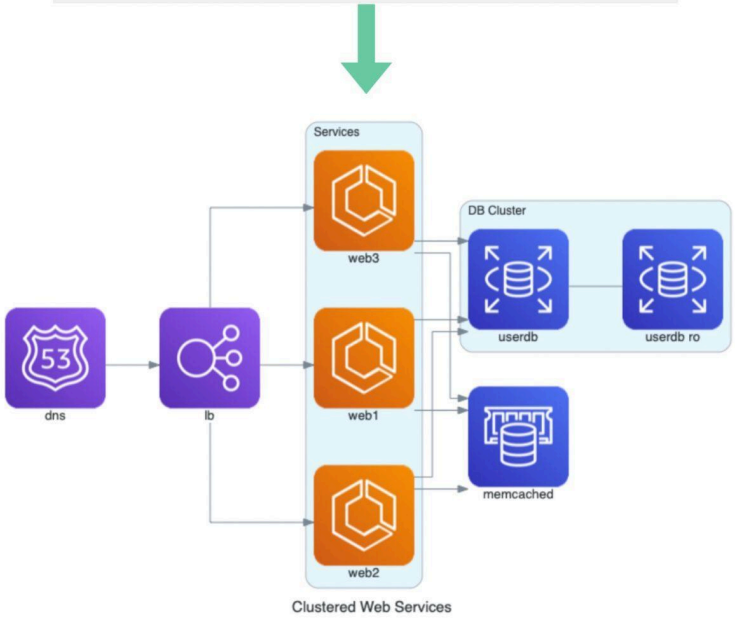
Would it be nice if the code we wrote automatically turned into architecture diagrams?

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

Diagram as Code

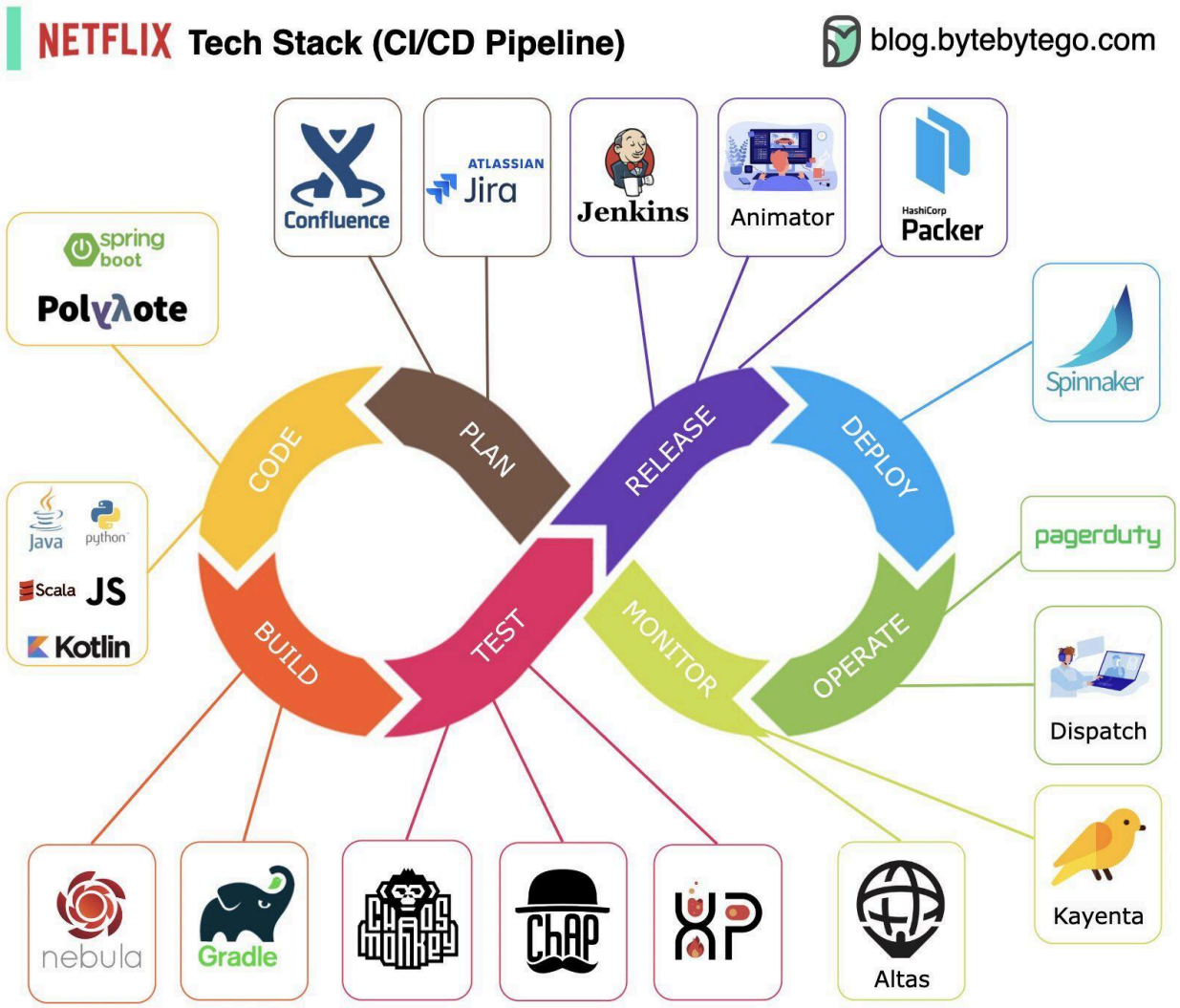
```
from diagrams import Cluster, Diagram
from diagrams.aws.compute import ECS
from diagrams.aws.database import ElastiCache, RDS
from diagrams.aws.network import ELB
from diagrams.aws.network import Route53

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.

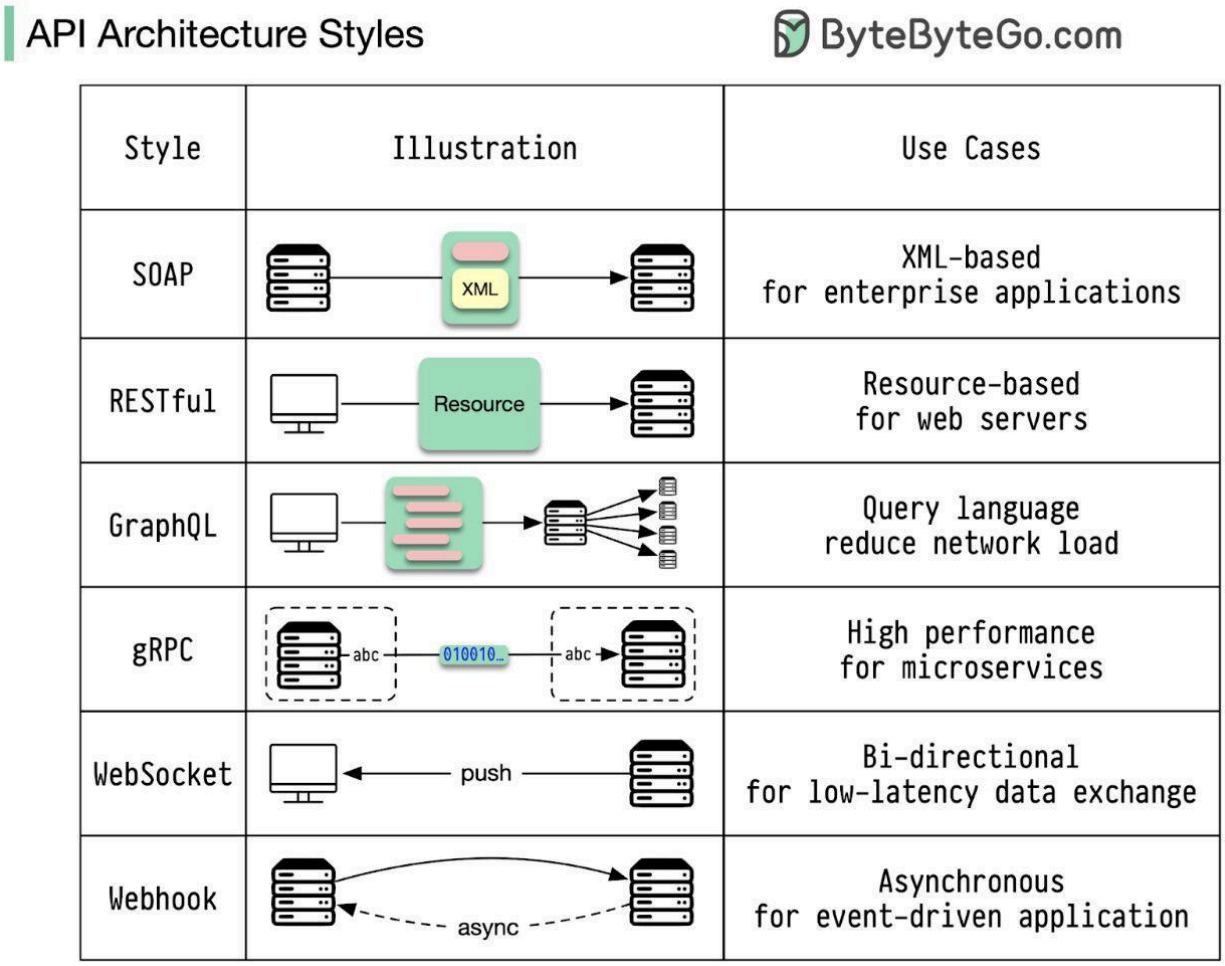
18 Key Design Patterns Every Developer Should Know

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

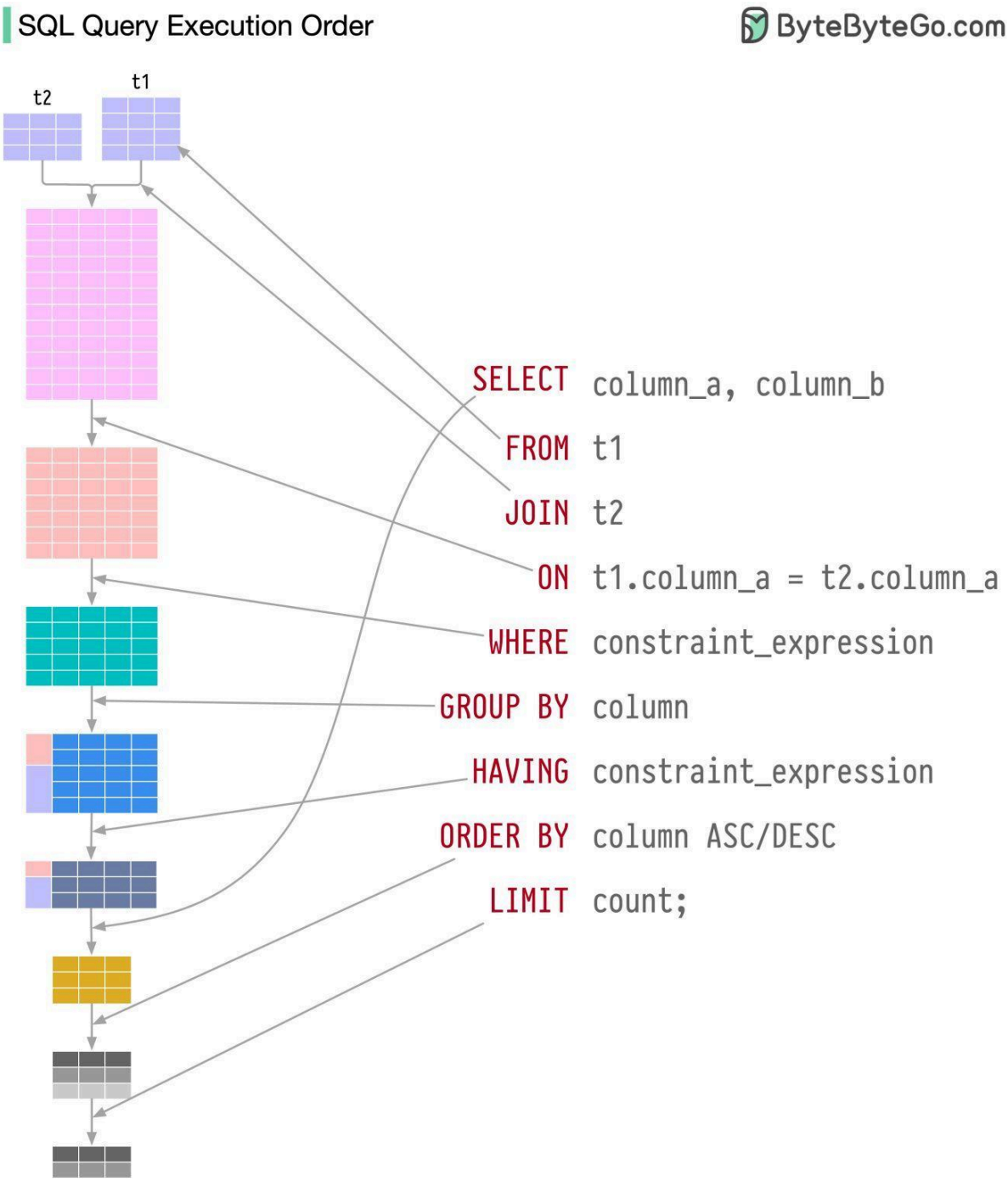
- gRPC:
  Modern, high-performance, Protocol Buffers
  Suitable for microservices architectures

- 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

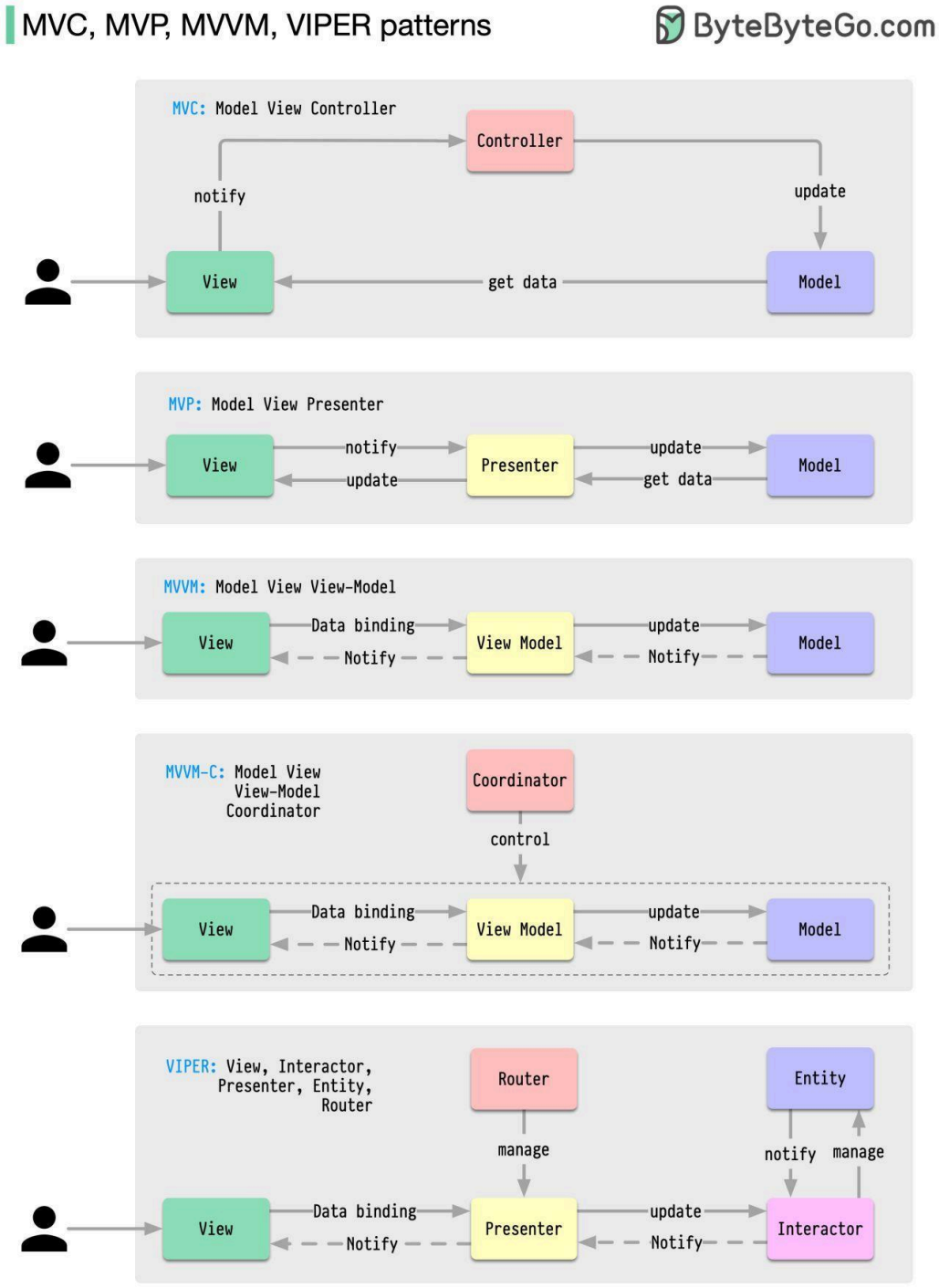
- Optimizing the internal representation and creating an execution plan that utilizes index information
- Executing the plan and returning the results

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?

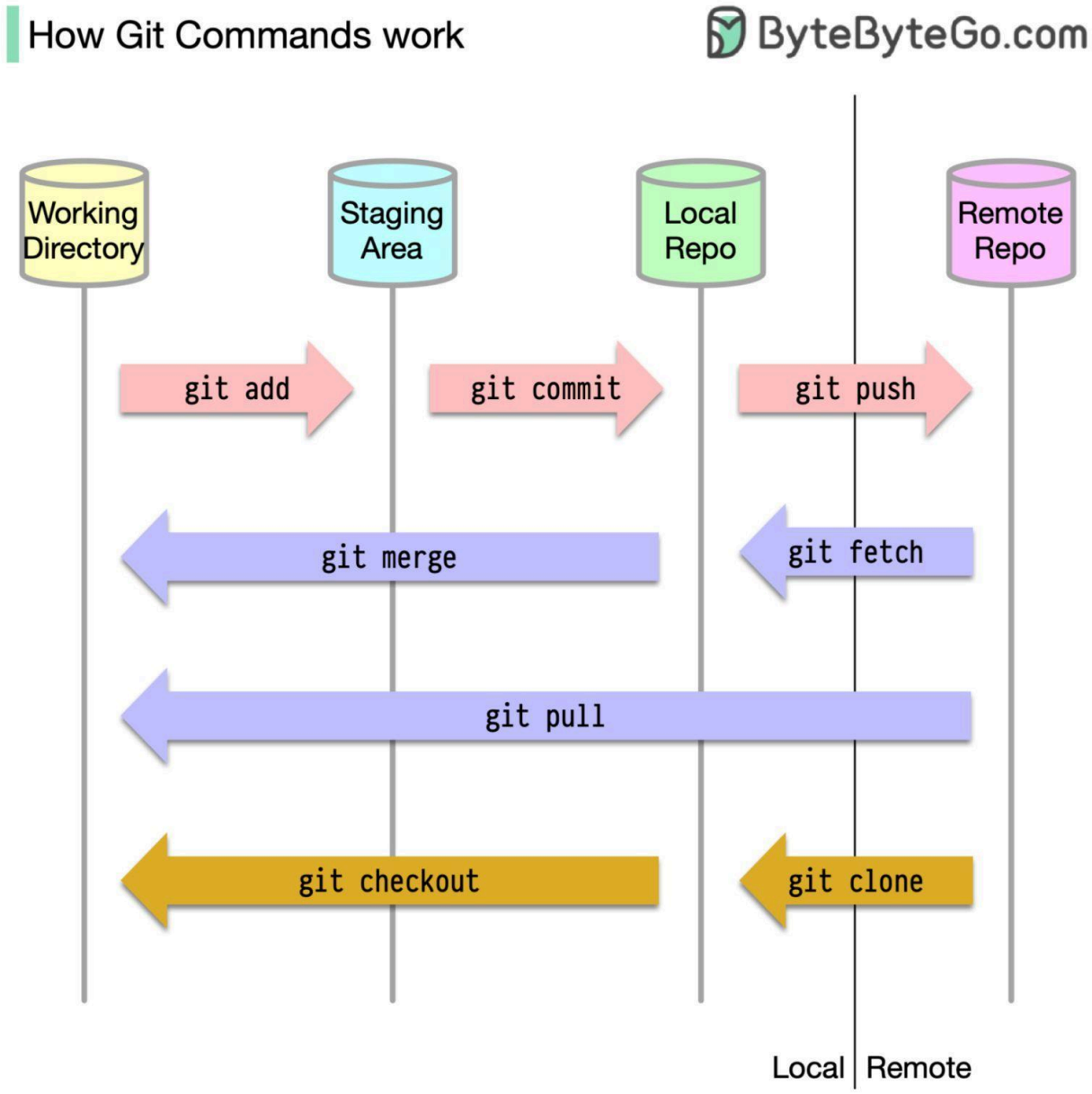


These architecture patterns are among the most commonly used in app development, whether on iOS or Android platforms. Developers have introduced them to overcome the limitations of earlier patterns. So, how do they differ?

