time, bidirectional, persistent connections
  Perfect for low-latency data exchange

- Webhook:
  Event-driven, HTTP callbacks, asynchronous
  Notifies systems when events occur

Over to you: Are there any other famous styles we missed?

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

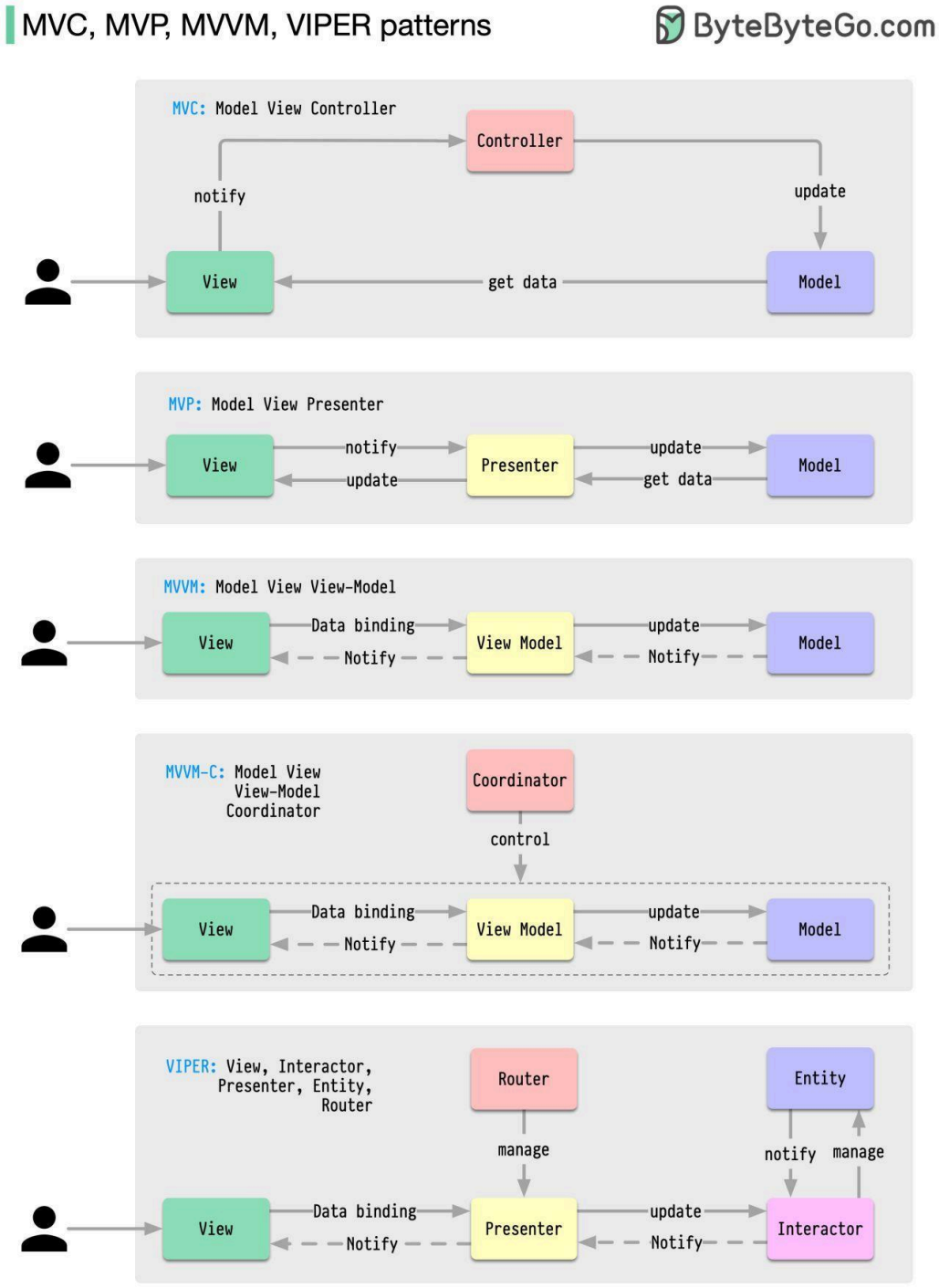
- Optimizing the internal representation and creating an execution plan that utilizes index information
- Executing the plan and returning the results

The execution of SQL is highly complex and involves many considerations, such as:

- The use of indexes and caches
- The order of table joins
- Concurrency control
- Transaction management

Over to you: what is your favorite SQL statement?

What distinguishes MVC, MVP, MVVM, MVVM-C, and VIPER architecture patterns from each other?

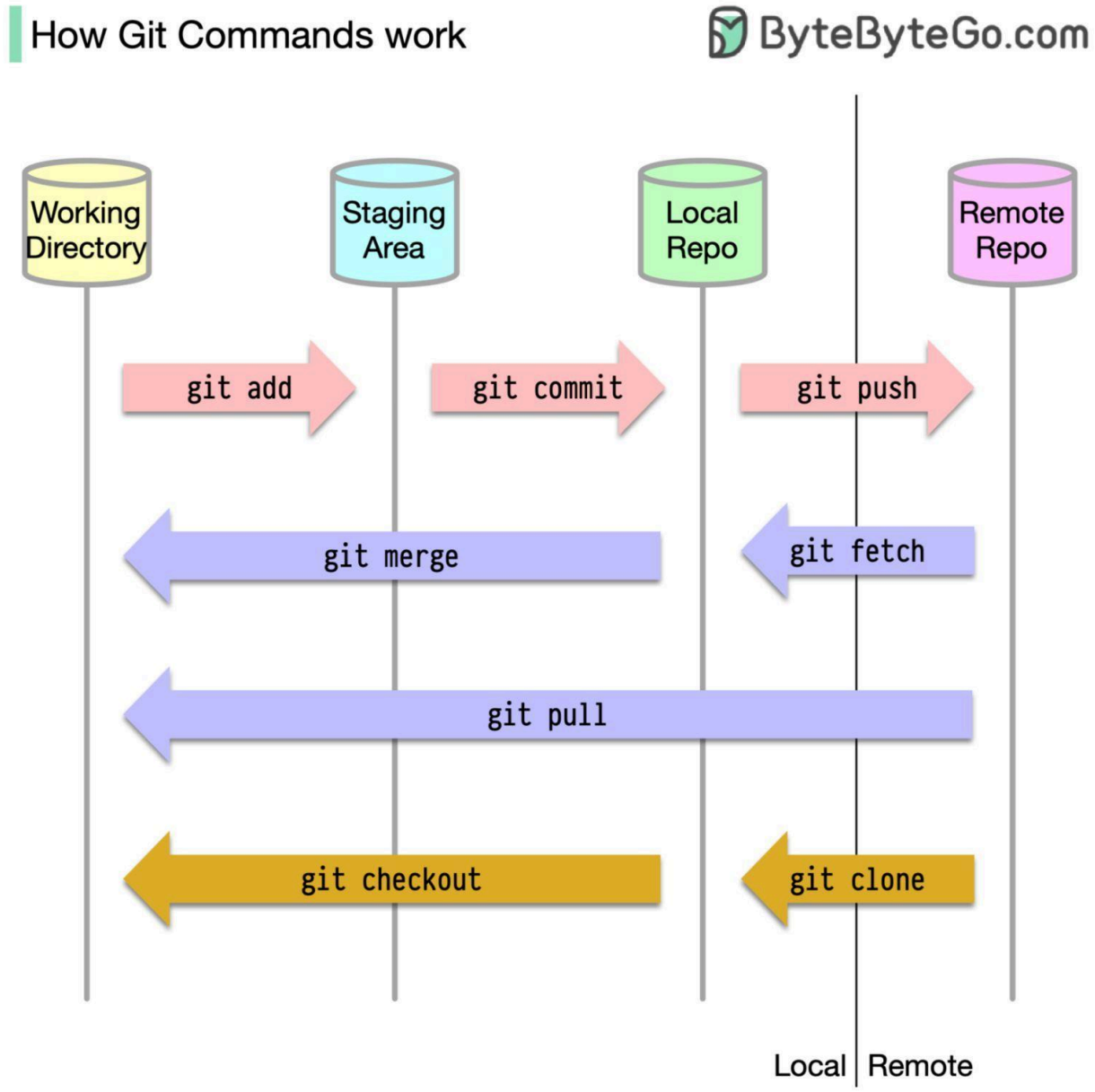


These architecture patterns are among the most commonly used in app development, whether on iOS or Android platforms. Developers have introduced them to overcome the limitations of earlier patterns. So, how do they differ?

- 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

Over to you: keep in mind that this is not an exhaustive list of architecture patterns. Other notable patterns include Flux and Redux. How do they compare to the ones mentioned here?

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:

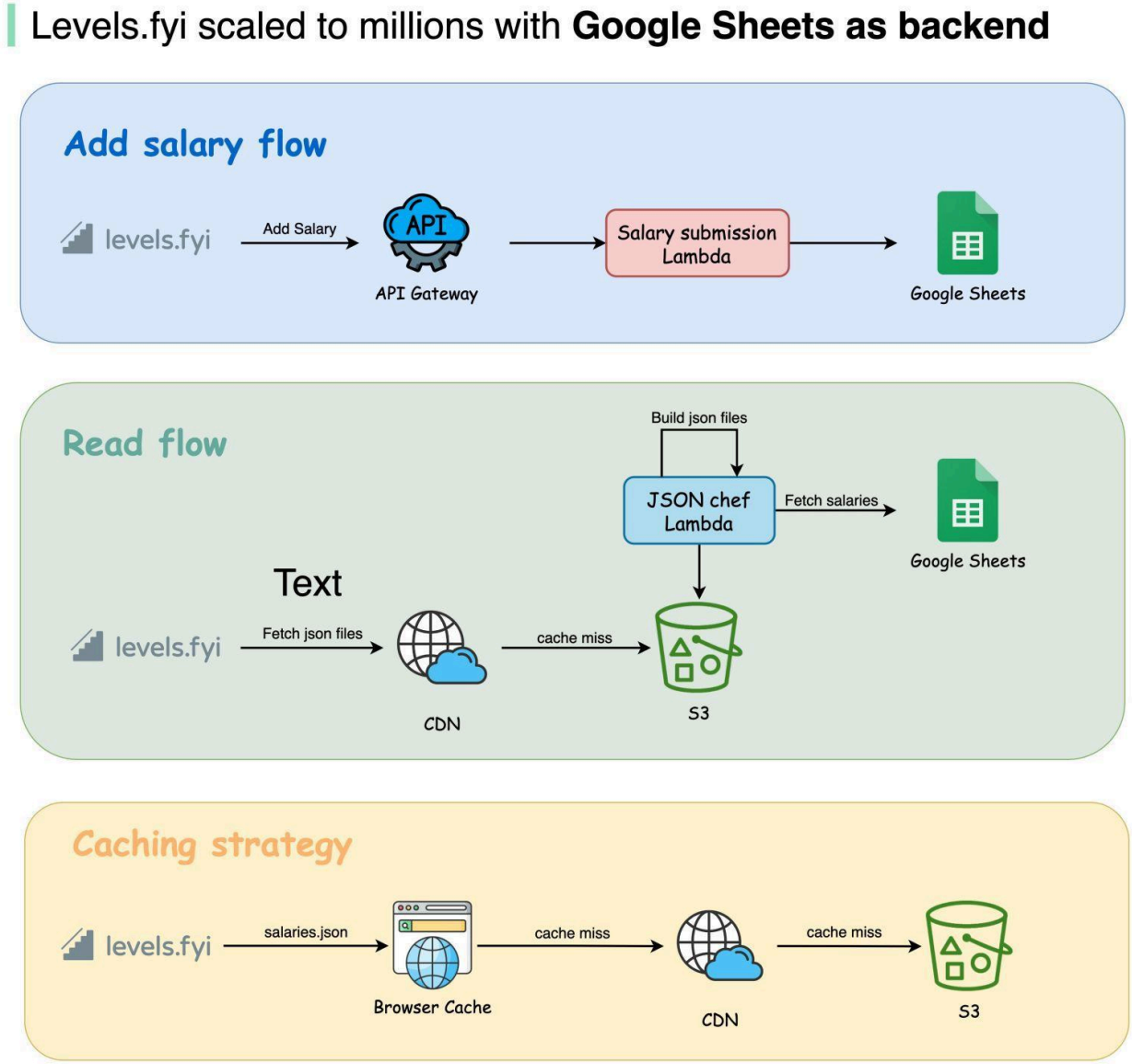
- Working directory: where we edit files
- Staging area: a temporary location where files are kept for the next commit

- 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.

I read something unbelievable today: Levels. fyi scaled to millions of users using Google Sheets as a backend!



Source: https://www.levels.fyi/blog/scaling-to-millions-with-google-sheets.html

They started off on Google Forms and Sheets, which helped them reach millions of monthly active users before switching to a proper backend.

To be fair, they do use serverless computing, but using Google Sheets as the database is an interesting choice.

Why do they use Google Sheets as a backend? Using their own words: "It seems like a pretty

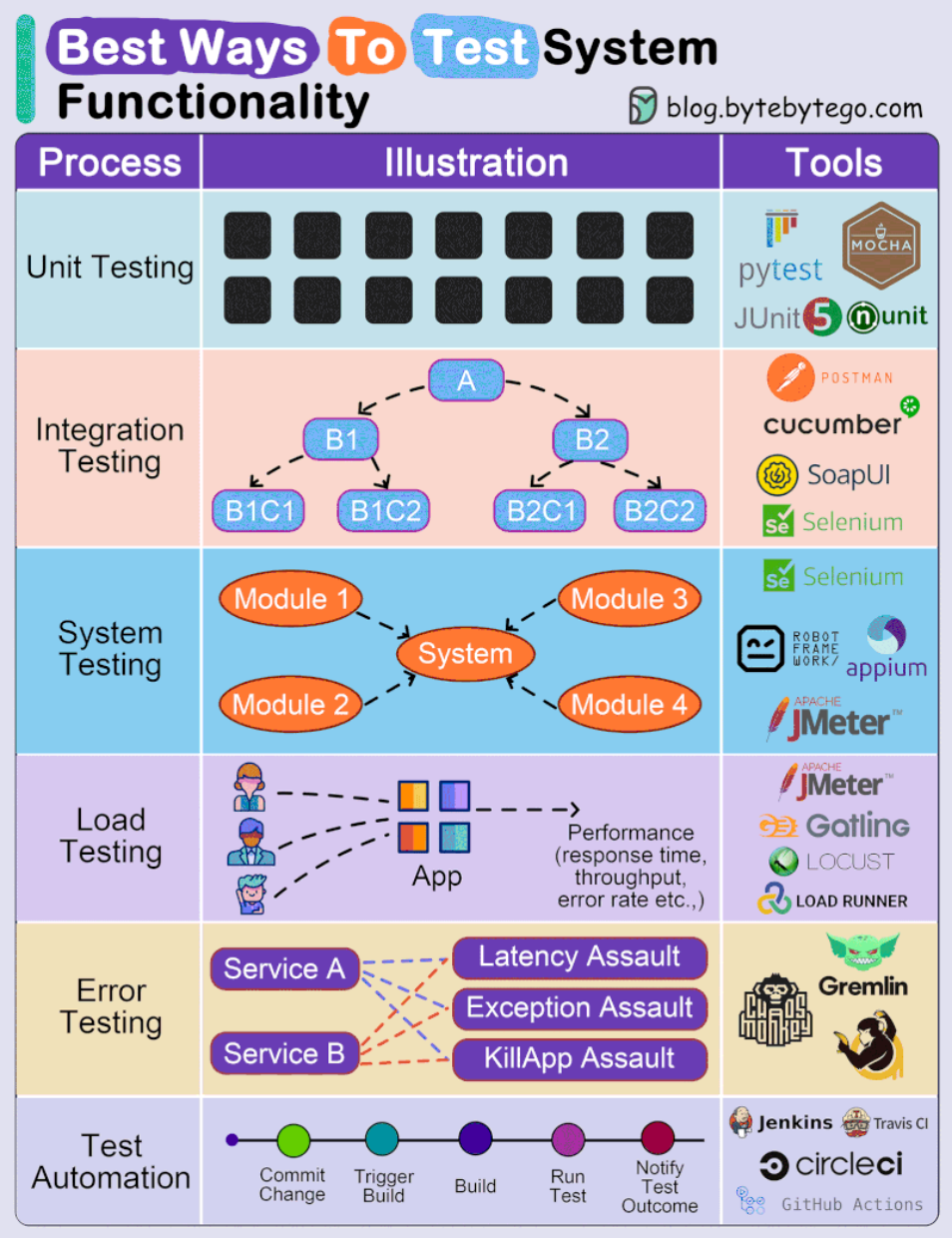
counterintuitive idea for a site with our traffic volume to not have a backend or any fancy infrastructure, but our philosophy to building products has always been, start simple and iterate. This allows us to move fast and focus on what’s important".

