System Design 101

Explain complex systems using visuals and simple terms.

Whether you're preparing for a System Design Interview or you simply want to understand how systems work beneath the surface, we hope this repository will help you achieve that.

Table of Contents

monolithic? How can it save 90% cost?
  - How does Disney Hotstar capture 5 Billion Emojis during a tournament?
  - How Discord Stores Trillions Of Messages
  - How do video live streamings work on YouTube, TikTok live, or Twitch?

Communication protocols

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

REST API vs. GraphQL

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

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

REST

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

GraphQL

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

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

Neither API approach is a silver bullet. Carefully evaluating requirements and tradeoffs is important to pick the right style. Both REST and GraphQL are valid options for exposing data and powering modern applications.

How does gRPC work?

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

The diagram below illustrates the overall data flow for **gRPC**.

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

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

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

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

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

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

What is a webhook?

The diagram below shows a comparison between polling and Webhook.

Assume we run an eCommerce website. The clients send orders to the order service via the API gateway, which goes to the payment service for payment transactions. The payment service then talks to an external payment service provider (PSP) to complete the transactions.

There are two ways to handle communications with the external PSP.

1. Short polling

After sending the payment request to the PSP, the payment service keeps asking the PSP about the payment status. After several rounds, the PSP finally returns with the status.

Short polling has two drawbacks:

- Constant polling of the status requires resources from the payment service.
- The External service communicates directly with the payment service, creating security vulnerabilities.

2. Webhook

We can register a webhook with the external service. It means: call me back at a certain URL when you have updates on the request. When the PSP has completed the processing, it will invoke the HTTP request to update the payment status.

In this way, the programming paradigm is changed, and the payment service doesn't need to waste resources to poll the payment status anymore.

What if the PSP never calls back? We can set up a housekeeping job to check payment status every hour.

Webhooks are often referred to as reverse APIs or push APIs because the server sends HTTP requests to the client. We need to pay attention to 3 things when using a webhook:

1. We need to design a proper API for the external service to call.

2. We need to set up proper rules in the API gateway for security reasons.
3. We need to register the correct URL at the external service.

How to improve API performance?

The diagram below shows 5 common tricks to improve API performance.

Pagination

This is a common optimization when the size of the result is large. The results are streaming back to the client to improve the service responsiveness.

Asynchronous Logging

Synchronous logging deals with the disk for every call and can slow down the system. Asynchronous logging sends logs to a lock-free buffer first and immediately returns. The logs will be flushed to the disk periodically. This significantly reduces the I/O overhead.

Caching

We can store frequently accessed data into a cache. The client can query the cache first instead of visiting the database directly. If there is a cache miss, the client can query from the database. Caches like Redis store data in memory, so the data access is much faster than the database.

Payload Compression

The requests and responses can be compressed using gzip etc so that the transmitted data size is much smaller. This speeds up the upload and download.

Connection Pool

When accessing resources, we often need to load data from the database. Opening the closing db connections adds significant overhead. So we should connect to the db via a pool of open connections. The connection pool is responsible for managing the connection lifecycle.

HTTP 1.0 -> HTTP 1.1 -> HTTP 2.0 -> HTTP 3.0 (QUIC)

What problem does each generation of HTTP solve?

The diagram below illustrates the key features.

- HTTP 1.0 was finalized and fully documented in 1996. Every request to the same server requires a separate TCP connection.
- HTTP 1.1 was published in 1997. A TCP connection can be left open for reuse (persistent connection), but it doesn't solve the HOL (head-of-line) blocking issue.

HOL blocking - when the number of allowed parallel requests in the browser is used up, subsequent requests need to wait for the former ones to complete.

- HTTP 2.0 was published in 2015. It addresses HOL issue through request multiplexing, which eliminates HOL blocking at the application layer, but HOL still exists at the transport (TCP) layer.