- 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:

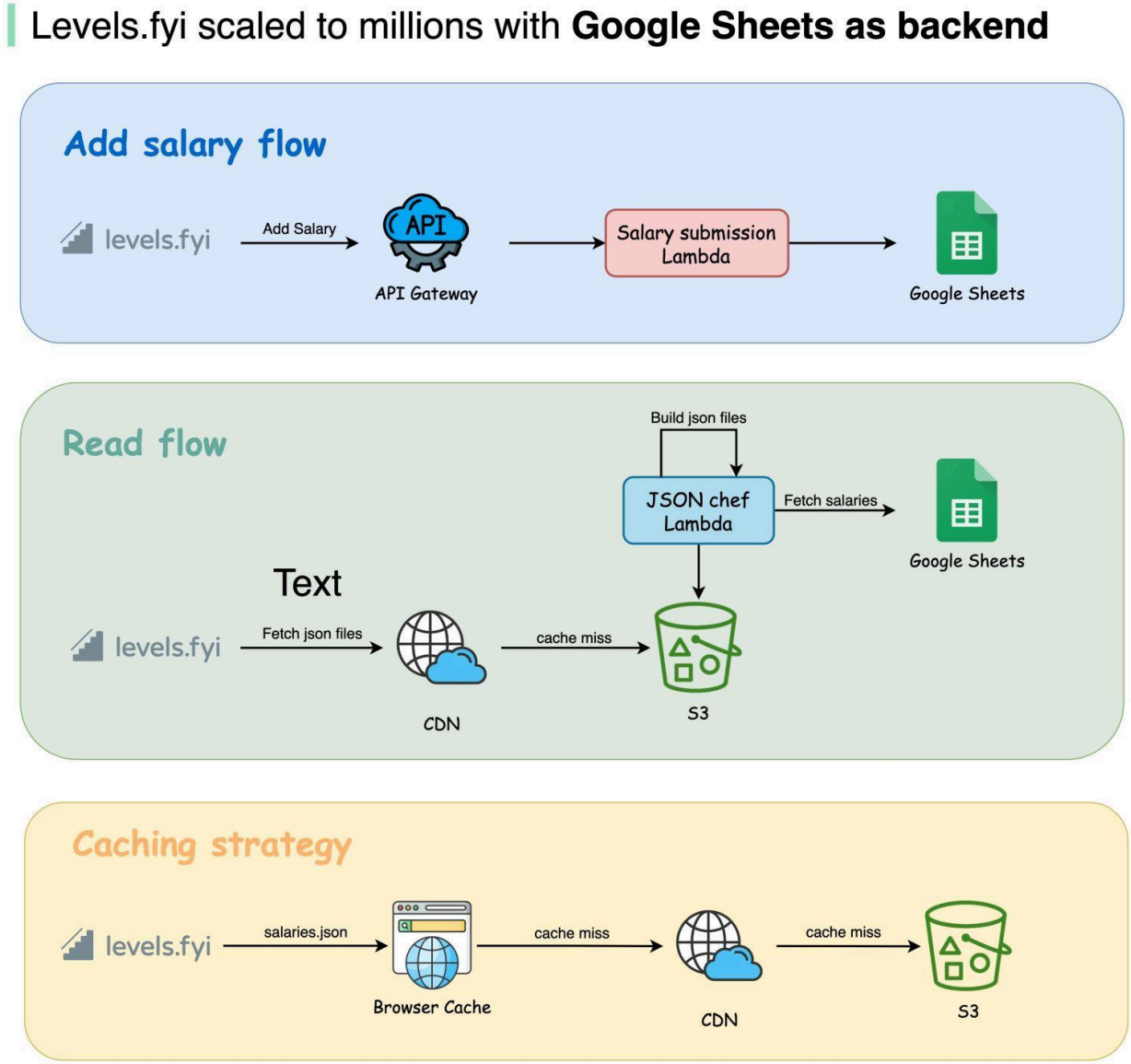
- Working directory: where we edit files
- Staging area: a temporary location where files are kept for the next commit

- 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

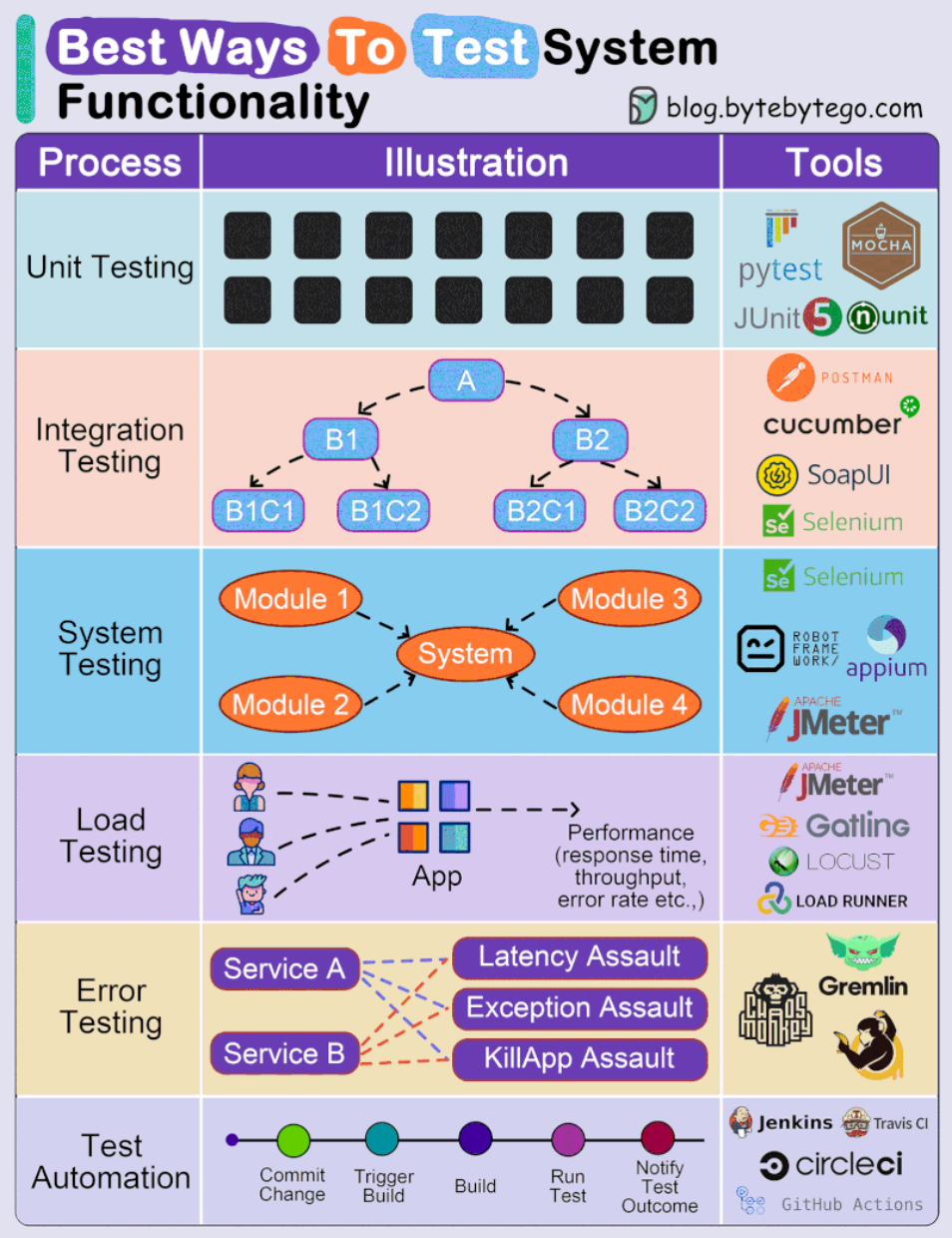
counterintuitive idea for a site with our traffic volume to not have a backend or any fancy infrastructure, but our philosophy to building products has always been, start simple and iterate. This allows us to move fast and focus on what’s important".

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.

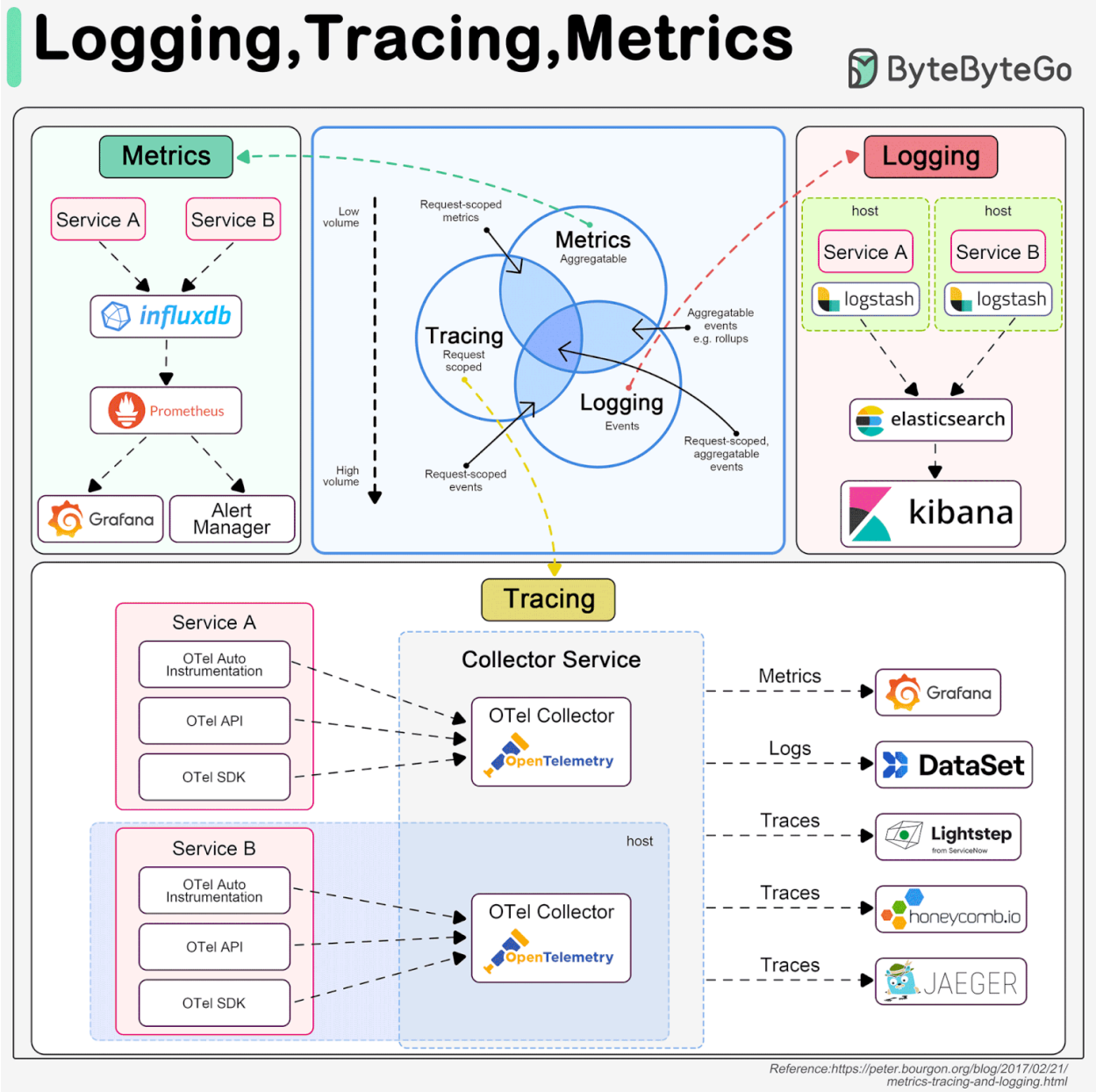
5. Error Testing: Evaluates how the software handles invalid inputs and error conditions.

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.

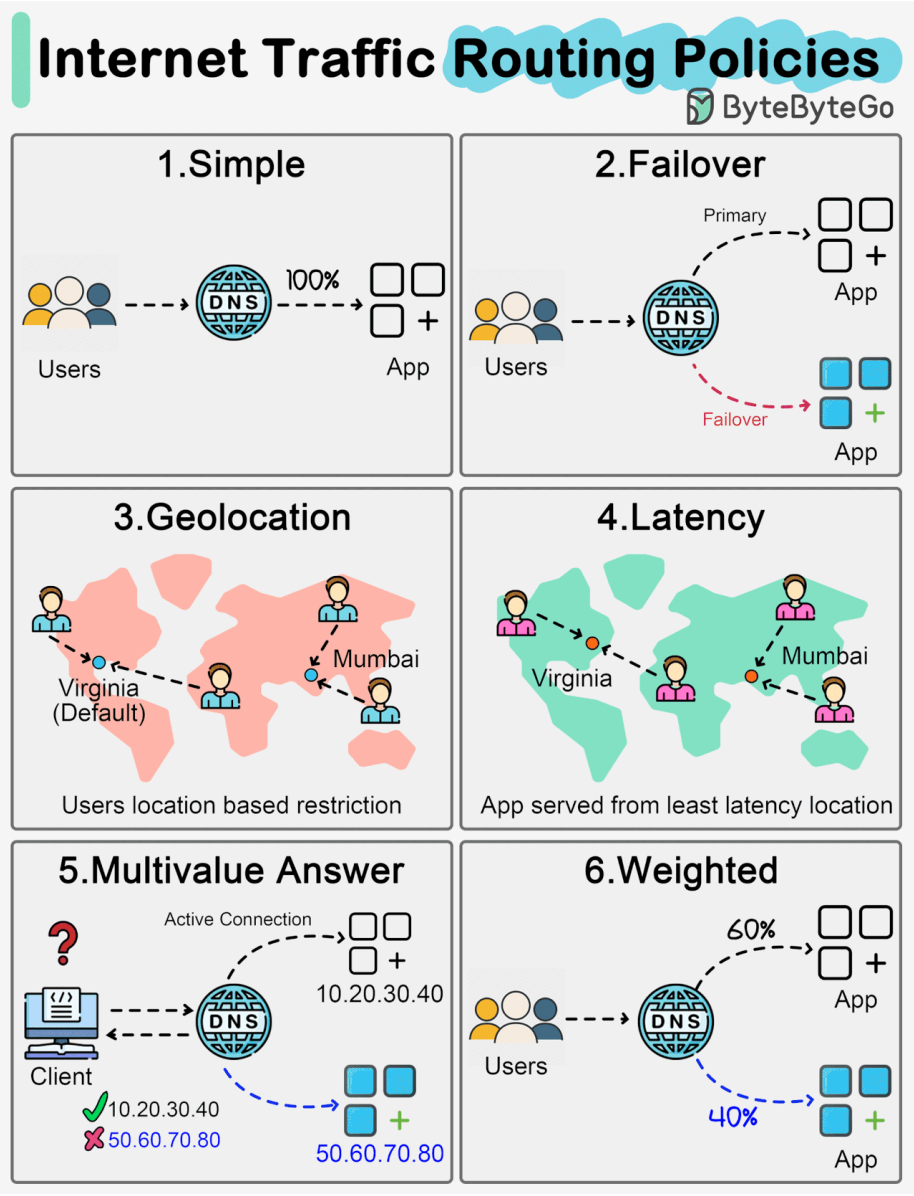
- Tracing
  Tracing is usually request-scoped. For example, a user request goes through the API gateway, load balancer, service A, service B, and database, which can be visualized in the tracing systems. This is useful when we are trying to identify the bottlenecks in the system. We use OpenTelemetry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

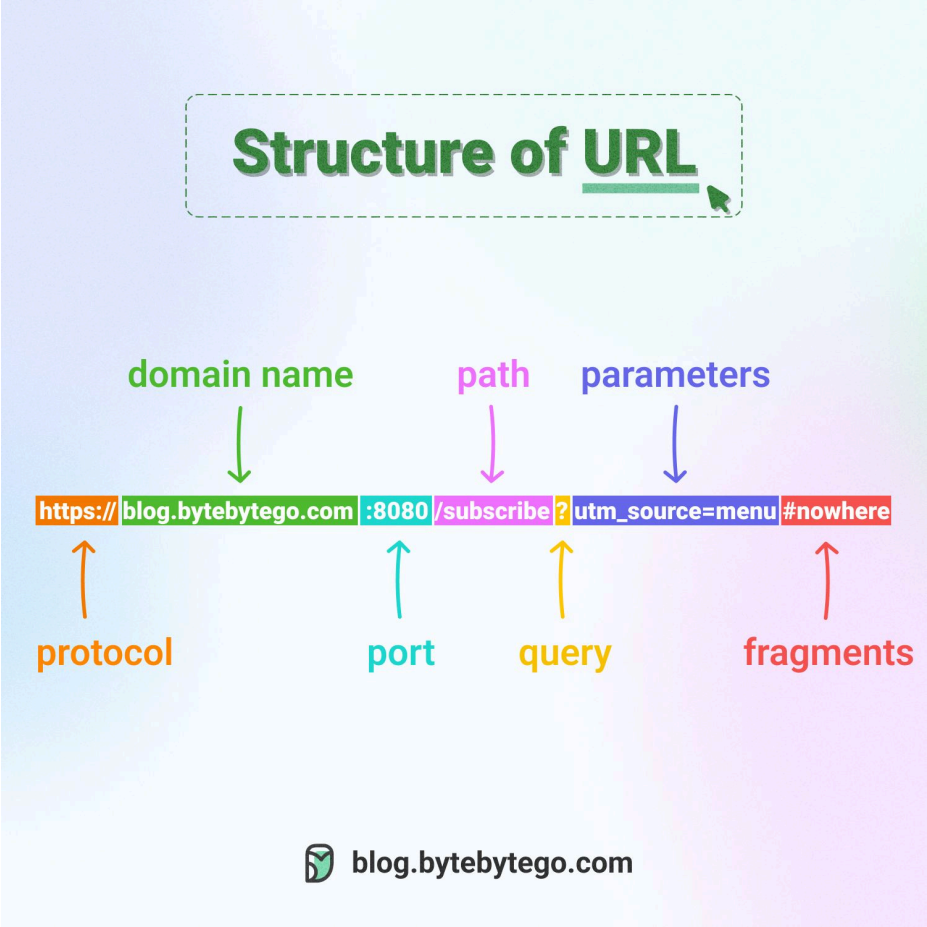
An interesting list of subjects that should be mandatory in schools by startup_rules.

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.

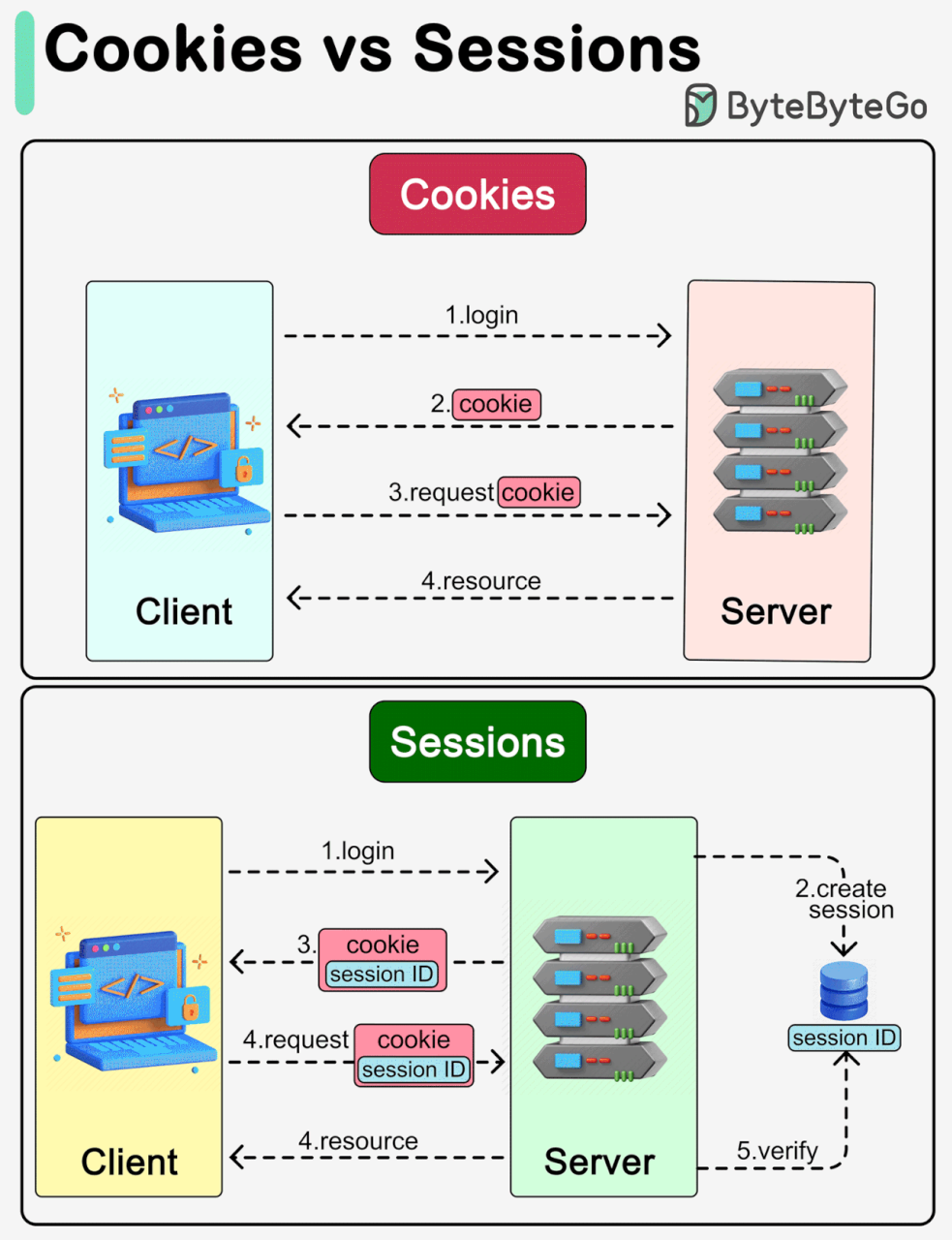


URLs comprise several components:

- 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.

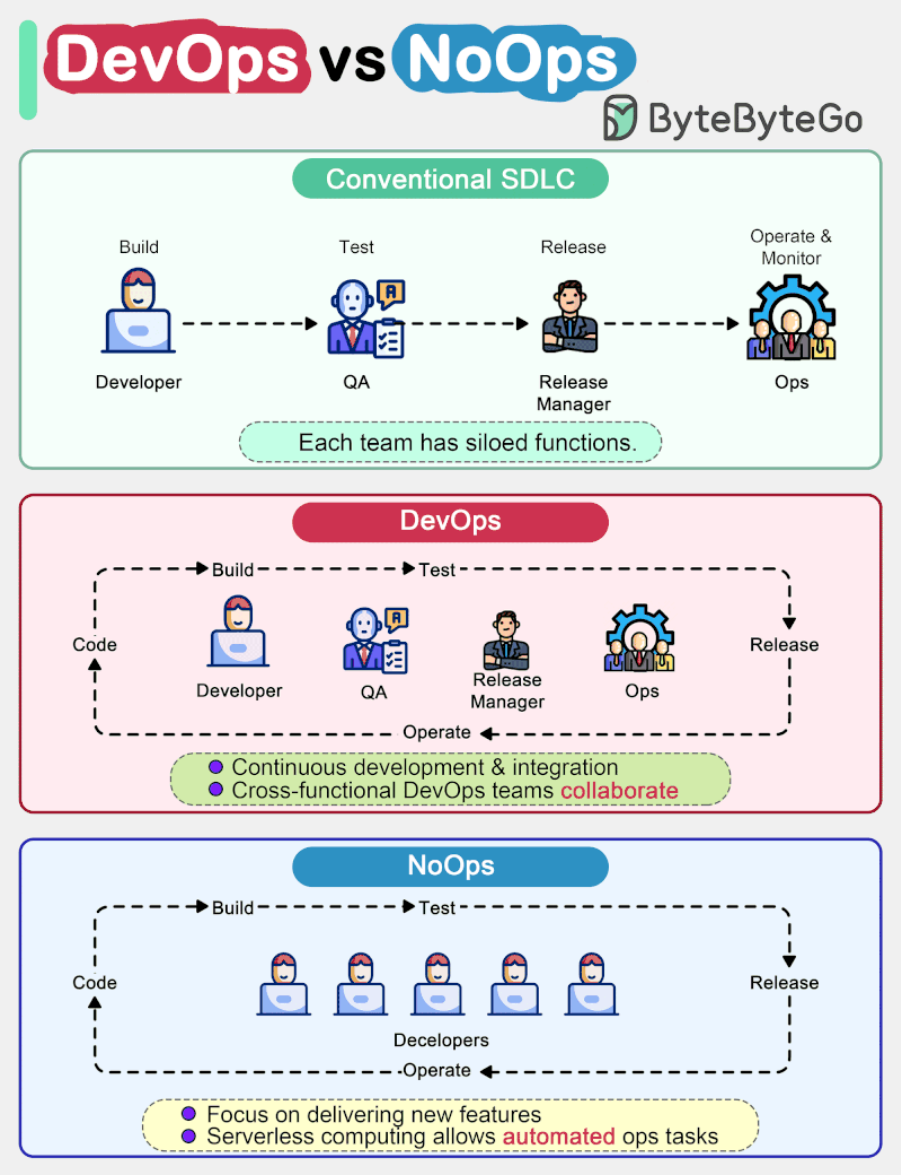
- Cookies
  Cookies typically have size limits (4KB). They carry small pieces of information and are stored on the users' devices. Cookies are sent with each subsequent user request. Users

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.

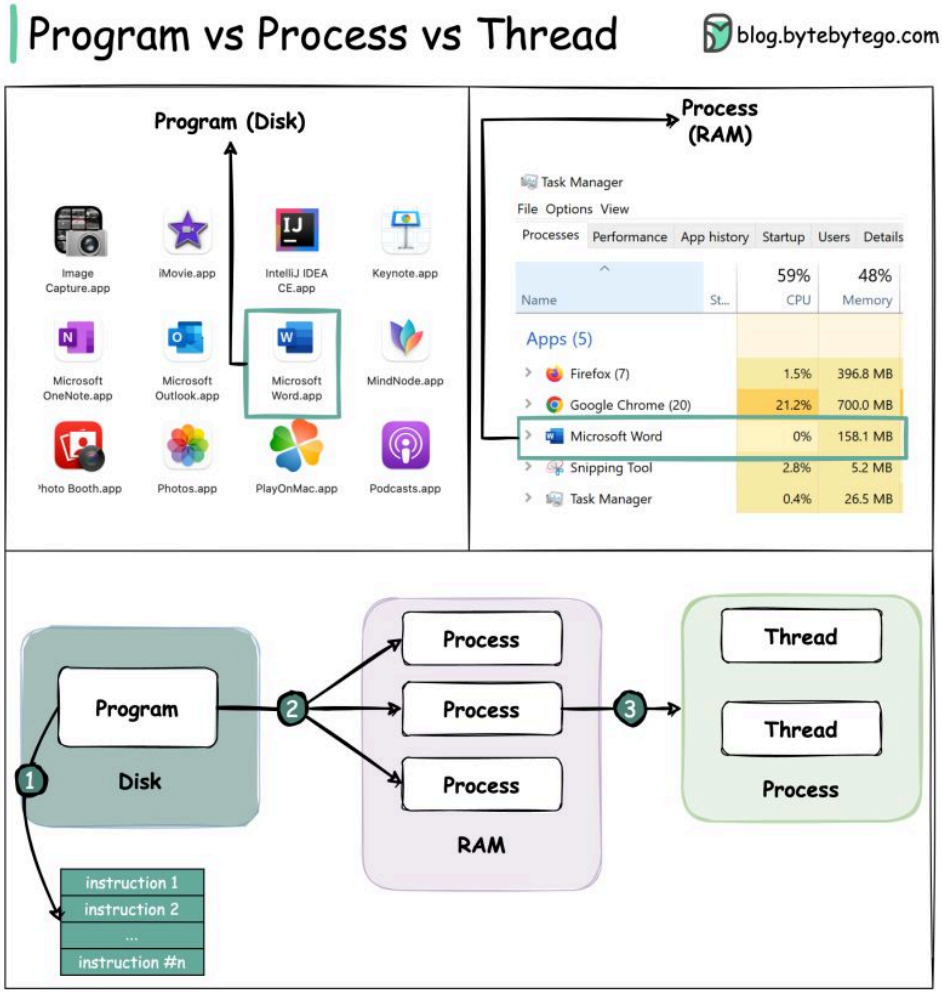
NoOps is a pragmatic and effective methodology for startups or smaller-scale applications, which moves shortens the SDLC even more than DevOps.