What are your thoughts? The link to the original article is embedded at the bottom of the diagram.

Best ways to test system functionality

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.

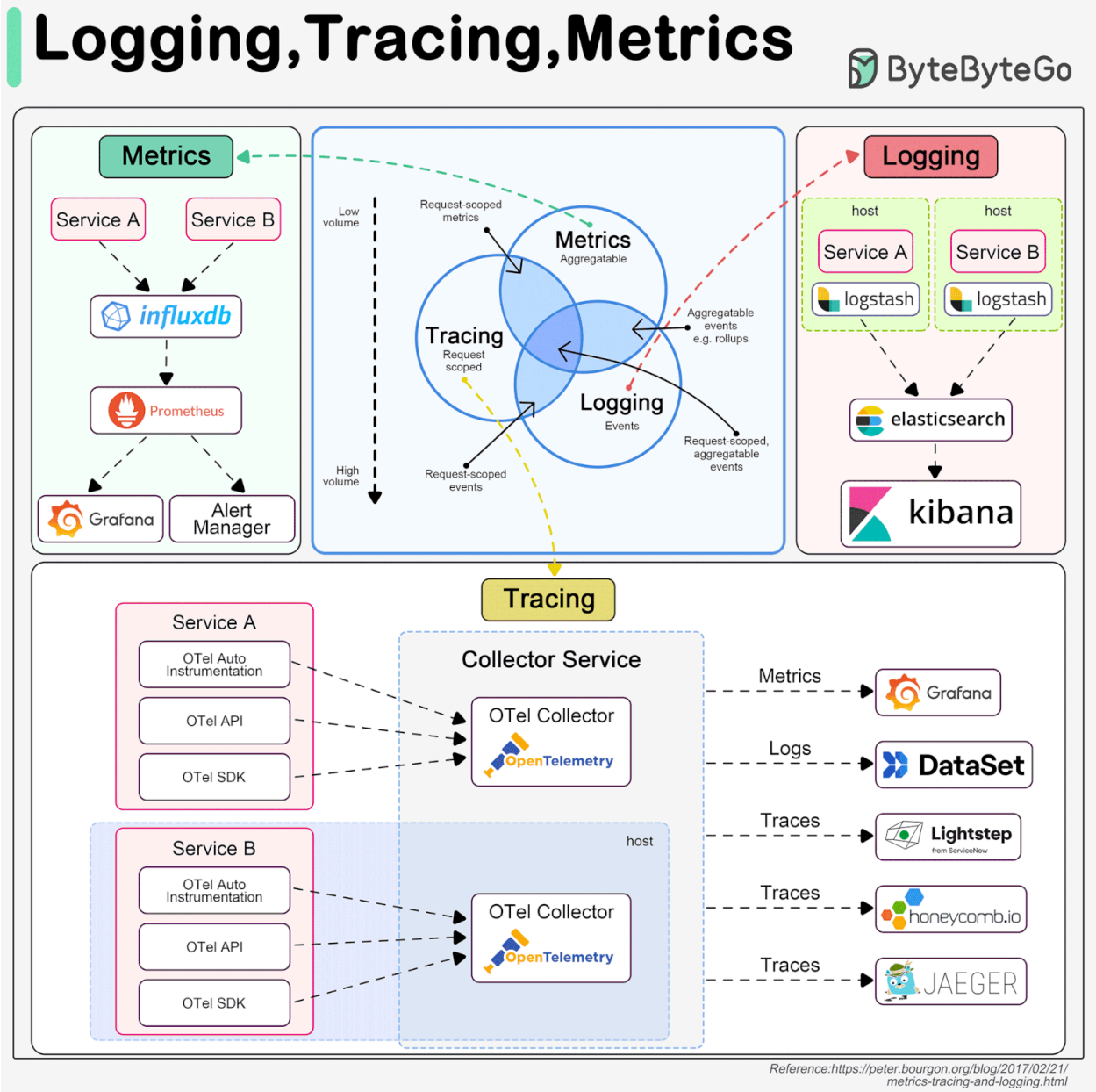
5. Error Testing: Evaluates how the software handles invalid inputs and error conditions.

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?

Logging, tracing and metrics are 3 pillars of system observability

The diagram below shows their definitions and typical architectures.



- Logging
  Logging records discrete events in the system. For example, we can record an incoming request or a visit to databases as events. It has the highest volume. ELK (Elastic-Logstash-Kibana) stack is often used to build a log analysis platform. We often define a standardized logging format for different teams to implement, so that we can leverage keywords when searching among massive amounts of logs.

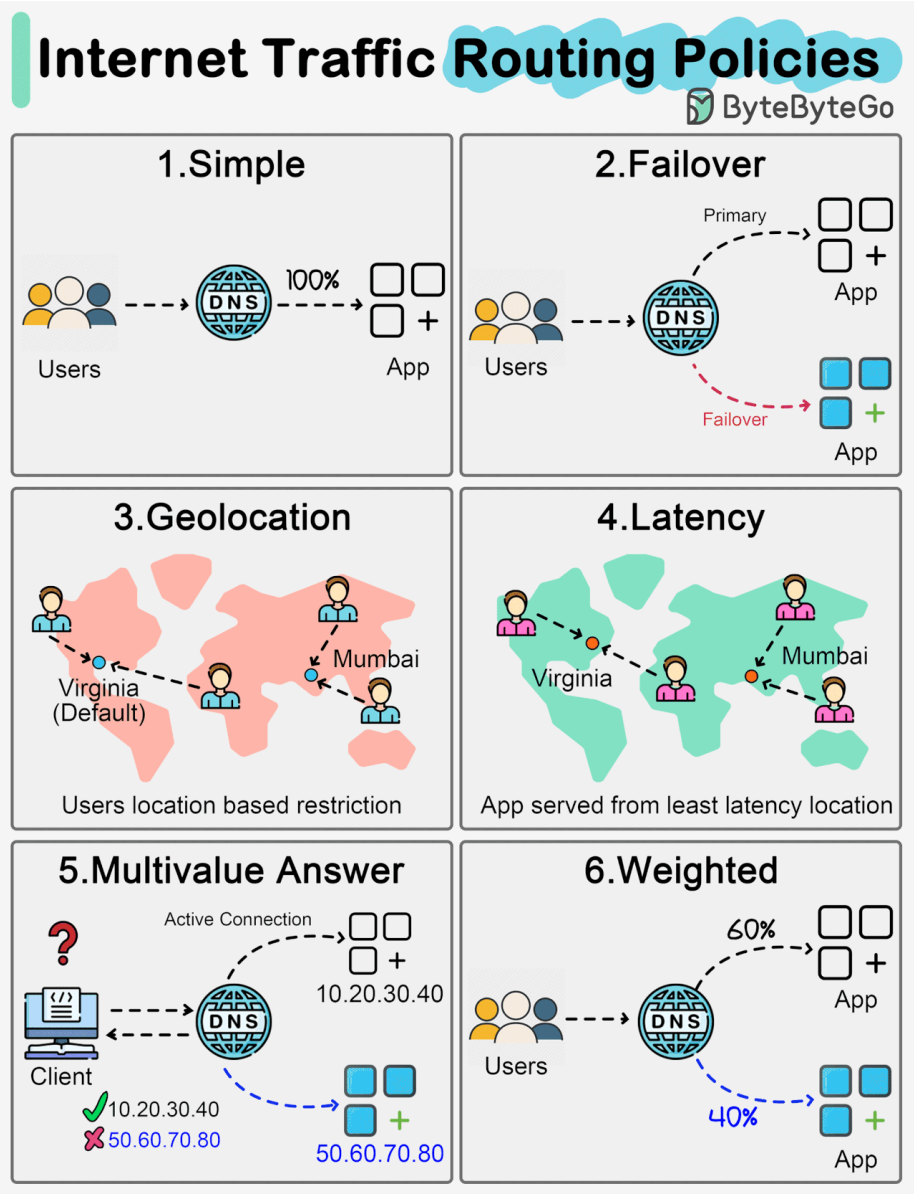
- Tracing
  Tracing is usually request-scoped. For example, a user request goes through the API gateway, load balancer, service A, service B, and database, which can be visualized in the tracing systems. This is useful when we are trying to identify the bottlenecks in the system. We use OpenTelemetry to showcase the typical architecture, which unifies the 3 pillars in a single framework.

- Metrics
  Metrics are usually aggregatable information from the system. For example, service QPS, API responsiveness, service latency, etc. The raw data is recorded in time-series databases like InfluxDB. Prometheus pulls the data and transforms the data based on pre-defined alerting rules. Then the data is sent to Grafana for display or to the alert manager which then sends out email, SMS, or Slack notifications or alerts.

Over to you: Which tools have you used for system monitoring?

Internet Traffic Routing Policies

Internet traffic routing policies (DNS policies) play a crucial role in efficiently managing and directing network traffic. Let's discuss the different types of policies.



1. Simple: Directs all traffic to a single endpoint based on a standard DNS query without any special conditions or requirements.

2. Failover: Routes traffic to a primary endpoint but automatically switches to a secondary endpoint if the primary is unavailable.

3. Geolocation: Distributes traffic based on the geographic location of the requester, aiming to provide localized content or services.

4. Latency: Directs traffic to the endpoint that provides the lowest latency for the requester, enhancing user experience with faster response times.

5. Multivalue Answer: Responds to DNS queries with multiple IP addresses, allowing the client to select an endpoint. However, it should not be considered a replacement for a load balancer.

6. Weighted Routing Policy: Distributes traffic across multiple endpoints with assigned weights, allowing for proportional traffic distribution based on these weights.

Over to you: Which DNS policy do you find most relevant to your network management needs?

Subjects that should be mandatory in schools

In the age of AI, what subjects should be taught in schools?

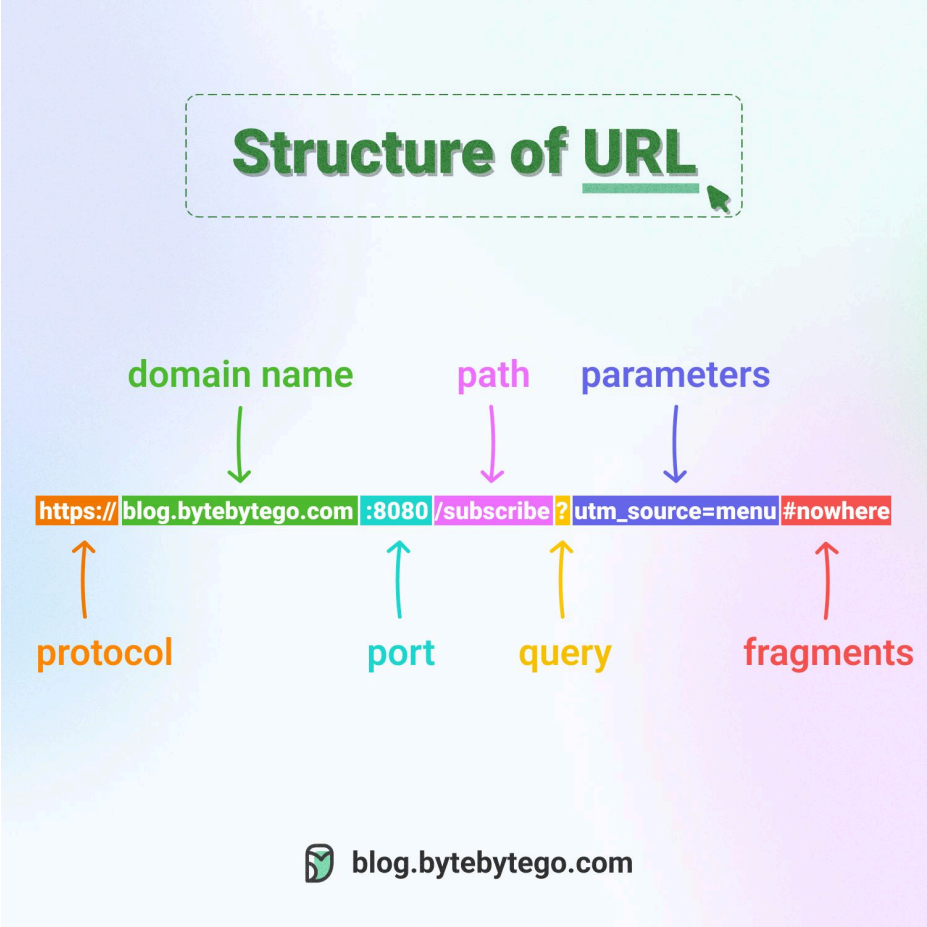
An interesting list of subjects that should be mandatory in schools by startup_rules.

While academics are essential, it's crucial to acknowledge that many elements in this diagram would have been beneficial to learn earlier.

Over to you: What else should be on the list? What are the top 3 skills you wish schools would teach?

Do you know all the components of a URL?

Uniform Resource Locator (URL) is a term familiar to most people, as it is used to locate resources on the internet. When you type a URL into a web browser's address bar, you are accessing a "resource", not just a webpage.

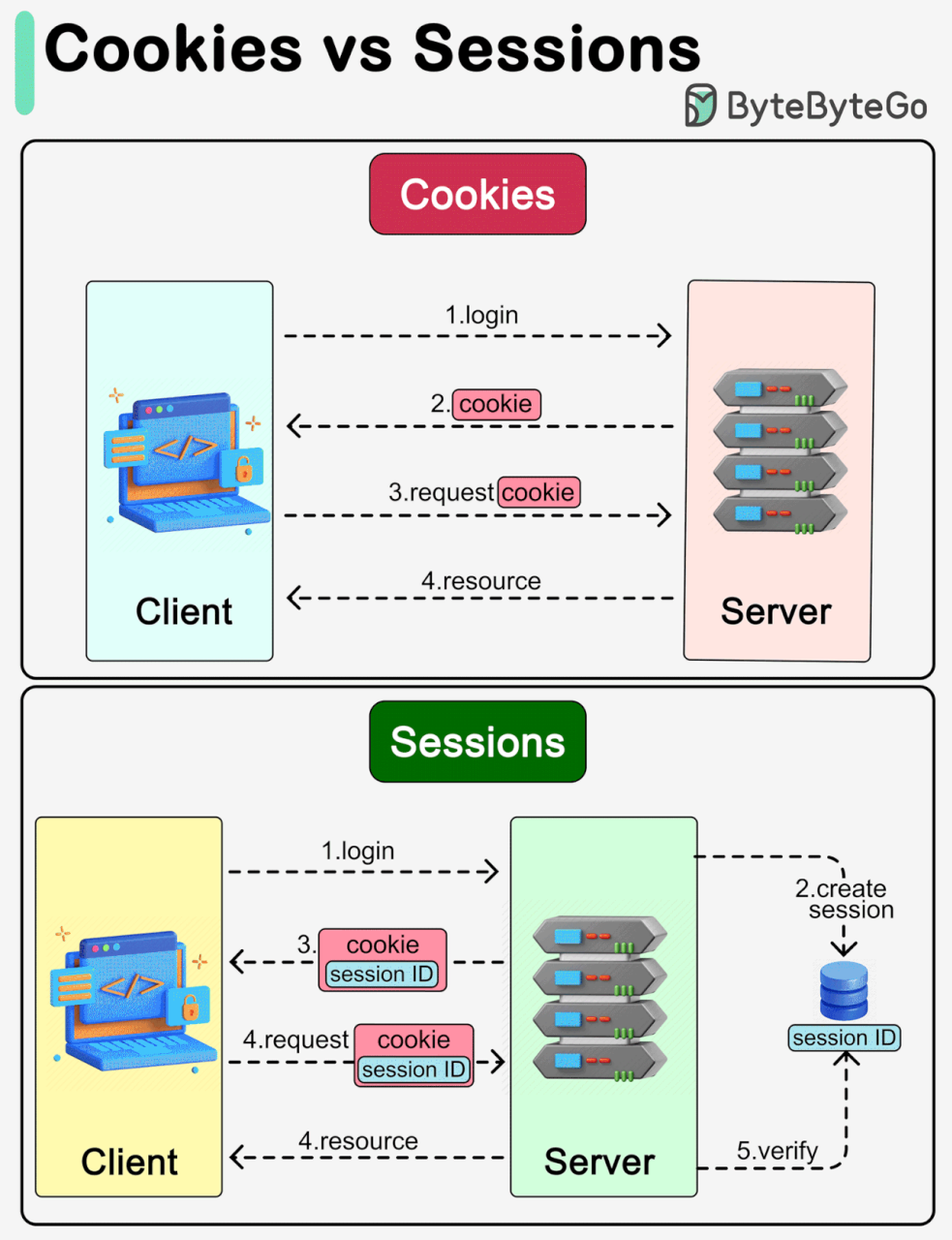


URLs comprise several components:

- The protocol or scheme, such as http, https, and ftp.
- The domain name and port, separated by a period (.)
- The path to the resource, separated by a slash (/)
- The parameters, which start with a question mark (?) and consist of key-value pairs, such as a=b&c=d.
- The fragment or anchor, indicated by a pound sign (#), which is used to bookmark a specific section of the resource.