  As you can see in the diagram, HTTP 2.0 introduced the concept of HTTP "streams": an abstraction that allows multiplexing different HTTP exchanges onto the same TCP connection. Each stream doesn't need to be sent in order.

- HTTP 3.0 first draft was published in 2020. It is the proposed successor to HTTP 2.0. It uses QUIC instead of TCP for the underlying transport protocol, thus removing HOL blocking in the transport layer.

QUIC is based on UDP. It introduces streams as first-class citizens at the transport layer. QUIC streams share the same QUIC connection, so no additional handshakes and slow starts are required to create new ones, but QUIC streams are delivered independently such that in most cases packet loss affecting one stream doesn't affect others.

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.

Code First vs. API First

The diagram below shows the differences between code-first development and API-first development. Why do we want to consider API first design?

- Microservices increase system complexity and we have separate services to serve different functions of the system. While this kind of architecture facilitates decoupling and segregation of duty, we need to handle the various communications among services.

It is better to think through the system's complexity before writing the code and carefully defining the boundaries of the services.

- Separate functional teams need to speak the same language and the dedicated functional teams are only responsible for their own components and services. It is recommended that the organization speak the same language via API design.

We can mock requests and responses to validate the API design before writing code.

- Improve software quality and developer productivity Since we have ironed out most of the uncertainties when the project starts, the overall development process is smoother, and the software quality is greatly improved.

Developers are happy about the process as well because they can focus on functional development instead of negotiating sudden changes.

The possibility of having surprises toward the end of the project lifecycle is reduced.

Because we have designed the API first, the tests can be designed while the code is being developed. In a way, we also have TDD (Test Driven Design) when using API first development.

HTTP status codes

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)

What does API gateway do?

The diagram below shows the details.

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.

How do we design effective and safe APIs?

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

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

TCP/IP encapsulation

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

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

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

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

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

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

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

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

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

Why is Nginx called a "reverse" proxy?

The diagram below shows the differences between a and a .

A forward proxy is a server that sits between user devices and the internet.

A forward proxy is commonly used for:

1. Protecting clients
2. Circumventing browsing restrictions
3. Blocking 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:

1. Protecting servers
2. Load balancing
3. Caching static contents
4. Encrypting and decrypting SSL communications

What are the common load-balancing algorithms?

The diagram below shows 6 common 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.

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.

CI/CD

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 and 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 the 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:

- The developer commits code changes to the source control

- CI server detects changes and triggers the build
- Code is compiled, and tested (unit, integration tests)
- Test results reported to the 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

Netflix Tech Stack (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.

Architecture patterns

MVC, MVP, MVVM, MVVM-C, and VIPER

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)

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.

Database

A nice cheat sheet of different databases in cloud services

Choosing the right database for your project is a complex task. Many 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 need to be more accurate.

8 Data Structures That Power Your Databases

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.

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

How is an SQL statement executed in the database?

The diagram below shows the process. Note that the architectures for different databases are different, the diagram demonstrates some common designs.

Step 1 - A SQL statement is sent to the database via a transport layer protocol (e.g.TCP).

Step 2 - The SQL statement is sent to the command parser, where it goes through syntactic and semantic analysis, and a query tree is generated afterward.

Step 3 - The query tree is sent to the optimizer. The optimizer creates an execution plan.

Step 4 - The execution plan is sent to the executor. The executor retrieves data from the execution.

Step 5 - Access methods provide the data fetching logic required for execution, retrieving data from the storage engine.

Step 6 - Access methods decide whether the SQL statement is read-only. If the query is read-only (SELECT statement), it is passed to the buffer manager for further processing. The buffer manager looks for the data in the cache or data files.

Step 7 - If the statement is an UPDATE or INSERT, it is passed to the transaction manager for further processing.

Step 8 - During a transaction, the data is in lock mode. This is guaranteed by the lock manager. It also ensures the transaction's ACID properties.

CAP theorem

The CAP theorem is one of the most famous terms in computer science, but I bet different developers have different understandings. Let's examine what it is and why it can be confusing.