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.

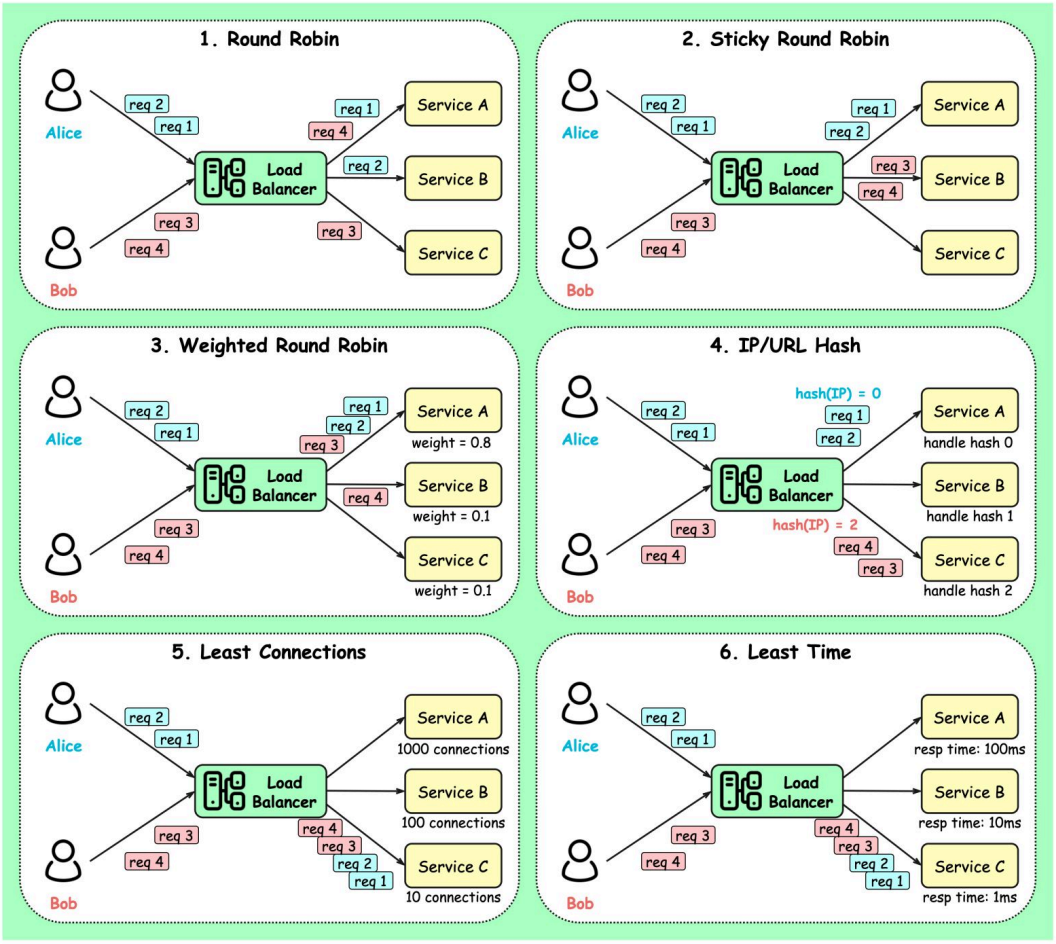
Main differences between process and thread:

- 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.

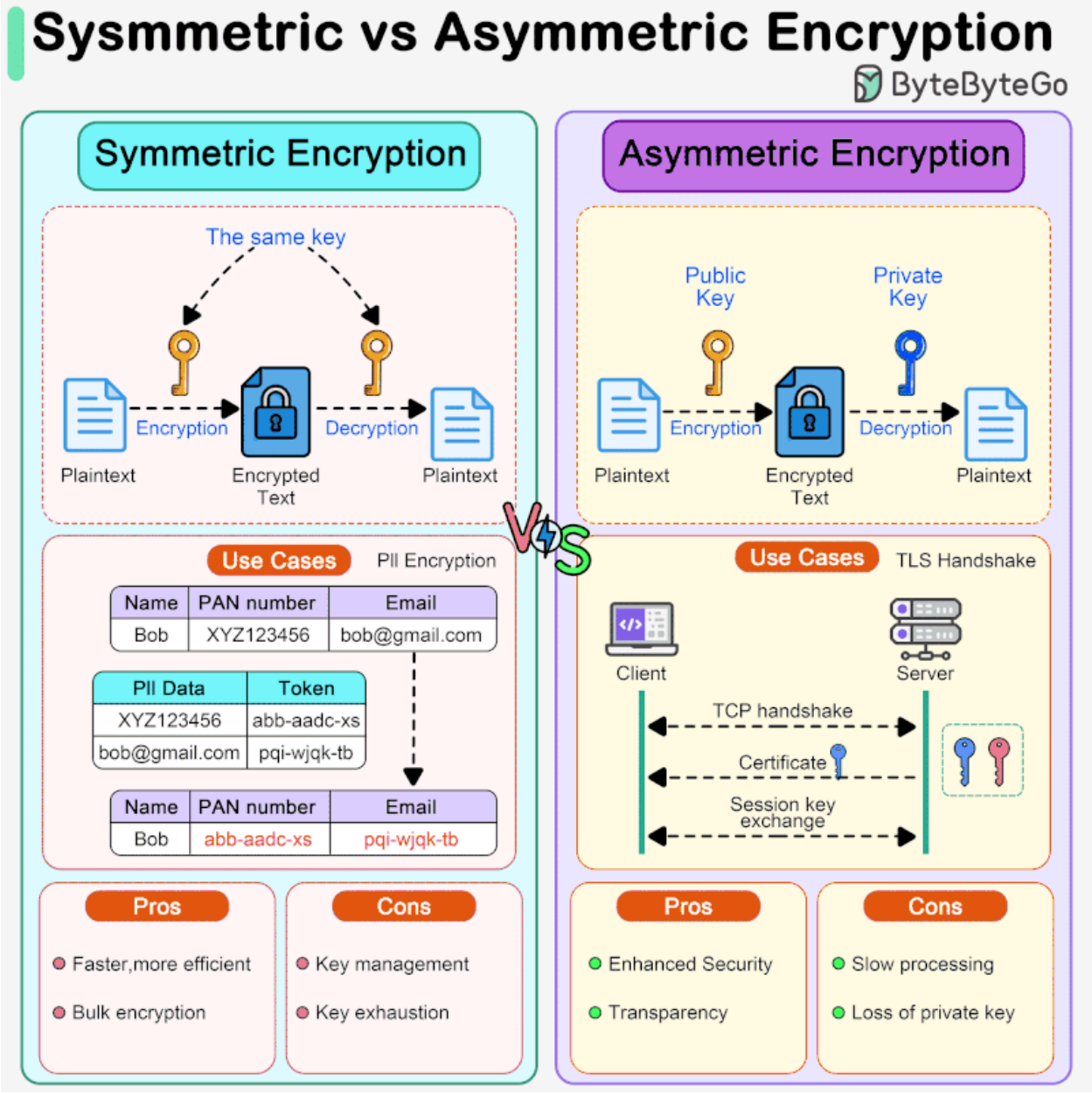
Dynamic Algorithms

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.

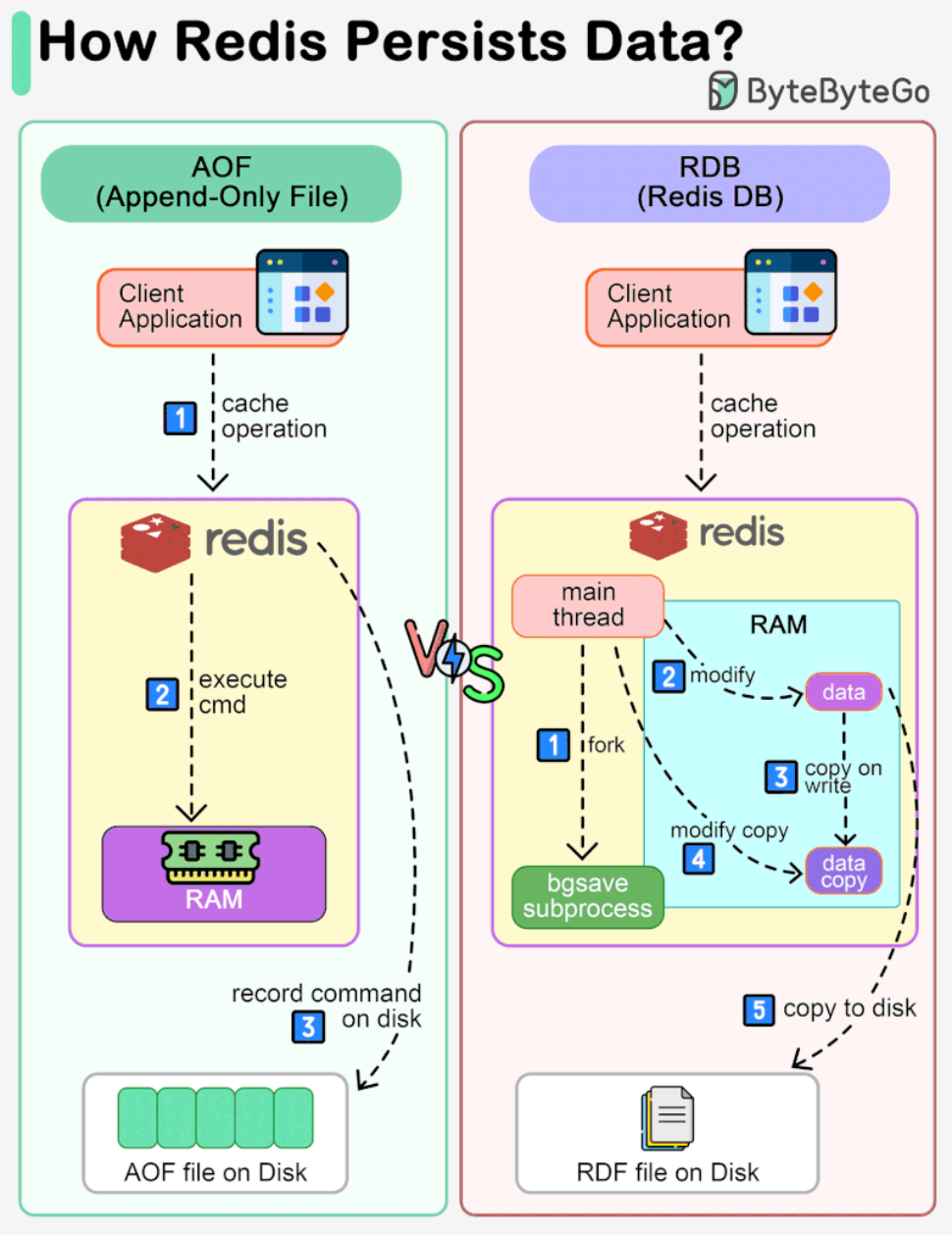
- Asymmetric encryption uses a pair of keys: a public key and a private key. The public key is freely distributed and used to encrypt data, while the private key is kept secret and used to decrypt the data. It is more secure than symmetric encryption because the private key is never shared. However, 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.

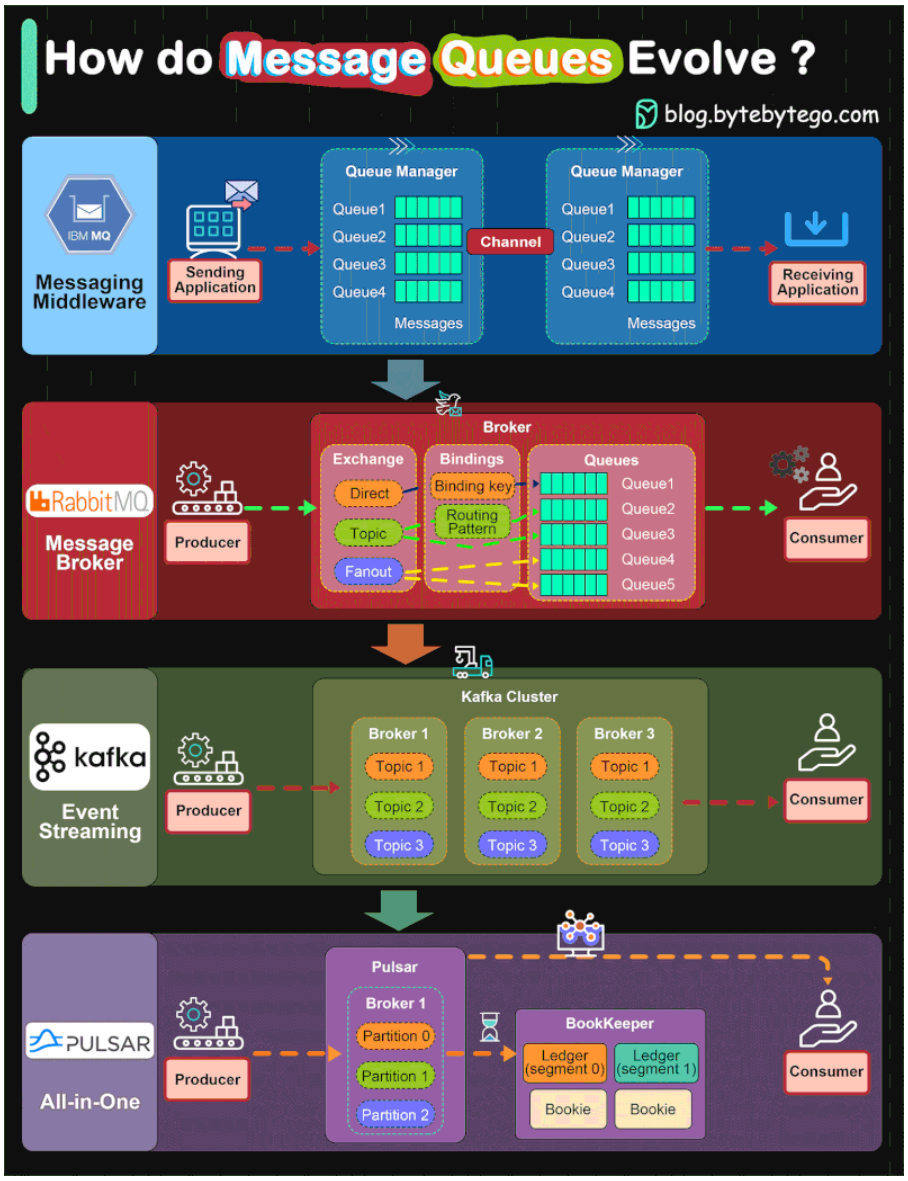
  Step 1: The main thread forks the‘ bgsave’ sub-process, which shares all the in-memory data of the main thread. ‘bgsave’ reads the data from the main thread and writes it to the RDB file.

  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.

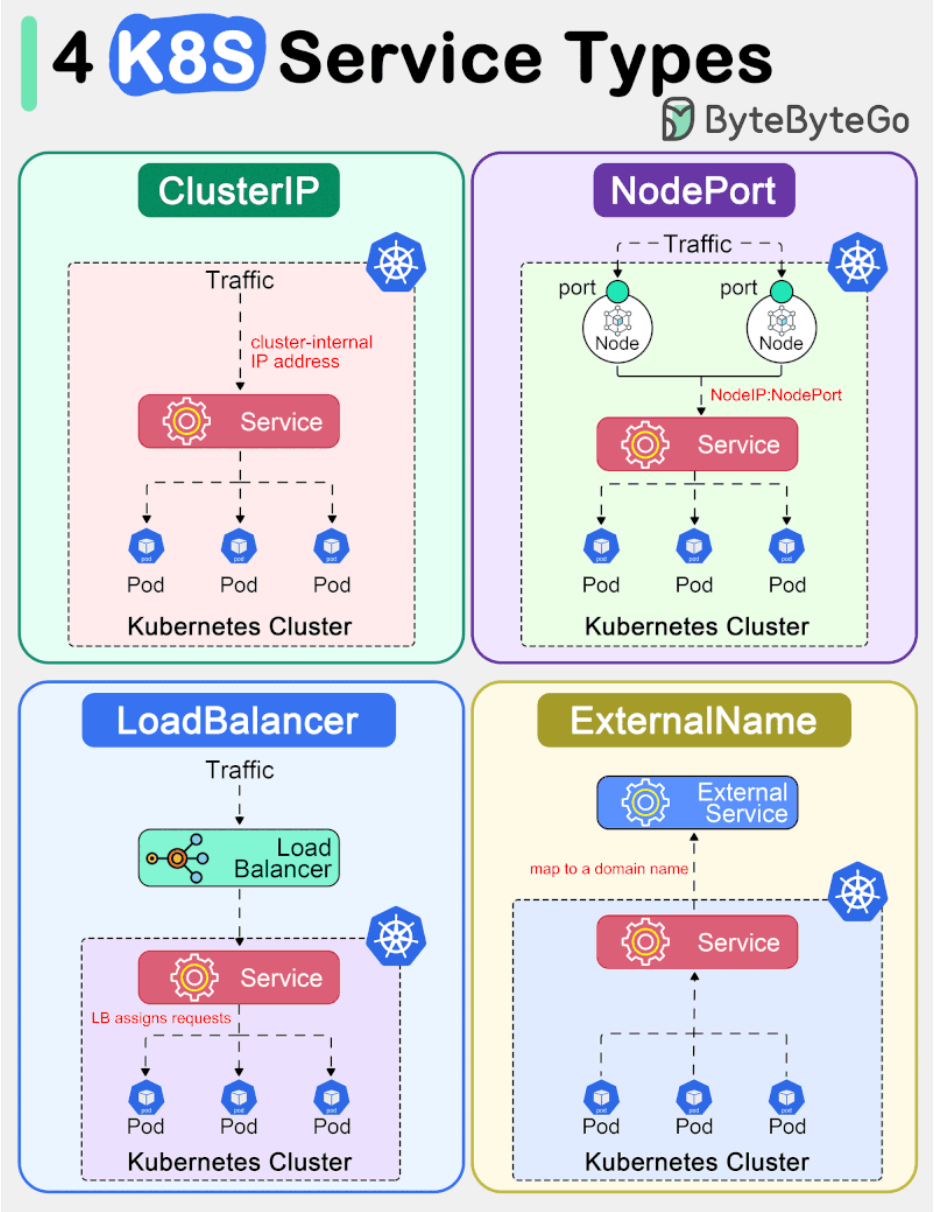
- Pulsar
  Pulsar, developed originally by Yahoo, is an all-in-one messaging and streaming platform. Compared with Kafka, Pulsar incorporates many useful features from other products and supports a wide range of capabilities. Also, Pulsar architecture is more cloud-native, providing better support for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

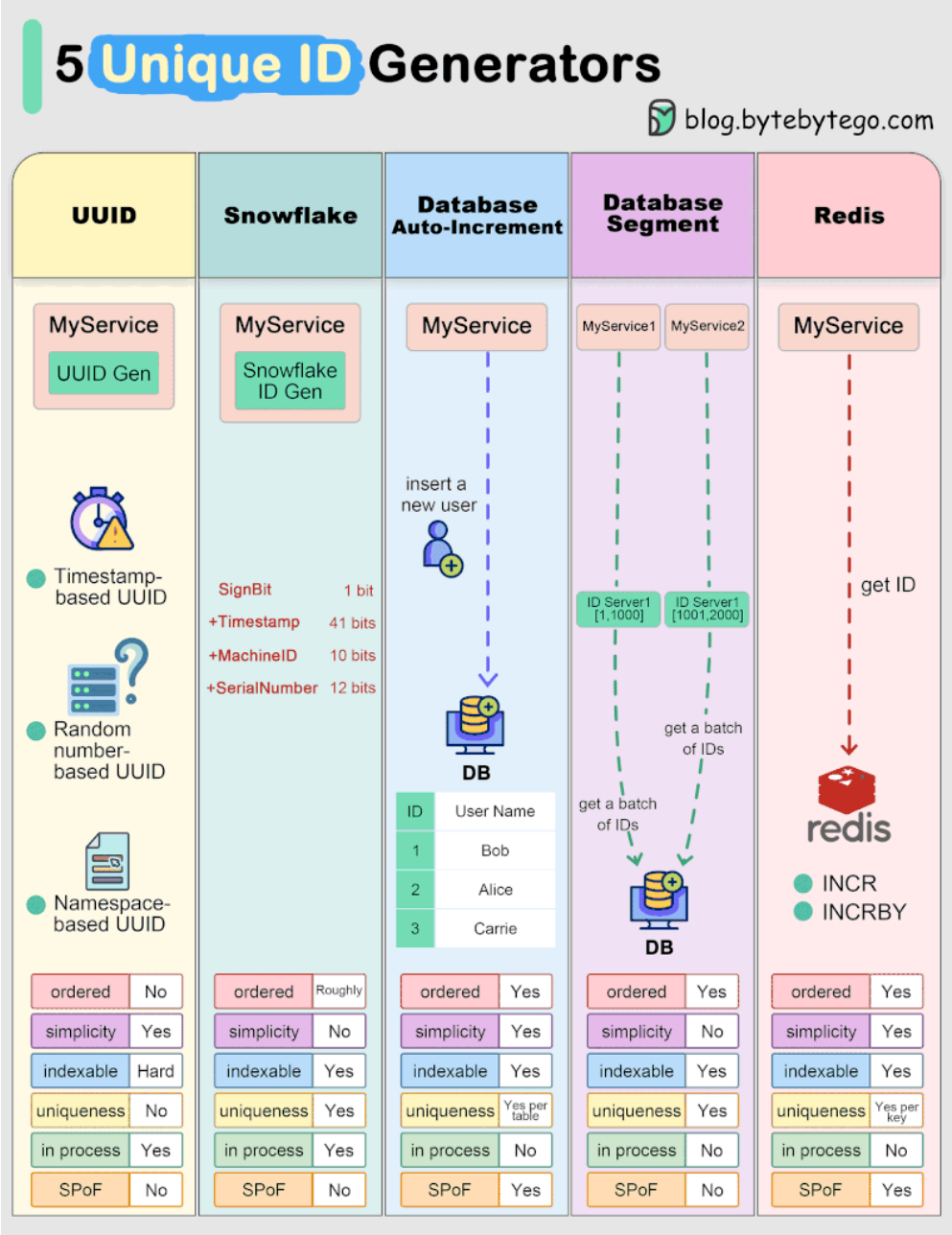
- NodePort
  This exposes the service outside of the cluster by adding a cluster-wide port on top of ClusterIP. We can request the service by NodeIP:NodePort.