What are the differences between cookies and sessions?

The diagram below shows how they work.



Cookies and sessions are both used to carry user information over HTTP requests, including user login status, user permissions, etc.

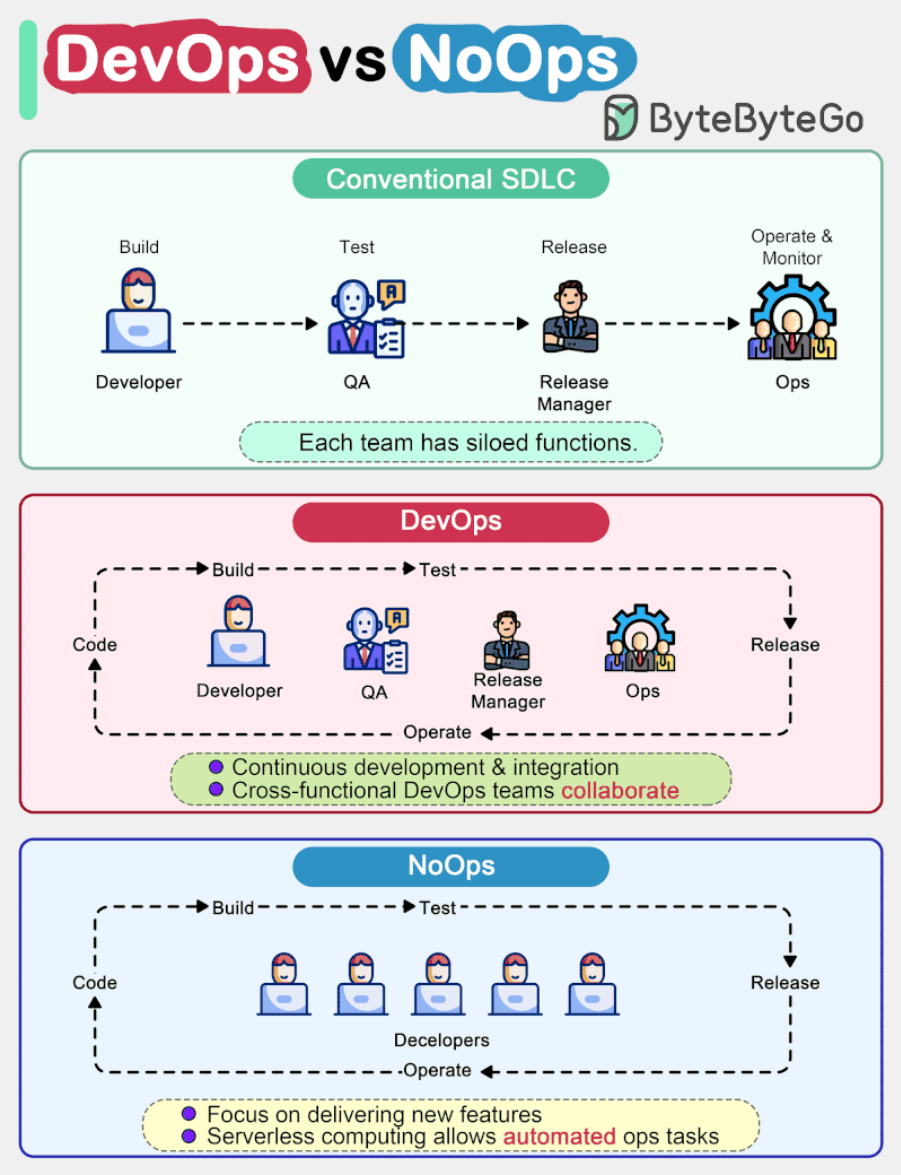
- Cookies
  Cookies typically have size limits (4KB). They carry small pieces of information and are stored on the users' devices. Cookies are sent with each subsequent user request. Users

can choose to ban cookies in their browsers.

- Sessions
  Unlike cookies, sessions are created and stored on the server side. There is usually a unique session ID generated on the server, which is attached to a specific user session. This session ID is returned to the client side in a cookie. Sessions can hold larger amounts of data. Since the session data is not directly accessed by the client, the session offers more security.

How do DevOps, NoOps change the software development lifecycle (SDLC)?

The diagram below compares traditional SDLC, DevOps and NoOps.



In a traditional software development, code, build, test, release and monitoring are siloed functions. Each stage works independently and hands over to the next stage.

DevOps, on the other hand, encourages continuous development and collaboration between developers and operations. This shortens the overall life cycle and provides continuous software delivery.

NoOps is a newer concept with the development of serverless computing. Since we can architect the system using FaaS (Function-as-a-Service) and BaaS (Backend-as-a-Service), the cloud service providers can take care of most operations tasks. The developers can focus on feature development and automate operations tasks.

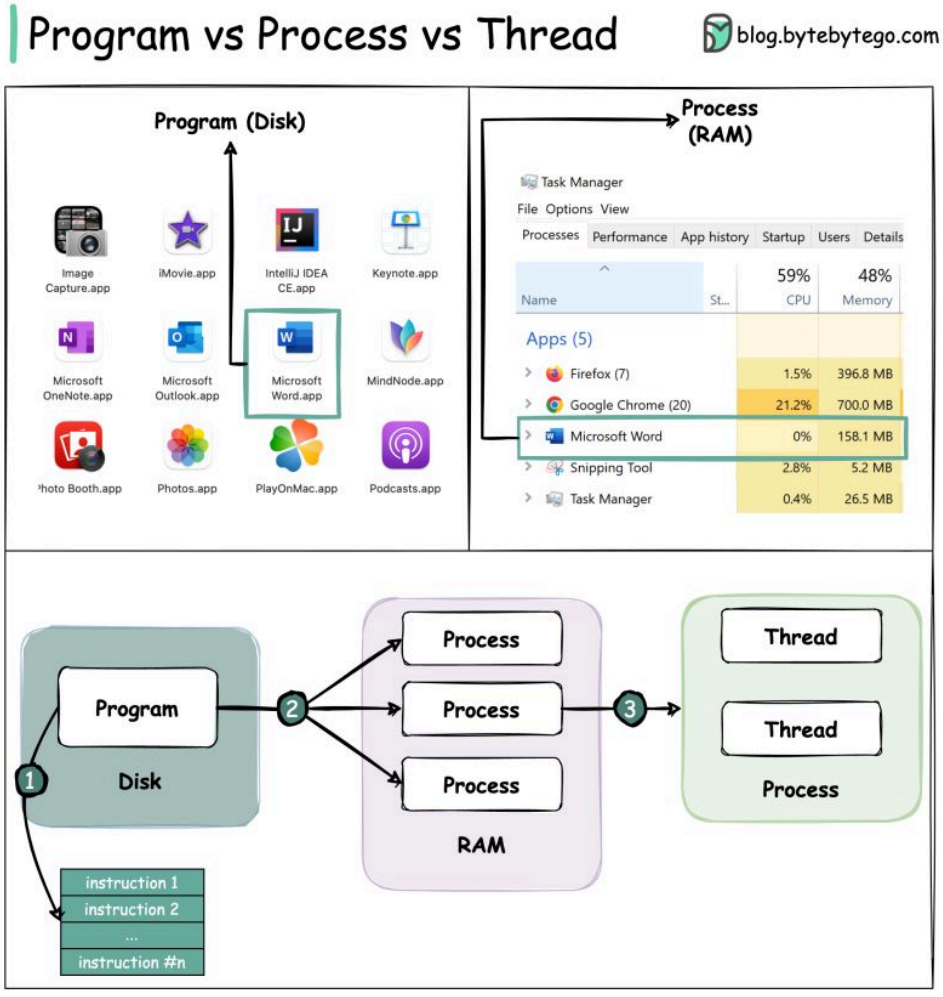
NoOps is a pragmatic and effective methodology for startups or smaller-scale applications, which moves shortens the SDLC even more than DevOps.

Popular interview question: What is the difference between Process and Thread?

To better understand this question, let's first take a look at what a Program is. A Program is an executable file containing a set of instructions and passively stored on disk. One program can have multiple processes. For example, the Chrome browser creates a different process for every single tab.

A Process means a program is in execution. When a program is loaded into the memory and becomes active, the program becomes a process. The process requires some essential resources such as registers, program counter, and stack.

A Thread is the smallest unit of execution within a process.



The following process explains the relationship between program, process, and thread.

1. The program contains a set of instructions.

2. The program is loaded into memory. It becomes one or more running processes.
3. When a process starts, it is assigned memory and resources. A process can have one or more threads. For example, in the Microsoft Word app, a thread might be responsible for spelling checking and the other thread for inserting text into the doc.

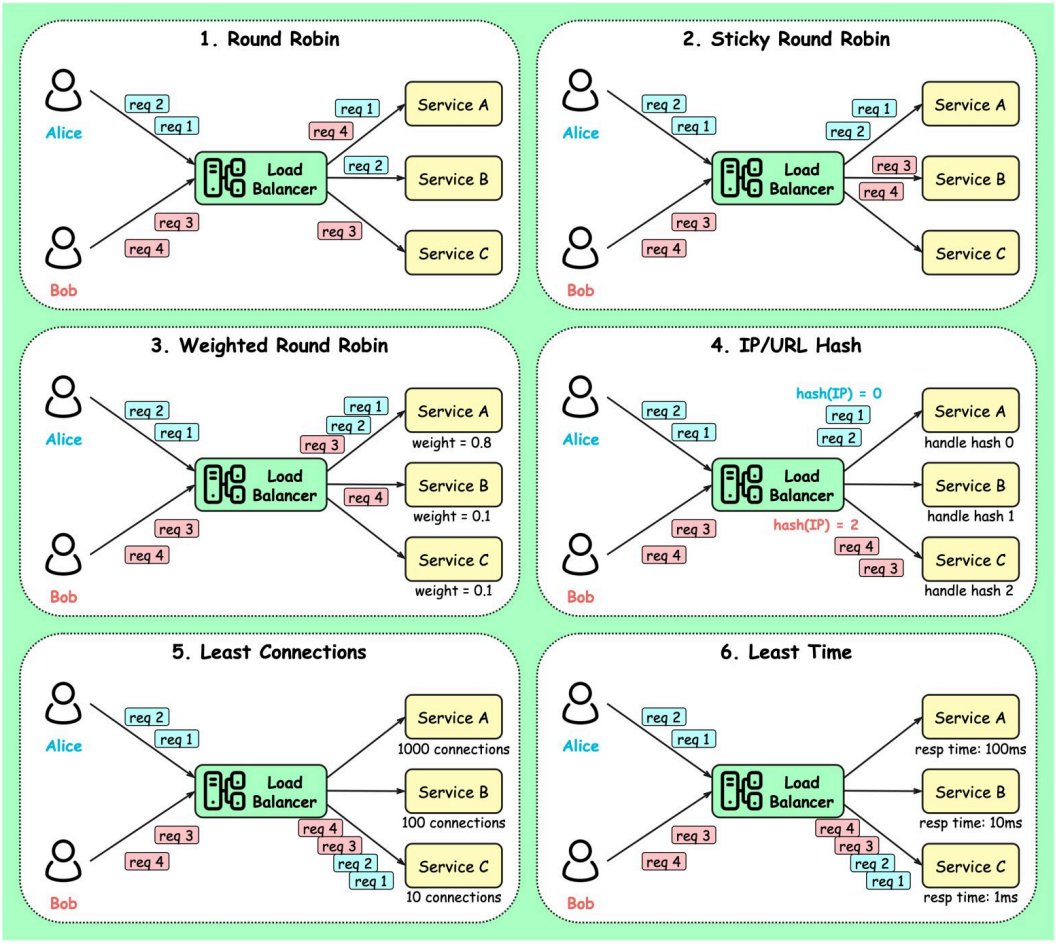
Main differences between process and thread:

- Processes are usually independent, while threads exist as subsets of a process.
- Each process has its own memory space. Threads that belong to the same process share the same memory.
- A process is a heavyweight operation. It takes more time to create and terminate.
- Context switching is more expensive between processes.
- Inter-thread communication is faster for threads.

Over to you:

1. Some programming languages support coroutine. What is the difference between coroutine and thread?
2. How to list running processes in Linux?

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.

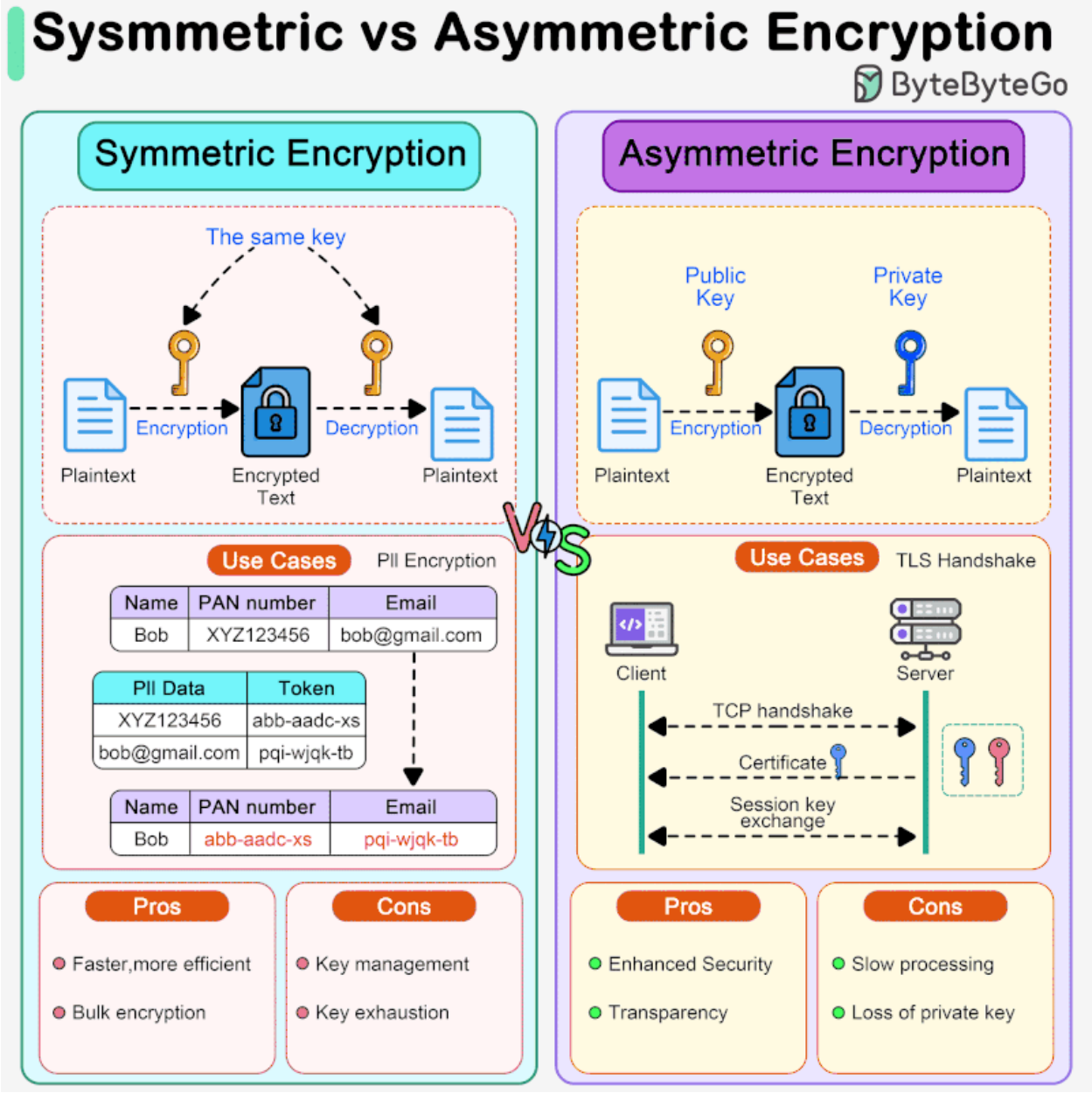
Dynamic Algorithms

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.

Symmetric encryption vs asymmetric encryption

Symmetric encryption and asymmetric encryption are two types of cryptographic techniques used to secure data and communications, but they differ in their methods of encryption and decryption.