CAP theorem states that a distributed system can't provide more than two of these three guarantees simultaneously.

Consistency: consistency means all clients see the same data at the same time no matter which node they connect to.

Availability: availability means any client that requests data gets a response even if some of the nodes are down.

Partition Tolerance: a partition indicates a communication break between two nodes. Partition tolerance means the system continues to operate despite network partitions.

The "2 of 3" formulation can be useful, **but this simplification could be misleading**.

1. Picking a database is not easy. Justifying our choice purely based on the CAP theorem is not enough. For example, companies don't choose Cassandra for chat applications simply because it is an AP system. There is a list of good characteristics that make Cassandra a desirable option for storing chat messages. We need to dig deeper.
2. "CAP prohibits only a tiny part of the design space: perfect availability and consistency in the presence of partitions, which are rare". Quoted from the paper: CAP Twelve Years Later: How the "Rules" Have Changed.
3. The theorem is about 100% availability and consistency. A more realistic discussion would be the trade-offs between latency and consistency when there is no network partition. See PACELC theorem for more details.

Is the CAP theorem actually useful?

I think it is still useful as it opens our minds to a set of tradeoff discussions, but it is only part of the story. We need to dig deeper when picking the right database.

Types of Memory and Storage

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

SQL language

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

There are 5 components of the SQL language:

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

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

Cache

Data is cached everywhere

This diagram illustrates where we cache data in a typical architecture.

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

1. Client apps: HTTP responses can be cached by the browser. We request data over HTTP for the first time, and it is returned with an expiry policy in the HTTP header; we request data again, and the client app tries to retrieve the data from the browser cache first.
2. CDN: CDN caches static web resources. The clients can retrieve data from a CDN node nearby.
3. Load Balancer: The load Balancer can cache resources as well.
4. Messaging infra: Message brokers store messages on disk first, and then consumers retrieve them at their own pace. Depending on the retention policy, the data is cached in Kafka clusters for a period of time.

5. Services: There are multiple layers of cache in a service. If the data is not cached in the CPU cache, the service will try to retrieve the data from memory. Sometimes the service has a second-level cache to store data on disk.
6. Distributed Cache: Distributed cache like Redis holds key-value pairs for multiple services in memory. It provides much better read/write performance than the database.
7. Full-text Search: we sometimes need to use full-text searches like Elastic Search for document search or log search. A copy of data is indexed in the search engine as well.
8. Database: Even in the database, we have different levels of caches:

- WAL(Write-ahead Log): data is written to WAL first before building the B tree index
- Bufferpool: A memory area allocated to cache query results
- Materialized View: Pre-compute query results and store them in the database tables for better query performance
- Transaction log: record all the transactions and database updates
- Replication Log: used to record the replication state in a database cluster

Why is Redis so fast?

There are 3 main reasons as shown in the diagram below.

1. Redis is a RAM-based data store. RAM access is at least 1000 times faster than random disk access.
2. Redis leverages IO multiplexing and single-threaded execution loop for execution efficiency.
3. Redis leverages several efficient lower-level data structures.

Question: Another popular in-memory store is Memcached. Do you know the differences between Redis and Memcached?

You might have noticed the style of this diagram is different from my previous posts. Please let me know which one you prefer.

How can Redis be used?

There is more to Redis than just caching.

Redis can be used in a variety of scenarios as shown in the diagram.

- Session

  We can use Redis to share user session data among different services.

- Cache

  We can use Redis to cache objects or pages, especially for hotspot data.

- Distributed lock

We can use a Redis string to acquire locks among distributed services.

- Counter

  We can count how many likes or how many reads for articles.

- Rate limiter

  We can apply a rate limiter for certain user IPs.

- Global ID generator

  We can use Redis Int for global ID.

- Shopping cart

  We can use Redis Hash to represent key-value pairs in a shopping cart.

- Calculate user retention

  We can use Bitmap to represent the user login daily and calculate user retention.