- 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

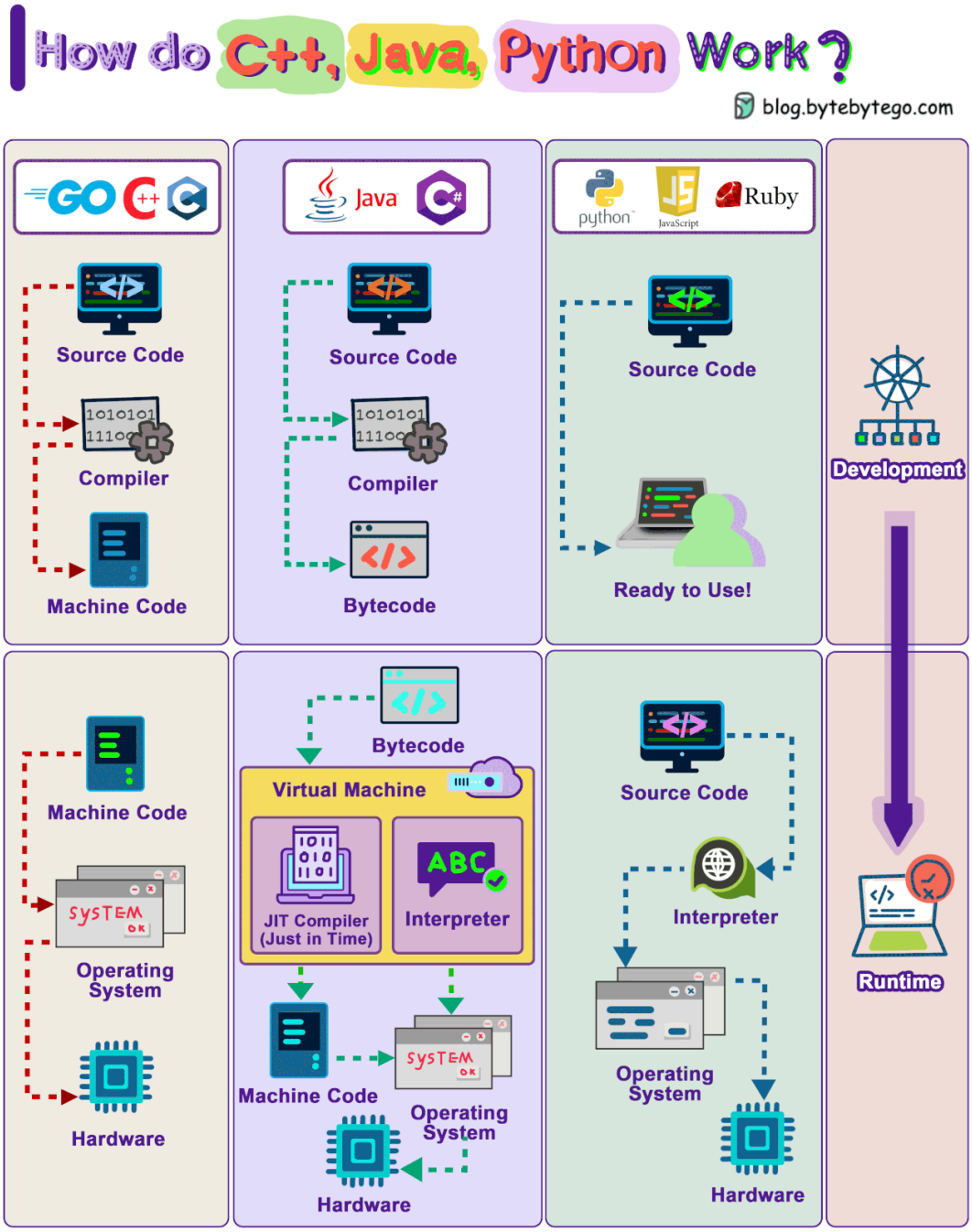
   We can also use Redis key-value pair to generate unique IDs. Redis stores data in memory, so this approach offers better performance than the database.

- 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#.

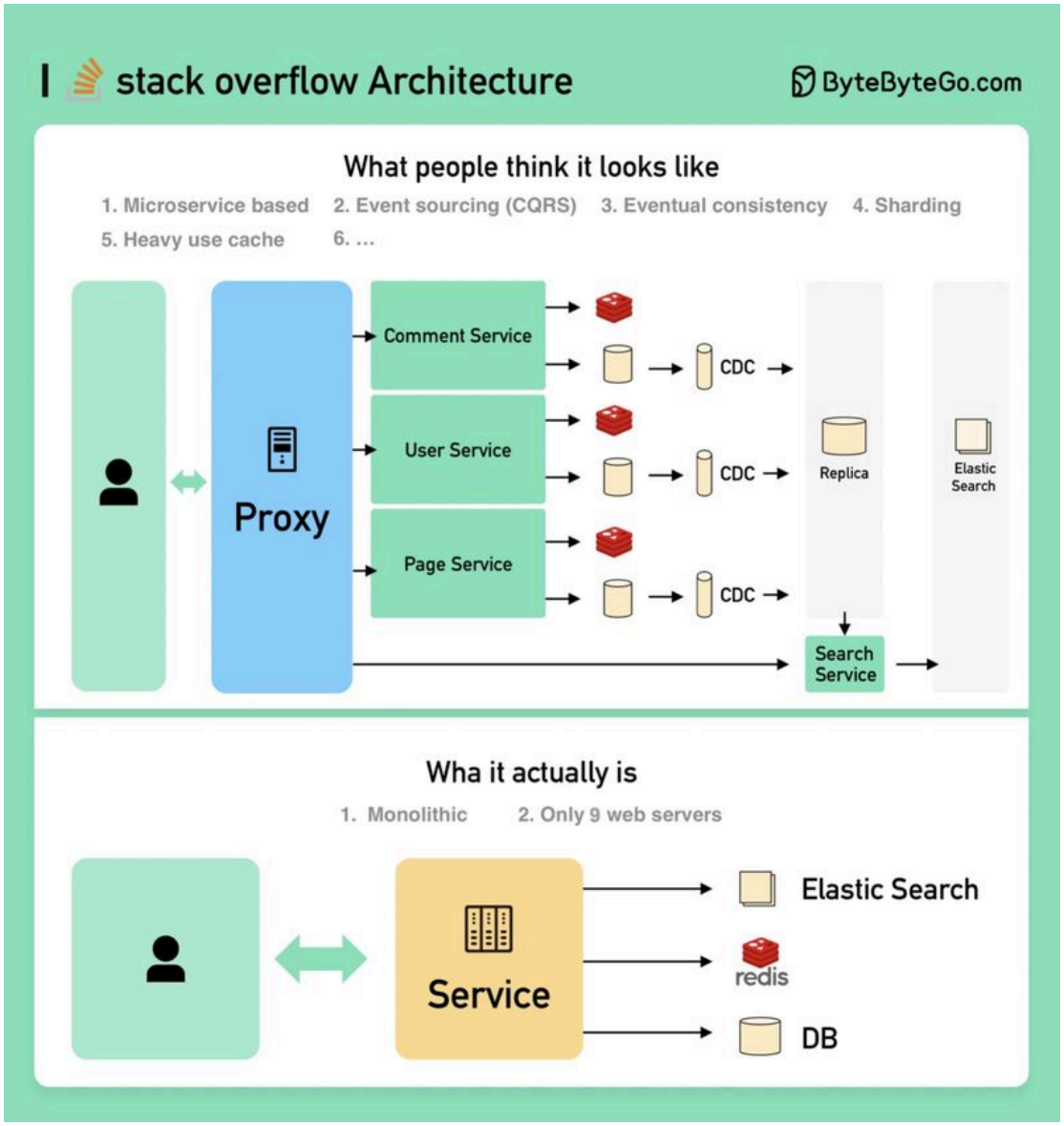
Languages that are interpreted don't undergo compilation. Instead, their code is processed by an interpreter during execution. Python, Javascript, and Ruby are some examples.

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.

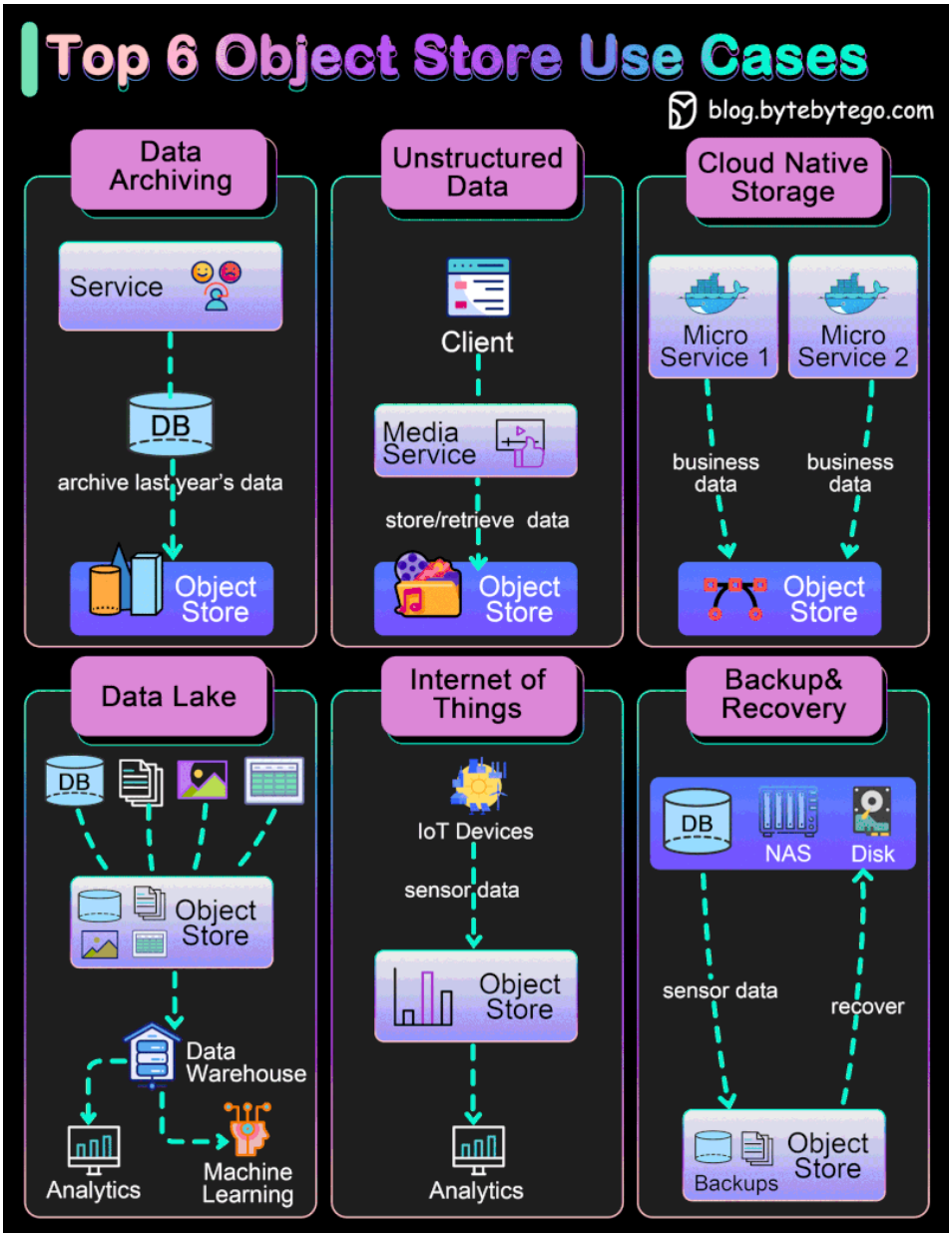
What it actually is

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)

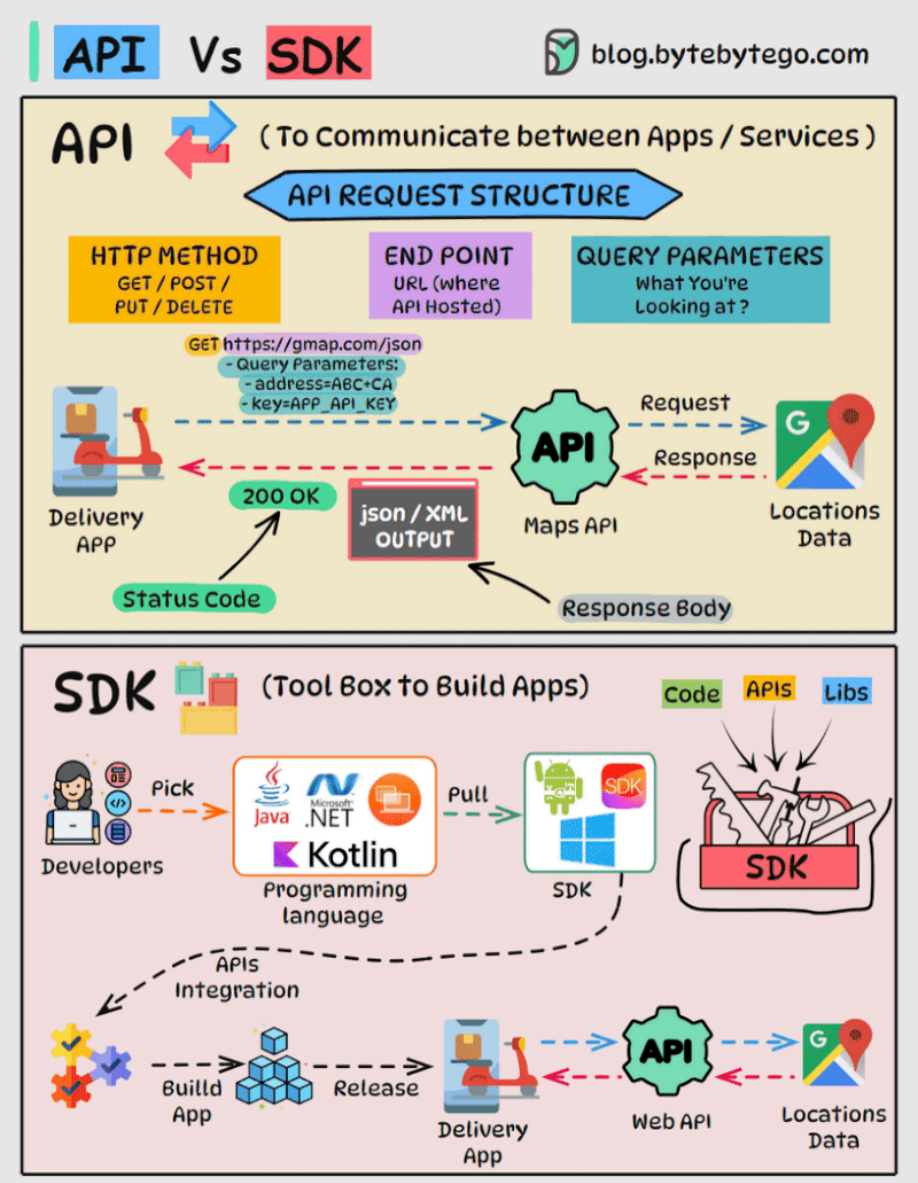
  IoT sensors produce all kinds of data. An object store can store this type of time series and later run analytics or AI algorithms on them. Major public cloud providers provide pipelines to ingest raw IoT data into the object store.

- 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.

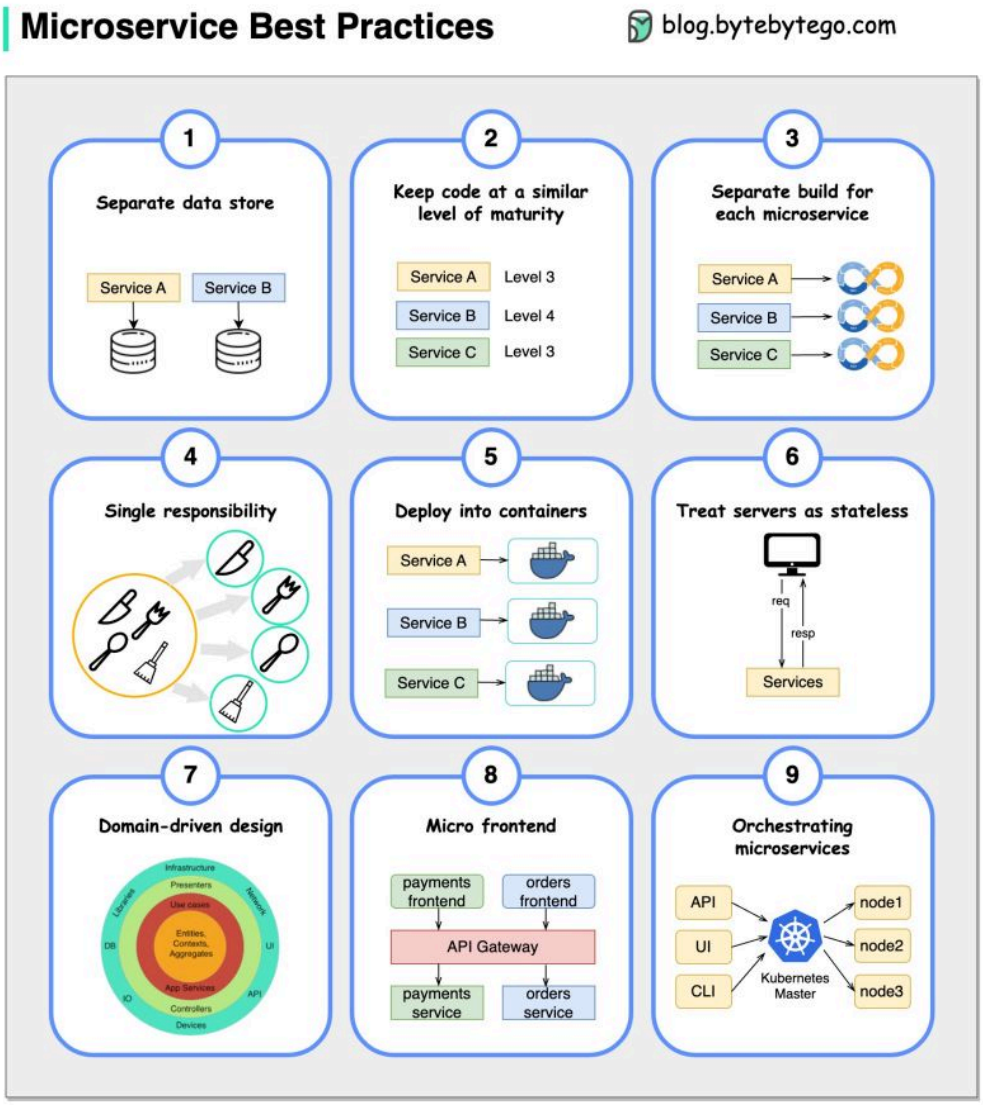
1. Offers higher-level abstractions, simplifying development for a specific platform.
2. Tailored to specific platforms or frameworks, ensuring compatibility and optimal performance on that platform.
3. Offer access to advanced features and capabilities specific to the platform, which might be otherwise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

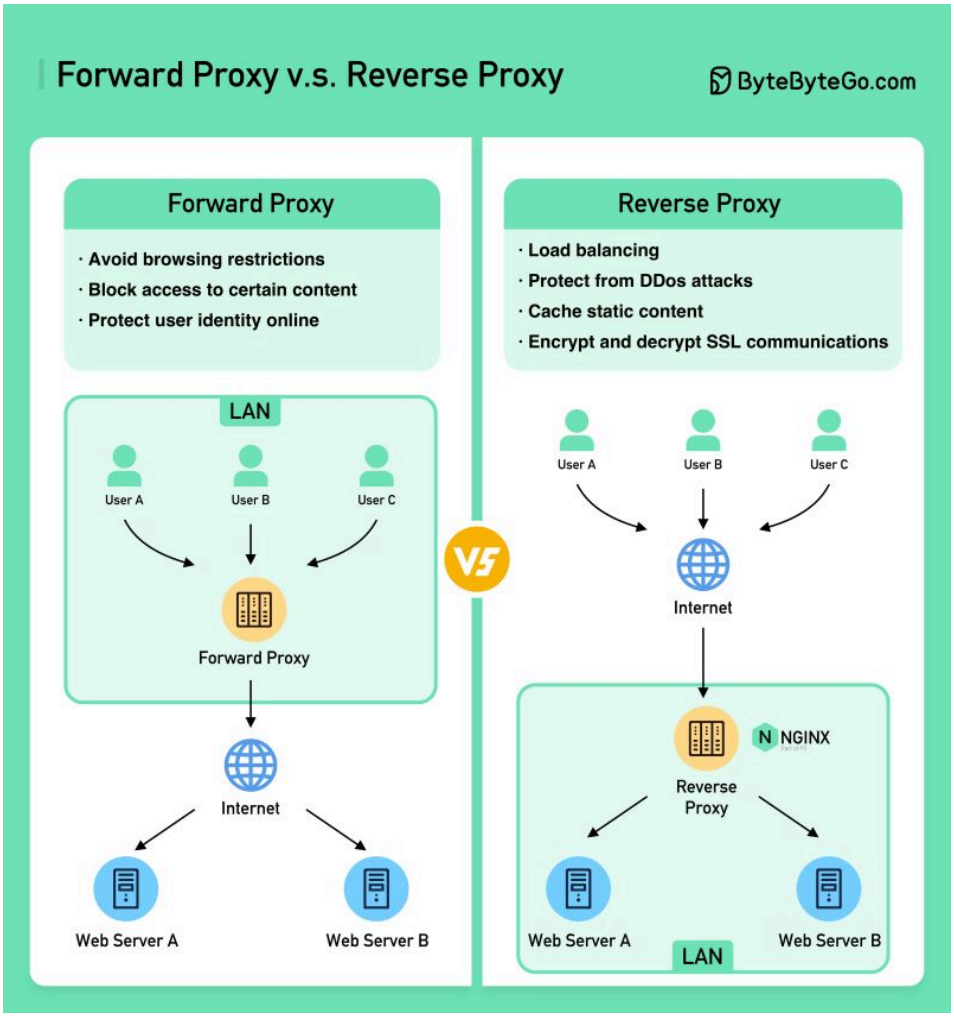
A picture is worth a thousand words: 9 best practices for developing microservices



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



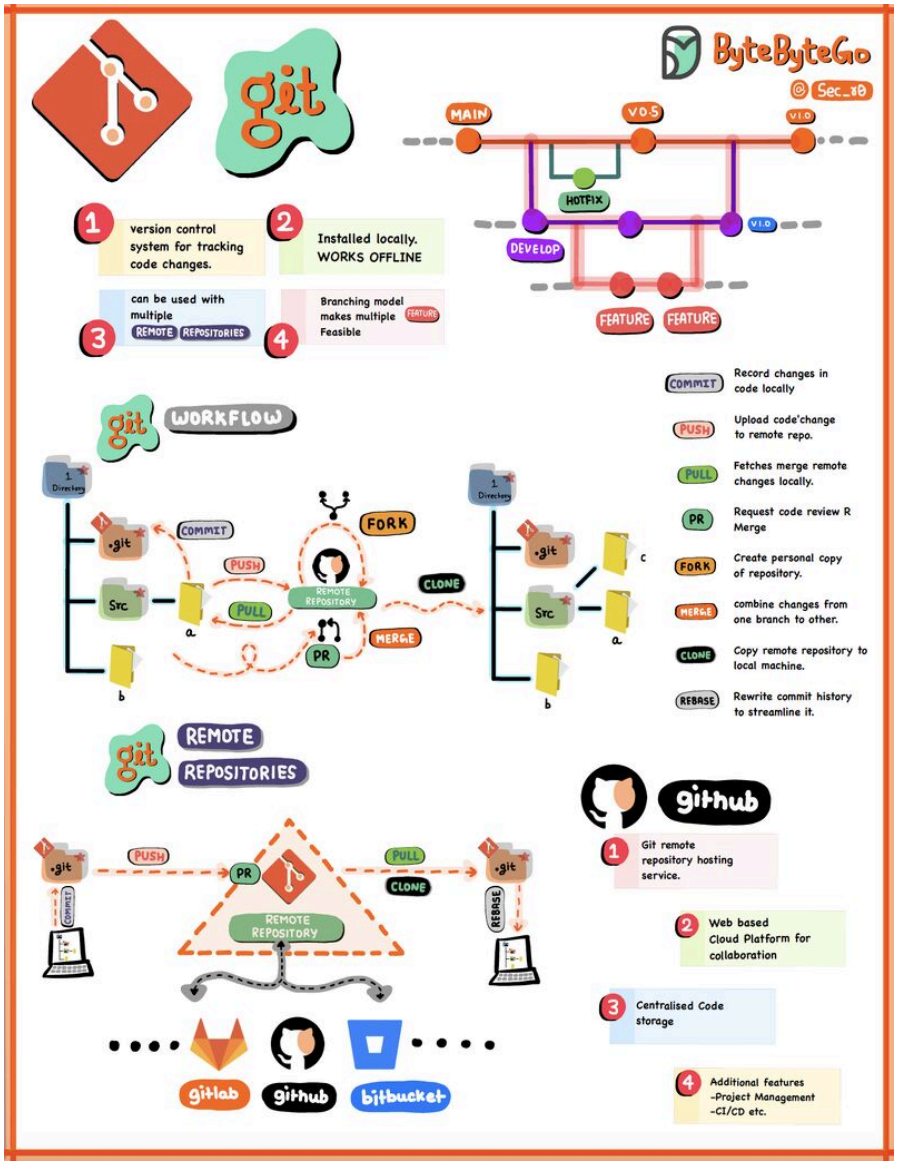
A forward proxy is a server that sits between user devices and the internet. A forward proxy is commonly used for:

- 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.