- In symmetric encryption, a single key is used for both encryption and decryption of data. It is faster and can be applied to bulk data encryption/decryption. For example, we can use it to encrypt massive amounts of PII (Personally Identifiable Information) data. It poses challenges in key management because the sender and receiver share the same key.

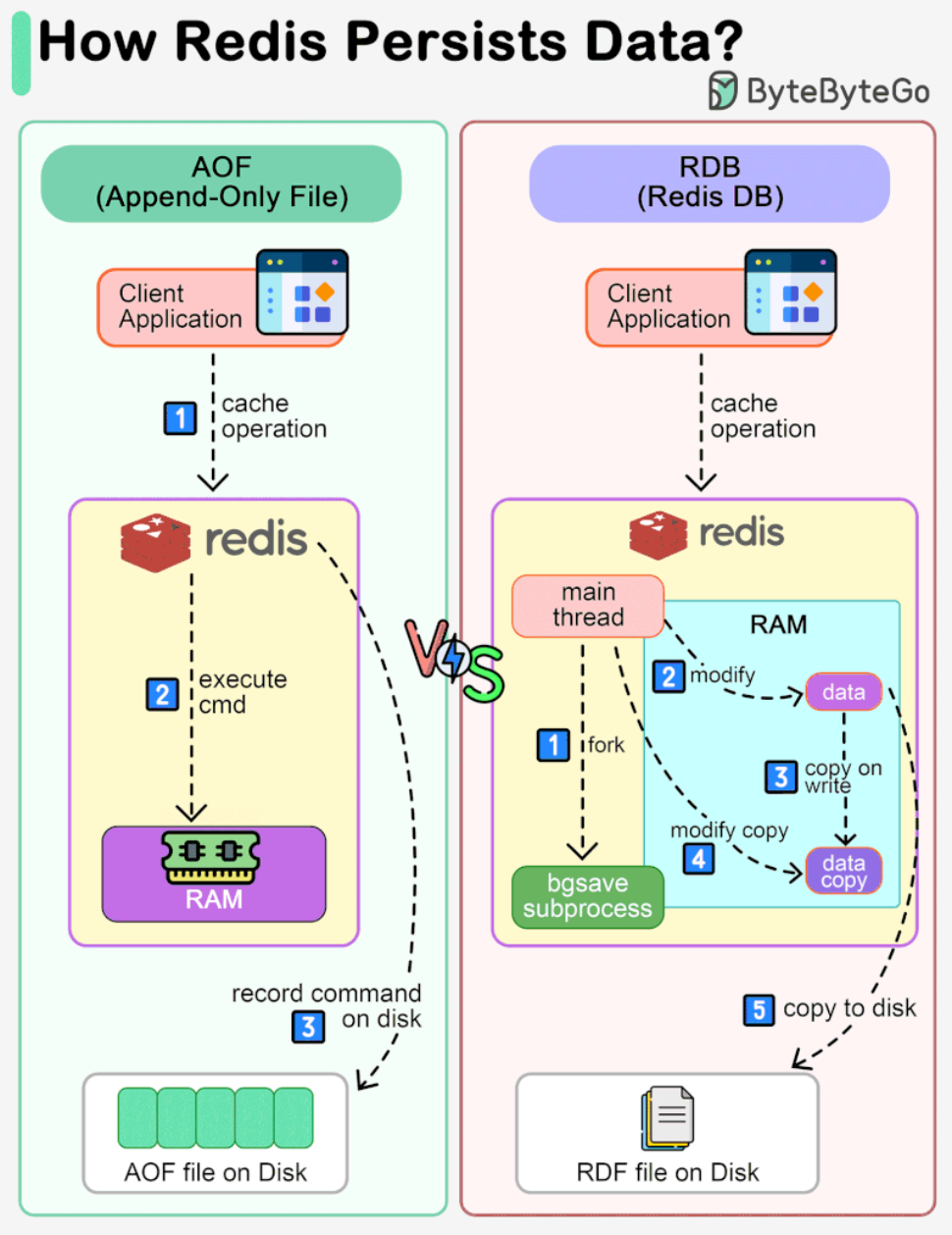
- Asymmetric encryption uses a pair of keys: a public key and a private key. The public key is freely distributed and used to encrypt data, while the private key is kept secret and used to decrypt the data. It is more secure than symmetric encryption because the private key is never shared. However, asymmetric encryption is slower because of the complexity of key generation and maths computations. For example, HTTPS uses asymmetric encryption to exchange session keys during TLS handshake, and after that, HTTPS uses symmetric encryption for subsequent communications.

How does Redis persist data?

Redis is an in-memory database. If the server goes down, the data will be lost.

The diagram below shows two ways to persist Redis data on disk:

1. AOF (Append-Only File)
2. RDB (Redis Database)



Note that data persistence is not performed on the critical path and doesn't block the write process in Redis.

- AOF
  Unlike a write-ahead log, the Redis AOF log is a write-after log. Redis executes commands to modify the data in memory first and then writes it to the log file. AOF log records the commands instead of the data. The event-based design simplifies data recovery. Additionally, AOF records commands after the command has been executed in memory, so it does not block the current write operation.

- RDB
  The restriction of AOF is that it persists commands instead of data. When we use the AOF log for recovery, the whole log must be scanned. When the size of the log is large, Redis takes a long time to recover. So Redis provides another way to persist data - RDB.

  RDB records snapshots of data at specific points in time. When the server needs to be recovered, the data snapshot can be directly loaded into memory for fast recovery.

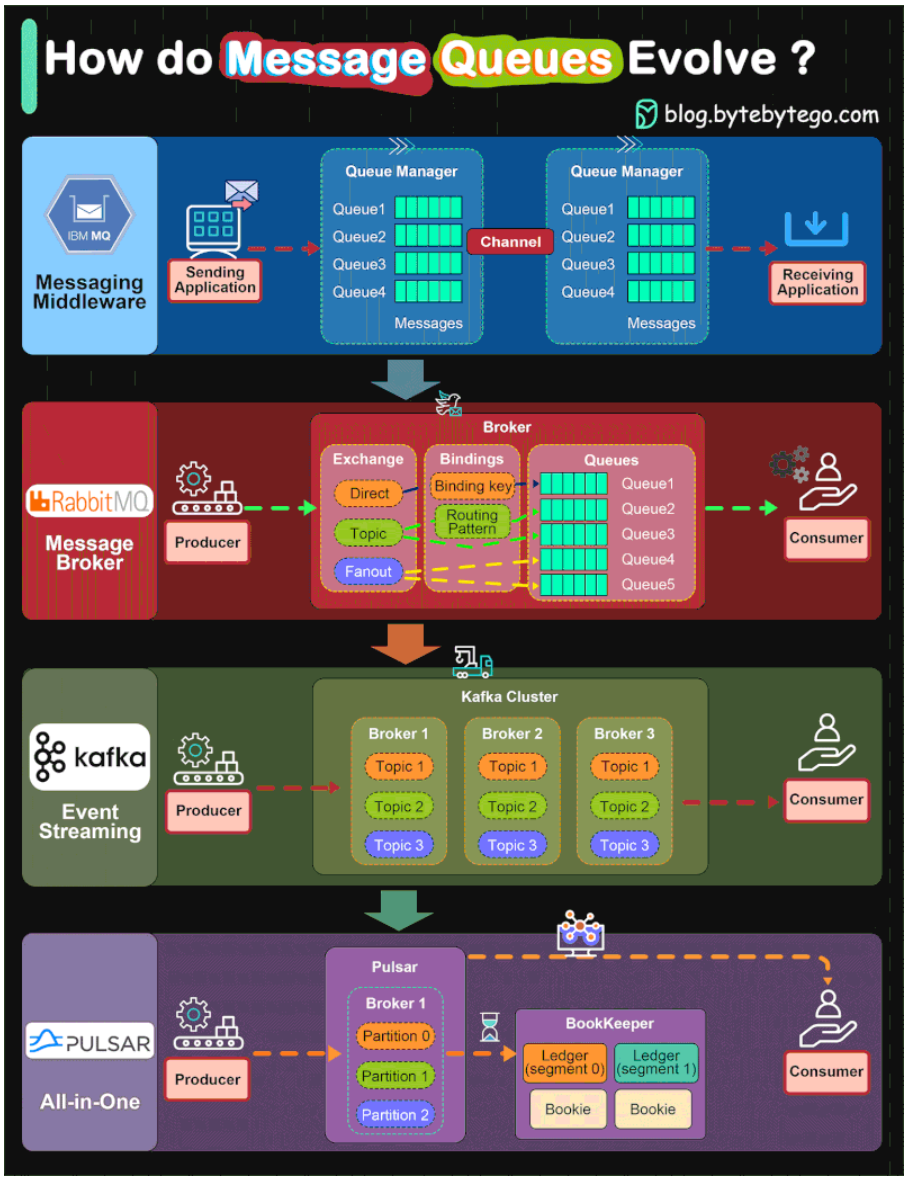
  Step 1: The main thread forks the‘ bgsave’ sub-process, which shares all the in-memory data of the main thread. ‘bgsave’ reads the data from the main thread and writes it to the RDB file.

  Steps 2 and 3: If the main thread modifies data, a copy of the data is created.

  Steps 4 and 5: The main thread then operates on the data copy. Meanwhile ‘bgsave’ sub-process continues to write data to the RDB file.

- Mixed
  Usually in production systems, we can choose a mixed approach, where we use RDB to record data snapshots from time to time and use AOF to record the commands since the last snapshot.

IBM MQ -> RabbitMQ -> Kafka ->Pulsar, How do message queue architectures evolve?



- IBM MQ
  IBM MQ was launched in 1993. It was originally called MQSeries and was renamed WebSphere MQ in 2002. It was renamed to IBM MQ in 2014. IBM MQ is a very successful product widely used in the financial sector. Its revenue still reached 1 billion dollars in 2020.

- RabbitMQ
  RabbitMQ architecture differs from IBM MQ and is more similar to Kafka concepts. The producer publishes a message to an exchange with a specified exchange type. It can be direct, topic, or fanout. The exchange then routes the message into the queues based on

different message attributes and the exchange type. The consumers pick up the message accordingly.

- Kafka
  In early 2011, LinkedIn open sourced Kafka, which is a distributed event streaming platform. It was named after Franz Kafka. As the name suggested, Kafka is optimized for writing. It offers a high-throughput, low-latency platform for handling real-time data feeds. It provides a unified event log to enable event streaming and is widely used in internet companies.

  Kafka defines producer, broker, topic, partition, and consumer. Its simplicity and fault tolerance allow it to replace previous products like AMQP-based message queues.

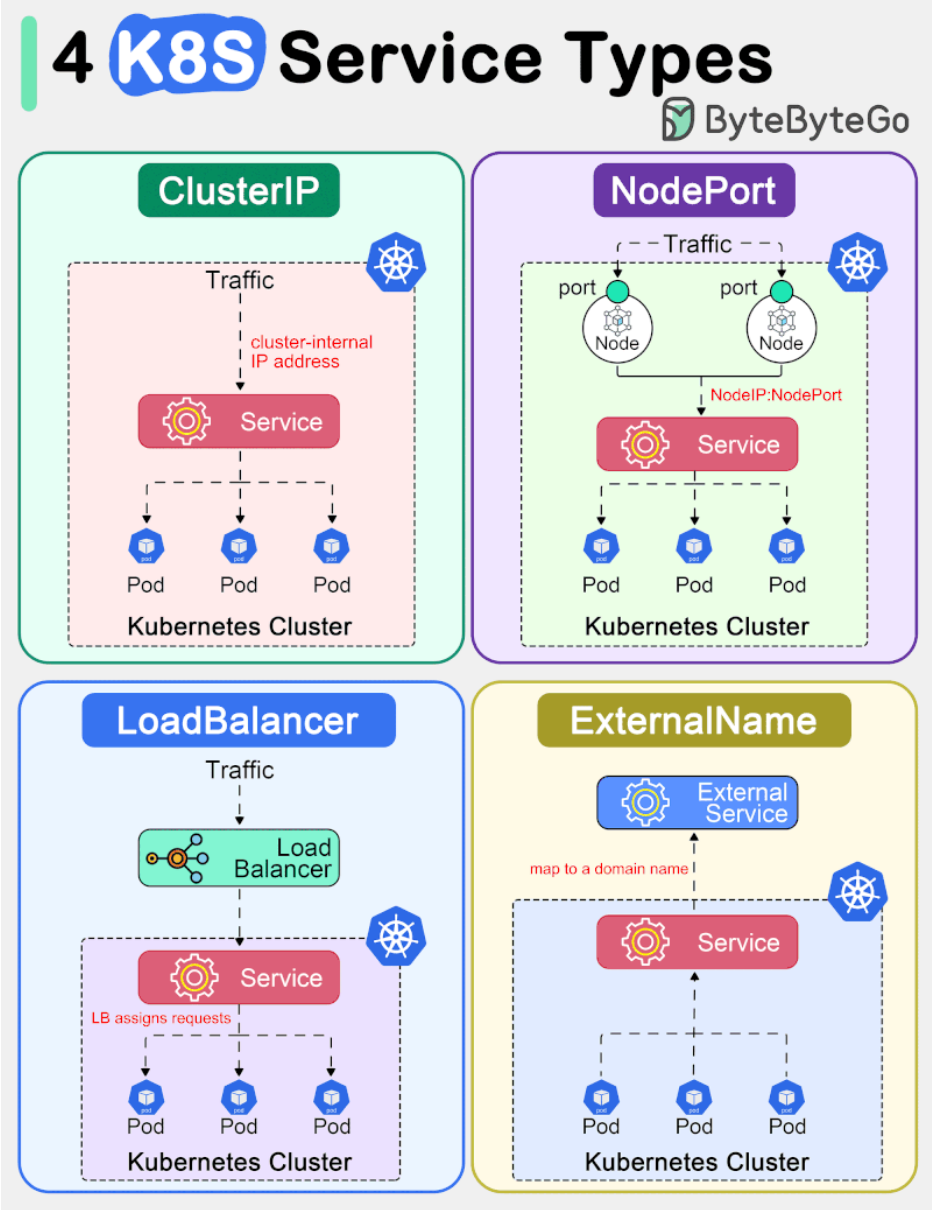
- Pulsar
  Pulsar, developed originally by Yahoo, is an all-in-one messaging and streaming platform. Compared with Kafka, Pulsar incorporates many useful features from other products and supports a wide range of capabilities. Also, Pulsar architecture is more cloud-native, providing better support for cluster scaling and partition migration, etc.

  There are two layers in Pulsar architecture: the serving layer and the persistent layer. Pulsar natively supports tiered storage, where we can leverage cheaper object storage like AWS S3 to persist messages for a longer term.

Over to you: which message queues have you used?

Top 4 Kubernetes Service Types in one diagram

The diagram below shows 4 ways to expose a Service.



In Kubernetes, a Service is a method for exposing a network application in the cluster. We use a Service to make that set of Pods available on the network so that users can interact with it.

There are 4 types of Kubernetes services: ClusterIP, NodePort, LoadBalancer and ExternalName. The “type” property in the Service's specification determines how the service is exposed to the network.

- ClusterIP
  ClusterIP is the default and most common service type. Kubernetes will assign a

cluster-internal IP address to ClusterIP service. This makes the service only reachable within the cluster.