- Message queue

  We can use List for a message queue.

- Ranking

  We can use ZSet to sort the articles.

Top caching strategies

Designing large-scale systems usually requires careful consideration of caching. Below are five caching strategies that are frequently utilized.

Microservice architecture

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.

Microservice Best Practices

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

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.

-

- 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.

- 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.

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 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 the multiple data copies between application context and kernel context.

Payment systems

How to learn payment systems?

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

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

1. The cardholder pays a merchant $100 to buy a product.

2. The merchant benefits from the use of the credit card with higher sales volume and needs to compensate the issuer and the card network for providing the payment service. The acquiring bank sets a fee with the merchant, called the "merchant discount fee."

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

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

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

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

Why should the issuing bank be compensated?

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

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 up 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.

Payment Systems Around The World Series (Part 1): Unified Payments Interface (UPI) in India

What's UPI? UPI is an instant real-time payment system developed by the National Payments Corporation of India.

It accounts for 60% of digital retail transactions in India today.

UPI = payment markup language + standard for interoperable payments

DevOps

DevOps vs. SRE vs. Platform Engineering. What is the difference?

The concepts of DevOps, SRE, and Platform Engineering have emerged at different times and have been developed by various individuals and organizations.

DevOps as a concept was introduced in 2009 by Patrick Debois and Andrew Shafer at the Agile conference. They sought to bridge the gap between software development and operations by promoting a collaborative culture and shared responsibility for the entire software development lifecycle.

SRE, or Site Reliability Engineering, was pioneered by Google in the early 2000s to address operational challenges in managing large-scale, complex systems. Google developed SRE practices and tools, such as the Borg cluster management system and the Monarch monitoring system, to improve the reliability and efficiency of their services.

Platform Engineering is a more recent concept, building on the foundation of SRE engineering. The precise origins of Platform Engineering are less clear, but it is generally understood to be an extension of the DevOps and SRE practices, with a focus on delivering a comprehensive platform for product development that supports the entire business perspective.

It's worth noting that while these concepts emerged at different times. They are all related to the broader trend of improving collaboration, automation, and efficiency in software development and operations.

What is k8s (Kubernetes)?

K8s is a container orchestration system. It is used for container deployment and management. Its design is greatly impacted by Google's internal system Borg.

A k8s cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.

The worker node(s) host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers, and a cluster usually runs multiple nodes, providing fault tolerance and high availability.

- Control Plane Components

1. API Server

   The API server talks to all the components in the k8s cluster. All the operations on pods are executed by talking to the API server.

2. Scheduler

The scheduler watches pod workloads and assigns loads on newly created pods.

3. Controller Manager

   The controller manager runs the controllers, including Node Controller, Job Controller, EndpointSlice Controller, and ServiceAccount Controller.

4. Etcd

   etcd is a key-value store used as Kubernetes' backing store for all cluster data.

- Nodes

1. Pods

   A pod is a group of containers and is the smallest unit that k8s administers. Pods have a single IP address applied to every container within the pod.

2. Kubelet

   An agent that runs on each node in the cluster. It ensures containers are running in a Pod.

3. Kube Proxy

   Kube-proxy is a network proxy that runs on each node in your cluster. It routes traffic coming into a node from the service. It forwards requests for work to the correct containers.

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.

How does Docker work?

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

There are 3 components in Docker architecture:

- Docker client

  The docker client talks to the Docker daemon.

- Docker host

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

- Docker registry

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

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

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

GIT

How Git Commands work

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

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

Most Git commands primarily move files between these four locations.

How does Git Work?

The diagram below shows the Git workflow.

Git is a distributed version control system.

Every developer maintains a local copy of the main repository and edits and commits to the local copy.

The commit is very fast because the operation doesn't interact with the remote repository.

If the remote repository crashes, the files can be recovered from the local repositories.

Git merge vs. Git rebase

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 a 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 Services

A nice cheat sheet of different cloud services (2023 edition)

What is cloud native?