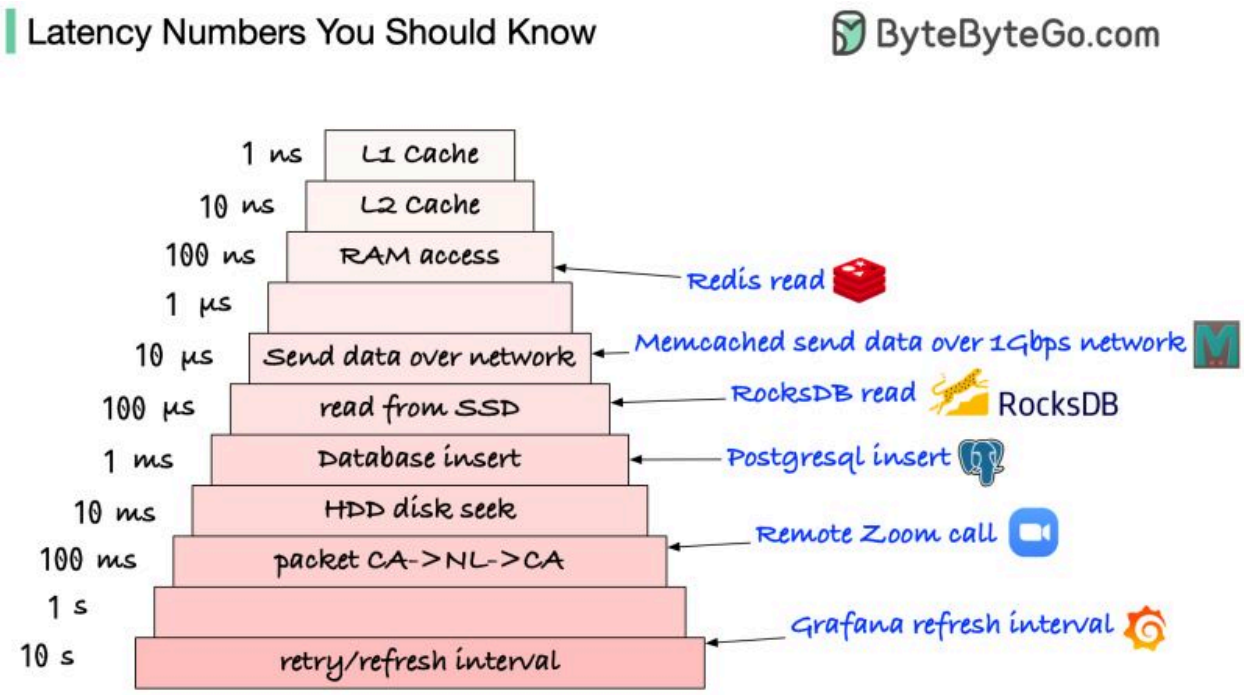
First, meet Git, a fundamental tool for developers. It operates locally, allowing you to track changes in your code, much like taking snapshots of your project's progress. This makes collaboration with your team a breeze, even when you're working on the same project.

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

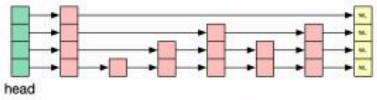
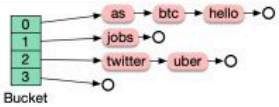
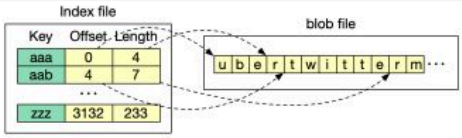
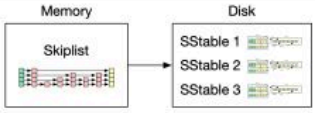
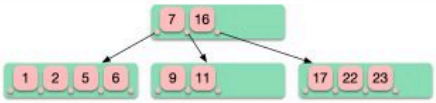
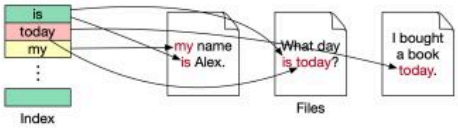
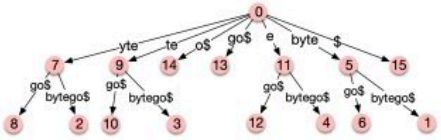
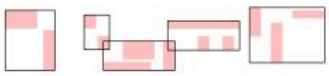
1 us = 10^-6 seconds = 1,000 ns

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

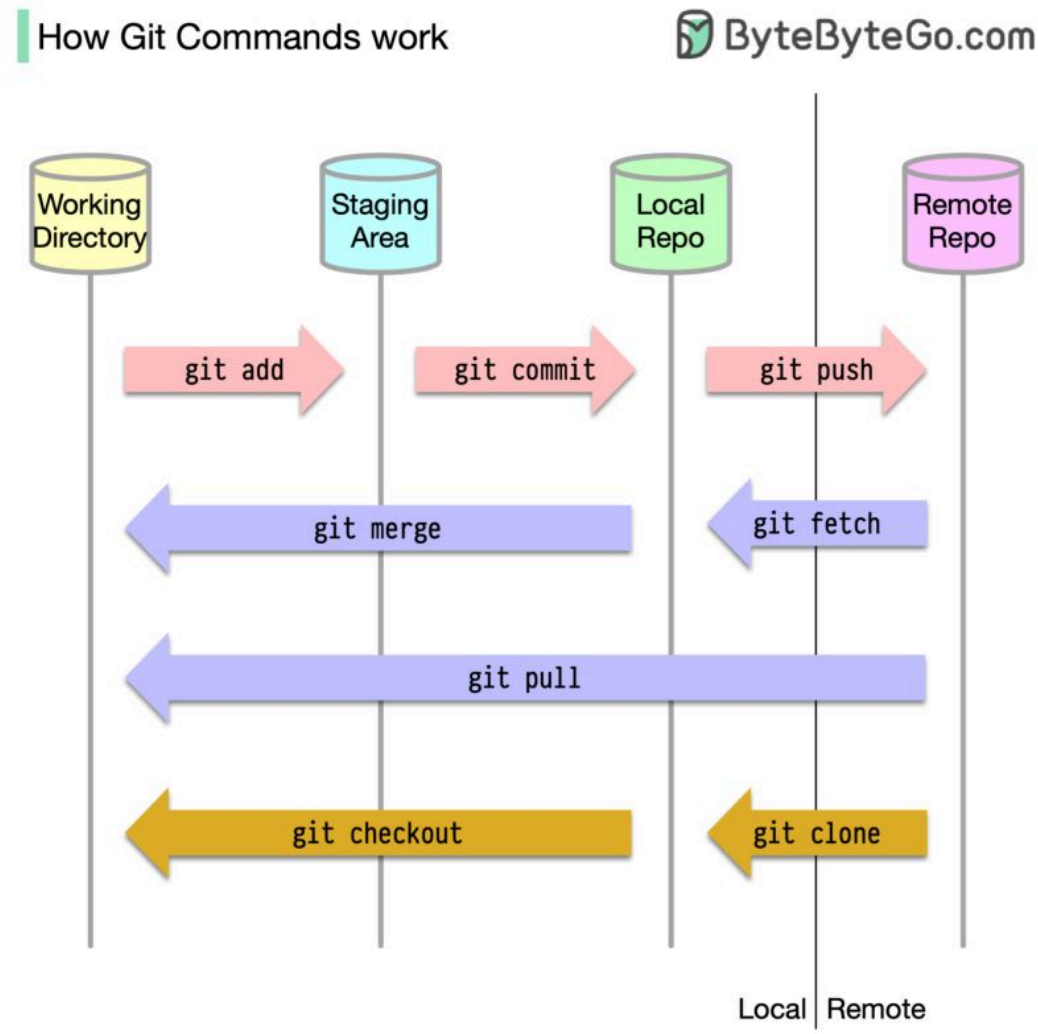
This is not an exhaustive list of all database index types.

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:

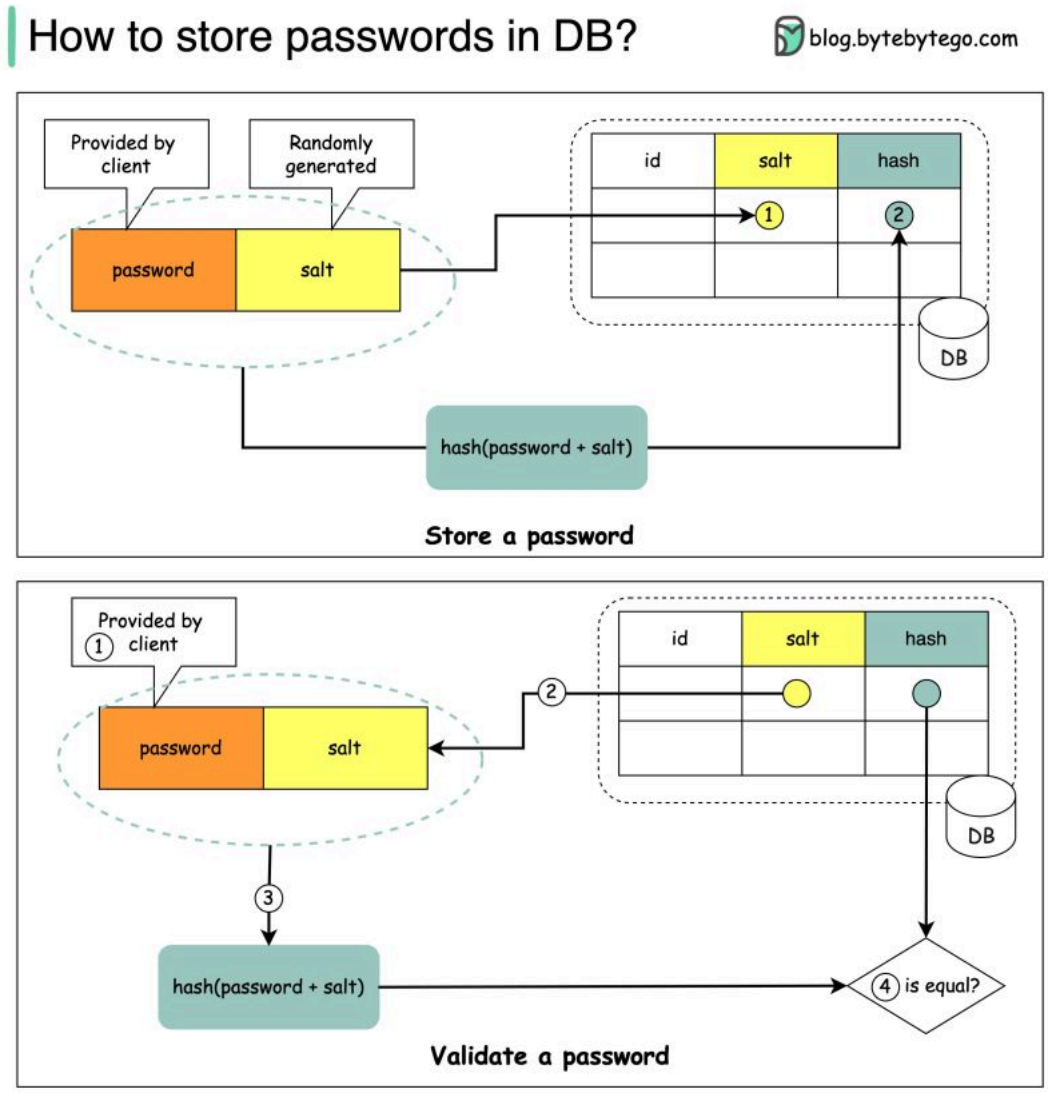
- Working directory: where we edit files
- Staging area: a temporary location where files are kept for the next commit
- Local repository: contains the code that has been committed
- Remote repository: the remote server that stores the code

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*).

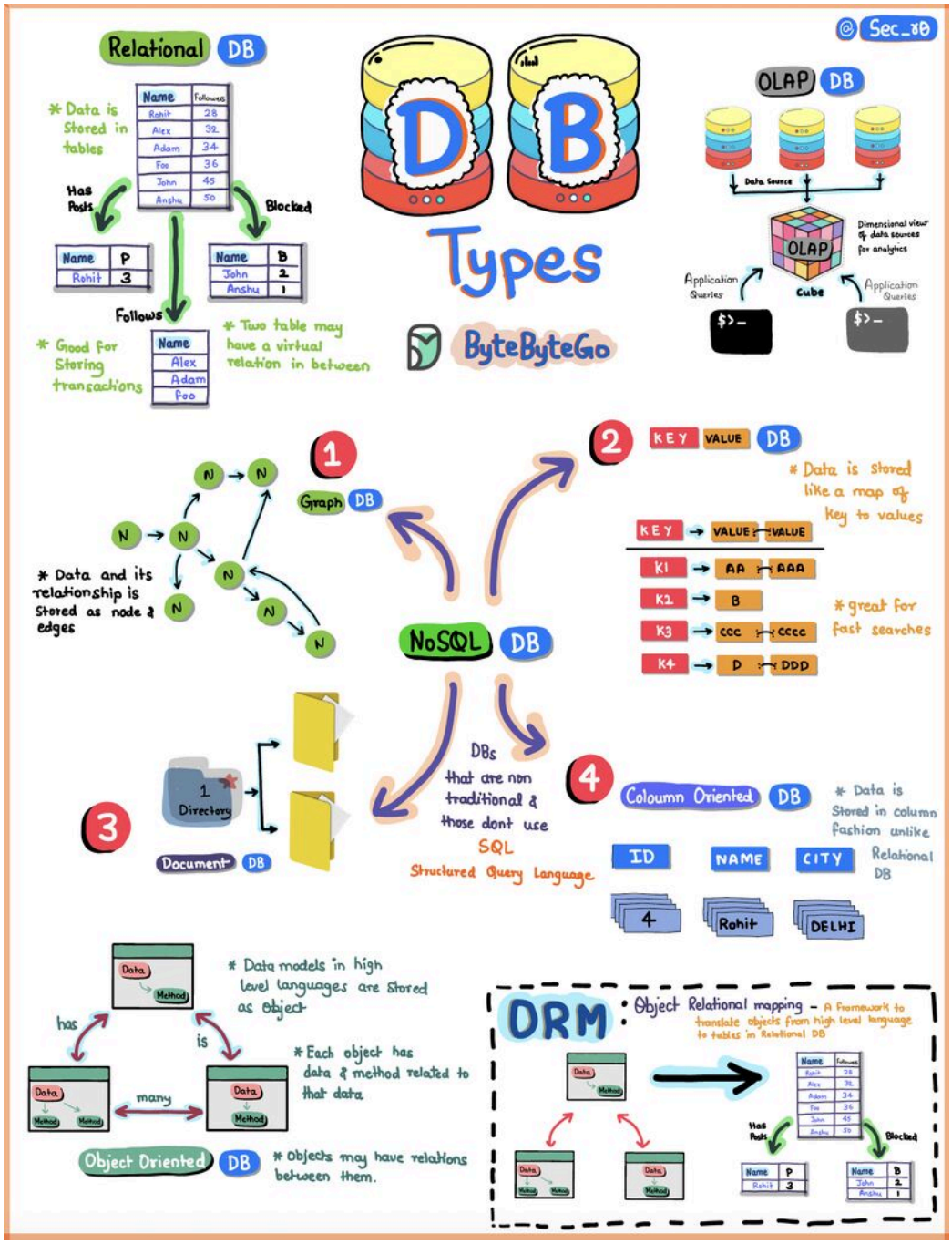
How to validate a password?
To validate a password, it can go through the following process:

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.

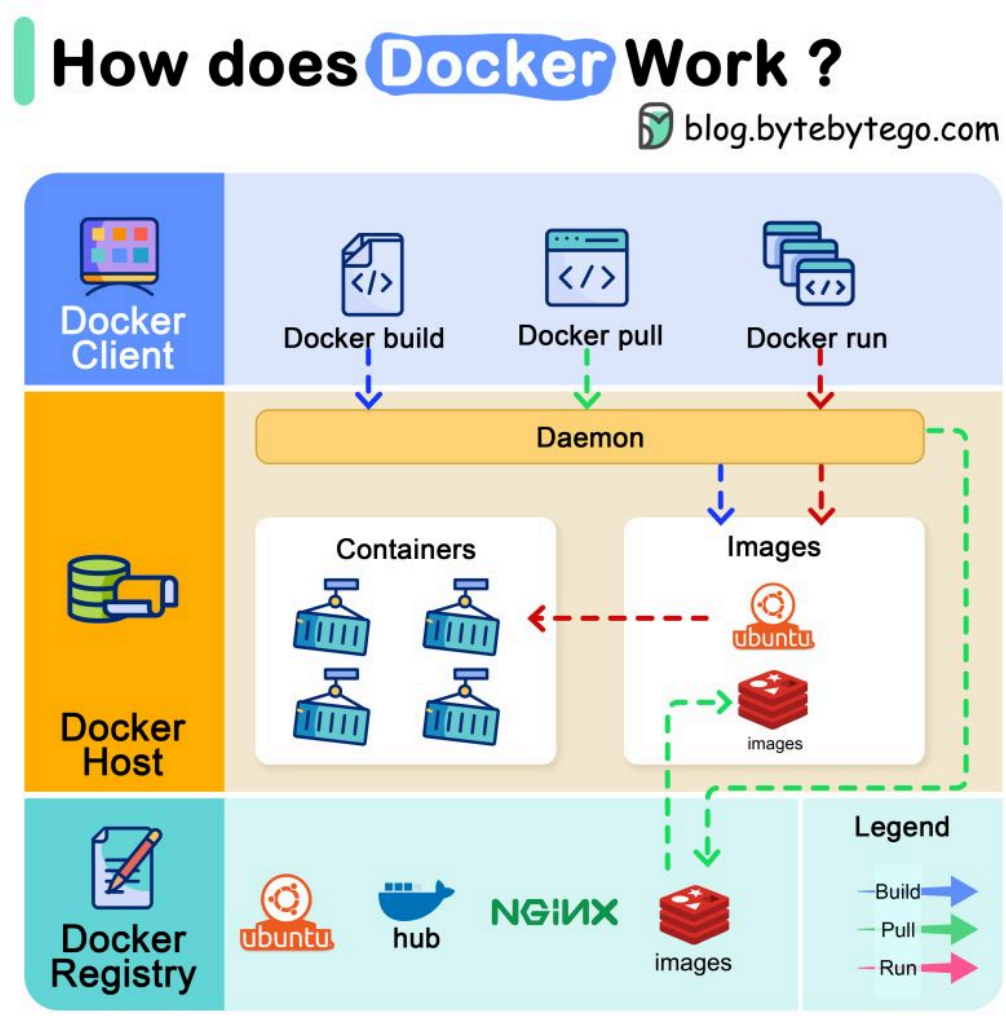
NoSQL DBs: These rebels have their own cool club, saying "No" to traditional SQL ways. NoSQL databases come in four exciting flavors:

- 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

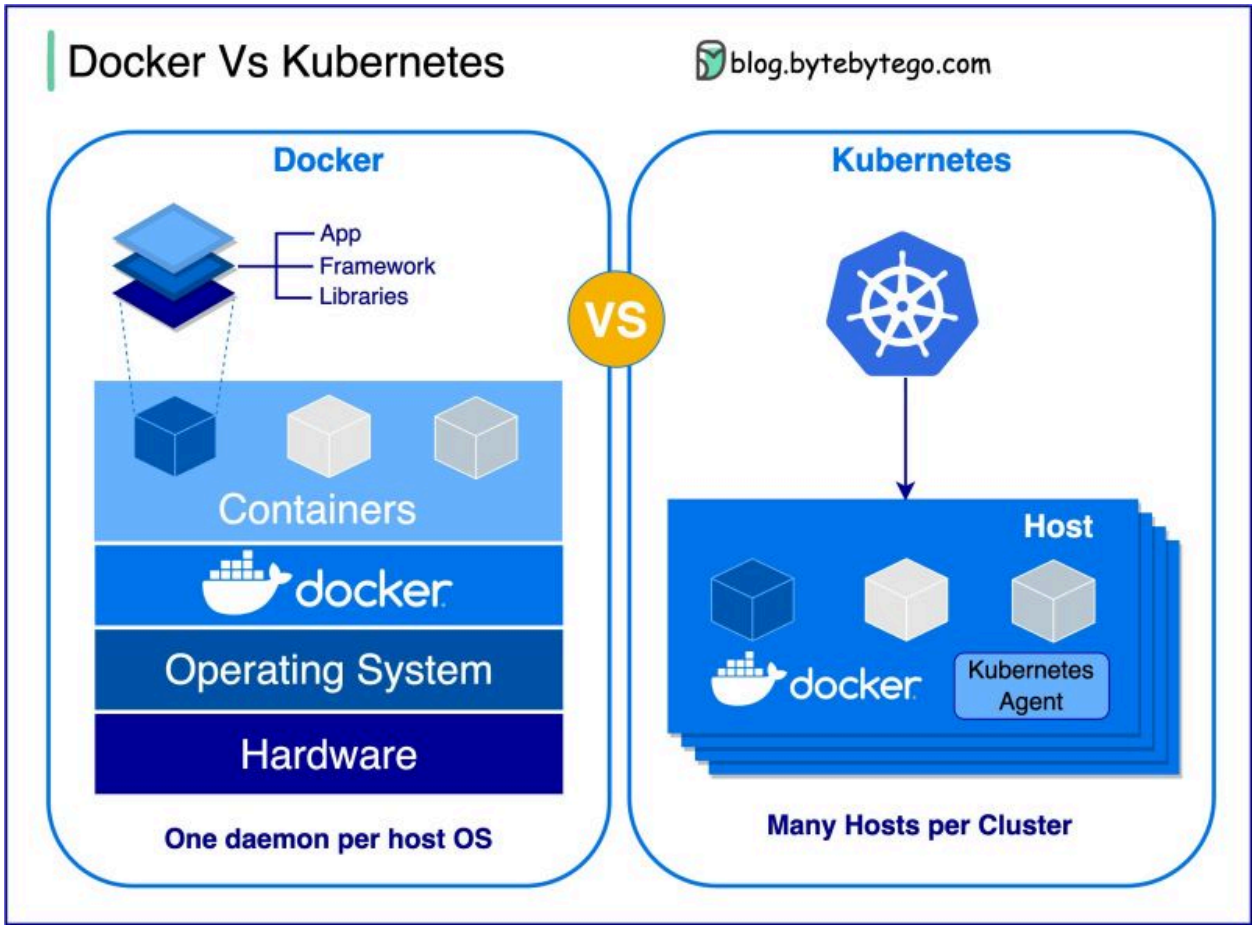
This is where Docker images are stored. Docker Hub, for instance, is a widely-used public registry.

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.

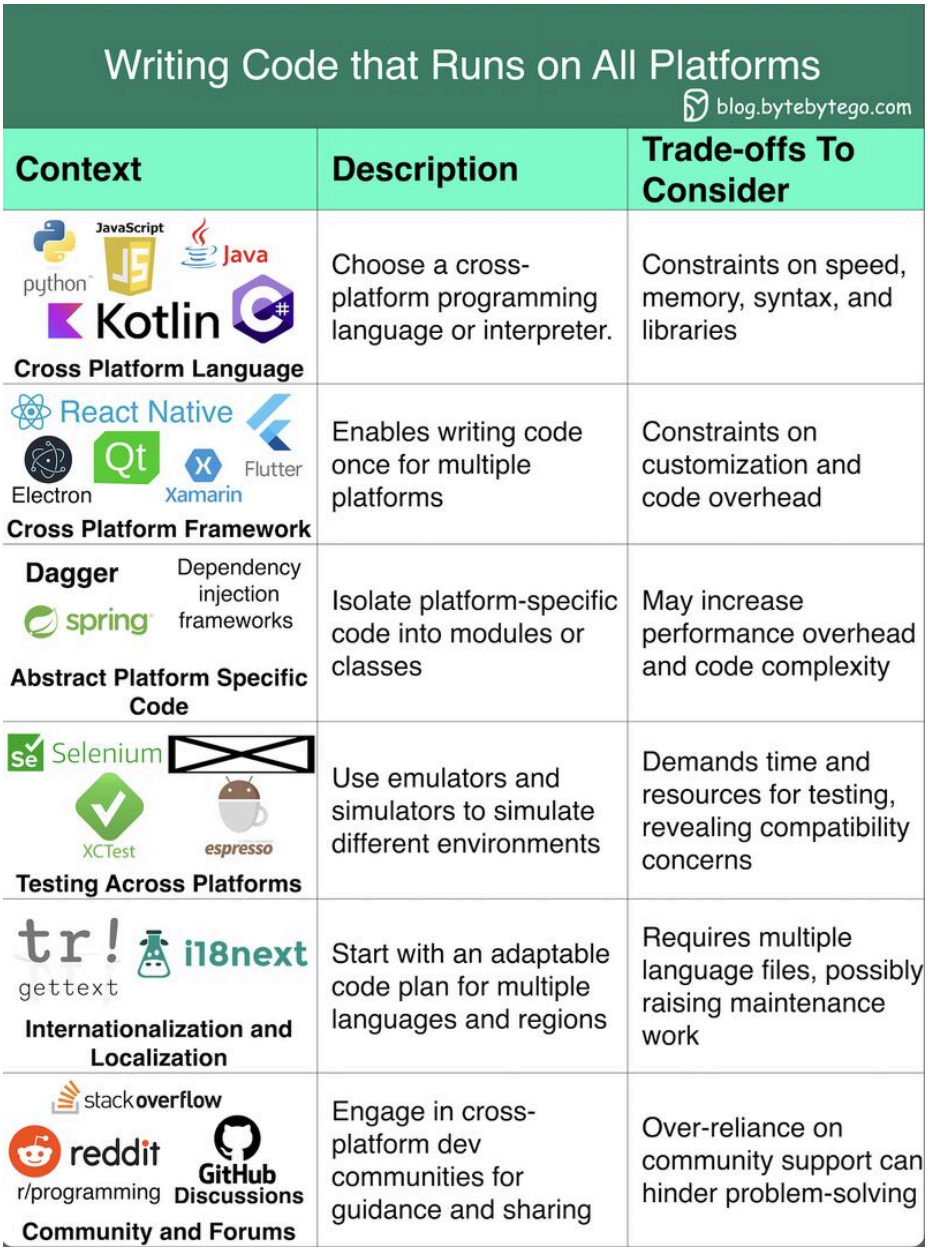
In short, Docker focuses on containerization and running containers on individual hosts, while Kubernetes specializes in managing and orchestrating containers at scale across a cluster of hosts.