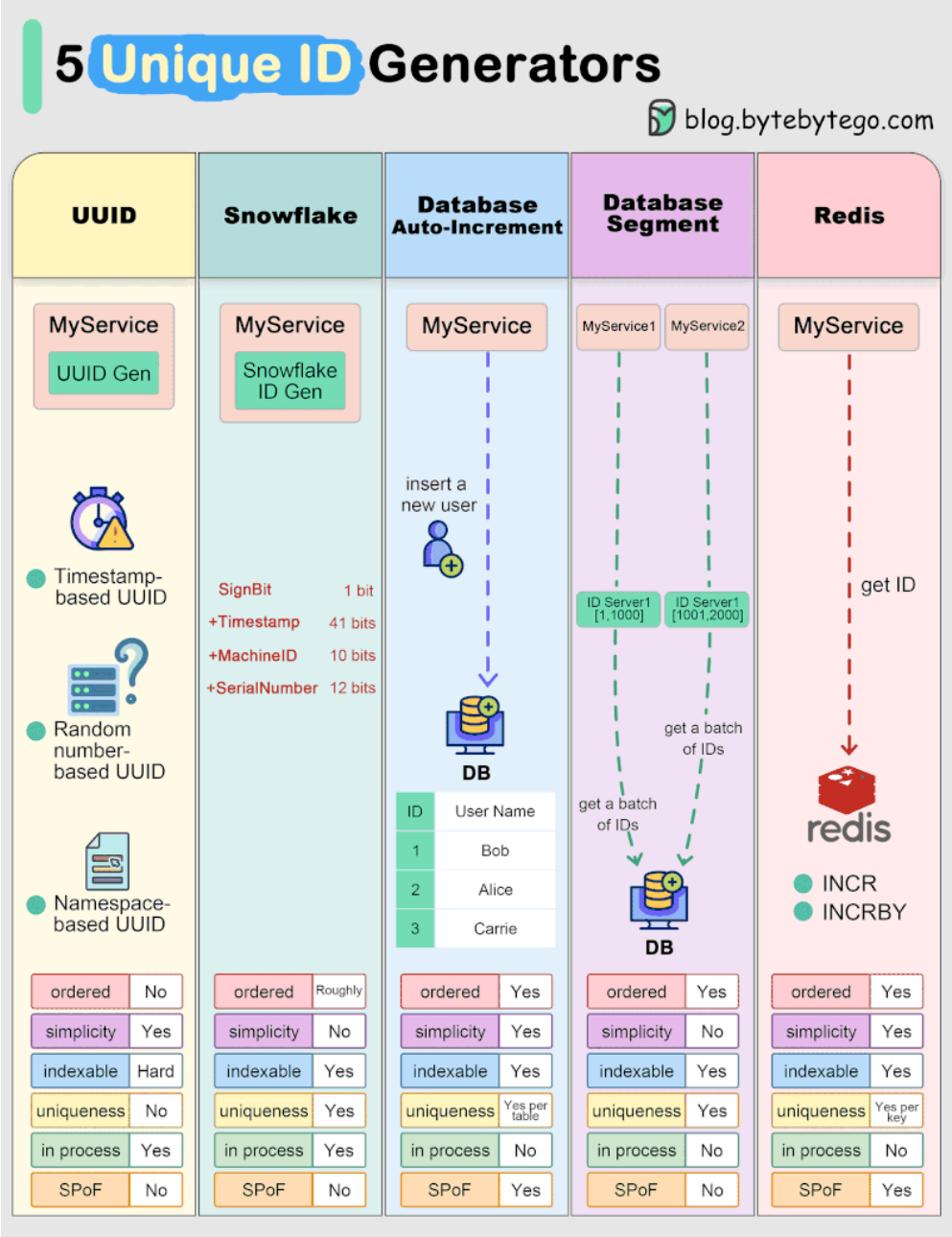
- NodePort
  This exposes the service outside of the cluster by adding a cluster-wide port on top of ClusterIP. We can request the service by NodeIP:NodePort.

- LoadBalancer
  This exposes the Service externally using a cloud provider’s load balancer.

- ExternalName
  This maps a Service to a domain name. This is commonly used to create a service within Kubernetes to represent an external database.

Explaining 5 unique ID generators in distributed systems

The diagram below shows how they work. Each generator has its pros and cons.



1. UUID

   A UUID has 128 bits. It is simple to generate and no need to call another service. However, it is not sequential and inefficient for database indexing. Additionally, UUID doesn't guarantee global uniqueness. We need to be careful with ID conflicts (although the chances are slim.)

2. Snowflake

   Snowflake's ID generation process has multiple components: timestamp, machine ID, and serial number. The first bit is unused to ensure positive IDs. This generator doesn't need to talk to an ID generator via the network, so is fast and scalable.

   Snowflake implementations vary. For example, data center ID can be added to the "MachineID" component to guarantee global uniqueness.

3. DB auto-increment

   Most database products offer auto-increment identity columns. Since this is supported in the database, we can leverage its transaction management to handle concurrent visits to the ID generator. This guarantees uniqueness in one table. However, this involves network communications and may expose sensitive business data to the outside. For example, if we use this as a user ID, our business competitors will have a rough idea of the total number of users registered on our website.

4. DB segment

   An alternative approach is to retrieve IDs from the database in batches and cache them in the ID servers, each ID server handling a segment of IDs. This greatly saves the I/O pressure on the database.

5. Redis

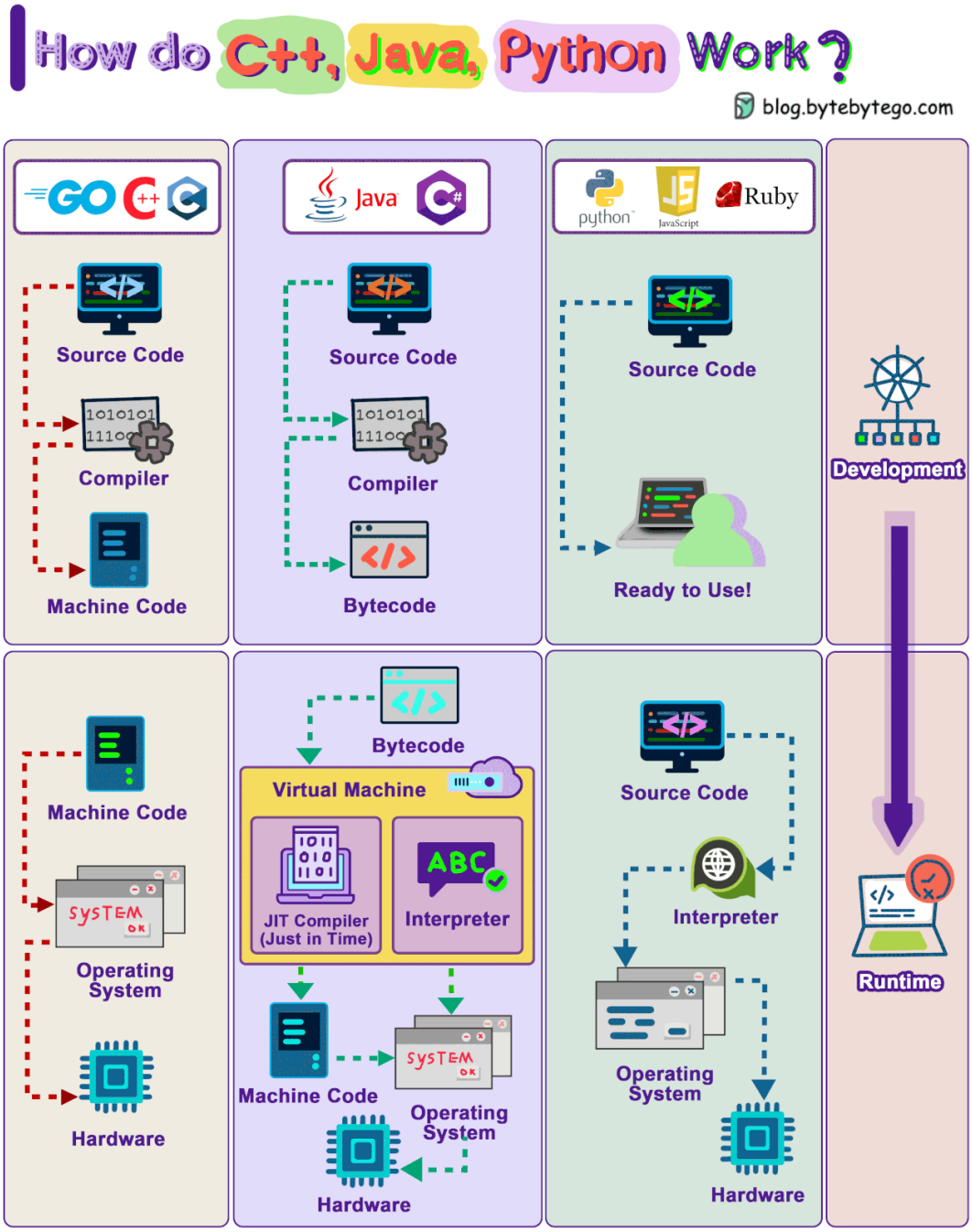
   We can also use Redis key-value pair to generate unique IDs. Redis stores data in memory, so this approach offers better performance than the database.

- Over to you - What ID generator have you used?

How Do C++, Java, and Python Function?

We just made a video on this topic.

The illustration details the processes of compilation and execution.



Languages that compile transform source code into machine code using a compiler. This machine code can subsequently be run directly by the CPU. For instance: C, C++, Go.

In contrast, languages like Java first convert the source code into bytecode. The Java Virtual Machine (JVM) then runs the program. Occasionally, a Just-In-Time (JIT) compiler translates the source code into machine code to enhance execution speed. Some examples are Java and C#.

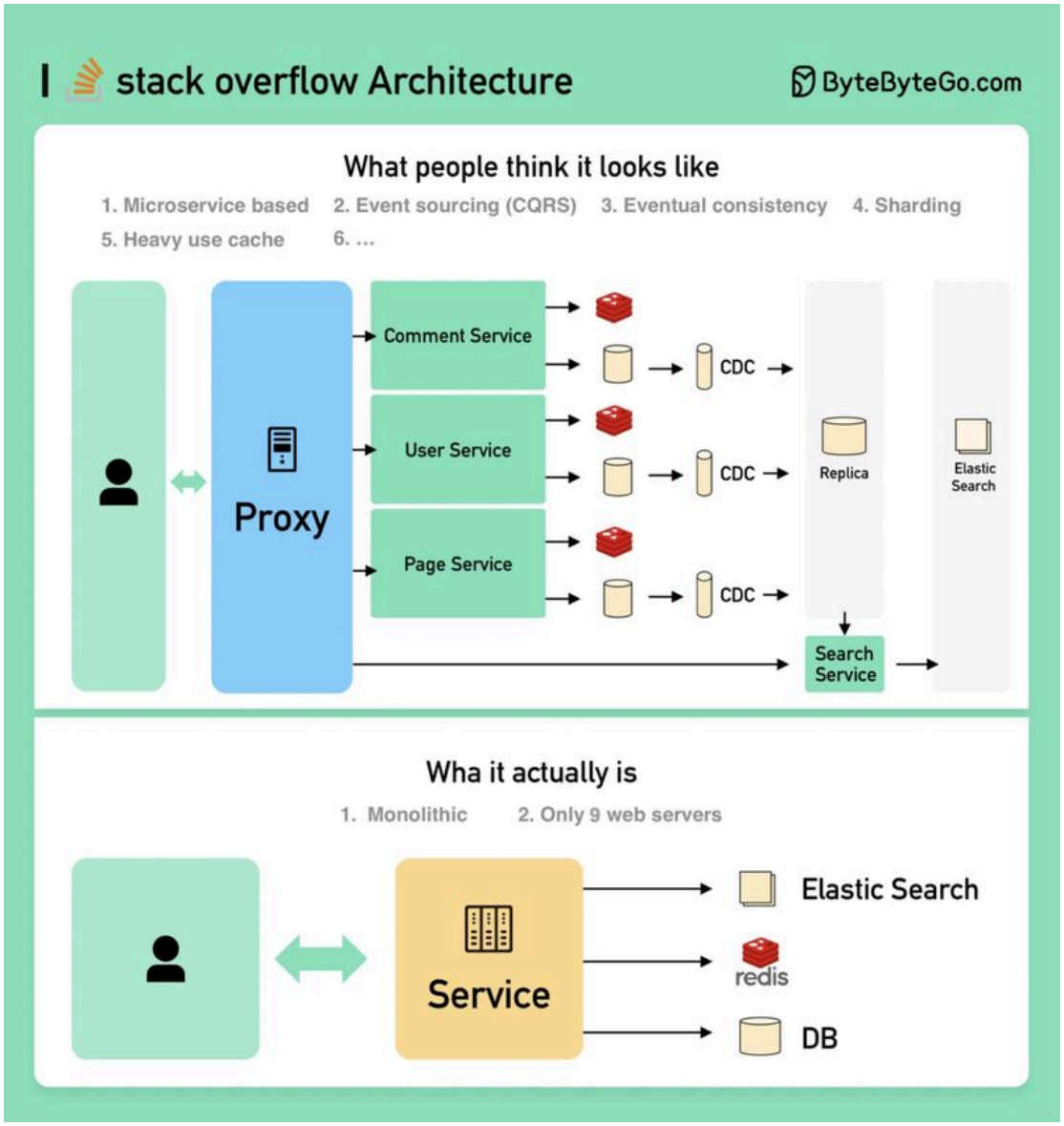
Languages that are interpreted don't undergo compilation. Instead, their code is processed by an interpreter during execution. Python, Javascript, and Ruby are some examples.

Generally, compiled languages have a speed advantage over interpreted ones.

Watch the whole video here: https://lnkd.in/ezpN2jH5

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.

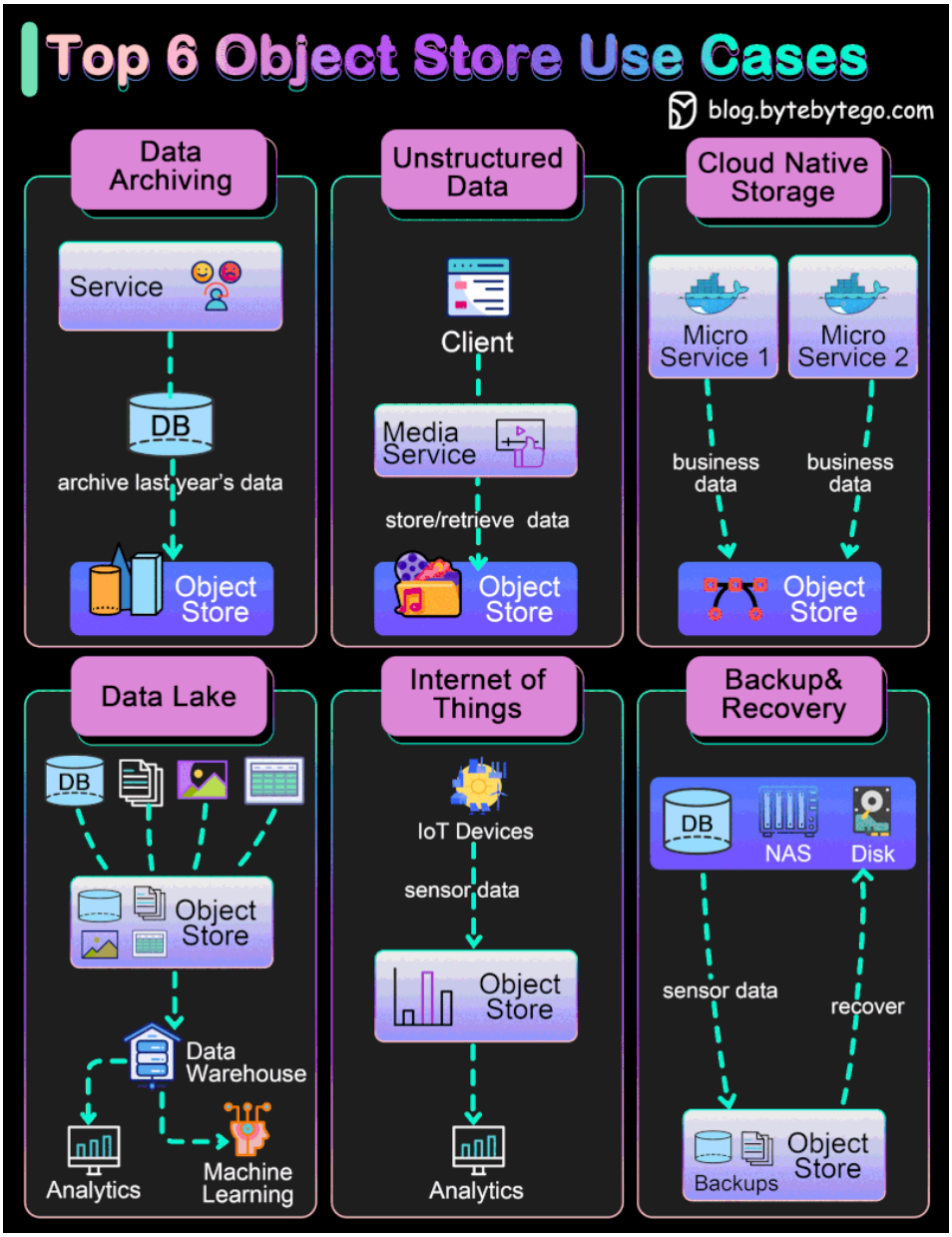
What it actually is

Stack Overflow serves all the traffic with only 9 on-premise web servers, and it's on 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?

Explain the Top 6 Use Cases of Object Stores



- What is an object store?

  Object store uses objects to store data. Compared with file storage which uses a hierarchical structure to store files, or block storage which divides files into equal block sizes, object storage stores metadata together with the objects. Typical products include AWS S3, Google Cloud Storage, and Azure Blob Storage.