Below is a diagram showing the evolution of architecture and processes since the 1980s.

Organizations can build and run scalable applications on public, private, and hybrid clouds using cloud native technologies.

This means the applications are designed to leverage cloud features, so they are resilient to load and easy to scale.

Cloud native includes 4 aspects:

1. Development process

   This has progressed from waterfall to agile to DevOps.

2. Application Architecture

   The architecture has gone from monolithic to microservices. Each service is designed to be small, adaptive to the limited resources in cloud containers.

3. Deployment & packaging

   The applications used to be deployed on physical servers. Then around 2000, the applications that were not sensitive to latency were usually deployed on virtual servers. With cloud native applications, they are packaged into docker images and deployed in containers.

4. Application infrastructure

   The applications are massively deployed on cloud infrastructure instead of self-hosted servers.

Developer productivity tools

Visualize JSON files

Nested JSON files are hard to read.

JsonCrack generates graph diagrams from JSON files and makes them easy to read.

Additionally, the generated diagrams can be downloaded as images.

Automatically turn code into architecture diagrams

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, Alibaba Cloud, Oracle Cloud, etc.

Github repo

Linux

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.

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.

This diagram below shows 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

Security

How does HTTPS work?

Hypertext Transfer Protocol Secure (HTTPS) is an extension of the Hypertext Transfer Protocol (HTTP.) HTTPS transmits encrypted data using Transport Layer Security (TLS.) If the data is hijacked online, all the hijacker gets is binary code.

How is the data encrypted and decrypted?

Step 1 - The client (browser) and the server establish a TCP connection.

Step 2 - The client sends a “client hello” to the server. The message contains a set of necessary encryption algorithms (cipher suites) and the latest TLS version it can support. The server responds with a “server hello” so the browser knows whether it can support the algorithms and TLS version.

The server then sends the SSL certificate to the client. The certificate contains the public key, host name, expiry dates, etc. The client validates the certificate.

Step 3 - After validating the SSL certificate, the client generates a session key and encrypts it using the public key. The server receives the encrypted session key and decrypts it with the private key.

Step 4 - Now that both the client and the server hold the same session key (symmetric encryption), the encrypted data is transmitted in a secure bi-directional channel.

Why does HTTPS switch to symmetric encryption during data transmission? There are two main reasons:

1. Security: The asymmetric encryption goes only one way. This means that if the server tries to send the encrypted data back to the client, anyone can decrypt the data using the public key.
2. Server resources: The asymmetric encryption adds quite a lot of mathematical overhead. It is not suitable for data transmissions in long sessions.

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.

Top 4 Forms of Authentication Mechanisms

1. SSH Keys:

   Cryptographic keys are used to access remote systems and servers securely

2. OAuth Tokens:

   Tokens that provide limited access to user data on third-party applications

3. SSL Certificates:

   Digital certificates ensure secure and encrypted communication between servers and clients

4. Credentials:

   User authentication information is used to verify and grant access to various systems and services

Session, cookie, JWT, token, SSO, and OAuth 2.0 - what are they?

These terms are all related to user identity management. When you log into a website, you declare who you are (identification). Your identity is verified (authentication), and you are granted the necessary permissions (authorization). Many solutions have been proposed in the past, and the list keeps growing.

From simple to complex, here is my understanding of user identity management:

- WWW-Authenticate is the most basic method. You are asked for the username and password by the browser. As a result of the inability to control the login life cycle, it is seldom used today.
- A finer control over the login life cycle is session-cookie. The server maintains session storage, and the browser keeps the ID of the session. A cookie usually only works with browsers and is not mobile app friendly.
- To address the compatibility issue, the token can be used. The client sends the token to the server, and the server validates the token. The downside

is that the token needs to be encrypted and decrypted, which may be time-consuming.