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.

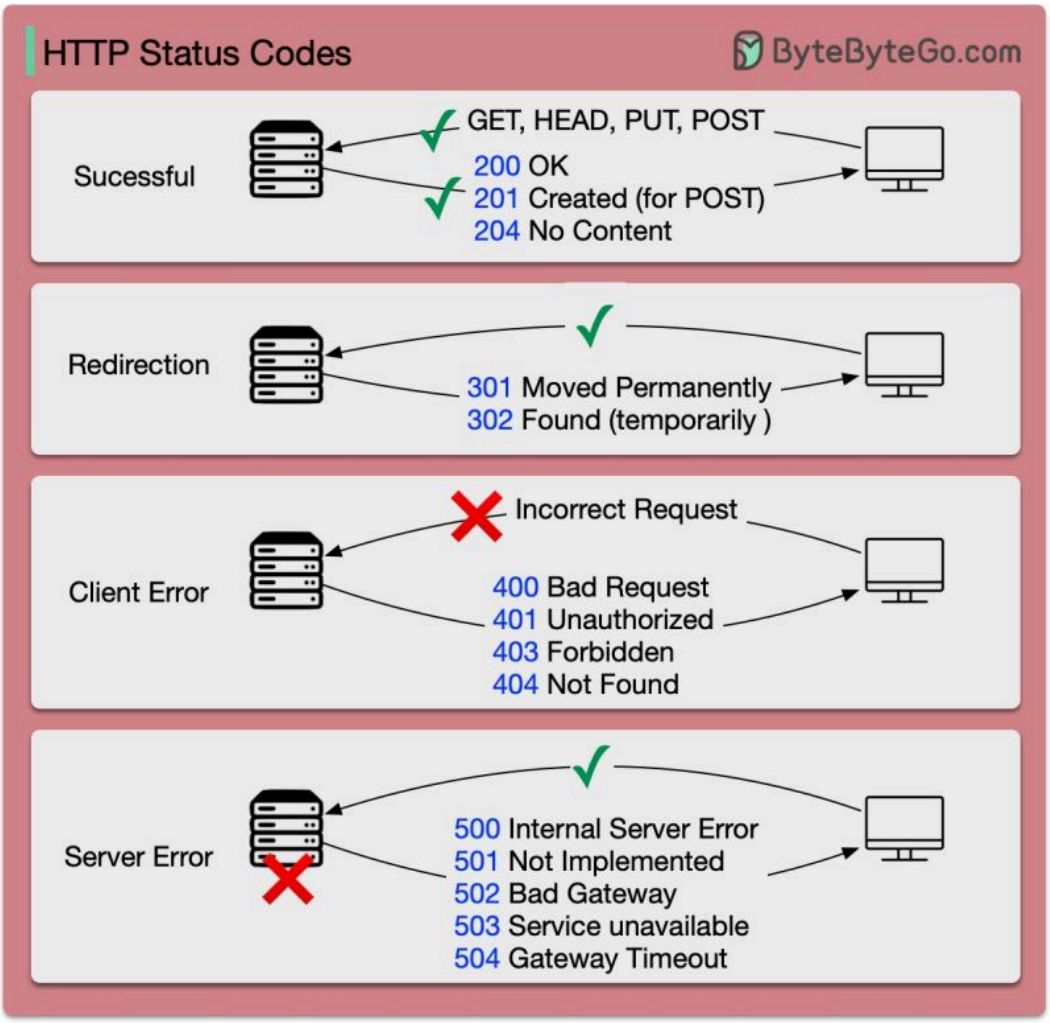
It reduces redundancy, simplifies maintenance, ensures consistency, boosting satisfaction and market reach.

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)

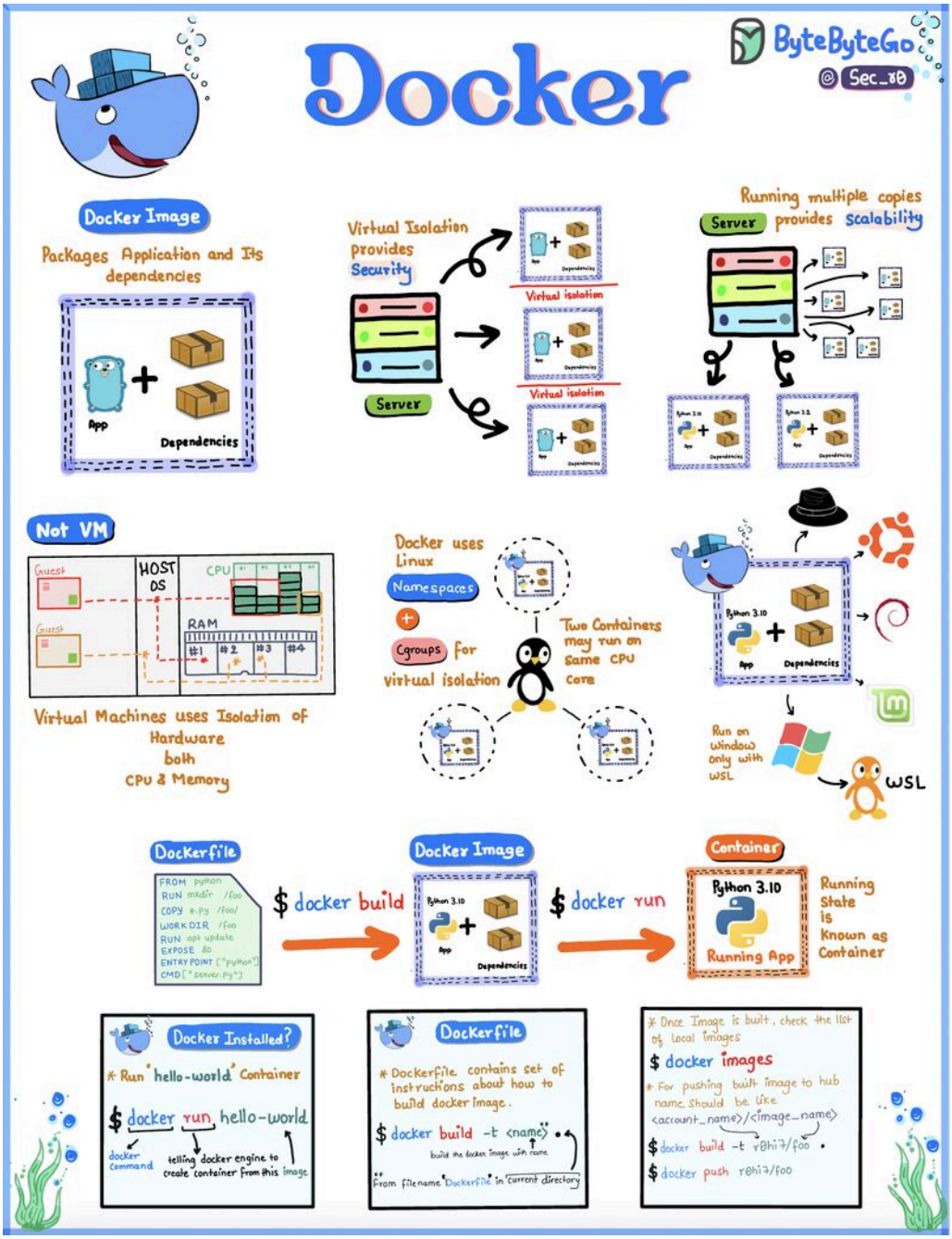
Server Error (500-599)

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:

Over to you: HTTP status code 401 is for Unauthorized. Can you explain the difference between authentication and authorization, and which one does code 401 check for?

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

Docker 101: Streamlining App Deployment



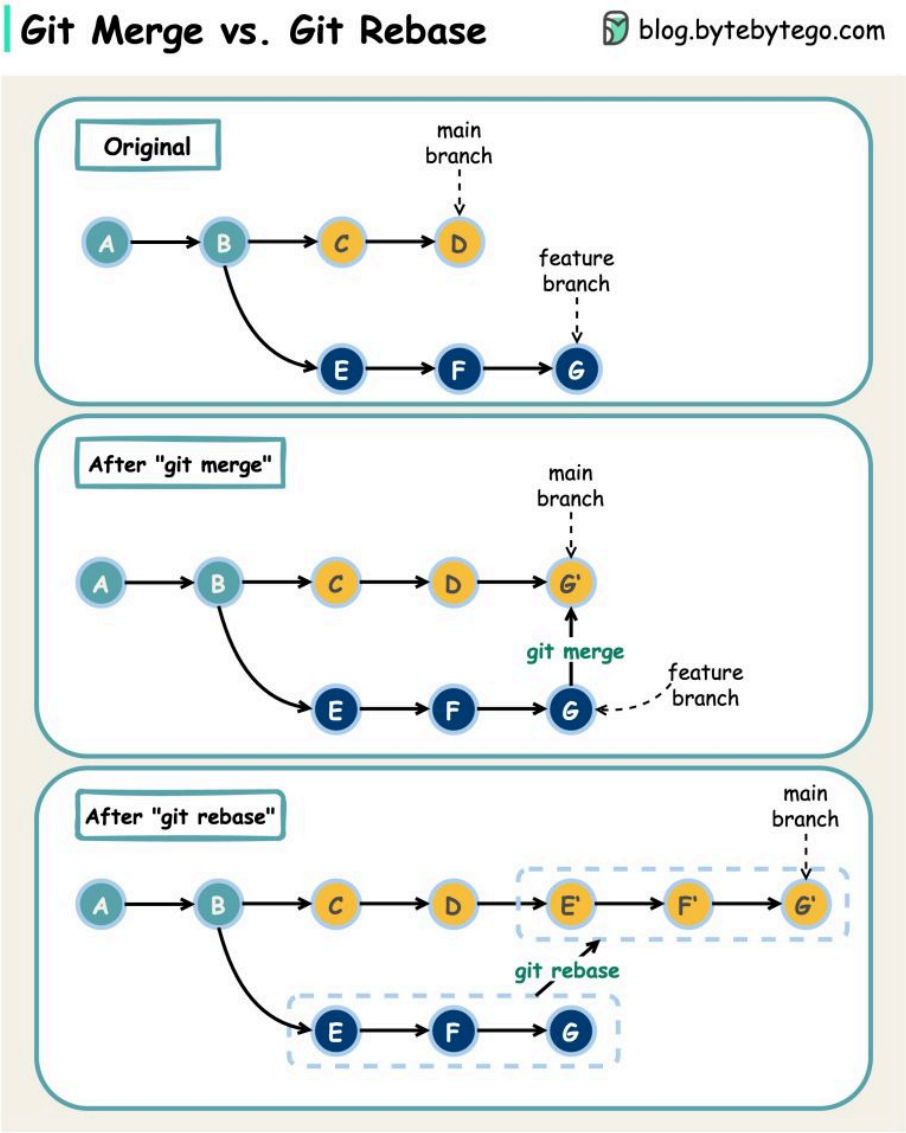
Fed up with the "it works on my machine" dilemma? Docker could be your salvation!

Docker revolutionizes software development and deployment. Explore the essentials:

1. Bundle Everything: Docker packs your app and its dependencies into a portable container – code, runtime, tools, libraries, and settings – a tidy, self-contained package.

2. Virtual Isolation: Containers offer packaging and isolation. Run diverse apps with different settings on a single host without conflicts, thanks to Linux namespaces and cgroups.
3. Not VMs: Unlike resource-heavy VMs, Docker containers share the host OS kernel, delivering speed and efficiency. No VM overhead, just rapid starts and easy management. ⚡
4. Windows Compatibility: Docker, rooted in Linux, works on Windows too. Docker Desktop for Windows uses a Linux-based VM, enabling containerization for Windows apps.

Git Merge vs. Rebase vs. Squash Commit



What are the differences?

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

Git Merge

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

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

Git Rebase

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

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

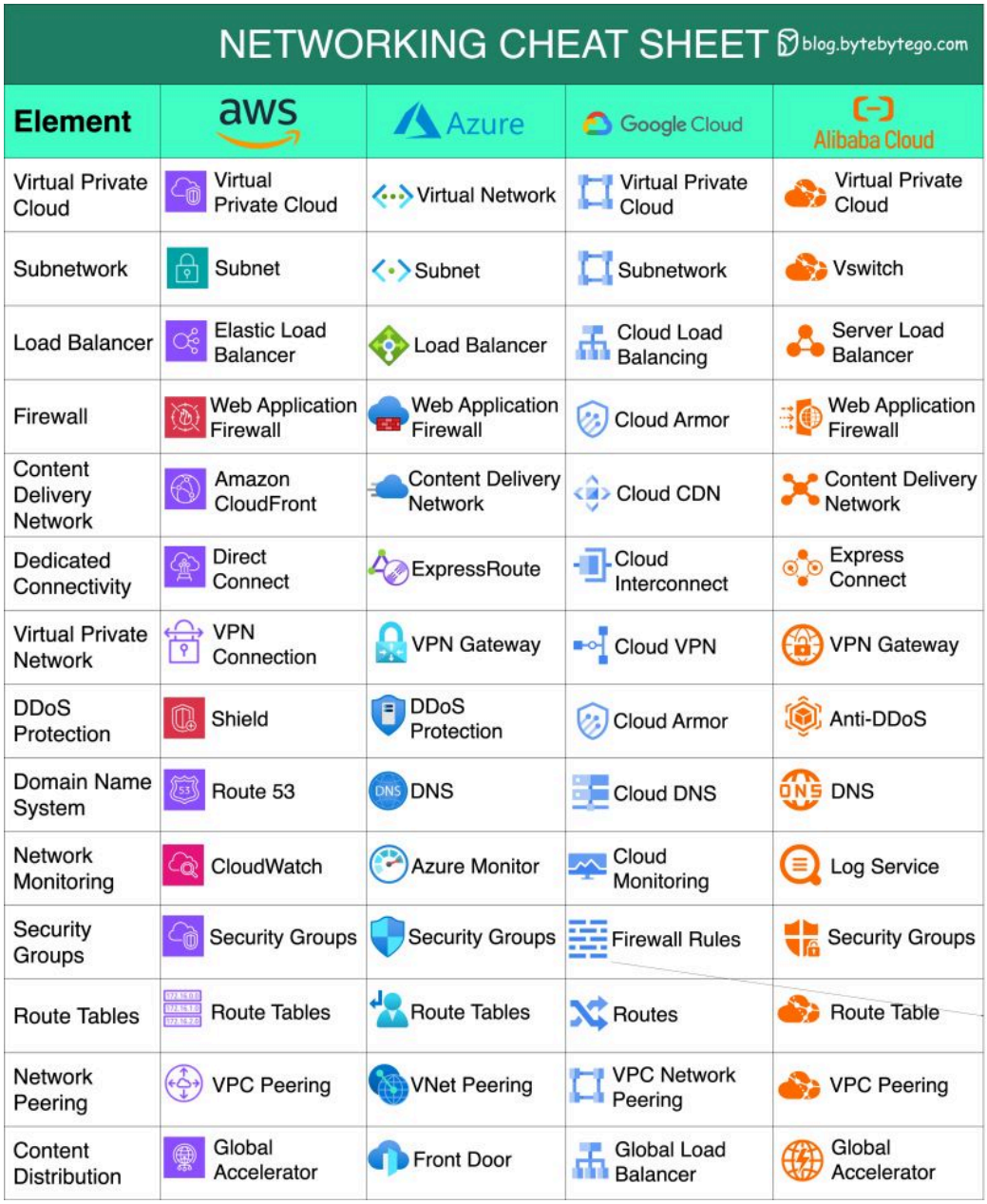
Rebase can be dangerous if "the golden rule of git rebase" is not followed.

The Golden Rule of Git Rebase

Never use it on public branches!

Cloud Network Components Cheat Sheet

Network components form the backbone of cloud infrastructure, enabling connectivity, scalability, and functionality in cloud services.



Element	aws	Azure	Google Cloud	Alibaba Cloud
Virtual Private Cloud	Virtual Private Cloud	Virtual Network	Virtual Private Cloud	Virtual Private Cloud
Subnetwork	Subnet	Subnet	Subnetwork	Vswitch
Load Balancer	Elastic Load Balancer	Load Balancer	Cloud Load Balancing	Server Load Balancer
Firewall	Web Application Firewall	Web Application Firewall	Cloud Armor	Web Application Firewall
Content Delivery Network	Amazon CloudFront	Content Delivery Network	Cloud CDN	Content Delivery Network
Dedicated Connectivity	Direct Connect	ExpressRoute	Cloud Interconnect	Express Connect
Virtual Private Network	VPN Connection	VPN Gateway	Cloud VPN	VPN Gateway
DDoS Protection	Shield	DDoS Protection	Cloud Armor	Anti-DDoS
Domain Name System	Route 53	DNS	Cloud DNS	DNS
Network Monitoring	CloudWatch	Azure Monitor	Cloud Monitoring	Log Service
Security Groups	Security Groups	Security Groups	Firewall Rules	Security Groups
Route Tables	Route Tables	Route Tables	Routes	Route Table
Network Peering	VPC Peering	VNet Peering	VPC Network Peering	VPC Peering
Content Distribution	Global Accelerator	Front Door	Global Load Balancer	Global Accelerator

These components include routers, load balancers, and firewalls, which ensure data flows efficiently and securely between servers and clients.

Additionally, Content Delivery Networks (CDNs) optimize content delivery by caching data at edge locations, reducing latency and improving user experience.

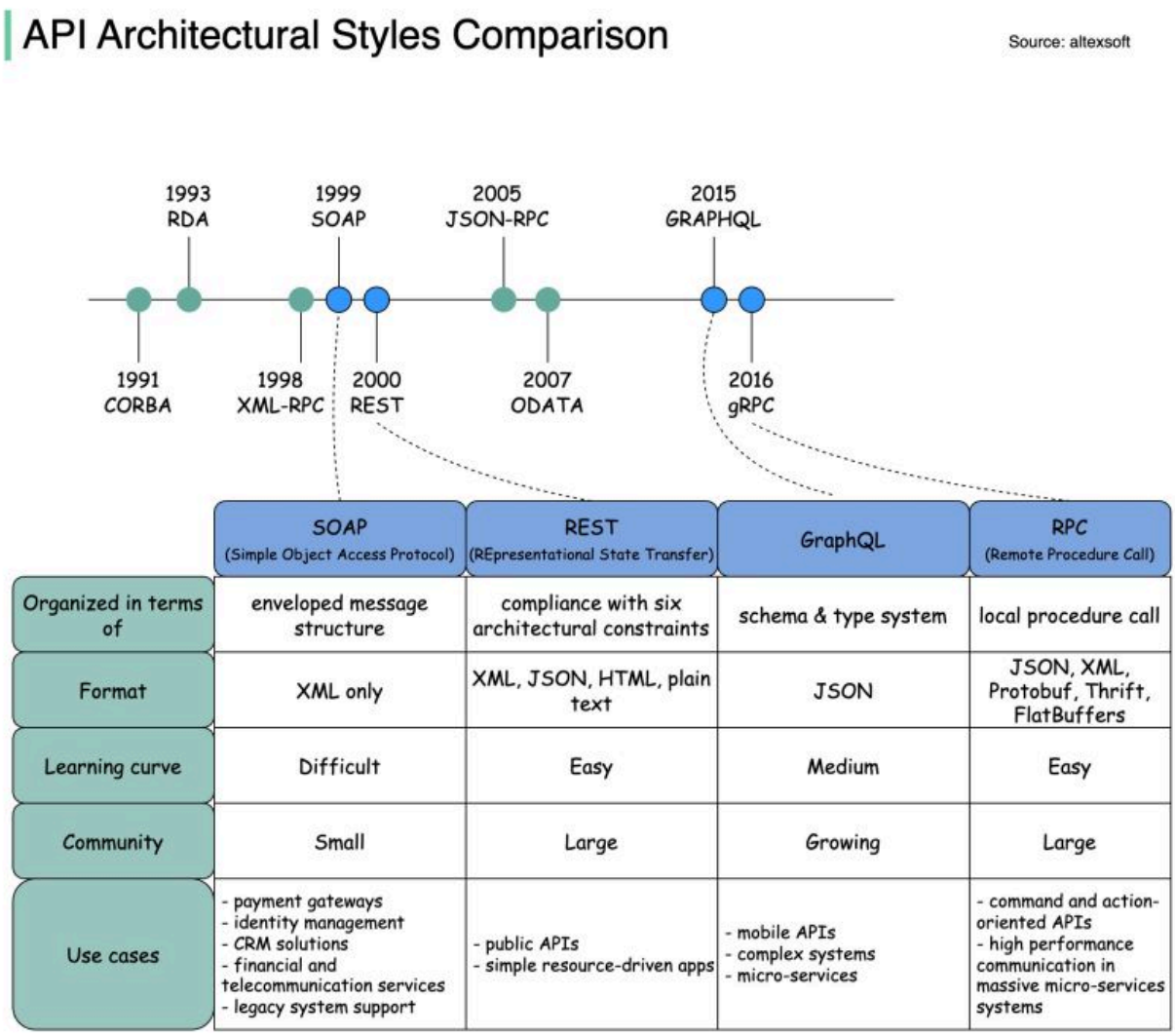
In essence, these network elements work together to create a robust and responsive cloud ecosystem that underpins modern digital services and applications.

This cheat sheet offers a concise yet comprehensive comparison of key network elements across the four major cloud providers.

Over to you: How did you tackle the complexity of configuring and managing these network components?

SOAP vs REST vs GraphQL vs RPC

The diagram below illustrates the API timeline and API styles comparison.

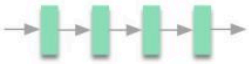
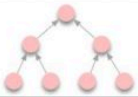
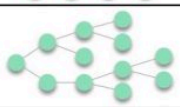
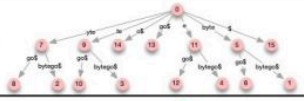
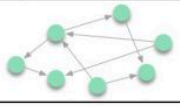
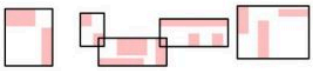
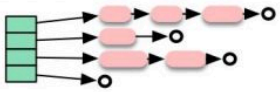


Over time, different API architectural styles are released. Each of them has its own patterns of standardizing data exchange.

You can check out the use cases of each style in the diagram.