  An object store provides flexibility in formats and scales easily.

- Case 1: Data Archiving

With the ever-growing amounts of business data, we cannot store all the data in core storage systems. We need to have layers of storage plan. An object store can be used to archive old data that exists for auditing or client statements. This is a cost-effective approach.

- Case 2: Unstructured Data Storage

  We often need to deal with unstructured data or semi-structured data. In the past, they were usually stored as blobs in the relational database, which was quite inefficient. An object store is a good match for music, video files, and text documents. Companies like Spotify or Netflix uses object store to persist their media files.

- Case 3: Cloud Native Storage

  For cloud-native applications, we need the data storage system to be flexible and scalable. Major public cloud providers have easy API access to their object store products and can be used for economical storage choices.

- Case 4: Data Lake

  There are many types of data in a distributed system. An object store-backed data lake provides a good place for different business lines to dump their data for later analytics or machine learning. The efficient reads and writes of the object store facilitate more steps down the data processing pipeline, including ETL(Extract-Transform-Load) or constructing a data warehouse.

- Case 5: Internet of Things (IoT)

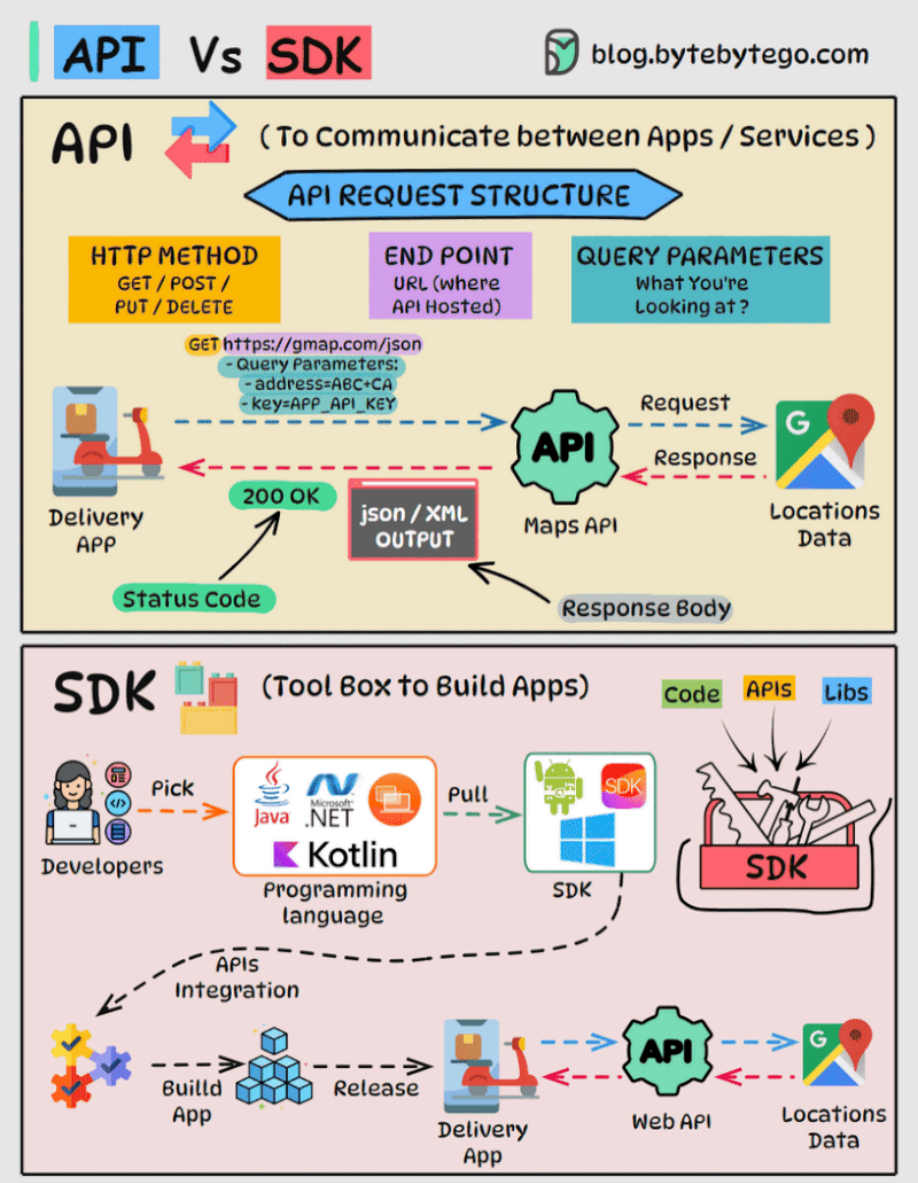
  IoT sensors produce all kinds of data. An object store can store this type of time series and later run analytics or AI algorithms on them. Major public cloud providers provide pipelines to ingest raw IoT data into the object store.

- Case 6: Backup and Recovery

  An object store can be used to store database or file system backups. Later, the backups can be loaded for fast recovery. This improves the system's availability.

Over to you: What did you use object store for?

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.

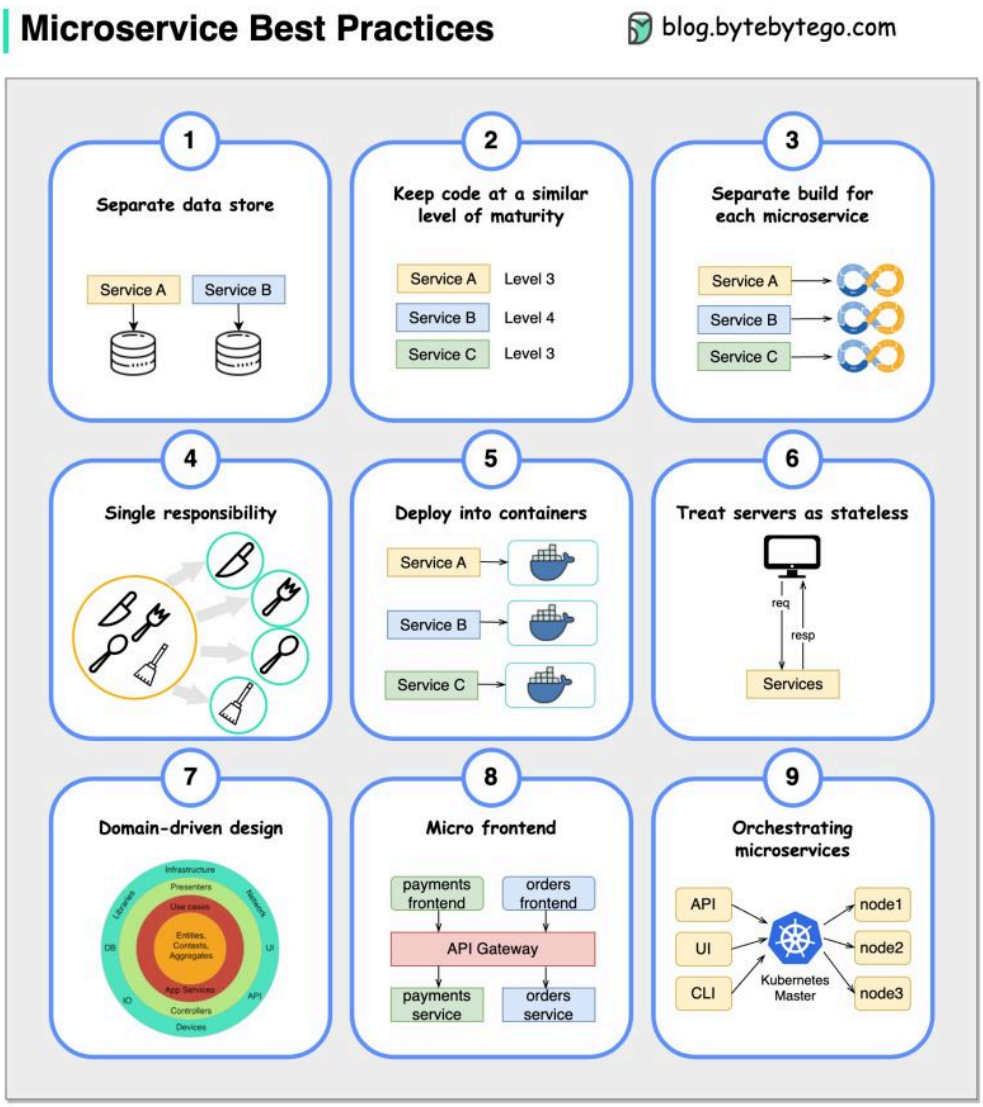
1. Offers higher-level abstractions, simplifying development for a specific platform.
2. Tailored to specific platforms or frameworks, ensuring compatibility and optimal performance on that platform.
3. Offer access to advanced features and capabilities specific to the platform, which might be otherwise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

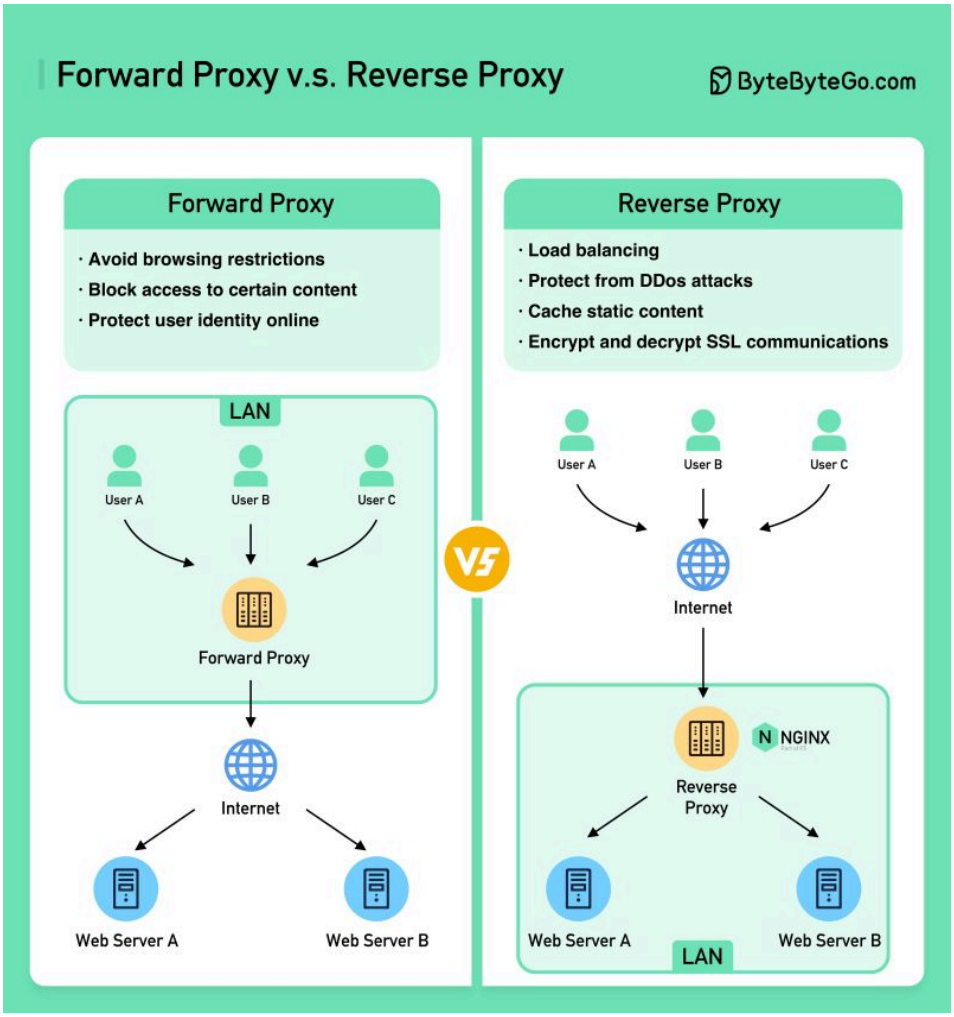
A picture is worth a thousand words: 9 best practices for developing microservices



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

Proxy Vs reverse proxy



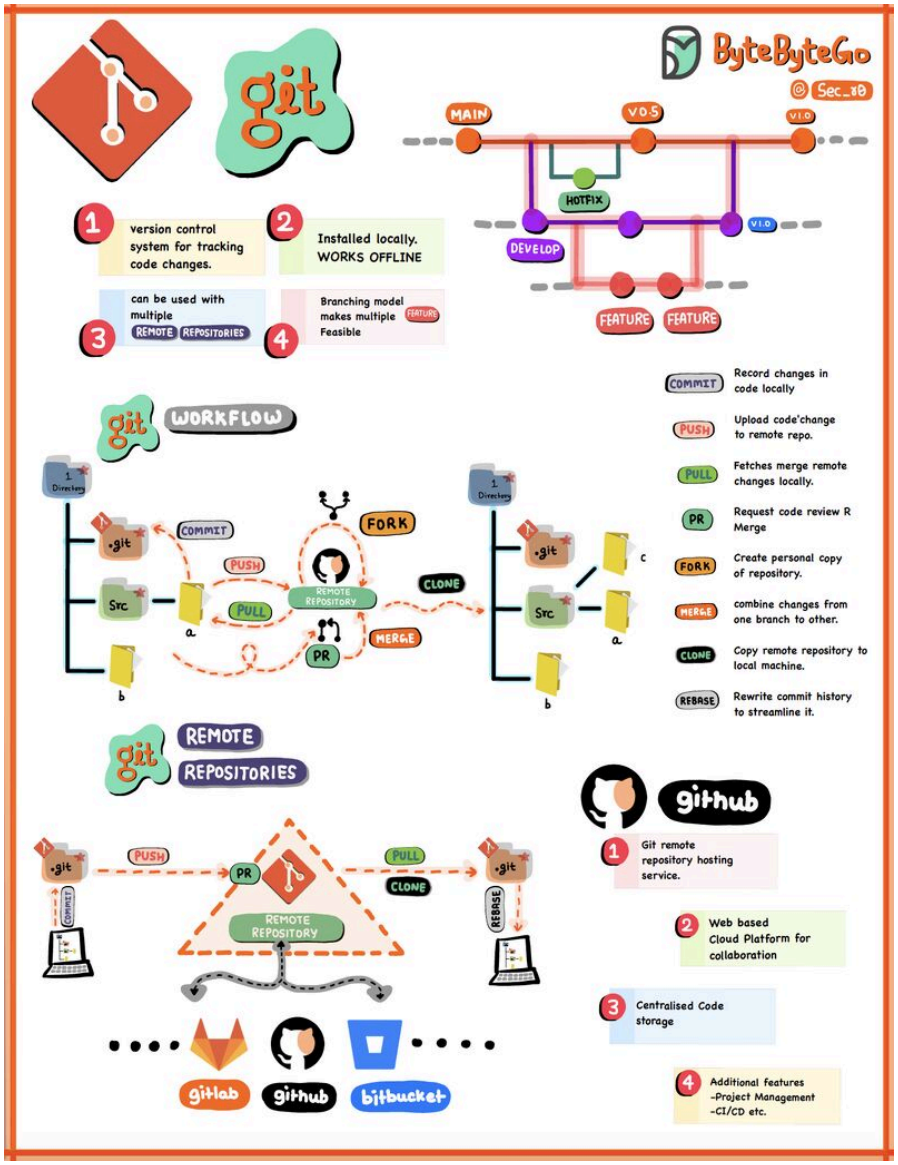
A forward proxy is a server that sits between user devices and the internet. A forward proxy is commonly used for:

- Protect clients
- Avoid browsing restrictions
- Block access to certain content

A reverse proxy is a server that accepts a request from the client, forwards the request to web servers, and returns the results to the client as if the proxy server had processed the request. A reverse proxy is good for:

- Protect servers
- Load balancing
- Cache static contents
- Encrypt and decrypt SSL communications

Git Vs Github



Dive into the fascinating world of version control.