- JWT is a standard way of representing tokens. This information can be verified and trusted because it is digitally signed. Since JWT contains the signature, there is no need to save session information on the server side.
- By using SSO (single sign-on), you can sign on only once and log in to multiple websites. It uses CAS (central authentication service) to maintain cross-site information.
- By using OAuth 2.0, you can authorize one website to access your information on another website.

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

Things NOT to do

- Storing passwords in plain text is not a good idea because anyone with internal access can see them.
- Storing password hashes directly is not sufficient because it is 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. 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.

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.

How does Google Authenticator (or other types of 2-factor authenticators) work?

Google Authenticator is commonly used for logging into our accounts when 2-factor authentication is enabled. How does it guarantee security?

Google Authenticator is a software-based authenticator that implements a two-step verification service. The diagram below provides detail.

There are two stages involved:

- Stage 1 - The user enables Google two-step verification.
- Stage 2 - The user uses the authenticator for logging in, etc.

Let's look at these stages.

Stage 1

Steps 1 and 2: Bob opens the web page to enable two-step verification. The front end requests a secret key. The authentication service generates the secret key for Bob and stores it in the database.

Step 3: The authentication service returns a URI to the front end. The URI is composed of a key issuer, username, and secret key. The URI is displayed in the form of a QR code on the web page.

Step 4: Bob then uses Google Authenticator to scan the generated QR code. The secret key is stored in the authenticator.

Stage 2 Steps 1 and 2: Bob wants to log into a website with Google two-step verification. For this, he needs the password. Every 30 seconds, Google Authenticator generates a 6-digit password using TOTP (Time-based One Time Password) algorithm. Bob uses the password to enter the website.

Steps 3 and 4: The frontend sends the password Bob enters to the backend for authentication. The authentication service reads the secret key from the database and generates a 6-digit password using the same TOTP algorithm as the client.

Step 5: The authentication service compares the two passwords generated by the client and the server, and returns the comparison result to the frontend. Bob can proceed with the login process only if the two passwords match.

Is this authentication mechanism safe?

- Can the secret key be obtained by others?

  We need to make sure the secret key is transmitted using HTTPS. The authenticator client and the database store the secret key, and we need to make sure the secret keys are encrypted.

- Can the 6-digit password be guessed by hackers?

  No. The password has 6 digits, so the generated password has 1 million potential combinations. Plus, the password changes every 30 seconds. If hackers want to guess the password in 30 seconds, they need to enter 30,000 combinations per second.

Real World Case Studies

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: Netflix uses GraphQL.

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

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

Messaging/streaming: Netflix employs Apache Kafka and 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, Atlas, and more for CI/CD processes.

Twitter Architecture 2022

Yes, this is the real Twitter architecture. It is posted by Elon Musk and redrawn by us for better readability.

Evolution of Airbnb's microservice architecture over the past 15 years

Airbnb's microservice architecture went through 3 main stages.

Monolith (2008 - 2017)

Airbnb began as a simple marketplace for hosts and guests. This is built in a Ruby on Rails application - the monolith.

What's the challenge?

- Confusing team ownership + unowned code
- Slow deployment

Microservices (2017 - 2020)

Microservice aims to solve those challenges. In the microservice architecture, key services include:

- Data fetching service
- Business logic data service
- Write workflow service
- UI aggregation service
- Each service had one owning team

What's the challenge?

Hundreds of services and dependencies were difficult for humans to manage.

Micro + macroservices (2020 - present)

This is what Airbnb is working on now. The micro and macroservice hybrid model focuses on the unification of APIs.

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 its own standard or adopt a shared standard by incorporating the 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 Nx, 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.

How will you design the Stack Overflow website?

If your answer is on-premise servers and monolith (on the bottom of the following image), 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 like the top portion of the picture.

- Microservice is used to decompose the system into small components.
- Each service has its own database. Use cache heavily.
- The service is sharded.
- The services talk to each other asynchronously through message queues.
- The service is implemented using Event Sourcing with CQRS.
- 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.

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.

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 architecture 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**”.

How does Disney Hotstar capture 5 Billion Emojis during a tournament?