10 Key Data Structures We Use Every Day

10 Data Structures Used in Daily Life — ByteByteGo.com

Data Structure	Illustration	Use Cases
List		Twitter feeds
Array		Math operations Large data sets
Stack		Undo/Redo of word editor
Queue		Printer jobs User actions in game
Heap		Task scheduling
Tree		HTML document AI decision
Suffix Tree		Search string in document
Graph		Friendship tracking Path finding
R-tree		Nearest neighbour
Hash Table		Caching systems

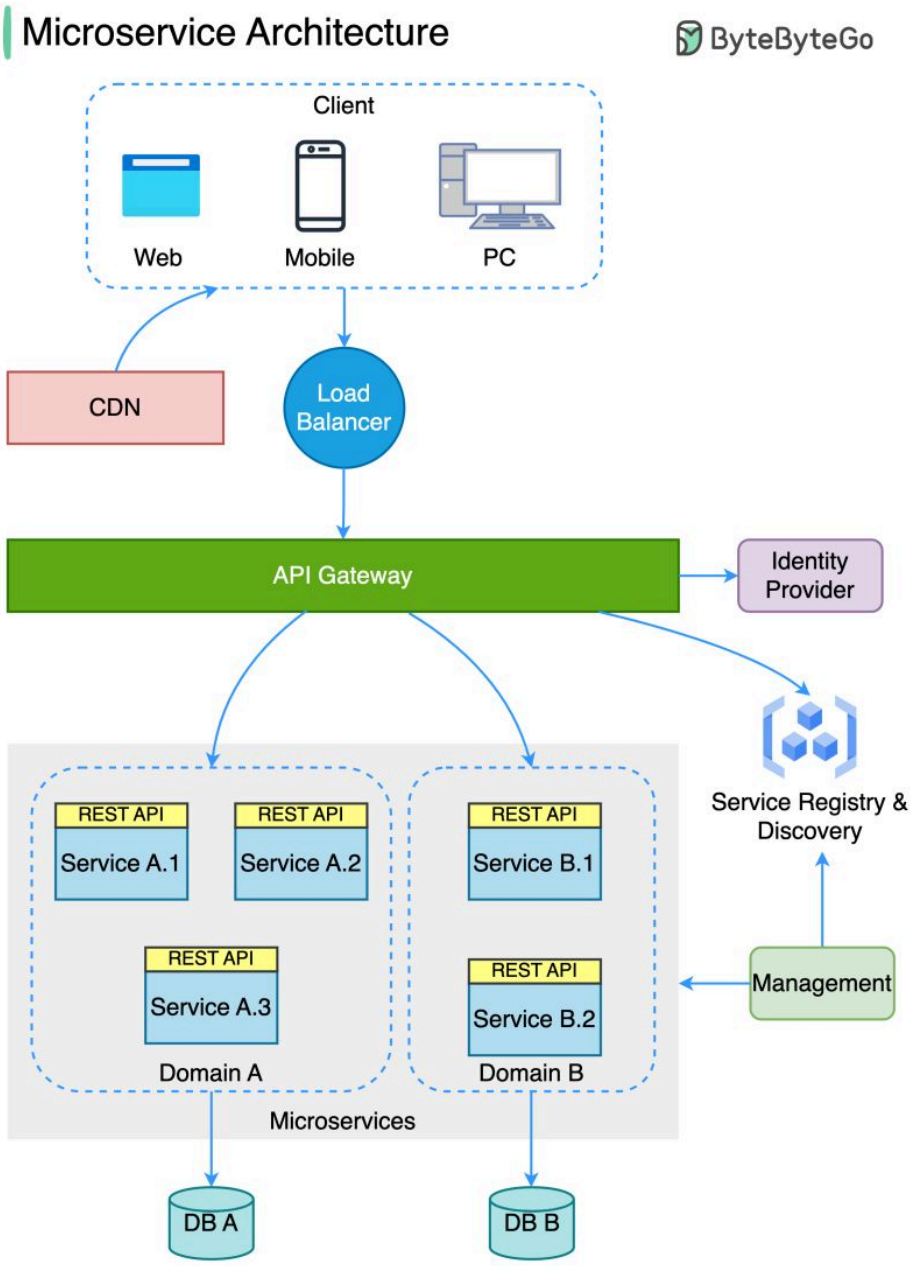
- list: keep your Twitter feeds
- stack: support undo/redo of the word editor
- queue: keep printer jobs, or send user actions in-game
- hash table: cashing systems
- Array: math operations
- heap: task scheduling
- tree: keep the HTML document, or for AI decision
- suffix tree: for searching string in a document
- graph: for tracking friendship, or path finding

- r-tree: for finding the nearest neighbor
- vertex buffer: for sending data to GPU for rendering

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

What does a typical microservice architecture look like? 👇

The diagram below shows a typical microservice architecture.



- Load Balancer: This distributes incoming traffic across multiple backend services.
- CDN (Content Delivery Network): CDN is a group of geographically distributed servers that hold static content for faster delivery. The clients look for content in CDN first, then progress to backend services.

- API Gateway: This handles incoming requests and routes them to the relevant services. It talks to the identity provider and service discovery.
- Identity Provider: This handles authentication and authorization for users.
- Service Registry & Discovery: Microservice registration and discovery happen in this component, and the API gateway looks for relevant services in this component to talk to.
- Management: This component is responsible for monitoring the services.
- Microservices: Microservices are designed and deployed in different domains. Each domain has its own database. The API gateway talks to the microservices via REST API or other protocols, and the microservices within the same domain talk to each other using RPC (Remote Procedure Call).

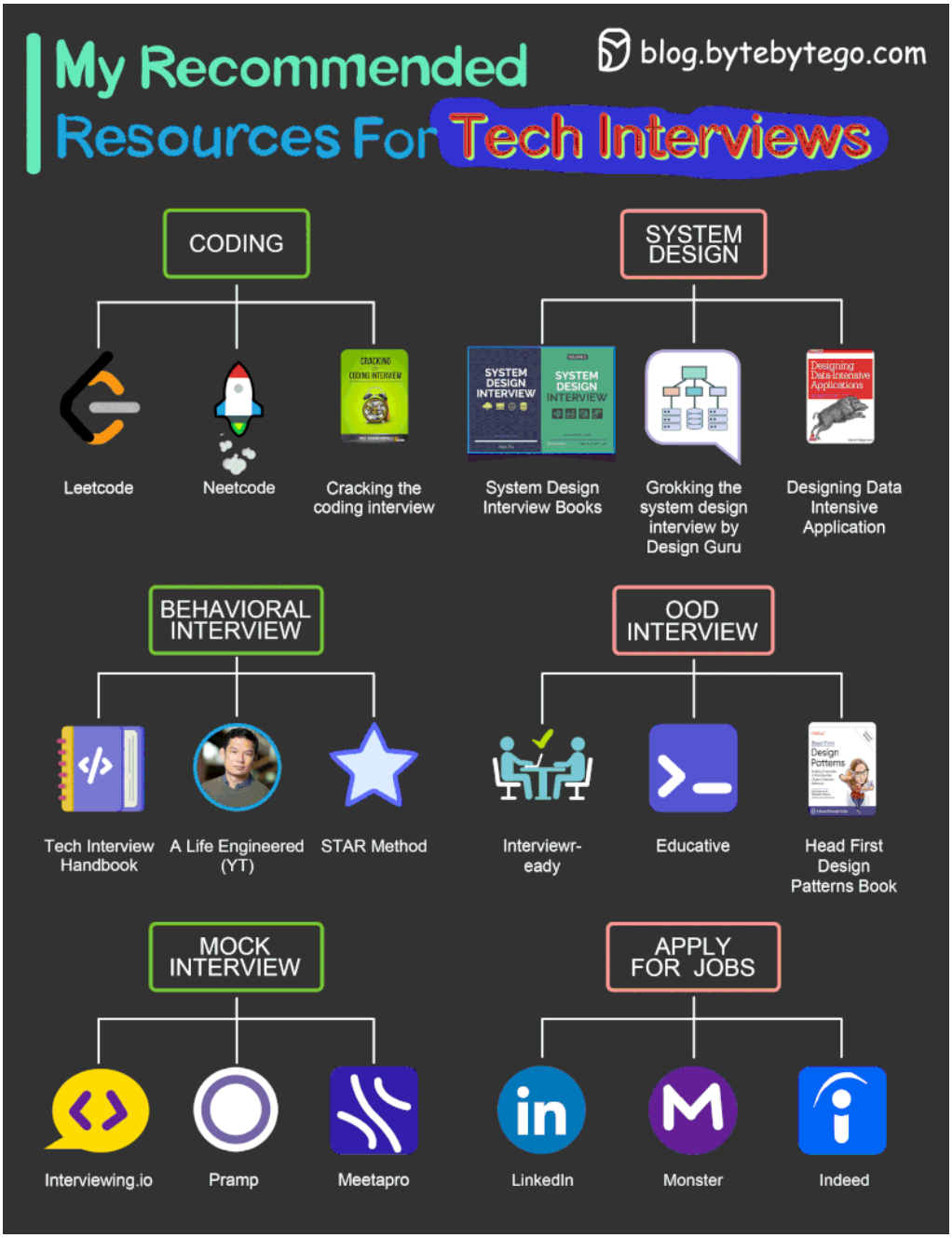
Benefits of microservices:

- They can be quickly designed, deployed, and horizontally scaled.
- Each domain can be independently maintained by a dedicated team.
- Business requirements can be customized in each domain and better supported, as a result.

Over to you:

1. What are the drawbacks of the microservice architecture?
2. Have you seen a monolithic system be transformed into microservice architecture? How long does it take?

My recommended materials for cracking your next technical interview



Coding

- Leetcode
- Cracking the coding interview book
- Neetcode

System Design Interview

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

Behavioral interview

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

OOD Interview

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

Mock interviews

- Interviewingio
- Pramp
- Meetapro

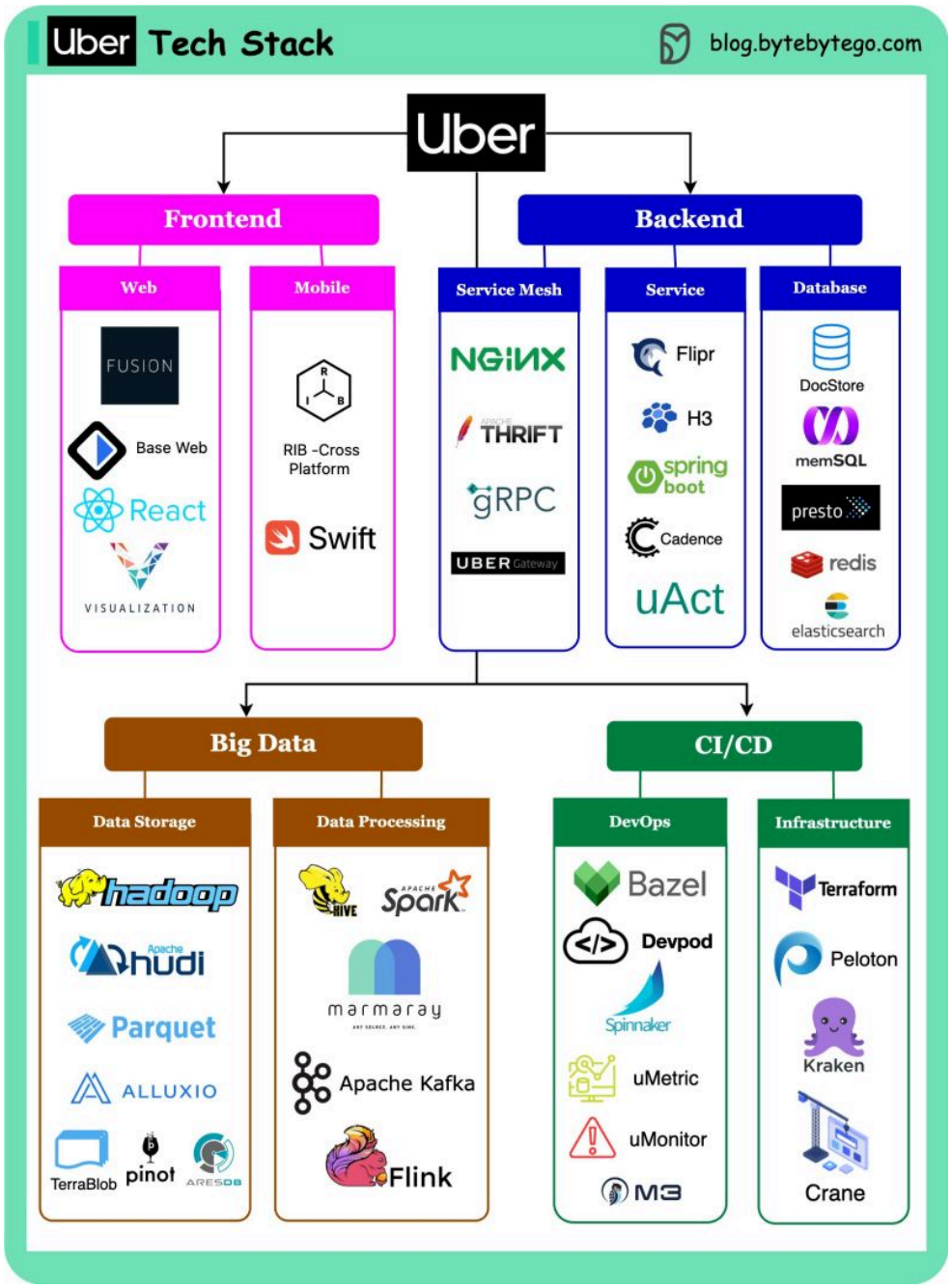
Apply for Jobs

- Linkedin
- Monster
- Indeed

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

Uber Tech Stack

This post is based on research from many Uber engineering blogs and open-source projects. If you come across any inaccuracies, please feel free to inform us. The corresponding links are added in the comment section.



Web frontend: Uber builds Fusion.js as a modern React framework to create robust web applications. They also develop visualization.js for geospatial visualization scenarios.

Mobile side: Uber builds the RIB cross-platform with the VIPER architecture instead of MVC. This architecture can work with different languages: Swift for iOS, and Java for Android.

Service mesh: Uber built Uber Gateway as a dynamic configuration on top of NGINX. The service uses gRPC and QUIC for client-server communication, and Apache Thrift for API definition.

Service side: Uber built a unified configuration store named Flipr (later changed to UCDP), H3 as a location-index store library. They use Spring Boot for Java-based services, uAct for event-driven architecture, and Cadence for async workflow orchestration.

Database end: the OLTP mainly uses the strongly-consistent DocStore, which employs MySQL and PostgreSQL, along with the RocksDB database engine.

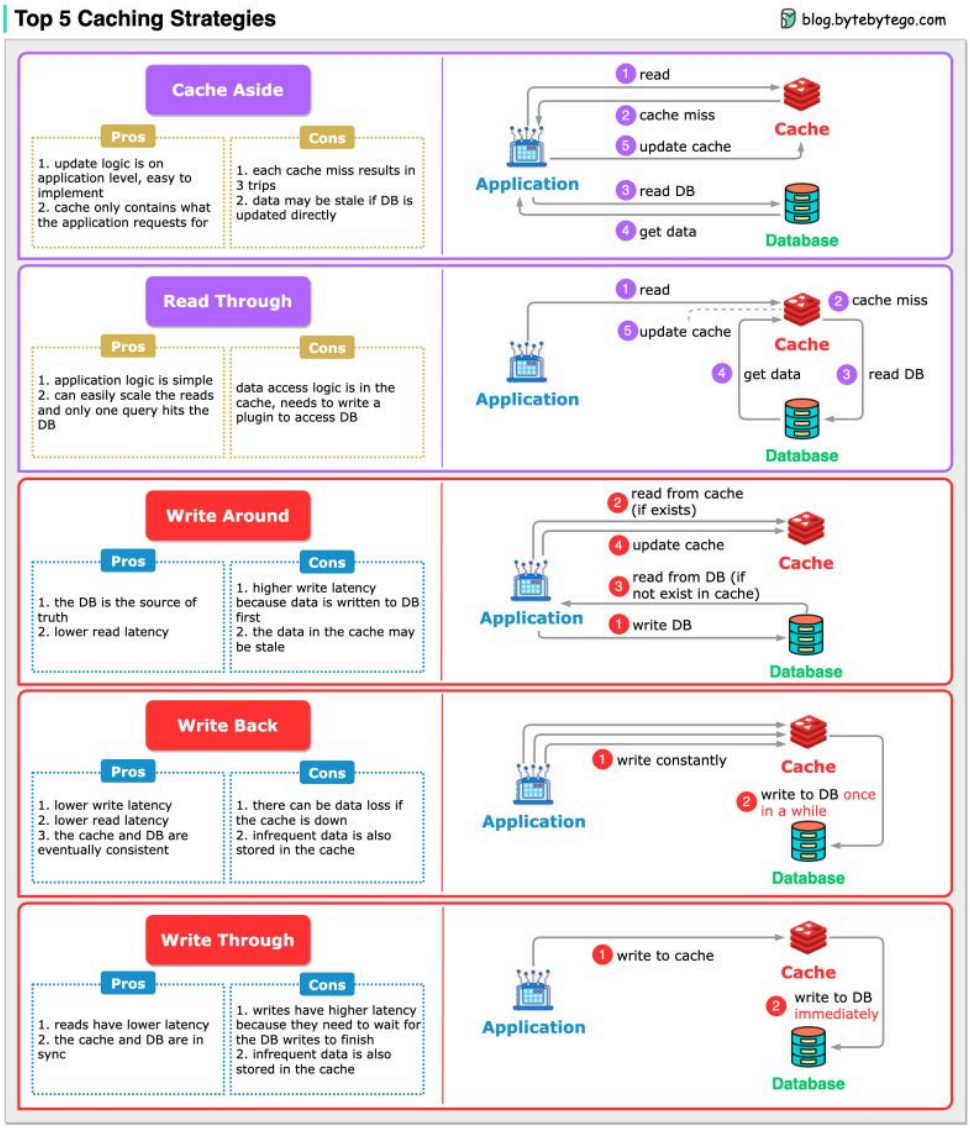
Big data: managed through the Hadoop family. Hudi and Parquet are used as file formats, and Alluxio serves as cache. Time-series data is stored in Pinot and AresDB.

Data processing: Hive, Spark, and the open-source data ingestion framework Marmaray. Messaging and streaming middleware include Apache Kafka and Apache Flink.

DevOps side: Uber utilizes a Monorepo, with a simplified development environment called devpod. Continuous delivery is managed through Netflix Spinnaker, metrics are emitted to uMetric, alarms on uMonitor, and a consistent observability database M3.

Top 5 Caching Strategies

When we introduce a cache into the architecture, synchronization between the cache and the database becomes inevitable.



Let's look at 5 common strategies how we keep the data in sync.

- Read Strategies:

  Cache aside
  Read through

- Write Strategies:

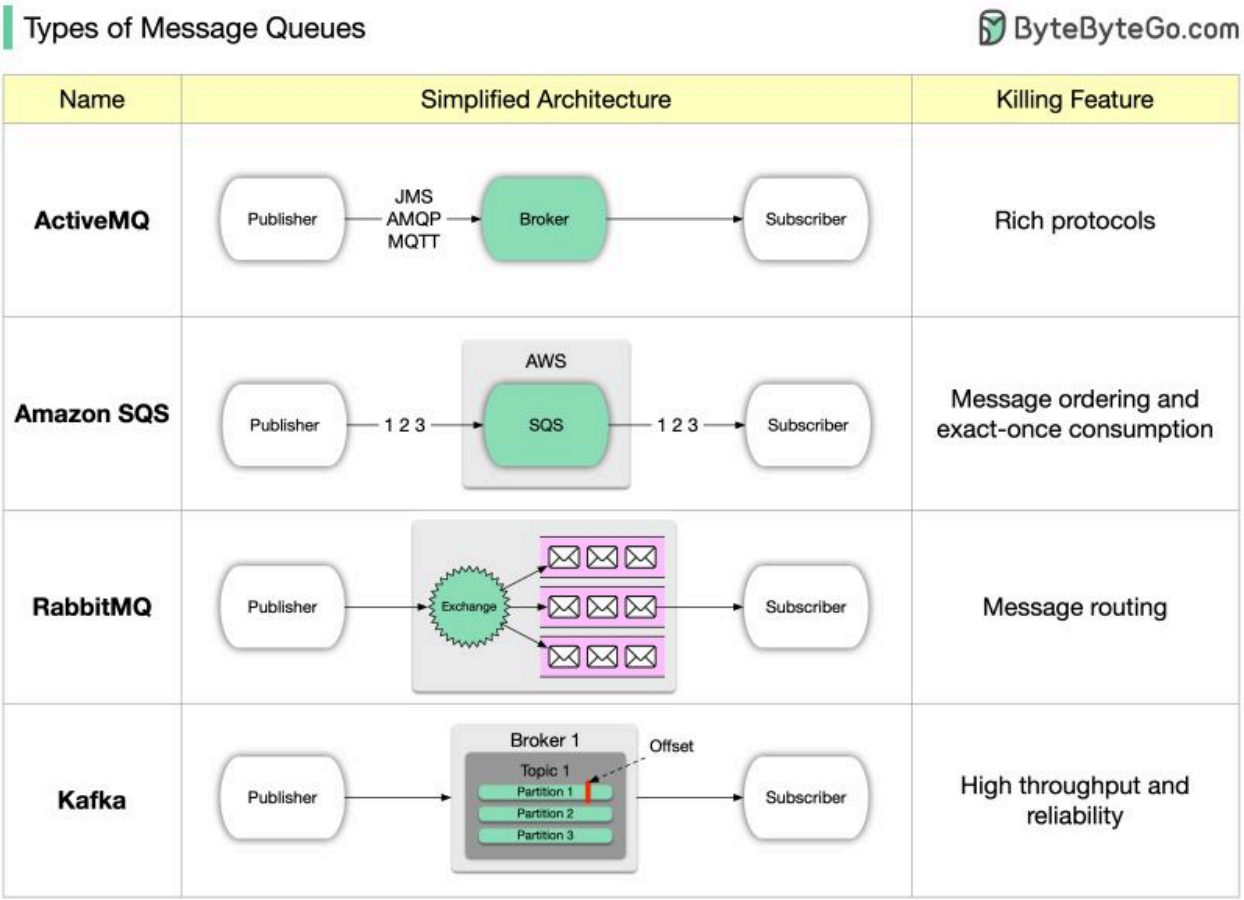
  Write around

Write back
Write through

The caching strategies are often used in combination. For example, write-around is often used together with cache-aside to make sure the cache is up-to-date.

Over to you: What strategies have you used?

How many message queues do you know?



Like a post office, a message queue helps computer programs to communicate in an organized manner. Imagine little digital envelopes being passed around to keep everything on track. There are few key features to consider when selecting message queues:

- Speed: How fast messages are sent and received
- Scalability: Can it grow with more messages
- Reliability: Will it make sure messages don’t get lost
- Durability: Can it keep messages safe over time
- Ease of Use: Is it simple to set up and manage
- Ecosystem: Are there helpful tools available
- Integration: Can it play nice with other software
- Protocol Support: What languages can it speak

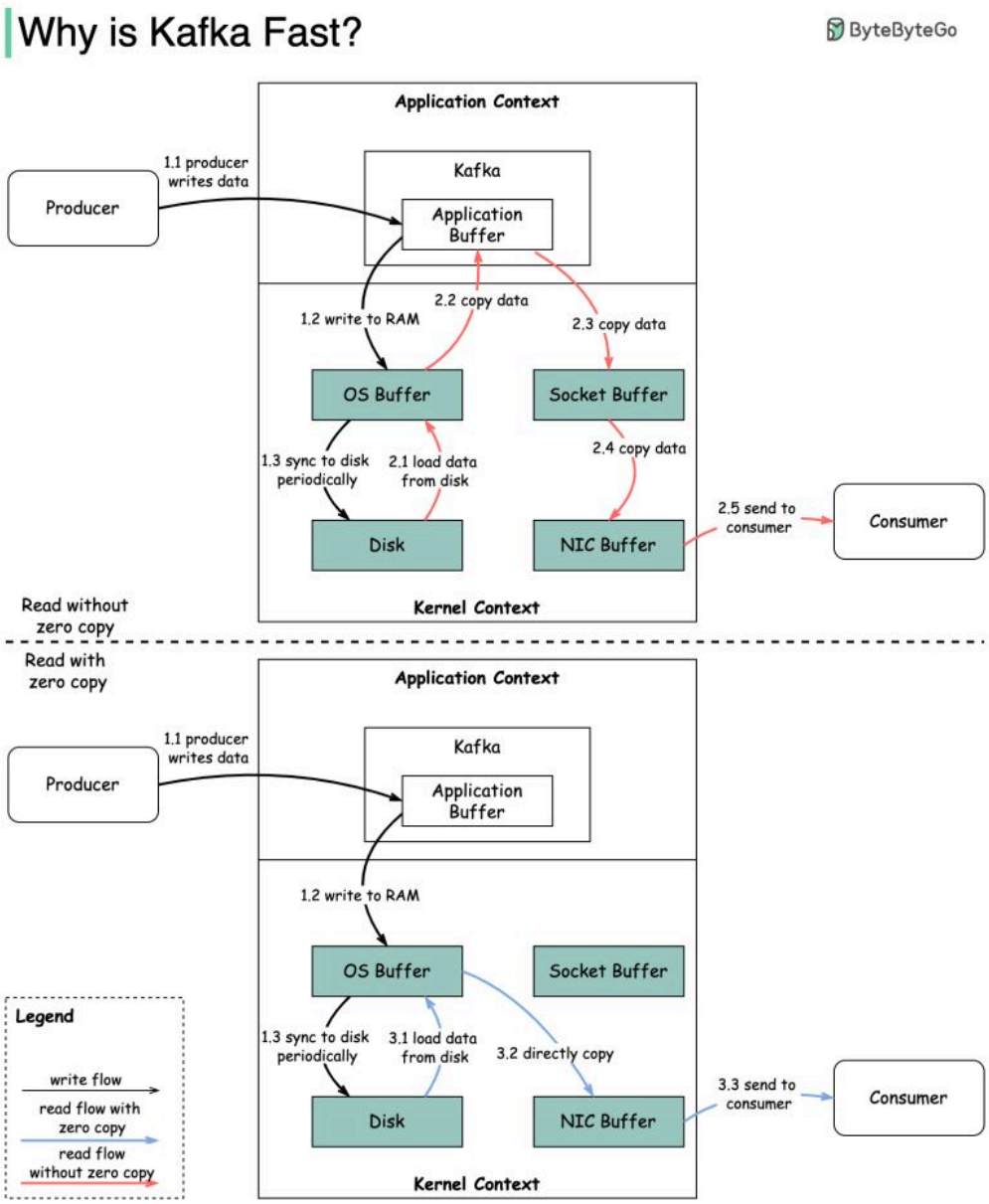
Try out a message queue and practice sending and receiving messages until you're comfortable. Choose an easy one like Kafka and experiment with sending and receiving messages. Read books or take online courses as you get more comfortable. Build little projects and learn from those who have already been there. Soon, you'll know everything about message queues.