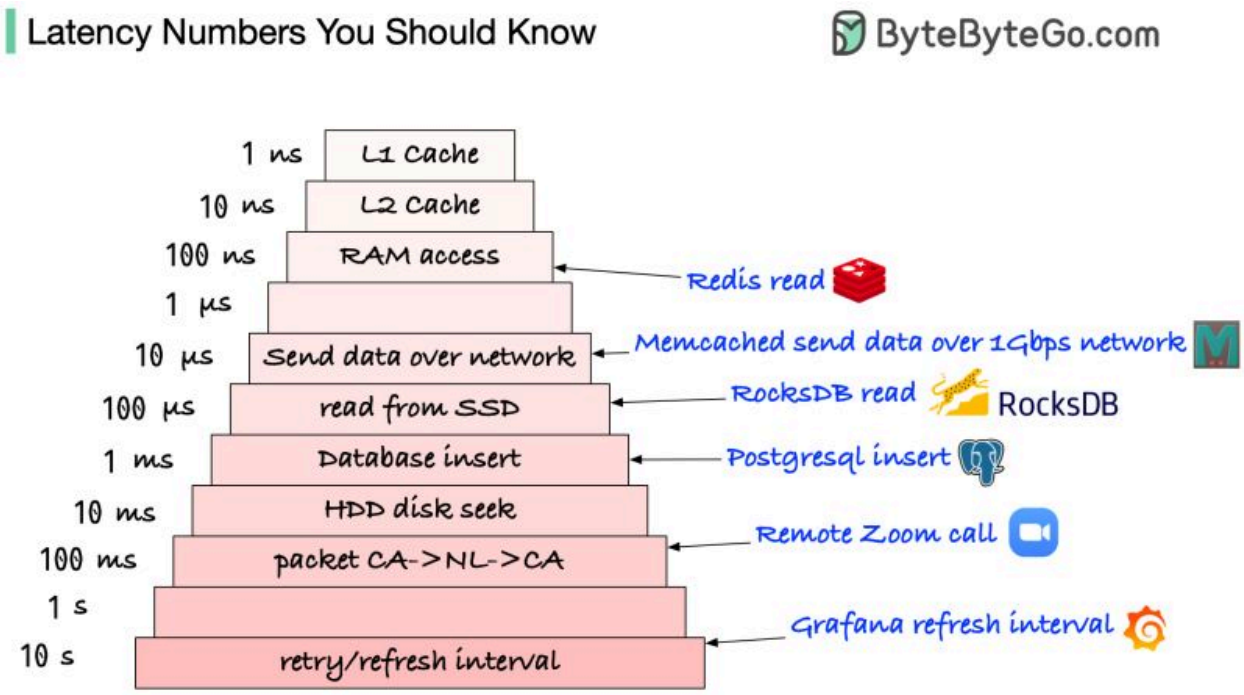
First, meet Git, a fundamental tool for developers. It operates locally, allowing you to track changes in your code, much like taking snapshots of your project's progress. This makes collaboration with your team a breeze, even when you're working on the same project.

Now, let's talk about GitHub. It's more than just a platform; it's a powerhouse for hosting Git repositories online. With GitHub, you can streamline team collaboration and code sharing.

Learning Git and GitHub is a fundamental part of software engineering, so definitely try your best to master them 🚀

Which latency numbers should you know

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 (default value on Grafana).

Notes:

1 ns = 10^-9 seconds

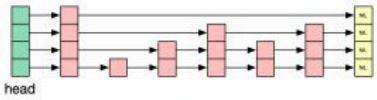
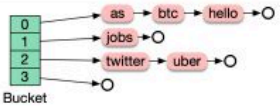
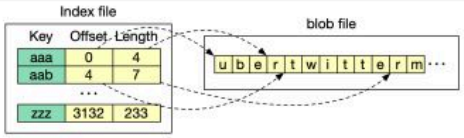
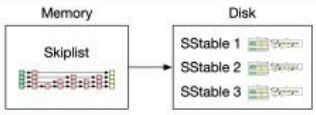
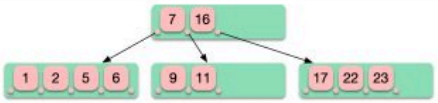
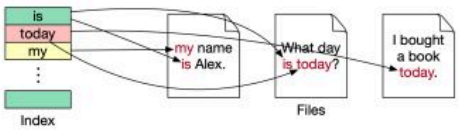
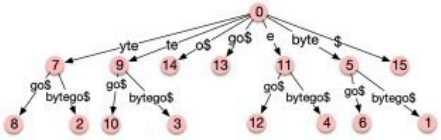
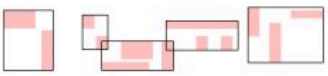
1 us = 10^-6 seconds = 1,000 ns

1 ms = 10^-3 seconds = 1,000 us = 1,000,000 ns

Eight Data Structures That Power Your Databases. Which one should we pick?

The answer will vary depending on your use case. Data can be indexed in memory or on disk. Similarly, data formats vary, such as numbers, strings, geographic coordinates, etc. The system might be write-heavy or read-heavy. All of these factors affect your choice of database index format.

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

The following are some of the most popular data structures used for indexing data:

- 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

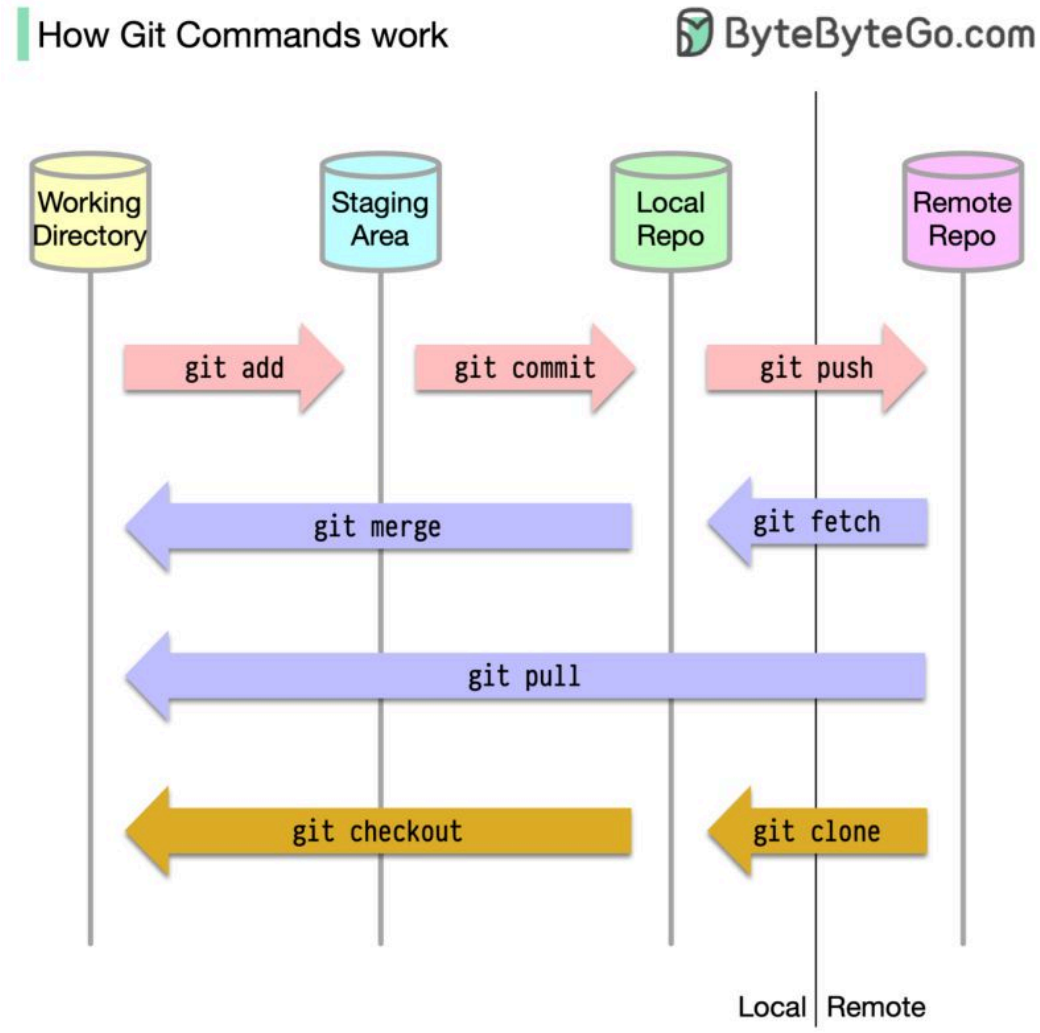
This is not an exhaustive list of all database index types.

Over to you:

1. Which one have you used and for what purpose?
2. There is another one called “reverse index”. Do you know the difference between “reverse index” and “inverted index”?

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:

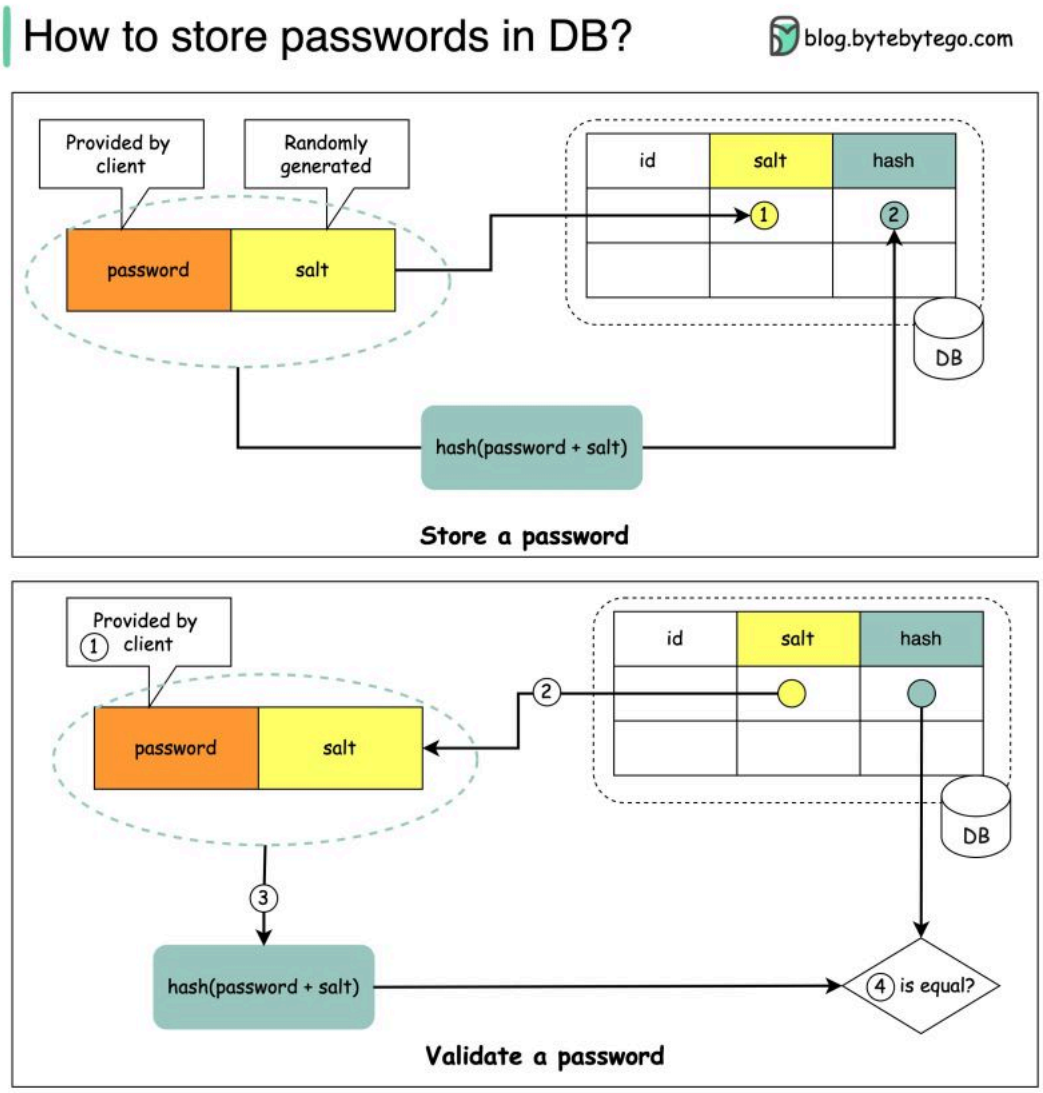
- Working directory: where we edit files
- Staging area: a temporary location where files are kept for the next commit
- Local repository: contains the code that has been committed
- Remote repository: the remote server that stores the code

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.

How to store passwords safely in the database and how to validate a password?

Let's take a look.



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 pruned 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: *hash*(*password* + *salt*).

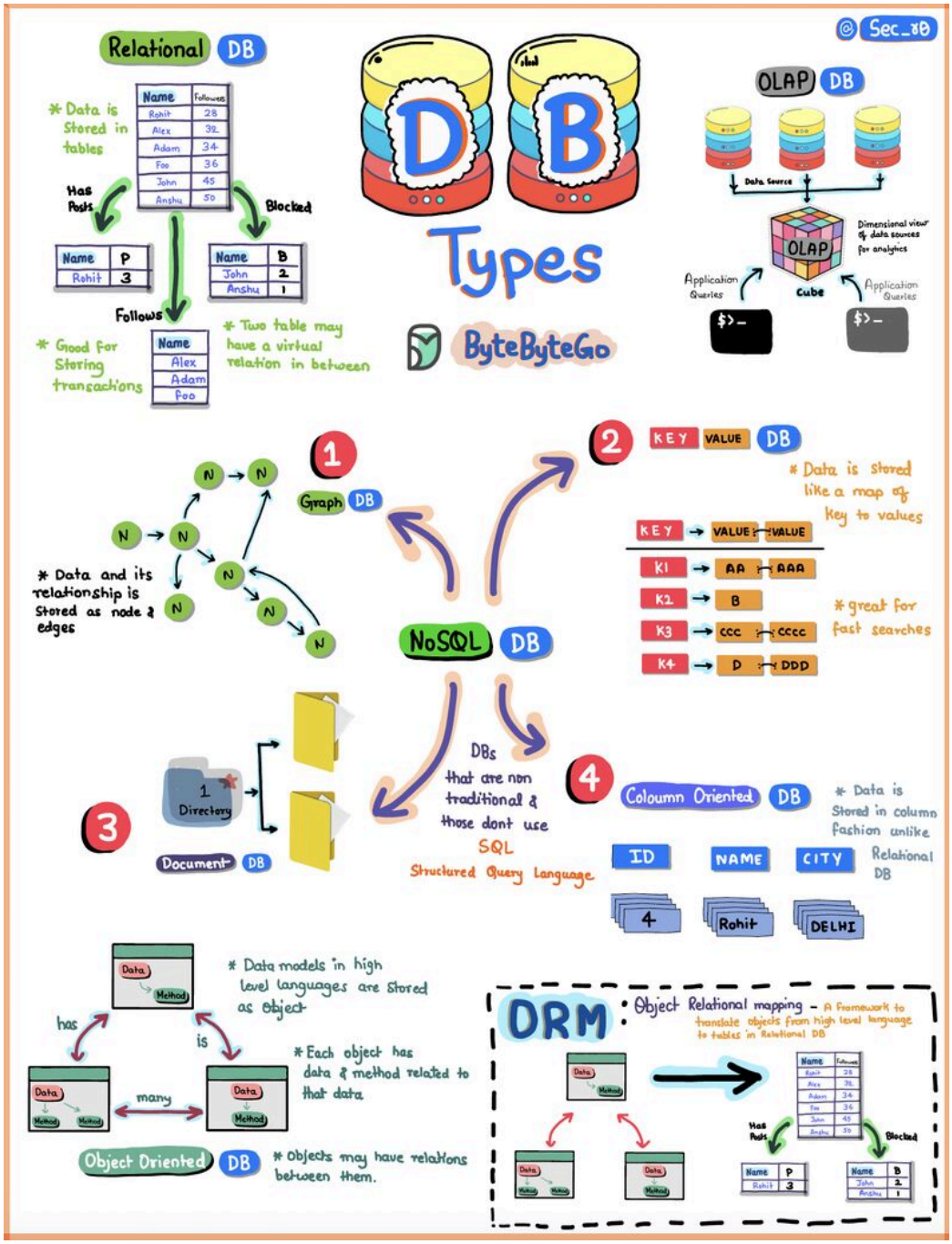
How to validate a password?
To validate a password, it can go through the following process:

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?

What is a database? What are some common types of databases?

First off, what's a database? Think of it as a digital playground where we organize and store loads of information in a structured manner. Now, let's shake things up and look at the main types of databases.



Relational DB: Imagine it's like organizing data in neat tables. Think of it as the well-behaved sibling, keeping everything in order.

OLAP DB: Online Analytical Processing (OLAP) is a technology optimized for reporting and analysis purposes.