1. Clients send emojis through standard HTTP requests. You can think of Golang Service as a typical Web Server. Golang is chosen because it supports concurrency well. Threads in Golang are lightweight.
2. Since the write volume is very high, Kafka (message queue) is used as a buffer.
3. Emoji data are aggregated by a streaming processing service called Spark. It aggregates data every 2 seconds, which is configurable. There is a trade-off to be made based on the interval. A shorter interval means emojis are delivered to other clients faster but it also means more computing resources are needed.
4. Aggregated data is written to another Kafka.
5. The PubSub consumers pull aggregated emoji data from Kafka.
6. Emojis are delivered to other clients in real-time through the PubSub infrastructure. The PubSub infrastructure is interesting. Hotstar considered the following protocols: Socketio, NATS, MQTT, and gRPC, and settled with MQTT.

A similar design is adopted by LinkedIn which streams a million likes/sec.

How Discord Stores Trillions Of Messages

The diagram below shows the evolution of message storage at Discord:

MongoDB Cassandra ScyllaDB

In 2015, the first version of Discord was built on top of a single MongoDB replica. Around Nov 2015, MongoDB stored 100 million messages and the RAM couldn’t hold the data and index any longer. The latency became unpredictable. Message storage needs to be moved to another database. Cassandra was chosen.

In 2017, Discord had 12 Cassandra nodes and stored billions of messages.

At the beginning of 2022, it had 177 nodes with trillions of messages. At this point, latency was unpredictable, and maintenance operations became too expensive to run.

There are several reasons for the issue:

- Cassandra uses the LSM tree for the internal data structure. The reads are more expensive than the writes. There can be many concurrent reads on a server with hundreds of users, resulting in hotspots.

- Maintaining clusters, such as compacting SSTables, impacts performance.
- Garbage collection pauses would cause significant latency spikes

ScyllaDB is Cassandra compatible database written in C++. Discord redesigned its architecture to have a monolithic API, a data service written in Rust, and ScyllaDB-based storage.

The p99 read latency in ScyllaDB is 15ms compared to 40-125ms in Cassandra. The p99 write latency is 5ms compared to 5-70ms in Cassandra.

How do video live streamings work on YouTube, TikTok live, or Twitch?

Live streaming differs from regular streaming because the video content is sent via the internet in real-time, usually with a latency of just a few seconds.

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

Step 1: The raw video data is captured by a microphone and camera. The data is sent to the server side.

Step 2: The video data is compressed and encoded. For example, the compressing algorithm separates the background and other video elements. After compression, the video is encoded to standards such as H.264. The size of the video data is much smaller after this step.

Step 3: The encoded data is divided into smaller segments, usually seconds in length, so it takes much less time to download or stream.

Step 4: The segmented data is sent to the streaming server. The streaming server needs to support different devices and network conditions. This is called 'Adaptive Bitrate Streaming.' This means we need to produce multiple files at different bitrates in steps 2 and 3.

Step 5: The live streaming data is pushed to edge servers supported by CDN (Content Delivery Network.) Millions of viewers can watch the video from an edge server nearby. CDN significantly lowers data transmission latency.

Step 6: The viewers' devices decode and decompress the video data and play the video in a video player.

Steps 7 and 8: If the video needs to be stored for replay, the encoded data is sent to a storage server, and viewers can request a replay from it later.

Standard protocols for live streaming include:

- RTMP (Real-Time Messaging Protocol): This was originally developed by Macromedia to transmit data between a Flash player and a server. Now it is used for streaming video data over the internet. Note that video conferencing applications like Skype use RTC (Real-Time Communication) protocol for lower latency.

- HLS (HTTP Live Streaming): It requires the H.264 or H.265 encoding. Apple devices accept only HLS format.
- DASH (Dynamic Adaptive Streaming over HTTP): DASH does not support Apple devices.
- Both HLS and DASH support adaptive bitrate streaming.

License