Why is Kafka fast?

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

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

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



- Step 1.1 - 1.3: Producer writes data to the disk

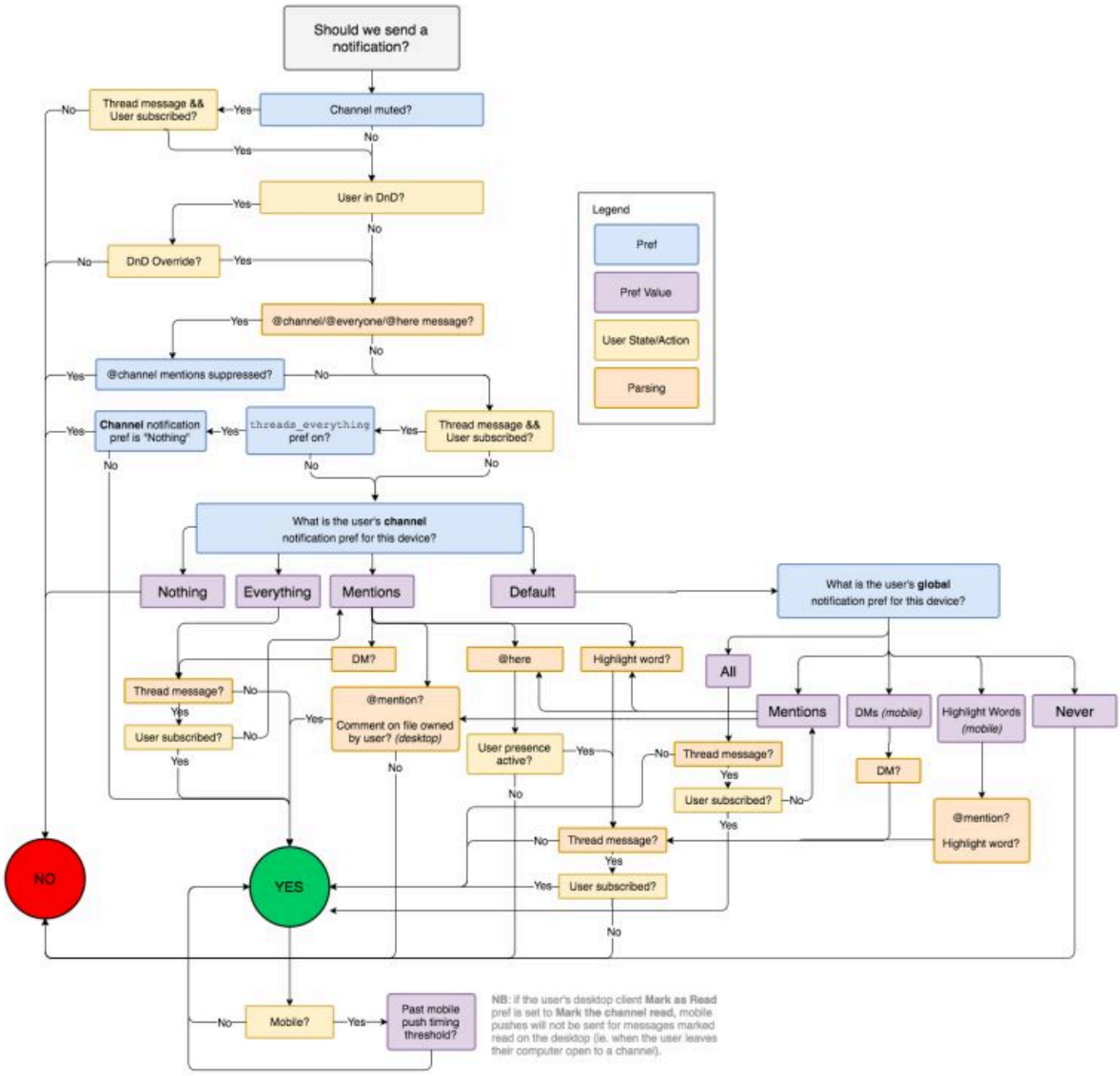
- Step 2: Consumer reads data without zero-copy
  2.1: The data is loaded from disk to OS cache
  2.2 The data is copied from OS cache to Kafka application
  2.3 Kafka application copies the data into the socket buffer
  2.4 The data is copied from socket buffer to network card
  2.5 The network card sends data out to the consumer

- Step 3: Consumer reads data with zero-copy
  3.1: The data is loaded from disk to OS cache
  3.2 OS cache directly copies the data to the network card via sendfile() command
  3.3 The network card sends data out to the consumer

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

How slack decides to send a notification

This is the flowchart of how slack decides to send a notification.



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

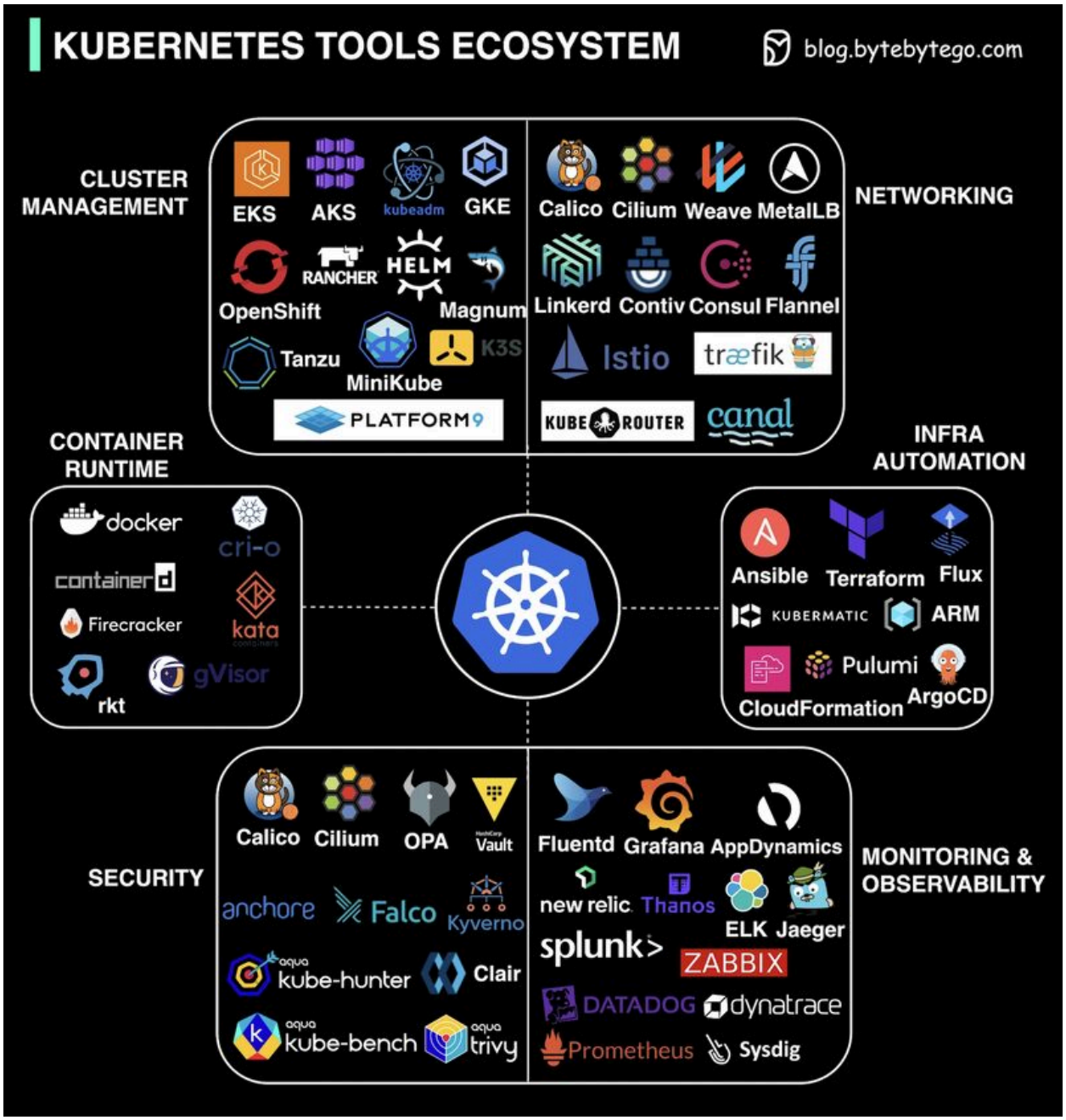
When we have a great design, users may not notice the complexity because it feels like the feature is just working as intended.

What's your takeaway from this diagram?

Source: [Slack Engineering Blog]

Kubernetes Tools Ecosystem

Kubernetes, the leading container orchestration platform, boasts a vast ecosystem of tools and components that collectively empower organizations to efficiently deploy, manage, and scale containerized applications.



Kubernetes practitioners need to be well-versed in these tools to ensure the reliability, security, and performance of containerized applications within Kubernetes clusters.

To introduce a holistic view of the Kubernetes ecosystem, we've created an illustration covering the aspects of:

1. Security
2. Networking
3. Container Runtime
4. Cluster Management
5. Monitoring and Observability
6. Infrastructure Orchestration

Over to you:

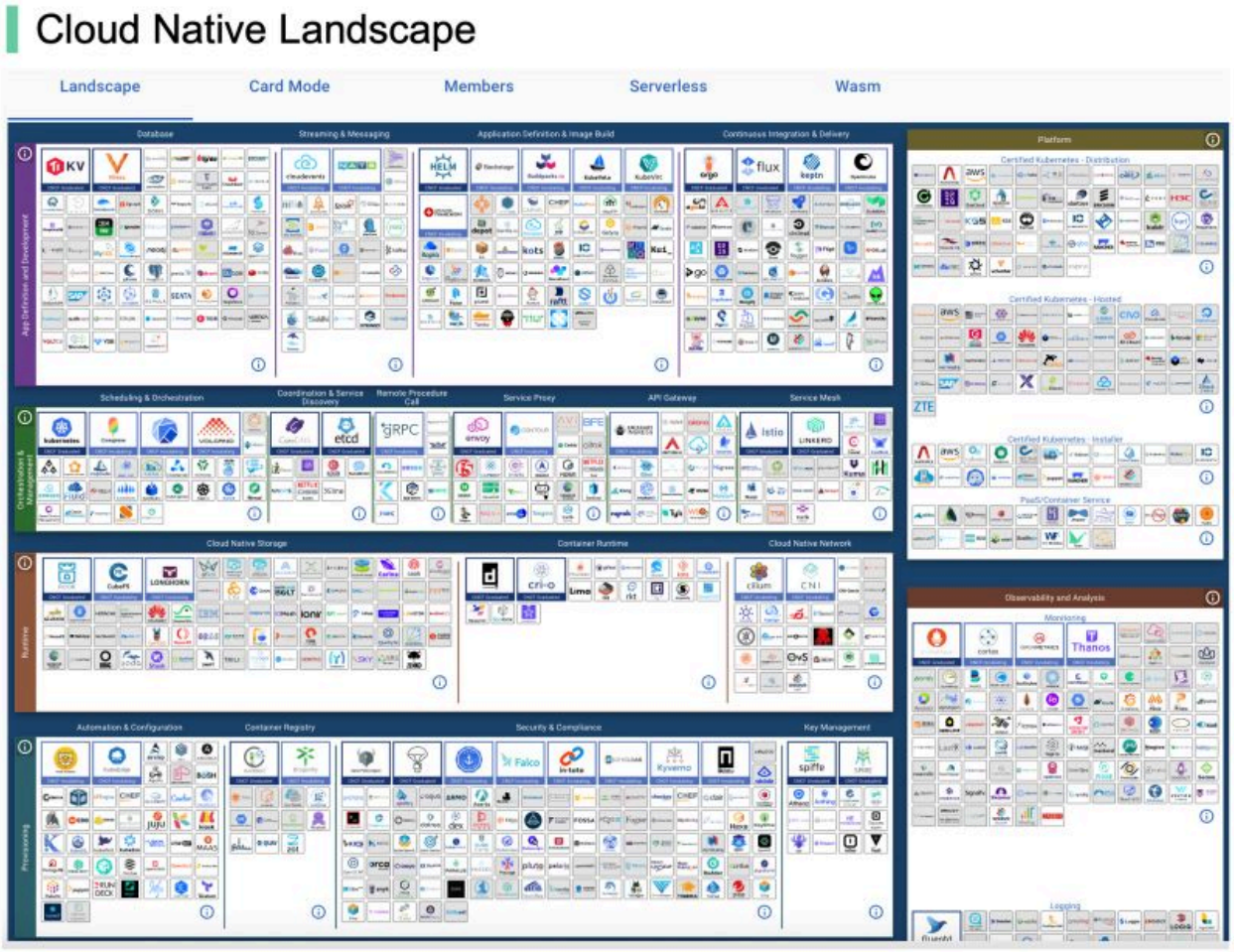
How have Kubernetes tools enhanced your containerized application management?

Cloud Native Landscape

Many Are Looking for the Definitive Guide on How to Choose the Right Stack

The ANSWER is...

There is no one-size-fits-all guide; it all depends on your specific needs, and picking the right stack is HARD.



Fortunately, at this point in time, technology is usually no longer a limiting factor. Most startups should be able to get by with most technologies they find. So spend less time on picking the perfect tech; instead, focus on your customers and keep building.

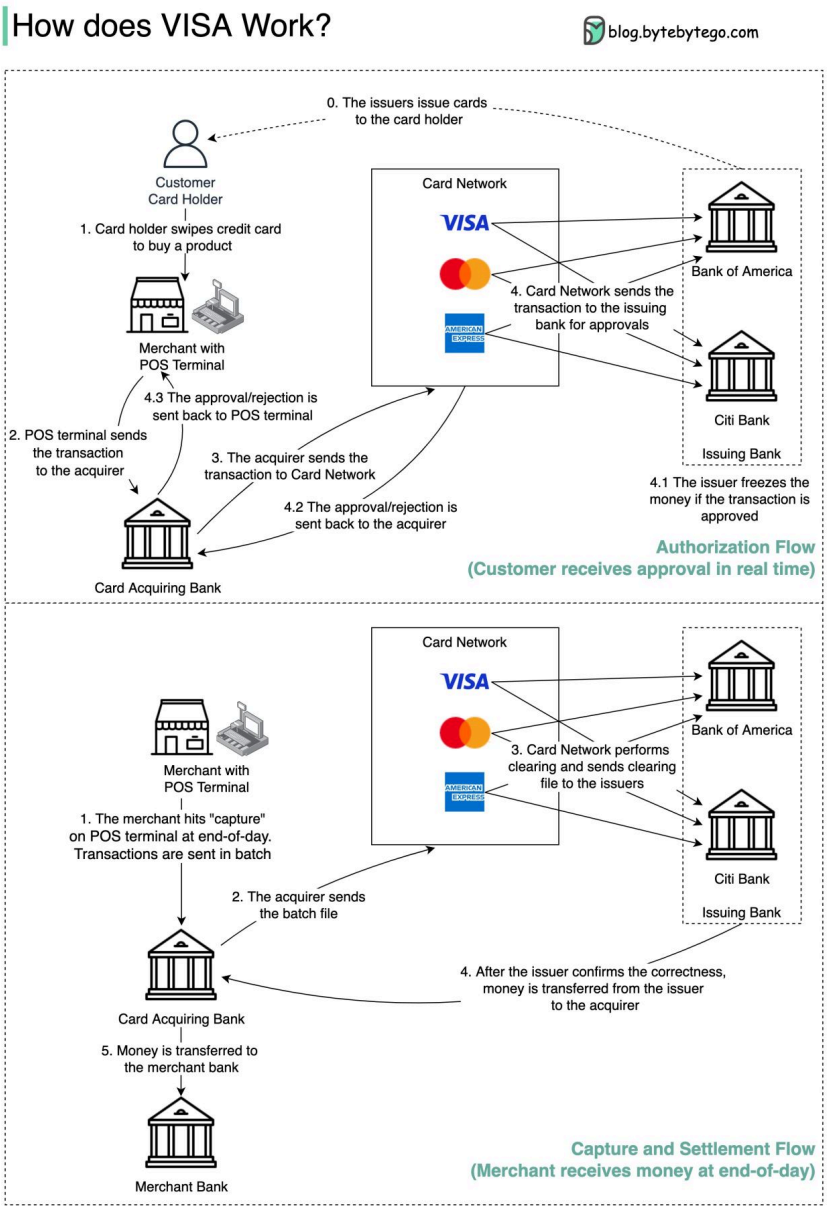
Over to you all: What do you think is causing this fragmentation in tech stack choices?

Source: [CNCF Cloud Native Interactive Landscape]

How does VISA work when we swipe a credit card at a merchant's shop?

VISA, Mastercard, and American Express act as card networks for the clearing and settling of funds. The card acquiring bank and the card issuing bank can be – and often are – different. If banks were to settle transactions one by one without an intermediary, each bank would have to settle the transactions with all the other banks. This is quite inefficient.

The diagram below shows VISA's role in the credit card payment process. There are two flows involved. Authorization flow happens when the customer swipes the credit card. Capture and settlement flow happens when the merchant wants to get the money at the end of the day.



- Authorization Flow

  Step 0: The card issuing bank issues credit cards to its customers.

  Step 1: The cardholder wants to buy a product and swipes the credit card at the Point of Sale (POS) terminal in the merchant's shop.

  Step 2: The POS terminal sends the transaction to the acquiring bank, which has provided the POS terminal.

  Steps 3 and 4: The acquiring bank sends the transaction to the card network, also called the card scheme. The card network sends the transaction to the issuing bank for approval.

  Steps 4.1, 4.2 and 4.3: The issuing bank freezes the money if the transaction is approved. The approval or rejection is sent back to the acquirer, as well as the POS terminal.

- Capture and Settlement Flow

  Steps 1 and 2: The merchant wants to collect the money at the end of the day, so they hit "capture" on the POS terminal. The transactions are sent to the acquirer in batch. The acquirer sends the batch file with transactions to the card network.

  Step 3: The card network performs clearing for the transactions collected from different acquirers, and sends the clearing files to different issuing banks.

  Step 4: The issuing banks confirm the correctness of the clearing files, and transfer money to the relevant acquiring banks.

  Step 5: The acquiring bank then transfers money to the merchant's bank.

  Step 4: The card network clears the transactions from different acquiring banks. Clearing is a process in which mutual offset transactions are netted, so the number of total transactions is reduced.

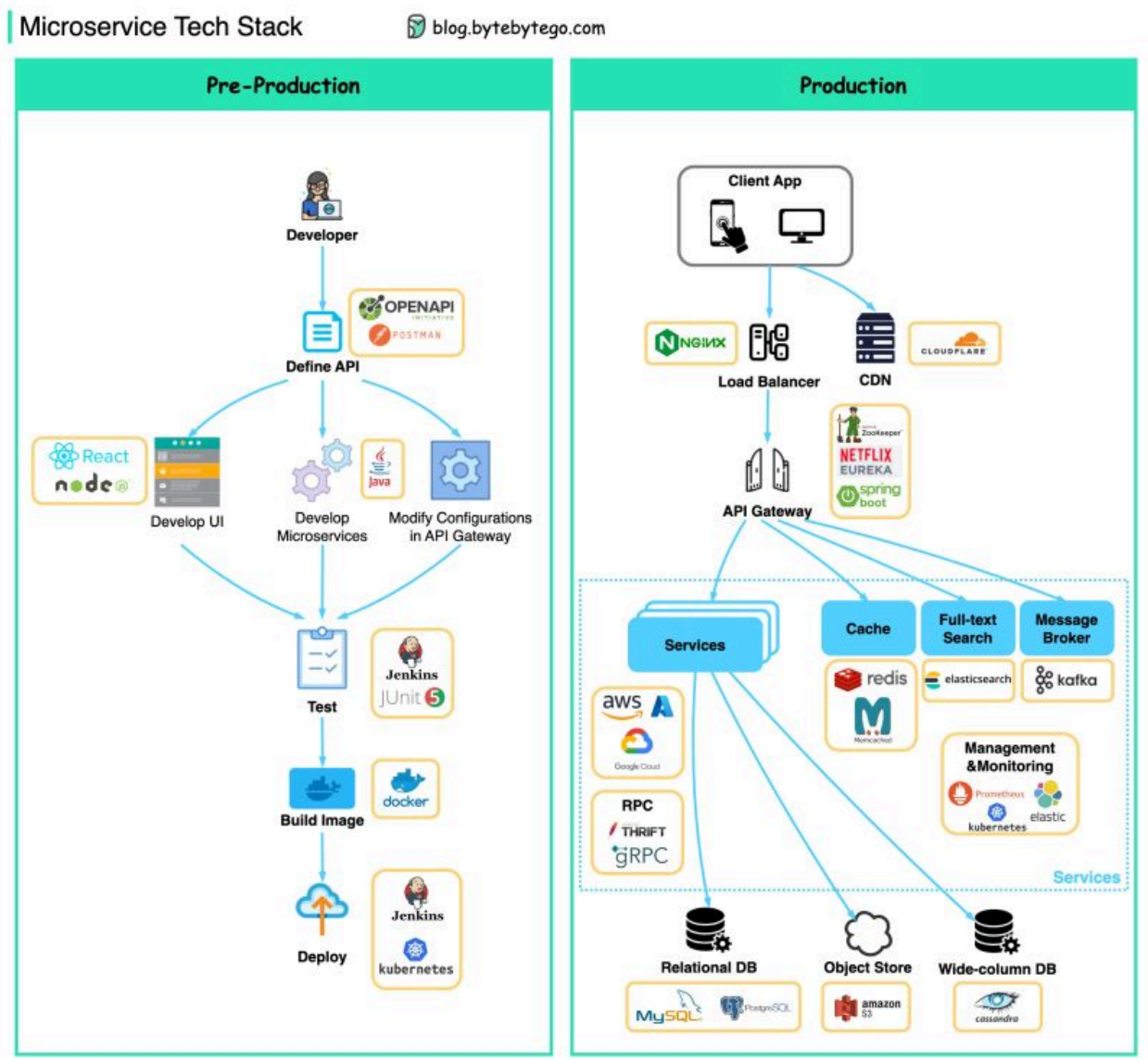
In the process, the card network takes on the burden of talking to each bank and receives service fees in return.

Over to you: Do you think this flow is way too complicated? What will be the future of payments in your opinion?

A simple visual guide to help people understand the key considerations when designing or using caching systems

What tech stack is commonly used for microservices?

Below you will find a diagram showing the microservice tech stack, both for the development phase and for production.



▶️Pre-production

- Define API - This establishes a contract between frontend and backend. We can use Postman or OpenAPI for this.
- Development - Node.js or react is popular for frontend development, and java/python/go for backend development. Also, we need to change the configurations in the API gateway according to API definitions.

- Continuous Integration - JUnit and Jenkins for automated testing. The code is packaged into a Docker image and deployed as microservices.

▶️ **Production**

- NGinx is a common choice for load balancers. Cloudflare provides CDN (Content Delivery Network).
- API Gateway - We can use spring boot for the gateway, and use Eureka/Zookeeper for service discovery.
- The microservices are deployed on clouds. We have options among AWS, Microsoft Azure, or Google GCP.
- Cache and Full-text Search - Redis is a common choice for caching key-value pairs. ElasticSearch is used for full-text search.
- Communications - For services to talk to each other, we can use messaging infra Kafka or RPC.
- Persistence - We can use MySQL or PostgreSQL for a relational database, and Amazon S3 for object store. We can also use Cassandra for the wide-column store if necessary.
- Management & Monitoring - To manage so many microservices, the common Ops tools include Prometheus, Elastic Stack, and Kubernetes.

Over to you: Did I miss anything? Please comment on what you think is necessary to learn microservices.