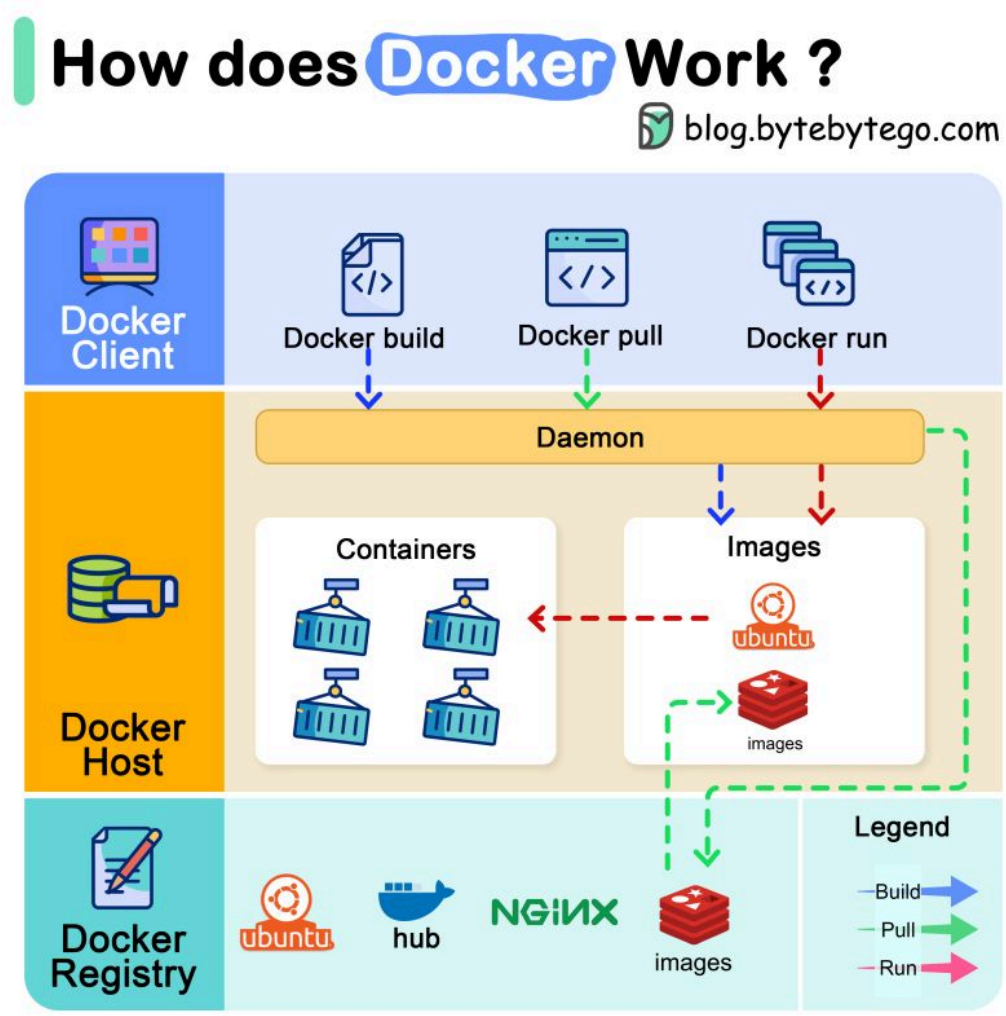
NoSQL DBs: These rebels have their own cool club, saying "No" to traditional SQL ways. NoSQL databases come in four exciting flavors:

- Graph DB: Think of social networks, where relationships between people matter most. It's like mapping who's friends with whom.
- Key-value Store DB: It's like a treasure chest, with each item having its unique key. Finding what you need is a piece of cake.
- Document DB: A document database is a kind of database that stores information in a format similar to JSON. It's different from traditional databases and is made for working with documents instead of tables.
- Column DB: Imagine slicing and dicing your data like a chef prepping ingredients. It's efficient and speedy.

Over to you: So, the next time you hear about databases, remember, it's a wild world out there - from orderly tables to rebellious NoSQL variants! Which one is your favorite? Share your thoughts!

How does Docker Work? Is Docker still relevant?

We just made a video on this topic.



Docker's architecture comprises three main components:

- Docker Client

  This is the interface through which users interact. It communicates with the Docker daemon.

- Docker Host

  Here, the Docker daemon listens for Docker API requests and manages various Docker objects, including images, containers, networks, and volumes.

- Docker Registry

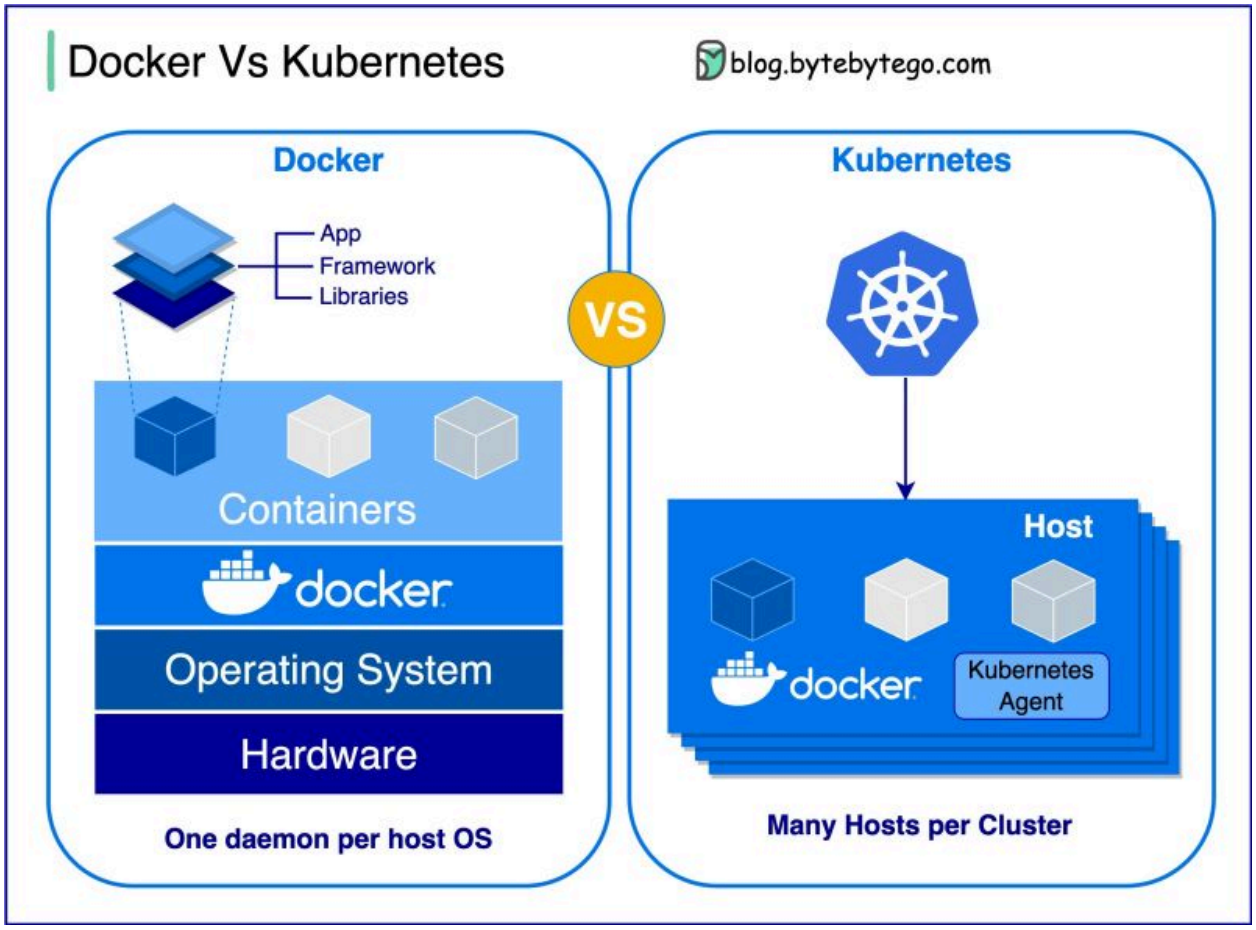
This is where Docker images are stored. Docker Hub, for instance, is a widely-used public registry.

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.

Is Docker still relevant? Watch the whole video here: https://lnkd.in/eKDkkq_m

Docker vs. Kubernetes. Which one should we use?



What is Docker?

Docker is an open-source platform that allows you to package, distribute, and run applications in isolated containers. It focuses on containerization, providing lightweight environments that encapsulate applications and their dependencies.

What is Kubernetes?

Kubernetes, often referred to as K8s, is an open-source container orchestration platform. It provides a framework for automating the deployment, scaling, and management of containerized applications across a cluster of nodes.

How are both different from each other?

Docker: Docker operates at the individual container level on a single operating system host.

You must manually manage each host and setting up networks, security policies, and storage for multiple related containers can be complex.

Kubernetes: Kubernetes operates at the cluster level. It manages multiple containerized applications across multiple hosts, providing automation for tasks like load balancing, scaling, and ensuring the desired state of applications.

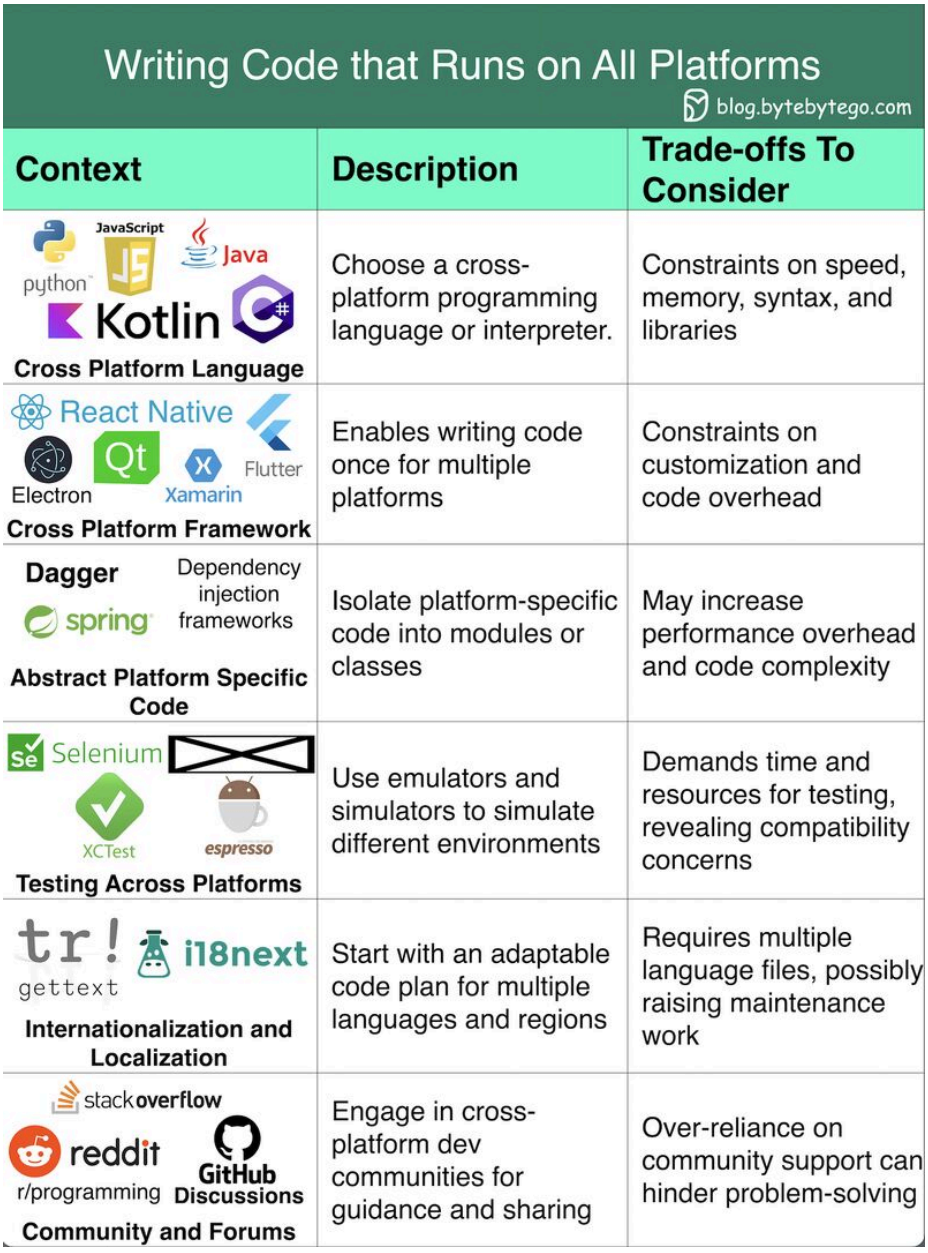
In short, Docker focuses on containerization and running containers on individual hosts, while Kubernetes specializes in managing and orchestrating containers at scale across a cluster of hosts.

Over to you: What challenges prompted you to switch from Docker to Kubernetes for managing containerized applications?

Writing Code that Runs on All Platforms

Developing code that functions seamlessly across different platforms is a crucial skill for modern programmers.

The need arises from the fact that users access software on a wide range of devices and operating systems. Achieving this universal compatibility can be complex due to differences in hardware, software environments, and user expectations.



Creating code that works on all platforms requires careful planning and understanding of the unique challenges presented by each platform.

Better planning and comprehension of cross-platform development not only streamline the process but also contribute to the long-term success of a software project.

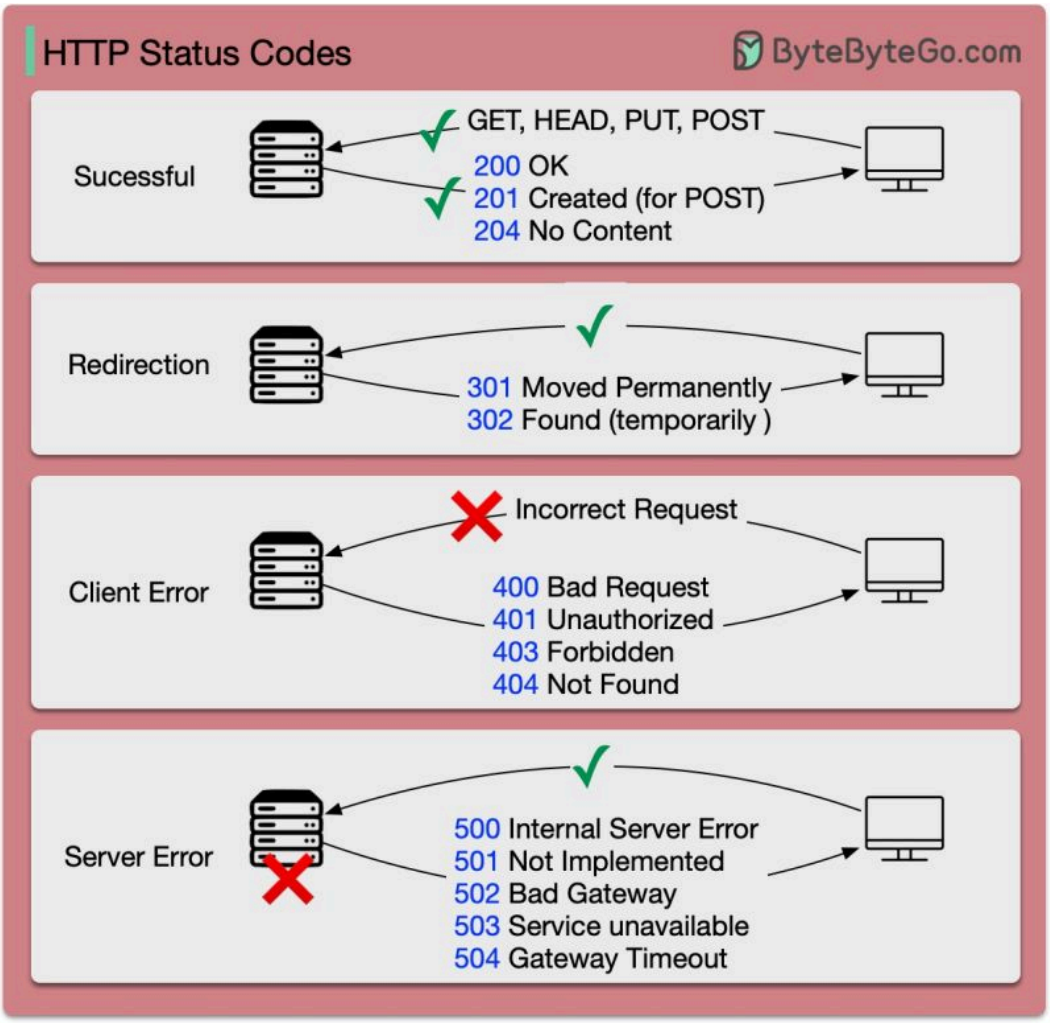
It reduces redundancy, simplifies maintenance, ensures consistency, boosting satisfaction and market reach.

Here are key factors for cross-platform compatibility

Over to you: How have you tackled cross-platform compatibility challenges in your projects? Share your insights and experiences!

HTTP Status Code You Should Know

We just made a YouTube video on this topic. The link to the video is at the end of the post.



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: